The Python Library Reference

リリース *3.8.19*

Guido van Rossum
and the Python development team

4 月 29, 2024

目次

第 1 章　はじめに 3
1.1 利用可能性について . . . 4

第 2 章　組み込み関数 5

第 3 章　組み込み定数 35
3.1 site モジュールで追加される定数 . . . 36

第 4 章　組み込み型 37
4.1 真理値判定 . . . 37
4.2 ブール演算 --- and, or, not . . . 38
4.3 比較 . . . 38
4.4 数値型 int, float, complex . . . 39
4.5 イテレータ型 . . . 46
4.6 シーケンス型 --- list, tuple, range . . . 47
4.7 テキストシーケンス型 --- str . . . 55
4.8 バイナリシーケンス型 --- bytes, bytearray, memoryview . . . 68
4.9 set（集合）型 --- set, frozenset . . . 94
4.10 マッピング型 --- dict . . . 97
4.11 コンテキストマネージャ型 . . . 103
4.12 その他の組み込み型 . . . 104
4.13 特殊属性 . . . 107
4.14 Integer string conversion length limitation . . . 108

第 5 章　組み込み例外 113
5.1 基底クラス . . . 114
5.2 具象例外 . . . 115
5.3 警告 . . . 122
5.4 例外のクラス階層 . . . 123

第 6 章　テキスト処理サービス 125
6.1 string --- 一般的な文字列操作 . . . 125
6.2 re --- 正規表現操作 . . . 139
6.3 difflib --- 差分の計算を助ける . . . 163

6.4 textwrap --- テキストの折り返しと詰め込み . . . 176
6.5 unicodedata --- Unicode データベース . . . 181
6.6 stringprep --- インターネットのための文字列調製 . . . 183
6.7 readline --- GNU readline のインタフェース . . . 185
6.8 rlcompleter --- GNU readline 向け補完関数 . . . 190

第 7 章 バイナリデータ処理 193
7.1 struct --- バイト列をパックされたバイナリデータとして解釈する . . . 193
7.2 codecs --- codec レジストリと基底クラス . . . 200

第 8 章 データ型 225
8.1 datetime --- 基本的な日付型および時間型 . . . 225
8.2 calendar --- 一般的なカレンダーに関する関数群 . . . 269
8.3 collections --- コンテナデータ型 . . . 275
8.4 collections.abc --- コレクションの抽象基底クラス . . . 295
8.5 heapq --- ヒープキューアルゴリズム . . . 301
8.6 bisect --- 配列二分法アルゴリズム . . . 306
8.7 array --- 効率のよい数値アレイ . . . 309
8.8 weakref --- 弱参照 . . . 312
8.9 types --- 動的な型生成と組み込み型に対する名前 . . . 321
8.10 copy --- 浅いコピーおよび深いコピー操作 . . . 328
8.11 pprint --- データ出力の整然化 . . . 329
8.12 reprlib --- もう一つの repr() の実装 . . . 336
8.13 enum --- 列挙型のサポート . . . 338

第 9 章 数値と数学モジュール 361
9.1 numbers --- 数の抽象基底クラス . . . 361
9.2 math --- 数学関数 . . . 365
9.3 cmath --- 複素数のための数学関数 . . . 373
9.4 decimal --- 十進固定及び浮動小数点数の算術演算 . . . 377
9.5 fractions --- 有理数 . . . 410
9.6 random --- 擬似乱数を生成する . . . 413
9.7 statistics --- 数理統計関数 . . . 421

第 10 章 関数型プログラミング用モジュール 435
10.1 itertools --- 効率的なループ実行のためのイテレータ生成関数 . . . 435
10.2 functools --- 高階関数と呼び出し可能オブジェクトの操作 . . . 452
10.3 operator --- 関数形式の標準演算子 . . . 462

第 11 章 ファイルとディレクトリへのアクセス 471
11.1 pathlib --- オブジェクト指向のファイルシステムパス . . . 471
11.2 os.path --- 共通のパス名操作 . . . 491
11.3 fileinput --- 複数の入力ストリームをまたいだ行の繰り返し処理をサポートする . . . 498
11.4 stat --- stat() の結果を解釈する . . . 501
11.5 filecmp --- ファイルおよびディレクトリの比較 . . . 507

11.6 tempfile --- 一時ファイルやディレクトリの作成 . . . 510
11.7 glob --- Unix 形式のパス名のパターン展開 . . . 515
11.8 fnmatch --- Unix ファイル名のパターンマッチ . . . 517
11.9 linecache --- テキストラインにランダムアクセスする . . . 518
11.10 shutil --- 高水準のファイル操作 . . . 519

第 12 章 データの永続化 535
12.1 pickle --- Python オブジェクトの直列化 . . . 535
12.2 copyreg --- pickle サポート関数を登録する . . . 556
12.3 shelve --- Python オブジェクトの永続化 . . . 557
12.4 marshal --- 内部使用向けの Python オブジェクト整列化 . . . 561
12.5 dbm --- Unix " データベース" へのインタフェース . . . 562
12.6 sqlite3 --- SQLite データベースに対する DB-API 2.0 インタフェース . . . 568

第 13 章 データ圧縮とアーカイブ 595
13.1 zlib --- gzip 互換の圧縮 . . . 595
13.2 gzip --- gzip ファイルのサポート . . . 600
13.3 bz2 --- bzip2 圧縮のサポート . . . 604
13.4 lzma --- LZMA アルゴリズムを使用した圧縮 . . . 609
13.5 zipfile --- ZIP アーカイブの処理 . . . 616
13.6 tarfile --- tar アーカイブファイルの読み書き . . . 628

第 14 章 ファイルフォーマット 649
14.1 csv --- CSV ファイルの読み書き . . . 649
14.2 configparser --- 設定ファイルのパーサー . . . 657
14.3 netrc --- netrc ファイルの処理 . . . 678
14.4 xdrlib --- XDR データのエンコードおよびデコード . . . 679
14.5 plistlib --- Mac OS X .plist ファイルの生成と解析 . . . 683

第 15 章 暗号関連のサービス 687
15.1 hashlib --- セキュアハッシュおよびメッセージダイジェスト . . . 687
15.2 hmac --- メッセージ認証のための鍵付きハッシュ化 . . . 700
15.3 secrets --- 機密を扱うために安全な乱数を生成する . . . 702

第 16 章 汎用オペレーティングシステムサービス 705
16.1 os --- 雑多なオペレーティングシステムインタフェース . . . 705
16.2 io --- ストリームを扱うコアツール . . . 773
16.3 time --- 時刻データへのアクセスと変換 . . . 790
16.4 argparse --- コマンドラインオプション、引数、サブコマンドのパーサー . . . 803
16.5 getopt --- C 言語スタイルのコマンドラインオプションパーサ . . . 840
16.6 logging --- Python 用ロギング機能 . . . 843
16.7 logging.config --- ロギングの環境設定 . . . 863
16.8 logging.handlers --- ロギングハンドラ . . . 876
16.9 getpass --- 可搬性のあるパスワード入力機構 . . . 893
16.10 curses --- 文字セル表示を扱うための端末操作 . . . 894

16.11 curses.textpad --- curses プログラムのためのテキスト入力ウィジェット 917
16.12 curses.ascii --- ASCII 文字に関するユーティリティ 919
16.13 curses.panel --- curses のためのパネルスタック拡張 922
16.14 platform --- 実行中プラットフォームの固有情報を参照する 923
16.15 errno --- 標準の errno システムシンボル 927
16.16 ctypes --- Python のための外部関数ライブラリ 935

第 17 章　並行実行 977
17.1 threading --- スレッドベースの並列処理 977
17.2 multiprocessing --- プロセスベースの並列処理 994
17.3 multiprocessing.shared_memory --- 異なるプロセスから参照可能な共有メモリ 1048
17.4 concurrent パッケージ 1054
17.5 concurrent.futures -- 並列タスク実行 1054
17.6 subprocess --- サブプロセス管理 1062
17.7 sched --- イベントスケジューラ 1085
17.8 queue --- 同期キュークラス 1087
17.9 contextvars --- コンテキスト変数 1092
17.10 _thread --- 低水準の スレッド API 1096
17.11 _dummy_thread --- _thread の代替モジュール 1099
17.12 dummy_threading --- threading の代替モジュール 1099

第 18 章　ネットワーク通信とプロセス間通信 1101
18.1 asyncio --- 非同期 I/O 1101
18.2 socket --- 低水準ネットワークインターフェース 1211
18.3 ssl --- ソケットオブジェクトに対する TLS/SSL ラッパー 1242
18.4 select --- I/O 処理の完了を待機する 1288
18.5 selectors --- 高水準の I/O 多重化 1297
18.6 asyncore --- 非同期ソケットハンドラ 1302
18.7 asynchat --- 非同期ソケットコマンド/レスポンスハンドラ 1307
18.8 signal --- 非同期イベントにハンドラを設定する 1310
18.9 mmap --- メモリマップファイル 1320

第 19 章　インターネット上のデータの操作 1327
19.1 email --- 電子メールと MIME 処理のためのパッケージ 1327
19.2 json --- JSON エンコーダおよびデコーダ 1402
19.3 mailcap --- mailcap ファイルの操作 1415
19.4 mailbox --- 様々な形式のメールボックス操作 1416
19.5 mimetypes --- ファイル名を MIME 型へマップする 1438
19.6 base64 --- Base16, Base32, Base64, Base85 データの符号化 1442
19.7 binhex --- binhex4 形式ファイルのエンコードおよびデコード 1446
19.8 binascii --- バイナリデータと ASCII データとの間での変換 1447
19.9 quopri --- MIME quoted-printable 形式データのエンコードおよびデコード 1450
19.10 uu --- uuencode 形式のエンコードとデコード 1451

第 20 章　構造化マークアップツール 1453

20.1 html --- HyperText Markup Language のサポート . . . 1453
20.2 html.parser--- HTML および XHTML のシンプルなパーサー . . . 1454
20.3 html.entities --- HTML 一般実体の定義 . . . 1459
20.4 XML を扱うモジュール群 . . . 1460
20.5 xml.etree.ElementTree --- ElementTree XML API . . . 1462
20.6 xml.dom --- 文書オブジェクトモデル (DOM) API . . . 1486
20.7 xml.dom.minidom --- 最小限の DOM の実装 . . . 1500
20.8 xml.dom.pulldom --- 部分的な DOM ツリー構築のサポート . . . 1506
20.9 xml.sax --- SAX2 パーサのサポート . . . 1508
20.10 xml.sax.handler --- SAX ハンドラの基底クラス . . . 1510
20.11 xml.sax.saxutils --- SAX ユーティリティ . . . 1517
20.12 xml.sax.xmlreader --- XML パーサのインタフェース . . . 1518
20.13 xml.parsers.expat --- Expat を使った高速な XML 解析 . . . 1523

第 21 章　インターネットプロトコルとサポート 1537

21.1 webbrowser --- 便利なウェブブラウザコントローラー . . . 1537
21.2 cgi --- CGI (ゲートウェイインタフェース規格) のサポート . . . 1540
21.3 cgitb --- CGI スクリプトのトレースバック管理機構 . . . 1549
21.4 wsgiref --- WSGI ユーティリティとリファレンス実装 . . . 1550
21.5 urllib --- URL を扱うモジュール群 . . . 1563
21.6 urllib.request --- URL を開くための拡張可能なライブラリ . . . 1563
21.7 urllib.response --- urllib で使用するレスポンスクラス . . . 1588
21.8 urllib.parse --- URL を解析して構成要素にする . . . 1588
21.9 urllib.error --- urllib.request が投げる例外 . . . 1599
21.10 urllib.robotparser --- robots.txt のためのパーザ . . . 1600
21.11 http --- HTTP モジュール群 . . . 1601
21.12 http.client --- HTTP プロトコルクライアント . . . 1604
21.13 ftplib --- FTP プロトコルクライアント . . . 1612
21.14 poplib --- POP3 プロトコルクライアント . . . 1619
21.15 imaplib --- IMAP4 プロトコルクライアント . . . 1623
21.16 nntplib --- NNTP プロトコルクライアント . . . 1631
21.17 smtplib --- SMTP プロトコルクライアント . . . 1639
21.18 smtpd --- SMTP サーバー . . . 1648
21.19 telnetlib --- Telnet クライアント . . . 1652
21.20 uuid --- **RFC 4122** に基づく UUID オブジェクト . . . 1656
21.21 socketserver --- ネットワークサーバのフレームワーク . . . 1660
21.22 http.server --- HTTP サーバ . . . 1671
21.23 http.cookies --- HTTP の状態管理 . . . 1678
21.24 http.cookiejar --- HTTP クライアント用の Cookie 処理 . . . 1683
21.25 xmlrpc --- XMLRPC サーバーとクライアントモジュール . . . 1694
21.26 xmlrpc.client --- XML-RPC クライアントアクセス . . . 1694
21.27 xmlrpc.server --- 基本的な XML-RPC サーバー . . . 1704
21.28 ipaddress --- IPv4/IPv6 操作ライブラリ . . . 1711

第 22 章　マルチメディアサービス 1727
22.1 audioop --- 生の音声データを操作する . . . 1727
22.2 aifc --- AIFF および AIFC ファイルの読み書き . . . 1731
22.3 sunau --- Sun AU ファイルの読み書き . . . 1734
22.4 wave --- WAV ファイルの読み書き . . . 1738
22.5 chunk --- IFF チャンクデータの読み込み . . . 1741
22.6 colorsys --- 色体系間の変換 . . . 1743
22.7 imghdr --- 画像の形式を決定する . . . 1744
22.8 sndhdr --- サウンドファイルの識別 . . . 1745
22.9 ossaudiodev --- OSS 互換オーディオデバイスへのアクセス . . . 1745

第 23 章　国際化 1753
23.1 gettext --- 多言語対応に関する国際化サービス . . . 1753
23.2 locale --- 国際化サービス . . . 1765

第 24 章　プログラムのフレームワーク 1775
24.1 turtle --- タートルグラフィックス . . . 1775
24.2 cmd --- 行指向のコマンドインタープリタのサポート . . . 1816
24.3 shlex --- 単純な字句解析 . . . 1822

第 25 章　Tk を用いたグラフィカルユーザインターフェイス 1831
25.1 tkinter --- Tcl/Tk の Python インタフェース . . . 1831
25.2 tkinter.ttk --- Tk のテーマ付きウィジェット . . . 1845
25.3 tkinter.tix --- Tk の拡張ウィジェット . . . 1868
25.4 tkinter.scrolledtext --- スクロールするテキストウィジェット . . . 1875
25.5 IDLE . . . 1875
25.6 他のグラフィカルユーザインタフェースパッケージ . . . 1890

第 26 章　開発ツール 1893
26.1 typing --- 型ヒントのサポート . . . 1893
26.2 pydoc --- ドキュメント生成とオンラインヘルプシステム . . . 1917
26.3 doctest --- 対話的な実行例をテストする . . . 1919
26.4 unittest --- ユニットテストフレームワーク . . . 1948
26.5 unittest.mock --- モックオブジェクトライブラリ . . . 1987
26.6 unittest.mock --- 入門 . . . 2034
26.7 2to3 - Python 2 から 3 への自動コード変換 . . . 2058
26.8 test --- Python 用回帰テストパッケージ . . . 2065
26.9 test.support --- テストのためのユーティリティ関数 . . . 2068
26.10 test.support.script_helper --- Utilities for the Python execution tests . . . 2085

第 27 章　デバッグとプロファイル 2087
27.1 Audit events table . . . 2087
27.2 bdb --- デバッガーフレームワーク . . . 2092
27.3 faulthandler --- Python traceback のダンプ . . . 2098
27.4 pdb --- Python デバッガ . . . 2101

27.5 Python プロファイラ . . . 2109
27.6 timeit --- 小さなコード断片の実行時間計測 . . . 2120
27.7 trace --- Python 文実行のトレースと追跡 . . . 2126
27.8 tracemalloc --- メモリ割り当ての追跡 . . . 2129

第 28 章 ソフトウェア・パッケージと配布 2143
28.1 distutils --- Python モジュールの構築とインストール . . . 2143
28.2 ensurepip --- pip インストーラのブートストラップ . . . 2144
28.3 venv --- 仮想環境の作成 . . . 2146
28.4 zipapp --- 実行可能な Python zip 書庫を管理する . . . 2156

第 29 章 Python ランタイムサービス 2165
29.1 sys --- システムパラメータと関数 . . . 2165
29.2 sysconfig --- Python の構成情報にアクセスする . . . 2189
29.3 builtins --- 組み込みオブジェクト . . . 2194
29.4 __main__ --- トップレベルのスクリプト環境 . . . 2194
29.5 warnings --- 警告の制御 . . . 2195
29.6 dataclasses --- データクラス . . . 2203
29.7 contextlib --- with 文コンテキスト用ユーティリティ . . . 2213
29.8 abc --- 抽象基底クラス . . . 2228
29.9 atexit --- 終了ハンドラ . . . 2233
29.10 traceback --- スタックトレースの表示または取得 . . . 2235
29.11 __future__ --- future 文の定義 . . . 2243
29.12 gc --- ガベージコレクタインターフェース . . . 2244
29.13 inspect --- 活動中のオブジェクトの情報を取得する . . . 2249
29.14 site --- サイト固有の設定フック . . . 2268

第 30 章 カスタム Python インタプリタ 2273
30.1 code --- インタプリタ基底クラス . . . 2273
30.2 codeop --- Python コードをコンパイルする . . . 2276

第 31 章 モジュールのインポート 2279
31.1 zipimport --- Zip アーカイブからモジュールを import する . . . 2279
31.2 pkgutil --- パッケージ拡張ユーティリティ . . . 2282
31.3 modulefinder --- スクリプト中で使われているモジュールを検索する . . . 2285
31.4 runpy --- Python モジュールの位置特定と実行 . . . 2287
31.5 importlib --- import の実装 . . . 2290
31.6 importlib.metadata を使う . . . 2316

第 32 章 Python 言語サービス 2321
32.1 parser --- Python 解析木にアクセスする . . . 2321
32.2 ast --- 抽象構文木 . . . 2326
32.3 symtable --- コンパイラの記号表へのアクセス . . . 2334
32.4 symbol --- Python 解析木と共に使われる定数 . . . 2337
32.5 token --- Python 解析木と共に使われる定数 . . . 2338

32.6 keyword --- Python キーワードチェック . . . 2342
32.7 tokenize --- Python ソースのためのトークナイザ . . . 2342
32.8 tabnanny --- あいまいなインデントの検出 . . . 2347
32.9 pyclbr --- Python モジュールブラウザサポート . . . 2348
32.10 py_compile --- Python ソースファイルのコンパイル . . . 2350
32.11 compileall --- Python ライブラリをバイトコンパイル . . . 2353
32.12 dis --- Python バイトコードの逆アセンブラ . . . 2357
32.13 pickletools --- pickle 開発者のためのツール群 . . . 2375

第 33 章 各種サービス 2379
33.1 formatter --- 汎用の出力書式化機構 . . . 2379

第 34 章 MS Windows 固有のサービス 2385
34.1 msilib --- Microsoft インストーラーファイルの読み書き . . . 2385
34.2 msvcrt --- MS VC++ 実行時システムの有用なルーチン群 . . . 2392
34.3 winreg --- Windows レジストリへのアクセス . . . 2395
34.4 winsound --- Windows 用の音声再生インタフェース . . . 2406

第 35 章 Unix 固有のサービス 2409
35.1 posix --- 最も一般的な POSIX システムコール群 . . . 2409
35.2 pwd --- パスワードデータベースへのアクセスを提供する . . . 2410
35.3 spwd --- シャドウパスワードデータベース . . . 2412
35.4 grp --- グループデータベースへのアクセス . . . 2413
35.5 crypt --- Unix パスワードをチェックするための関数 . . . 2414
35.6 termios --- POSIX スタイルの端末制御 . . . 2416
35.7 tty --- 端末制御のための関数群 . . . 2418
35.8 pty --- 擬似端末ユーティリティ . . . 2418
35.9 fcntl --- fcntl および ioctl システムコール . . . 2420
35.10 pipes --- シェルパイプラインへのインタフェース . . . 2423
35.11 resource --- リソース使用状態の情報 . . . 2424
35.12 nis --- Sun の NIS (Yellow Pages) へのインタフェース . . . 2430
35.13 syslog --- Unix syslog ライブラリルーチン群 . . . 2431

第 36 章 取って代わられたモジュール群 2433
36.1 optparse --- コマンドラインオプション解析器 . . . 2433
36.2 imp --- import 内部へのアクセス . . . 2467

第 37 章 ドキュメント化されていないモジュール 2475
37.1 プラットフォーム固有のモジュール . . . 2475

付録 A 章 用語集 2477

付録 B 章 このドキュメントについて 2495
B.1 Python ドキュメント 貢献者 . . . 2495

付録 C 章 歴史とライセンス 2497

C.1 Python の歴史 2497
C.2 Terms and conditions for accessing or otherwise using Python 2498
C.3 Licenses and Acknowledgements for Incorporated Software 2502

付録 D 章 Copyright 2517

参考文献 2519

参考文献 2519

Python モジュール索引 2521

Python モジュール索引 2521

索引 2525

索引 2525

reference-index ではプログラミング言語 Python の厳密な構文とセマンティクスについて説明されていますが、このライブラリリファレンスマニュアルでは Python とともに配付されている標準ライブラリについて説明します。また Python 配布物に収められていることの多いオプションのコンポーネントについても説明します。

Python の標準ライブラリはとても拡張性があり、下の長い目次のリストで判るように幅広いものを用意しています。このライブラリには、例えばファイル I/O のように、Python プログラマが直接アクセスできないシステム機能へのアクセス機能を提供する (C で書かれた) 組み込みモジュールや、日々のプログラミングで生じる多くの問題に標準的な解決策を提供する Python で書かれたモジュールが入っています。これら数多くのモジュールには、プラットフォーム固有の事情をプラットフォーム独立な API へと昇華させることにより、Python プログラムに移植性を持たせ、それを高めるという明確な意図があります。

Windows 向けの Python インストーラはたいてい標準ライブラリのすべてを含み、しばしばそれ以外の追加のコンポーネントも含んでいます。Unix 系のオペレーティングシステムの場合は Python は一揃いのパッケージとして提供されるのが普通で、オプションのコンポーネントを手に入れるにはオペレーティングシステムのパッケージツールを使うことになるでしょう。

標準ライブラリに加えて、数千のコンポーネントが (独立したプログラムやモジュールからパッケージ、アプリケーション開発フレームワークまで) 成長し続けるコレクションとして Python Package Index から入手可能です。

第

ONE

はじめに

この ”Python ライブラリ” には様々な内容が収録されています。

このライブラリには、数値型やリスト型のような、通常は言語の ” 核” をなす部分とみなされるデータ型が含まれています。Python 言語のコア部分では、これらの型に対してリテラル表現形式を与え、意味づけ上のいくつかの制約を与えていますが、完全にその意味づけを定義しているわけではありません。(一方で、言語のコア部分では演算子のスペルや優先順位のような構文法的な属性を定義しています。)

このライブラリにはまた、組み込み関数と例外が納められています --- 組み込み関数および例外は、全ての Python で書かれたコード上で、`import` 文を使わずに使うことができるオブジェクトです。これらの組み込み要素のうちいくつかは言語のコア部分で定義されていますが、大半は言語コアの意味づけ上不可欠なものではないのでここでしか記述されていません。

とはいえ、このライブラリの大部分に収録されているのはモジュールのコレクションです。このコレクションを細分化する方法はいろいろあります。あるモジュールは C 言語で書かれ、Python インタプリタに組み込まれています; 一方別のモジュールは Python で書かれ、ソースコードの形式で取り込まれます。またあるモジュールは、例えば実行スタックの追跡結果を出力するといった、Python に非常に特化したインタフェースを提供し、一方他のモジュールでは、特定のハードウェアにアクセスするといった、特定のオペレーティングシステムに特化したインタフェースを提供し、さらに別のモジュールでは WWW (ワールドワイドウェブ) のような特定のアプリケーション分野に特化したインタフェースを提供しています。モジュールによっては全てのバージョン、全ての移植版の Python で利用することができたり、背後にあるシステムがサポートしている場合にのみ使えたり、Python をコンパイルしてインストールする際に特定の設定オプションを選んだときにのみ利用できたりします。

このマニュアルは ” 内部から外部へ” と構成されています。つまり、最初に組み込みの関数を記述し、組み込みのデータ型、例外、そして最後に各モジュールと続きます。モジュールは関係のあるものでグループ化して一つの章にしています。

つまり、このマニュアルを最初から読み始め、読み飽きたところで次の章に進めば、Python ライブラリで利用できるモジュールやサポートしているアプリケーション領域の概要をそこそこ理解できるということです。もちろん、このマニュアルを小説のように読む必要は **ありません** --- (マニュアルの先頭部分にある) 目次にざっと目を通したり、(最後尾にある) 索引でお目当ての関数やモジュール、用語を探すことだってできます。もしランダムな項目について勉強してみたいのなら、ランダムにページを選び (*random* 参照)、そこから 1, 2 節読むこともできます。このマニュアルの各節をどんな順番で読むかに関わらず、**組み込み関数** の章から始めるとよいでしょう。マニュアルの他の部分は、この節の内容について知っているものとして書かれているからです。

それでは、ショーの始まりです！

1.1 利用可能性について

- 「利用できる環境：Unix」の意味はこの関数が Unix システムにあることが多いということです。このことは特定の OS における存在を主張するものではありません。
- 特に記述がない場合、「利用できる環境：Unix」と書かれている関数は、Unix をコアにしている Mac OS X でも利用することができます。

第

TWO

組み込み関数

Python インタプリタには数多くの関数と型が組み込まれており、いつでも利用できます。それらをここにアルファベット順に挙げます。

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

`abs`(*x*)

数の絶対値を返します。引数は整数または浮動小数点数です。引数が複素数なら、その絶対値 (magnitude) が返されます。*x* に `__abs__()` が定義されている場合、`abs(x)` は `x.__abs__()` を返します。

`all`(*iterable*)

iterable の全ての要素が真ならば (もしくは *iterable* が空ならば) `True` を返します。以下のコードと等価です:

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

any(*iterable*)

iterable のいずれかの要素が真ならば True を返します。*iterable* が空なら False を返します。以下のコードと等価です:

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

ascii(*object*)

repr() と同様、オブジェクトの印字可能な表現を含む文字列を返しますが、*repr()* によって返された文字列中の非 ASCII 文字は \x 、\u 、\U エスケープを使ってエスケープされます。これは Python 2 の *repr()* によって返されるのと同じ文字列を作ります。

bin(*x*)

整数を先頭に "0b" が付いた 2 進文字列に変換します。結果は Python の式としても使える形式になります。

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

先頭に "0b" が付いて欲しい、もしくは付いて欲しくない場合には、次の方法のどちらでも使えます。

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

より詳しいことは *format()* も参照してください。

class bool(*[x]*)

ブール値、即ち True または False のどちらかを返します。*x* は標準の **真理値判定手続き** を用いて変換されます。*x* が偽または省略されている場合、この関数は False を返します。それ以外の場合、True を返します。*bool* クラスは *int* クラスの派生クラスです (**数値型** *int, float, complex* を参照してください)。このクラスからさらに派生することはできません。ブール値のインスタンスは False と True のみです (**ブール値** を参照してください)。

バージョン 3.7 で変更: *x* は位置専用引数になりました。

breakpoint(**args, **kws*)

この関数により、呼び出された箇所からデバッガへ移行します。特に、この関数は args および kws をそのまま *sys.breakpointhook()* に渡して呼び出します。デフォルトでは、sys.breakpointhook() は引数無しで *pdb.set_trace()* を呼び出します。このケースでは、*pdb.set_trace()* は単なる便利な関数なので、明示的に *pdb* をインポートしたり、デバッガに入るためにキーをたくさん打ち込む必要はありません。ただし、*sys.breakpointhook()* は他の関数を設定することもでき、*breakpoint()* は自動的にその関数を呼び出します。これにより、最適なデバッガに移行できます。

引数 breakpointhook 付きで **監査イベント** builtins.breakpoint を送出します。

バージョン 3.7 で追加.

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

新しいバイト配列を返します。*bytearray* クラスは 0 <= x < 256 の範囲の整数からなる変更可能な配列です。**ミュータブルなシーケンス型** に記述されている変更可能な配列に対する普通のメソッドの大半を備えています。また、*bytes* 型が持つメソッドの大半も備えています（see *bytes* と *bytearray* **の操作**）。

オプションの *source* 引数は、配列を異なる方法で初期化するのに使われます:

- **文字列** の場合、*encoding* (と、オプションの *errors*) 引数も与えなければなりません。このとき *bytearray()* は文字列を *str.encode()* でバイトに変換して返します。
- **整数** の場合、配列はそのサイズになり、null バイトで初期化されます。
- バッファインタフェース に適合するオブジェクトの場合、そのオブジェクトの読み出し専用バッファがバイト配列の初期化に使われます。
- **イテラブル** の場合、範囲 0 <= x < 256 内の整数のイテラブルでなければならず、それらが配列の初期の内容として使われます。

引数がなければ、長さ 0 の配列が生成されます。

バイナリシーケンス型 --- *bytes, bytearray, memoryview* と *bytearray* **オブジェクト** も参照してください。

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

範囲 0 <= x < 256 の整数のイミュータブルなシーケンスである ”bytes” オブジェクトを返します。*bytes* は *bytearray* のイミュータブル版です。オブジェクトを変化させないようなメソッドや、インデクシングやスライシングのふるまいは、これと同様のものです。

従って、コンストラクタ引数は *bytearray()* のものと同様に解釈されます。

バイト列オブジェクトはリテラルでも生成できます。strings を参照してください。

バイナリシーケンス型 --- *bytes, bytearray, memoryview*, **バイトオブジェクト**, *bytes* と *bytearray* **の操作** も参照してください。

callable(*object*)

object 引数が呼び出し可能オブジェクトであれば *True* を、そうでなければ *False* を返します。この関数が True を返しても、呼び出しは失敗する可能性がありますが、False であれば、*object* の呼び出しは決して成功しません。なお、クラスは呼び出し可能 (クラスを呼び出すと新しいインスタンスを返します) です。また、インスタンスはクラスが __call__() メソッドを持つなら呼び出し可能です。

バージョン 3.2 で追加: この関数は Python 3.0 で一度取り除かれましたが、Python 3.2 で復活しました。

chr(*i*)

Unicode コードポイントが整数 *i* である文字を表す文字列を返します。例えば chr(97) は文字列 'a'

を、chr(8364) は文字列 '€' を返します。*ord()* の逆です。

引数の有効な範囲は 0 から 1,114,111 (16 進数で 0x10FFFF) です。*i* が範囲外の場合 *ValueError* が送出されます。

@classmethod

メソッドをクラスメソッドへ変換します。

クラスメソッドは、インスタンスメソッドが暗黙の第一引数としてインスタンスをとるように、第一引数としてクラスをとります。クラスメソッドを宣言するには、以下のイディオムを使います:

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

@classmethod 形式は関数 **デコレータ** です。詳しくは function を参照してください。

クラスメソッドは、(C.f() のように) クラスから呼び出すことも、(C().f() のように) インスタンスから呼び出すこともできます。インスタンスはそのクラスが何であるかを除いて無視されます。クラスメソッドが派生クラスから呼び出される場合は、その派生クラスオブジェクトが暗黙の第一引数として渡されます。

クラスメソッドは C++ や Java の静的メソッドとは異なります。それが欲しければ、*staticmethod()* を参照してください。

クラスメソッドについて詳しい情報は types を参照してください。

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

source をコードオブジェクト、もしくは、AST オブジェクトにコンパイルします。コードオブジェクトは *exec()* 文で実行したり、*eval()* 呼び出しで評価できます。*source* は通常の文字列、バイト列、AST オブジェクトのいずれでもかまいません。AST オブジェクトへの、また、AST オブジェクトからのコンパイルの方法は、*ast* モジュールのドキュメントを参照してください。

filename 引数には、コードの読み出し元のファイルを与えなければなりません; ファイルから読み出されるのでなければ、認識可能な値を渡して下さい ('<string>' が一般的に使われます)。

mode 引数は、コンパイルされるコードの種類を指定します; *source* が一連の文から成るなら 'exec'、単一の式から成るなら 'eval'、単一の対話的文の場合 'single' です。(後者の場合、評価が None 以外である式文が印字されます)。

オプション引数 *flags* および *dont_inherit* は、*source* のコンパイルにどの future statements を作用させるかを制御します。どちらも与えられていない (または両方ともゼロ) ならば、*compile()* を呼び出している側のコードで有効な future 文とともにコードをコンパイルします。*flags* 引数が与えられていて *dont_inherit* は与えられていない (またはゼロ) 場合は、それに加えて *flags* に指定された future 文が使われます。*dont_inherit* がゼロでない整数の場合、*flags* 引数がそのまま使われ、compile を呼び出している側で有効な future 文は無視されます。

future 文はビットフィールドで指定されます。ビットフィールドはビット単位の OR を取ることで複数の文を指定することができます。特定の機能を指定するために必要なビットフィールドは、*__future__* モジュールの _Feature インスタンスにおける compiler_flag 属性で得られます。

オプション引数 *flags* は、コンパイルされたコードでトップレベルの `await`, `async for`, `async with` 構文が許されるかどうかも制御します。`ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` ビットがセットされた場合、返されるオブジェクトは `co_code` に `CO_COROUTINE` セットを持ち、`await eval(code_object)` を通して対話的に実行できます。

引数 *optimize* は、コンパイラの最適化レベルを指定します; デフォルトの値 `-1` は、インタプリタの `-O` オプションで与えられるのと同じ最適化レベルを選びます。明示的なレベルは、`0` (最適化なし、`__debug__` は真)、`1` (assert は取り除かれ、`__debug__` は偽)、`2` (docstring も取り除かれる) です。

この関数は、コンパイルされたソースが不正である場合 *SyntaxError* を、ソースがヌルバイトを含む場合 *ValueError* を送出します。

Python コードをパースしてその AST 表現を得たいのであれば、*ast.parse()* を参照してください。

引数 `source`, `filename` を指定して **監査イベント** `compile` を送出します。

注釈: 複数行に渡るコードの文字列を `'single'` や `'eval'` モードでコンパイルするとき、入力は一つ以上の改行文字で終端されなければなりません。これは、*code* モジュールで不完全な文と完全な文を検知しやすくするためです。

> **警告:** AST オブジェクトにコンパイルしているときに、十分に大きい文字列や複雑な文字列によって Python の抽象構文木コンパイラのスタックが深さの限界を越えることで、Python インタプリタをクラッシュさせられます。

バージョン 3.2 で変更: Windows や Mac の改行も受け付けます。また `'exec'` モードでの入力が改行で終わっている必要もありません。*optimize* 引数が追加されました。

バージョン 3.5 で変更: 以前は *source* にヌルバイトがあったときに *TypeError* を送出していました。

バージョン 3.8 で追加: `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` フラグを渡してトップレベルの `await`, `async for`, および `async with` のサポートを有効化することができるようになりました。

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

値 *real* + *imag**1j の複素数を返すか、文字列や数を複素数に変換します。第一引数が文字列なら、それが複素数と解釈され、この関数は第二引数無しで呼び出されなければなりません。第二引数は文字列であってはなりません。それぞれの引数は (複素数を含む) 任意の数値型です。*imag* が省略された場合、標準の値はゼロで、このコンストラクタは *int* や *float* のような数値変換としてはたらきます。両方の引数が省略された場合、`0j` を返します。

一般的な Python オブジェクト `x` に対して、`complex(x)` は `x.__complex__()` に処理を委譲します。`__complex__()` が定義されていない場合は `__float__()` にフォールバックします。`__float__()` も定義されていない場合は `__index__()` にフォールバックします。

注釈: 文字列から変換するとき、その文字列は中央の `+` や `-` 演算子の周りに空白を含んではなりませ

ん。例えば、complex('1+2j') はいいですが、complex('1 + 2j') は *ValueError* を送出します。

複素数型については **数値型** *int, float, complex* に説明があります。

バージョン 3.6 で変更: コードリテラル中で桁をグループ化するのにアンダースコアを利用できます。

バージョン 3.8 で変更: __complex__() と __float__() が定義されていない場合、__index__() へフォールバックします。

delattr(*object*, *name*)

setattr() の親戚です。引数はオブジェクトと文字列です。文字列はオブジェクトの属性のうち一つの名前でなければなりません。この関数は、オブジェクトが許すなら、指名された属性を削除します。例えば、delattr(x, 'foobar') は del x.foobar と等価です。

class dict(***kwarg*)

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

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

新しい辞書を作成します。*dict* オブジェクトは辞書クラスです。このクラスに関するドキュメンテーションは *dict* と **マッピング型** *--- dict* を参照してください。

他のコンテナについては、ビルトインの *list*, *set*, *tuple* クラスおよび *collections* モジュールを参照してください。

dir(*[object]*)

引数がない場合、現在のローカルスコープにある名前のリストを返します。引数がある場合、そのオブジェクトの有効な属性のリストを返そうと試みます。

オブジェクトが __dir__() という名のメソッドを持つなら、そのメソッドが呼び出され、属性のリストを返さなければなりません。これにより、カスタムの __getattr__() や __getattribute__() 関数を実装するオブジェクトは、*dir()* が属性を報告するやり方をカスタマイズできます。

オブジェクトが __dir__() を提供していない場合、定義されていればオブジェクトの *__dict__* 属性から、そして型オブジェクトから、情報を収集しようと試みます。結果のリストは完全であるとは限らず、また、カスタムの __getattr__() を持つ場合、不正確になるかもしれません。

デフォルトの *dir()* メカニズムは、完全というより最重要な情報を作成しようとするため、異なる型のオブジェクトでは異なって振る舞います:

- オブジェクトがモジュールオブジェクトの場合、リストにはモジュールの属性の名前が含まれます。
- オブジェクトが型オブジェクトやクラスオブジェクトの場合、リストにはその属性の名前と、再帰的にたどったその基底クラスの属性が含まれます。
- それ以外の場合には、リストにはオブジェクトの属性名、クラス属性名、再帰的にたどった基底クラスの属性名が含まれます。

返されるリストはアルファベット順に並べられています。例えば:

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

注釈: *dir()* は主に対話プロンプトでの使用に便利なように提供されているので、厳密性や一貫性を重視して定義された名前のセットというよりも、むしろ興味を引くような名前のセットを返そうとします。また、この関数の細かい動作はリリース間で変わる可能性があります。例えば、引数がクラスであるとき、メタクラス属性は結果のリストに含まれません。

---

divmod(*a*, *b*)

2 つの (複素数でない) 数を引数として取り、整数の除法を行ったときの商と剰余からなる対を返します。混合した被演算子型では、二項算術演算子での規則が適用されます。整数では、結果は `(a // b, a % b)` と同じです。浮動小数点数では、結果は `(q, a % b)` で、ここで *q* は通常 `math.floor(a / b)` ですが、それより 1 小さくなることもあります。いずれにせよ、`q * b + a % b` は *a* に非常に近く、`a % b` がゼロでなければその符号は *b* と同じで、`0 <= abs(a % b) < abs(b)` です。

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

enumerate オブジェクトを返します。*iterable* は、シーケンスか *iterator* か、あるいはイテレーションをサポートするその他のオブジェクトでなければなりません。*enumerate()* によって返されたイテレータの *__next__()* メソッドは、(デフォルトでは 0 となる *start* からの) カウントと、*iterable* 上のイテレーションによって得られた値を含むタプルを返します。

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

次と等価です:

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

eval(*expression*[, *globals*[, *locals*]])

文字列とオプションの引数 *globals*、*locals* をとります。*globals* を与える場合は辞書でなくてはなりません。*locals* を与える場合は任意のマッピングオブジェクトにできます。

expression 引数は Python 式 (技術的な言い方では、条件のリスト) として構文解析され評価されます。このとき辞書 *globals* および *locals* はそれぞれグローバルおよびローカルな名前空間として使われます。*globals* 辞書が与えられ、__builtins__ をキーとする値が含まれていない場合、*expression* が構文解析される前に、組み込みモジュール *builtins* の辞書への参照がキー __builtins__ の値として挿入されます。よって、*expression* は通常、標準の *builtins* モジュールへの完全なアクセスを有し、制限された環境は伝播します。*locals* 辞書が省略された場合、デフォルトは *globals* 辞書です。辞書が両方とも省略された場合、表現式は *eval()* が呼び出されている環境の *globals* 辞書と *locals* 辞書の下で実行されます。*eval()* は、それが実行される環境の **ネストされたスコープ** (非ローカルのオブジェクト) を参照できないことに注意してください。

返される値は、式が評価された結果になります。構文エラーは例外として報告されます。例:

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

この関数は (*compile()* で生成されるような) 任意のコードオブジェクトを実行するのにも利用できます。この場合、文字列の代わりにコードオブジェクトを渡します。このコードオブジェクトが、引数 *mode* を 'exec' としてコンパイルされている場合、*eval()* が返す値は None になります。

ヒント: *exec()* 関数により文の動的な実行がサポートされています。*globals()* および *locals()* 関数は、それぞれ現在のグローバルおよびローカルな辞書を返すので、それらを *eval()* や *exec()* に渡して使うことができます。

リテラルだけを含む式の文字列を安全に評価できる関数、*ast.literal_eval()* も参照してください。

引数 code_object を指定して **監査イベント** exec を送出します。

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

この関数は Python コードの動的な実行をサポートします。*object* は文字列かコードオブジェクトでなければなりません。文字列なら、その文字列は一連の Python 文として解析され、そして (構文エラーが生じない限り) 実行されます。[*1] コードオブジェクトなら、それは単純に実行されます。どの場合でも、実行されるコードはファイル入力として有効であることが期待されます (リファレンスマニュアルの節 "file-input" を参照)。なお、nonlocal, yield および return 文は、*exec()* 関数に渡されたコードの文脈中においてさえ、関数定義の外では使えません。返り値は None です。

いずれの場合でも、オプションの部分が省略されると、コードは現在のスコープ内で実行されます。*globals* だけが与えられたなら、辞書でなくてはならず (辞書のサブクラスではない)、グローバル変数とローカル変数の両方に使われます。*globals* と *locals* が与えられたなら、それぞれグローバル変数とローカル変数として使われます。*locals* を指定する場合は何らかのマップ型オブジェクトでなければなりません。モジュールレベルでは、グローバルとローカルは同じ辞書です。exec が *globals* と *locals* として別のオブジェクトを取った場合、コードはクラス定義に埋め込まれたかのように実行されます。

[*1] なお、パーサは Unix スタイルの行末の記法しか受け付けません。コードをファイルから読んでいるなら、必ず、改行変換モードで Windows や Mac スタイルの改行を変換してください。

globals 辞書がキー `__builtins__` に対する値を含まなければ、そのキーに対して、組み込みモジュール *builtins* の辞書への参照が挿入されます。ですから、実行されるコードを *exec()* に渡す前に、*globals* に自作の `__builtins__` 辞書を挿入することで、コードがどの組み込みを利用できるか制御できます。

引数 `code_object` を指定して **監査イベント** `exec` を送出します。

注釈: 組み込み関数 *globals()* および *locals()* は、それぞれ現在のグローバルおよびローカルの辞書を返すので、それらを *exec()* の第二、第三引数にそのまま渡して使うと便利なことがあります。

注釈: 標準では *locals* は後に述べる関数 *locals()* のように動作します: 標準の *locals* 辞書に対する変更を試みてはいけません。*exec()* の呼び出しが返る時にコードが *locals* に与える影響を知りたいなら、明示的に *locals* 辞書を渡してください。

`filter`(*function*, *iterable*)

iterable の要素のうち *function* が真を返すものでイテレータを構築します。*iterable* はシーケンスか、反復をサポートするコンテナか、イテレータです。*function* が `None` なら、恒等関数を仮定します。すなわち、*iterable* の偽である要素がすべて除去されます。

なお、`filter(function, iterable)` は、関数が `None` でなければジェネレータ式 `(item for item in iterable if function(item))` と同等で、関数が `None` なら `(item for item in iterable if item)` と同等です。

function が偽を返すような *iterable* の各要素を返す補完的関数は、*itertools.filterfalse()* を参照してください。

`class float`($[x]$)

数または文字列 x から生成された浮動小数点数を返します。

引数が文字列の場合、10 進数を含んだ文字列にしてください。先頭に符号が付いていたり、空白中に埋め込まれていてもかまいません。符号として `'+'` か `'-'` を追加できます。`'+'` は、作られる値に何の影響も与えません。引数は NaN (not-a-number) や正負の無限大を表す文字列でもかまいません。正確には、入力は、前後の空白を取り除いた後に以下の文法に従う必要があります:

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

ここで `floatnumber` は floating で記述されている Python の浮動小数点数リテラルです。大文字か小文字かは関係なく、例えば "inf"、"Inf"、"INFINITY" 、"iNfINity" は全て正の無限大として使え

る綴りです。

一方で、引数が整数または浮動小数点数なら、(Python の浮動小数点数の精度で) 同じ値の浮動小数点数が返されます。引数が Python の浮動小数点数の範囲外なら、*OverflowError* が送出されます。

一般の Python オブジェクト `x` に対して、`float(x)` は `x.__float__()` に委譲します。`__float__()` が定義されていない場合、`__index__()` へフォールバックします。

引数が与えられなければ、`0.0` が返されます。

例:

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

浮動小数点数型については、**数値型** *int, float, complex* も参照してください。

バージョン 3.6 で変更: コードリテラル中で桁をグループ化するのにアンダースコアを利用できます。

バージョン 3.7 で変更: *x* は位置専用引数になりました。

バージョン 3.8 で変更: `__float__()` が定義されていない場合、`__index__()` へフォールバックします。

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

value を *format_spec* で制御される ” 書式化された” 表現に変換します。*format_spec* の解釈は *value* 引数の型に依存しますが、ほとんどの組み込み型で使われる標準的な構文が存在します: **書式指定ミニ言語仕様** 。

デフォルトの *format_spec* は空の文字列です。それは通常 *str(value)* の呼び出しと同じ結果になります。

`format(value, format_spec)` の呼び出しは、`type(value).__format__(value, format_spec)` に翻訳され、これは value の `__format__()` メソッドの検索をするとき、インスタンス辞書を回避します。このメソッドの探索が *object* に到達しても *format_spec* が空にならなかったり、*format_spec* や返り値が文字列でなかったりした場合、*TypeError* が送出されます。

バージョン 3.4 で変更: *format_spec* が空の文字列でない場合 `object().__format__(format_spec)` は *TypeError* を送出します。

class frozenset([*iterable*])

新しい *frozenset* オブジェクトを返します。オプションで *iterable* から得られた要素を含みます。`frozenset` はビルトインクラスです。このクラスに関するドキュメントは *frozenset* と *set* (**集合) 型** *--- set, frozenset* を参照してください。

他のコンテナについては、ビルトインクラス *set*, *list*, *tuple*, *dict* や *collections* モジュールを見てください。

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

object の指名された属性の値を返します。*name* は文字列でなくてはなりません。文字列がオブジェクトの属性の一つの名前であった場合、戻り値はその属性の値になります。例えば、`getattr(x, 'foobar')` は `x.foobar` と等価です。指名された属性が存在しない場合、*default* が与えられていればそれが返され、そうでない場合には *AttributeError* が送出されます。

globals()

現在のグローバルシンボルテーブルを表す辞書を返します。これは常に現在のモジュール (関数やメソッドの中では、それを呼び出したモジュールではなく、それを定義しているモジュール) の辞書です。

hasattr(*object*, *name*)

引数はオブジェクトと文字列です。文字列がオブジェクトの属性名の一つであった場合 `True` を、そうでない場合 `False` を返します。(この関数は、`getattr(object, name)` を呼び出して *AttributeError* を送出するかどうかを見ることで実装されています。)

hash(*object*)

オブジェクトのハッシュ値を (存在すれば) 返します。ハッシュ値は整数です。これらは辞書を検索する際に辞書のキーを高速に比較するために使われます。等しい値となる数値は等しいハッシュ値を持ちます (1 と 1.0 のように型が異なっていてもです)。

注釈: 独自の `__hash__()` メソッドを実装したオブジェクトを使う場合、*hash()* が実行するマシンのビット幅に合わせて戻り値を切り捨てることに注意してください。詳しくは `__hash__()` を参照してください。

---

help([*object*])

組み込みヘルプシステムを起動します。(この関数は対話的な使用のためのものです。) 引数が与えられていない場合、対話的ヘルプシステムはインタプリタコンソール上で起動します。引数が文字列の場合、文字列はモジュール、関数、クラス、メソッド、キーワード、またはドキュメントの項目名として検索され、ヘルプページがコンソール上に印字されます。引数がその他のオブジェクトの場合、そのオブジェクトに関するヘルプページが生成されます。

help() を呼び出したときに関数の引数リストにスラッシュ (/) が現れた場合は、スラッシュより前の引数が位置専用引数だという意味であることに注意してください。より詳しいことは、位置専用引数についての FAQ の記事 を参照してください。

この関数は、*site* モジュールから、組み込みの名前空間に移されました。

バージョン 3.4 で変更: *pydoc* と *inspect* への変更により、呼び出し可能オブジェクトの報告されたシグニチャがより包括的で一貫性のあるものになりました。

hex(*x*)

Convert an integer number to a lowercase hexadecimal string prefixed with "0x". If *x* is not a Python *int* object, it has to define an `__index__()` method that returns an integer. Some

examples:

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

整数を大文字の 16 進文字列や小文字の 16 進文字列、先頭の "0x" 付きや "0x" 無しに変換したい場合は、次に挙げる方法が使えます:

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

より詳しいことは *format()* も参照してください。

16 を底として 16 進数文字列を整数に変換するには *int()* も参照してください。

注釈: 浮動小数点数の 16 進文字列表記を得たい場合には、*float.hex()* メソッドを使って下さい。

---

id(*object*)

オブジェクトの "識別値" を返します。この値は整数で、このオブジェクトの有効期間中は一意かつ定数であることが保証されています。有効期間が重ならない 2 つのオブジェクトは同じ *id()* 値を持つかもしれません。

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

引数 `id` を指定して **監査イベント** `builtins.id` を送出します。

input([*prompt*])

引数 *prompt* が存在すれば、それが末尾の改行を除いて標準出力に書き出されます。次に、この関数は入力から 1 行を読み込み、文字列に変換して (末尾の改行を除いて) 返します。EOF が読み込まれたとき、*EOFError* が送出されます。例:

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

readline モジュールが読み込まれていれば、*input()* はそれを使って精緻な行編集やヒストリ機能を提供します。

引数 `prompt` 付きで **監査イベント** `builtins.input` を送出します。

引数 `result` 付きで **監査イベント** `builtins.input/result` を送出します。

class int([*x*])

class int(*x*, *base=10*)

数値または文字列 *x* から作成された整数オブジェクトを返します。引数が与えられない場合には `0` を返します。*x* に `__int__()` が定義されている場合は、`int(x)` は `x.__int__()` を返します。*x* に `__index__()` が定義されている場合は、`x.__index__()` を返します。*x* に `__trunc__()` が定義されている場合は、`x.__trunc__()` を返します。浮動小数点数については、これは 0 に近い側へ切り捨てます。

x が数値でない、あるいは *base* が与えられた場合、*x* は文字列、*bytes* インスタンス、*bytearray* インスタンスのいずれかで、基数 *base* の 整数リテラル で表されたものでなければなりません。オプションで、リテラルの前に `+` あるいは `-` を (中間のスペースなしで) 付けることができます。また、リテラルは余白で囲むことができます。基数 n のリテラルは、0 から n-1 の数字に値 10-35 を持つ `a` から `z` (または `A` から `Z`) を加えたもので構成されます。デフォルトの *base* は 10 です。許される値は 0 と 2--36 です。基数 2, 8, 16 のリテラルは、別の記法としてコード中の整数リテラルのように `0b`/`0B`, `0o`/`0O`, `0x`/`0X` を前に付けることができます。基数 0 はコードリテラルとして厳密に解釈することを意味します。その結果、実際の基数は 2, 8, 10, 16 のどれかになります。したがって `int('010', 0)` は有効ではありませんが、`int('010')` や `int('010', 8)` は有効です。

整数型については、**数値型** *int, float, complex* も参照してください。

バージョン 3.4 で変更: *base* が *int* のインスタンスでなく、*base* オブジェクトが `base.__index__` メソッドを持っている場合、そのメソッドを呼んで底に対する整数を得ることができます。以前のバージョンでは `base.__index__` ではなく `base.__int__` を使用していました。

バージョン 3.6 で変更: コードリテラル中で桁をグループ化するのにアンダースコアを利用できます。

バージョン 3.7 で変更: *x* は位置専用引数になりました。

バージョン 3.8 で変更: `__int__()` が定義されていない場合、`__index__()` へフォールバックします。

バージョン 3.8.14 で変更: *int* string inputs and string representations can be limited to help avoid denial of service attacks. A *ValueError* is raised when the limit is exceeded while converting a string *x* to an *int* or when converting an *int* into a string would exceed the limit. See the *integer string conversion length limitation* documentation.

isinstance(*object*, *classinfo*)

object 引数が *classinfo* 引数のインスタンスであるか、(直接、間接、または **仮想**) サブクラスのインスタンスの場合に `True` を返します。*object* が与えられた型のオブジェクトでない場合、この関数は常に `False` を返します。*classinfo* が型オブジェクトのタプル (あるいは再帰的に複数のタプル) の場合、*object* がそれらのいずれかのインスタンスであれば `True` を返します。*classinfo* が型や型からなるタプル、あるいは複数のタプルのいずれでもない場合、*TypeError* 例外が送出されます。

issubclass(*class*, *classinfo*)

class が *classinfo* の (直接または間接的な、あるいは *virtual*) サブクラスである場合に `True` を返します。クラスはそれ自身のサブクラスとみなされます。*classinfo* はクラスオブジェクトからなるタプルでもよく、この場合には *classinfo* のすべてのエントリが調べられます。その他の場合では、例外 *TypeError* が送出されます。

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

イテレータ オブジェクトを返します。第二引数があるかどうかで、第一引数の解釈は大きく異なります。第二引数がない場合、*object* は反復プロトコル (`__iter__()` メソッド) か、シーケンスプロトコル (引数が 0 から開始する `__getitem__()` メソッド) をサポートする集合オブジェクトでなければなりません。これらのプロトコルが両方ともサポートされていない場合、*TypeError* が送出されます。第二引数 *sentinel* が与えられているなら、*object* は呼び出し可能オブジェクトでなければなりません。この場合に生成されるイテレータは、*__next__()* を呼ぶ毎に *object* を引数無しで呼び出します。返された値が *sentinel* と等しければ、*StopIteration* が送出され、そうでなければ、戻り値がそのまま返されます。

イテレータ型 も見てください。

2 引数形式の *iter()* の便利な利用方法の 1 つは、ブロックリーダーの構築です。例えば、バイナリのデータベースファイルから固定幅のブロックをファイルの終端に到達するまで読み出すには次のようにします:

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

len(*s*)

オブジェクトの長さ (要素の数) を返します。引数はシーケンス (文字列、バイト列、タプル、リスト、range 等) かコレクション (辞書、集合、凍結集合等) です。

class list([*iterable*])

list は、実際には関数ではなくミュータブルなシーケンス型で、リスト型 *(list)* と シーケンス型 --- *list, tuple, range* にドキュメント化されています。

locals()

現在のローカルシンボルテーブルを表す辞書を更新して返します。関数ブロックで *locals()* を呼び出したときは自由変数が返されますが、クラスブロックでは返されません。モジュールレベルでは、*locals()* と *globals()* は同じ辞書であることに注意してください。

注釈: この辞書の内容は変更してはいけません; 変更しても、インタプリタが使うローカル変数や自由変数の値には影響しません。

---

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

function を、結果を返しながら *iterable* の全ての要素に適用するイテレータを返します。追加の *iterable* 引数が渡されたなら、*function* はその数だけの引数を取らなければならず、全てのイテラブルから並行して取られた要素に適用されます。複数のイテラブルが与えられたら、このイテレータはその中の最短のイテラブルが尽きた時点で止まります。関数の入力がすでに引数タプルに配置されている場合は、*itertools.starmap()* を参照してください。

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

iterable の中で最大の要素、または 2 つ以上の引数の中で最大のものを返します。

位置引数が 1 つだけ与えられた場合、それはは空でない *iterable* でなくてはいけません。その iterable の最大の要素が返されます。2 つ以上のキーワード無しの位置引数が与えられた場合、その位置引数の中で最大のものが返されます。

任意のキーワード専用引数が 2 つあります。*key* 引数は引数を 1 つ取る順序関数 (*list.sort()* のもののように) を指定します。*default* 引数は与えられたイテラブルが空の場合に返すオブジェクトを指定します。イテラブルが空で *default* が与えられていない場合 *ValueError* が送出されます。

最大の要素が複数あるとき、この関数はそのうち最初に現れたものを返します。これは、`sorted(iterable, key=keyfunc, reverse=True)[0]` や `heapq.nlargest(1, iterable, key=keyfunc)` のような、他のソート安定性を維持するツールと両立します。

バージョン 3.4 で追加: *default* キーワード専用引数。

バージョン 3.8 で変更: *key* 引数が `None` であることを許容します。

class memoryview(*obj*)

与えられたオブジェクトから作られた ”メモリビュー” オブジェクトを返します。詳しくは **メモリビュー** を参照してください。

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

iterable の中で最小の要素、または 2 つ以上の引数の中で最小のものを返します。

位置引数が 1 つだけ与えられた場合、それはは空でない *iterable* でなくてはいけません。その iterable の最小の要素が返されます。2 つ以上のキーワード無しの位置引数が与えられた場合、その位置引数の中で最小のものが返されます。

任意のキーワード専用引数が 2 つあります。*key* 引数は引数を 1 つ取る順序関数 (*list.sort()* のもののように) を指定します。*default* 引数は与えられたイテラブルが空の場合に返すオブジェクトを指定します。イテラブルが空で *default* が与えられていない場合 *ValueError* が送出されます。

最小の要素が複数あるとき、この関数はそのうち最初に現れたものを返します。これは、`sorted(iterable, key=keyfunc)[0]` や `heapq.nsmallest(1, iterable, key=keyfunc)` のような、他のソート安定性を維持するツールと両立します。

バージョン 3.4 で追加: *default* キーワード専用引数。

バージョン 3.8 で変更: *key* 引数が `None` であることを許容します。

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

iterator の *__next__()* メソッドを呼び出すことにより、次の要素を取得します。イテレータが尽きている場合、*default* が与えられていればそれが返され、そうでなければ *StopIteration* が送出されます。

class object

特徴を持たない新しいオブジェクトを返します。*object* は全てのクラスの基底クラスです。これは、Python のクラスの全てのインスタンスに共通のメソッド群を持ちます。この関数はいかなる引数も受け付けません。

注釈: *object* は *__dict__* を **持たない** ので、*object* クラスのインスタンスに任意の属性を代入することはできません。

`oct`(*x*)

整数を先頭に "0o" が付いた 8 進文字列に変換します。結果は Python の式としても使える形式になります。

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

整数を接頭辞の "0o" 付きや "0o" 無しの 8 進文字列に変換したい場合は、次に挙げる方法が使えます。

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

より詳しいことは *format()* も参照してください。

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

file を開き、対応する **ファイルオブジェクト** を返します。ファイルを開くことができなければ、*OSError* が送出されます。この関数の利用例について、tut-files を参照してください。

file は *path-like object* で、開くファイルの (絶対または現在のワーキングディレクトリに対する相対) パス名を与えるものか、または、ラップするファイルの整数のファイルディスクリプタです。(ファイルディスクリプタが与えられた場合は、それは *closefd* が `False` に設定されていない限り、返された I/O オブジェクトが閉じられるときに閉じられます。)

mode はオプションの文字列で、ファイルが開かれるモードを指定します。デフォルトは `'r'` で、読み込み用にテキストモードで開くという意味です。その他のよく使われる値は、書き込み (ファイルがすでに存在する場合はそのファイルを切り詰めます) 用の `'w'`、排他的な生成用の `'x'`、追記用の `'a'` です (**いくつかの** Unix システムでは、**全て** の書き込みが現在のファイルシーク位置に関係なくファイルの末尾に追加されます)。テキストモードでは、*encoding* が指定されていない場合に使われるエンコーディングはプラットフォームに依存します:`locale.getpreferredencoding(False)` を使って現在のロケールエンコーディングを取得します。(raw バイト列の読み書きには、バイナリモードを使い、*encoding* は未指定のままとします) 指定可能なモードは次の表の通りです。

文字	意味
'r'	読み込み用に開く (デフォルト)
'w'	書き込み用に開き、まずファイルを切り詰める
'x'	排他的な生成に開き、ファイルが存在する場合は失敗する
'a'	書き込み用に開き、ファイルが存在する場合は末尾に追記する
'b'	バイナリモード
't'	テキストモード (デフォルト)
'+'	更新用に開く (読み込み・書き込み用)

デフォルトのモードは 'r' (開いてテキストの読み込み、'rt' と同義) です。モード 'w+' と 'w+b' はファイルを開いて切り詰めます。モード 'r+' と 'r+b' はファイルを切り詰めずに開きます。

概要 で触れられているように、Python はバイナリとテキストの I/O を区別します。(*mode* 引数に 'b' を含めて) バイナリモードで開かれたファイルは、内容をいかなるデコーディングもせずに *bytes* オブジェクトとして返します。(デフォルトや、*mode* 引数に 't' が含まれたときの) テキストモードでは、ファイルの内容は *str* として返され、バイト列はまず、プラットフォーム依存のエンコーディングか、*encoding* が指定された場合は指定されたエンコーディングを使ってデコードされます。

さらに 'U' という許可されているモード文字がありますが、これの効果は無くなっていて非推奨とされています。以前はこのモード文字は、テキストモードでの *universal newlines* を有効にしていましたが、Python 3.0 ではそれがデフォルトの挙動となりました。より詳細なことは *newline* 引数のドキュメントを参照してください。

注釈: Python は、下層のオペレーティングシステムがテキストファイルをどう認識するかには依存しません; すべての処理は Python 自身で行われ、よってプラットフォーム非依存です。

---

buffering はオプションの整数で、バッファリングの方針を設定するのに使われます。バッファリングを無効にする (バイナリモードでのみ有効) には 0、行単位バッファリング (テキストモードでのみ有効) には 1、固定値のチャンクバッファの大きさをバイト単位で指定するには 1 以上の整数を渡してください。*buffering* 引数が与えられていないとき、デフォルトのバッファリング方針は以下のように動作します:

- バイナリファイルは固定サイズのチャンクでバッファリングされます。バッファサイズは、下層のデバイスの「ブロックサイズ」を決定するヒューリスティックを用いて選択され、それが不可能な場合は代わりに *io.DEFAULT_BUFFER_SIZE* が使われます。多くのシステムでは、典型的なバッファサイズは 4096 か 8192 バイト長になるでしょう。
- 「対話的な」テキストファイル (*isatty()* が True を返すファイル) は行バッファリングを使用します。その他のテキストファイルは、上で説明したバイナリファイル用の方針を使用します。

encoding はファイルのエンコードやデコードに使われる *text encoding* の名前です。このオプションはテキストモードでのみ使用してください。デフォルトエンコーディングはプラットフォーム依存 (*locale.getpreferredencoding()* が返すもの) ですが、Python でサポートされているエンコーディングはどれでも使えます。詳しくは *codecs* モジュール内のサポートしているエンコーディングの

リストを参照してください。

errors はオプションの文字列で、エンコードやデコードでのエラーをどのように扱うかを指定するものです。バイナリモードでは使用できません。様々な標準のエラーハンドラが使用可能です (**エラーハンドラ** に列記されています) が、*codecs.register_error()* に登録されているエラー処理の名前も使用可能です。標準のエラーハンドラの名前には、以下のようなものがあります:

- `'strict'` はエンコーディングエラーがあると例外 *ValueError* を発生させます。デフォルト値である `None` も同じ効果です。
- `'ignore'` はエラーを無視します。エンコーディングエラーを無視することで、データが失われる可能性があることに注意してください。
- `'replace'` は、不正な形式のデータが存在した場所に (`'?'` のような) 置換マーカーを挿入します。
- `'surrogateescape'` は正しくないバイト列を、Unicode の Private Use Area (私用領域) にある U+DC80 から U+DCFF のコードポイントで示します。データを書き込む際に `surrogateescape` エラーハンドラが使われると、これらの私用コードポイントは元と同じバイト列に変換されます。これはエンコーディングが不明なファイルを処理するのに便利です。
- `'xmlcharrefreplace'` はファイルへの書き込み時のみサポートされます。そのエンコーディングでサポートされない文字は、`&#nnn;` 形式の適切な XML 文字参照で置換されます。
- `'backslashreplace'` は不正なデータを Python のバックスラッシュ付きのエスケープシーケンスで置換します。
- `'namereplace'` (書き込み時のみサポートされています) はサポートされていない文字を `\N{...}` エスケープシーケンスで置換します。

newline は *universal newlines* モードの動作を制御します (テキストモードでのみ動作します)。`None`, `''`, `'\n'`, `'\r'`, `'\r\n'` のいずれかです。これは以下のように動作します:

- ストリームからの入力の読み込み時、*newline* が `None` の場合、ユニバーサル改行モードが有効になります。入力中の行は `'\n'`, `'\r'`, または `'\r\n'` で終わり、呼び出し元に返される前に `'\n'` に変換されます。`''` の場合、ユニバーサル改行モードは有効になりますが、行末は変換されずに呼び出し元に返されます。その他の正当な値の場合、入力行は与えられた文字列でのみ終わり、行末は変換されずに呼び出し元に返されます。
- ストリームへの出力の書き込み時、*newline* が `None` の場合、全ての `'\n'` 文字はシステムのデフォルトの行セパレータ *os.linesep* に変換されます。*newline* が `''` または `'\n'` の場合は変換されません。*newline* がその他の正当な値の場合、全ての `'\n'` 文字は与えられた文字列に変換されます。

closefd が `False` で、ファイル名ではなくてファイル記述子が与えられた場合、下層のファイル記述子はファイルが閉じられた後も開いたままとなります。ファイル名が与えられた場合、*closefd* は `True` (デフォルト値) でなければなりません。そうでない場合エラーが送出されます。

呼び出し可能オブジェクトを *opener* として与えることで、カスタムのオープナーが使えます。そしてファイルオブジェクトの下層のファイル記述子は、*opener* を (*file*, *flags*) で呼び出して得られます。

opener は開いたファイル記述子を返さなければなりません。(*os.open* を *opener* として渡すと、None を渡したのと同様の機能になります)。

新たに作成されたファイルは **継承不可** です。

次の例は *os.open()* 関数の *dir_fd* 引数を使い、与えられたディレクトリからの相対パスで指定されたファイルを開きます:

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

open() 関数が返す *file object* の型はモードに依存します。*open()* をファイルをテキストモード ('w', 'r', 'wt', 'rt', など) で開くのに使ったときは *io.TextIOBase* (特に *io.TextIOWrapper*) のサブクラスを返します。ファイルをバッファリング付きのバイナリモードで開くのに使ったときは *io.BufferedIOBase* のサブクラスを返します。実際のクラスは様々です。読み込みバイナリモードでは *io.BufferedReader* を返します。書き込みバイナリモードや追記バイナリモードでは *io.BufferedWriter* を返します。読み書きモードでは *io.BufferedRandom* を返します。バッファリングが無効なときは raw ストリーム、すなわち *io.RawIOBase* のサブクラスである *io.FileIO* を返します。

fileinput 、(*open()* が宣言された場所である) *io* 、*os* 、*os.path* 、*tempfile* 、*shutil* などの、ファイル操作モジュールも参照してください。

引数 file, mode, flags を指定して **監査イベント** open を送出します。

mode と flags の 2 つの引数は呼び出し時の値から修正されたり、推量により設定されたりする可能性があります。

バージョン 3.3 で変更:

- *opener* 引数を追加しました。
- 'x' モードを追加しました。
- 以前は *IOError* が送出されていました、それは現在 *OSError* のエイリアスになりました。
- 既存のファイルを 排他的生成モード ('x') で開いた場合、*FileExistsError* を送出するようになりました。

バージョン 3.4 で変更:

- ファイルが継承不可になりました。

Deprecated since version 3.4, will be removed in version 3.9: 'U' モード。

バージョン 3.5 で変更:

- システムコールが中断されシグナルハンドラが例外を送出しなかった場合、この関数は *InterruptedError* 例外を送出する代わりにシステムコールを再試行するようになりました (論拠については **PEP 475** を参照してください)。
- 'namereplace' エラーハンドラが追加されました。

バージョン 3.6 で変更:

- *os.PathLike* を実装したオブジェクトを受け入れるようになりました。
- Windows では、コンソールバッファのオープンは、*io.FileIO* ではなく、*io.RawIOBase* のサブクラスを返すでしょう。

ord(*c*)

1 文字の Unicode 文字を表す文字列に対し、その文字の Unicode コードポイントを表す整数を返します。例えば、ord('a') は整数 97 を返し、ord('€') (ユーロ記号) は 8364 を返します。これは *chr()* の逆です。

pow(*base*, *exp*[, *mod*])

base の *exp* 乗を返します; *mod* があれば、*base* の *exp* 乗に対する *mod* の剰余を返します (pow(base, exp) % mod より効率よく計算されます)。二引数の形式 pow(base, exp) は、冪乗演算子を使った base**exp と等価です。

引数は数値型でなくてはなりません。型混合の場合、二項算術演算における型強制規則が適用されます。*int* 被演算子に対しては、第二引数が負でない限り、結果は (型強制後の) 被演算子と同じ型になります; 負の場合、全ての引数は浮動小数点に変換され、浮動小数点の結果が返されます。例えば、10**2 は 100 を返しますが、10**-2 は 0.01 を返します。

base と *exp* が *int* オペランドで *mod* が存在するとき、*mod* もまた整数型でなければならず、かつゼロであってはいけません。*mod* が存在して *exp* が負の整数の場合、*base* は *mod* と互いに素 (最大公約数が 1) でなければなりません。この場合、*inv_base* を *base* に対する *mod* を法とするモジュラ逆数 (*base* と *inv_base* の積を *mod* で割った余りが 1 になるような数) として、pow(inv_base, -exp, mod) が返されます。

以下は “97“ を法とする 38 のモジュラ逆数の計算例です:

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

バージョン 3.8 で変更: *int* オペランドに対して、三引数形式の pow で第二引数に負の値を取ることができるようになりました。これによりモジュラ逆数の計算が可能になります。

バージョン 3.8 で変更: キーワード引数を取ることができるようになりました。以前は位置引数だけがサポートされていました。

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

objects を *sep* で区切りながらテキストストリーム *file* に表示し、最後に *end* を表示します。*sep* 、*end* 、*file* 、*flush* を与える場合、キーワード引数として与える必要があります。

キーワードなしの引数はすべて、*str()* がするように文字列に変換され、*sep* で区切られながらストリームに書き出され、最後に *end* が続きます。*sep* と *end* の両方とも、文字列でなければなりません。これらを `None` にすると、デフォルトの値が使われます。*objects* が与えられなければ、*print()* は *end* だけを書き出します。

file 引数は、`write(string)` メソッドを持つオブジェクトでなければなりません。指定されないか、`None` である場合、*sys.stdout* が使われます。表示される引数は全てテキスト文字列に変換されますから、*print()* はバイナリモードファイルオブジェクトには使用できません。代わりに `file.write(...)` を使ってください。

出力がバッファ化されるかどうかは通常 *file* で決まりますが、*flush* キーワード引数が真ならストリームは強制的にフラッシュされます。

バージョン 3.3 で変更: キーワード引数 *flush* が追加されました。

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

property 属性を返します。

fget は属性値を取得するための関数です。*fset* は属性値を設定するための関数です。*fdel* は属性値を削除するための関数です。*doc* は属性の docstring を作成します。

典型的な使用法は、属性 `x` の処理の定義です:

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

    def getx(self):
        return self._x

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

    def delx(self):
        del self._x

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

c が *C* のインスタンスならば、`c.x` は getter を呼び出し、`c.x = value` は setter を、`del c.x` は deleter を呼び出します。

doc は、与えられれば property 属性のドキュメント文字列になります。与えられなければ、property は *fget* のドキュメント文字列 (もしあれば) をコピーします。そのため *property()* を **デコレータ** として使えば、読み出し専用 property を作るのは容易です:

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

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

`@property` デコレータは `voltage()` を同じ名前のまま 読み出し専用属性の "getter" にし、*voltage* のドキュメント文字列を "Get the current voltage." に設定します。

property オブジェクトは `getter`, `setter`, `deleter` メソッドを持っています。これらのメソッドをデコレータとして使うと、対応するアクセサ関数がデコレートされた関数に設定された、property のコピーを作成できます。これを一番分かりやすく説明する例があります:

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

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

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

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

このコードは最初の例と等価です。追加の関数には、必ず元の property と同じ名前 (この例では `x`) を与えて下さい。

返される property オブジェクトも、コンストラクタの引数に対応した `fget`, `fset`, および `fdel` 属性を持ちます。

バージョン 3.5 で変更: 属性オブジェクトのドックストリングが書き込み可能になりました。

class range(*stop*)

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

range は、実際には関数ではなくイミュータブルなシーケンス型で、*range* と **シーケンス型** --- *list*, *tuple*, *range* にドキュメント化されています。

repr(*object*)

オブジェクトの印字可能な表現を含む文字列を返します。この関数は多くの型について、*eval()* に渡されたときと同じ値を持つようなオブジェクトを表す文字列を生成しようとします。そうでない場合は、山括弧に囲まれたオブジェクトの型の名前と追加の情報 (大抵の場合はオブジェクトの名前とアド

レスを含みます) を返します。クラスは、__repr__() メソッドを定義することで、この関数によりそのクラスのインスタンスが返すものを制御することができます。

reversed(*seq*)

要素を逆順に取り出すイテレータ (reverse *iterator*) を返します。*seq* は __reversed__() メソッドを持つか、シーケンス型プロトコル (__len__() メソッド、および、0 以上の整数を引数とする __getitem__() メソッド) をサポートするオブジェクトでなければなりません。

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

number の小数部を *ndigists* 桁に丸めた値を返します。*ndigits* が省略されたり、None だった場合、入力値に最も近い整数を返します。

round() をサポートする組み込み型では、値は 10 のマイナス *ndigits* 乗の倍数の中で最も近いものに丸められます; 二つの倍数が同じだけ近いなら、偶数を選ぶ方に (そのため、例えば round(0.5) と round(-0.5) は両方とも 0 に、round(1.5) は 2 に) 丸められます。*ndigits* には任意の整数値が有効となります (正の整数、ゼロ、負の整数)。返り値は *ndigits* が指定されていないか None の場合は整数、そうでなければ返り値は *number* と同じ型です。

一般的な Python オブジェクト number に対して、round は処理を *number.__round__* に移譲します。

注釈: 浮動小数点数に対する *round()* の振る舞いは意外なものかもしれません: 例えば、round(2.675, 2) は予想通りの 2.68 ではなく 2.67 を与えます。これはバグではありません: これはほとんどの小数が浮動小数点数で正確に表せないことの結果です。詳しくは tut-fp-issues を参照してください。

---

class set([*iterable*])

オプションで *iterable* の要素を持つ、新しい *set* オブジェクトを返します。set は組み込みクラスです。このクラスについて詳しい情報は *set* や *set* (集合) 型 --- *set, frozenset* を参照してください。

他のコンテナについては *collections* モジュールや組み込みの *frozenset* 、*list* 、*tuple* 、*dict* クラスを参照してください。

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

getattr() の相方です。引数はオブジェクト、文字列、それから任意の値です。文字列は既存の属性または新たな属性の名前にできます。この関数は指定したオブジェクトが許せば、値を属性に関連付けます。例えば、setattr(x, 'foobar', 123) は x.foobar = 123 と等価です。

class slice(*stop*)

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

range(start, stop, step) で指定されるインデクスの集合を表す、**スライス** オブジェクトを返します。引数 *start* および *step* はデフォルトでは None です。スライスオブジェクトは読み出し専用の属性 start、stop および step を持ち、これらは単に引数で使われた 値 (またはデフォルト値) を返します。これらの値には、その他のはっきりと した機能はありません。しかしながら、これらの値は Numerical Python および、その他のサードパーティによる拡張で利用されています。スライスオブジェクトは拡張されたインデクス指定構文が使われる際にも生成されます。例えば a[start:stop:step] や a[start:stop, i] です。この関数の代替となるイテレータを返す関数、*itertools.islice()* も参

照してください。

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

iterable の要素を並べ替えた新たなリストを返します。

2 つのオプション引数があり、これらはキーワード引数として指定されなければなりません。

key には 1 引数関数を指定します。これは *iterable* の各要素から比較キーを展開するのに使われます (例えば、`key=str.lower` のように指定します)。デフォルト値は `None` です (要素を直接比較します)。

reverse は真偽値です。`True` がセットされた場合、リストの要素は個々の比較が反転したものとして並び替えられます。

旧式の *cmp* 関数を *key* 関数に変換するには *functools.cmp_to_key()* を使用してください。

組み込みの *sorted()* 関数は安定なことが保証されています。同等な要素の相対順序を変更しないことが保証されていれば、ソートは安定です。これは複数のパスでソートを行なうのに役立ちます（例えば部署でソートしてから給与の等級でソートする場合)。

ソートの例と簡単なチュートリアルは sortinghowto を参照して下さい。

@staticmethod

メソッドを静的メソッドへ変換します。

静的メソッドは暗黙の第一引数を受け取りません。静的メソッドを宣言するには、このイディオムを使ってください:

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

`@staticmethod` 形式は関数 **デコレータ** です。詳しくは function を参照してください。

静的メソッドは (`C.f()` のよう) クラスから呼び出したり、(`C().f()` のように) インスタンスから呼び出したりできます。

Python における静的メソッドは Java や C++ における静的メソッドと類似しています。クラスコンストラクタの代替を生成するのに役立つ変種、*classmethod()* も参照してください。

あらゆるデコレータと同じく、`staticmethod` は普通の関数のように呼べ、その返り値で処理が行えます。この機能は、クラス本体から関数を参照する必要があり、かつ、インスタンスメソッドに自動変換されるのを避けたいケースで必要になります。そのようなケースでは、このイディオムが使えます:

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

静的メソッドについて詳しい情報は types を参照してください。

class str(*object=''*)

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

object の *str* 版を返します。詳細は *str()* を参照してください。

str は組み込みの文字列 クラス です。文字列に関する一般的な情報は、テキストシーケンス型 --- *str* を参照してください。

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

start と *iterable* の要素を左から右へ合計し、総和を返します。*iterable* の要素は通常は数値で、start の値は文字列であってはなりません。

使う場面によっては、*sum()* よりもいい選択肢があります。文字列からなるシーケンスを結合する高速かつ望ましい方法は `''.join(sequence)` を呼ぶことです。浮動小数点数値を拡張された精度で加算するには、*math.fsum()* を参照してください。一連のイテラブルを連結するには、*itertools.chain()* の使用を考えてください。

バージョン 3.8 で変更: *start* パラメータをキーワード引数として指定することができるようになりました。

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

メソッドの呼び出しを *type* の親または兄弟クラスに委譲するプロキシオブジェクトを返します。これはクラスの中でオーバーライドされた継承メソッドにアクセスするのに便利です。

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

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

object-or-type の *__mro__* 属性は、*getattr()* と *super()* の 両方で使われる、メソッド解決の探索順序を列記します。この属性は動的で、継承の階層構造が更新されれば、随時変化します。

第 2 引数が省かれたなら、返されるスーパーオブジェクトは束縛されません。第 2 引数がオブジェクトであれば、`isinstance(obj, type)` は真でなければなりません。第 2 引数が型であれば、`issubclass(type2, type)` は真でなければなりません (これはクラスメソッドに役に立つでしょう)。

super の典型的な用途は 2 つあります。第一に、単継承のクラス階層構造で *super* は名前を明示することなく親クラスを参照するのに使え、それゆえコードをメンテナンスしやすくなります。この用途は他のプログラミング言語で見られる *super* の用途によく似ています。

2 つ目の用途は動的な実行環境において協調的 (cooperative) な多重継承をサポートすることです。これは Python に特有の用途で、静的にコンパイルされる言語や、単継承のみをサポートする言語には見られないものです。この機能により、同じ名前のメソッドを実装する複数の基底クラスを使った ” ダイヤモンド型* の継承構造を実装することができます。良い設計は、そのような実装において、どのような場合でも同じ呼び出しシグネチャを持つように強制します。(理由は呼び出しの順序が実行時に決定されること、呼び出し順序はクラス階層構造の変化に順応すること、そして呼び出し順序が実行時まで未知の兄弟クラスが含まれる場合があることです)。

両方の用途において、典型的なスーパークラスの呼び出しは次のようになります:

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

メソッドのルックアップに加えて、*super()* は属性のルックアップに対しても同様に動作します。考えうる用途のひとつは親クラスや兄弟クラスの *descriptors* (デスクリプタ) を呼び出すことです。

なお、*super()* は `super().__getitem__(name)` のような明示的なドット表記属性探索の束縛処理の一部として実装されています。これは、`__getattribute__()` メソッドを予測可能な順序でクラスを検索するように実装し、協調的な多重継承をサポートすることで実現されています。従って、*super()* は文や `super()[name]` のような演算子を使った暗黙の探索向けには定義されていません。

また、*super()* の使用は引数無しの形式を除きメソッド内部に限定されないことにも注目して下さい。2 引数の形式は、必要な要素を正確に指定するので、適当な参照を作ることができます。クラス定義中における引数無しの形式は、定義されているクラスを取り出すのに必要な詳細を、通常の方法で現在のインスタンスにアクセスするようにコンパイラが埋めるのではたらきます。

super() を用いて協調的なクラスを設計する方法の実践的な提案は、guide to using super() を参照してください。

class tuple([*iterable*])

tuple は、実際は関数ではなくイミュータブルなシーケンス型で、**タプル型** *(tuple)* と **シーケンス型** *--- list, tuple, range* にドキュメント化されています。

class type(*object*)

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

引数が 1 つだけの場合、*object* の型を返します。返り値は型オブジェクトで、一般に *object.__class__* によって返されるのと同じオブジェクトです。

オブジェクトの型の判定には、*isinstance()* 組み込み関数を使うことが推奨されます。これはサブクラスを考慮するからです。

引数が 3 つの場合、新しい型オブジェクトを返します。これは本質的には `class` 文の動的な書式です。*name* 文字列はクラス名で、*__name__* 属性になります。*bases* 基底クラスのタプルで、*__bases__* 属性になります; 空の場合は全てのクラスの基底クラスである *object* が追加されます。*dict* は、クラス本体の属性とメソッドの定義を含む辞書です; 辞書は *__dict__* 属性になる前にコピーされたり、ラップされることがあります。以下の 2 つの文は同じ *type* オブジェクトを生成します:

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

型オブジェクト も参照してください。

三引数形式の呼び出しに与えられたキーワード引数は、(*metaclass* を除く) クラス定義におけるキーワード引数と同様に、適切なメタクラスの機構 (通常は `__init_subclass__()`) に渡されます。

class-customization も参照してください。

バージョン 3.6 で変更: `type.__new__` をオーバーライドしていない:class:*type* のサブクラスは、オブジェクトの型を得るのに 1 引数形式を利用することができません。

vars([*object*])

モジュール、クラス、インスタンス、あるいはそれ以外の *__dict__* 属性を持つオブジェクトの、*__dict__* 属性を返します。

モジュールやインスタンスのようなオブジェクトは、更新可能な *__dict__* 属性を持っています。ただし、それ以外のオブジェクトでは *__dict__* 属性への書き込みが制限されている場合があります。書き込みに制限がある例としては、辞書を直接更新されることを防ぐために *types.MappingProxyType* を使っているクラスがあります。

引数がなければ、*vars()* は *locals()* のように振る舞います。ただし、辞書 locals への更新は無視されるため、辞書 locals は読み出し時のみ有用であることに注意してください。

指定されたオブジェクトに *__dict__* 属性がない場合 (たとえばそのクラスが `__slots__` 属性を定義している場合)、*TypeError* 例外が送出されます。

zip(**iterables*)

それぞれのイテラブルから要素を集めたイテレータを作ります。

この関数はタプルのイテレータを返し、その i 番目のタプルは引数シーケンスまたはイテラブルそれぞれの i 番目の要素を含みます。このイテレータは、入力イテラブルの中で最短のものが尽きたときに止まります。単一のイテラブル引数が与えられたときは、1 要素のタプルからなるイテレータを返します。引数がなければ、空のイテレータを返します。次と等価です:

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

イテラブルの左から右への評価順序は保証されています。そのため `zip(*[iter(s)]*n)` を使ってデータ系列を長さ n のグループにクラスタリングするイディオムが使えます。これは、各出力タプルがイテレータを n 回呼び出した結果となるよう、**同じ** イテレータを n 回繰り返します。これは入力を長さ n のチャンクに分割する効果があります。

zip() は、長い方のイテラブルの終端にある対にならない値を考慮したい場合は、等しくない長さの入力に対して使うべきではありません。そのような値が重要な場合、代わりに *itertools.zip_longest()* を使ってください。

zip() に続けて * 演算子を使うと、zip したリストを元に戻せます:

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> list(zipped)
```

(次のページに続く)

(前のページからの続き)

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

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

注釈: これは *importlib.import_module()* とは違い、日常の Python プログラミングでは必要ない高等な関数です。

この関数は import 文により呼び出されます。(*builtins* モジュールをインポートして builtins.__import__ に代入することで) この関数を置き換えて import 文のセマンティクスを変更することができますが、同様のことをするのに通常はインポートフック (**PEP 302** 参照) を利用する方が簡単で、かつデフォルトのインポート実装が使用されていることを仮定するコードとの間で問題が起きないので、このやり方は **強く** 推奨されません。*__import__()* を直接使用することも推奨されず、*importlib.import_module()* の方が好まれます。

この関数は、モジュール *name* をインポートし、*globals* と *locals* が与えられれば、パッケージのコンテキストで名前をどう解釈するか決定するのに使います。*fromlist* は *name* で与えられるモジュールからインポートされるべきオブジェクトまたはサブモジュールの名前を与ます。標準の実装では *locals* 引数はまったく使われず、*globals* は import 文のパッケージコンテキストを決定するためにのみ使われます。

level は絶対と相対どちらのインポートを使うかを指定します。0 (デフォルト) は絶対インポートのみ実行します。正の *level* の値は、*__import__()* を呼び出したディレクトリから検索対象となる親ディレクトリの数を示します (詳細は **PEP 328** を参照してください)。

name 変数が package.module 形式であるとき、通常は、*name* で指名されたモジュール **ではなく**、最上位のパッケージ (最初のドットまでの名前) が返されます。しかしながら、空でない *fromlist* 引数が与えられると、*name* で指名されたモジュールが返されます。

例えば、文 import spam は、以下のコードのようなバイトコードに帰結します:

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

文 import spam.ham は、この呼び出しになります:

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

ここで *__import__()* がどのように最上位モジュールを返しているかに注意して下さい。import 文により名前が束縛されたオブジェクトになっています。

一方で、文 from spam.ham import eggs, sausage as saus は、以下となります

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

ここで、*__import__()* から spam.ham モジュールが返されます。このオブジェクトから、インポートされる名前が取り出され、それぞれの名前として代入されます。

単純に名前からモジュール (パッケージの範囲内であるかも知れません) をインポートしたいなら、*importlib.import_module()* を使ってください。

バージョン 3.3 で変更: 負の *level* の値はサポートされなくなりました (デフォルト値の 0 に変更されます)。

脚注

第

THREE

組み込み定数

組み込み名前空間にはいくつかの定数があります。定数の一覧:

False
: *bool* 型の偽値です。False への代入は不正で、*SyntaxError* を送出します。

True
: *bool* 型の真値です。True への代入は不正で、*SyntaxError* を送出します。

None
: 型 NoneType の唯一の値です。None は、関数にデフォルト引数が渡されなかったときなどに、値の非存在を表すのに頻繁に用いられます。None への代入は不正で、*SyntaxError* を送出します。

NotImplemented
: 特殊な二項演算のメソッド (e.g. __eq__(), __lt__(), __add__(), __rsub__(), etc.) が、他の型に対して演算が実装されていないことを示すために返す特殊値です。インプレースの特殊な二項演算のメソッド (e.g. __imul__(), __iand__(), etc.) も同じ理由でこの値を返すことがあります。真偽値は真です。

注釈: 二項演算の (あるいはインプレースの) メソッドが NotImplemented を返した場合、インタープリタはもう一方の型で定義された対の演算で代用を試みます (あるいは演算によっては他の代替手段も試みます)。試行された演算全てが NotImplemented を返した場合、インタープリタは適切な例外を送出します。NotImplemented を正しく返さないと、誤解を招きかねないエラーメッセージになったり、NotImplemented が Python コードに返されるようなことになります。

例として **算術演算の実装** を参照してください。

注釈: NotImplementedError と NotImplemented は、似たような名前と目的を持っていますが、相互に変換できません。利用する際には、*NotImplementedError* を参照してください。

Ellipsis
: Ellipsis リテラル "..." と同じです。主に拡張スライス構文やユーザ定義のコンテナデータ型において使われる特殊な値です。

__debug__

この定数は、Python が -O オプションを有効にして開始されたのでなければ真です。assert 文も参照して下さい。

注釈: 名前 *None* 、*False* 、*True* 、*__debug__* は再代入できない (これらに対する代入は、たとえ属性名としてであっても *SyntaxError* が送出されます) ので、これらは「真の」定数であると考えられます。

3.1 site モジュールで追加される定数

site モジュール (-S コマンドラインオプションが指定されない限り、スタートアップ時に自動的にインポートされます) は組み込み名前空間にいくつかの定数を追加します。それらは対話的インタープリタシェルで有用ですが、プログラム中では使うべきではありません。

quit(*code=None*)

exit(*code=None*)

表示されたときに "Use quit() or Ctrl-D (i.e. EOF) to exit" のようなメッセージを表示し、呼び出されたときには指定された終了コードを伴って *SystemExit* を送出するオブジェクトです。

copyright

credits

表示あるいは呼び出されたときに、それぞれ著作権あるいはクレジットのテキストが表示されるオブジェクトです。

license

表示されたときに "Type license() to see the full license text" というメッセージを表示し、呼び出されたときには完全なライセンスのテキストをページャのような形式で (1 画面分づつ) 表示するオブジェクトです。

第

FOUR

組み込み型

以下のセクションでは、インタプリタに組み込まれている標準型について記述します。

主要な組み込み型は、数値、シーケンス、マッピング、クラス、インスタンス、および例外です。

コレクションクラスには、ミュータブルなものがあります。コレクションのメンバをインプレースに足し、引き、または並べ替えて、特定の要素を返さないメソッドは、コレクション自身ではなく None を返します。

演算には、複数の型でサポートされているものがあります; 特に、ほぼ全てのオブジェクトは、等価比較でき、真理値を判定でき、(*repr()* 関数や、わずかに異なる *str()* 関数によって) 文字列に変換できます。オブジェクトが *print()* 関数で印字されるとき、文字列に変換する関数が暗黙に使われます。

4.1 真理値判定

どのようなオブジェクトでも真理値として判定でき、if や while の条件あるいは以下のブール演算の被演算子として使えます。

オブジェクトは、デフォルトでは真と判定されます。ただしそのクラスが __bool__() メソッドを定義していて、それが False を返す場合、または __len__() メソッドを定義していて、それが 0 を返す場合は偽と判定されます。[*1] 主な組み込みオブジェクトで偽と判定されるものを次に示します:

- 偽であると定義されている定数: None と False
- 数値型におけるゼロ: 0, 0.0, 0j, Decimal(0), Fraction(0, 1)
- 空のシーケンスまたはコレクション: '', (), [], {}, set(), range(0)

ブール値の結果を返す演算および組み込み関数は、特に注釈のない限り常に偽値として 0 または False を返し、真値として 1 または True を返します。(重要な例外: ブール演算 or および and は常に被演算子のうちの一つを返します。)

[*1] これらの特殊なメソッドのさらなる情報については、Python リファレンスマニュアル (customization) を参照してください。

4.2 ブール演算 --- and, or, not

以下にブール演算を、優先順位が低い順に示します:

演算	結果	注釈
x or y	x が偽なら y, そうでなければ x	(1)
x and y	x が偽なら x, そうでなければ y	(2)
not x	x が偽なら True, そうでなければ False	(3)

注釈:

(1) この演算子は短絡評価されます。つまり第一引数が偽のときにのみ、第二引数が評価されます。

(2) この演算子は短絡評価されます。つまり第一引数が真のときにのみ、第二引数が評価されます。

(3) not は非ブール演算子よりも優先度が低いので、not a == b は not (a == b) と解釈され、a == not b は構文エラーです。

4.3 比較

Python には 8 種の比較演算があります。比較演算の優先順位は全て同じです (ブール演算より高い優先順位です)。比較は任意に連鎖できます; 例えば、x < y <= z は x < y and y <= z とほぼ等価ですが、この y は一度だけしか評価されません (どちらにしても、x < y が偽となれば z は評価されません)。

以下の表に比較演算をまとめます:

演算	意味
<	より小さい
<=	以下
>	より大きい
>=	以上
==	等しい
!=	等しくない
is	同一のオブジェクトである
is not	同一のオブジェクトでない

異なる数値型の場合を除き、異なる型のオブジェクト同士は等価になることはありません。== 演算子は常に定義されていますが、いくつかのオブジェクト型 (たとえばクラスオブジェクト) では is と同等になります。<, <=, > および >= 演算子は、それらの意味が明快である場合に限って定義されます; たとえば、オペランドのいずれかが複素数である場合、これらの演算子は *TypeError* 例外を送出します。

あるクラスの同一でないインスタンスは、通常等価でないとされますが、そのクラスが __eq__() メソッドを定義している場合は除きます。

クラスのインスタンスは、そのクラスがメソッド `__lt__()` 、`__le__()` 、`__gt__()` 、`__ge__()` のうち十分なものを定義していない限り、同じクラスの別のインスタンスや他の型のオブジェクトとは順序付けできません (一般に、比較演算子の通常の意味を求めるなら、`__lt__()` と `__eq__()` だけで十分です)。

`is` および `is not` 演算子の振る舞いはカスタマイズできません。また、これらはいかなる 2 つのオブジェクトにも適用でき、決して例外を送出しません。

`in` と `not in` という構文上で同じ優先度を持つ演算子がさらに 2 つあり、*iterable* または `__contains__()` を実装した型でサポートされています。

4.4 数値型 int, float, complex

数値型には 3 種類あります: **整数** 、**浮動小数点数** 、**複素数** です。さらに、ブール型は整数のサブタイプです。整数には精度の制限がありません。浮動小数点型はたいていは C の `double` を使って実装されています; あなたのプログラムが動作するマシンでの浮動小数点型の精度と内部表現は、*sys.float_info* から利用できます。複素数は実部と虚部を持ち、それぞれ浮動小数点数です。複素数 z から実部および虚部を取り出すには、`z.real` および `z.imag` を使ってください。(標準ライブラリには、さらに分数のための数値型 *fractions.Fraction* や、ユーザによる精度の定義が可能な浮動小数点数のための *decimal.Decimal* があります。)

数値は、数値リテラルによって、あるいは組み込み関数や演算子の戻り値として生成されます。(十六進、八進、二進数を含む) 修飾のない整数リテラルは、整数を与えます。小数点または指数表記を含む数値リテラルは浮動小数点数を与えます。数値リテラルに `'j'` または `'J'` をつけると虚数 (実部がゼロの複素数) を与え、それに整数や浮動小数点数を加えて実部と虚部を持つ複素数を得られます。

Python は型混合の算術演算に完全に対応しています: ある二項算術演算子の被演算子の数値型が互いに異なるとき、" より狭い方" の型の被演算子はもう片方の型に合わせて広げられます。ここで整数は浮動小数点数より狭く、浮動小数点数は複素数より狭いです。たくさんの異なる型の数値間での比較は、それらの厳密な数で比較したかのように振る舞います。[*2]

コンストラクタ *int()* 、*float()* 、*complex()* で、特定の型の数を生成できます。

全ての (複素数を除く) 組み込み数値型は以下の演算に対応しています (演算の優先順位については、operator-summary を参照してください):

[*2] この結果として、リスト `[1, 2]` は `[1.0, 2.0]` と等しいと見なされます。タプルの場合も同様です。

演算	結果	注釈	完全なドキュメント
`x + y`	x と y の和		
`x - y`	x と y の差		
`x * y`	x と y の積		
`x / y`	x と y の商		
`x // y`	x と y の商を切り下げたもの	(1)	
`x % y`	`x / y` の剰余	(2)	
`-x`	x の符号反転		
`+x`	x そのまま		
`abs(x)`	x の絶対値または大きさ		*abs()*
`int(x)`	x の整数への変換	(3)(6)	*int()*
`float(x)`	x の浮動小数点数への変換	(4)(6)	*float()*
`complex(re, im)`	実部 re, 虚部 im の複素数。im の既定値はゼロ。	(6)	*complex()*
`c.conjugate()`	複素数 c の共役複素数		
`divmod(x, y)`	`(x // y, x % y)` からなるペア	(2)	*divmod()*
`pow(x, y)`	x の y 乗	(5)	*pow()*
`x ** y`	x の y 乗	(5)	

注釈:

(1) 整数の除算とも呼ばれます。結果の型は整数型とは限りませんが、結果の値は整数です。結果は常に負の無限大の方向に丸められます: `1//2` は `0` 、`(-1)//2` は `-1` 、`1//(-2)` は `-1` 、そして `(-1)//(-2)` は `0` です。

(2) 複素数型には使えません。適用可能な場合には代わりに *abs()* で浮動小数点型に変換してください。

(3) 浮動小数点数から整数への変換は C 言語と同様の方法で丸め、または切り捨てられます; より明確に定義された変換を行う場合は、*math.floor()* と *math.ceil()* を参照してください。

(4) 浮動小数点数は、文字列 "nan" と "inf" を、オプションの接頭辞 "+" または "-" と共に、非数 (Not a Number (NaN)) や正、負の無限大として受け付けます。

(5) Python は、プログラム言語一般でそうであるように、`pow(0, 0)` および `0 ** 0` を `1` と定義します。

(6) 受け付けられる数値リテラルは数字 `0` から `9` または等価な Unicode (`Nd` プロパティを持つコードポイント) を含みます。

`Nd` プロパティを持つコードポイントの完全なリストは http://www.unicode.org/Public/12.1.0/ucd/extracted/DerivedNumericType.txt をご覧ください。

全ての *numbers.Real* 型 (*int* 、*float*) は以下の演算も含みます:

演算	結果
math.trunc(x)	x を *Integral* (整数) に切り捨てます
round(x[, n])	x を n 桁に丸めます。丸め方は偶数丸めです。n が省略されれば 0 がデフォルトとなります。
math.floor(x)	x 以下の最大の *Integral* (整数) を返します
math.ceil(x)	x 以上の最小の *Integral* (整数) を返します

その他の数値演算は、*math* や *cmath* モジュールをご覧ください。

4.4.1 整数型におけるビット単位演算

ビット単位演算は整数についてのみ意味を持ちます。ビット単位演算の結果は、あたかも両方の値の先頭を無限個の符号ビットで埋めたものに対して計算したかのような値になります。

二項ビット単位演算の優先順位は全て、数値演算よりも低く、比較よりも高くなっています; 単項演算 `~` の優先順位は他の単項数値演算 (`+` および `-`) と同じです。

以下の表では、ビット単位演算を優先順位が低い順に並べています:

演算	結果	注釈
`x | y`	x と y のビット単位 **論理和**	(4)
`x ^ y`	x と y のビット単位 **排他的論理和**	(4)
`x & y`	x と y のビット単位 **論理積**	(4)
`x << n`	x の n ビット左シフト	(1)(2)
`x >> n`	x の n ビット右シフト	(1)(3)
`~x`	x のビット反転	

注釈:

(1) 負値のシフト数は不正であり、*ValueError* が送出されます。

(2) n ビットの左シフトは、`pow(2, n)` による乗算と等価です。

(3) n ビットの右シフトは、`pow(2, n)` による切り捨て除算と等価です。

(4) 桁の長い方の値に少なくとも 1 つ余計に符号ビットを付け加えた幅 (計算するビット幅は `1 + max(x.bit_length(), y.bit_length())` かそれ以上) でこれらの計算を行えば、無限個の符号ビットがあるかのように計算したのと同じ結果を得るのに十分です。

4.4.2 整数型における追加のメソッド

整数型は *numbers.Integral* **抽象基底クラス** を実装します。さらに、追加のメソッドをいくつか提供します:

int.bit_length()

整数を、符号と先頭の 0 は除いて二進法で表すために必要なビットの数を返します:

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

正確には、x が非 0 なら、x.bit_length() は 2**(k-1) <= abs(x) < 2**k を満たす唯一の正の整数 k です。同様に、abs(x) が十分小さくて対数を適切に丸められるとき、k = 1 + int(log(abs(x), 2)) です。x が 0 なら、x.bit_length() は 0 を返します。

次と等価です:

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

バージョン 3.1 で追加.

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

整数を表すバイト列を返します。

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

整数は *length* バイトで表されます。整数が与えられた数のバイトで表せなければ、*OverflowError* が送出されます。

byteorder 引数は、整数を表すのに使われるバイトオーダーを決定します。*byteorder* が "big" なら、最上位のバイトがバイト配列の最初に来ます。*byteorder* が "little" なら、最上位のバイトがバイト配列の最後に来ます。ホストシステムにネイティブのバイトオーダーを要求するには、*sys.byteorder* をバイトオーダーの値として使ってください。

signed 引数は、整数を表すのに 2 の補数を使うかどうかを決定します。*signed* が False で、負の整数が与えられたなら、*OverflowError* が送出されます。*signed* のデフォルト値は False です。

バージョン 3.2 で追加.

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

与えられたバイト列の整数表現を返します。

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

引数 *bytes* は *bytes-like object* か、または bytes を生成する iterable でなければなりません。

byteorder 引数は、整数を表すのに使われるバイトオーダーを決定します。*byteorder* が `"big"` なら、最上位のバイトがバイト配列の最初に来ます。*byteorder* が `"little"` なら、最上位のバイトがバイト配列の最後に来ます。ホストシステムにネイティブのバイトオーダーを要求するには、*sys.byteorder* をバイトオーダーの値として使ってください。

signed 引数は、整数を表すのに 2 の補数を使うかどうかを決定します。

バージョン 3.2 で追加.

int.as_integer_ratio()

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

バージョン 3.8 で追加.

4.4.3 浮動小数点数に対する追加のメソッド

浮動小数点数型は、*numbers.Real* **抽象基底クラス** を実装しています。浮動小数点型はまた、以下の追加のメソッドを持ちます。

float.as_integer_ratio()

比が元の浮動小数点数とちょうど同じで分母が正である、一対の整数を返します。無限大に対しては *OverflowError* を、非数 (NaN) に対しては *ValueError* を送出します。

float.is_integer()

浮動小数点数インスタンスが有限の整数値なら `True` を、そうでなければ `False` を返します:

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

16 進表記の文字列へ、または、16 進表記からの変換をサポートする二つのメソッドがあります。Python の浮動小数点数は内部的には 2 進数で保持されるので、浮動小数点数の *10* **進数** へまたは *10* **進数** からの変換には若干の丸め誤差があります。それに対し、16 進表記では、浮動小数点数を正確に表現できます。これはデバッグのときや、数学的な用途 (numerical work) に便利でしょう。

`float.hex()`
: 浮動小数点数の 16 進文字列表現を返します。有限の浮動小数点数に対し、この表現は常に `0x` で始まり `p` と指数が続きます。

`classmethod float.fromhex(`*s*`)`
: 16 進文字列表現 *s* で表される、浮動小数点数を返すクラスメソッドです。文字列 *s* は、前や後にホワイトスペースを含んでいても構いません。

float.fromhex() はクラスメソッドですが、*float.hex()* はインスタンスメソッドであることに注意して下さい。

16 進文字列表現は以下の書式となります:

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

`sign` は必須ではなく、`+` と `-` のどちらかです。`integer` と `fraction` は 16 進数の文字列で、`exponent` は 10 進数で符号もつけられます。大文字・小文字は区別されず、最低でも 1 つの 16 進数文字を整数部もしくは小数部に含む必要があります。この制限は C99 規格のセクション 6.4.4.2 で規定されていて、Java 1.5 以降でも使われています。特に、*float.hex()* の出力は C や Java コード中で、浮動小数点数の 16 進表記として役に立つでしょう。また、C の `%a` 書式や、Java の `Double.toHexString` で書きだされた文字列は *float.fromhex()* で受け付けられます。

なお、指数部は 16 進数ではなく 10 進数で書かれ、係数に掛けられる 2 の累乗を与えます。例えば、16 進文字列 `0x3.a7p10` は浮動小数点数 `(3 + 10./16 + 7./16**2) * 2.0**10` すなわち `3740.0` を表します:

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

逆変換を `3740.0` に適用すると、同じ数を表す異なる 16 進文字列表現を返します:

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

4.4.4 数値型のハッシュ化

数 `x` と `y` に対して、型が異なっていたとしても、`x == y` であれば必ず `hash(x) == hash(y)` であることが要請されます (詳細は `__hash__()` メソッドドキュメントを参照してください)。実装の簡単さと 複数の数値型 (*int* 、*float* 、*decimal.Decimal* 、*fractions.Fraction* を含みます) 間の効率のため、Python の 数値型に対するハッシュ値はある単一の数学的関数に基づいていて、その関数はすべての有理数に対し定義されているため、*int* と *fractions.Fraction* のすべてのインスタンスと、*float* と *decimal.Decimal* のすべての有限なインスタンスに 対して適用されます。本質的には、この関数は定数の素数 `P` に対して `P` を法とする還元で与えられます。値 `P` は、*sys.hash_info* の `modulus` 属性として Python で利用できます。

CPython implementation detail: 現在使われている素数は、32 bit C long のマシンでは `P = 2**31 - 1`、64-bit C long のマシンでは `P = 2**61 - 1` です。

詳細な規則はこうです:

- `x = m / n` が非負の有理数で、`n` が `P` で割り切れないなら、`invmod(n, P)` を `n` を `P` で割った剰余の (剰余演算の意味での) 逆数を与えるものとして、`hash(x)` を `m * invmod(n, P) % P` と定義します。
- `x = m / n` が非負の有理数で、`n` が `P` で割り切れる (が `m` は割り切れない) なら、`n` は `P` で割った余りの逆数を持たず、上の規則は適用できません。この場合、`hash(x)` を定数 `sys.hash_info.inf` と定義します。
- `x = m / n` が負の有理数なら、`hash(x)` を `-hash(-x)` と定義します。その結果のハッシュが `-1` なら、`-2` に置き換えます。
- 特定の値 `sys.hash_info.inf`、`-sys.hash_info.inf`、`sys.hash_info.nan` は、正の無限大、負の無限大、nan を (それぞれ) 表すのに使われます。(すべてのハッシュ可能な nan は同じハッシュ値を持ちます。)
- 複素 (*complex*) 数 `z` に対して、実部と虚部のハッシュ値は、`hash(z.real) + sys.hash_info.imag * hash(z.imag)` の `2**sys.hash_info.width` を法とする還元を計算することにより組み合わせられ、よってこれは `range(-2**(sys.hash_info.width - 1), 2**(sys.hash_info.width - 1))` に収まります。再び、結果が `-1` なら、`-2` で置き換えられます。

上述の規則をわかりやすくするため、有理数 *float* や、*complex* のハッシュを計算する組み込みのハッシュと等価な Python コードの例を挙げます:

```
import sys, math

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

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

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

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

(次のページに続く)

(前のページからの続き)

```
    return hash_value

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

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

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

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

4.5 イテレータ型

Python はコンテナでの反復処理の概念をサポートしています。この概念は 2 つの別々のメソッドを使って実装されています; これらのメソッドを使ってユーザ定義のクラスで反復を行えるようにできます。後に詳しく述べるシーケンスは、必ず反復処理メソッドをサポートしています。

コンテナオブジェクトに反復処理をサポートさせるためには、以下のメソッドを定義しなければなりません:

`container.__iter__()`
: イテレータオブジェクトを返します。オブジェクトは後述するイテレータプロトコルをサポートする必要があります。もしコンテナが異なる型の反復処理をサポートするなら、それらの反復処理毎に追加のメソッドを提供しても構いません (複数の形式の反復処理を提供するオブジェクトの例として、幅優先探索と深さ優先探索をサポートする木構造が挙げられます)。このメソッドは Python/C API での Python オブジェクトの型構造体の `tp_iter` スロットに対応します。

イテレータオブジェクト自体は以下の 2 つのメソッドをサポートする必要があります。これらのメソッドは 2 つ合わせて *iterator protocol*: (イテレータプロトコル) を成します:

`iterator.__iter__()`
: イテレータオブジェクト自体を返します。このメソッドはコンテナとイテレータの両方を `for` および `in` 文で使えるようにするために必要です。このメソッドは Python/C API において Python オブジェクトを表す型構造体の `tp_iter` スロットに対応します。

`iterator.__next__()`
: コンテナの次のアイテムを返します。もしそれ以上アイテムが無ければ *StopIteration* 例外を送出し

ます。このメソッドは Python/C API での Python オブジェクトの型構造体の `tp_iternext` スロットに対応します。

Python では、いくつかのイテレータオブジェクトを定義して、一般のシーケンス型、特殊なシーケンス型、辞書型、その他の特殊な形式に渡って反復をサポートしています。特殊型は、イテレータプロトコルの実装以外では重要ではありません。

イテレータの *__next__()* メソッドが一旦 *StopIteration* を送出したなら、以降の呼び出しでも例外を送出し続けなければなりません。この特性に従わない実装は壊れているとみなされます。

4.5.1 ジェネレータ型

Python における *generator* (ジェネレータ) は、イテレータプロトコルを実装する便利な方法を提供します。コンテナオブジェクトの `__iter__()` メソッドがジェネレータとして実装されていれば、そのメソッドは `__iter__()` および `__next__()` メソッドを提供するイテレータオブジェクト (厳密にはジェネレータオブジェクト) を自動的に返します。ジェネレータに関する詳細な情報は、yield 式のドキュメント にあります。

4.6 シーケンス型 --- list, tuple, range

基本的なシーケンス型は 3 つあります: リスト、タプル、range オブジェクトです。**バイナリデータ** や **テキスト文字列** を処理するように仕立てられたシーケンス型は、セクションを割いて解説します。

4.6.1 共通のシーケンス演算

以下の表にある演算は、ほとんどのミュータブル、イミュータブル両方のシーケンスでサポートされています。カスタムのシーケンス型にこれらの演算を完全に実装するのが簡単になるように、*collections.abc.Sequence* ABC が提供されています。

以下のテーブルで、シーケンス演算を優先順位が低い順に挙げます。表内で、s と t は同じ型のシーケンス、n、i、j、k は整数、x は s に課された型と値の条件を満たす任意のオブジェクトです。

`in` および `not in` 演算の優先順位は比較演算と同じです。`+` (結合) および `*` (繰り返し) の優先順位は対応する数値演算と同じです。[*3]

[*3] パーザが演算対象の型を識別できるようにするために、このような優先順位でなければならないのです。

演算	結果	注釈
`x in s`	s のある要素が x と等しければ `True` , そうでなければ `False`	(1)
`x not in s`	s のある要素が x と等しければ `False`, そうでなければ `True`	(1)
`s + t`	s と t の結合	(6)(7)
`s * n` または `n * s`	s 自身を n 回足すのと同じ	(2)(7)
`s[i]`	s の 0 から数えて i 番目の要素	(3)
`s[i:j]`	s の i から j までのスライス	(3)(4)
`s[i:j:k]`	s の i から j まで、k 毎のスライス	(3)(5)
`len(s)`	s の長さ	
`min(s)`	s の最小の要素	
`max(s)`	s の最大の要素	
`s.index(x[, i[, j]])`	s 中で x が最初に出現するインデックス (インデックス i 以降からインデックス j までの範囲)	(8)
`s.count(x)`	s 中に x が出現する回数	

同じ型のシーケンスは比較もサポートしています。特に、タプルとリストは対応する要素を比較することで辞書式順序で比較されます。つまり、等しいとされるためには、すべての要素が等しく、両シーケンスの型も長さも等しくなければなりません。(完全な詳細は言語リファレンスの comparisons を参照してください。)

注釈:

(1) `in` および `not in` 演算は、一般に単純な包含判定にのみ使われますが、(*str*, *bytes*, *bytearray* のような) 特殊なシーケンスでは部分シーケンス判定にも使われます:

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

(2) 0 未満の値 n は 0 として扱われます (これは s と同じ型の空のシーケンスを表します)。シーケンス s の要素はコピーされないので注意してください; コピーではなく要素に対する参照カウントが増えます。これは Python に慣れていないプログラマをよく悩ませます。例えば以下のコードを考えます:

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

ここで、`[[]]` が空リストを含む 1 要素のリストなので、`[[]] * 3` の 3 要素はこの一つの空リスト (への参照) です。`lists` のいずれかの要素を変更すると、その一つのリストが変更されます。別々のリストのリストを作るにはこうします:

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

(次のページに続く)

(前のページからの続き)

```
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

別の説明が FAQ エントリ faq-multidimensional-list にあります。

(3) i または j が負の数の場合、インデックスはシーケンスの末端からの相対インデックスになります: `len(s) + i` または `len(s) + j` が代わりに使われます。ただし `-0` はやはり `0` であることに注意してください。

(4) s の i から j へのスライスは `i <= k < j` となるようなインデックス k を持つ要素からなるシーケンスとして定義されます。i または j が `len(s)` よりも大きい場合、`len(s)` を使います。i が省略されるか `None` だった場合、`0` を使います。j が省略されるか `None` だった場合、`len(s)` を使います。i が j 以上の場合、スライスは空のシーケンスになります。

(5) s の「i から j までステップが k のスライス」は、インデックス `x = i + n*k` (ただし `n` は `0 <= n < (j-i)/k` を満たす任意の整数) を持つ要素からなるシーケンスとして定義されます。言い換えるとインデックスは `i`, `i+k`, `i+2*k`, `i+3*k` と続き、j に達したところでストップします (ただし j は含みません)。k が正の数である場合、i または j が `len(s)` より大きければ `len(s)` を代わりに使用します。k が負の数である場合、i または j が `len(s) - 1` より大きければ `len(s) - 1` を代わりに使用します。i または j を省略または `None` を指定すると、"端" (どちらの端かは k の符号に依存) の値を代わりに使用します。なお k はゼロにできないので注意してください。また k に `None` を指定すると、`1` が指定されたものとして扱われます。

(6) イミュータブルなシーケンスの結合は、常に新しいオブジェクトを返します。これは、結合の繰り返しでシーケンスを構築する実行時間コストがシーケンスの長さの合計の二次式になることを意味します。実行時間コストを線形にするには、代わりに以下のいずれかにしてください:

- *str* オブジェクトを結合するには、リストを構築して最後に *str.join()* を使うか、*io.StringIO* インスタンスに書き込んで完成してから値を取得してください
- *bytes* オブジェクトを結合するなら、同様に *bytes.join()* や *io.BytesIO* を使うか、*bytearray* オブジェクトでインプレースに結合できます。*bytearray* オブジェクトはミュータブルで、効率のいい割り当て超過機構を備えています
- *tuple* オブジェクトを結合するなら、代わりに *list* を拡張してください
- その他の型については、関連するクラスのドキュメントを調べてください

(7) シーケンス型には、(*range* のように) 特殊なパターンに従う項目のシーケンスのみをサポートするものがあり、それらはシーケンスの結合や繰り返しをサポートしません。

(8) `index` は x が s 中に見つからないとき *ValueError* を送出します。追加の引数 i と j は、すべての実装がサポートしているわけではありません。追加の引数を渡すのは、おおよそ `s[i:j].index(x)` を使うのと等価ですが、データをコピーしなくて済むし、返されるのはスライスの最初ではなくシーケンスの最初からの相対インデクスです。

4.6.2 イミュータブルなシーケンス型

イミュータブルなシーケンス型が一般に実装している演算のうち、ミュータブルなシーケンス型がサポートしていないのは、組み込みの *hash()* だけです。

このサポートにより、*tuple* インスタンスのようなイミュータブルなシーケンスは、*dict* のキーとして使え、*set* や *frozenset* インスタンスに保存できます。

ハッシュ不可能な値を含むイミュータブルなシーケンスをハッシュ化しようとすると、*TypeError* となります。

4.6.3 ミュータブルなシーケンス型

以下のテーブルにある演算は、ほとんどのミュータブルなシーケンスでサポートされています。カスタムのシーケンス型にこれらの演算を完全に実装するのが簡単になるように、*collections.abc.MutableSequence* ABC が提供されています。

このテーブルで、s はミュータブルなシーケンス型のインスタンス、t は任意のイテラブルオブジェクト、x は s に課された型と値の条件を満たす任意のオブジェクト (例えば、*bytearray* は値の制限 `0 <= x <= 255` に合う整数のみを受け付けます) です。

演算	結果	注釈
`s[i] = x`	s の要素 i を x と入れ替えます	
`s[i:j] = t`	s の i から j 番目までのスライスをイテラブル t の内容に入れ替えます	
`del s[i:j]`	`s[i:j] = []` と同じです	
`s[i:j:k] = t`	`s[i:j:k]` の要素を t の要素と入れ替えます	(1)
`del s[i:j:k]`	リストから `s[i:j:k]` の要素を削除します	
`s.append(x)`	x をシーケンスの最後に加えます (`s[len(s):len(s)] = [x]` と同じ)	
`s.clear()`	s から全ての要素を取り除きます (`del s[:]` と同じ)	(5)
`s.copy()`	s の浅いコピーを作成します (`s[:]` と同じ)	(5)
`s.extend(t)` または `s += t`	s を t の内容で拡張します (ほとんど `s[len(s):len(s)] = t` と同じ)	
`s *= n`	s をその内容を n 回繰り返したもので更新	(6)
`s.insert(i, x)`	s の i で与えられたインデックスに x を挿入します。(`s[i:i] = [x]` と同じ)	
`s.pop()` または `s.pop(i)`	s から i 番目の要素を取り出し、また取り除きます	(2)
`s.remove(x)`	s から `s[i]` が `x` が等価となる最初の要素を取り除きます	(3)
`s.reverse()`	s をインプレースに逆転させます	(4)

注釈:

(1) t は置き換えるスライスと同じ長さでなければいけません。

(2) オプションの引数 i は標準で `-1` なので、標準では最後の要素をリストから除去して返します。

(3) `remove()` は s に x が見つからなければ *ValueError* を送出します。

(4) `reverse()` メソッドは、大きなシーケンスを反転するときの容量の節約のため、シーケンスをインプレースに変化させます。副作用としてこの演算が行われることをユーザに気づかせるために、これは反転したシーケンスを返しません。

(5) `clear()` および `copy()` は、スライシング操作をサポートしないミュータブルなコンテナ (*dict* や *set* など) のインタフェースとの一貫性のために含まれています。`copy()` は *collections.abc.MutableSequence* ABC の一部ではありませんが、ほとんどのミュータブルなシーケンスクラスが提供しています。

バージョン 3.3 で追加: `clear()` および `copy()` メソッド。

(6) 値 n は整数であるか、`__index__()` を実装したオブジェクトです。n の値がゼロまたは負数の場合、シーケンスをクリアします。**共通のシーケンス演算** で `s * n` について説明したとおり、シーケンスの要素はコピーされないので注意してください; コピーではなく要素に対する参照カウントが増えます。

4.6.4 リスト型 (list)

リストはミュータブルなシーケンスで、一般的に同種の項目の集まりを格納するために使われます (厳密な類似の度合いはアプリケーションによって異なる場合があります)。

class list([*iterable*])

リストの構成にはいくつかの方法があります:

- 角括弧の対を使い、空のリストを表す: `[]`
- 角括弧を使い、項目をカンマで区切る: `[a]`、`[a, b, c]`
- リスト内包表記を使う: `[x for x in iterable]`
- 型コンストラクタを使う: `list()` または `list(iterable)`

コンストラクタは、*iterable* の項目と同じ項目で同じ順のリストを構築します。*iterable* は、シーケンス、イテレートをサポートするコンテナ、またはイテレータオブジェクトです。*iterable* が既にリストなら、`iterable[:]` と同様にコピーが作られて返されます。例えば、`list('abc')` は `['a', 'b', 'c']` を、`list( (1, 2, 3) )` は `[1, 2, 3]` を返します。引数が与えられなければ、このコンストラクタは新しい空のリスト `[]` を作成します。

リストを作る方法は、他にも組み込み関数 *sorted()* などいろいろあります。

リストは **共通の** および **ミュータブルの** シーケンス演算をすべて実装します。リストは、更に以下のメソッドも提供します:

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

このメソッドは、項目間の `<` 比較のみを用いてリストをインプレースにソートします。例外は抑

制されません。比較演算がどこかで失敗したら、ソート演算自体が失敗します (そしてリストは部分的に変更された状態で残されるでしょう)。

sort() は、キーワードでしか渡せない 2 つの引数 (**キーワード専用引数**) を受け付けます:

key は一引数をとる関数を指定し、リストのそれぞれの要素から比較キーを取り出すのに使います (例えば、`key=str.lower`)。それぞれの項目に対応するキーは一度計算され、ソート処理全体に使われます。デフォルトの値 `None` は、別のキー値を計算せず、リストの値が直接ソートされることを意味します。

2.x 形式の *cmp* 関数を *key* 関数に変換するために、*functools.cmp_to_key()* ユーティリティが利用できます。

reverse は真偽値です。`True` がセットされた場合、リストの要素は個々の比較が反転したものとして並び替えられます。

このメソッドは、大きなシーケンスをソートするときの容量の節約のため、シーケンスをインプレースに変化させます。副作用としてこの演算が行われることをユーザに気づかせるために、これはソートしたシーケンスを返しません (新しいリストインスタンスを明示的に要求するには *sorted()* を使ってください)。

sort() メソッドは安定していることが保証されています。ソートは、等しい要素の相対順序が変更されないことが保証されていれば、安定しています。これは複数パスのソートを行なう (例えば部署でソートして、それから給与の等級でソートする) のに役立ちます。

ソートの例と簡単なチュートリアルは sortinghowto を参照して下さい。

CPython implementation detail: リストがソートされている間、または変更しようとする試みの影響中、あるいは検査中でさえ、リストは未定義です。Python の C 実装では、それらが続いている間、リストは空として出力され、リストがソート中に変更されていることを検知できたら *ValueError* を送出します。

4.6.5 タプル型 (tuple)

タプルはイミュータブルなシーケンスで、一般的に異種のデータの集まり (組み込みの *enumerate()* で作られた 2-タプルなど) を格納するために使われます。タプルはまた、同種のデータのイミュータブルなシーケンスが必要な場合 (*set* インスタンスや *dict* インスタンスに保存できるようにするためなど) にも使われます。

`class tuple(`[*iterable*]`)`

タプルの構成にはいくつかの方法があります:

- 丸括弧の対を使い、空のタプルを表す: `()`
- カンマを使い、単要素のタプルを表す: `a,` または `(a,)`
- 項目をカンマで区切る: `a, b, c` または `(a, b, c)`
- 組み込みの *tuple()* を使う: `tuple()` または `tuple(iterable)`

コンストラクタは、*iterable* の項目と同じ項目で同じ順のタプルを構築します。*iterable* は、シーケンス、イテレートをサポートするコンテナ、またはイテレータオブジェクトです。*iterable* が既にタプルなら、そのまま返されます。例えば、`tuple('abc')` は `('a', 'b', 'c')` を、`tuple( [1, 2, 3] )` は `(1, 2, 3)` を返します。引数が与えられなければ、このコンストラクタは新しい空のタプル `()` を作成します。

なお、タプルを作るのはカンマであり、丸括弧ではありません。丸括弧は省略可能ですが、空のタプルの場合や構文上の曖昧さを避けるのに必要な時は例外です。例えば、`f(a, b, c)` は三引数の関数呼び出しですが、`f((a, b, c))` は 3-タプルを唯一の引数とする関数の呼び出しです。

タプルは **共通の** シーケンス演算をすべて実装します。

異種のデータの集まりで、インデックスによってアクセスするよりも名前によってアクセスしたほうが明確になるものには、単純なタプルオブジェクトよりも *collections.namedtuple()* が向いているかもしれません。

4.6.6 range

range 型は、数のイミュータブルなシーケンスを表し、一般に `for` ループにおいて特定の回数のループに使われます。

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

range コンストラクタの引数は整数 (組み込みの *int* または `__index__` 特殊メソッドを実装するオブジェクト) でなければなりません。*step* 引数が省略された場合のデフォルト値は `1` です。*start* 引数が省略された場合のデフォルト値は `0` です。*step* が `0` の場合、*ValueError* が送出されます。

step が正の場合、range `r` の内容は式 `r[i] = start + step*i` で決定されます。ここで、`i >= 0` かつ `r[i] < stop` です。

step が負の場合も、range `r` の内容は式 `r[i] = start + step*i` で決定されます。ただし、制約条件は `i >= 0` かつ `r[i] > stop` です。

`r[0]` が値の制約を満たさない場合、range オブジェクトは空になります。range は負のインデックスをサポートしますが、これらは正のインデックスにより決定されるシーケンスの末尾からのインデックス指定として解釈されます。

range は *sys.maxsize* より大きい絶対値を含むことができますが、いくつかの機能 (*len()* など) は *OverflowError* を送出することがあります。

range の例:

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

(次のページに続く)

(前のページからの続き)

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

range は **共通の** シーケンス演算を、結合と繰り返し以外すべて実装します (range オブジェクトは厳格なパターンに従うシーケンスのみを表せ、繰り返しと結合はたいていそのパターンを破るという事実によります)。

start
: 引数 *start* の値 (この引数が与えられていない場合は 0)

stop
: 引数 *stop* の値

step
: 引数 *step* の値 (この引数が与えられていない場合は 1)

range 型が通常の *list* や *tuple* にまさる点は、*range* オブジェクトがサイズや表す範囲にかかわらず常に一定の (小さな) 量のメモリを使うことです (start、stop、step の値のみを保存し、後は必要に応じて個々の項目や部分 range を計算するためです)。

range オブジェクトは *collections.abc.Sequence* ABC を実装し、包含判定、要素インデックス検索、スライシングのような機能を提供し、負のインデックスをサポートします (**シーケンス型** *--- list, tuple, range* を参照):

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

== および != による range オブジェクトの等価性の判定は、これらをシーケンスとして比較します。つまり、二つの range オブジェクトは同じ値のシーケンスを表すなら等しいとみなされます。(なお、二つの等しいとされる range オブジェクトが異なる *start*, *stop* および *step* 属性を持つことがあります。例えば

`range(0) == range(2, 1, 3)` や `range(0, 3, 2) == range(0, 4, 2)`。)

バージョン 3.2 で変更: シーケンス ABC を実装。スライスと負のインデックスのサポート。*int* オブジェクトの帰属判定を、すべてのアイテムをイテレートする代わりに、定数時間で行います。

バージョン 3.3 で変更: (オブジェクトの同一性に基づいて比較する代わりに) range オブジェクトをそれらが定義する値のシーケンスに基づいて比較するように '==' と '!=' を定義しました。

バージョン 3.3 で追加: 属性 *start*, *stop* および *step*。

参考:

- linspace レシピ には、遅延評価される浮動小数点版の range の実装方法が載っています。

4.7 テキストシーケンス型 --- str

Python のテキストデータは *str* オブジェクト、すなわち **文字列** として扱われます。文字列は Unicode コードポイントのイミュータブルな **シーケンス** です。文字列リテラルには様々な記述方法があります:

- シングルクォート: `'"ダブル" クォートを埋め込むことができます'`
- ダブルクォート: `"' シングル' クォートを埋め込むことができます"`。
- 三重引用符: `'''三つのシングルクォート'''`, `"""三つのダブルクォート"""`

三重引用符文字列は、複数行に分けることができます。関連付けられる空白はすべて文字列リテラルに含まれます。

単式の一部であり間に空白のみを含む文字列リテラルは、一つの文字列リテラルに暗黙に変換されます。つまり、`("spam " "eggs") == "spam eggs"` です。

エスケープシーケンスを含む文字列や、ほとんどのエスケープシーケンス処理を無効にする `r` ("raw") 接頭辞などの、文字列リテラルの様々な形式は、strings を参照してください。

文字列は他のオブジェクトに *str* コンストラクタを使うことでも生成できます。

"character" 型が特別に用意されているわけではないので、文字列のインデックス指定を行うと長さ 1 の文字列を作成します。つまり、空でない文字列 *s* に対し、`s[0] == s[0:1]` です。

ミュータブルな文字列型もありませんが、ミュータブルな断片から効率よく文字列を構成するのに *str.join()* や *io.StringIO* が使えます。

バージョン 3.3 で変更: Python 2 シリーズとの後方互換性のため、文字列リテラルの `u` 接頭辞が改めて許可されました。それは文字列リテラルとしての意味には影響がなく、`r` 接頭辞と結合することはできません。

class str(*object=''*)

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

object の **文字列** 版を返します。*object* が与えられなかった場合、空文字列が返されます。それ以外の場合 `str()` の動作は、*encoding* や *errors* が与えられたかどうかによって次のように変わります。

encoding も *errors* も与えられない場合、`str(object)` は `object.__str__()` の結果を返します。これは ” 略式の” つまり読み易い *object* の文字列表現です。文字列オブジェクトに対してはその文字列自体を返します。*object* が `__str__()` メソッドを持たない場合、*str()* は代わりに *repr(object)* の結果を返します。

encoding か *errors* の少なくとも一方が与えられた場合、*object* は *bytes-like object* (たとえば *bytes* や *bytearray*) でなくてはなりません。*object* が *bytes* (もしくは *bytearray*) オブジェクトである場合は、`str(bytes, encoding, errors)` は *bytes.decode(encoding, errors)* と等価です。そうでない場合は、*bytes.decode()* が呼ばれる前に buffer オブジェクトの下層にある bytes オブジェクトが取得されます。buffer オブジェクトについて詳しい情報は、**バイナリシーケンス型** --- *bytes, bytearray, memoryview* や bufferobjects を参照してください。

encoding 引数や *errors* 引数無しに *bytes* オブジェクトを *str()* に渡すと、略式の文字列表現を返す 1 つ目の場合に該当します。(Python のコマンドラインオプション `-b` も参照してください) 例えば:

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

`str` クラスとそのメソッドについて詳しくは、**テキストシーケンス型** --- *str* や **文字列メソッド** の節を参照してください。フォーマットされた文字列を出力するには、f-strings と **カスタムの文字列書式化** の節を参照してください。加えて、**テキスト処理サービス** の節も参照してください。

4.7.1 文字列メソッド

文字列は **共通の** シーケンス演算全てに加え、以下に述べるメソッドを実装します。

文字列は、二形式の文字列書式化をサポートします。一方は柔軟さが高くカスタマイズできます (*str.format()*、**書式指定文字列の文法** 、および **カスタムの文字列書式化** を参照してください)。他方は C 言語の `printf` 形式の書式化に基づいてより狭い範囲と型を扱うもので、正しく扱うのは少し難しいですが、扱える場合ではたいていこちらのほうが高速です (*printf* **形式の文字列書式化**)。

標準ライブラリの **テキスト処理サービス** 節は、その他テキストに関する様々なユーティリティ (*re* モジュールによる正規表現サポートなど) を提供するいくつかのモジュールをカバーしています。

`str.capitalize()`

最初の文字を大文字にし、残りを小文字にした文字列のコピーを返します。

バージョン 3.8 で変更: 最初の文字が大文字ではなくタイトルケースに置き換えられるようになりました。つまり二重音字のような文字はすべての文字が大文字にされるのではなく、最初の文字だけ大文字にされるようになります。

`str.casefold()`

文字列の casefold されたコピーを返します。casefold された文字列は、大文字小文字に関係ないマッチに使えます。

casefold は、小文字化と似ていますが、より積極的です。これは文字列の大文字小文字の区別をすべて取り去ることを意図しているためです。例えば、ドイツ語の小文字 `'ß'` は `"ss"` と同じです。これは既に小文字なので、*lower()* は `'ß'` に何もしませんが、*casefold()* はこれを `"ss"` に変換します。

casefold のアルゴリズムは Unicode Standard のセクション 3.13 に記述されています。

バージョン 3.3 で追加.

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

width の長さをもつ中央寄せされた文字列を返します。パディングには *fillchar* で指定された値 (デフォルトでは ASCII スペース) が使われます。*width* が `len(s)` 以下なら元の文字列が返されます。

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

[*start*, *end*] の範囲に、部分文字列 *sub* が重複せず出現する回数を返します。オプション引数 *start* および *end* はスライス表記と同じように解釈されます。

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

文字列のエンコードされたバージョンをバイト列オブジェクトとして返します。標準のエンコーディングは `'utf-8'` です。標準とは異なるエラー処理を行うために *errors* を与えることができます。標準のエラー処理は `'strict'` で、エンコードに関するエラーは *UnicodeError* を送出します。他に利用できる値は `'ignore'`, `'replace'`, `'xmlcharrefreplace'`, `'backslashreplace'` および関数 *codecs.register_error()* によって登録された名前です。これについてはセクション **エラーハンドラ** を参照してください。利用可能なエンコーディングの一覧は、セクション **標準エンコーディング** を参照してください。

バージョン 3.1 で変更: キーワード引数のサポートが追加されました。

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

文字列が指定された *suffix* で終わるなら `True` を、そうでなければ `False` を返します。*suffix* は見つけたい複数の接尾語のタプルでも構いません。オプションの *start* があれば、その位置から判定を始めます。オプションの *end* があれば、その位置で比較を止めます。

str.expandtabs(*tabsize=8*)

文字列内の全てのタブ文字が 1 つ以上のスペースで置換された、文字列のコピーを返します。スペースの数は現在の桁 (column) 位置と *tabsize* に依存します。タブ位置は *tabsize* 文字毎に存在します (デフォルト値である 8 の場合、タブ位置は 0, 8, 16 などになります)。文字列を展開するため、まず現桁位置がゼロにセットされ、文字列が 1 文字ずつ調べられます。文字がタブ文字 (`\t`) であれば、現桁位置が次のタブ位置と一致するまで、1 つ以上のスペースが結果の文字列に挿入されます。(タブ文字自体はコピーされません。) 文字が改行文字 (`\n` もしくは `\r`) の場合、文字がコピーされ、現桁位置は 0 にリセットされます。その他の文字は変更されずにコピーされ、現桁位置は、その文字の表示のされ方 (訳注: 全角、半角など) に関係なく、1 ずつ増加します。

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

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

文字列のスライス `s[start:end]` に部分文字列 *sub* が含まれる場合、その最小のインデックスを返します。オプション引数 *start* および *end* はスライス表記と同様に解釈されます。*sub* が見つからなかった場合 `-1` を返します。

注釈: *find()* メソッドは、*sub* の位置を知りたいときにのみ使うべきです。*sub* が部分文字列であるかどうかのみを調べるには、in 演算子を使ってください:

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

str.format(*args, **kwargs)

文字列の書式化操作を行います。このメソッドを呼び出す文字列は通常の文字、または、{} で区切られた置換フィールドを含みます。それぞれの置換フィールドは位置引数のインデックスナンバー、または、キーワード引数の名前を含みます。返り値は、それぞれの置換フィールドが対応する引数の文字列値で置換された文字列のコピーです。

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

書式指定のオプションについては、書式指定文字列を規定する **書式指定文字列の文法** を参照してください。

注釈: 数値 (*int*, *float*, *complex*, *decimal.Decimal* とサブクラス) を n の整数表現型 (例: '{:n}'.format(1234)) でフォーマットするとき、LC_CTYPE ロケールと LC_NUMERIC ロケールの一方または両方が 1 バイトより長い非 ASCII 文字であると同時に異なる値である場合、この関数は localeconv() の decimal_point と thousands_sep フィールドを読み取るため一時的に LC_CTYPE ロケールに LC_NUMERIC のロケール値を設定します。この一時的な変更は他のスレッドの動作に影響します。

バージョン 3.7 で変更: 数値を n の整数表現型でフォーマットするとき、この関数は一時的に LC_CTYPE ロケールに LC_NUMERIC のロケール値を設定する場合があります。

str.format_map(*mapping*)

str.format(**mapping) と似ていますが、mapping は *dict* にコピーされず、直接使われます。これは例えば mapping が dict のサブクラスであるときに便利です:

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

バージョン 3.2 で追加.

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

find() と同様ですが、部分文字列が見つからなかったとき *ValueError* を送出します。

str.isalnum()

文字列中の全ての文字が英数字で、かつ 1 文字以上あるなら True を、そうでなければ False を返します。文字 c は以下のいずれかが True を返せば英数字です: c.isalpha() 、c.isdecimal() 、c.isdigit() 、c.isnumeric() 。

str.isalpha()

文字列中の全ての文字が英字で、かつ 1 文字以上あるなら True を、そうでなければ False を返します。英字は、Unicode 文字データベースで "Letter" として定義されているもので、すなわち、一般カテゴリプロパティ "Lm"、"Lt"、"Lu"、"Ll"、"Lo" のいずれかをもつものです。なお、これは Unicode 標準で定義されている "Alphabetic" プロパティとは異なるものです。

str.isascii()

文字列が空であるか、文字列の全ての文字が ASCII である場合に True を、それ以外の場合に False を返します。ASCII 文字のコードポイントは U+0000-U+007F の範囲にあります。

バージョン 3.7 で追加.

str.isdecimal()

文字列中の全ての文字が十進数字で、かつ 1 文字以上あるなら True を、そうでなければ False を返します。十進数字とは十進数を書くのに使われる文字のことで、たとえば U+0660 (ARABIC-INDIC DIGIT ZERO) なども含みます。正式には、Unicode の一般カテゴリ "Nd" に含まれる文字を指します。

str.isdigit()

文字列中の全ての文字が数字で、かつ 1 文字以上あるなら True を、そうでなければ False を返します。ここでの数字とは、十進数字に加えて、互換上付き数字のような特殊操作を必要とする数字を含みます。また 10 を基数とした表現ができないカローシュティー数字のような体系の文字も含みます。正式には、数字とは、プロパティ値 Numeric_Type=Digit または Numeric_Type=Decimal を持つ文字です。

str.isidentifier()

文字列が、identifiers 節の言語定義における有効な識別子であれば True を返します。

文字列 s が def や class のような予約済みの識別子か判定するには *keyword.iskeyword()* を呼び出してください。

例:

```
>>> from keyword import iskeyword

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

str.islower()

文字列中の大小文字の区別のある文字[*4] 全てが小文字で、かつ大小文字の区別のある文字が 1 文字以上あるなら True を、そうでなければ False を返します。

[*4] 大小文字の区別のある文字とは、一般カテゴリプロパティが "Lu" (Letter, uppercase (大文字))、"Ll" (Letter, lowercase (小文字))、"Lt" (Letter、titlecase (先頭が大文字)) のいずれかであるものです。

str.isnumeric()
: 文字列中の全ての文字が数を表す文字で、かつ 1 文字以上あるなら True を、そうでなければ False を返します。数を表す文字は、数字と、Unicode の数値プロパティを持つ全ての文字を含みます。たとえば U+2155 (VULGAR FRACTION ONE FIFTH)。正式には、数を表す文字は、プロパティ値 Numeric_Type=Digit、Numeric_Type=Decimal または Numeric_Type=Numeric を持つものです。

str.isprintable()
: 文字列中のすべての文字が印字可能であるか、文字列が空であれば True を、そうでなければ False を返します。非印字可能文字は、Unicode 文字データベースで "Other" または "Separator" と定義されている文字の、印字可能と見なされる ASCII space (0x20) 以外のものです。(なお、この文脈での印字可能文字は、文字列に *repr()* が呼び出されるときにエスケープすべきでない文字のことです。これは *sys.stdout* や *sys.stderr* に書き込まれる文字列の操作とは関係ありません。)

str.isspace()
: 文字列が空白文字だけからなり、かつ 1 文字以上ある場合には True を返し、そうでない場合は False を返します。

  Unicode 文字データベース (*unicodedata* を参照) で一般カテゴリが Zs ("Seperator, space") であるか、双方向クラスが WS、B、S のいずれかである場合、その文字は **空白文字** *(whitespace)* です。

str.istitle()
: 文字列がタイトルケース文字列であり、かつ 1 文字以上ある場合、例えば大文字は大小文字の区別のない文字の後にのみ続き、小文字は大小文字の区別のある文字の後ろにのみ続く場合には True を返します。そうでない場合は False を返します。

str.isupper()
: 文字列中の大小文字の区別のある文字[*4] 全てが大文字で、かつ大小文字の区別のある文字が 1 文字以上あるなら True を、そうでなければ False を返します。

str.join(*iterable*)
: *iterable* 中の文字列を結合した文字列を返します。*iterable* に *bytes* オブジェクトのような非文字列の値が存在するなら、*TypeError* が送出されます。要素間のセパレータは、このメソッドを提供する文字列です。

str.ljust(*width*[, *fillchar*])
: 長さ *width* の左揃えした文字列を返します。パディングは指定された *fillchar* (デフォルトでは ASCII スペース) を使って行われます。*width* が len(s) 以下ならば、元の文字列が返されます。

str.lower()
: 全ての大小文字の区別のある文字[*4] が小文字に変換された、文字列のコピーを返します。

  使われる小文字化のアルゴリズムは Unicode Standard のセクション 3.13 に記述されています。

str.lstrip([*chars*])
: 文字列の先頭の文字を除去したコピーを返します。引数 *chars* は除去される文字の集合を指定する文字列です。*chars* が省略されるか None の場合、空白文字が除去されます。*chars* 文字列は接頭辞ではなく、その値に含まれる文字の組み合わせ全てがはぎ取られます:

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

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

この静的メソッドは *str.translate()* に使える変換テーブルを返します。

引数を 1 つだけ与える場合、それは Unicode 序数 (整数) または文字 (長さ 1 の文字列) を、Unicode 序数、(任意長の) 文字列、または None に対応づける辞書でなければなりません。このとき、文字で指定したキーは序数に変換されます。

引数を 2 つ指定する場合、それらは同じ長さの文字列である必要があり、結果の辞書では、x のそれぞれの文字が y の同じ位置の文字に対応付けられます。第 3 引数を指定する場合、文字列を指定する必要があり、それに含まれる文字が None に対応付けられます。

str.partition(*sep*)

文字列を *sep* の最初の出現位置で区切り、3 要素のタプルを返します。タプルの内容は、区切りの前の部分、区切り文字列そのもの、そして区切りの後ろの部分です。もし区切れなければ、タプルには元の文字列そのものとその後ろに二つの空文字列が入ります。

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

文字列をコピーし、現れる部分文字列 *old* 全てを *new* に置換して返します。オプション引数 *count* が与えられている場合、先頭から *count* 個の *old* だけを置換します。

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

文字列中の領域 s[start:end] に *sub* が含まれる場合、その最大のインデックスを返します。オプション引数 *start* および *end* はスライス表記と同様に解釈されます。*sub* が見つからなかった場合 -1 を返します。

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

rfind() と同様ですが、*sub* が見つからなかった場合 *ValueError* を送出します。

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

width の長さをもつ右寄せした文字列を返します。パディングには *fillchar* で指定された文字 (デフォルトでは ASCII スペース) が使われます。*width* が len(s) 以下の場合、元の文字列が返されます。

str.rpartition(*sep*)

文字列を *sep* の最後の出現位置で区切り、3 要素のタプルを返します。タプルの内容は、区切りの前の部分、区切り文字列そのもの、そして区切りの後ろの部分です。もし区切れなければ、タプルには二つの空文字列とその後ろに元の文字列そのものが入ります。

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

sep を区切り文字とした、文字列中の単語のリストを返します。*maxsplit* が与えられた場合、文字列の**右端**から最大 *maxsplit* 回分割を行います。*sep* が指定されていない、あるいは None のとき、全ての空白文字が区切り文字となります。右から分割していくことを除けば、*rsplit()* は後ほど詳しく述べる *split()* と同様に振る舞います。

str.rstrip([*chars*])

文字列の末尾部分を除去したコピーを返します。引数 *chars* は除去される文字集合を指定する文字列です。*chars* が省略されるか None の場合、空白文字が除去されます。*chars* 文字列は接尾語ではなく、そこに含まれる文字の組み合わせ全てがはぎ取られます:

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

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

文字列を *sep* をデリミタ文字列として区切った単語のリストを返します。*maxsplit* が与えられていれば、最大で *maxsplit* 回分割されます (つまり、リストは最大 maxsplit+1 要素になります)。*maxsplit* が与えられないか -1 なら、分割の回数に制限はありません (可能なだけ分割されます)。

sep が与えられた場合、連続した区切り文字はまとめられず、空の文字列を区切っていると判断されます (例えば '1,,2'.split(',') は ['1', '', '2'] を返します)。引数 *sep* は複数の文字にもできます (例えば '1<>2<>3'.split('<>') は ['1', '2', '3'] を返します)。区切り文字を指定して空の文字列を分割すると、[''] を返します。

例えば:

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

sep が指定されていないか None の場合、異なる分割アルゴリズムが適用されます。連続する空白文字はひとつのデリミタとみなされます。また、文字列の先頭や末尾に空白があっても、結果の最初や最後に空文字列は含まれません。よって、空文字列や空白だけの文字列を None デリミタで分割すると [] が返されます。

例えば:

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

str.splitlines([*keepends*])

文字列を改行部分で分解し、各行からなるリストを返します。*keepends* に真が与えらない限り、返されるリストに改行は含まれません。

このメソッドは以下の行境界で分解します。特に、以下の境界は *universal newlines* のスーパーセットです。

表現	説明
\n	改行
\r	復帰
\r\n	改行 + 復帰
\v or \x0b	垂直タブ
\f or \x0c	改ページ
\x1c	ファイル区切り
\x1d	グループ区切り
\x1e	レコード区切り
\x85	改行 (C1 制御コード)
\u2028	行区切り
\u2029	段落区切り

バージョン 3.2 で変更: \v と \f が行境界のリストに追加されました。

例えば:

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

split() とは違って、デリミタ文字列 *sep* が与えられたとき、このメソッドは空文字列に空リストを返し、終末の改行は結果に行を追加しません:

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

比較のために `split('\n')` は以下のようになります:

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

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

文字列が指定された *prefix* で始まるなら True を、そうでなければ False を返します。*prefix* は見つけたい複数の接頭語のタプルでも構いません。オプションの *start* があれば、その位置から判定を始めます。オプションの *end* があれば、その位置で比較を止めます。

str.strip([*chars*])

文字列の先頭および末尾部分を除去したコピーを返します。引数 *chars* は除去される文字集合を指定する文字列です。*chars* が省略されるか None の場合、空白文字が除去されます。*chars* 文字列は接頭語でも接尾語でもなく、そこに含まれる文字の組み合わせ全てがはぎ取られます:

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

文字列の最も外側の先頭および末尾から、引数 *chars* 値がはぎ取られます。文字列の先頭から *chars* の文字集合に含まれない文字に達するまで、文字が削除されます。文字列の末尾に対しても同様の操作が行われます。例えば、次のようになります:

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

str.swapcase()

大文字が小文字に、小文字が大文字に変換された、文字列のコピーを返します。なお、`s.swapcase().swapcase() == s` が真であるとは限りません。

str.title()

文字列を、単語ごとに大文字から始まり、残りの文字のうち大小文字の区別があるものは全て小文字にする、タイトルケースにして返します。

例えば:

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

このアルゴリズムは、連続した文字の集まりという、言語から独立した単純な単語の定義を使います。この定義は多くの状況ではうまく機能しますが、短縮形や所有格のアポストロフィが単語の境界になってしまい、望みの結果を得られない場合があります:

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

正規表現を使うことでアポストロフィに対応できます:

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

str.translate(*table***)**

与えられた変換テーブルに基づいて文字列を構成する各文字をマッピングし、マッピング後の文字列のコピーを返します。変換テーブルは、`__getitem__()` によるインデックス指定を実装するオブジェクトである必要があります。一般的には、*mapping* または *sequence* です。Unicode 序数 (整数) でインデックス指定する場合、変換テーブルのオブジェクトは次のいずれも行うことができます。Unicode 序

数または文字列を返して文字を 1 文字以上の別の文字にマッピングすること、None を返して返り値の文字列から指定した文字を削除すること、例外 *LookupError* を送出して文字をその文字自身にマッピングすること。

文字から文字への異なる形式のマッピングから変換マップを作成するために、*str.maketrans()* が使えます。

文字のマッピングを好みに合わせてより柔軟に変更する方法については、*codecs* モジュールも参照してください。

str.upper()

全ての大小文字の区別のある文字[4] が大文字に変換された、文字列のコピーを返します。なお s.upper().isupper() は、s が大小文字の区別のある文字を含まなかったり、結果の文字の Unicode カテゴリが "Lu" ではなく例えば "Lt" (Letter, titlecase) などであったら、False になりえます。

使われる大文字化のアルゴリズムは Unicode Standard のセクション 3.13 に記述されています。

str.zfill(*width*)

長さが *width* になるよう ASCII '0' で左詰めした文字列のコピーを返します。先頭が符号接頭辞 ('+'/'-') だった場合、'0' は符号の前ではなく **後** に挿入されます。*width* が len(s) 以下の場合元の文字列を返します。

例えば:

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

4.7.2 printf 形式の文字列書式化

注釈: ここで解説されているフォーマット操作には、(タプルや辞書を正しく表示するのに失敗するなどの) よくある多くの問題を引き起こす、様々な欠陥が出現します。新しい フォーマット済み文字列リテラル や *str.format()* インターフェースや **テンプレート文字列** が、これらの問題を回避する助けになるでしょう。これらの代替手段には、それ自身に、トレードオフや、簡潔さ、柔軟さ、拡張性といった利点があります。

文字列オブジェクトには固有の操作: % 演算子 (モジュロ) があります。この演算子は文字列 **書式化** または **補間** 演算子とも呼ばれます。format % values (*format* は文字列) とすると、*format* 中の % 変換指定は *values* 中のゼロ個またはそれ以上の要素で置換されます。この動作は C 言語における sprintf() に似ています。

format が単一の引数しか要求しない場合、*values* はタプルでない単一のオブジェクトでもかまいません。[5] それ以外の場合、*values* はフォーマット文字列中で指定された項目と正確に同じ数の要素からなるタプルか、

[5] 従って、一個のタプルだけをフォーマット出力したい場合には出力したいタプルを唯一の要素とする単一のタプルを *values* に与えなくてはなりません。

単一のマップオブジェクトでなければなりません。

一つの変換指定子は 2 またはそれ以上の文字を含み、その構成要素は以下からなりますが、示した順に出現しなければなりません:

1. 指定子の開始を示す文字 '%' 。
2. マップキー (オプション)。丸括弧で囲った文字列からなります (例えば (somename)) 。
3. 変換フラグ (オプション)。一部の変換型の結果に影響します。
4. 最小のフィールド幅 (オプション)。'*' (アスタリスク) を指定した場合、実際の文字列幅が *values* タプルの次の要素から読み出されます。タプルには最小フィールド幅やオプションの精度指定の後に変換したいオブジェクトがくるようにします。
5. 精度 (オプション)。'.' (ドット) とその後に続く精度で与えられます。'*' (アスタリスク) を指定した場合、精度の桁数は *values* タプルの次の要素から読み出されます。タプルには精度指定の後に変換したい値がくるようにします。
6. 精度長変換子 (オプション)。
7. 変換型。

% 演算子の右側の引数が辞書の場合 (またはその他のマップ型の場合), 文字列中のフォーマットには、辞書に挿入されているキーを丸括弧で囲い、文字 '%' の直後にくるようにしたものが含まれていなければ **なりません** 。マップキーはフォーマット化したい値をマップから選び出します。例えば:

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

この場合、* 指定子をフォーマットに含めてはいけません (* 指定子は順番付けされたパラメタのリストが必要だからです)。

変換フラグ文字を以下に示します:

Flag	意味
'#'	値の変換に (下で定義されている) " 別の形式" を使います。
'0'	数値型に対してゼロによるパディングを行います。
'-'	変換された値を左寄せにします ('0' と同時に与えた場合、'0' を上書きします) 。
' '	(スペース) 符号付きの変換で正の数の場合、前に一つスペースを空けます (そうでない場合は空文字になります) 。
'+'	変換の先頭に符号文字 ('+' または '-') を付けます (" スペース" フラグを上書きします) 。

精度長変換子 (h, l, または L) を使うことができますが、Python では必要ないため無視されます。-- つまり、例えば %ld は %d と等価です。

変換型を以下に示します:

変換	意味	注釈
'd'	符号付き 10 進整数。	
'i'	符号付き 10 進整数。	
'o'	符号付き 8 進数。	(1)
'u'	旧式の型 -- 'd' と同じです。	(6)
'x'	符号付き 16 進数 (小文字)。	(2)
'X'	符号付き 16 進数 (大文字)。	(2)
'e'	指数表記の浮動小数点数 (小文字)。	(3)
'E'	指数表記の浮動小数点数 (大文字)。	(3)
'f'	10 進浮動小数点数。	(3)
'F'	10 進浮動小数点数。	(3)
'g'	浮動小数点数。指数部が -4 以上または精度以下の場合には小文字指数表記、それ以外の場合には 10 進表記。	(4)
'G'	浮動小数点数。指数部が -4 以上または精度以下の場合には大文字指数表記、それ以外の場合には 10 進表記。	(4)
'c'	文字一文字 (整数または一文字からなる文字列を受理します)。	
'r'	文字列 (Python オブジェクトを *repr()* で変換します)。	(5)
's'	文字列 (Python オブジェクトを *str()* で変換します)。	(5)
'a'	文字列 (Python オブジェクトを *ascii()* で変換します)。	(5)
'%'	引数を変換せず、返される文字列中では文字 '%' になります。	

注釈:

(1) 別の形式を指定（訳注: 変換フラグ # を使用）すると 8 進数を表す接頭辞 ('0o') が最初の数字の前に挿入されます。

(2) 別の形式を指定（訳注: 変換フラグ # を使用）すると 16 進数を表す接頭辞 '0x' または '0X' (使用するフォーマット文字が 'x' か 'X' に依存します) が最初の数字の前に挿入されます。

(3) この形式にした場合、変換結果には常に小数点が含まれ、それはその後ろに数字が続かない場合にも適用されます。

指定精度は小数点の後の桁数を決定し、そのデフォルトは 6 です。

(4) この形式にした場合、変換結果には常に小数点が含まれ他の形式とは違って末尾の 0 は取り除かれません。

指定精度は小数点の前後の有効桁数を決定し、そのデフォルトは 6 です。

(5) 精度が N なら、出力は N 文字に切り詰められます。

(6) **PEP 237** を参照してください。

Python 文字列には明示的な長さ情報があるので、%s 変換において '\0' を文字列の末端と仮定したりはしません。

バージョン 3.1 で変更: 絶対値が 1e50 を超える数値の %f 変換が %g 変換に置き換えられなくなりました。

4.8 バイナリシーケンス型 --- bytes, bytearray, memoryview

バイナリデータを操作するためのコア組み込み型は *bytes* および *bytearray* です。これらは、別のバイナリオブジェクトのメモリにコピーを作成すること無くアクセスするための バッファプロトコル を利用する *memoryview* でサポートされています。

array モジュールは、32 ビット整数や IEEE754 倍精度浮動小数点値のような基本データ型の、効率的な保存をサポートしています。

4.8.1 バイトオブジェクト

bytes はバイトの不変なシーケンスです。多くのメジャーなプロトコルが ASCII テキストエンコーディングをベースにしているので、bytes オブジェクトは ASCII 互換のデータに対してのみ動作する幾つかのメソッドを提供していて、文字列オブジェクトと他の多くの点で近いです。

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

まず、bytes リテラルの構文は文字列リテラルとほぼ同じで、b というプリフィックスを付けます:

- シングルクォート: `b'still allows embedded "double" quotes'`
- ダブルクォート: `b"still allows embedded 'single' quotes"`.
- 3 重クォート: `b'''3 single quotes'''`, `b"""3 double quotes"""`

bytes リテラルでは (ソースコードのエンコーディングに関係なく) ASCII 文字のみが許可されています。127 より大きい値を bytes リテラルに記述する場合は適切なエスケープシーケンスを書く必要があります。

文字列リテラルと同じく、bytes リテラルでも r プリフィックスを用いてエスケープシーケンスの処理を無効にすることができます。bytes リテラルの様々な形式やサポートされているエスケープシーケンスについては strings を参照してください。

bytes リテラルと repr 出力は ASCII テキストをベースにしたものですが、bytes オブジェクトは、各値が `0 <= x < 256` の範囲に収まるような整数 (この制限に違反しようとすると *ValueError* が発生します) の不変なシーケンスとして振る舞います。多くのバイナリフォーマットが ASCII テキストを元にした要素を持っていたり何らかのテキスト操作アルゴリズムによって操作されるものの、任意のバイナリデータが一般にテキストになっているわけではないことを強調するためにこのように設計されました (何も考えずにテキスト操作アルゴリズムを ASCII 非互換なバイナリデータフォーマットに対して行うとデータを破壊することがあります)。

リテラル以外に、幾つかの方法で bytes オブジェクトを作ることができます:

- 指定された長さの、0 で埋められた bytes オブジェクト: `bytes(10)`
- 整数の iterable から: `bytes(range(20))`

- 既存のバイナリデータからバッファプロトコルでコピーする: `bytes(obj)`

bytes ビルトイン関数も参照してください。

16 進数で 2 桁の数は正確に 1 バイトに相当するため、16 進整はバイナリデータを表現する形式として広く使われています。従って、bytes 型にはその形式でデータを読み取るための追加のクラスメソッドがあります。

classmethod fromhex(*string*)

この *bytes* のクラスメソッドは、与えられた文字列オブジェクトをデコードして bytes オブジェクトを返します。それぞれのバイトを 16 進数 2 桁で表現した文字列を指定しなければなりません。ASCII 空白文字は無視されます。

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

バージョン 3.7 で変更: *bytes.fromhex()* は文字列にある空白だけでなく、ASCII の空白文字全てをスキップするようになりました。

bytes オブジェクトをその 16 進表記に変換するための、反対向きの変換関数があります。

hex(*[sep[, bytes_per_sep]]*)

インスタンス内の 1 バイトにつき 2 つの 16 進数を含む、文字列オブジェクトを返します。

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

If you want to make the hex string easier to read, you can specify a single character separator *sep* parameter to include in the output. By default between each byte. A second optional *bytes_per_sep* parameter controls the spacing. Positive values calculate the separator position from the right, negative values from the left.

```
>>> value = b'\xf0\xf1\xf2'
>>> value.hex('-')
'f0-f1-f2'
>>> value.hex('_', 2)
'f0_f1f2'
>>> b'UUDDLRLRAB'.hex(' ', -4)
'55554444 4c524c52 4142'
```

バージョン 3.5 で追加.

バージョン 3.8 で変更: *bytes.hex()* が、16 進数出力の各バイトを分割するセパレータを挿入するためのオプションパラメータ *sep* と *bytes_per_sep* をサポートするようになりました。

bytes オブジェクトは (タプルに似た) 整数のシーケンスなので、bytes オブジェクト *b* について、`b[0]` は整数になり、`b[0:1]` は長さ 1 の bytes オブジェクトになります。(この動作は、文字列に対するインデックス指定もスライスも長さ 1 の文字列を返すのと対照的です。)

bytes オブジェクトの repr 出力はリテラル形式 (`b'...'`) になります。`bytes([46, 46, 46])` などの形式よりも便利な事が多いからです。bytes オブジェクトはいつでも `list(b)` で整数のリストに変換できます。

注釈: Python 2.x ユーザーへ: Python 2.x では多くの場面で 8bit 文字列 (2.x が提供しているビルトインのバイナリデータ型) と Unicode 文字列の間の暗黙の変換が許可されていました。これは Python がもともと 8bit 文字列しか持っていなくて、あとから Unicode テキストが追加されたので、後方互換性を維持するためのワークアラウンドでした。Python 3.x ではこれらの暗黙の変換はなくなりました。8-bit バイナリデータと Unicode テキストは明確に違うもので、bytes オブジェクトと文字列オブジェクトを比較すると常に等しくなりません。

4.8.2 bytearray オブジェクト

bytearray オブジェクトは *bytes* オブジェクトの可変なバージョンです。

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

bytearray に専用のリテラル構文はないので、コンストラクタを使って作成します:

- 空のインスタンスを作る: `bytearray()`
- 指定された長さの 0 で埋められたインスタンスを作る: `bytearray(10)`
- 整数の iterable から: `bytearray(range(20))`
- 既存のバイナリデータからバッファプロトコルを通してコピーする: `bytearray(b'Hi!')`

bytearray オブジェクトは可変なので、*bytes* と *bytearray* の操作 で解説されている bytes オブジェクトと共通の操作に加えて、*mutable* シーケンス操作もサポートしています。

bytearray ビルトイン関数も参照してください。

16 進数で 2 桁の数は正確に 1 バイトに相当するため、16 進整はバイナリデータを表現する形式として広く使われています。従って、bytearray 型にはその形式でデータを読み取るための追加のクラスメソッドがあります。

classmethod fromhex(*string*)

この *bytearray* のクラスメソッドは、与えられた文字列オブジェクトをデコードして bytearray オブジェクトを返します。それぞれのバイトを 16 進数 2 桁で表現した文字列を指定しなければなりません。ASCII 空白文字は無視されます。

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

バージョン 3.7 で変更: *bytearray.fromhex()* は文字列にある空白だけでなく、ASCII の空白文字全てをスキップするようになりました。

bytearray オブジェクトをその 16 進表記に変換するための、反対向きの変換関数があります。

hex([*sep*[, *bytes_per_sep*]])

インスタンス内の 1 バイトにつき 2 つの 16 進数を含む、文字列オブジェクトを返します。

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

バージョン 3.5 で追加.

バージョン 3.8 で変更: *bytes.hex()* と同様に、*bytearray.hex()* が、16 進数出力の各バイトを分割するセパレータを挿入するためのオプションパラメータ *sep* と *bytes_per_sep* をサポートするようになりました。

bytearray オブジェクトは整数のシーケンス (リストのようなもの) なので、bytearray オブジェクト *b* について、`b[0]` は整数になり、`b[0:1]` は長さ 1 の bytearray オブジェクトになります。(これは、文字列においてインデックス指定もスライスも長さ 1 の文字列を返すのと対照的です。)

bytearray オブジェクトの表記はバイトのリテラル形式 (`bytearray(b'...')`) を使用します。これは `bytearray([46, 46, 46])` などの形式よりも便利な事が多いためです。bytearray オブジェクトはいつでも `list(b)` で整数のリストに変換できます。

4.8.3 bytes と bytearray の操作

bytes と bytearray は両方共 **一般のシーケンス操作** をサポートしています。また、両方とも *bytes-like object* をサポートしている任意のオブジェクトを対象に操作することもできます。この柔軟性により bytes と bytearray を自由に混ぜてもエラーを起こすことなく扱うことができます。ただし、操作の結果のオブジェクトはその操作の順序に依存することになります。

注釈: 文字列のメソッドが引数として bytes を受け付けないのと同様、bytes オブジェクトと bytearray オブジェクトのメソッドは引数として文字列を受け付けません。例えば、以下のように書かなければなりません:

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

および:

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

いくつかの bytes と bytearray の操作は ASCII と互換性のあるバイナリフォーマットが使われていると仮定していますので、フォーマットの不明なバイナリデータに対して使うことは避けるべきです。こうした制約については以下で説明します。

注釈: これらの ASCII ベースの演算を使って ASCII ベースではないバイナリデータを操作すると、データを破壊する恐れがあります。

以下の bytes および bytearray オブジェクトのメソッドは、任意のバイナリデータに対して使用できます。

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

[*start*, *end*] の範囲に、部分シーケンス *sub* が重複せず出現する回数を返します。オプション引数 *start* および *end* はスライス表記と同じように解釈されます。

検索対象の部分シーケンスは、任意の *bytes-like object* または 0 から 255 の範囲の整数にできます。

バージョン 3.3 で変更: 部分シーケンスとして 0 から 255 の範囲の整数も受け取れるようになりました。

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

与えられたバイト列からデコードされた文字列を返します。デフォルトのエンコーディングは `'utf-8'` です。*errors* を与えて異なるエラー処理法を設定できます。*errors* のデフォルトは `'strict'` で、エンコーディングエラーが *UnicodeError* を送出します。設定できる他の値は、`'ignore'` 、`'replace'` 、その他の *codecs.register_error()* を通して登録された名前で、節 **エラーハンドラ** を参照してください。可能なエンコーディングのリストは、**標準エンコーディング** を参照してください。

注釈: 引数 *encoding* を *str* に渡すと *bytes-like object* を直接デコードすることができます。つまり、一時的な bytes や bytearray オブジェクトを作成する必要はありません。

バージョン 3.1 で変更: キーワード引数のサポートが追加されました。

bytes.endswith(*suffix*[, *start*[, *end*]])
bytearray.endswith(*suffix*[, *start*[, *end*]])

バイナリデータが指定された *suffix* で終わる場合は `True` を、そうでなければ `False` を返します。*suffix* は見つけたい複数の接尾語のタプルでも構いません。オプションの *start* が指定されている場合、その位置から判定を開始します。オプションの *end* が指定されている場合、その位置で比較を終了します。

検索対象の接尾語 (複数も可) は、任意の *bytes-like object* にできます。

bytes.find(*sub*[, *start*[, *end*]])
bytearray.find(*sub*[, *start*[, *end*]])

スライス `s[start:end]` に部分シーケンス *sub* が含まれる場合、データ中のその *sub* の最小のインデックスを返します。オプション引数 *start* および *end* はスライス表記と同様に解釈されます。*sub* が見つからなかった場合、`-1` を返します。

検索対象の部分シーケンスは、任意の *bytes-like object* または 0 から 255 の範囲の整数にできます。

注釈: *find()* メソッドは、*sub* の位置を知りたいときにのみ使うべきです。*sub* が部分文字列 (訳注: おそらく原文の誤り、正しくは部分シーケンス) であるかどうかのみを調べるには、`in` 演算子を使ってください:

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

バージョン 3.3 で変更: 部分シーケンスとして 0 から 255 の範囲の整数も受け取れるようになりました。

bytes.index(*sub*[, *start*[, *end*]])
bytearray.index(*sub*[, *start*[, *end*]])
find() と同様ですが、部分シーケンスが見つからなかった場合 *ValueError* を送出します。

検索対象の部分シーケンスは、任意の *bytes-like object* または 0 から 255 の範囲の整数にできます。

バージョン 3.3 で変更: 部分シーケンスとして 0 から 255 の範囲の整数も受け取れるようになりました。

bytes.join(*iterable*)
bytearray.join(*iterable*)
iterable 中のバイナリデータを結合した bytes または bytearray オブジェクトを返します。*iterable* に *str* オブジェクトなど *bytes-like objects* ではない値が含まれている場合、*TypeError* が送出されます。なお要素間のセパレータは、このメソッドを提供する bytes または bytearray オブジェクトとなります。

static bytes.maketrans(*from*, *to*)
static bytearray.maketrans(*from*, *to*)
この静的メソッドは、*bytes.translate()* に渡すのに適した変換テーブルを返します。このテーブルは、*from* 中の各バイトを *to* の同じ位置にあるバイトにマッピングします。*from* と *to* は両方とも同じ長さの *bytes-like objects* でなければなりません。

バージョン 3.1 で追加.

bytes.partition(*sep*)
bytearray.partition(*sep*)
区切り *sep* が最初に出現する位置でシーケンスを分割し、3 要素のタプルを返します。タプルの内容は、区切りの前の部分、その区切りオブジェクトまたはその bytearray 型のコピー、そして区切りの後ろの部分です。もし区切れなければ、タプルには元のシーケンスのコピーと、その後ろに二つの空の bytes または bytearray オブジェクトが入ります。

検索する区切りとしては、任意の *bytes-like object* を指定できます。

bytes.replace(*old*, *new*[, *count*])
bytearray.replace(*old*, *new*[, *count*])
部分シーケンス *old* を全て *new* に置換したシーケンスを返します。オプション引数 *count* が与えられている場合、先頭から *count* 個の *old* だけを置換します。

検索する部分シーケンスおよび置換後の部分シーケンスとしては、任意の *bytes-like object* を指定できます。

注釈: bytearray のこのメソッドはインプレースでは動作 **しません** -- 一切変化が無い場合でも、常に

新しいオブジェクトを生成します。

---

bytes.**rfind**(*sub*[, *start*[, *end*]])
bytearray.**rfind**(*sub*[, *start*[, *end*]])

シーケンス中の領域 `s[start:end]` に *sub* が含まれる場合、その最大のインデックスを返します。オプション引数 *start* および *end* はスライス表記と同様に解釈されます。*sub* が見つからなかった場合 `-1` を返します。

検索対象の部分シーケンスは、任意の *bytes-like object* または 0 から 255 の範囲の整数にできます。

バージョン 3.3 で変更: 部分シーケンスとして 0 から 255 の範囲の整数も受け取れるようになりました。

bytes.**rindex**(*sub*[, *start*[, *end*]])
bytearray.**rindex**(*sub*[, *start*[, *end*]])

rfind() と同様ですが、部分シーケンス *sub* が見つからなかった場合 *ValueError* を送出します。

検索対象の部分シーケンスは、任意の *bytes-like object* または 0 から 255 の範囲の整数にできます。

バージョン 3.3 で変更: 部分シーケンスとして 0 から 255 の範囲の整数も受け取れるようになりました。

bytes.**rpartition**(*sep*)
bytearray.**rpartition**(*sep*)

区切り *sep* が最後に出現する位置でシーケンスを分割し、3 要素のタプルを返します。タプルの内容は、区切りの前の部分、その区切りオブジェクトまたはその bytearray 型のコピー、そして区切りの後ろの部分です。もし区切れなければ、タプルには二つの空の bytes または bytearray オブジェクトと、その後ろに元のシーケンスのコピーが入ります。

検索する区切りとしては、任意の *bytes-like object* を指定できます。

bytes.**startswith**(*prefix*[, *start*[, *end*]])
bytearray.**startswith**(*prefix*[, *start*[, *end*]])

バイナリデータが指定された *prefix* で始まる場合は `True` を、そうでなければ `False` を返します。*prefix* は見つけたい複数の接頭語のタプルでも構いません。オプションの *start* が指定されている場合、その位置から判定を開始します。オプションの *end* が指定されている場合、その位置で比較を終了します。

検索対象の接頭語 (複数も可) は、任意の *bytes-like object* にできます。

bytes.**translate**(*table*, /, *delete=b''*)
bytearray.**translate**(*table*, /, *delete=b''*)

オプション引数 *delete* に現れるすべてのバイトを除去し、残ったバイトを与えられた変換テーブルに従ってマップした、バイト列やバイト配列オブジェクトのコピーを返します。変換テーブルは長さ 256 のバイト列オブジェクトでなければなりません。

変換テーブルの作成に、*bytes.maketrans()* メソッドを使うこともできます。

文字を削除するだけの変換には、*table* 引数を `None` に設定してください:

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

バージョン 3.6 で変更: *delete* はキーワード引数として指定可能になりました。

以下の bytes および bytearray オブジェクトのメソッドは、ASCII と互換性のあるバイナリフォーマットが使われていると仮定していますが、適切な引数を指定すれば任意のバイナリデータに使用できます。なお、このセクションで紹介する bytearray のメソッドはすべてインプレースで動作 **せず** 、新しいオブジェクトを生成します。

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

長さ *width* の中央寄せされたシーケンスのコピーを返します。パディングには *fillbyte* で指定された値 (デフォルトでは ASCII スペース) が使われます。*bytes* オブジェクトの場合、*width* が `len(s)` 以下なら元のシーケンスが返されます。

注釈: bytearray のこのメソッドはインプレースでは動作 **しません** -- 一切変化が無い場合でも、常に新しいオブジェクトを生成します。

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

長さ *width* の左寄せされたシーケンスのコピーを返します。パディングには *fillbyte* で指定された値 (デフォルトでは ASCII スペース) が使われます。*bytes* オブジェクトの場合、*width* が `len(s)` 以下なら元のシーケンスが返されます。

注釈: bytearray のこのメソッドはインプレースでは動作 **しません** -- 一切変化が無い場合でも、常に新しいオブジェクトを生成します。

`bytes.lstrip`([*chars*])
`bytearray.lstrip`([*chars*])

先頭から特定のバイト値を除去したコピーを返します。引数 *chars* は除去されるバイト値の集合を指定するバイナリシーケンスです — この名前は、このメソッドが通常は ASCII 文字列に対して使われることに由来しています。*chars* が省略されるか `None` の場合、ASCII の空白文字 (訳注: 空白文字の定義については *bytearray.isspace()* を参照) が除去されます。なお *chars* 引数と一致する接頭辞が除去されるのではなく、それに含まれるバイトの組み合わせ全てが除去されます:

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

除去対象のバイト値を含むバイナリシーケンスには、任意の *bytes-like object* を指定できます。

注釈: bytearray のこのメソッドはインプレースでは動作 **しません** -- 一切変化が無い場合でも、常に新しいオブジェクトを生成します。

bytes.**rjust**(*width*[, *fillbyte*])
bytearray.**rjust**(*width*[, *fillbyte*])

長さ *width* の右寄せされたシーケンスのコピーを返します。パディングには *fillbyte* で指定された値 (デフォルトでは ASCII スペース) が使われます。*bytes* オブジェクトの場合、*width* が `len(s)` 以下なら元のシーケンスが返されます。

注釈: bytearray のこのメソッドはインプレースでは動作 **しません** -- 一切変化が無い場合でも、常に新しいオブジェクトを生成します。

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

sep を区切りとして、同じ型の部分シーケンスに分割します。*maxsplit* が与えられた場合、シーケンスの **右端** から最大 *maxsplit* 回だけ分割を行います。*sep* が指定されていないか `None` のとき、ASCII 空白文字の組み合わせで作られる部分シーケンスすべてが区切りとなります。右から分割していくことを除けば、*rsplit()* は後ほど詳しく述べる *split()* と同様に振る舞います。

bytes.**rstrip**([*chars*])
bytearray.**rstrip**([*chars*])

末尾から特定のバイト値を除去したコピーを返します。引数 *chars* は除去されるバイト値の集合を指定するバイナリシーケンスです — この名前は、このメソッドが通常は ASCII 文字列に対して使われることに由来しています。*chars* が省略されるか `None` の場合、ASCII の空白文字 (訳注: 空白文字の定義については *bytearray.isspace()* を参照) が除去されます。なお *chars* 引数と一致する接尾辞が除去されるのではなく、それに含まれるバイトの組み合わせ全てが除去されます:

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

除去対象のバイト値を含むバイナリシーケンスには、任意の *bytes-like object* を指定できます。

注釈: bytearray のこのメソッドはインプレースでは動作 **しません** -- 一切変化が無い場合でも、常に新しいオブジェクトを生成します。

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

sep を区切りとして、同じ型の部分シーケンスに分割します。*maxsplit* が与えられ、かつ負の数でない場合、シーケンスの **左端** から最大 *maxsplit* 回だけ分割を行います (したがって結果のリストの要素数

は最大で maxsplit+1 になります)。*maxsplit* が指定されていないか -1 のとき、分割の回数に制限はありません (可能なだけ分割されます)。

sep が与えられた場合、連続した区切り用バイト値はまとめられず、空の部分シーケンスを区切っていると判断されます (例えば b'1,,2'.split(b',') は [b'1', b'', b'2'] を返します)。引数 *sep* は複数バイトのシーケンスにもできます (例えば b'1<>2<>3'.split(b'<>') は [b'1', b'2', b'3'] を返します)。空のシーケンスを分割すると、分割するオブジェクトの型によって [b''] または [bytearray(b'')] が返ります。引数 *sep* には、あらゆる *bytes-like object* を指定できます。

例えば:

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

sep が指定されていないか None の場合、異なる分割アルゴリズムが適用されます。連続する ASCII 空白文字はひとつの区切りとみなされ、またシーケンスの先頭や末尾に空白があっても、結果の最初や最後に空のシーケンスは含まれません。したがって区切りを指定せずに空のシーケンスや ASCII 空白文字だけのシーケンスを分割すると、[] が返されます。

例えば:

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

bytes.**strip**([*chars*])
bytearray.**strip**([*chars*])

先頭および末尾から特定のバイト値を除去したコピーを返します。引数 *chars* は除去されるバイト値の集合を指定するバイナリシーケンスです — この名前は、このメソッドが通常は ASCII 文字列に対して使われることに由来しています。*chars* が省略されるか None の場合、ASCII の空白文字 (訳注: 空白文字の定義については *bytearray.isspace()* を参照) が除去されます。なお *chars* 引数と一致する接頭辞および接尾辞が除去されるのではなく、それに含まれるバイトの組み合わせ全てが除去されます:

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

除去対象のバイト値を含むバイナリシーケンスには、任意の *bytes-like object* を指定できます。

注釈: bytearray のこのメソッドはインプレースでは動作 **しません** -- 一切変化が無い場合でも、常に新しいオブジェクトを生成します。

以下の bytes および bytearray オブジェクトのメソッドは、ASCII と互換性のあるバイナリフォーマットが使われていると仮定しており、任意のバイナリデータに対して使用すべきではありません。なお、このセクションで紹介する bytearray のメソッドはすべてインプレースで動作 **せず** 、新しいオブジェクトを生成します。

bytes.capitalize()
bytearray.capitalize()

各バイトを ASCII 文字と解釈して、最初のバイトを大文字にし、残りを小文字にしたシーケンスのコピーを返します。ASCII 文字と解釈できないバイト値は、変更されません。

注釈: bytearray のこのメソッドはインプレースでは動作 **しません** -- 一切変化が無い場合でも、常に新しいオブジェクトを生成します。

bytes.expandtabs(*tabsize=8*)
bytearray.expandtabs(*tabsize=8*)

桁 (column) 位置と指定されたタブ幅 (tab size) に応じて、全ての ASCII タブ文字を 1 つ以上の ASCII スペース文字に置換したシーケンスのコピーを返します。ここで *tabsize* バイトごとの桁位置をタブ位置とします (デフォルト値である 8 の場合、タブ位置は 0 桁目、8 桁目、16 桁目、と続いていきます)。シーケンスを展開するにあたって、まず現桁位置をゼロに設定し、シーケンスを 1 バイトずつ調べていきます。もしバイト値が ASCII タブ文字 (`b'\t'`) であれば、現桁位置が次のタブ位置と一致するまで 1 つ以上の ASCII スペース文字を結果のシーケンスに挿入していきます（ASCII タブ文字自体はコピーしません)。もしバイト値が ASCII 改行文字 (`b'\n'` もしくは `b'\r'`) であれば、そのままコピーした上で現桁位置を 0 にリセットします。その他のバイト値については変更せずにコピーし、そのバイト値の表示のされ方（訳注: 全角、半角など）に関わらず現桁位置を 1 つ増加させます:

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

注釈: bytearray のこのメソッドはインプレースでは動作 **しません** -- 一切変化が無い場合でも、常に新しいオブジェクトを生成します。

bytes.isalnum()
bytearray.isalnum()

シーケンスが空でなく、かつ全てのバイト値が ASCII 文字のアルファベットまたは数字である場合は `True` を、そうでなければ `False` を返します。ここでの ASCII 文字のアルファベットとはシーケンス `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'` に含まれるバイト値です。

ASCII 文字の数字とは b'0123456789' に含まれるバイト値です。

例えば:

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

bytes.isalpha()

bytearray.isalpha()

シーケンスが空でなく、かつ全てのバイト値が ASCII 文字のアルファベットである場合は True を、そうでなければ False を返します。ここでの ASCII 文字のアルファベットとはシーケンス b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ' に含まれるバイト値です。

例えば:

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

bytes.isascii()

bytearray.isascii()

シーケンスが空であるか、シーケンスの全てのバイトが ASCII である場合に True を、それ以外の場合に False を返します。ASCII バイトは 0-0x7F の範囲にあります。

バージョン 3.7 で追加.

bytes.isdigit()

bytearray.isdigit()

シーケンスが空でなく、かつ全てのバイト値が ASCII 文字の数字である場合は True を、そうでなければ False を返します。ここでの ASCII 文字の数字とは b'0123456789' に含まれるバイト値です。

例えば:

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

bytes.islower()

bytearray.islower()

シーケンス中に小文字アルファベットの ASCII 文字が一つ以上あり、かつ大文字アルファベットの ASCII 文字が一つも無い場合に True を返します。そうでなければ False を返します。

例えば:

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

(次のページに続く)

(前のページからの続き)

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

ここでの小文字の ASCII 文字とは `b'abcdefghijklmnopqrstuvwxyz'` に含まれるバイト値です。また大文字の ASCII 文字とは `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` に含まれるバイト値です。

bytes.isspace()
bytearray.isspace()

シーケンスが空でなく、かつ全てのバイト値が ASCII 空白文字である場合は `True` を、そうでなければ `False` を返します。ここでの ASCII 空白文字とはシーケンス `b' \t\n\r\x0b\f'` に含まれるバイト値です (半角スペース、タブ、ラインフィード、キャリッジリターン、垂直タブ、フォームフィード) 。

bytes.istitle()
bytearray.istitle()

シーケンスが空でなく、かつ ASCII のタイトルケース文字列になっている場合は `True` を、そうでなければ `False` を返します。「タイトルケース文字列」の定義については *bytes.title()* を参照してください。

例えば:

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

bytes.isupper()
bytearray.isupper()

シーケンス中に大文字アルファベットの ASCII 文字が一つ以上あり、かつ小文字アルファベットの ASCII 文字が一つも無い場合に `True` を返します。そうでなければ `False` を返します。

例えば:

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

ここでの小文字の ASCII 文字とは `b'abcdefghijklmnopqrstuvwxyz'` に含まれるバイト値です。また大文字の ASCII 文字とは `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` に含まれるバイト値です。

bytes.lower()
bytearray.lower()

シーケンスに含まれる大文字アルファベットの ASCII 文字を全て小文字アルファベットに変換したシーケンスのコピーを返します。

例えば:

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

ここでの小文字の ASCII 文字とは `b'abcdefghijklmnopqrstuvwxyz'` に含まれるバイト値です。また大文字の ASCII 文字とは `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` に含まれるバイト値です。

注釈: bytearray のこのメソッドはインプレースでは動作 **しません** -- 一切変化が無い場合でも、常に新しいオブジェクトを生成します。

---

bytes.splitlines(*keepends=False*)

bytearray.splitlines(*keepends=False*)

バイナリシーケンスを ASCII の改行コードで分割し、各行をリストにして返します。このメソッドは *universal newlines* アプローチで行を分割します。*keepends* 引数に真を与えた場合を除き、改行コードは結果のリストに含まれません。

例えば:

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

split() とは違って、空シーケンスに対して区切り *sep* を与えて呼び出すと空のリストを返します。またシーケンス末尾に改行コードがある場合、(訳註: その後ろに空行があるとは判断せず) 余分な行を生成することはありません:

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

bytes.swapcase()

bytearray.swapcase()

シーケンスに含まれる小文字アルファベットの ASCII 文字を全て大文字アルファベットに変換し、さらに大文字アルファベットを同様に小文字アルファベットに変換したシーケンスのコピーを返します。

例えば:

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

ここでの小文字の ASCII 文字とは `b'abcdefghijklmnopqrstuvwxyz'` に含まれるバイト値です。また大文字の ASCII 文字とは `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` に含まれるバイト値です。

str.swapcase() とは違い、バイナリバージョンのこちらでは `bin.swapcase().swapcase() == bin` が常に成り立ちます。一般的に Unicode 文字の大文字小文字変換は対称的ではありませんが、ASCII 文字の場合は対称的です。

注釈: bytearray のこのメソッドはインプレースでは動作 **しません** -- 一切変化が無い場合でも、常に新しいオブジェクトを生成します。

bytes.title()
bytearray.title()

タイトルケース化したバイナリシーケンスを返します。具体的には、各単語が大文字アルファベットの ASCII 文字で始まり、かつ残りの文字が小文字アルファベットになっているシーケンスが返ります。大文字小文字の区別が無いバイト値については変更されずそのままになります。

例えば:

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

ここでの小文字の ASCII 文字とは `b'abcdefghijklmnopqrstuvwxyz'` に含まれるバイト値です。また大文字の ASCII 文字とは `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` に含まれるバイト値です。その他のバイト値については、大文字小文字の区別はありません。

このアルゴリズムは、連続した文字の集まりという、言語から独立した単純な単語の定義を使います。この定義は多くの状況ではうまく機能しますが、短縮形や所有格のアポストロフィが単語の境界になってしまい、望みの結果を得られない場合があります:

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

正規表現を使うことでアポストロフィに対応できます:

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

注釈: bytearray のこのメソッドはインプレースでは動作 **しません** -- 一切変化が無い場合でも、常に新しいオブジェクトを生成します。

bytes.upper()
bytearray.upper()

シーケンスに含まれる小文字アルファベットの ASCII 文字を全て大文字アルファベットに変換したシーケンスのコピーを返します。

例えば:

```
>>> b'Hello World'.upper()
b'HELLO WORLD'
```

ここでの小文字の ASCII 文字とは `b'abcdefghijklmnopqrstuvwxyz'` に含まれるバイト値です。また大文字の ASCII 文字とは `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` に含まれるバイト値です。

注釈: bytearray のこのメソッドはインプレースでは動作 **しません** -- 一切変化が無い場合でも、常に新しいオブジェクトを生成します。

`bytes.zfill(`*width*`)`
`bytearray.zfill(`*width*`)`

長さが *width* になるよう ASCII `b'0'` で左詰めしたシーケンスのコピーを返します。先頭が符号接頭辞 (`b'+'`/`b'-'`) だった場合、`b'0'` は符号の前ではなく **後** に挿入されます。*bytes* オブジェクトの場合、*width* が `len(seq)` 以下であれば元のシーケンスが返ります。

例えば:

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

注釈: bytearray のこのメソッドはインプレースでは動作 **しません** -- 一切変化が無い場合でも、常に新しいオブジェクトを生成します。

4.8.4 printf 形式での bytes の書式化

注釈: ここで述べる書式化演算には様々な癖があり、よく間違いの元になっています (タプルや辞書を正しく表示できないなど)。もし表示する値がタプルや辞書かもしれない場合、それをタプルに包むようにしてください。

---

bytes オブジェクト (`bytes`/`bytearray`) には固有の操作: `%` 演算子 (モジュロ) があります。この演算子は bytes の **書式化** または **補間** 演算子とも呼ばれます。`format % values` (*format* は bytes オブジェクト) とすると、*format* 中の `%` 変換指定は *values* 中のゼロ個またはそれ以上の要素で置換されます。この動作は C 言語における `sprintf()` に似ています。

format が単一の引数しか要求しない場合、*values* はタプルではない単一のオブジェクトで問題ありません。[*5] それ以外の場合、*values* は書式シーケンス (訳註: 先の例での *format*) 中で指定された項目と正確に同じ数の要素を含むタプルか、単一のマッピング型のオブジェクト (たとえば辞書) でなければなりません。

一つの変換指定子は 2 またはそれ以上の文字を含み、その構成要素は以下からなりますが、示した順に出現しなければなりません:

1. 指定子の開始を示す文字 `'%'` 。
2. マップキー (オプション)。丸括弧で囲った文字列からなります (例えば `(somename)`) 。
3. 変換フラグ (オプション)。一部の変換型の結果に影響します。
4. 最小のフィールド幅 (オプション)。`'*'` (アスタリスク) を指定した場合、実際の文字列幅が *values* タプルの次の要素から読み出されます。タプルには最小フィールド幅やオプションの精度指定の後に変換したいオブジェクトがくるようにします。
5. 精度 (オプション)。`'.'` (ドット) とその後に続く精度で与えられます。`'*'` (アスタリスク) を指定した場合、精度の桁数は *values* タプルの次の要素から読み出されます。タプルには精度指定の後に変換したい値がくるようにします。
6. 精度長変換子 (オプション)。
7. 変換型。

% 演算子の右側の引数が辞書の場合 (またはその他のマッピング型の場合) 、bytes オブジェクト中のフォーマットには、辞書のキーを丸括弧で囲って文字 `'%'` の直後に書いたものが含まれていなければ **なりません** 。マップキーは書式化したい値をマッピングから選び出します。例えば:

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

この場合、* 指定子をフォーマットに含めてはいけません (* 指定子は順番付けされたパラメタのリストが必要だからです)。

変換フラグ文字を以下に示します:

Flag	意味
`'#'`	値の変換に (下で定義されている) ” 別の形式” を使います。
`'0'`	数値型に対してゼロによるパディングを行います。
`'-'`	変換された値を左寄せにします (`'0'` と同時に与えた場合、`'0'` を上書きします) 。
`' '`	(スペース) 符号付きの変換で正の数の場合、前に一つスペースを空けます (そうでない場合は空文字になります) 。
`'+'`	変換の先頭に符号文字 (`'+'` または `'-'`) を付けます (” スペース” フラグを上書きします) 。

精度長変換子 (`h`, `l`, または `L`) を使うことができますが、Python では必要ないため無視されます。-- つまり、例えば `%ld` は `%d` と等価です。

変換型を以下に示します:

変換	意味	注釈
'd'	符号付き 10 進整数。	
'i'	符号付き 10 進整数。	
'o'	符号付き 8 進数。	(1)
'u'	旧式の型 -- 'd' と同じです。	(8)
'x'	符号付き 16 進数 (小文字)。	(2)
'X'	符号付き 16 進数 (大文字)。	(2)
'e'	指数表記の浮動小数点数 (小文字)。	(3)
'E'	指数表記の浮動小数点数 (大文字)。	(3)
'f'	10 進浮動小数点数。	(3)
'F'	10 進浮動小数点数。	(3)
'g'	浮動小数点数。指数部が -4 以上または精度以下の場合には小文字指数表記、それ以外の場合には 10 進表記。	(4)
'G'	浮動小数点数。指数部が -4 以上または精度以下の場合には大文字指数表記、それ以外の場合には 10 進表記。	(4)
'c'	1 バイト (整数または要素 1 つの bytes/bytearray オブジェクトを受理します)	
'b'	バイナリシーケンス (buffer protocol をサポートするか、__bytes__() メソッドがあるオブジェクト)	(5)
's'	's' は 'b' の別名です。Python 2/3 の両方を対象としたコードでのみ使用すべきです。	(6)
'a'	バイナリシーケンス (Python オブジェクトを repr(obj).encode('ascii', 'backslashreplace') で変換します)。	(5)
'r'	'r' は 'a' の別名です。Python 2/3 の両方を対象としたコードでのみ使用すべきです。	(7)
'%'	引数を変換せず、返される文字列中では文字 '%' になります。	

注釈:

(1) 別の形式を指定（訳注: 変換フラグ # を使用）すると 8 進数を表す接頭辞 ('0o') が最初の数字の前に挿入されます。

(2) 別の形式を指定（訳注: 変換フラグ # を使用）すると 16 進数を表す接頭辞 '0x' または '0X' (使用するフォーマット文字が 'x' か 'X' に依存します) が最初の数字の前に挿入されます。

(3) この形式にした場合、変換結果には常に小数点が含まれ、それはその後ろに数字が続かない場合にも適用されます。

指定精度は小数点の後の桁数を決定し、そのデフォルトは 6 です。

(4) この形式にした場合、変換結果には常に小数点が含まれ他の形式とは違って末尾の 0 は取り除かれません。

指定精度は小数点の前後の有効桁数を決定し、そのデフォルトは 6 です。

(5) 精度が N なら、出力は N 文字に切り詰められます。

(6) b'%s' は非推奨ですが、3.x 系では削除されません。

(7) `b'%r'` は非推奨ですが、3.x 系では削除されません。

(8) **PEP 237** を参照してください。

注釈: bytearray のこのメソッドはインプレースでは動作 **しません** -- 一切変化が無い場合でも、常に新しいオブジェクトを生成します。

参考:

PEP 461 - bytes と bytearray への % 書式化の追加

バージョン 3.5 で追加.

4.8.5 メモリビュー

memoryview オブジェクトは、Python コードが バッファプロトコル をサポートするオブジェクトの内部データへ、コピーすることなくアクセスすることを可能にします。

class memoryview(*obj*)

obj を参照する *memoryview* を作成します。*obj* はバッファプロトコルをサポートしていなければなりません。バッファプロトコルをサポートする組み込みオブジェクトには、*bytes* 、*bytearray* などがあります。

memoryview は元となるオブジェクト *obj* が扱うメモリーの最小単位を **要素** として扱います。多くの単純なオブジェクト、例えば *bytes* や *bytearray* では、要素は単バイトになりますが、他の *array.array* 等の型では、要素はより大きくなりえます。

メモリビューの長さ `len(view)` は、*tolist* で得られるリストの長さとなります。`view.ndim = 0` なら、長さは 1 です。`view.ndim = 1` なら、長さはビューの要素数と等しいです。より高次元では、長さはビューのネストされたリスト表現の長さと等しいです。要素一つあたりのバイト数は *itemsize* 属性から取得できます。

memoryview はスライスおよびインデックス指定で内容を取得できます。一次元のスライスは部分ビューになります:

```
>>> v = memoryview(b'abcefg')
>>> v[1]
98
>>> v[-1]
103
>>> v[1:4]
<memory at 0x7f3ddc9f4350>
>>> bytes(v[1:4])
b'bce'
```

もしメモリビューの *format* が *struct* モジュールによって定義されているネイティブのフォーマット指定子であれば、整数または整数のタプルでのインデックス指定により適切な型の **要素** *1* **つ** を得る

ことができます。一次元のメモリビューでは、整数または整数 1 つのタプルでインデックス指定できます。多次元のメモリビューでは、その次元数を *ndim* としたとき、ちょうど *ndim* 個の整数からなるタプルでインデックス指定できます。ゼロ次元のメモリビューでは、空のタプルでインデックス指定できます。

format が単バイト単位ではない例を示します:

```
>>> import array
>>> a = array.array('l', [-11111111, 22222222, -33333333, 44444444])
>>> m = memoryview(a)
>>> m[0]
-11111111
>>> m[-1]
44444444
>>> m[::2].tolist()
[-11111111, -33333333]
```

メモリビューの参照しているオブジェクトが書き込み可能であれば、一次元スライスでの代入が可能です。ただしサイズの変更はできません:

```
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
bytearray(b'zbcefg')
>>> v[1:4] = b'123'
>>> data
bytearray(b'z123fg')
>>> v[2:3] = b'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rvalue have different structures
>>> v[2:6] = b'spam'
>>> data
bytearray(b'z1spam')
```

'B', 'b', 'c' いずれかのフォーマットのハッシュ可能な (読み出し専用の) 型の 1 次元メモリビューもまた、ハッシュ可能です。ハッシュは `hash(m) == hash(m.tobytes())` として定義されています:

```
>>> v = memoryview(b'abcefg')
>>> hash(v) == hash(b'abcefg')
True
>>> hash(v[2:4]) == hash(b'ce')
True
>>> hash(v[::-2]) == hash(b'abcefg'[::-2])
True
```

バージョン 3.3 で変更: 1 次元のメモリビューがスライス可能になりました。'B', 'b', 'c' いずれかのフォーマットの 1 次元のメモリビューがハッシュ可能になりました。

バージョン 3.4 で変更: memoryview は自動的に *collections.abc.Sequence* へ登録されるようになりました。

バージョン 3.5 で変更: メモリビューは整数のタプルでインデックス指定できるようになりました。

memoryview にはいくつかのメソッドがあります:

__eq__(*exporter*)

memoryview と **PEP 3118** エクスポーターは、shape が同じで、*struct* のフォーマットで解釈したときの値が同じ場合に同値になります。

tolist() がサポートしている *struct* フォーマットの一部では、`v.tolist() == w.tolist()` が成り立つときに `v == w` になります:

```
>>> import array
>>> a = array.array('I', [1, 2, 3, 4, 5])
>>> b = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> c = array.array('b', [5, 3, 1])
>>> x = memoryview(a)
>>> y = memoryview(b)
>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[::-2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True
```

どちらかの書式文字列が *struct* モジュールにサポートされていなければ、(書式文字列とバッファの内容が同一でも) オブジェクトは常に等しくないものとして比較されます:

```
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [("x", c_long), ("y", c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

浮動小数点数の場合と同様 memoryview オブジェクトに対する `v is w` は `v == w` を意味しないことに注意してください。

バージョン 3.3 で変更: 以前のバージョンは、要素フォーマットと論理的な配列構造を無視して生のメモリを比較していました。

tobytes(*order=None*)

バッファ中のデータをバイト文字列として返します。これはメモリビューに対して *bytes* コンストラクタを呼び出すのと同等です。

```
>>> m = memoryview(b"abc")
>>> m.tobytes()
b'abc'
>>> bytes(m)
b'abc'
```

連続でない配列については、結果はすべての要素がバイトに変換されたものを含むフラットなリスト表現に等しくなります。*tobytes()* は、*struct* モジュール文法にないものを含むすべての書式文字列をサポートします。

バージョン 3.8 で追加: *order* は {'C', 'F', 'A'} のいずれかを取ることができます。*order* が 'C' か 'F' の場合、元の配列は C または Fortran のデータ並びにそれぞれ変換されます。連続したデータに対するビューの場合、'A' は物理メモリ上のデータの正確なコピーを返します。特に、メモリ上における Fortran のデータ並びは保存されます。不連続なデータに対するビューの場合、データはまず C のデータ並びに変換されます。*order=None* は *order='C'* と同じです。

hex([*sep*[, *bytes_per_sep*]])

バッファ中の各バイトを 2 つの 16 進数で表した文字列を返します:

```
>>> m = memoryview(b"abc")
>>> m.hex()
'616263'
```

バージョン 3.5 で追加.

バージョン 3.8 で変更: *bytes.hex()* と同様に、*memoryview.hex()* は、16 進数出力のバイト文字列を分割するセパレータを挿入するためのオプションパラメータ *sep* と *bytes_per_sep* をサポートするようになりました。

tolist()

バッファ中のデータを要素のリストとして返します。

```
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]
```

バージョン 3.3 で変更: *tolist()* が *struct* モジュール文法に含まれるすべての単一文字の native フォーマットと多次元の表現をサポートするようになりました。

toreadonly()

読み込み専用のメモリビューオブジェクトを返します。元のメモリビューオブジェクトは変更されません。

```
>>> m = memoryview(bytearray(b'abc'))
>>> mm = m.toreadonly()
>>> mm.tolist()
[89, 98, 99]
>>> mm[0] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot modify read-only memory
>>> m[0] = 43
>>> mm.tolist()
[43, 98, 99]
```

バージョン 3.8 で追加.

release()

memoryview オブジェクトによって晒されている、元になるバッファを解放します。多くのオブジェクトはビューに支配されているときに特殊なふるまいをします (例えば、*bytearray* は大きさの変更を一時的に禁止します)。ですから、release() を呼び出すことは、これらの制約をできるだけ早く取り除く (そしてぶら下がったリソースをすべて解放する) のに便利です。

このメソッドが呼ばれた後、このビュー上のそれ以上の演算は *ValueError* を送出します (複数回呼ばれえる *release()* 自身は除きます):

```
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

コンテキストマネージャプロトコルは、with 文を使って同様の効果を得るのに使えます:

```
>>> with memoryview(b'abc') as m:
...     m[0]
...
97
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

バージョン 3.2 で追加.

cast(*format*[, *shape*])

memoryview を新しいフォーマットか shape にキャストします。*shape* はデフォルトで [byte_length//new_itemsize] で、1 次元配列になります。戻り値は memoryview ですが、バッファー自体はコピーされません。サポートされている変換は 1 次元配列 -> C 言語型の連続配列 と C 言語型の連続配列 -> 1 次元配列 です（参考: *contiguous*）。

キャスト後のフォーマットは *struct* 文法の単一要素ネイティブフォーマットに制限されます。

フォーマットのうちの一つはバイトフォーマット ('B', 'b', 'c') でなければなりません。結果のバイト長はオリジナルの長さと同じでなければなりません。

1 次元 long から 1 次元 unsigned byte へのキャスト:

```
>>> import array
>>> a = array.array('l', [1,2,3])
>>> x = memoryview(a)
>>> x.format
'l'
>>> x.itemsize
8
>>> len(x)
3
>>> x.nbytes
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24
```

1 次元 unsigned byte から 1 次元 char へのキャスト:

```
>>> b = bytearray(b'zyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview: invalid value for format "B"
>>> y = x.cast('c')
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')
```

1 次元 byte から 3 次元 int へ、そして 1 次元 signed char へのキャスト:

```
>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
>>> y.format
'i'
>>> y.itemsize
4
>>> len(y)
```

(次のページに続く)

(前のページからの続き)

```
2
>>> y.nbytes
48
>>> z = y.cast('b')
>>> z.format
'b'
>>> z.itemsize
1
>>> len(z)
48
>>> z.nbytes
48
```

1 次元 unsigned long から 2 次元 unsigned long へのキャスト:

```
>>> buf = struct.pack("L"*6, *list(range(6)))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)
2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]
```

バージョン 3.3 で追加.

バージョン 3.5 で変更: 単バイトのビューへキャストする場合、キャスト元のフォーマットについて制約は無くなりました。

読み出し専用の属性もいくつか使えます:

obj

memoryview が参照しているオブジェクト:

```
>>> b  = bytearray(b'xyz')
>>> m = memoryview(b)
>>> m.obj is b
True
```

バージョン 3.3 で追加.

nbytes

`nbytes == product(shape) * itemsize == len(m.tobytes())`. その配列が連続表現において利用するスペースです。これは `len(m)` と一致するとは限りません:

```
>>> import array
>>> a = array.array('i', [1,2,3,4,5])
>>> m = memoryview(a)
>>> len(m)
```

(次のページに続く)

(前のページからの続き)

```
5
>>> m.nbytes
20
>>> y = m[::2]
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12
```

多次元配列:

```
>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in range(12)])
>>> x = memoryview(buf)
>>> y = x.cast('d', shape=[3,4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5, 15.0, 16.5]]
>>> len(y)
3
>>> y.nbytes
96
```

バージョン 3.3 で追加.

readonly

メモリが読み出し専用かどうかを示す真偽値です。

format

ビューの中の各要素に対する (*struct* モジュールスタイルの) フォーマットを含む文字列。memoryview は、任意のフォーマット文字列を使ってエクスポーターから作成することができます。しかし、いくつかのメソッド (例えば *tolist()*) はネイティブの単一要素フォーマットに制限されます。

バージョン 3.3 で変更: フォーマット `'B'` は struct モジュール構文で扱われるようになりました。これは `memoryview(b'abc')[0] == b'abc'[0] == 97` ということを意味します。

itemsize

memoryview の各要素のバイト単位の大きさ:

```
>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001, 32002]))
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True
```

ndim
メモリが表す多次元配列が何次元かを示す整数です。

shape
メモリが表している N 次元配列の形状を表す、長さ *ndim* の整数のタプルです。

バージョン 3.3 で変更: ndim = 0 の場合は None ではなく空のタプルとなるよう変更されました。

strides
配列のそれぞれの次元に対して、それぞれの要素にアクセスするのに必要なバイト数を表す、長さ *ndim* の整数のタプルです。

バージョン 3.3 で変更: ndim = 0 の場合は None ではなく空のタプルとなるよう変更されました。

suboffsets
PIL スタイルの配列の内部で利用している値。この値はただの情報として公開されています。

c_contiguous
メモリーが C 形式の順序で連続しているかどうかを示す真偽値（参考: *contiguous* ）。

バージョン 3.3 で追加.

f_contiguous
メモリーが Fortran 形式の順序で連続しているかどうかを示す真偽値（参考: *contiguous* ）。

バージョン 3.3 で追加.

contiguous
メモリーが連続しているかどうかを示す真偽値（参考: *contiguous* ）。

バージョン 3.3 で追加.

4.9 set（集合）型 --- set, frozenset

set オブジェクトは、固有の *hashable* オブジェクトの順序なしコレクションです。通常の用途には、帰属テスト、シーケンスからの重複除去、積集合、和集合、差集合、対称差 (排他的論理和) のような数学的演算の計算が含まれます。(他のコンテナについては組み込みの *dict*, *list*, *tuple* クラスや *collections* モジュールを参照してください。)

集合は、他のコレクションと同様、`x in set`, `len(set)`, `for x in set` をサポートします。コレクションには順序がないので、集合は挿入の順序や要素の位置を記録しません。従って、集合はインデクシング、スライシング、その他のシーケンス的な振舞いをサポートしません。

set および *frozenset* という、2 つの組み込みの集合型があります。*set* はミュータブルで、`add()` や `remove()` のようなメソッドを使って内容を変更できます。ミュータブルなため、ハッシュ値を持たず、また辞書のキーや他の集合の要素として用いることができません。一方、*frozenset* 型はイミュータブルで、**ハッシュ可能** です。作成後に内容を改変できないため、辞書のキーや他の集合の要素として用いることができます。

空でない *set* (*frozenset* ではない) は、*set* コンストラクタに加え、要素を波括弧中にカンマで区切って列挙することでも生成できます。例: `{'jack', 'sjoerd'}`。

どちらのクラスのコンストラクタも同様に働きます:

class set([*iterable*])
class frozenset([*iterable*])

iterable から要素を取り込んだ、新しい *set* もしくは *frozenset* オブジェクトを返します。集合の要素は **ハッシュ可能** なものでなくてはなりません。集合の集合を表現するためには、内側の集合は *frozenset* オブジェクトでなくてはなりません。*iterable* が指定されない場合、新しい空の集合が返されます。

集合はいくつかの方法で生成できます:

- 波括弧内にカンマ区切りで要素を列挙する: `{'jack', 'sjoerd'}`
- 集合内包表記を使う: `{c for c in 'abracadabra' if c not in 'abc'}`
- 型コンストラクタを使う: `set()`, `set('foobar')`, `set(['a', 'b', 'foo'])`

set および *frozenset* のインスタンスは以下の操作を提供します:

len(s)
集合 *s* の要素数 (*s* の濃度) を返します。

x in s
x が *s* のメンバーに含まれるか判定します。

x not in s
x が *s* のメンバーに含まれていないことを判定します。

isdisjoint(*other*)
集合が *other* と共通の要素を持たないとき、`True` を返します。集合はそれらの積集合が空集合となるときのみ、互いに素 (disjoint) となります。

issubset(*other*)
set <= other
set の全ての要素が *other* に含まれるか判定します。

set < other
set が *other* の真部分集合であるかを判定します。つまり、`set <= other and set != other` と等価です。

issuperset(*other*)
set >= other
other の全ての要素が set に含まれるか判定します。

set > other
set が *other* の真上位集合であるかを判定します。つまり、`set >= other and set != other` と等価です。

union(**others*)
set | other | ...
set と全ての other の要素からなる新しい集合を返します。

intersection(**others*)
set & other & ...
set と全ての other に共通する要素を持つ、新しい集合を返します。

difference(**others*)
set - other - ...
set に含まれて、かつ、全ての other に含まれない要素を持つ、新しい集合を返します。

symmetric_difference(*other*)
set ^ other
set と *other* のいずれか一方だけに含まれる要素を持つ新しい集合を返します。

copy()
集合の浅いコピーを返します。

なお、演算子でない版の *union()*, *intersection()*, *difference()*, *symmetric_difference()*, *issubset()*, *issuperset()* メソッドは、任意のイテラブルを引数として受け付けます。対して、演算子を使う版では、引数は集合でなくてはなりません。これは、`set('abc') & 'cbs'` のような誤りがちな構文を予防し、より読みやすい `set('abc').intersection('cbs')` を支持します。

set と *frozenset* のどちらも、集合同士の比較をサポートします。二つの集合は、それぞれの集合の要素全てが他方にも含まれている (互いに他方の部分集合である) とき、かつそのときに限り等しいです。一方の集合が他方の集合の真部分集合である (部分集合であるが等しくない) とき、かつそのときに限り一方の集合は他方の集合より小さいです。一方の集合が他方の集合の真上位集合である (上位集合であるが等しくない) とき、かつそのときに限り一方の集合は他方の集合より大きいです。

set のインスタンスは、*frozenset* のインスタンスと、要素に基づいて比較されます。例えば、`set('abc') == frozenset('abc')` や `set('abc') in set([frozenset('abc')])` は `True` を返します。

部分集合と等価性の比較は全順序付けを行う関数へと一般化することはできません。例えば、互いに素である二つの非空集合は、等しくなく、他方の部分集合でもありませんから、以下の **すべて** に `False` を返します: `a<b`, `a==b`, そして `a>b`.

集合は半順序（部分集合関係）しか定義しないので、集合のリストにおける *list.sort()* メソッドの出力は未定義です。

集合の要素は、辞書のキーのように、**ハッシュ可能** でなければなりません。

set インスタンスと *frozenset* インスタンスを取り混ぜての二項演算は、第一被演算子の型を返します。例えば: `frozenset('ab') | set('bc')` は *frozenset* インスタンスを返します。

以下の表に挙げる演算は *set* に適用されますが、*frozenset* のイミュータブルなインスタンスには適用されません:

update(**others*)

set |= other | ...
全ての other の要素を追加し、set を更新します。

intersection_update(*others)
set &= other & ...
元の set と全ての other に共通する要素だけを残して set を更新します。

difference_update(*others)
set -= other | ...
other に含まれる要素を取り除き、set を更新します。

symmetric_difference_update(*other*)
set ^= other
どちらかにのみ含まれて、共通には持たない要素のみで set を更新します。

add(*elem*)
要素 *elem* を set に追加します。

remove(*elem*)
要素 *elem* を set から取り除きます。*elem* が set に含まれていなければ *KeyError* を送出します。

discard(*elem*)
要素 *elem* が set に含まれていれば、取り除きます。

pop()
s から任意の要素を取り除き、それを返します。集合が空の場合、*KeyError* を送出します

clear()
set の全ての要素を取り除きます。

なお、演算子でない版の *update()*, *intersection_update()*, *difference_update()*, および *symmetric_difference_update()* メソッドは、任意のイテラブルを引数として受け付けます。

__contains__(), *remove()*, *discard()* メソッドの引数 *elem* は集合かもしれないことに注意してください。その集合と等価な *frozenset* の検索をサポートするために、*elem* から一時的な frozenset を作成します。

4.10 マッピング型 --- dict

マッピング オブジェクトは、**ハッシュ可能** な値を任意のオブジェクトに対応付けます。マッピングはミュータブルなオブジェクトです。現在、標準のマッピング型は辞書 (*dictionary*) だけです。(他のコンテナについては組み込みの *list*, *set*, および *tuple* クラスと、*collections* モジュールを参照してください。)

辞書のキーは **ほぼ** 任意の値です。**ハッシュ可能** でない値、つまり、リストや辞書その他のミュータブルな型 (オブジェクトの同一性ではなく値で比較されるもの) はキーとして使用できません。キーとして使われる数値型は通常の数値比較のルールに従います: もしふたつの数値が (例えば 1 と 1.0 のように) 等しければ、同じ辞書の項目として互換的に使用できます。(ただし、コンピュータは浮動小数点数を近似値として保管するので、辞書型のキーとして使用するのはたいてい賢くありません。)

辞書は `key: value` 対のカンマ区切りのリストを波括弧でくくることで作成できます。例えば: `{'jack': 4098, 'sjoerd': 4127}` あるいは `{4098: 'jack', 4127: 'sjoerd'}` 。あるいは、*dict* コンストラクタでも作成できます。

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

オプションの位置引数と空の可能性もあるキーワード引数の集合により初期化された新しい辞書を返します。

辞書はいくつかの方法で生成できます:

- 波括弧内にカンマ区切りで `key: value` 対を列挙する: `{'jack': 4098, 'sjoerd': 4127}` あるいは `{4098: 'jack', 4127: 'sjoerd'}`
- 辞書内包表記を使う: `{}`, `{x: x ** 2 for x in range(10)}`
- 型コンストラクタを使う: `dict()`, `dict([('foo', 100), ('bar', 200)])`, `dict(foo=100, bar=200)`

位置引数が何も与えられなかった場合、空の辞書が作成されます。位置引数が与えられ、それがマッピングオブジェクトだった場合、そのマッピングオブジェクトと同じキーと値のペアを持つ辞書が作成されます。それ以外の場合、位置引数は *iterable* オブジェクトでなければなりません。iterable のそれぞれの要素自身は、ちょうど 2 個のオブジェクトを持つイテラブルでなければなりません。それぞれの要素の最初のオブジェクトは新しい辞書のキーになり、2 番目のオブジェクトはそれに対応する値になります。同一のキーが 2 回以上現れた場合は、そのキーの最後の値が新しい辞書での対応する値になります。

キーワード引数が与えられた場合、キーワード引数とその値が位置引数から作られた辞書に追加されます。既に存在しているキーが追加された場合、キーワード引数の値は位置引数の値を置き換えます。

例を出すと、次の例は全て `{"one": 1, "two": 2, "three": 3}` に等しい辞書を返します:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

最初の例のようにキーワード引数を与える方法では、キーは有効な Python の識別子でなければなりません。それ以外の方法では、辞書のキーとして有効などんなキーでも使えます。

以下は辞書型がサポートする操作です (それゆえ、カスタムのマップ型もこれらの操作をサポートするべきです):

list(d)
辞書 *d* で使われている全てのキーのリストを返します。

len(d)
辞書 *d* の項目数を返します。

d[key]
d のキー *key* の項目を返します。マップに *key* が存在しなければ、*KeyError* を送出します。

辞書のサブクラスが `__missing__()` メソッドを定義していて、*key* が存在しない場合、`d[key]` 演算はこのメソッドをキー *key* を引数として呼び出します。`d[key]` 演算は、`__missing__(key)` の呼び出しによって返された値をそのまま返すか、送出されたものをそのまま送出します。他の演算やメソッドは `__missing__()` を呼び出しません。`__missing__()` が定義されていない場合、*KeyError* が送出されます。`__missing__()` はメソッドでなければならず、インスタンス変数であってはなりません:

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

ここでお見せした例は *collections.Counter* 実装の一部です。これとは違った `__missing__` が *collections.defaultdict* で使われています。

d[key] = value
`d[key]` に *value* を設定します。

del d[key]
d から `d[key]` を削除します。マップに *key* が存在しなければ、*KeyError* を送出します。

key in d
d がキー *key* を持っていれば `True` を、そうでなければ、`False` を返します。

key not in d
`not key in d` と等価です。

iter(d)
辞書のキーに渡るイテレータを返します。これは `iter(d.keys())` へのショートカットです。

clear()
辞書の全ての項目を消去します。

copy()
辞書の浅いコピーを返します。

classmethod fromkeys(*iterable*[, *value*]**)**
iterable からキーを取り、値を *value* に設定した、新しい辞書を作成します。

fromkeys() は新しい辞書を返すクラスメソッドです。*value* はデフォルトで `None` となります。

作られる辞書内のすべての値が同一のインスタンスを指すことになるため、*value* にミュータブルなオブジェクト (例えば空のリスト) を指定しても通常意味はありません。別々の値を指すようにしたい場合は、代わりに 辞書内包表記 を使用してください。

get(*key*[, *default*])
key が辞書にあれば *key* に対する値を、そうでなければ *default* を返します。*default* が与えられなかった場合、デフォルトでは None となります。そのため、このメソッドは *KeyError* を送出することはありません。

items()
辞書の項目 ((key, value) 対) の新しいビューを返します。**ビューオブジェクトのドキュメント** を参照してください。

keys()
辞書のキーの新しいビューを返します。**ビューオブジェクトのドキュメント** を参照してください。

pop(*key*[, *default*])
key が辞書に存在すればその値を辞書から消去して返し、そうでなければ *default* を返します。*default* が与えらず、かつ *key* が辞書に存在しなければ *KeyError* を送出します。

popitem()
任意の (key, value) 対を辞書から消去して返します。対は LIFO (後入れ、先出し) の順序で返却されます。

集合のアルゴリズムで使われるのと同じように、*popitem()* は辞書に繰り返し適用して消去するのに便利です。辞書が空であれば、*popitem()* の呼び出しは *KeyError* を送出します。

バージョン 3.7 で変更: LIFO 順序が保証されるようになりました。以前のバージョンでは、*popitem()* は任意の key/value 対を返していました。

reversed(d)
辞書のキーに渡る逆イテレータを返します。これは reversed(d.keys()) へのショートカットです。

バージョン 3.8 で追加.

setdefault(*key*[, *default*])
もし、*key* が辞書に存在すれば、その値を返します。そうでなければ、値を *default* として *key* を挿入し、*default* を返します。*default* のデフォルトは None です。

update([*other*])
辞書の内容を *other* のキーと値で更新します。既存のキーは上書きされます。返り値は None です。

update() は、他の辞書オブジェクトでもキー/値の対のイテラブル (タプル、もしくは、長さが 2 のイテラブル) でも、どちらでも受け付けます。キーワード引数が指定されれば、そのキー/値の対で辞書を更新します: d.update(red=1, blue=2)。

values()
辞書の値の新しいビューを返します。**ビューオブジェクトのドキュメント** を参照してください。

dict.values() で得られた 2 つのビューの等しさを比較すると、必ず False が返ります。dict.values() どうしを比較したときも同様です:

```
>>> d = {'a': 1}
>>> d.values() == d.values()
False
```

複数の辞書は、(順序に関係なく) 同じ (key, value) の対を持つ場合に、そしてその場合にのみ等しくなります。順序比較 ('<', '<=', '>=', '>') は *TypeError* を送出します。

辞書は挿入順序を保存するようになりました。キーの更新は順序には影響が無いことに注意してください。いったん削除されてから再度追加されたキーは末尾に挿入されます。:

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(d)
['one', 'two', 'three', 'four']
>>> list(d.values())
[1, 2, 3, 4]
>>> d["one"] = 42
>>> d
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
>>> del d["two"]
>>> d["two"] = None
>>> d
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

バージョン 3.7 で変更: 辞書の順序が挿入順序であることが保証されるようになりました。この振る舞いは CPython 3.6 の実装詳細でした。

辞書と辞書のビューは reversed() で順序を逆にすることができます:

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(reversed(d))
['four', 'three', 'two', 'one']
>>> list(reversed(d.values()))
[4, 3, 2, 1]
>>> list(reversed(d.items()))
[('four', 4), ('three', 3), ('two', 2), ('one', 1)]
```

バージョン 3.8 で変更: 辞書がリバース可能になりました。

参考:

dict の読み出し専用ビューを作るために *types.MappingProxyType* を使うことができます。

4.10.1 辞書ビューオブジェクト

dict.keys(), *dict.values()*, *dict.items()* によって返されるオブジェクトは、**ビューオブジェクト** です。これらは、辞書の項目の動的なビューを提供し、辞書が変更された時、ビューはその変更を反映します。

辞書ビューは、イテレートすることで対応するデータを yield できます。また、帰属判定をサポートします:

`len(dictview)`
: 辞書の項目数を返します。

`iter(dictview)`
: 辞書のキー、値、または (`(key, value)` のタプルとして表される) 項目に渡るイテレータを返します。

  キーと値は挿入順序で反復されます。これにより、`(value, key)` の対の列を `pairs = zip(d.values(), d.keys())` のように *zip()* で作成できます。同じリストを作成する他の方法は、`pairs = [(v, k) for (k, v) in d.items()]` です。

  辞書の項目の追加や削除中にビューをイテレートすると、*RuntimeError* を送出したり、すべての項目に渡ってイテレートできなかったりします。

  バージョン 3.7 で変更: 辞書の順序が挿入順序であると保証されるようになりました。

`x in dictview`
: x が元の辞書のキー、値、または項目 (項目の場合、x は `(key, value)` タプルです) にあるとき `True` を返します。

`reversed(dictview)`
: 辞書のキーもしくは値、項目の順序を逆にしたイテレーターを返します。戻り値のビューは、挿入された順とは逆の順でイテレートします。

  バージョン 3.8 で変更: 辞書のビューがリバース可能になりました。

キーのビューは、項目が一意的でハッシュ可能であるという点で、集合に似ています。すべての値がハッシュ可能なら、`(key, value)` 対も一意的でハッシュ可能なので、要素のビューも集合に似ています。(値のビューは、要素が一般に一意的でないことから、集合に似ているとは考えられません。) 集合に似ているビューに対して、抽象基底クラス *collections.abc.Set* で定義されている全ての演算 (例えば、==、<、^) が利用できます。

辞書ビューの使用法の例:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504
```

(次のページに続く)

(前のページからの続き)

```
>>> # keys and values are iterated over in the same order (insertion order)
>>> list(keys)
['eggs', 'sausage', 'bacon', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['bacon', 'spam']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'}
{'juice', 'sausage', 'bacon', 'spam'}
```

4.11 コンテキストマネージャ型

Python の `with` 文は、コンテキストマネージャによって定義される実行時コンテキストの概念をサポートします。これは、文の本体が実行される前に進入し文の終わりで脱出する実行時コンテキストを、ユーザ定義クラスが定義できるようにする一対のメソッドで実装されます:

`contextmanager.__enter__()`

実行時コンテキストに入り、このオブジェクトまたは他の実行時コンテキストに関連したオブジェクトを返します。このメソッドが返す値はこのコンテキストマネージャを使う `with` 文の `as` 節の識別子に束縛されます。

自分自身を返すコンテキストマネージャの例として **ファイルオブジェクト** があります。ファイルオブジェクトは ___enter___() から自分自身を返し、*open()* が `with` 文のコンテキスト式として使われるようにします。

関連オブジェクトを返すコンテキストマネージャの例としては *decimal.localcontext()* が返すものがあります。このマネージャはアクティブな 10 進数コンテキストをオリジナルのコンテキストのコピーにセットしてそのコピーを返します。こうすることで, `with` 文の本体の内部で、外側のコードに影響を与えずに、10 進数コンテキストを変更できます。

`contextmanager.__exit__`(*exc_type*, *exc_val*, *exc_tb*)

実行時コンテキストから抜け、(発生していた場合) 例外を抑制するかどうかを示すブール値フラグを返します。`with` 文の本体の実行中に例外が発生した場合、引数にはその例外の型と値とトレースバック情報を渡します。そうでない場合、引数は全て `None` となります。

このメソッドから真値が返されると `with` 文は例外の発生を抑え、`with` 文の直後の文に実行を続けます。そうでなければ、このメソッドの実行を終えると例外の伝播が続きます。このメソッドの実行中に起きた例外は `with` 文の本体の実行中に起こった例外を置き換えてしまいます。

渡された例外を明示的に再送出すべきではありません。その代わりに、このメソッドが偽の値を返すことでメソッドの正常終了と送出された例外を抑制しないことを伝えるべきです。このようにすればコンテキストマネージャは *__exit__()* メソッド自体が失敗したのかどうかを簡単に見分けることができます。

Python は、易しいスレッド同期、ファイルなどのオブジェクトの即時クローズ、アクティブな小数算術コンテキストの単純な操作をサポートするために、いくつかのコンテキストマネージャを用意しています。各型はコンテキスト管理プロトコルを実装しているという以上の特別の取り扱いを受けるわけではありません。例については *contextlib* モジュールを参照してください。

Python の **ジェネレータ** と *contextlib.contextmanager* **デコレータ** はこのプロトコルの簡便な実装方法を提供します。ジェネレータ関数を *contextlib.contextmanager* デコレータでデコレートすると、デコレートされないジェネレータ関数が作成するイテレータの代わりに、必要な `__enter__()` および `__exit__()` メソッドを実装したコンテキストマネージャを返すようになります。

これらのメソッドのために Python/C API の中の Python オブジェクトの型構造体に特別なスロットが作られたわけではないことに注意してください。これらのメソッドを定義したい拡張型はこれらを通常の Python からアクセスできるメソッドとして提供しなければなりません。実行時コンテキストを準備するオーバーヘッドに比べたら、一回のクラス辞書の探索のオーバーヘッドは無視できます。

4.12 その他の組み込み型

インタプリタは、その他いくつかの種類のオブジェクトをサポートしています。これらのほとんどは 1 つまたは 2 つの演算だけをサポートしています。

4.12.1 モジュール (module)

モジュールに対する唯一の特殊な演算は属性アクセス: `m.name` です。ここで *m* はモジュールで、*name* は *m* のシンボルテーブル上に定義された名前にアクセスします。モジュール属性に代入することもできます。(なお、`import` 文は、厳密にいえば、モジュールオブジェクトに対する演算ではありません; `import foo` は *foo* と名づけられたモジュールオブジェクトの存在を必要とはせず、*foo* と名づけられたモジュールの (外部の) **定義** を必要とします。)

全てのモジュールにある特殊属性が *__dict__* です。これはモジュールのシンボルテーブルを含む辞書です。この辞書を書き換えると実際にモジュールのシンボルテーブルを変更することができますが、*__dict__* 属性を直接代入することはできません (`m.__dict__['a'] = 1` と書いて `m.a` を `1` に定義することはできますが、`m.__dict__ = {}` と書くことはできません)。*__dict__* を直接書き換えることは推奨されません。

インタプリタ内に組み込まれたモジュールは、`<module 'sys' (built-in)>` のように書かれます。ファイルから読み出された場合、`<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>` と書かれます。

4.12.2 クラスおよびクラスインスタンス

これらについては objects および class を参照してください。

4.12.3 関数

関数オブジェクトは関数定義によって生成されます。関数オブジェクトに対する唯一の操作は、それを呼び出すことです: `func(argument-list)` 。

関数オブジェクトには実際には二種類あります: 組み込み関数とユーザ定義関数です。どちらも同じ操作 (関数の呼び出し) をサポートしますが、実装は異なるので、オブジェクトの型も異なります。

詳細は、function を参照してください。

4.12.4 メソッド

メソッドは属性表記を使って呼び出される関数です。メソッドには二種類あります: (リストの `append()` のような) 組み込みメソッドと、クラスインスタンスのメソッドです。組み込みメソッドは、それをサポートする型と一緒に記述されています。

インスタンスを通してメソッド (クラスの名前空間内で定義された関数) にアクセスすると、特殊なオブジェクトが得られます。それは束縛メソッド (*bound method*) オブジェクトで、インスタンスメソッド (*instance method*) とも呼ばれます。呼び出された時、引数リストに `self` 引数が追加されます。束縛メソッドには 2 つの特殊読み出し専用属性があります。`m.__self__` はそのメソッドが操作するオブジェクトで、`m.__func__` はそのメソッドを実装している関数です。`m(arg-1, arg-2, ..., arg-n)` の呼び出しは、`m.__func__(m.__self__, arg-1, arg-2, ..., arg-n)` の呼び出しと完全に等価です。

関数オブジェクトと同様に、メソッドオブジェクトは任意の属性の取得をサポートしています。しかし、メソッド属性は実際には下層の関数オブジェクト (`meth.__func__`) に記憶されているので、バインドされるメソッドにメソッド属性を設定することは許されていません。メソッドに属性を設定しようとすると *`AttributeError`* が送出されます。メソッドの属性を設定するためには、次のようにその下層の関数オブジェクトに明示的に設定する必要があります:

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method'  # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'method' object has no attribute 'whoami'
>>> c.method.__func__.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

詳細は、types を参照してください。

4.12.5 コードオブジェクト

コードオブジェクトは、関数本体のような ” 擬似コンパイルされた” Python の実行可能コードを表すために実装系によって使われます。コードオブジェクトはグローバルな実行環境への参照を持たない点で関数オブジェクトとは異なります。コードオブジェクトは組み込み関数 *compile()* によって返され、また関数オブジェクトの `__code__` 属性として取り出せます。*code* モジュールも参照してください。

`__code__` へのアクセスは `object.__getattr__` に `obj` と `"__code__"` を渡して行いますが、監査イベント を送出します。

コードオブジェクトは、組み込み関数 *exec()* や *eval()* に (ソース文字列の代わりに) 渡すことで、実行や評価できます。

詳細は、types を参照してください。

4.12.6 型オブジェクト

型オブジェクトは様々なオブジェクト型を表します。オブジェクトの型は組み込み関数 *type()* でアクセスされます。型オブジェクトには特有の操作はありません。標準モジュール *types* には全ての組み込み型名が定義されています。

型はこのように書き表されます: `<class 'int'>` 。

4.12.7 ヌルオブジェクト

このオブジェクトは明示的に値を返さない関数によって返されます。このオブジェクトには特有の操作はありません。ヌルオブジェクトは一つだけで、`None` (組み込み名) と名づけられています。`type(None)()` は同じシングルトンを作成します。

`None` と書き表されます。

4.12.8 Ellipsis オブジェクト

このオブジェクトは一般にスライシングによって使われます (slicings を参照してください)。特殊な演算は何もサポートしていません。Ellipsis オブジェクトは一つだけで、その名前は *Ellipsis* (組み込み名) です。`type(Ellipsis)()` は単一の *Ellipsis* を作成します。

`Ellipsis` または `...` と書き表されます。

4.12.9 NotImplemented オブジェクト

このオブジェクトは、対応していない型に対して比較演算や二項演算が求められたとき、それらの演算から返されます。詳細は comparisons を参照してください。`NotImplemented` オブジェクトは一つだけです。`type(NotImplemented)()` はこの単一のインスタンスを作成します。

`NotImplemented` と書き表されます。

4.12.10 ブール値

ブール値は二つの定数オブジェクト `False` および `True` です。これらは真理値を表すのに使われます (ただし他の値も偽や真とみなされます)。数値処理のコンテキスト (例えば算術演算子の引数として使われた場合) では、これらはそれぞれ 0 および 1 と同様に振舞います。任意の値に対して、真理値と解釈できる場合、組み込み関数 *bool()* は値をブール値に変換するのに使われます (上述の **真理値判定** の節を参照してください)。

それぞれ `False` および `True` と書き表されます。

4.12.11 内部オブジェクト

この情報は types を参照してください。スタックフレームオブジェクト、トレースバックオブジェクト、スライスオブジェクトについて記述されています。

4.13 特殊属性

実装は、いくつかのオブジェクト型に対して、適切な場合には特殊な読み出し専用の属性を追加します。そのうちいくつかは *dir()* 組込み関数で報告されません。

object.__dict__
: オブジェクトの (書き込み可能な) 属性を保存するために使われる辞書またはその他のマッピングオブジェクトです。

instance.__class__
: クラスインスタンスが属しているクラスです。

class.__bases__
: クラスオブジェクトの基底クラスのタプルです。

definition.__name__
: クラス、関数、メソッド、デスクリプタ、ジェネレータインスタンスの名前です。

definition.__qualname__
: クラス、関数、メソッド、デスクリプタ、ジェネレータインスタンスの **修飾名** です。

  バージョン 3.3 で追加.

class.__mro__
: この属性はメソッドの解決時に基底クラスを探索するときに考慮されるクラスのタプルです。

class.mro()
: このメソッドは、メタクラスによって、そのインスタンスのメソッド解決の順序をカスタマイズするために、上書きされるかも知れません。このメソッドはクラスのインスタンス化時に呼ばれ、その結果は *__mro__* に格納されます。

class.__subclasses__()
: それぞれのクラスは、それ自身の直接のサブクラスへの弱参照を保持します。このメソッドはそれらの参照のうち、生存しているもののリストを返します。例:

```
>>> int.__subclasses__()
[<class 'bool'>]
```

4.14 Integer string conversion length limitation

CPython has a global limit for converting between *int* and *str* to mitigate denial of service attacks. This limit *only* applies to decimal or other non-power-of-two number bases. Hexadecimal, octal, and binary conversions are unlimited. The limit can be configured.

The *int* type in CPython is an arbitrary length number stored in binary form (commonly known as a "bignum"). There exists no algorithm that can convert a string to a binary integer or a binary integer to a string in linear time, *unless* the base is a power of 2. Even the best known algorithms for base 10 have sub-quadratic complexity. Converting a large value such as `int('1' * 500_000)` can take over a second on a fast CPU.

Limiting conversion size offers a practical way to avoid CVE-2020-10735.

The limit is applied to the number of digit characters in the input or output string when a non-linear conversion algorithm would be involved. Underscores and the sign are not counted towards the limit.

When an operation would exceed the limit, a *ValueError* is raised:

```
>>> import sys
>>> sys.set_int_max_str_digits(4300)  # Illustrative, this is the default.
>>> _ = int('2' * 5432)
Traceback (most recent call last):
...
ValueError: Exceeds the limit (4300) for integer string conversion: value has 5432 digits; use␣
↪sys.set_int_max_str_digits() to increase the limit.
>>> i = int('2' * 4300)
>>> len(str(i))
4300
>>> i_squared = i*i
>>> len(str(i_squared))
Traceback (most recent call last):
...
```

(次のページに続く)

(前のページからの続き)

```
ValueError: Exceeds the limit (4300) for integer string conversion: value has 8599 digits; use␣
↪sys.set_int_max_str_digits() to increase the limit.
>>> len(hex(i_squared))
7144
>>> assert int(hex(i_squared), base=16) == i*i  # Hexadecimal is unlimited.
```

The default limit is 4300 digits as provided in *sys.int_info.default_max_str_digits*. The lowest limit that can be configured is 640 digits as provided in *sys.int_info.str_digits_check_threshold*.

Verification:

```
>>> import sys
>>> assert sys.int_info.default_max_str_digits == 4300, sys.int_info
>>> assert sys.int_info.str_digits_check_threshold == 640, sys.int_info
>>> msg = int('578966293710682886880994035146873798396722250538762761564'
...           '9252925514383915483333812743580549779436104706260696366600'
...           '571186405732').to_bytes(53, 'big')
...
```

バージョン 3.8.14 で追加.

4.14.1 Affected APIs

The limitation only applies to potentially slow conversions between *int* and *str* or *bytes*:

- `int(string)` with default base 10.
- `int(string, base)` for all bases that are not a power of 2.
- `str(integer)`.
- `repr(integer)`.
- any other string conversion to base 10, for example `f"{integer}"`, `"{}".format(integer)`, or `b"%d" % integer`.

The limitations do not apply to functions with a linear algorithm:

- `int(string, base)` with base 2, 4, 8, 16, or 32.
- *int.from_bytes()* and *int.to_bytes()*.
- *hex()*, *oct()*, *bin()*.
- 書式指定ミニ言語仕様 for hex, octal, and binary numbers.
- *str* to *float*.
- *str* to *decimal.Decimal*.

4.14.2 Configuring the limit

Before Python starts up you can use an environment variable or an interpreter command line flag to configure the limit:

- `PYTHONINTMAXSTRDIGITS`, e.g. `PYTHONINTMAXSTRDIGITS=640 python3` to set the limit to 640 or `PYTHONINTMAXSTRDIGITS=0 python3` to disable the limitation.
- `-X int_max_str_digits`, e.g. `python3 -X int_max_str_digits=640`
- `sys.flags.int_max_str_digits` contains the value of `PYTHONINTMAXSTRDIGITS` or `-X int_max_str_digits`. If both the env var and the `-X` option are set, the `-X` option takes precedence. A value of *-1* indicates that both were unset, thus a value of `sys.int_info.default_max_str_digits` was used during initialization.

From code, you can inspect the current limit and set a new one using these *sys* APIs:

- *sys.get_int_max_str_digits()* and *sys.set_int_max_str_digits()* are a getter and setter for the interpreter-wide limit. Subinterpreters have their own limit.

Information about the default and minimum can be found in *sys.int_info*:

- *sys.int_info.default_max_str_digits* is the compiled-in default limit.
- *sys.int_info.str_digits_check_threshold* is the lowest accepted value for the limit (other than 0 which disables it).

バージョン 3.8.14 で追加.

ご用心: Setting a low limit *can* lead to problems. While rare, code exists that contains integer constants in decimal in their source that exceed the minimum threshold. A consequence of setting the limit is that Python source code containing decimal integer literals longer than the limit will encounter an error during parsing, usually at startup time or import time or even at installation time - anytime an up to date `.pyc` does not already exist for the code. A workaround for source that contains such large constants is to convert them to `0x` hexadecimal form as it has no limit.

Test your application thoroughly if you use a low limit. Ensure your tests run with the limit set early via the environment or flag so that it applies during startup and even during any installation step that may invoke Python to precompile `.py` sources to `.pyc` files.

4.14.3 Recommended configuration

The default `sys.int_info.default_max_str_digits` is expected to be reasonable for most applications. If your application requires a different limit, set it from your main entry point using Python version agnostic code as these APIs were added in security patch releases in versions before 3.11.

Example:

```
>>> import sys
>>> if hasattr(sys, "set_int_max_str_digits"):
...     upper_bound = 68000
...     lower_bound = 4004
...     current_limit = sys.get_int_max_str_digits()
...     if current_limit == 0 or current_limit > upper_bound:
...         sys.set_int_max_str_digits(upper_bound)
...     elif current_limit < lower_bound:
...         sys.set_int_max_str_digits(lower_bound)
```

If you need to disable it entirely, set it to `0`.

脚注

第

FIVE

組み込み例外

Python において、すべての例外は *BaseException* から派生したクラスのインスタンスでなければなりません。特定のクラスを言及する except 節を伴う try 文において、その節はそのクラスから派生した例外クラスも処理しますが、そのクラスの派生 元 の例外クラスは処理しません。サブクラス化の関係にない 2 つの例外クラスは、それらが同じ名前だった場合でも等しくなりえません。

以下に挙げる組み込み例外は、インタプリタや組み込み関数によって生成されます。特に注記しないかぎり、これらはエラーの詳しい原因を示す ” 関連値 (associated value)” を持ちます。この値は、複数の情報 (エラーコードや、そのコードを説明する文字列など) の文字列かタプルです。関連値は通常、例外クラスのコンストラクタに引数として渡されます。

ユーザによるコードも組み込み例外を送出できます。これを使って、例外ハンドラをテストしたり、インタプリタが同じ例外を送出する状況と ” ちょうど同じような” エラー条件であることを報告したりできます。しかし、ユーザのコードが適切でないエラーを送出するのを妨げる方法はないので注意してください。

組み込み例外クラスは新たな例外を定義するためにサブクラス化することができます。新しい例外は、*Exception* クラスかそのサブクラスの一つから派生することをお勧めします。*BaseException* からは派生しないで下さい。例外を定義する上での詳しい情報は、Python チュートリアルの tut-userexceptions の項目にあります。

except または finally 節内で例外を送出 (または再送出) するとき、__context__ は自動的に最後に捕まった例外に設定されます。新しい例外が処理されなければ、最終的に表示されるトレースバックは先に起きた例外も最後の例外も含みます。

(現在処理中の例外を raise を使って再送出するのではなく,) 新規に例外を送出する場合、raise と一緒に from を使うことで暗黙の例外コンテキストを明示的に捕捉することができます:

```
raise new_exc from original_exc
```

from に続く式は例外か None でなくてはなりません。式は送出される例外の __cause__ として設定されます。__cause__ を設定することは、__suppress_context__ 属性を暗黙的に True に設定することにもなるので、raise new_exc from None を使うことで効率的に古い例外を新しいもので置き換えて表示する (例えば、*KeyError* を *AttributeError* に置き換え)、古い例外はデバッグ時の調査で使えるよう __context__ に残すことができます。

デフォルトの traceback 表示コードは、例外自体の traceback に加え、これらの連鎖された例外を表示します。__cause__ で明示的に連鎖させた例外は、存在するならば常に表示されます。__context__ で暗黙に連

鎖させた例外は、__cause__ が *None* かつ __suppress_context__ が false の場合にのみ表示されます。

いずれにせよ、連鎖された例外に続いて、その例外自体は常に表示されます。そのため、traceback の最終行には、常に送出された最後の例外が表示されます。

5.1 基底クラス

以下の例外は、主に他の例外の基底クラスとして使われます。

exception BaseException

全ての組み込み例外の基底クラスです。ユーザ定義の例外に直接継承されることは意図されていません (継承には *Exception* を使ってください)。このクラスのインスタンスに *str()* が呼ばれた場合、インスタンスへの引数の表現か、引数が無い場合には空文字列が返されます。

args

例外コンストラクタに与えられた引数のタプルです。組み込み例外は普通、エラーメッセージを与える一つの文字列だけを引数として呼ばれますが、中には (*OSError* など) いくつかの引数を必要とし、このタプルの要素に特別な意味を込めるものもあります。

with_traceback(*tb*)

このメソッドは *tb* を例外の新しいトレースバックとして設定し、例外オブジェクトを返します。これは通常次のような例外処理コードに使われます:

```
try:
    ...
except SomeException:
    tb = sys.exc_info()[2]
    raise OtherException(...).with_traceback(tb)
```

exception Exception

システム終了以外の全ての組み込み例外はこのクラスから派生しています。全てのユーザ定義例外もこのクラスから派生させるべきです。

exception ArithmeticError

算術上の様々なエラーに対して送出される組み込み例外 *OverflowError*, *ZeroDivisionError*, *FloatingPointError* の基底クラスです。

exception BufferError

バッファ に関連する操作が行えなかったときに送出されます。

exception LookupError

マッピングまたはシーケンスで使われたキーやインデクスが無効な場合に送出される例外 *IndexError* および *KeyError* の基底クラスです。*codecs.lookup()* によって直接送出されることもあります。

5.2 具象例外

以下の例外は、通常送出される例外です。

exception AssertionError
assert 文が失敗した場合に送出されます。

exception AttributeError
属性参照 (attribute-references を参照) や代入が失敗した場合に送出されます (オブジェクトが属性の参照や属性の代入をまったくサポートしていない場合には *TypeError* が送出されます)。

exception EOFError
input() が何もデータを読まずに end-of-file (EOF) に達した場合に送出されます。(注意: io.IOBase.read() と *io.IOBase.readline()* メソッドは、EOF に達すると空文字列を返します。)

exception FloatingPointError
現在は使われていません。

exception GeneratorExit
ジェネレータ や コルーチン が閉じられたときに送出されます。generator.close() と coroutine.close() を参照してください。この例外はエラーではなく技術的なものなので、*Exception* ではなく *BaseException* を直接継承しています。

exception ImportError
import 文でモジュールをロードしようとして問題が発生すると送出されます。from ... import の中の”from list” (訳注：... の部分) の名前が見つからないときにも送出されます。

コンストラクタのキーワード専用引数を使って name および path 属性を設定できます。設定された場合、インポートを試みられたモジュールの名前と、例外を引き起こしたファイルへのパスとを、それぞれ表します。

バージョン 3.3 で変更: name および path 属性が追加されました。

exception ModuleNotFoundError
ImportError のサブクラスで、import 文でモジュールが見つからない場合に送出されます。また、*sys.modules* に None が含まれる場合にも送出されます。

バージョン 3.6 で追加.

exception IndexError
シーケンスの添字が範囲外の場合に送出されます。(スライスのインデクスはシーケンスの範囲に収まるように暗黙のうちに調整されます; インデクスが整数でない場合、*TypeError* が送出されます。)

exception KeyError
マッピング (辞書) のキーが、既存のキーの集合内に見つからなかった場合に送出されます。

exception KeyboardInterrupt
ユーザが割り込みキー (通常は Control-C または Delete) を押した場合に送出されます。実行中、割り込みは定期的に監視されます。*Exception* を捕捉するコードに誤って捕捉されてインタプリタの終了が阻害されないように、この例外は *BaseException* を継承しています。

exception MemoryError

ある操作中にメモリが不足したが、その状況は (オブジェクトをいくつか消去することで) まだ復旧可能かもしれない場合に送出されます。この例外の関連値は、メモリ不足になった (内部) 操作の種類を示す文字列です。下層のメモリ管理アーキテクチャ (C の `malloc()` 関数) のために、インタプリタが現状から完璧に復旧できるとはかぎらないので注意してください。それでも、プログラムの暴走が原因の場合に備えて実行スタックのトレースバックを出力できるように、例外が送出されます。

exception NameError

ローカルまたはグローバルの名前が見つからなかった場合に送出されます。これは非修飾の (訳注: `spam.egg` ではなく単に `egg` のような) 名前のみに適用されます。関連値は見つからなかった名前を含むエラーメッセージです。

exception NotImplementedError

この例外は *RuntimeError* から派生しています。ユーザ定義の基底クラスにおいて、抽象メソッドが派生クラスでオーバライドされることを要求する場合にこの例外を送出しなくてはなりません。またはクラスは実装中であり本来の実装を追加する必要があることを示します。

注釈: 演算子やメソッドがサポートされていないことを示す目的でこの例外を使用するべきではありません。そのようなケースではオペレータやメソッドを未定義のままとするか、サブクラスの場合は *None* を設定してください。

注釈: `NotImplementedError` と `NotImplemented` は、似たような名前と目的を持っていますが、相互に変換できません。利用する際には、*NotImplemented* を参照してください。

---

exception OSError([*arg*]**)**

exception OSError(*errno*, *strerror*[, *filename*[, *winerror*[, *filename2*]]]**)**

この例外はシステム関数がシステム関連のエラーを返した場合に送出されます。例えば "file not found" や "disk full" のような I/O の失敗が発生したときです (引数の型が不正な場合や、他の偶発的なエラーは除きます)。

コンストラクタの 2 番目の形式は下記の対応する属性を設定します。指定されなかった場合属性はデフォルトで *None* です。後方互換性のために、引数が 3 つ渡された場合、*args* 属性は最初の 2 つの要素のみからなるタプルを持ちます。

コンストラクタは実際には、*OS exceptions* で述べられている *OSError* のサブクラスを返すことがよくあります。特定のサブクラスは最終的な *errno* 値によります。この挙動は *OSError* を直接またはエイリアスで構築し、サブクラス化時に継承されなかった場合にのみ発生します。

errno

C 変数 `errno` に由来する数値エラーコードです。

winerror

Windows において、ネイティブ Windows エラーコードを与えます。そして *errno* 属性は POSIX でいうネイティブエラーコードへのおよその翻訳です。

Windows では、*winerror* コンストラクタ引数が整数の場合 *errno* 属性は Windows エラーコードから決定され、*errno* 引数は無視されます。他のプラットフォームでは *winerror* 引数は無視され、*winerror* 属性は存在しません。

strerror
OS が提供するような、対応するエラーメッセージです。POSIX では `perror()` で、Windows では `FormatMessage()` で体裁化されます。

filename
filename2
ファイルシステムパスが 1 つ関与する例外 (例えば *open()* や *os.unlink()*) の場合、*filename* は関数に渡されたファイル名です。ファイルシステムパスが 2 つ関与する関数 (例えば *os.rename()*) の場合、*filename2* は関数に渡された 2 つ目のファイル名です。

バージョン 3.3 で変更: *EnvironmentError*, *IOError*, *WindowsError*, *socket.error*, *select.error*, `mmap.error` が *OSError* に統合されました。コンストラクタはサブクラスを返すかもしれません。

バージョン 3.4 で変更: *filename* 属性がファイルシステムのエンコーディングでエンコードやデコードされた名前から、関数に渡された元々のファイル名になりました。また、*filename2* コンストラクタ引数が追加されました。

exception OverflowError
算術演算の結果が表現できない大きな値になった場合に送出されます。これは整数では起こりません (むしろ *MemoryError* が送出されることになるでしょう)。しかし、歴史的な理由のため、要求された範囲の外の整数に対して OverflowError が送出されることがあります。C の浮動小数点演算の例外処理は標準化されていないので、ほとんどの浮動小数点演算もチェックされません。

exception RecursionError
この例外は *RuntimeError* を継承しています。インタープリタが最大再帰深度 (*sys.getrecursionlimit()* を参照) の超過を検出すると送出されます。

バージョン 3.5 で追加: 以前は *RuntimeError* をそのまま送出していました。

exception ReferenceError
weakref.proxy() によって生成された弱参照 (weak reference) プロキシを使って、ガーベジコレクションによって回収された後の参照対象オブジェクトの属性にアクセスした場合に送出されます。弱参照については *weakref* モジュールを参照してください。

exception RuntimeError
他のカテゴリに分類できないエラーが検出された場合に送出されます。関連値は、何が問題だったのかをより詳細に示す文字列です。

exception StopIteration
組込み関数 *next()* と *iterator* の *__next__()* メソッドによって、そのイテレータが生成するアイテムがこれ以上ないことを伝えるために送出されます。

この例外オブジェクトには一つの属性 `value` があり、例外を構成する際に引数として与えられ、デフォルトは *None* です。

generator や *coroutine* 関数が返るとき、新しい *StopIteration* インスタンスが送出されます。関数の返り値は例外のコンストラクタの value 引数として使われます。

ジェネレータのコードが直接的あるいは間接的に *StopIteration* を送出する場合は、*RuntimeError* に変換されます (*StopIteration* は変換後の例外の原因として保持されます)。

バージョン 3.3 で変更: value 属性とジェネレータ関数が値を返すためにそれを使う機能が追加されました。

バージョン 3.5 で変更: from __future__ import generator_stop による RuntimeError への変換が導入されました。**PEP 479** を参照してください。

バージョン 3.7 で変更: **PEP 479** が全てのコードでデフォルトで有効化されました: ジェネレータから送出された *StopIteration* は *RuntimeError* に変換されます。

exception StopAsyncIteration

イテレーションを停止するために、*asynchronous iterator* オブジェクトの __anext__() メソッドによって返される必要があります。

バージョン 3.5 で追加.

exception SyntaxError

パーザが構文エラーに遭遇した場合に送出されます。この例外は import 文、組み込み関数 *exec()* や *eval()* 、初期化スクリプトの読み込みや標準入力で (対話的な実行時にも) 起こる可能性があります。

The *str()* of the exception instance returns only the error message.

filename

構文エラーが発生したファイルの名前。

lineno

ファイルのエラーが発生した行番号。1 から数えはじめるため、ファイルの最初の行の lineno は 1 です。

offset

行のエラーが発生した列番号。1 から数えはじめるため、行の最初の文字の offset は 1 です。

text

エラーを含むソースコードのテキスト。

exception IndentationError

正しくないインデントに関する構文エラーの基底クラスです。これは *SyntaxError* のサブクラスです。

exception TabError

タブとスペースを一貫しない方法でインデントに使っているときに送出されます。これは *IndentationError* のサブクラスです。

exception SystemError

インタプリタが内部エラーを発見したが、状況は全ての望みを棄てさせるほど深刻ではないと思われる場合に送出されます。関連値は (下位層で) どの動作が失敗したかを示す文字列です。

使用中の Python インタプリタの作者または保守担当者にこのエラーを報告してください。このとき、Python インタプリタのバージョン (sys.version 。Python の対話的セッションを開始した際にも出力されます)、正確なエラーメッセージ (例外の関連値) を忘れずに報告してください。可能な場合にはエラーを引き起こしたプログラムのソースコードも報告してください。

exception SystemExit

この例外は *sys.exit()* 関数から送出されます。*Exception* をキャッチするコードに誤ってキャッチされないように、*Exception* ではなく *BaseException* を継承しています。これにより例外は上の階層に適切に伝わり、インタープリタを終了させます。この例外が処理されなかった場合はスタックのトレースバックを表示せずに Python インタープリタは終了します。コンストラクタは *sys.exit()* に渡されるオプション引数と同じものを受け取ります。値が整数の場合、システムの終了ステータス (C 言語の exit() 関数に渡すもの) を指定します。値が None の場合、終了ステータスは 0 です。それ以外の型の場合 (例えば str)、オブジェクトの値が表示され、終了ステータスは 1 です。

sys.exit() は、クリーンアップのための処理 (try 文の finally 節) が実行されるようにするため、またデバッガが制御不能になるリスクを冒さずにスクリプトを実行できるようにするために例外に変換されます。即座に終了することが真に強く必要であるとき (例えば、*os.fork()* を呼んだ後の子プロセス内) には *os._exit()* 関数を使うことができます。

code

コンストラクタに渡された終了ステータス又はエラーメッセージ。(デフォルトは None)

exception TypeError

組み込み演算または関数が適切でない型のオブジェクトに対して適用された際に送出されます。関連値は型の不整合に関して詳細を述べた文字列です。

この例外は、そのオブジェクトで実行しようとした操作がサポートされておらず、その予定もない場合にユーザーコードから送出されるかもしれません。オブジェクトでその操作をサポートするつもりだが、まだ実装を提供していないのであれば、送出する適切な例外は *NotImplementedError* です。

誤った型の引数が渡された場合は (例えば、*int* が期待されるのに、*list* が渡された) *TypeError* となるべきです。しかし、誤った値 (例えば、期待する範囲外の数) が引数として渡された場合は、*ValueError* となるべきです。

exception UnboundLocalError

関数やメソッド内のローカルな変数に対して参照を行ったが、その変数には値が代入されていなかった場合に送出されます。*NameError* のサブクラスです。

exception UnicodeError

Unicode に関するエンコードまたはデコードのエラーが発生した際に送出されます。*ValueError* のサブクラスです。

UnicodeError はエンコードまたはデコードのエラーの説明を属性として持っています。例えば、err.object[err.start:err.end] は、無効な入力のうちコーデックが処理に失敗した箇所を表します。

encoding

エラーを送出したエンコーディングの名前です。

reason
そのコーデックエラーを説明する文字列です。

object
コーデックがエンコードまたはデコードしようとしたオブジェクトです。

start
object の最初の無効なデータのインデクスです。

end
object の最後の無効なデータの次のインデクスです。

exception UnicodeEncodeError
Unicode 関連のエラーがエンコード中に発生した際に送出されます。*UnicodeError* のサブクラスです。

exception UnicodeDecodeError
Unicode 関連のエラーがデコード中に発生した際に送出されます。*UnicodeError* のサブクラスです。

exception UnicodeTranslateError
Unicode 関連のエラーが変換中に発生した際に送出されます。*UnicodeError* のサブクラスです。

exception ValueError
演算子や関数が、正しい型だが適切でない値を持つ引数を受け取ったときや、*IndexError* のようなより詳細な例外では記述できない状況で送出されます。

exception ZeroDivisionError
除算や剰余演算の第二引数が 0 であった場合に送出されます。関連値は文字列で、その演算における被演算子と演算子の型を示します。

以下の例外は、過去のバージョンとの後方互換性のために残されています; Python 3.3 より、これらは *OSError* のエイリアスです。

exception EnvironmentError

exception IOError

exception WindowsError
Windows でのみ利用できます。

5.2.1 OS 例外

以下の例外は *OSError* のサブクラスで、システムエラーコードに依存して送出されます。

exception BlockingIOError
ある操作が、ノンブロッキング操作に設定されたオブジェクト (例えばソケット) をブロックしそうになった場合に送出されます。errno EAGAIN, EALREADY, EWOULDBLOCK および EINPROGRESS に対応します。

BlockingIOError は、*OSError* の属性に加えて一つの属性を持ちます:

characters_written
ストリームがブロックされるまでに書き込まれた文字数を含む整数です。この属性は *io* からのバッファ I/O クラスを使っているときに利用できます。

exception ChildProcessError
子プロセスの操作が失敗した場合に送出されます。errno ECHILD に対応します。

exception ConnectionError
コネクション関係の問題の基底クラス。

サブクラスは *BrokenPipeError*, *ConnectionAbortedError*, *ConnectionRefusedError*, *ConnectionResetError* です。

exception BrokenPipeError
ConnectionError のサブクラスで、もう一方の端が閉じられたパイプに書き込こもうとするか、書き込みのためにシャットダウンされたソケットに書き込こもうとした場合に発生します。errno EPIPE と ESHUTDOWN に対応します。

exception ConnectionAbortedError
ConnectionError のサブクラスで、接続の試行が通信相手によって中断された場合に発生します。errno ECONNABORTED に対応します。

exception ConnectionRefusedError
ConnectionError のサブクラスで、接続の試行が通信相手によって拒否された場合に発生します。errno ECONNREFUSED に対応します。

exception ConnectionResetError
ConnectionError のサブクラスで、接続が通信相手によってリセットされた場合に発生します。errno ECONNRESET に対応します。

exception FileExistsError
すでに存在するファイルやディレクトリを作成しようとした場合に送出されます。errno EEXIST に対応します。

exception FileNotFoundError
要求されたファイルやディレクトリが存在しない場合に送出されます。errno ENOENT に対応します。

exception InterruptedError
システムコールが入力信号によって中断された場合に送出されます。errno *EINTR* に対応します。

バージョン 3.5 で変更: シグナルハンドラが例外を送出せず、システムコールが信号で中断された場合 Python は *InterruptedError* を送出する代わりにシステムコールを再試行するようになりました (論拠については **PEP 475** を参照してください) 。

exception IsADirectoryError
ディレクトリに (*os.remove()* などの) ファイル操作が要求された場合に送出されます。errno EISDIR に対応します。

exception NotADirectoryError
ディレクトリ以外のものに (*os.listdir()* などの) ディレクトリ操作が要求された場合に送出されま

す。errno ENOTDIR に対応します。

exception PermissionError
十分なアクセス権、例えばファイルシステム権限のない操作が試みられた場合に送出されます。errno EACCES および EPERM に対応します。

exception ProcessLookupError
与えられたプロセスが存在しない場合に送出されます。errno ESRCH に対応します。

exception TimeoutError
システム関数がシステムレベルでタイムアウトした場合に送出されます。errno ETIMEDOUT に対応します。

バージョン 3.3 で追加: 上記のすべての *OSError* サブクラスが追加されました。

参考:

PEP 3151 - OS および IO 例外階層の手直し

5.3 警告

次の例外は警告カテゴリとして使われます。詳細については **警告カテゴリ** のドキュメントを参照してください。

exception Warning
警告カテゴリの基底クラスです。

exception UserWarning
ユーザコードによって生成される警告の基底クラスです。

exception DeprecationWarning
他の Python 開発者へ向けて警告を発するときの、廃止予定の機能についての警告の基底クラスです。

exception PendingDeprecationWarning
古くなって将来的に廃止される予定だが、今のところは廃止されていない機能についての警告の基底クラスです。

近々起こる可能性のある機能廃止について警告を発することはまれなので、このクラスはめったに使われず、既に決まっている廃止については *DeprecationWarning* が望ましいです。

exception SyntaxWarning
曖昧な構文に対する警告の基底クラスです。

exception RuntimeWarning
あいまいなランタイム挙動に対する警告の基底クラスです。

exception FutureWarning
Python で書かれたアプリケーションのエンドユーザーへ向けて警告を発するときの、廃止予定の機能についての警告の基底クラスです。

exception ImportWarning

モジュールインポートの誤りと思われるものに対する警告の基底クラスです。

exception UnicodeWarning

Unicode に関連した警告の基底クラスです。

exception BytesWarning

bytes や *bytearray* に関連した警告の基底クラスです。

exception ResourceWarning

リソースの使用に関連した警告の基底クラスです。デフォルトの警告フィルタでは無視されます。

バージョン 3.2 で追加.

5.4 例外のクラス階層

組み込み例外のクラス階層は以下のとおりです:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
```

(次のページに続く)

(前のページからの続き)

```
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

第

SIX

テキスト処理サービス

この章で説明するモジュールは幅広い文字列操作や様々なテキスト処理サービスを提供しています。

codecs モジュールはテキスト処理と高い関連性を持った **バイナリデータ処理** のなかで説明されています。加えて Python の組み込み文字列型の **テキストシーケンス型** --- *str* のドキュメントも参照して下さい。

6.1 string --- 一般的な文字列操作

ソースコード: Lib/string.py

参考:

テキストシーケンス型 --- *str*

文字列メソッド

6.1.1 文字列定数

このモジュールで定義されている定数は以下の通りです:

string.ascii_letters
: 後述の *ascii_lowercase* と *ascii_uppercase* を合わせたもの。この値はロケールに依存しません。

string.ascii_lowercase
: 小文字 `'abcdefghijklmnopqrstuvwxyz'` 。この値はロケールに依存せず、固定です。

string.ascii_uppercase
: 大文字 `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 。この値はロケールに依存せず、固定です。

string.digits
: 文字列 `'0123456789'` です。

string.hexdigits
: 文字列 `'0123456789abcdefABCDEF'` です。

string.octdigits
: 文字列 '01234567' です。

string.punctuation
: C ロケールにおいて、区切り文字 (punctuation characters) として扱われる ASCII 文字の文字列です: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~.

string.printable
: 印刷可能な ASCII 文字で構成される文字列です。*digits*, *ascii_letters*, *punctuation* および *whitespace* を組み合わせたものです。

string.whitespace
: 空白 (whitespace) として扱われる ASCII 文字全てを含む文字列です。ほとんどのシステムでは、これはスペース (space)、タブ (tab)、改行 (linefeed)、復帰 (return)、改頁 (formfeed)、垂直タブ (vertical tab) です。

6.1.2 カスタムの文字列書式化

組み込みの文字列 (string) クラスには、**PEP 3101** で記述されている *format()* メソッドによって複雑な変数置換と値のフォーマットを行う機能があります。*string* モジュールの *Formatter* クラスでは、組み込みの *format()* メソッドと同じ実装を使用して、独自の文字列フォーマットの振る舞いを作成してカスタマイズすることができます。

class string.Formatter
: *Formatter* クラスは、以下のメソッドを持ちます:

  format(*format_string*, */*, **args*, ***kwargs*)
  : 主要な API メソッドです。書式文字列と、任意の位置引数およびキーワード引数のセットを取ります。これは、*vformat()* を呼び出す単なるラッパーです。

    バージョン 3.7 で変更: 書式文字列は **位置専用** の引数となりました。

  vformat(*format_string*, *args*, *kwargs*)
  : この関数はフォーマットの実際の仕事をします。この関数は、*args および **kwargs シンタックスを使用して、辞書を個々の引数として unpack してから再度 pack するのではなく、引数としてあらかじめ用意した辞書を渡したい場合のために、独立した関数として公開されます。*vformat()* は、書式文字列を文字データと置換フィールドに分解する仕事をします。それは、以下に記述する様々なメソッドを呼び出します。

  さらに、*Formatter* ではサブクラスによって置き換えられることを意図した次のようないくつかのメソッドが定義されています。

  parse(*format_string*)
  : format_string を探査し、タプル、(*literal_text*, *field_name*, *format_spec*, *conversion*) のイテラブルを返します。これは *vformat()* が文字列を文字としての文字データや置換フィールドに展開するために使用されます。

    タプルの値は、概念的に文字としての文字データと、それに続く単一の置換フィールドを表現しま

す。文字としての文字データが無い場合は (ふたつの置換フィールドが連続した場合などに起き得ます) 、*literal_text* は長さが 0 の文字列となります。置換フィールドが無い場合は、*field_name*, *format_spec* および *conversion* が `None` となります。

get_field(*field_name*, *args*, *kwargs*)

引数として与えた *parse()* (上記参照) により返される *field_name* を書式指定対象オブジェクトに変換します。返り値はタプル、(obj, used_key) です。デフォルトでは **PEP 3101** に規定される "0[name]" や "label.title" のような形式の文字列を引数としてとります。*args* と *kwargs* は *vformat()* に渡されます。返り値 *used_key* は、*get_value()* の *key* 引数と同じ意味を持ちます。

get_value(*key*, *args*, *kwargs*)

与えられたフィールドの値を取り出します。*key* 引数は整数でも文字列でも構いません。整数の場合は、位置引数 *args* のインデックス番号を示します。文字列の場合は、名前付きの引数 *kwargs* を意味します。

args 引数は、*vformat()* への位置引数のリストに設定され、*kwargs* 引数は、キーワード引数の辞書に設定されます。

フィールド名が (ピリオドで区切られた) いくつかの要素からなっている場合、最初の要素のみがこれらの関数に渡されます。残りの要素に関しては、通常の属性またはインデックスアクセスと同様に処理されます。

つまり、例えば、フィールドが '0.name' と表現されるとき、*get_value()* は、*key* 引数が 0 として呼び出されます。属性 `name` は、組み込みの *getattr()* 関数が呼び出され、*get_value()* が返されたのちに検索されます。

インデックスまたはキーワードが存在しないアイテムを参照した場合、*IndexError* または *KeyError* が送出されます。

check_unused_args(*used_args*, *args*, *kwargs*)

希望に応じて未使用の引数がないか確認する機能を実装します。この関数への引数は、書式指定文字列で実際に参照されるすべての引数のキーの set (位置引数の整数、名前付き引数の文字列) と、vformat に渡される *args* と *kwargs* への参照です。使用されない引数の set は、これらのパラメータから計算されます。*check_unused_args()* は、確認の結果が偽である場合に例外を送出するものとみなされます。

format_field(*value*, *format_spec*)

format_field() は単純に組み込みのグローバル関数 *format()* を呼び出します。このメソッドは、サブクラスをオーバーライドするために提供されます。

convert_field(*value*, *conversion*)

(*get_field()* が返す) 値を (*parse()* メソッドが返すタプルの形式で) 与えられた変換タイプとして変換します。デフォルトバージョンは 's' (str), 'r' (repr), 'a' (ascii) 変換タイプを理解します。

6.1.3 書式指定文字列の文法

str.format() メソッドと *Formatter* クラスは、文字列の書式指定に同じ文法を共有します (ただし、*Formatter* サブクラスでは、独自の書式指定文法を定義することが可能です)。この文法は フォーマット済み文字列リテラル の文法と関係してはいますが、少し洗練されておらず、特に任意の式がサポートされていません。

書式指定文字列は波括弧 `{}` に囲まれた ”置換フィールド” を含みます。波括弧に囲まれた部分以外は全て単純な文字として扱われ、変更を加えることなく出力へコピーされます。波括弧を文字として扱う必要がある場合は、二重にすることでエスケープすることができます: `{{` および `}}` 。

置換フィールドの文法は以下です:

```
replacement_field  ::=  "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name         ::=  arg_name ("." attribute_name | "[" element_index "]")*
arg_name           ::=  [identifier | digit+]
attribute_name     ::=  identifier
element_index      ::=  digit+ | index_string
index_string       ::=  <any source character except "]"> +
conversion         ::=  "r" | "s" | "a"
format_spec        ::=  <described in the next section>
```

もっと簡単にいうと、置換フィールドは *field_name* で始められます。これによって指定したオブジェクトの値が、置換フィールドの代わりに書式化され出力に挿入されます。*field_name* の後に、感嘆符 `'!'` を挟んで *conversion* フィールドを続けることができます。最後にコロン `':'` を挟んで、*format_spec* を書くことができます。これは、置換される値の非デフォルトの書式を指定します。

書式指定ミニ言語仕様 節も参照して下さい。

field_name それ自身は、数かキーワードのいずれかである *arg_name* から始まります。それが数である場合、位置引数を参照します。また、それがキーワードである場合、指定されたキーワード引数を参照します。書式文字列中で数の arg_names が順に 0, 1, 2, ... である場合、それらはすべて (いくつかではありません) 省略することができます。そして数 0, 1, 2, ... は、自動的にその順で挿入されます。*arg_name* は引用符で区切られていないので、書式文字列内の任意の辞書キー (例えば文字列 `'10'` や `':-]'` など) を指定することはできません。*arg_name* の後に任意の数のインデックス式または属性式を続けることができます。`'.name'` 形式の式は *getattr()* を使用して指定された属性を選択します。一方、`'[index]'` 形式の式は `__getitem__()` を使用してインデックス参照を行います。

バージョン 3.1 で変更: *str.format()* を使い、位置引数指定を省略することができます。`'{} {}'.format(a, b)` は `'{0} {1}'.format(a, b)` と同じになります。

バージョン 3.4 で変更: *Formatter* を使い、位置引数指定を省略することができます。

簡単な書式指定文字列の例を挙げます:

```
"First, thou shalt count to {0}"  # References first positional argument
"Bring me a {}"                   # Implicitly references the first positional argument
"From {} to {}"                   # Same as "From {0} to {1}"
"My quest is {name}"              # References keyword argument 'name'
"Weight in tons {0.weight}"       # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}"   # First element of keyword argument 'players'.
```

置換 *(conversion)* フィールドにより書式変換前に型の強制変換が実施されます。通常、値の書式変換は `__format__()` によって実施されます。しかしながら、場合によっては、文字列として変換することを強制したり、書式指定の定義をオーバーライドしたくなることもあります。`__format__()` の呼び出し前に値を文字列に変換すると、通常の書式変換の処理は飛ばされます。

現在 3 つの変換フラグがサポートされています: 値に対して *str()* を呼ぶ `'!s'` 、*repr()* を呼ぶ `'!r'` 、*ascii()* を呼ぶ `'!a'`。

いくつかの例です:

```
"Harold's a clever {0!s}"         # Calls str() on the argument first
"Bring out the holy {name!r}"     # Calls repr() on the argument first
"More {!a}"                       # Calls ascii() on the argument first
```

format_spec フィールドは、フィールド幅、文字揃え、埋め方、精度などの、値を表現する仕様を含みます。それぞれの値の型は、”formatting mini-language” 、または、*format_spec* の実装で定義されます。

ほとんどの組み込み型は、次のセクションに記載された共通の formatting mini-language をサポートします。

format_spec フィールド内には入れ子になった置換フィールドを含めることもできます。入れ子になった置換フィールドにはフィールド名、変換フラグ、書式指定を含めることができますが、さらに入れ子の階層を含めることはできません。format_spec 中の置換フィールドは *format_spec* 文字列が解釈される前に置き換えられます。これにより、値の書式を動的に指定することができます。

書式指定例 のいくつかの例も参照して下さい。

書式指定ミニ言語仕様

書式指定 (”Format specifications”) は書式指定文字列の個々の値を表現する方法を指定するための、置換フィールドで使用されます (**書式指定文字列の文法** および f-strings を参照してください) 。それらは、組み込み関数の *format()* 関数に直接渡されます。それぞれの書式指定可能な型について、書式指定がどのように解釈されるかが規定されます。

多くの組み込み型は、書式指定に関して以下のオプションを実装します。しかしながら、いくつかの書式指定オプションは数値型でのみサポートされます。

一般的な取り決めとして、空の書式指定は、値に対して *str()* を呼び出したときと同じ結果を与えます。通常、空でない書式指定はその結果を変更します。

一般的な書式指定子 (*standard format specifier*) の書式は以下です:

```
format_spec      ::=  [[fill]align][sign][#][0][width][grouping_option][.precision][type]
fill             ::=  <any character>
align            ::=  "<" | ">" | "=" | "^"
sign             ::=  "+" | "-" | " "
width            ::=  digit+
grouping_option  ::=  "_" | ","
precision        ::=  digit+
type             ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x"
```

有効な *align* 値を指定する場合、その前に *fill* 文字を付けることができます。この文字には任意の文字を指定でき、省略された場合はデフォルトの空白文字となります。formatted string literal の中や *str.format()* メソッドを使う場合はリテラルの波括弧 ("{" と "}") を *fill* 文字として使えないことに注意してください。ただし、波括弧を入れ子になった置換フィールド内に挿入することはできます。この制限は *format()* 関数には影響しません。

様々な align オプションの意味は以下のとおりです:

オプション	意味
'<'	利用可能なスペースにおいて、左詰めを強制します (ほとんどのオブジェクトにおいてのデフォルト)。
'>'	利用可能なスペースにおいて、右詰めを強制します (いくつかのオブジェクトにおいてのデフォルト)。
'='	符号 (があれば) の後ろを埋めます。'+000000120' のような形で表示されます。このオプションは数値型に対してのみ有効です。フィールド幅の直前が '0' の時はこれがデフォルトになります。
'^'	利用可能なスペースにおいて、中央寄せを強制します。

最小のフィールド幅が定義されない限り、フィールド幅はデータを表示するために必要な幅と同じになることに注意して下さい。そのため、その場合には、align オプションは意味を持ちません。

sign オプションは数値型に対してのみ有効であり、以下のうちのひとつとなります:

オプション	意味
'+'	符号の使用を、正数、負数の両方に対して指定します。
'-'	符号の使用を、負数に対してのみ指定します (デフォルトの挙動です)。
空白	空白を正数の前に付け、負号を負数の前に使用することを指定します。

The '#' option causes the "alternate form" to be used for the conversion. The alternate form is defined differently for different types. This option is only valid for integer, float and complex types. For integers, when binary, octal, or hexadecimal output is used, this option adds the prefix respective '0b', '0o', or

'0x' to the output value. For float and complex the alternate form causes the result of the conversion to always contain a decimal-point character, even if no digits follow it. Normally, a decimal-point character appears in the result of these conversions only if a digit follows it. In addition, for 'g' and 'G' conversions, trailing zeros are not removed from the result.

',' オプションは、千の位のセパレータにカンマを使うことを合図します。ロケール依存のセパレータには、代わりに 'n' の整数表現形式を使ってください。

バージョン 3.1 で変更: ',' オプションが追加されました (**PEP 378** も参照)。

'_' オプションは、浮動小数点数の表現型と整数の表現型 'd' における千倍ごとの区切り文字にアンダースコアを使うというしるしです。整数の表現型の 'b', 'o', 'x', 'X' では、4 桁ごとにアンダースコアが挿入されます。他の表現型でこのオプションを指定するとエラーになります。

バージョン 3.6 で変更: '_' オプションが追加されました (**PEP 515** も参照)。

width は 10 進数の整数で、接頭辞、セパレータ、他のフォーマット文字を含んだ最小の合計フィールド幅を定義します。指定されない場合、フィールド幅はその内容により決定されます。

alignment が明示的に与えられない場合、*width* フィールドにゼロ ('0') 文字を前置することは、数値型のための符号を意識した 0 パディングを可能にします。これは *fill* 文字に '0' を指定して、*alignment* タイプに '=' を指定したことと等価です。

precision は 10 進数で、'f' および 'F' で指定される浮動小数点数の小数点以下、あるいは 'g' および 'G' で指定される浮動小数点数の小数点の前後に表示される桁数を指定します。非数型に対しては、最大フィールド幅を表します。言い換えると、フィールドの内容から何文字を使用するかということです。*precision* は整数型に対しては使うことができません。

最後に、*type* は、データがどのように表現されるかを決定します。

利用可能な文字列の表現型は以下です:

型	意味
's'	文字列。これがデフォルトの値で、多くの場合省略されます。
None	's' と同じです。

利用可能な整数の表現型は以下です:

型	意味
'b'	2 進数。出力される数値は 2 を基数とします。
'c'	文字。数値を対応する Unicode 文字に変換します。
'd'	10 進数。出力される数値は 10 を基数とします。
'o'	8 進数。出力される数値は 8 を基数とします。
'x'	16 進数。出力される数値は 16 を基数とします。10 進で 9 を超える数字には小文字が使われます。
'X'	16 進数。出力される数値は 16 を基数とします。10 進で 9 を越える数字には大文字が使われます。
'n'	数値。現在のロケールに従い、区切り文字を挿入することを除けば、'd' と同じです。
None	'd' と同じです。

これらの表現型に加えて、整数は ('n' と None を除く) 以下の浮動小数点数の表現型で書式指定できます。そうすることで整数は書式変換される前に *float()* を使って浮動小数点数に変換されます。

利用可能な *float* と *Decimal* の表現型は以下です:

型	意味
'e'	Scientific notation. For a given precision p, formats the number in scientific notation with the letter 'e' separating the coefficient from the exponent. The coefficient has one digit before and p digits after the decimal point, for a total of p + 1 significant digits. With no precision given, uses a precision of 6 digits after the decimal point for *float*, and shows all coefficient digits for *Decimal*. If no digits follow the decimal point, the decimal point is also removed unless the # option is used.
'E'	指数表記です。大文字の 'E' を使うことを除いては、'e' と同じです。
'f'	Fixed-point notation. For a given precision p, formats the number as a decimal number with exactly p digits following the decimal point. With no precision given, uses a precision of 6 digits after the decimal point for *float*, and uses a precision large enough to show all coefficient digits for *Decimal*. If no digits follow the decimal point, the decimal point is also removed unless the # option is used.
'F'	固定小数点数表記です。nan が NAN に、inf が INF に変換されることを除き 'f' と同じです。
'g'	General format. For a given precision p >= 1, this rounds the number to p significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude. A precision of 0 is treated as equivalent to a precision of 1. The precise rules are as follows: suppose that the result formatted with presentation type 'e' and precision p-1 would have exponent exp. Then, if m <= exp < p, where m is -4 for floats and -6 for *Decimals*, the number is formatted with presentation type 'f' and precision p-1-exp. Otherwise, the number is formatted with presentation type 'e' and precision p-1. In both cases insignificant trailing zeros are removed from the significand, and the decimal point is also removed if there are no remaining digits following it, unless the '#' option is used. With no precision given, uses a precision of 6 significant digits for *float*. For *Decimal*, the coefficient of the result is formed from the coefficient digits of the value; scientific notation is used for values smaller than 1e-6 in absolute value and values where the place value of the least significant digit is larger than 1, and fixed-point notation is used otherwise. 正と負の無限大と 0 および NaN は精度に関係なくそれぞれ inf, -inf, 0, -0 および nan となります。
'G'	汎用フォーマットです。数値が大きくなったとき、'E' に切り替わることを除き、'g' と同じです。無限大と NaN の表示も大文字になります。
'n'	数値です。現在のロケールに合わせて、数値分割文字が挿入されることを除き、'g' と同じです。
'%'	パーセンテージです。数値は 100 倍され、固定小数点数フォーマット ('f') でパーセント記号付きで表示されます。
None	For *float* this is the same as 'g', except that when fixed-point notation is used to format the result, it always includes at least one digit past the decimal point. The precision used is as large as needed to represent the given value faithfully. For *Decimal*, this is the same as either 'g' or 'G' depending on the value of context.capitals for the current decimal context. The overall effect is to match the output of *str()* as altered by the other format modifiers.

書式指定例

この節では、*str.format()* 構文の例を紹介し、さらに従来の %-書式と比較します。

多くの場合、新構文に {} を加え、% の代わりに : を使うことで、古い %-書式に類似した書式になります。例えば、'%03.2f' は '{:03.2f}' と変換できます。

以下の例で示すように、新構文はさらに新たに様々なオプションもサポートしています。

位置引数を使ったアクセス:

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 3.1+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')      # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')   # arguments' indices can be repeated
'abracadabra'
```

名前を使ったアクセス:

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

引数の属性へのアクセス:

```
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
...  'and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part -5.0.'
>>> class Point:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

引数の要素へのアクセス:

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
```

%s と %r の置き換え:

```
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

テキストの幅を指定した整列:

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:*^30}'.format('centered')  # use '*' as a fill char
'***********centered***********'
```

%+f と %-f, % f の置換、そして符号の指定:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # show it always
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)  # show a space for positive numbers
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # show only the minus -- same as '{:f}; {:f}'
'3.140000; -3.140000'
```

%x と %o の置換、そして値に対する異なる底の変換:

```
>>> # format also supports binary numbers
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42)
'int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010'
```

千の位のセパレータにカンマを使用する:

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

パーセントを表示する:

```
>>> points = 19
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 86.36%'
```

型特有の書式指定を使う:

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

引数をネストする、さらに複雑な例:

```
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...
'left<<<<<<<<<<<<'
'^^^^^center^^^^^'
'>>>>>>>>>>>right'
>>>
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
>>> int(_, 16)
3232235521
>>>
>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print('{0:{width}{base}}'.format(num, base=base, width=width), end=' ')
...     print()
...
    5     5     5   101
    6     6     6   110
    7     7     7   111
    8     8    10  1000
    9     9    11  1001
   10     A    12  1010
   11     B    13  1011
```

6.1.4 テンプレート文字列

テンプレート文字列では **PEP 292** で解説されている単純な文字列置換ができます。テンプレート文字列の主な使い道は国際化 (i18n) です。というのは、その国際化の文脈において、より簡潔な文法と機能を持つテンプレート文字列を使うと、Python にある他の組み込みの文字列フォーマット機能よりも翻訳がしやすいからです。テンプレート文字列の上に構築された国際化のためのライブラリの例として、flufl.i18n を調べてみてください。

テンプレート文字列は `$` に基づいた置換をサポートしていて、次の規則が使われています:

- `$$` はエスケープ文字です; `$` 一つに置換されます。
- `$identifier` は `"identifier"` のマッピングキーに合致する置換プレースホルダーを指定します。デフォルトでは、`"identifier"` は大文字と小文字を区別しない ASCII 英数字 (アンダースコアを含む) からなら文字列に制限されています。文字列はアンダースコアか ASCII 文字から始まるものでなければなりません。`$` の後に識別子に使えない文字が出現すると、そこでプレースホルダ名の指定が終わります。
- `${identifier}` は `$identifier` と同じです。プレースホルダ名の後ろに識別子として使える文字列が続いていて、それをプレースホルダ名の一部として扱いたくない場合、例えば `"${noun}ification"` のような場合に必要な書き方です。

上記以外の書き方で文字列中に $ を使うと *ValueError* を送出します。

string モジュールでは、上記のような規則を実装した *Template* クラスを提供しています。*Template* のメソッドを以下に示します:

class string.Template(*template*)

コンストラクタはテンプレート文字列になる引数を一つだけ取ります。

substitute(*mapping={}*, */*, ***kwds*)

テンプレート置換を行い、新たな文字列を生成して返します。*mapping* はテンプレート中のプレースホルダに対応するキーを持つような任意の辞書類似オブジェクトです。辞書を指定する代わりに、キーワード引数も指定でき、その場合にはキーワードをプレースホルダ名に対応させます。*mapping* と *kwds* の両方が指定され、内容が重複した場合には、*kwds* に指定したプレースホルダを優先します。

safe_substitute(*mapping={}*, */*, ***kwds*)

substitute() と同じですが、プレースホルダに対応するものを *mapping* や *kwds* から見つけられなかった場合に、*KeyError* 例外を送出する代わりにもとのプレースホルダがそのまま入ります。また、*substitute()* とは違い、規則外の書き方で $ を使った場合でも、*ValueError* を送出せず単に $ を返します。

その他の例外も発生し得る一方で、このメソッドが「安全 (safe) 」と呼ばれているのは、置換操作は常に、例外を送出する代わりに利用可能な文字列を返そうとするからです。別の見方をすれば、*safe_substitute()* は区切り間違いによるぶら下がり (dangling delimiter) や波括弧の非対応、Python の識別子として無効なプレースホルダ名を含むような不正なテンプレートを何も警告せずに無視するため、安全とはいえないのです。

Template のインスタンスは、次のような public な属性を提供しています:

template

コンストラクタの引数 *template* に渡されたオブジェクトです。通常、この値を変更すべきではありませんが、読み出し専用アクセスを強制しているわけではありません。

Template の使い方の例を以下に示します:

```
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
...
ValueError: Invalid placeholder in string: line 1, col 11
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
...
KeyError: 'what'
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

さらに進んだ使い方: *Template* のサブクラスを派生して、プレースホルダの書式、区切り文字、テンプレート文字列の解釈に使われている正規表現全体をカスタマイズできます。こうした作業には、以下のクラス属性をオーバライドします:

- *delimiter* -- プレースホルダの開始を示すリテラル文字列です。デフォルトの値は `$` です。実装系はこの文字列に対して必要に応じて *re.escape()* を呼び出すので、正規表現になってしまうような文字列にしては **なりません** 。さらにクラスを作成した後に delimiter を変更できない (つまり、別の delimiter を設定したいのであれば、サブクラスの名前空間で行わなければならない) ことに注意してください。

- *idpattern* -- This is the regular expression describing the pattern for non-braced placeholders. The default value is the regular expression `(?a:[_a-z][_a-z0-9]*)`. If this is given and *braceidpattern* is `None` this pattern will also apply to braced placeholders.

  注釈: *flags* のデフォルトは `re.IGNORECASE` なので、`[a-z]` というパターンはいくつかの非 ASCII 文字に適合できます。そのため、ここではローカルの `a` フラグを使っています。

  バージョン 3.7 で変更: *braceidpattern* を使用すると、中括弧の内側と外側で使用する別々のパターンを定義できます。

- *braceidpattern* -- This is like *idpattern* but describes the pattern for braced placeholders. Defaults to `None` which means to fall back to *idpattern* (i.e. the same pattern is used both inside and outside braces). If given, this allows you to define different patterns for braced and unbraced placeholders.

  バージョン 3.7 で追加.

- *flags* -- 代入の認識のために使用される正規表現をコンパイルする際に適用される正規表現フラグ。デフォルト値は `re.IGNORECASE` です。`re.VERBOSE` が常にフラグに追加されるということに注意してください。したがって、カスタムな *idpattern* は verbose 正規表現の規約に従わなければなりません。

  バージョン 3.2 で追加.

他にも、クラス属性 *pattern* をオーバライドして、正規表現パターン全体を指定できます。オーバライドを行う場合、*pattern* の値は 4 つの名前つきキャプチャグループ (capturing group) を持った正規表現オブジェクトでなければなりません。これらのキャプチャグループは、上で説明した規則と、無効なプレースホルダに対する規則に対応しています:

- *escaped* -- このグループはエスケープシーケンス、すなわちデフォルトパターンにおける `$$` に対応します。

- *named* -- このグループは波括弧でくくらないプレースホルダ名に対応します; キャプチャグループに区切り文字を含めてはなりません。

- *braced* -- このグループは波括弧でくくったプレースホルダ名に対応します; キャプチャグループに区切り文字を含めてはなりません。

- *invalid* -- このグループはそのほかの区切り文字のパターン (通常は区切り文字一つ) に対応し、正規表現の末尾に出現しなければなりません。

6.1.5 ヘルパー関数

string.capwords(*s*, *sep=None*)

str.split() を使って引数を単語に分割し、*str.capitalize()* を使ってそれぞれの単語の先頭の文字を大文字に変換し、*str.join()* を使ってつなぎ合わせます。オプションの第 2 引数 *sep* が与えられないか None の場合、この置換処理は文字列中の連続する空白文字をスペース一つに置き換え、先頭と末尾の空白を削除します、それ以外の場合には *sep* は split と join に使われます。

6.2 re --- 正規表現操作

ソースコード: Lib/re.py

このモジュールは Perl に見られる正規表現マッチング操作と同様のものを提供します。

パターンおよび検索される文字列には、Unicode 文字列 (*str*) や 8 ビット文字列 (*bytes*) を使います。ただし、Unicode 文字列と 8 ビット文字列の混在はできません。つまり、Unicode 文字列にバイト列のパターンでマッチングしたり、その逆はできません。同様に、置換時の置換文字列はパターンおよび検索文字列の両方と同じ型でなくてはなりません。

正規表現では、特殊な形式を表すためや、特殊文字をその特殊な意味を発動させず使うために、バックスラッシュ文字 (`'\'`) を使います。こうしたバックスラッシュの使い方は、Python の文字列リテラルにおける同じ文字の使い方と衝突します。例えば、リテラルのバックスラッシュにマッチさせるには、パターン文字列として `'\\\\'` と書かなければなりません。なぜなら、正規表現は \\ でなければならないうえ、それぞれのバックスラッシュは標準の Python 文字列リテラルで \\ と表現せねばならないからです。Python の文字列リテラルにおいて、バックスラッシュの使用による不正なエスケープ文字がある場合は、*DeprecationWarning* が発生し、将来的には *SyntaxError* になることにも注意してください。この動作は、正規表現として有効な文字列に対しても同様です。

これを解決するには、正規表現パターンに Python の raw 文字列記法を使います。`'r'` を前置した文字列リテラル内ではバックスラッシュが特別扱いされません。従って `"\n"` が改行一文字からなる文字列であるのに対して、`r"\n"` は `'\'` と `'n'` の二文字からなる文字列です。通常、Python コード中では、パターンをこの raw 文字列記法を使って表現します。

重要なこととして、大抵の正規表現操作は、モジュールレベルの関数としても、**コンパイル済み正規表現** のメソッドとしても利用できます。関数は正規表現オブジェクトを前もってコンパイルする必要がない近道ですが、微調整のための変数が減ります。

参考:

サードパーティの regex モジュールは、標準ライブラリの *re* モジュールと互換な API を持ちながら、追加の機能とより徹底した Unicode サポートを提供します。

6.2.1 正規表現のシンタックス

正規表現 (または RE) は、その表現にマッチ (match) する文字列の集合を指定します。このモジュールの関数を使えば、ある文字列が与えられた正規表現にマッチするか (または、与えられた正規表現がある文字列にマッチするか、と言い換えても同じことになります) を検査できます。

正規表現を連結することで新しい正規表現を作れます。A と B がともに正規表現であれば AB も正規表現です。一般的に、ある文字列 p が A にマッチし、別の文字列 q が B にマッチするなら、文字列 pq は AB にマッチします。ただし、A または B に優先度の低い演算が含まれる場合や、A と B との間に境界条件がある場合や、番号付けされたグループ参照をしている場合、を除きます。こうして、ここで述べるような簡単な基本表現から、複雑な表現を容易に構築できます。正規表現に関する理論と実装の詳細については Friedl 本 [Frie09] か、コンパイラの構築に関するテキストを参照してください。

以下で正規表現の形式を簡単に説明します。詳細な情報ややさしい説明は、regex-howto を参照してください。

正規表現には、特殊文字と通常文字の両方を含められます。`'A'` 、`'a'` 、または `'0'` のようなほとんどの通常文字は、最も単純な正規表現です。これは単純に、その文字自体にマッチします。通常文字は連結できるので、`last` は文字列 `'last'` にマッチします。(この節では以降、正規表現は一般にクオートを使わず **この特殊スタイルで** 表記し、マッチ対象の文字列は、`'` **シングルクオートで括って**`'` 表記します。)

`'|'` や `'('` といったいくつかの文字は特殊です。特殊文字は通常文字の種別を表したり、周辺の通常文字に対する解釈方法に影響します。

繰り返しの修飾子 (`*`、`+`、`?`、`{m,n}` など) は直接入れ子にはできません。これは、非貪欲な修飾子の接尾辞 `?` や他の実装での他の修飾子との曖昧さを回避します。内側で繰り返したものをさらに繰り返すには、丸括弧が使えます。例えば、正規表現 `(?:a{6})*` は 6 の倍数個の `'a'` 文字にマッチします。

特殊文字を以下に示します:

`.` (ドット) デフォルトのモードでは改行以外の任意の文字にマッチします。*DOTALL* フラグが指定されていれば改行も含む全ての文字にマッチします。

`^` (キャレット) 文字列の先頭にマッチし、*MULTILINE* モードでは各改行の直後にもマッチします。

`$` 文字列の末尾、あるいは文字列の末尾の改行の直前にマッチし、*MULTILINE* モードでは改行の前にもマッチします。`foo` は 'foo' と 'foobar' の両方にマッチしますが、正規表現 `foo$` は 'foo' だけにマッチします。興味深いことに、`'foo1\nfoo2\n'` を `foo.$` で検索した場合、通常は 'foo2' だけにマッチしますが、*MULTILINE* モードでは 'foo1' にもマッチします。`$` だけで `'foo\n'` を検索した場合、2 つの (空の) マッチを見つけます: 1 つは改行の直前で、もう 1 つは文字列の末尾です。

`*` 直前の正規表現を 0 回以上、できるだけ多く繰り返したものにマッチさせる結果の正規表現にします。例えば `ab*` は 'a'、'ab'、または 'a' に任意個数の 'b' を続けたものにマッチします。

`+` 直前の正規表現を 1 回以上繰り返したものにマッチさせる結果の正規表現にします。例えば `ab+` は 'a' に 1 つ以上の 'b' が続いたものにマッチし、単なる 'a' にはマッチしません。

`?` 直前の正規表現を 0 回か 1 回繰り返したものにマッチさせる結果の正規表現にします。例えば `ab?` は 'a' あるいは 'ab' にマッチします。

?, +?, ?? '' 、'+' 、および '?' 修飾子は全て **貪欲** *(greedy)* マッチで、できるだけ多くのテキストにマッチします。この挙動が望ましくない時もあります。例えば正規表現 <.*> が '<a> b <c>' に対してマッチされると、'<a>' だけでなく文字列全体にマッチしてしまいます。修飾子の後に ? を追加すると、**非貪欲** *(non-greedy)* あるいは **最小** *(minimal)* のマッチが行われ、できるだけ **少ない** 文字にマッチします。正規表現 <.*?> を使うと '<a>' だけにマッチします。

{m} 直前の正規表現をちょうど m 回繰り返したものにマッチさせるよう指定します。それより少ないマッチでは正規表現全体がマッチしません。例えば、a{6} は 6 個ちょうどの 'a' 文字にマッチしますが、5 個ではマッチしません。

{m,n} 直前の正規表現を m 回から n 回、できるだけ多く繰り返したものにマッチさせる結果の正規表現にします。例えば、a{3,5} は、3 個から 5 個の 'a' 文字にマッチします。m を省略すると下限は 0 に指定され、n を省略すると上限は無限に指定されます。例として、a{4,}b は 'aaaab' や、1,000 個の 'a' 文字に 'b' が続いたものにマッチしますが、'aaab' にはマッチしません。コンマは省略できません、省略すると修飾子が上で述べた形式と混同されてしまうからです。

{m,n}? 結果の正規表現は、前にある正規表現を、m 回から n 回まで繰り返したものにマッチし、できるだけ **少なく** 繰り返したものにマッチするようにします。これは、前の修飾子の非貪欲版です。例えば、6 文字文字列 'aaaaaa' では、a{3,5} は、5 個の 'a' 文字にマッチしますが、a{3,5}? は 3 個の文字にマッチするだけです。

\ 特殊文字をエスケープ ('*' や '?' などの文字にマッチできるようにする) し、または特殊シーケンスを合図します。特殊シーケンスは後で議論します。

パターンを表現するのに raw 文字列を使っていないのであれば、Python ももまた、バックスラッシュを文字列リテラルでエスケープシーケンスとして使うことを思い出して下さい。そのエスケープシーケンスを Python のパーザが認識しないなら、そのバックスラッシュとそれに続く文字が結果の文字列に含まれます。しかし、Python が結果のシーケンスを認識するなら、そのバックスラッシュは 2 回繰り返さなければいけません。これは複雑で理解しにくいので、ごく単純な表現以外は、全て raw 文字列を使うことを強く推奨します。

[] 文字の集合を指定するのに使います。集合の中では:

- 文字を個別に指定できます。[amk] は 'a' 、'm' または 'k' にマッチします。
- 連続した文字の範囲を、'-' を 2 つの文字で挟んで指定できます。例えば、[a-z] はあらゆる小文字の ASCII 文字にマッチします。[0-5][0-9] は 00 から 59 まで全ての 2 桁の数字にマッチします。[0-9A-Fa-f] は任意の 16 進数字にマッチします。- がエスケープされているか (例: [a\-z])、先頭や末尾の文字にされていると (例: [-a] や [a-])、リテラル '-' にマッチします。
- 集合の中では、特殊文字はその特殊な意味を失います。例えば [(+*)] はリテラル文字 '(' 、'+' 、'*' 、または ')' のどれにでもマッチします。
- \w や \S のような文字クラス (後述) も集合の中で受理されますが、それにマッチする文字は *ASCII* や *LOCALE* モードが有効であるかに依存します。
- **補集合** をとって範囲内にない文字にマッチできます。集合の最初の文字が '^' なら、集合に **含まれない** 全ての文字にマッチします。例えば、[^5] は '5' を除くあらゆる文字にマッチし、[^^]

は '^' を除くあらゆる文字にマッチします。^ は集合の最初の文字でなければ特別の意味を持ちません。

- 集合の中でリテラル ']' にマッチさせるには、その前にバックスラッシュをつけるか、集合の先頭に置きます。例えば、[()[\]{}] と []()[{}] はどちらも括弧にマッチします。
- Unicode Technical Standard #18 にあるような集合の入れ子や集合操作が将来追加される可能性があります。これは構文を変化させるもので、この変化を容易にするために、さしあたって曖昧な事例には *FutureWarning* が送出されます。これはリテラル '[' で始まる集合や、リテラル文字の連続 '--' 、'&&' 、'~~' および '||' を含む集合を含みます。警告を避けるにはバックスラッシュでエスケープしてください。

バージョン 3.7 で変更: 文字セットが将来意味論的に変化する構造を含むなら *FutureWarning* が送出されます。

| *A* と *B* を任意の正規表現として、A|B は *A* と *B* のいずれかにマッチする正規表現を作成します。この方法で任意の数の正規表現を '|' で分離できます。これはグループ (下記参照) 中でも使えます。対象文字列を走査するとき、'|' で分離された正規表現は左から右へ順に試されます。一つのパターンが完全にマッチしたとき、そのパターン枝が受理されます。つまり、ひとたび *A* がマッチしてしまえば、例え *B* によって全体のマッチが長くなるとしても、*B* はもはや走査されません。言いかえると、'|' 演算子は決して貪欲にはなりません。リテラル '|' にマッチするには、\| を使うか、[|] のように文字クラス中に囲みます。

(...) 丸括弧で囲まれた正規表現にマッチするとともに、グループの開始と終了を表します。グループの中身は以下で述べるように、マッチが実行された後で回収したり、その文字列中で以降 \number 特殊シーケンスでマッチしたりできます。リテラル '(' や ')' にマッチするには、\(や \) を使うか、文字クラス中に囲みます: [(]、[)] 。

(?...) これは拡張記法です ('(' に続く '?' はそれ以上の意味を持ちません) 。'?' に続く最初の文字がこの構造の意味と特有の構文を決定します。拡張は一般に新しいグループを作成しません。ただし (?P<name>...) はこの法則の唯一の例外です。現在サポートされている拡張は以下の通りです。

(?aiLmsux) ('a' 、'i' 、'L' 、'm' 、's' 、'u' 、'x' の集合から 1 文字以上。) このグループは空文字列にマッチします。文字は正規表現全体に、対応するフラグを設定します。*re.A* (ASCII 限定マッチング)、*re.I* (大文字・小文字を区別しない)、*re.L* (ロケール依存)、*re.M* (複数行)、*re.S* (ドットが全てにマッチ)、re.U (Unicode マッチング)、*re.X* (冗長)。(各フラグについては **モジュールコンテンツ** で説明します。) これは、*flag* 引数を *re.compile()* 関数に渡すのではなく、フラグを正規表現の一部として含めたいときに便利です。フラグは表現文字列の先頭で使うべきです。

(?:...) 普通の丸括弧の、キャプチャしない版です。丸括弧で囲まれた正規表現にマッチしますが、このグループがマッチした部分文字列は、マッチを実行したあとで回収することも、そのパターン中で以降参照することも **できません** 。

(?aiLmsux-imsx:...) ('a' 、'i' 、'L' 、'm' 、's' 、'u' 、'x' の集合から 0 文字以上、必要ならさらに '-' に続けて 'i' 、'm' 、's' 、'x' の集合から 1 文字以上。) 文字は表現の一部に、対応するフラグを設定または除去します。*re.A* (ASCII 限定マッチング)、*re.I* (大文字・小文字を区別しない)、*re.L* (ロケール依存)、*re.M* (複数行)、*re.S* (ドットが全てにマッチ)、re.U (Unicode マッチング)、*re.X* (冗長)。(各フラグについては **モジュールコンテンツ** で説明します。)

文字 'a' 、'L' および 'u' は相互に排他であり、組み合わせることも '-' に続けることもできません。その代わり、これらの内一つがインライングループ中に現れると、外側のグループでのマッチングモードを上書きします。Unicode パターン中では (?a:...) は ASCII 限定マッチングに切り替え、(?u:...) は Unicode マッチング (デフォルト) に切り替えます。バイト列パターン中では、(?L:...) はロケール依存マッチングに切り替え、(?a:...) は ASCII 限定マッチング (デフォルト) に切り替えます。この上書きは狭いインライングループにのみ影響し、元のマッチングモードはグループ外では復元されます。

バージョン 3.6 で追加.

バージョン 3.7 で変更: 文字 'a' 、'L' および 'u' もグループ中で使えます。

(?P<name>...) 通常の丸括弧に似ていますが、このグループがマッチした部分文字列はシンボリックグループ名 *name* でアクセスできます。グループ名は有効な Python 識別子でなければならず、各グループ名は 1 個の正規表現内で一度だけ定義されていなければなりません。シンボリックグループは、そのグループが名前付けされていなかったかのように番号付けされたグループでもあります。

名前付きグループは 3 つのコンテキストで参照できます。パターンが (?P<quote>['\"]).*?(?P=quote) (シングルまたはダブルクオートで囲まれた文字列にマッチ) ならば:

グループ "quote" を参照するコンテキスト	参照する方法
その同じパターン中	• (?P=quote) (示したとおり) • \1
マッチオブジェクト *m* の処理時	• m.group('quote') • m.end('quote') (など)
re.sub() の *repl* 引数へ渡される文字列中	• \g<quote> • \g<1> • \1

(?P=name) 名前付きグループへの後方参照です。これは *name* という名前の既出のグループがマッチした文字列にマッチします。

(?#...) コメントです。括弧の中身は単純に無視されます。

(?=...) ... が次に続くものにマッチすればマッチしますが、文字列をまったく消費しません。これは **先読みアサーション** *(lookahead assertion)* と呼ばれます。例えば、Isaac (?=Asimov) は 'Isaac ' に、その後に 'Asimov' が続く場合にのみ、マッチします。

(?!...) ... が次に続くものにマッチしなければマッチします。これは **否定先読みアサーション** *(negative lookahead assertion)* です。例えば、Isaac (?!Asimov) は 'Isaac ' に、その後に 'Asimov' が続か **ない** 場合にのみ、マッチします。

`(?<=...)` その文字列における現在位置の前に、現在位置で終わる `...` とのマッチがあれば、マッチします。これは **後読みアサーション** と呼ばれます。`(?<=abc)def` は、後読みは 3 文字をバックアップし、含まれているパターンがマッチするか検査するので `'abcdef'` にマッチを見つけます。含まれるパターンは、固定長の文字列にのみマッチしなければなりません。すなわち、`abc` や `a|b` は許されますが、`a*` や `a{3,4}` は許されません。肯定後読みアサーションで始まるパターンは、検索される文字列の先頭とは決してマッチしないことに注意して下さい。*match()* 関数ではなく *search()* 関数を使う方が望ましいでしょう:

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

この例ではハイフンに続く単語を探します:

```
>>> m = re.search(r'(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

バージョン 3.5 で変更: 固定長のグループ参照をサポートするようになりました。

`(?<!...)` その文字列における現在位置の前に `...` とのマッチがなければ、マッチします。これは **否定後読みアサーション** *(negative lookbehind assertion)* と呼ばれます。肯定後読みアサーションと同様に、含まれるパターンは固定長の文字列にのみマッチしなければなりません。否定後読みアサーションで始まるパターンは検索される文字列の先頭でマッチできます。

`(?(id/name)yes-pattern|no-pattern)` 与えられた *id* や *name* のグループが存在すれば `yes-pattern` との、存在しなければ `no-pattern` とのマッチを試みます。`no-pattern` はオプションであり省略できます。例えば、`(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)` は貧弱な E-mail マッチングパターンで、`'<user@host.com>'` や `'user@host.com'` にはマッチしますが、`'<user@host.com'` や `'user@host.com>'` にはマッチしません。

特殊シーケンスは `'\'` と以下のリストの文字から構成されます。通常文字が ASCII 数字でも ASCII 文字でもなければ、結果の正規表現は 2 番目の文字にマッチします。例えば、`\$` は文字 `'$'` にマッチします。

`\number` 同じ番号のグループの中身にマッチします。グループは 1 から始まる番号をつけられます。例えば、`(.+) \1` は、`'the the'` あるいは `'55 55'` にマッチしますが、`'thethe'` にはマッチしません (グループの後のスペースに注意して下さい)。この特殊シーケンスは最初の 99 グループのうちの一つとのマッチにのみ使えます。*number* の最初の桁が 0 であるか、*number* が 3 桁の 8 進数であれば、それはグループのマッチとしてではなく、8 進値 *number* を持つ文字として解釈されます。文字クラスの `'['` と `']'` の間では全ての数値エスケープが文字として扱われます。

`\A` 文字列の先頭でのみマッチします。

`\b` 空文字列にマッチしますが、単語の先頭か末尾でのみです。単語は単語文字の並びとして定義されます。形式的には、`\b` は `\w` と `\W` 文字 (またはその逆) との、あるいは `\w` と文字列の先頭・末尾との境界として定義されます。例えば、`r'\bfoo\b'` は `'foo'`、`'foo.'`、`'(foo)'`、`'bar foo baz'` にはマッチしますが、`'foobar'` や `'foo3'` にはマッチしません。

デフォルトの Unicode 英数字は Unicode パターン中で使われるものと同じですが、これは *ASCII* フラグを使って変更できます。*LOCALE* フラグが使われているなら単語の境界は現在のロケールによって決定されます。Python の文字列リテラルとの互換性のため、文字列範囲中では、\b は後退 (backspace) 文字を表します。

\B 空文字列にマッチしますが、それが単語の先頭か末尾 **でない** ときのみです。つまり r'py\B' は 'python' 、'py3' 、'py2' にマッチしますが、'py' 、'py.' 、または 'py!' にはマッチしません。\B は \b のちょうど反対で、Unicode パターンにおける単語文字は Unicode 英数字およびアンダースコアですが、これは *ASCII* フラグを使って変更できます。*LOCALE* フラグが使われているなら単語の境界は現在のロケールによって決定されます。

\d

Unicode (str) **パターンでは**: 任意の Unicode 10 進数字 (Unicode 文字カテゴリ [Nd]) にマッチします。これは [0-9] とその他多数の数字を含みます。*ASCII* フラグが使われているなら [0-9] のみにマッチします。

8 **ビット** (bytes) **パターンでは**: 任意の 10 進数字にマッチします。これは [0-9] と等価です。

\D 10 進数字でない任意の文字にマッチします。これは \d の反対です。*ASCII* フラグが使われているならこれは [^0-9] と等価になります。

\s

Unicode (str) **パターンでは**: Unicode 空白文字 (これは [\t\n\r\f\v] その他多くの文字、例えば多くの言語におけるタイポグラフィ規則で定義されたノーブレークスペースなどを含みます) にマッチします。*ASCII* フラグが使われているなら、[\t\n\r\f\v] のみにマッチします。

8 **ビット** (bytes) **パターンでは**: ASCII 文字セットで空白文字と見なされる文字にマッチします。これは [\t\n\r\f\v] と等価です。

\S 空白文字ではない任意の文字にマッチします。これは \s の反対です。*ASCII* フラグが使われているならこれは [^ \t\n\r\f\v] と等価になります。

\w

Unicode (str) **パターンでは**: Unicode 単語文字にマッチします。これはあらゆる言語で単語の一部になりうるほとんどの文字、数字、およびアンダースコアを含みます。*ASCII* フラグが使われているなら、[a-zA-Z0-9_] のみにマッチします。

8 **ビット** (bytes) **パターンでは**: ASCII 文字セットで英数字と見なされる文字にマッチします。これは [a-zA-Z0-9_] と等価です。*LOCALE* フラグが使われているなら、現在のロケールで英数字と見なされる文字およびアンダースコアにマッチします。

\W 単語文字ではない任意の文字にマッチします。これは \w の反対です。*ASCII* フラグが使われているなら、これは [^a-zA-Z0-9_] と等価になります。*LOCALE* フラグが使われているなら、現在のロケールの英数字でもアンダースコアでもない文字にマッチします。

\Z 文字列の末尾でのみマッチします。

Python 文字列リテラルでサポートされている標準エスケープのほとんども正規表現パーザで受理されます:

```
\a      \b      \f      \n
\N      \r      \t      \u
\U      \v      \x      \\
```

(\b は単語の境界を表すのに使われ、文字クラス中でのみ ” 後退 (backspace)” 文字を意味することに注意してください。)

'\u'、'\U' および '\N' エスケープシーケンスは、Unicode パターン内でのみ認識されます。バイト列ではエラーとなります。ASCII 文字のエスケープで未知のものは将来使うために予約されていて、エラーとして扱われます。

8 進エスケープは限られた形式でのみ含まれます。その最初の桁が 0 であるか、それが 3 桁の 8 進数であるならば、それは 8 進エスケープと見なされます。そうでなければ、それはグループ参照です。文字列リテラルでは、8 進エスケープは常にたかだか 3 桁長です。

バージョン 3.3 で変更: '\u' と '\U' エスケープシーケンスが追加されました。

バージョン 3.6 で変更: '\' と ASCII 文字からなる未知のエスケープはエラーになります。

バージョン 3.8 で変更: '\N{name}' エスケープシーケンスが追加されました。文字列リテラルでは、同名の Unicode 文字に展開されます。('\N{EM DASH}' など)

6.2.2 モジュールコンテンツ

このモジュールはいくつかの関数、定数、例外を定義します。このうちいくつかの関数は、コンパイル済み正規表現がそなえる完全な機能のメソッドを簡易にしたものです。些細なものを除くほとんどのアプリケーションは常にコンパイル済み形式を使います。

バージョン 3.6 で変更: フラグ定数は、*enum.IntFlag* のサブクラスである RegexFlag のインスタンスになりました。

re.compile(*pattern*, *flags=0*)

正規表現パターンを **正規表現オブジェクト** にコンパイルし、以下に述べる *match()* 、*search()* その他のメソッドを使ってマッチングに使えるようにします。

式の挙動は *flags* の値を指定することで加減できます。値は以下の変数のうち任意のものを、ビット単位 OR (| 演算子) で組み合わせたものです。

シーケンス

```
prog = re.compile(pattern)
result = prog.match(string)
```

は、以下と同等です

```
result = re.match(pattern, string)
```

が、*re.compile()* を使い、結果の正規表現オブジェクトを保存して再利用するほうが、一つのプログラムでその表現を何回も使うときに効率的です。

注釈: *re.compile()* やモジュールレベルのマッチング関数に渡された最新のパターンはコンパイル済みのものがキャッシュされるので、一度に正規表現を少ししか使わないプログラムでは正規表現をコンパイルする必要はありません。

re.A
re.ASCII
: \w 、\W 、\b 、\B 、\d 、\D 、\s 、および \S に、完全な Unicode マッチングではなく ASCII 限定マッチングを行わせます。これは Unicode パターンでのみ意味があり、バイト列パターンでは無視されます。インラインフラグの (?a) に相当します。

後方互換性のため、re.U フラグ (と同義の re.UNICODE および埋め込みで使用する (?u)) はまだ存在しますが、Python 3 では文字列のマッチがデフォルトで Unicode (そしてバイト列では Unicode マッチングが扱えない) なので冗長です。

re.DEBUG
: コンパイル済み表現に関するデバッグ情報を表示します。相当するインラインフラグはありません。

re.I
re.IGNORECASE
: 大文字・小文字を区別しないマッチングを行います; [A-Z] のような正規表現は小文字にもマッチします。*re.ASCII* フラグを使い、非 ASCII マッチが無効化されていない限り、(Ü が ü にマッチするような) 完全な Unicode マッチングも有効です。*re.LOCALE* フラグも一緒に使われていない限り、現在のロケールがこのフラグの効果を変更することはありません。インラインフラグの (?i) に相当します。

Unicode パターン [a-z] または [A-Z] が *IGNORECASE* フラグとあわせて使われたとき、52 の ASCII 文字に加えて 4 の非 ASCII 文字 'İ' (U+0130, Latin capital letter I with dot above) 、'ı' (U+0131, Latin small letter dotless i) 、'ſ' (U+017F, Latin small letter long s) および 'K' (U+212A, Kelvin sign) にマッチすることに注意してください。*ASCII* フラグが使われているなら、文字 'a' から 'z' および 'A' から 'Z' にのみマッチします。

re.L
re.LOCALE
: \w 、\W 、\b 、\B および大文字・小文字を区別しないマッチングを、現在のロケールに依存させます。ロケールの仕組みは信頼できず、一度に一つの " 文化" しか扱えず、8 ビットロケールでしか働かないので、このフラグを使うことは推奨されません。Python 3 において Unicode (str) パターンでは Unicode マッチングはデフォルトですでに有効にされていて、異なるロケールや言語を扱えます。インラインフラグの (?L) に相当します。

バージョン 3.6 で変更: *re.LOCALE* はバイト列パターンにのみ使え、*re.ASCII* と互換ではありません。

バージョン 3.7 で変更: *re.LOCALE* フラグがあるコンパイル済み正規表現オブジェクトはコンパイル時のロケールに依存しなくなりました。マッチング時のロケールのみがマッチングの結果に影響します。

re.M

re.MULTILINE
: 指定されていると、パターン文字 '^' は文字列の先頭で、および各行の先頭 (各改行の直後) で、マッチします。そしてパターン文字 '$' は文字列の末尾で、および各行の末尾 (各改行の直前) で、マッチします。デフォルトでは、'^' は文字列の先頭でのみ、'$' は文字列の末尾および文字列の末尾の改行 (もしあれば) の直前でのみマッチします。インラインフラグの (?m) に相当します。

re.S
re.DOTALL
: '.' 特殊文字を、改行を含むあらゆる文字にマッチさせます。このフラグがなければ、'.' は、改行 **以外の** あらゆる文字とマッチします。インラインフラグの (?s) に相当します。

re.X
re.VERBOSE
: このフラグは正規表現を、パターンの論理的な節を視覚的に分割し、コメントを加えることで、見た目よく読みやすく書けるようにします。パターン中の空白は、文字クラス中にあるときと、エスケープされていないバックスラッシュの後にあるときと、*? 、(?: や (?P<...> のようなトークン中を除いて無視されます。ある行が文字クラス中でもエスケープされていないバックスラッシュの後でもない # を含むなら、一番左のそのような # から行末までの全ての文字は無視されます。

  つまり、10 進数字にマッチする下記のふたつの正規表現オブジェクトは、機能的に等価です:

```
a = re.compile(r"""\d +  # the integral part
                   \.    # the decimal point
                   \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

  インラインフラグの (?x) に相当します。

re.search(*pattern*, *string*, *flags=0*)
: *string* を走査し、正規表現 *pattern* がマッチを生じさせる最初の場所を探して、対応する **マッチオブジェクト** を返します。文字列内にパターンにマッチする場所がなければ None を返します。これは文字列のどこかで長さ 0 のマッチを見つけるのとは異なることに注意してください。

re.match(*pattern*, *string*, *flags=0*)
: *string* の先頭で 0 個以上の文字が正規表現 *pattern* にマッチすれば、対応する **マッチオブジェクト** を返します。文字列がパターンにマッチしなければ None を返します。これは長さ 0 のマッチとは異なることに注意して下さい。

  MULTILINE モードにおいても、*re.match()* は各行の先頭でマッチするのではなく、文字列の先頭でのみマッチすることに注意してください。

  string 中のどこででもマッチさせたいなら、代わりに *search()* を使ってください (*search() vs. match()* も参照してください)。

re.fullmatch(*pattern*, *string*, *flags=0*)
: *string* 全体が正規表現 *pattern* にマッチするなら、対応する **マッチオブジェクト** を返します。文字列がパターンにマッチしないなら None を返します。これは長さ 0 のマッチとは異なることに注意して下さい。

バージョン 3.4 で追加.

re.split(*pattern*, *string*, *maxsplit=0*, *flags=0*)

string を、出現した *pattern* で分割します。*pattern* 中でキャプチャの丸括弧が使われていれば、パターン中の全てのグループのテキストも結果のリストの一部として返されます。*maxsplit* が 0 でなければ、分割は最大 *maxsplit* 回起こり、残りの文字列はリストの最終要素として返されます。

```
>>> re.split(r'\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split(r'\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

セパレータ中にキャプチャグループがあり、それが文字列の先頭にマッチするなら、結果は空文字列で始まります。同じことが文字列の末尾にも言えます。

```
>>> re.split(r'(\W+)', '...words, words...')
['', '...', 'words', ', ', 'words', '...', '']
```

そうして、結果のリストにおいて、セパレータの構成要素は常に同じ相対的インデックスに見つかります。

パターンへの空マッチは、直前の空マッチに隣接していないときのみ文字列を分割します。

```
>>> re.split(r'\b', 'Words, words, words.')
['', 'Words', ', ', 'words', ', ', 'words', '.']
>>> re.split(r'\W*', '...words...')
['', '', 'w', 'o', 'r', 'd', 's', '', '']
>>> re.split(r'(\W*)', '...words...')
['', '...', '', '', 'w', '', 'o', '', 'r', '', 'd', '', 's', '...', '', '', '']
```

バージョン 3.1 で変更: オプションの flags 引数が追加されました。

バージョン 3.7 で変更: 空文字列にマッチしうるパターンでの分割をサポートするようになりました。

re.findall(*pattern*, *string*, *flags=0*)

string 中の *pattern* による全ての重複しないマッチを、文字列のリストとして返します。*string* は左から右へ走査され、マッチは見つかった順で返されます。パターン中に 1 つ以上のグループがあれば、グループのリストを返します。パターンに複数のグループがあればタプルのリストになります。空マッチは結果に含まれます。

バージョン 3.7 で変更: 空でないマッチが前の空マッチの直後から始められるようになりました。

re.finditer(*pattern*, *string*, *flags=0*)

string 中の正規表現 *pattern* の重複しないマッチ全てに渡る **マッチオブジェクト** を yield する **イテレータ** を返します。*string* は左から右へ走査され、マッチは見つかった順で返されます。空マッチは結果に含まれます。

バージョン 3.7 で変更: 空でないマッチが前の空マッチの直後から始められるようになりました。

re.sub(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

string 中に出現する最も左の重複しない *pattern* を置換 *repl* で置換することで得られる文字列を返します。パターンが見つからない場合、*string* がそのまま返されます。*repl* は文字列または関数です。*repl* が文字列の場合は、その中の全てのバックスラッシュエスケープが処理されます。\n は 1 つの改行文字に変換され、\r はキャリッジリターンに変換される、などです。ASCII 文字のエスケープで未知のものは将来使うために予約されていて、エラーとして扱われます。それ以外の \& のような未知のエスケープは残されます。\6 のような後方参照は、パターンのグループ 6 がマッチした部分文字列で置換されます。例えば:

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
...        r'static PyObject*\npy_\1(void)\n{',
...        'def myfunc():')
'static PyObject*\npy_myfunc(void)\n{'
```

repl が関数であれば、それは重複しない *pattern* が出現するたびに呼び出されます。この関数は一つの **マッチオブジェクト** 引数を取り、置換文字列を返します。例えば:

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
'Baked Beans & Spam'
```

パターンは、文字列でも **パターンオブジェクト** でも構いません。

オプション引数 *count* は出現したパターンを置換する最大の回数です。*count* は非負整数です。省略されるか 0 なら、出現した全てが置換されます。パターンへの空マッチは前の空マッチに隣接していないときのみ置換されるので、sub('x*', '-', 'abxd') は '-a-b--d-' を返します。

文字列型 *repl* 引数では、上で述べた文字エスケープや後方参照に加えて、\g<name> は (?P<name>...) 構文で定義された name という名前のグループがマッチした部分文字列を使い、\g<number> は対応するグループ番号を使います。よって \g<2> は \2 と等価ですが、\g<2>0 のような置換においても曖昧になりません。\20 は、グループ 20 への参照として解釈され、グループ 2 への参照にリテラル文字 '0' が続いたものとしては解釈されません。後方参照 \g<0> は正規表現とマッチした部分文字列全体で置き換わります。

バージョン 3.1 で変更: オプションの flags 引数が追加されました。

バージョン 3.5 で変更: マッチしなかったグループは空文字列に置き換えられます。

バージョン 3.6 で変更: *pattern* 中に '\' と ASCII 文字からなる未知のエスケープがあると、エラーになります。

バージョン 3.7 で変更: *repl* 中に '\' と ASCII 文字からなる未知のエスケープがあると、エラーになります。

バージョン 3.7 で変更: パターンへの空マッチは前の空でないマッチに隣接しているとき置き換えられます。

re.subn(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

sub() と同じ操作を行いますが、タプル (new_string、number_of_subs_made) を返します。

バージョン 3.1 で変更: オプションの flags 引数が追加されました。

バージョン 3.5 で変更: マッチしなかったグループは空文字列に置き換えられます。

re.escape(*pattern*)

pattern 中の特殊文字をエスケープします。これは正規表現メタ文字を含みうる任意のリテラル文字列にマッチしたい時に便利です。

```
>>> print(re.escape('http://www.python.org'))
http://www\.python\.org

>>> legal_chars = string.ascii_lowercase + string.digits + "!#$%&'*+-.^_`|~:"
>>> print('[%s]+' % re.escape(legal_chars))
[abcdefghijklmnopqrstuvwxyz0123456789!\#\$%\&'\*\+\-\.\^_`\|\~:]+

>>> operators = ['+', '-', '*', '/', '**']
>>> print('|'.join(map(re.escape, sorted(operators, reverse=True))))
/|\-|\+|\*\*|\*
```

この関数は、バックスラッシュのみをエスケープするべき *sub()* および *subn()* における置換文字列に使われてはなりません。例えば:

```
>>> digits_re = r'\d+'
>>> sample = '/usr/sbin/sendmail - 0 errors, 12 warnings'
>>> print(re.sub(digits_re, digits_re.replace('\\', r'\\'), sample))
/usr/sbin/sendmail - \d+ errors, \d+ warnings
```

バージョン 3.3 で変更: '_' 文字がエスケープされなくなりました。

バージョン 3.7 で変更: 正規表現で特別な意味を持つ文字だけがエスケープされます。結果として、'!'、'"'、'%'、"'"、','、'/'、':'、';'、'<'、'='、'>'、'@'、と "`" はもはやエスケープされません。

re.purge()

正規表現キャッシュをクリアします。

exception re.error(*msg*, *pattern=None*, *pos=None*)

ここの関数のいずれかに渡された文字列が有効な正規表現ではない (例: 括弧が対になっていない) とき、またはコンパイルやマッチングの際にその他なんらかのエラーが発生した場合に送出される例外です。文字列にパターンとマッチする部分がなくても、それはエラーではありません。エラーインスタンスには、次のような追加の属性があります。

msg

フォーマットされていないエラーメッセージです。

pattern
正規表現のパターンです。

pos
pattern のコンパイルに失敗した場所のインデックスです (None の場合もあります)。

lineno
pos に対応する行です (None の場合もあります)。

colno
pos に対応する列です (None の場合もあります)。

バージョン 3.5 で変更: 追加の属性が追加されました。

6.2.3 正規表現オブジェクト

コンパイル済み正規表現オブジェクトは以下のメソッドと属性をサポートします:

Pattern.**search**(*string*[, *pos*[, *endpos*]])
string を走査し、この正規表現がマッチを生じさせる最初の場所を探して、対応する **マッチオブジェクト** を返します。文字列内にパターンにマッチする場所がなければ None を返します。これは文字列内のある場所で長さが 0 のマッチが見つかった場合とは異なることに注意してください。

オプションの第二引数 *pos* は、文字列のどこから探し始めるかを指定するインデックスで、デフォルトでは 0 です。これは文字列のスライスと完全には同じではありません。パターン文字 '^' は本当の文字列の先頭と改行の直後でマッチしますが、検索を開始するインデックスでマッチするとは限りません。

オプションの引数 *endpos* は文字列がどこまで検索されるかを制限します。文字列の長さが *endpos* 文字だったかのようになるので、*pos* から endpos - 1 の文字に対してだけマッチを探します。*endpos* が *pos* よりも小さいと、マッチは見つかりません。そうでなければ、*rx* をコンパイル済み正規表現オブジェクトとして、rx.search(string, 0, 50) は rx.search(string[:50], 0) と等価です。

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")     # Match at index 0
<re.Match object; span=(0, 1), match='d'>
>>> pattern.search("dog", 1)  # No match; search doesn't include the "d"
```

Pattern.**match**(*string*[, *pos*[, *endpos*]])
string の **先頭** で 0 文字以上がこの正規表現とマッチするなら、対応する **マッチオブジェクト** を返します。文字列がパターンにマッチしなければ None を返します。これは長さ 0 のマッチとは異なることに注意してください。

オプションの *pos* および *endpos* 引数は *search()* メソッドのものと同じ意味です。

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.match("dog", 1)   # Match as "o" is the 2nd character of "dog".
<re.Match object; span=(1, 2), match='o'>
```

string 中のどこででもマッチさせたいなら、代わりに *search()* を使ってください (*search() vs. match()* も参照してください)。

Pattern.**fullmatch**(*string*[, *pos*[, *endpos*]])

string 全体がこの正規表現にマッチすれば、対応する **マッチオブジェクト** を返します。文字列がパターンにマッチしなければ None を返します。これは長さ 0 のマッチとは異なることに注意してください。

オプションの *pos* および *endpos* 引数は *search()* メソッドのものと同じ意味です。

```
>>> pattern = re.compile("o[gh]")
>>> pattern.fullmatch("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.fullmatch("ogre")     # No match as not the full string matches.
>>> pattern.fullmatch("doggie", 1, 3)   # Matches within given limits.
<re.Match object; span=(1, 3), match='og'>
```

バージョン 3.4 で追加.

Pattern.**split**(*string*, *maxsplit=0*)

split() 関数にこのコンパイル済みパターンを使うのと同じです。

Pattern.**findall**(*string*[, *pos*[, *endpos*]])

findall() 関数にこのコンパイル済みパターンを使うのと似ていますが、オプションの *pos* および *endpos* 引数で *search()* のように検索範囲を制限できます。

Pattern.**finditer**(*string*[, *pos*[, *endpos*]])

finditer() 関数にこのコンパイル済みパターンを使うのと似ていますが、オプションの *pos* および *endpos* 引数で *search()* のように検索範囲を制限できます。

Pattern.**sub**(*repl*, *string*, *count=0*)

sub() 関数にこのコンパイル済みパターンを使うのと同じです。

Pattern.**subn**(*repl*, *string*, *count=0*)

subn() 関数にこのコンパイル済みパターンを使うのと同じです。

Pattern.**flags**

正規表現のマッチングフラグです。これは *compile()* に与えられたフラグ、パターン中の (?...) インラインフラグ、およびパターンが Unicode 文字列だった時の UNICODE のような暗黙のフラグの組み合わせです。

Pattern.**groups**

パターン中のキャプチャグループの数です。

Pattern.**groupindex**

(?P<id>) で定義されたあらゆるシンボリックグループ名をグループ番号へ写像する辞書です。シンボリックグループがパターン中で全く使われていなければ、この辞書は空です。

Pattern.**pattern**

パターンオブジェクトがコンパイルされた元のパターン文字列です。

バージョン 3.7 で変更: *copy.copy()* および *copy.deepcopy()* をサポートするようになりました。コンパイル済み正規表現オブジェクトはアトミックであると見なされます。

6.2.4 マッチオブジェクト

マッチオブジェクトのブール値は常に True です。*match()* および *search()* はマッチがないとき None を返すので、マッチがあるか単純な if 文で判定できます。

```
match = re.search(pattern, string)
if match:
    process(match)
```

マッチオブジェクトは以下のメソッドおよび属性をサポートしています:

Match.**expand**(*template*)

テンプレート文字列 *template* に *sub()* メソッドの行うバックスラッシュ置換を行って得られる文字列を返します。\n のようなエスケープは適切な文字に変換され、数後方参照 (\1, \2) および名前付き後方参照 (\g<1>, \g<name>) は対応するグループの内容に置換されます。

バージョン 3.5 で変更: マッチしなかったグループは空文字列に置き換えられます。

Match.**group**([*group1*, ...])

このマッチの 1 つ以上のサブグループを返します。引数が 1 つなら結果は 1 つの文字列です。複数の引数があれば、結果は引数ごとに 1 項目のタプルです。引数がなければ、*group1* はデフォルトで 0 (マッチ全体が返される) です。*groupN* 引数が 0 なら、対応する返り値はマッチした文字列全体です。1 以上 99 以下なら、丸括弧による対応するグループにマッチする文字列です。グループ番号が負であるかパターン中で定義されたグループの数より大きければ、*IndexError* 例外が送出されます。あるグループがパターンのマッチしなかった部分に含まれているなら、対応する結果は None です。あるグループがパターンの複数回マッチした部分に含まれているなら、最後のマッチが返されます。

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)       # The entire match
'Isaac Newton'
>>> m.group(1)       # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)       # The second parenthesized subgroup.
'Newton'
>>> m.group(1, 2)    # Multiple arguments give us a tuple.
('Isaac', 'Newton')
```

正規表現が (?P<name>...) 構文を使うなら、*groupN* 引数はグループ名でグループを識別する文字列でも構いません。文字列引数がパターン中でグループ名として使われていなければ、*IndexError* 例外が送出されます。

やや複雑な例:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
```

(次のページに続く)

(前のページからの続き)

```
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

名前付きグループはインデックスでも参照できます:

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

あるグループが複数回マッチすると、その最後のマッチにのみアクセスできます:

```
>>> m = re.match(r"(..)+", "a1b2c3")  # Matches 3 times.
>>> m.group(1)                        # Returns only the last match.
'c3'
```

Match.__getitem__(*g*)

これは m.group(g) と同等です。これでマッチの個別のグループに簡単にアクセスできます:

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m[0]       # The entire match
'Isaac Newton'
>>> m[1]       # The first parenthesized subgroup.
'Isaac'
>>> m[2]       # The second parenthesized subgroup.
'Newton'
```

バージョン 3.6 で追加.

Match.**groups**(*default=None*)

このマッチの、1 からパターン中のグループ数まで、全てのサブグループを含むタプルを返します。*default* 引数はマッチに関係しなかったグループに使われます。デフォルトでは None です。

例えば:

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

少数位およびその後の全てをオプションにすると、全てのグループがマッチに関係するとは限りません。そういったグループは *default* 引数が与えられない限りデフォルトで None になります。

```
>>> m = re.match(r"(\d+)\.?(\d+)?", "24")
>>> m.groups()      # Second group defaults to None.
('24', None)
>>> m.groups('0')   # Now, the second group defaults to '0'.
('24', '0')
```

Match.**groupdict**(*default=None*)

このマッチの、全ての **名前付き** サブグループを含む、サブグループ名をキーとする辞書を返します。*default* 引数はマッチに関係しなかったグループに使われます。デフォルトは None です。例えば:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

Match.**start**([*group*])

Match.**end**([*group*])

group がマッチした部分文字列の先頭と末尾のインデックスを返します。*group* はデフォルトで 0 (マッチした部分文字列全体という意味) です。*group* が存在してかつマッチには寄与していなかったなら -1 を返します。マッチオブジェクト *m* と、マッチに寄与したグループ *g* に対して、グループ *g* がマッチした部分文字列 (m.group(g) と等価です) は以下の通りです

```
m.string[m.start(g):m.end(g)]
```

group が空文字列にマッチしていたら m.start(group) は m.end(group) と等しくなることに注意して下さい。例えば、m = re.search('b(c?)', 'cba') とすると、m.start(0) は 1 で、m.end(0) は 2 で、m.start(1) と m.end(1) はともに 2 であり、m.start(2) は *IndexError* 例外を発生します。

メールアドレスから *remove_this* を取り除く例:

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

Match.**span**([*group*])

マッチ *m* について、2 タプル (m.start(group), m.end(group)) を返します。*group* がマッチに寄与していなければ、これは (-1, -1) です。*group* はデフォルトで 0 、マッチ全体です。

Match.**pos**

正規表現オブジェクト の *search()* や *match()* に渡された *pos* の値です。これは正規表現エンジンがマッチを探し始める位置の文字列のインデックスです。

Match.**endpos**

正規表現オブジェクト の *search()* や *match()* に渡された *endpos* の値です。これは正規表現エンジンがそれ以上は進まない文字列のインデックスです。

Match.**lastindex**

最後にマッチしたキャプチャグループの整数インデックスです。どのグループも全くマッチしなければ None です。例えば、表現 (a)b 、((a)(b)) や ((ab)) が 'ab' に適用されると lastindex == 1 となり、同じ文字列に (a)(b) が適用されると lastindex == 2 となります。

Match.**lastgroup**

最後にマッチしたキャプチャグループの名前です。そのグループに名前がないか、どのグループも全くマッチしていなければ None です。

Match.re
: このマッチインスタンスを生じさせた *match()* または *search()* メソッドの属する **正規表現オブジェクト** です。

Match.string
: *match()* や *search()* へ渡された文字列です。

バージョン 3.7 で変更: *copy.copy()* および *copy.deepcopy()* をサポートするようになりました。マッチオブジェクトはアトミックであると見なされます。

6.2.5 正規表現の例

ペアの確認

この例では、マッチオブジェクトをより美しく表示するために、この補助関数を使用します:

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

あなたがポーカープログラムを書いているとします。プレイヤーの手札は 5 文字の文字列によって表され、それぞれの文字が 1 枚のカードを表します。"a" はエース、"k" はキング、"q" はクイーン、"j" はジャック、"t" は 10、そして "2" から "9" はその数字のカードを表します。

与えられた文字列が有効な手札であるか見るには、以下のようにできます:

```
>>> valid = re.compile(r"^[a2-9tjqk]{5}$")
>>> displaymatch(valid.match("akt5q"))  # Valid.
"<Match: 'akt5q', groups=()>"
>>> displaymatch(valid.match("akt5e"))  # Invalid.
>>> displaymatch(valid.match("akt"))    # Invalid.
>>> displaymatch(valid.match("727ak"))  # Valid.
"<Match: '727ak', groups=()>"
```

最後の手札、"727ak" 、はペア、すなわち同じ値の 2 枚のカードを含みます。正規表現でこれにマッチするには、このように後方参照を使えます:

```
>>> pair = re.compile(r".*(.).*\1")
>>> displaymatch(pair.match("717ak"))     # Pair of 7s.
"<Match: '717', groups=('7',)>"
>>> displaymatch(pair.match("718ak"))     # No pairs.
>>> displaymatch(pair.match("354aa"))     # Pair of aces.
"<Match: '354aa', groups=('a',)>"
```

ペアになっているのがどのカードか調べるには、このようにマッチオブジェクトの *group()* メソッドを使えます:

```
>>> pair = re.compile(r".*(.).*\1")
>>> pair.match("717ak").group(1)
'7'

# Error because re.match() returns None, which doesn't have a group() method:
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r".*(.).*\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'

>>> pair.match("354aa").group(1)
'a'
```

scanf() をシミュレートする

Python には現在のところ、scanf() に相当するものがありません。正規表現は一般的に、scanf() のフォーマット文字列より強力ですが、冗長でもあります。以下の表に、scanf() のフォーマットトークンと正規表現のおおよその対応付けを示します。

scanf() トークン	正規表現
%c	.
%5c	.{5}
%d	[-+]?\d+
%e, %E, %f, %g	[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?
%i	[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)
%o	[-+]?[0-7]+
%s	\S+
%u	\d+
%x, %X	[-+]?(0[xX])?[\dA-Fa-f]+

以下のような文字列からファイル名と数を抽出するには

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

以下のように scanf() フォーマットを使えます

```
%s - %d errors, %d warnings
```

等価な正規表現はこうです

```
(\S+) - (\d+) errors, (\d+) warnings
```

search() vs. match()

Python は正規表現ベースの 2 つの異なる基本的な関数、文字列の先頭でのみのマッチを確認する *re.match()* および、文字列中の位置にかかわらずマッチを確認する *re.search()* (これが Perl でのデフォルトの挙動です) を提供しています。

例えば:

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("c", "abcdef")   # Match
<re.Match object; span=(2, 3), match='c'>
```

'^' で始まる正規表現を *search()* で使って、マッチを文字列の先頭でのみに制限できます:

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("^c", "abcdef")  # No match
>>> re.search("^a", "abcdef")  # Match
<re.Match object; span=(0, 1), match='a'>
```

ただし、*MULTILINE* モードにおいて *match()* は文字列の先頭でのみマッチし、'^' で始まる正規表現で *search()* を使うと各行の先頭でマッチすることに注意してください。

```
>>> re.match('X', 'A\nB\nX', re.MULTILINE)  # No match
>>> re.search('^X', 'A\nB\nX', re.MULTILINE)  # Match
<re.Match object; span=(4, 5), match='X'>
```

電話帳を作る

split() は渡されたパターンで文字列を分割してリストにします。このメソッドは、テキストデータをデータ構造に変換して、読みやすくしたり、以下の例で実演する電話帳作成のように Python で編集したりしやすくするのに、非常に役に立ちます。

最初に、入力を示します。通常、これはファイルからの入力になるでしょう。ここでは、3 重引用符の書式とします。

```
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street
...
... Ronald Heathmore: 892.345.3428 436 Finley Avenue
... Frank Burger: 925.541.7625 662 South Dogwood Way
...
...
... Heather Albrecht: 548.326.4584 919 Park Place"""
```

各項目は 1 つ以上の改行で区切られています。まずは文字列を変換して、空行でない各行を項目とするリストにします:

```
>>> entries = re.split("\n+", text)
>>> entries
['Ross McFluff: 834.345.1254 155 Elm Street',
```

(次のページに続く)

(前のページからの続き)

```
'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
'Frank Burger: 925.541.7625 662 South Dogwood Way',
'Heather Albrecht: 548.326.4584 919 Park Place']
```

そして各項目を、ファーストネーム、ラストネーム、電話番号、住所に分割してリストにします。分割パターンである空白文字は住所にも含まれるので、*split()* の maxsplit 引数を使います:

```
>>> [re.split(":? ", entry, 3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

この :? パターンはラストネームの次のコロンにマッチして、分割結果のリストに出てこないようにします。maxsplit を 4 にすれば、家屋番号とストリート名を分割できます:

```
>>> [re.split(":? ", entry, 4) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]
```

テキストの秘匿

sub() は出現する各パターンを文字列で、または関数の返り値で置き換えます。この例ではテキストを「秘匿」する関数と合わせて *sub()* を使うところを実演します。具体的には、文中の各単語について、最初と最後の文字を除く全ての文字をランダムに並び替えます:

```
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + "".join(inner_word) + m.group(3)
>>> text = "Professor Abdolmalek, please report your absences promptly."
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reoprt yuor asnebces potlmrpy.'
```

全ての副詞を見つける

search() は最初のパターンにのみマッチしますが、*findall()* は出現する **全ての** パターンにマッチします。例えば、ライターがあるテキストの全ての副詞を見つけたいなら、以下のように *findall()* を使えます:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly", text)
['carefully', 'quickly']
```

全ての副詞とその位置を見つける

パターンの全てのマッチについて、マッチしたテキスト以上の情報が必要なら、文字列ではなく マッチオブジェクト を返す *finditer()* が便利です。先の例に続いて、ライターがあるテキストの全ての副詞 **およびその位置** を見つけたいなら、以下のように *finditer()* を使えます:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly", text):
...     print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
07-16: carefully
40-47: quickly
```

Raw 文字列記法

Raw 文字列記法 (`r"text"`) で正規表現をまともに保てます。それがなければ、正規表現中のバックスラッシュ (`'\'`) を個々にバックスラッシュを前置してエスケープしなければなりません。例えば、以下の 2 行のコードは機能的に等価です:

```
>>> re.match(r"\W(.)\1\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
>>> re.match("\\W(.)\\1\\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
```

リテラルのバックスラッシュにマッチさせたいなら、正規表現中ではエスケープする必要があります。Raw 文字列記法では、`r"\\"` になります。Raw 文字列記法を用いないと、`"\\\\"` としなくてはならず、以下のコードは機能的に等価です:

```
>>> re.match(r"\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
>>> re.match("\\\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
```

トークナイザを書く

トークナイザやスキャナ は文字列を解析し、文字のグループにカテゴリ分けします。これはコンパイラやインタプリタを書くうえで役立つ第一段階です。

テキストのカテゴリは正規表現で指定されます。この技法では、それらを一つのマスター正規表現に結合し、マッチの連続についてループします:

```
from typing import NamedTuple
import re

class Token(NamedTuple):
    type: str
    value: str
    line: int
    column: int
```

(次のページに続く)

(前のページからの続き)

```
def tokenize(code):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('ASSIGN',   r':='),           # Assignment operator
        ('END',      r';'),            # Statement terminator
        ('ID',       r'[A-Za-z]+'),    # Identifiers
        ('OP',       r'[+\-*/]'),      # Arithmetic operators
        ('NEWLINE',  r'\n'),           # Line endings
        ('SKIP',     r'[ \t]+'),       # Skip over spaces and tabs
        ('MISMATCH', r'.'),            # Any other character
    ]
    tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
    line_num = 1
    line_start = 0
    for mo in re.finditer(tok_regex, code):
        kind = mo.lastgroup
        value = mo.group()
        column = mo.start() - line_start
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'ID' and value in keywords:
            kind = value
        elif kind == 'NEWLINE':
            line_start = mo.end()
            line_num += 1
            continue
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise RuntimeError(f'{value!r} unexpected on line {line_num}')
        yield Token(kind, value, line_num, column)

statements = '''
    IF quantity THEN
        total := total + price * quantity;
        tax := price * 0.05;
    ENDIF;
'''

for token in tokenize(statements):
    print(token)
```

このトークナイザは以下の出力を作成します:

```
Token(type='IF', value='IF', line=2, column=4)
Token(type='ID', value='quantity', line=2, column=7)
Token(type='THEN', value='THEN', line=2, column=16)
Token(type='ID', value='total', line=3, column=8)
Token(type='ASSIGN', value=':=', line=3, column=14)
```

(次のページに続く)

(前のページからの続き)

```
Token(type='ID', value='total', line=3, column=17)
Token(type='OP', value='+', line=3, column=23)
Token(type='ID', value='price', line=3, column=25)
Token(type='OP', value='*', line=3, column=31)
Token(type='ID', value='quantity', line=3, column=33)
Token(type='END', value=';', line=3, column=41)
Token(type='ID', value='tax', line=4, column=8)
Token(type='ASSIGN', value=':=', line=4, column=12)
Token(type='ID', value='price', line=4, column=15)
Token(type='OP', value='*', line=4, column=21)
Token(type='NUMBER', value=0.05, line=4, column=23)
Token(type='END', value=';', line=4, column=27)
Token(type='ENDIF', value='ENDIF', line=5, column=4)
Token(type='END', value=';', line=5, column=9)
```

6.3 difflib --- 差分の計算を助ける

ソースコード: Lib/difflib.py

このモジュールは、シーケンスを比較するためのクラスや関数を提供しています。例えば、ファイルの差分を計算して、それを HTML や context diff, unified diff などいろいろなフォーマットで出力するために、このモジュールを利用できます。ディレクトリやファイル群を比較するためには、*filecmp* モジュールも参照してください。

class difflib.SequenceMatcher

柔軟性のあるクラスで、二つのシーケンスの要素が **ハッシュ可能** な型であれば、どの型の要素を含むシーケンスも比較可能です。基本的なアルゴリズムは、1980 年代の後半に発表された Ratcliff と Obershelp による” ゲシュタルトパターンマッチング” と大げさに名づけられたアルゴリズム以前から知られている、やや凝ったアルゴリズムです。その考え方は、”junk” 要素を含まない最も長い互いに隣接したマッチ列を探すことです。ここで、”junk” 要素とは、空行や空白などの、意味を持たない要素のことです。(junk を処理するのは、Ratcliff と Obershelp のアルゴリズムに追加された拡張です。) この考え方は、マッチ列の左右に隣接するシーケンスの断片に対して再帰的にあてはめられます。この方法では編集を最小にするシーケンスは生まれませんが、人間の目からみて「正しい感じ」にマッチする傾向があります。

実行時間: 基本的な Ratcliff-Obershelp アルゴリズムは、最悪の場合 3 乗、期待値で 2 乗となります。*SequenceMatcher* オブジェクトでは、最悪のケースで 2 乗、期待値は比較されるシーケンス中に共通に現れる要素数に非常にややこしく依存しています。最良の場合は線形時間になります。

自動 junk ヒューリスティック: *SequenceMatcher* は、シーケンスの特定の要素を自動的に junk として扱うヒューリスティックをサポートしています。このヒューリスティックは、各個要素がシーケンス内に何回現れるかを数えます。ある要素の重複数が (最初のものは除いて) 合計でシーケンスの 1% 以上になり、そのシーケンスが 200 要素以上なら、その要素は ”popular” であるものとしてマークされ、シーケンスのマッチングの目的からは junk として扱われます。このヒューリスティックは、*SequenceMatcher* の作成時に `autojunk` パラメタを `False` に設定することで無効化できます。

バージョン 3.2 で追加: *autojunk* パラメータ。

class difflib.Differ

テキスト行からなるシーケンスを比較するクラスです。人が読むことのできる差分を作成します。Differ クラスは *SequenceMatcher* クラスを利用して、行からなるシーケンスを比較したり、(ほぼ) 同一の行内の文字を比較したりします。

Differ クラスによる差分の各行は、2 文字のコードで始まります:

コード	意味
`'- '`	行はシーケンス 1 にのみ存在する
`'+ '`	行はシーケンス 2 にのみ存在する
`'  '`	行は両方のシーケンスで同一
`'? '`	行は入力シーケンスのどちらにも存在しない

'?' で始まる行は、行内のどこに差異が存在するかに注意を向けようとします。その行は、入力されたシーケンスのどちらにも存在しません。シーケンスがタブ文字を含むとき、これらの行は判別しづらいものになることがあります。

class difflib.HtmlDiff

このクラスは、二つのテキストを左右に並べて比較表示し、行間あるいは行内の変更点を強調表示するような HTML テーブル (またはテーブルの入った完全な HTML ファイル) を生成するために使います。テーブルは完全差分モード、コンテキスト差分モードのいずれでも生成できます。

このクラスのコンストラクタは以下のようになっています:

__init__(*tabsize=8*, *wrapcolumn=None*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

HtmlDiff のインスタンスを初期化します。

tabsize はオプションのキーワード引数で、タブストップ幅を指定します。デフォルトは 8 です。

wrapcolumn はオプションのキーワード引数で、テキストを折り返すカラム幅を指定します。デフォルトは `None` で折り返しを行いません。

linejunk および *charjunk* はオプションのキーワード引数で、*ndiff()* (*HtmlDiff* はこの関数を使って左右のテキストの差分を HTML で生成します) に渡されます。それぞれの引数のデフォルト値および説明は *ndiff()* のドキュメントを参照してください。

以下のメソッドが public になっています:

make_file(*fromlines*, *tolines*, *fromdesc=''*, *todesc=''*, *context=False*, *numlines=5*, ***, *charset='utf-8'*)

fromlines と *tolines* (いずれも文字列のリスト) を比較し、行間または行内の変更点が強調表示された行差分の入った表を持つ完全な HTML ファイルを文字列で返します。

fromdesc および *todesc* はオプションのキーワード引数で、差分表示テーブルにおけるそれぞれ差分元、差分先ファイルのカラムのヘッダになる文字列を指定します (いずれもデフォルト値は空文字列です)。

context および *numlines* はともにオプションのキーワード引数です。*context* を `True` にするとコンテキスト差分を表示し、デフォルトの `False` にすると完全なファイル差分を表示します。*numlines* のデフォルト値は `5` で、*context* が `True` の場合、*numlines* は強調部分の前後にあるコンテキスト行の数を制御します。*context* が `False` の場合、*numlines* は "next" と書かれたハイパーリンクをたどった時に到達する場所が次の変更部分より何行前にあるかを制御します (値をゼロにした場合、"next" ハイパーリンクを辿ると変更部分の強調表示がブラウザの最上部に表示されるようになります)。

注釈: *fromdesc* と *todesc* はエスケープされていない HTML として解釈されます。信頼できないソースからの入力を受け取る際には適切にエスケープされるべきです。

バージョン 3.5 で変更: *charset* キーワード専用引数が追加されました。HTML 文書のデフォルトの文字集合が `'ISO-8859-1'` から `'utf-8'` に変更されました。

make_table(*fromlines*, *tolines*, *fromdesc=''*, *todesc=''*, *context=False*, *numlines=5*)

fromlines と *tolines* (いずれも文字列のリスト) を比較し、行間または行内の変更点が強調表示された行差分の入った完全な HTML テーブルを文字列で返します。

このメソッドの引数は、*make_file()* メソッドの引数と同じです。

`Tools/scripts/diff.py` はこのクラスへのコマンドラインフロントエンドで、使い方を学ぶ上で格好の例題が入っています。

difflib.context_diff(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)

a と *b* (文字列のリスト) を比較し、差分 (差分形式の行を生成する **ジェネレータ**) を、context diff のフォーマット (以下「コンテキスト形式」) で返します。

コンテキスト形式は、変更があった行に前後数行を加えてある、コンパクトな表現方法です。変更箇所は、変更前/変更後に分けて表します。コンテキスト (変更箇所前後の行) の行数は *n* で指定し、デフォルト値は 3 です。

デフォルトで、diff 制御行 (`***` や `---` を含む行) は改行付きで生成されます。*io.IOBase.readlines()* で作られた入力が *io.IOBase.writelines()* で扱うのに適した diff になるので (なぜなら入力と出力の両方が改行付きのため) 、これは有用です。

行末に改行文字を持たない入力に対しては、出力でも改行文字を付加しないように *lineterm* 引数に `""` を渡してください。

コンテキスト形式は、通常、ヘッダにファイル名と変更時刻を持っています。この情報は、文字列 *fromfile*, *tofile*, *fromfiledate*, *tofiledate* で指定できます。変更時刻の書式は、通常、ISO 8601 フォーマットで表されます。指定しなかった場合のデフォルト値は、空文字列です。

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(context_diff(s1, s2, fromfile='before.py', tofile='after.py'))
*** before.py
```

(次のページに続く)

(前のページからの続き)

```
--- after.py
***************
*** 1,4 ****
! bacon
! eggs
! ham
  guido
--- 1,4 ----
! python
! eggy
! hamster
  guido
```

より詳細な例は、*difflib* **のコマンドラインインタフェース** を参照してください。

difflib.get_close_matches(*word*, *possibilities*, *n=3*, *cutoff=0.6*)

「十分」なマッチの上位のリストを返します。*word* はマッチさせたいシーケンス (大概は文字列) です。*possibilities* は *word* にマッチさせるシーケンスのリスト (大概は文字列のリスト) です。

オプションの引数 *n* (デフォルトでは 3) はメソッドの返すマッチの最大数です。*n* は 0 より大きくなければなりません。

オプションの引数 *cutoff* (デフォルトでは 0.6) は、区間 [0, 1] に入る小数の値です。*word* との一致率がそれ未満の *possibilities* の要素は無視されます。

possibilities の要素でマッチした上位 (多くても *n* 個) は、類似度のスコアに応じて (一番似たものを先頭に) ソートされたリストとして返されます。

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('pineapple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

difflib.ndiff(*a*, *b*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

a と *b* (文字列のリスト) を比較し、差分 (差分形式の行を生成する **ジェネレータ**) を、*Differ* のスタイルで返します。

オプションのキーワード引数 *linejunk* と *charjunk* には、フィルタ関数 (または None) を渡します。

linejunk: 文字列型の引数 1 つを受け取る関数です。文字列が junk の場合は真を、そうでない場合は偽を返します。デフォルトでは None です。モジュールレベルの関数 *IS_LINE_JUNK()* は、高々 1 つのシャープ記号 ('#') を除いて可視の文字を含まない行をフィルタリングするものです。しかし、下層にある *SequenceMatcher* クラスが、どの行が雑音となるほど頻繁に登場するかを動的に分析します。このクラスによる分析は、この関数を使用するよりも通常うまく動作します。

charjunk: 文字 (長さ 1 の文字列) を受け取る関数です。文字列が junk の場合は真を、そうでない場合は偽を返します。デフォルトでは、モジュールレベルの関数 *IS_CHARACTER_JUNK()* であり、これは空白文字類 (空白またはタブ、改行文字をこれに含めてはいけません) をフィルタして排除します。

Tools/scripts/ndiff.py は、この関数のコマンドラインのフロントエンド（インターフェイス）です。

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> print(''.join(diff), end="")
- one
?  ^
+ ore
?  ^
- two
- three
?  -
+ tree
+ emu
```

difflib.**restore**(*sequence*, *which*)

差分を生成した元の二つのシーケンスのうち一つを返します。

Differ.compare() または *ndiff()* によって生成された *sequence* を与えられると、行頭のプレフィクスを取りのぞいてファイル 1 または 2 (引数 *which* で指定される) に由来する行を復元します。

例:

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> diff = list(diff) # materialize the generated delta into a list
>>> print(''.join(restore(diff, 1)), end="")
one
two
three
>>> print(''.join(restore(diff, 2)), end="")
ore
tree
emu
```

difflib.**unified_diff**(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)

a と *b* (文字列のリスト) を比較し、差分 (差分形式の行を生成する ジェネレータ) を、unified diff フォーマット (以下「ユニファイド形式」) で返します。

ユニファイド形式は変更があった行にコンテキストとなる前後数行を加えた、コンパクトな表現方法です。変更箇所は (変更前/変更後を分離したブロックではなく) インラインスタイルで表されます。コンテキストの行数は、*n* で指定し、デフォルト値は 3 です。

デフォルトで、diff 制御行 (---, +++, @@ を含む行) は改行付きで生成されます。*io.IOBase.readlines()* で作られた入力が *io.IOBase.writelines()* で扱うのに適した diff になるので (なぜ

なら入力と出力の両方が改行付きのため) 、これは有用です。

行末に改行文字を持たない入力に対しては、出力でも改行文字を付加しないように *lineterm* 引数に "" を渡してください。

コンテキスト形式は、通常、ヘッダにファイル名と変更時刻を持っています。この情報は、文字列 *fromfile*, *tofile*, *fromfiledate*, *tofiledate* で指定できます。変更時刻の書式は、通常、ISO 8601 フォーマットで表されます。指定しなかった場合のデフォルト値は、空文字列です。

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(unified_diff(s1, s2, fromfile='before.py', tofile='after.py'))
--- before.py
+++ after.py
@@ -1,4 +1,4 @@
-bacon
-eggs
-ham
+python
+eggy
+hamster
 guido
```

より詳細な例は、*difflib* **のコマンドラインインタフェース** を参照してください。

difflib.**diff_bytes**(*dfunc*, *a*, *b*, *fromfile=b''*, *tofile=b''*, *fromfiledate=b''*, *tofiledate=b''*, *n=3*, *lineterm=b'\n'*)

dfunc を使用して *a* と *b* (bytes オブジェクトのリスト) を比較して、差分形式の行 (これも bytes オブジェクトです) を*dfunc* の戻り値の形式で返します。*dfunc* は、呼び出し可能である必要があります。一般に、これは *unified_diff()* または *context_diff()* です。

未知のエンコーディングまたは一貫性のないエンコーディングのデータ同士を比較できます。*n* 以外のすべての入力は、bytes オブジェクトである必要があります。*n* 以外のすべての入力を損失なく str に変換して、`dfunc(a, b, fromfile, tofile, fromfiledate, tofiledate, n, lineterm)` を呼び出すことにより動作します。*dfunc* の出力は、bytes 型に変換されます。これにより、受け取る差分形式の行のエンコーディングは、*a* と *b* の未知または一貫性のないエンコーディングと同一になります。

バージョン 3.5 で追加.

difflib.**IS_LINE_JUNK**(*line*)

無視できる行のとき `True` を返します。行 *line* は空白、または `'#'` ひとつのときに無視できます。それ以外のときには無視できません。古いバージョンでは *ndiff()* の引数 *linejunk* にデフォルトで使用されました。

difflib.**IS_CHARACTER_JUNK**(*ch*)

無視できる文字のとき `True` を返します。文字 *ch* が空白、またはタブ文字のときには無視できます。それ以外の時には無視できません。*ndiff()* の引数 *charjunk* としてデフォルトで使用されます。

参考:

Pattern Matching: The Gestalt Approach　John W. Ratcliff と D. E. Metzener による類似のアルゴリズム

に関する議論。Dr. Dobb's Journal 1988 年 7 月号掲載。

6.3.1 SequenceMatcher オブジェクト

SequenceMatcher クラスには、以下のようなコンストラクタがあります:

class difflib.SequenceMatcher(*isjunk=None*, *a=''*, *b=''*, *autojunk=True*)

オプションの引数 *isjunk* は、None (デフォルトの値です) にするか、単一の引数をとる関数でなければなりません。後者の場合、関数はシーケンスの要素を受け取り、要素が junk であり、無視すべきである場合に限り真を返すようにしなければなりません。*isjunk* に None を渡すと、`lambda x: False` を渡したのと同じになります; すなわち、いかなる要素も無視しなくなります。例えば以下のような引数を渡すと:

```
lambda x: x in " \t"
```

空白とタブ文字を無視して文字のシーケンスを比較します。

オプションの引数 *a* と *b* は、比較される文字列で、デフォルトでは空の文字列です。両方のシーケンスの要素は、**ハッシュ可能** である必要があります。

オプションの引数 *autojunk* は、自動 junk ヒューリスティックを無効にするために使えます。

バージョン 3.2 で追加: *autojunk* パラメータ。

SequenceMatcher オブジェクトは 3 つのデータ属性を持っています: *bjunk* は、*isjunk* が True であるような *b* の要素の集合です; *bpopular* は、(無効でなければ) ヒューリスティックによって popular であると考えられる非ジャンク要素の集合です; *b2j* は、*b* の残りの要素をそれらが生じる位置のリストに写像する dict です。この 3 つは *set_seqs()* または *set_seq2()* で *b* がリセットされる場合は常にリセットされます。

バージョン 3.2 で追加: *bjunk* および *bpopular* 属性。

SequenceMatcher オブジェクトは以下のメソッドを持ちます:

set_seqs(*a*, *b*)

比較される 2 つの文字列を設定します。

SequenceMatcher オブジェクトは、2 つ目のシーケンスについての詳細な情報を計算し、キャッシュします。1 つのシーケンスをいくつものシーケンスと比較する場合、まず *set_seq2()* を使って文字列を設定しておき、別の文字列を 1 つずつ比較するために、繰り返し *set_seq1()* を呼び出します。

set_seq1(*a*)

比較を行う 1 つ目のシーケンスを設定します。比較される 2 つ目のシーケンスは変更されません。

set_seq2(*b*)

比較を行う 2 つ目のシーケンスを設定します。比較される 1 つ目のシーケンスは変更されません。

find_longest_match(*alo*, *ahi*, *blo*, *bhi*)

`a[alo:ahi]` と `b[blo:bhi]` の中から、最長のマッチ列を探します。

isjunk が省略されたか None の時、*find_longest_match()* は a[i:i+k] が b[j:j+k] と等しいような (i, j, k) を返します。その値は alo <= i <= i+k <= ahi かつ blo <= j <= j+k <= bhi となります。(i', j', k') でも、同じようになります。さらに k >= k', i <= i' が i == i', j <= j' でも同様です。言い換えると、いくつものマッチ列すべてのうち、*a* 内で最初に始まるものを返します。そしてその *a* 内で最初のマッチ列すべてのうち *b* 内で最初に始まるものを返します。

```
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

引数 *isjunk* が与えられている場合、上記の通り、はじめに最長のマッチ列を判定します。ブロック内に junk 要素が見当たらないような追加条件の際はこれに該当しません。次にそのマッチ列を、その両側の junk 要素にマッチするよう、できる限り広げていきます。そのため結果となる列は、探している列のたまたま直前にあった同一の junk 以外の junk にはマッチしません。

以下は前と同じサンプルですが、空白を junk とみなしています。これは ' abcd' が 2 つ目の列の末尾にある ' abcd' にマッチしないようにしています。代わりに 'abcd' にはマッチします。そして 2 つ目の文字列中、一番左の 'abcd' にマッチします:

```
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

どんな列にもマッチしない時は、(alo, blo, 0) を返します。

このメソッドは *named tuple* Match(a, b, size) を返します。

get_matching_blocks()

マッチした互いに重複の無いシーケンスを表す、3 つ組の値のリストを返します。それぞれの値は (i, j, n) という形式で表され、a[i:i+n] == b[j:j+n] という関係を意味します。3 つの値は *i* と *j* の間で単調に増加します。

最後の 3 つ組はダミーで、(len(a), len(b), 0) という値を持ちます。これは n == 0 である唯一のタプルです。もし (i, j, n) と (i', j', n') がリストで並んでいる 3 つ組で、2 つ目が最後の 3 つ組でなければ、i+n < i' または j+n < j' です。言い換えると並んでいる 3 つ組は常に隣接していない同じブロックを表しています。

```
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2), Match(a=5, b=4, size=0)]
```

get_opcodes()

a を *b* にするための方法を記述する 5 つのタプルを返します。それぞれのタプルは (tag, i1, i2, j1, j2) という形式であらわされます。最初のタプルは i1 == j1 == 0 であり、*i1* はその前にあるタプルの *i2* と同じ値です。同様に *j1* は前の *j2* と同じ値になります。

tag の値は文字列であり、次のような意味です:

値	意味
'replace'	a[i1:i2] は b[j1:j2] に置き換えられる。
'delete'	a[i1:i2] は削除される。この時、j1 == j2 である。
'insert'	b[j1:j2] が a[i1:i1] に挿入される。この時 i1 == i2 である。
'equal'	a[i1:i2] == b[j1:j2] (サブシーケンスは等しい).

例えば:

```
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...     print('{:7}   a[{}:{}] --> b[{}:{}] {!r:>8} --> {!r}'.format(
...         tag, i1, i2, j1, j2, a[i1:i2], b[j1:j2]))
delete    a[0:1] --> b[0:0]      'q' --> ''
equal     a[1:3] --> b[0:2]     'ab' --> 'ab'
replace   a[3:4] --> b[2:3]      'x' --> 'y'
equal     a[4:6] --> b[3:5]     'cd' --> 'cd'
insert    a[6:6] --> b[5:6]       '' --> 'f'
```

get_grouped_opcodes(*n=3*)

最大 *n* 行までのコンテキストを含むグループを生成するような、**ジェネレータ** を返します。

このメソッドは、*get_opcodes()* で返されるグループの中から、似たような差異のかたまりに分け、間に挟まっている変更の無い部分を省きます。

グループは *get_opcodes()* と同じ書式で返されます。

ratio()

[0, 1] の範囲の浮動小数点数で、シーケンスの類似度を測る値を返します。

T が 2 つのシーケンスの要素数の総計だと仮定し、M をマッチした数とすると、この値は 2.0*M / T であらわされます。もしシーケンスがまったく同じ場合、値は 1.0 となり、まったく異なる場合には 0.0 となります。

このメソッドは *get_matching_blocks()* または *get_opcodes()* がまだ呼び出されていない場合には非常にコストが高いです。この場合、上限を素早く計算するために、*quick_ratio()* もしくは *real_quick_ratio()* を最初に試してみる方がいいかもしれません。

注釈: 注意: *ratio()* の呼び出しの結果は引数の順序に依存します。例えば次の通りです:

```
>>> SequenceMatcher(None, 'tide', 'diet').ratio()
0.25
>>> SequenceMatcher(None, 'diet', 'tide').ratio()
0.5
```

---

quick_ratio()
: *ratio()* の上界を、より高速に計算します。

real_quick_ratio()
: *ratio()* の上界を、非常に高速に計算します。

この文字列全体のマッチ率を返す 3 つのメソッドは、精度の異なる近似値を返します。quick_ratio() と real_quick_ratio() は、常に ratio() 以上の値を返します:

```
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

6.3.2 SequenceMatcher の例

この例は 2 つの文字列を比較します。空白を "junk" とします:

```
>>> s = SequenceMatcher(lambda x: x == " ",
...                     "private Thread currentThread;",
...                     "private volatile Thread currentThread;")
```

ratio() は、$[0, 1]$ の範囲の値を返し、シーケンスの類似度を測ります。経験によると、ratio() の値が 0.6 を超えると、シーケンスがよく似ていることを示します:

```
>>> print(round(s.ratio(), 3))
0.866
```

シーケンスのどこがマッチしているかにだけ興味のある時には get_matching_blocks() が手軽でしょう:

```
>>> for block in s.get_matching_blocks():
...     print("a[%d] and b[%d] match for %d elements" % block)
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 21 elements
a[29] and b[38] match for 0 elements
```

get_matching_blocks() が返す最後のタプルが常にダミーであることに注目してください。このダミーは (len(a), len(b), 0) であり、これはタプルの最後の要素（マッチする要素の数）が 0 となる唯一のケースです。

はじめのシーケンスがどのようにして 2 番目のものになるのかを知るには、get_opcodes() を使います:

```
>>> for opcode in s.get_opcodes():
...     print("%6s a[%d:%d] b[%d:%d]" % opcode)
 equal a[0:8] b[0:8]
```

(次のページに続く)

(前のページからの続き)

```
insert a[8:8] b[8:17]
 equal a[8:29] b[17:38]
```

参考:

- *SequenceMatcher* を使った、シンプルで使えるコードを知るには、このモジュールの関数 *get_close_matches()* を参照してください。
- Simple version control recipe *SequenceMatcher* で作った小規模アプリケーション。

6.3.3 Differ オブジェクト

Differ オブジェクトによって生成された差分が **最小** であるなどとは言いません。むしろ、最小の差分はしばしば直観に反しています。その理由は、どこでもできるとなれば一致を見いだしてしまうからで、ときには思いがけなく 100 ページも離れたマッチになってしまうのです。一致点を互いに隣接したマッチに制限することで、場合によって長めの差分を出力するというコストを掛けることにはなっても、ある種の局所性を保つことができるのです。

Differ は、以下のようなコンストラクタを持ちます:

class difflib.Differ(*linejunk=None*, *charjunk=None*)

オプションのキーワードパラメータ *linejunk* と *charjunk* は、フィルタ関数を渡します (使わないときは None):

linejunk: ひとつの文字列引数を受け取る関数です。文字列が junk のときに真を返します。デフォルトでは、None であり、どんな行であっても junk とは見なされません。

charjunk: この関数は文字 (長さ 1 の文字列) を引数として受け取り、文字が junk であるときに真を返します。デフォルトは None であり、どんな文字も junk とは見なされません。

これらの junk フィルター関数により、差分を発見するマッチングが高速化し、差分の行や文字が無視されることがなくなります。説明については、*find_longest_match()* メソッドの *isjunk* 引数の説明をご覧ください。

Differ オブジェクトは、以下の 1 つのメソッドを通して利用されます。(差分を生成します):

compare(*a*, *b*)

文字列からなる 2 つのシーケンスを比較し、差分 (を表す文字列からなるシーケンス) を生成します。

各シーケンスの要素は、改行で終わる独立した単一行からなる文字列でなければなりません。そのようなシーケンスは、ファイル風オブジェクトの *readlines()* メソッドによって得ることができます。(得られる) 差分は改行文字で終了する文字列のシーケンスとして得られ、ファイル風オブジェクトの *writelines()* メソッドによって出力できる形になっています。

6.3.4 Differ の例

以下の例は 2 つのテキストを比較しています。最初に、テキストを行毎に改行で終わる文字列のシーケンスにセットアップします (そのようなシーケンスは、ファイル風オブジェクトの `readlines()` メソッドからも得ることができます):

```
>>> text1 = '''  1. Beautiful is better than ugly.
...   2. Explicit is better than implicit.
...   3. Simple is better than complex.
...   4. Complex is better than complicated.
... '''.splitlines(keepends=True)
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = '''  1. Beautiful is better than ugly.
...   3.   Simple is better than complex.
...   4. Complicated is better than complex.
...   5. Flat is better than nested.
... '''.splitlines(keepends=True)
```

次に Differ オブジェクトをインスタンス化します:

```
>>> d = Differ()
```

注意: *Differ* オブジェクトをインスタンス化するとき、行 junk と文字 junk をフィルタリングする関数を渡すことができます。詳細は *Differ()* コンストラクタを参照してください。

最後に、2 つを比較します:

```
>>> result = list(d.compare(text1, text2))
```

`result` は文字列のリストなので、pretty-print してみましょう:

```
>>> from pprint import pprint
>>> pprint(result)
['    1. Beautiful is better than ugly.\n',
 '-   2. Explicit is better than implicit.\n',
 '-   3. Simple is better than complex.\n',
 '+   3.   Simple is better than complex.\n',
 '?     ++\n',
 '-   4. Complex is better than complicated.\n',
 '?            ^                     ---- ^\n',
 '+   4. Complicated is better than complex.\n',
 '?          ++++ ^                     ^\n',
 '+   5. Flat is better than nested.\n']
```

これは、複数行の文字列として、次のように出力されます:

```
>>> import sys
>>> sys.stdout.writelines(result)
    1. Beautiful is better than ugly.
-   2. Explicit is better than implicit.
-   3. Simple is better than complex.
+   3.   Simple is better than complex.
?     ++
-   4. Complex is better than complicated.
?            ^                     ---- ^
+   4. Complicated is better than complex.
?          ++++ ^                      ^
+   5. Flat is better than nested.
```

6.3.5 difflib のコマンドラインインタフェース

この例は、difflib を使って `diff` に似たユーティリティーを作成する方法を示します。これは、Python のソース配布物にも、`Tools/scripts/diff.py` として含まれています。

```
#!/usr/bin/env python3
""" Command line interface to difflib.py providing diffs in four formats:

* ndiff:    lists every line and highlights interline changes.
* context:  highlights clusters of changes in a before/after format.
* unified:  highlights clusters of changes in an inline format.
* html:     generates side by side comparison with change highlights.

"""

import sys, os, difflib, argparse
from datetime import datetime, timezone

def file_mtime(path):
    t = datetime.fromtimestamp(os.stat(path).st_mtime,
                               timezone.utc)
    return t.astimezone().isoformat()

def main():

    parser = argparse.ArgumentParser()
    parser.add_argument('-c', action='store_true', default=False,
                        help='Produce a context format diff (default)')
    parser.add_argument('-u', action='store_true', default=False,
                        help='Produce a unified format diff')
    parser.add_argument('-m', action='store_true', default=False,
                        help='Produce HTML side by side diff '
                             '(can use -c and -l in conjunction)')
    parser.add_argument('-n', action='store_true', default=False,
                        help='Produce a ndiff format diff')
    parser.add_argument('-l', '--lines', type=int, default=3,
                        help='Set number of context lines (default 3)')
```

(次のページに続く)

(前のページからの続き)

```
    parser.add_argument('fromfile')
    parser.add_argument('tofile')
    options = parser.parse_args()

    n = options.lines
    fromfile = options.fromfile
    tofile = options.tofile

    fromdate = file_mtime(fromfile)
    todate = file_mtime(tofile)
    with open(fromfile) as ff:
        fromlines = ff.readlines()
    with open(tofile) as tf:
        tolines = tf.readlines()

    if options.u:
        diff = difflib.unified_diff(fromlines, tolines, fromfile, tofile, fromdate, todate,␣
↪n=n)
    elif options.n:
        diff = difflib.ndiff(fromlines, tolines)
    elif options.m:
        diff = difflib.HtmlDiff().make_file(fromlines,tolines,fromfile,tofile,context=options.
↪c,numlines=n)
    else:
        diff = difflib.context_diff(fromlines, tolines, fromfile, tofile, fromdate, todate,␣
↪n=n)

    sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()
```

6.4 textwrap --- テキストの折り返しと詰め込み

ソースコード: Lib/textwrap.py

---

textwrap モジュールは、実際の処理を行う *TextWrapper* とともに、いくつかの便利な関数を提供しています。1 つか 2 つの文字列を wrap あるいは fill するだけの場合は便利関数で十分ですが、多くの処理を行う場合は効率のために *TextWrapper* のインスタンスを使うべきでしょう。

textwrap.wrap(*text*, *width=70*, ***kwargs*)

text (文字列) 内の段落を一つだけ折り返しを行います。したがって、すべての行が高々 *width* 文字の長さになります。最後に改行が付かない出力行のリストを返します。

オプションのキーワード引数は、以下で説明する *TextWrapper* のインスタンス属性に対応しています。*width* はデフォルトで 70 です。

wrap() の動作についての詳細は *TextWrapper.wrap()* メソッドを参照してください。

textwrap.fill(*text*, *width=70*, ***kwargs*)

text 内の段落を一つだけ折り返しを行い、折り返しが行われた段落を含む一つの文字列を返します。*fill()* はこれの省略表現です

```
"\n".join(wrap(text, ...))
```

特に、*fill()* は *wrap()* とまったく同じ名前のキーワード引数を受け取ります。

textwrap.shorten(*text*, *width*, ***kwargs*)

与えられた *text* を折りたたみ、切り詰めて、与えられた *width* に収まるようにします。

最初に、*text* 内の空白が折りたたまれます (すべての空白を、1 文字の空白文字で置き換えます)。結果が *width* 内に収まった場合、その結果が返されます。width に収まらない場合、残りの文字数と placeholder との和が width 内に収まるように、末尾から単語が切り捨てられます:

```
>>> textwrap.shorten("Hello  world!", width=12)
'Hello world!'
>>> textwrap.shorten("Hello  world!", width=11)
'Hello [...]'
>>> textwrap.shorten("Hello world", width=10, placeholder="...")
'Hello...'
```

オプションのキーワード引数は、以下で説明する *TextWrapper* インスタンスの属性に対応します。文字列が *TextWrapper* の *fill()* 関数に渡される前に、空白が折りたたまれます。そのため、*tabsize*、*expand_tabs*、*drop_whitespace*、*replace_whitespace* の値を変更しても、意味がありません。

バージョン 3.4 で追加.

textwrap.dedent(*text*)

text の各行に対し、共通して現れる先頭の空白を削除します。

この関数は通常、三重引用符で囲われた文字列をスクリーン/その他の左端にそろえ、なおかつソースコード中ではインデントされた形式を損なわないようにするために使われます。

タブとスペースはともにホワイトスペースとして扱われますが、同じではないことに注意してください: " hello" という行と "\thello" は、同じ先頭の空白文字をもっていないとみなされます。

空白文字しか含まない行は入力の際に無視され、出力の際に単一の改行文字に正規化されます。

例えば:

```
def test():
    # end first line with \ to avoid the empty line!
    s = '''\
    hello
      world
    '''
    print(repr(s))          # prints '    hello\n      world\n    '
    print(repr(dedent(s)))  # prints 'hello\n  world\n'
```

textwrap.**indent**(*text*, *prefix*, *predicate=None*)

text の中の選択された行の先頭に *prefix* を追加します。

行の分割は `text.splitlines(True)` で行います。

デフォルトでは、(改行文字を含む) 空白文字だけの行を除いてすべての行に *prefix* を追加します。

例えば:

```
>>> s = 'hello\n\n \nworld'
>>> indent(s, '  ')
'  hello\n\n \n  world'
```

省略可能な *predicate* 引数を使って、どの行をインデントするかを制御することができます。例えば、空行や空白文字のみの行にも *prefix* を追加するのは簡単です:

```
>>> print(indent(s, '+ ', lambda line: True))
+ hello
+
+
+ world
```

バージョン 3.3 で追加.

wrap()、*fill()*、*shorten()* は *TextWrapper* インスタンスを作成し、その一つのメソッドを呼び出すことで機能します。そのインスタンスは再利用されません。したがって、*wrap()* や *fill()* を使用して多くのテキスト文字列を処理するアプリケーションについては、独自の *TextWrapper* オブジェクトを作成する方が効率が良い方法でしょう。

テキストはなるべく空白か、ハイフンを含む語のハイフンの直後で折り返されます。*TextWrapper.break_long_words* が偽に設定されていなければ、必要な場合に長い語が分解されます。

class textwrap.**TextWrapper**(***kwargs*)

TextWrapper コンストラクタはたくさんのオプションのキーワード引数を受け取ります。それぞれのキーワード引数は一つのインスタンス属性に対応します。したがって、例えば

```
wrapper = TextWrapper(initial_indent="* ")
```

はこれと同じです

```
wrapper = TextWrapper()
wrapper.initial_indent = "* "
```

あなたは同じ *TextWrapper* オブジェクトを何回も再利用できます。また、使用中にインスタンス属性へ代入することでそのオプションのどれでも変更できます。

TextWrapper インスタンス属性 (とコンストラクタのキーワード引数) は以下の通りです:

width

(デフォルト: 70) 折り返しが行われる行の最大の長さ。入力行に *width* より長い単一の語が無い限り、*TextWrapper* は *width* 文字より長い出力行が無いことを保証します。

expand_tabs

(デフォルト: True) もし真ならば、そのときは *text* 内のすべてのタブ文字は *text* の expandtabs() メソッドを用いて空白に展開されます。

tabsize

(デフォルト: 8) *expand_tabs* が真の場合、*text* の中のすべての TAB 文字は tabsize と現在のカラムに応じて、ゼロ以上のスペースに展開されます。

バージョン 3.3 で追加.

replace_whitespace

(デフォルト: True) 真の場合、*wrap()* メソッドはタブの展開の後、wrap 処理の前に各種空白文字をスペース 1 文字に置換します。置換される空白文字は: TAB, 改行, 垂直 TAB, FF, CR ('\t\n\v\f\r') です。

注釈: *expand_tabs* が偽で *replace_whitespace* が真ならば、各タブ文字は 1 つの空白に置き換えられます。それはタブ展開と同じでは **ありません** 。

注釈: *replace_whitespace* が偽の場合、改行が行の途中で現れることで出力がおかしくなることがあります。このため、テキストを (*str.splitlines()* などを使って) 段落ごとに分けて別々に wrap する必要があります。

---

drop_whitespace

(デフォルト: True) 真の場合、(wrap 処理のあとインデント処理の前に) 各行の最初と最後の空白文字を削除します。ただし、段落の最初の空白については、次の文字が空白文字でない場合は削除されません。削除される空白文字が行全体に及ぶ場合は、行自体を削除します。

initial_indent

(default: '') wrap の出力の最初の行の先頭に付与する文字列。最初の行の長さに加算されます。空文字列の場合インデントされません。

subsequent_indent

(デフォルト: '') 一行目以外の折り返しが行われる出力のすべての行の先頭に付けられる文字列。一行目以外の各行の折り返しまでの長さにカウントされます。

fix_sentence_endings

(デフォルト: False) もし真ならば、*TextWrapper* は文の終わりを見つけようとし、確実に文がちょうど二つの空白で常に区切られているようにします。これは一般的に固定スペースフォントのテキストに対して望ましいです。しかし、文の検出アルゴリズムは完全ではありません: 文の終わりには、後ろに空白がある '.', '!' または '?' の中の一つ、ことによると '"' あるいは "'" が付随する小文字があると仮定しています。これに伴う一つの問題はアルゴリズムで下記の”Dr.” と

```
[...] Dr. Frankenstein's monster [...]
```

下記の”Spot.” の間の差異を検出できないことです

```
[...] See Spot. See Spot run [...]
```

fix_sentence_endings はデフォルトで偽です。

文検出アルゴリズムは” 小文字” の定義のために `string.lowercase` に依存し、同一行の文を区切るためにピリオドの後に二つの空白を使う慣習に依存しているため、英文テキストに限定されたものです。

break_long_words
(デフォルト: `True`) もし真ならば、そのとき *width* より長い行が確実にないようにするために、*width* より長い語は切られます。偽ならば、長い語は切られないでしょう。そして、*width* より長い行があるかもしれません。(*width* を超える分を最小にするために、長い語は単独で一行に置かれるでしょう。)

break_on_hyphens
(デフォルト: `True`) 真の場合、英語で一般的なように、ラップ処理は空白か合成語に含まれるハイフンの直後で行われます。偽の場合、空白だけが改行に適した位置として判断されます。ただし、本当に語の途中で改行が行われないようにするためには、*break_long_words* 属性を真に設定する必要があります。過去のバージョンでのデフォルトの振る舞いは、常にハイフンの直後での改行を許していました。

max_lines
(デフォルト `None`) `None` 以外の場合、出力は行数 *max_lines* を超えないようにされ、切り詰める際には出力の最後の行を *placeholder* に置き換えます。

バージョン 3.4 で追加.

placeholder
(デフォルト: `' [...]'`) 切り詰める場合に出力の最後の行に置く文字列です。

バージョン 3.4 で追加.

TextWrapper はモジュールレベルの簡易関数に類似したいくつかの公開メソッドも提供します:

wrap(*text*)
1 段落の文字列 *text* を、各行が *width* 文字以下になるように wrap します。wrap のすべてのオプションは *TextWrapper* インスタンスの属性から取得します。結果の、行末に改行のない行のリストを返します。出力の内容が空になる場合は、返すリストも空になります。

fill(*text*)
text 内の段落を一つだけ折り返しを行い、折り返しが行われた段落を含む一つの文字列を返します。

6.5 unicodedata --- Unicode データベース

This module provides access to the Unicode Character Database (UCD) which defines character properties for all Unicode characters. The data contained in this database is compiled from the UCD version 12.1.0.

このモジュールは、ユニコード標準付録 #44「 ユニコード文字データベース 」で定義されているのと同じ名前およびシンボルを使用します。このモジュールは次のような関数を定義します:

unicodedata.lookup(*name*)

名前に対応する文字を探します。その名前の文字が見つかった場合、その文字が返されます。見つからなかった場合には、*KeyError* を発生させます。

バージョン 3.3 で変更: name aliases[*1] と named sequences[*2] のサポートが追加されました。

unicodedata.name(*chr*[, *default*])

文字 *chr* に付いている名前を、文字列で返します。名前が定義されていない場合には *default* が返されますが、この引数が与えられていなければ *ValueError* を発生させます。

unicodedata.decimal(*chr*[, *default*])

文字 *chr* に割り当てられている十進数を、整数で返します。この値が定義されていない場合には *default* が返されますが、この引数が与えられていなければ *ValueError* を発生させます。

unicodedata.digit(*chr*[, *default*])

文字 *chr* に割り当てられている数値を、整数で返します。この値が定義されていない場合には *default* が返されますが、この引数が与えられていなければ *ValueError* を発生させます。

unicodedata.numeric(*chr*[, *default*])

文字 *chr* に割り当てられている数値を、float で返します。この値が定義されていない場合には *default* が返されますが、この引数が与えられていなければ *ValueError* を発生させます。

unicodedata.category(*chr*)

文字 *chr* に割り当てられた、汎用カテゴリを返します。

unicodedata.bidirectional(*chr*)

文字 *chr* に割り当てられた、双方向クラスを返します。そのような値が定義されていない場合、空の文字列が返されます。

unicodedata.combining(*chr*)

文字 *chr* に割り当てられた正規結合クラスを返します。結合クラス定義されていない場合、0 が返されます。

unicodedata.east_asian_width(*chr*)

ユニコード文字 *chr* に割り当てられた east asian width を文字列で返します。

[*1] http://www.unicode.org/Public/12.1.0/ucd/NameAliases.txt
[*2] http://www.unicode.org/Public/12.1.0/ucd/NamedSequences.txt

unicodedata.**mirrored**(*chr*)

文字 *chr* に割り当てられた、鏡像化のプロパティを返します。その文字が双方向テキスト内で鏡像化された文字である場合には 1 を、それ以外の場合には 0 を返します。

unicodedata.**decomposition**(*chr*)

文字 *chr* に割り当てられた、文字分解マッピングを、文字列型で返します。そのようなマッピングが定義されていない場合、空の文字列が返されます。

unicodedata.**normalize**(*form*, *unistr*)

Unicode 文字列 *unistr* の正規形 *form* を返します。*form* の有効な値は、'NFC'、'NFKC'、'NFD'、'NFKD' です。

Unicode 規格は標準等価性 (canonical equivalence) と互換等価性 (compatibility equivalence) に基づいて、様々な Unicode 文字列の正規形を定義します。Unicode では、複数の方法で表現できる文字があります。たとえば、文字 U+00C7 (LATIN CAPITAL LETTER C WITH CEDILLA) は、U+0043 (LATIN CAPITAL LETTER C) U+0327 (COMBINING CEDILLA) というシーケンスとしても表現できます。

各文字には 2 つの正規形があり、それぞれ正規形 C と正規形 D といいます。正規形 D (NFD) は標準分解 (canonical decomposition) としても知られており、各文字を分解された形に変換します。正規形 C (NFC) は標準分解を適用した後、結合済文字を再構成します。

互換等価性に基づいて、2 つの正規形が加えられています。Unicode では、一般に他の文字との統合がサポートされている文字があります。たとえば、U+2160 (ROMAN NUMERAL ONE) は事実上 U+0049 (LATIN CAPITAL LETTER I) と同じものです。しかし、Unicode では、既存の文字集合 (たとえば gb2312) との互換性のために、これがサポートされています。

正規形 KD (NFKD) は、互換分解 (compatibility decomposition) を適用します。すなわち、すべての互換文字を、等価な文字で置換します。正規形 KC (NFKC) は、互換分解を適用してから、標準分解を適用します。

2 つの unicode 文字列が正規化されていて人間の目に同じに見えても、片方が結合文字を持っていてもう片方が持っていない場合、それらは完全に同じではありません。

unicodedata.**is_normalized**(*form*, *unistr*)

Unicode 文字列 *unistr* が正規形 *form* かどうかを返します。*form* の有効な値は、'NFC'、'NFKC'、'NFD'、'NFKD' です。

バージョン 3.8 で追加.

更に、本モジュールは以下の定数を公開します:

unicodedata.**unidata_version**

このモジュールで使われている Unicode データベースのバージョン。

unicodedata.**ucd_3_2_0**

これはモジュール全体と同じメソッドを具えたオブジェクトですが、Unicode データベースバージョン 3.2 を代わりに使っており、この特定のバージョンの Unicode データベースを必要とするアプリケーション (IDNA など) のためものです。

例:

```
>>> import unicodedata
>>> unicodedata.lookup('LEFT CURLY BRACKET')
'{'
>>> unicodedata.name('/')
'SOLIDUS'
>>> unicodedata.decimal('9')
9
>>> unicodedata.decimal('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not a decimal
>>> unicodedata.category('A')  # 'L'etter, 'u'ppercase
'Lu'
>>> unicodedata.bidirectional('\u0660') # 'A'rabic, 'N'umber
'AN'
```

脚注

6.6 stringprep --- インターネットのための文字列調製

ソースコード: Lib/stringprep.py

(ホスト名のような) インターネット上にある存在に識別名をつける際、しばしば識別名間の ” 等価性” 比較を行う必要があります。厳密には、例えば大小文字の区別をするかしないかいったように、比較をどのように行うかはアプリケーションの領域に依存します。また、例えば ” 印字可能な” 文字で構成された識別名だけを許可するといったように、可能な識別名を制限することも必要となるかもしれません。

RFC 3454 では、インターネットプロトコル上で Unicode 文字列を ” 調製 (prepare)” するためのプロシジャを定義しています。文字列は通信路に載せられる前に調製プロシジャで処理され、その結果ある正規化された形式になります。RFC ではあるテーブルの集合を定義しており、それらはプロファイルにまとめられています。各プロファイルでは、どのテーブルを使い、stringprep プロシジャのどのオプション部分がプロファイルの一部になっているかを定義しています。stringprep プロファイルの一つの例は nameprep で、国際化されたドメイン名に使われます。

stringprep は **RFC 3454** のテーブルを公開しているに過ぎません。これらのテーブルは辞書やリストとして表現するにはバリエーションが大きすぎるので、このモジュールでは Unicode 文字データベースを内部的に利用しています。モジュールソースコード自体は mkstringprep.py ユーティリティを使って生成されました。

その結果、これらのテーブルはデータ構造体ではなく、関数として公開されています。RFC には 2 種類のテーブル: 集合およびマップ、が存在します。集合については、*stringprep* は ” 特性関数 (characteristic function)” 、すなわち引数が集合の一部である場合に True を返す関数を提供します。マッピングについては、マップ関数: キーが与えられると、それに関連付けられた値を返す関数を提供します。以下はこのモジュールで利用可能な全ての関数を列挙したものです。

stringprep.**in_table_a1**(*code*)
code がテーブル A.1 (Unicode 3.2 における未割り当てコードポイント: unassigned code point) かどうか判定します。

stringprep.**in_table_b1**(*code*)
code がテーブル B.1 (一般には何にも対応付けられていない: commonly mapped to nothing) かどうか判定します。

stringprep.**map_table_b2**(*code*)
テーブル B.2 (NFKC で用いられる大小文字の対応付け) に従って、*code* に対応付けられた値を返します。

stringprep.**map_table_b3**(*code*)
テーブル B.3 (正規化を伴わない大小文字の対応付け) に従って、*code* に対応付けられた値を返します。

stringprep.**in_table_c11**(*code*)
code がテーブル C.1.1 (ASCII スペース文字) かどうか判定します。

stringprep.**in_table_c12**(*code*)
code がテーブル C.1.2 (非 ASCII スペース文字) かどうか判定します。

stringprep.**in_table_c11_c12**(*code*)
code がテーブル C.1 (スペース文字、C.1.1 および C.1.2 の和集合) かどうか判定します。

stringprep.**in_table_c21**(*code*)
code がテーブル C.2.1 (ASCII 制御文字) かどうか判定します。

stringprep.**in_table_c22**(*code*)
code がテーブル C.2.2 (非 ASCII 制御文字) かどうか判定します。

stringprep.**in_table_c21_c22**(*code*)
code がテーブル C.2 (制御文字、C.2.1 および C.2.2 の和集合) かどうか判定します。

stringprep.**in_table_c3**(*code*)
code がテーブル C.3 (プライベート利用) かどうか判定します。

stringprep.**in_table_c4**(*code*)
code がテーブル C.4 (非文字コードポイント: non-character code points) かどうか判定します。

stringprep.**in_table_c5**(*code*)
code がテーブル C.5 (サロゲーションコード) かどうか判定します。

stringprep.**in_table_c6**(*code*)
code がテーブル C.6 (平文:plain text として不適切) かどうか判定します。

stringprep.**in_table_c7**(*code*)
code がテーブル C.7 (標準表現:canonical representation として不適切) かどうか判定します。

stringprep.**in_table_c8**(*code*)
code がテーブル C.8 (表示プロパティの変更または撤廃) かどうか判定します。

stringprep.in_table_c9(*code*)
: *code* がテーブル C.9 (タグ文字) かどうか判定します。

stringprep.in_table_d1(*code*)
: *code* がテーブル D.1 (双方向プロパティ "R" または "AL" を持つ文字) かどうか判定します。

stringprep.in_table_d2(*code*)
: *code* がテーブル D.2 (双方向プロパティ "L" を持つ文字) かどうか判定します。

6.7 readline --- GNU readline のインタフェース

readline モジュールでは、補完や Python インタプリタからの履歴ファイルの読み書きを容易にするための多くの関数を定義しています。このモジュールは直接、または *rlcompleter* モジュールを介して使うことができます。*rlcompleter* モジュールは対話的プロンプトで Python 識別子の補完をサポートするものです。このモジュールで利用される設定は、インタプリタの対話プロンプトならびに組み込みの *input()* 関数の両方の挙動に影響します。

readline のキーバインディングは初期化ファイルで設定できます。このファイルは、たいていはホームディレクトリに `.inputrc` という名前で置いてあります。GNU Readline マニュアルの Readline Init File を参照して、そのファイルの形式や可能な構成、Readline ライブラリ全体の機能を知ってください。

注釈: 下層の Readline ライブラリー API は GNU readline ではなく `libedit` ライブラリーで実装される可能性があります。macOS では *readline* モジュールはどのライブラリーが使われているかを実行時に検出します。

`libedit` の設定ファイルは GNU readline のものとは異なります。もし設定文字列をプログラムからロードしているなら、GNU readline と libedit を区別するために "libedit" という文字列が `readline.__doc__` に含まれているかどうかチェックしてください。

If you use *editline*/`libedit` readline emulation on macOS, the initialization file located in your home directory is named `.editrc`. For example, the following content in `~/.editrc` will turn ON *vi* keybindings and TAB completion:

```
python:bind -v
python:bind ^I rl_complete
```

6.7.1 初期化ファイル

以下の関数は初期化ファイルならびにユーザ設定関連のものです:

readline.parse_and_bind(*string*)
: *string* 引数で渡された最初の行を実行します。これにより下層のライブラリーの rl_parse_and_bind() が呼ばれます。

readline.read_init_file([*filename*])
: readline 初期化ファイルを実行します。デフォルトのファイル名は最後に使用されたファイル名です。これにより下層のライブラリーの rl_read_init_file() が呼ばれます。

6.7.2 行バッファ

以下の関数は行バッファを操作します:

readline.get_line_buffer()
: 行バッファ (下層のライブラリーの rl_line_buffer) の現在の内容を返します。

readline.insert_text(*string*)
: テキストをカーサー位置の行バッファに挿入します。これにより下層のライブラリーの rl_insert_text() が呼ばれますが、戻り値は無視されます。

readline.redisplay()
: スクリーンの表示を変更して行バッファの現在の内容を反映させます。これにより下層のライブラリーの rl_redisplay() が呼ばれます。

6.7.3 履歴ファイル

以下の関数は履歴ファイルを操作します:

readline.read_history_file([*filename*])
: readline 履歴ファイルを読み込み、履歴リストに追加します。デフォルトのファイル名は ~/.history です。これにより下層のライブラリーの read_history() が呼ばれます。

readline.write_history_file([*filename*])
: 履歴リストを readline 履歴ファイルに保存します。既存のファイルは上書きされます。デフォルトのファイル名は ~/.history です。これにより下層のライブラリーの write_history() が呼ばれます。

readline.append_history_file(*nelements*[, *filename*])
: 履歴の最後の *nelements* 項目をファイルに追加します。でふぉるのファイル名は ~/.history です。ファイルは存在していなくてはなりません。これにより下層のライブラリーの append_history() が呼ばれます。Python がこの機能をサポートするライブラリーのバージョンでコンパイルされたときのみ、この関数は存在します。

  バージョン 3.5 で追加.

readline.**get_history_length**()

readline.**set_history_length**(*length*)

Set or return the desired number of lines to save in the history file. The *write_history_file()* function uses this value to truncate the history file, by calling `history_truncate_file()` in the underlying library. Negative values imply unlimited history file size.

6.7.4 履歴リスト

以下の関数はグローバルな履歴リストを操作します:

readline.**clear_history**()

現在の履歴をクリアします。これにより下層のライブラリーの `clear_history()` が呼ばれます。Python がこの機能をサポートするライブラリーのバージョンでコンパイルされたときのみ、この関数は存在します。

readline.**get_current_history_length**()

履歴に現在ある項目の数を返します。(*get_history_length()* は履歴ファイルに書かれる最大行数を返します。)

readline.**get_history_item**(*index*)

現在の履歴の *index* 番目の項目を返します。添字は 1 から始まります。これにより下層のライブラリーの `history_get()` が呼ばれます。

readline.**remove_history_item**(*pos*)

履歴から指定された位置の項目を削除します。添字は 0 から始まります。これにより下層のライブラリーの `remove_history()` が呼ばれます。

readline.**replace_history_item**(*pos*, *line*)

指定された位置の項目を *line* で置き換えます。添字は 0 から始まります。これにより下層のライブラリーの `replace_history_entry()` が呼ばれます。

readline.**add_history**(*line*)

最後に入力したかのように、*line* を履歴バッファに追加します。これにより下層のライブラリーの `add_history()` が呼ばれます。

readline.**set_auto_history**(*enabled*)

Enable or disable automatic calls to `add_history()` when reading input via readline. The *enabled* argument should be a Boolean value that when true, enables auto history, and that when false, disables auto history.

バージョン 3.6 で追加.

CPython implementation detail: Auto history is enabled by default, and changes to this do not persist across multiple sessions.

6.7.5 開始フック

readline.set_startup_hook([*function*])

Set or remove the function invoked by the `rl_startup_hook` callback of the underlying library. If *function* is specified, it will be used as the new hook function; if omitted or `None`, any function already installed is removed. The hook is called with no arguments just before readline prints the first prompt.

readline.set_pre_input_hook([*function*])

Set or remove the function invoked by the `rl_pre_input_hook` callback of the underlying library. If *function* is specified, it will be used as the new hook function; if omitted or `None`, any function already installed is removed. The hook is called with no arguments after the first prompt has been printed and just before readline starts reading input characters. This function only exists if Python was compiled for a version of the library that supports it.

6.7.6 補完

The following functions relate to implementing a custom word completion function. This is typically operated by the Tab key, and can suggest and automatically complete a word being typed. By default, Readline is set up to be used by *rlcompleter* to complete Python identifiers for the interactive interpreter. If the *readline* module is to be used with a custom completer, a different set of word delimiters should be set.

readline.set_completer([*function*])

completer 関数を設定または削除します。*function* が指定された場合、新たな completer 関数として用いられます; 省略された場合や `None` の場合、現在インストールされている completer 関数は削除されます。completer 関数は `function(text, state)` の形式で、関数が文字列でない値を返すまで *state* を `0`, `1`, `2`, ..., にして呼び出します。この関数は *text* から始まる補完結果として次に来そうなものを返さなければなりません。

The installed completer function is invoked by the *entry_func* callback passed to `rl_completion_matches()` in the underlying library. The *text* string comes from the first parameter to the `rl_attempted_completion_function` callback of the underlying library.

readline.get_completer()

completer 関数を取得します。completer 関数が設定されていなければ `None` を返します。

readline.get_completion_type()

Get the type of completion being attempted. This returns the `rl_completion_type` variable in the underlying library as an integer.

readline.get_begidx()

readline.get_endidx()

Get the beginning or ending index of the completion scope. These indexes are the *start* and *end* arguments passed to the `rl_attempted_completion_function` callback of the underlying library.

readline.set_completer_delims(*string*)
readline.get_completer_delims()

Set or get the word delimiters for completion. These determine the start of the word to be considered for completion (the completion scope). These functions access the rl_completer_word_break_characters variable in the underlying library.

readline.set_completion_display_matches_hook([*function*])

Set or remove the completion display function. If *function* is specified, it will be used as the new completion display function; if omitted or None, any completion display function already installed is removed. This sets or clears the rl_completion_display_matches_hook callback in the underlying library. The completion display function is called as function(substitution, [matches], longest_match_length) once each time matches need to be displayed.

6.7.7 使用例

以下の例では、ユーザのホームディレクトリにある履歴ファイル .python_history の読み込みと保存を自動的に行うために、*readline* モジュールの履歴の読み書き関数をどのように使うかを示しています。以下のソースコードは通常、対話セッション中はユーザの PYTHONSTARTUP ファイルから自動的に実行されます:

```
import atexit
import os
import readline

histfile = os.path.join(os.path.expanduser("~"), ".python_history")
try:
    readline.read_history_file(histfile)
    # default history len is -1 (infinite), which may grow unruly
    readline.set_history_length(1000)
except FileNotFoundError:
    pass

atexit.register(readline.write_history_file, histfile)
```

Python が 対話モード で実行される時、このコードは実際には自動的に実行されます (*readline* の設定 を参照してください)。

次の例では上記と同じ目的を達成できますが、ここでは新規の履歴のみを追加することで、並行して対話セッションがサポートされます:

```
import atexit
import os
import readline
histfile = os.path.join(os.path.expanduser("~"), ".python_history")

try:
    readline.read_history_file(histfile)
    h_len = readline.get_current_history_length()
except FileNotFoundError:
```

(次のページに続く)

(前のページからの続き)

```
    open(histfile, 'wb').close()
    h_len = 0

def save(prev_h_len, histfile):
    new_h_len = readline.get_current_history_length()
    readline.set_history_length(1000)
    readline.append_history_file(new_h_len - prev_h_len, histfile)
atexit.register(save, h_len, histfile)
```

次の例では *code.InteractiveConsole* クラスを拡張し、履歴の保存・復旧をサポートします。

```
import atexit
import code
import os
import readline

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/.console-history")):
        code.InteractiveConsole.__init__(self, locals, filename)
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except FileNotFoundError:
                pass
            atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
        readline.set_history_length(1000)
        readline.write_history_file(histfile)
```

6.8 rlcompleter --- GNU readline 向け補完関数

ソースコード: Lib/rlcompleter.py

---

rlcompleter モジュールでは Python の識別子やキーワードを定義した *readline* モジュール向けの補完関数を定義しています。

このモジュールが Unix プラットフォームで import され、*readline* が利用できるときには、Completer クラスのインスタンスが自動的に作成され、complete() メソッドが *readline* 補完に設定されます。

以下はプログラム例です:

```
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline. <TAB PRESSED>
readline.__doc__          readline.get_line_buffer(  readline.read_init_file(
readline.__file__         readline.insert_text(      readline.set_completer(
readline.__name__         readline.parse_and_bind(
>>> readline.
```

rlcompleter モジュールは、Python の 対話モード と一緒に使用するのを意図して設計されています。Python を -S オプションをつけずに実行している場合、このモジュールが自動的にインポートされ、構成されます (*readline* の設定 を参照)。

readline のないプラットフォームでも、このモジュールで定義される Completer クラスは独自の目的に使えます。

6.8.1 Completer オブジェクト

Completer オブジェクトは以下のメソッドを持っています:

Completer.complete(*text*, *state*)

text の *state* 番目の補完候補を返します。

もし *text* がピリオド ('.') を含まない場合、*__main__* 、*builtins* で定義されている名前か、キーワード (*keyword* モジュールで定義されている) から補完されます。

ピリオドを含む名前の場合、副作用を出さずに名前を最後まで評価しようとします (関数を明示的に呼び出しはしませんが、__getattr__() を呼んでしまうことはあります) そして、*dir()* 関数でマッチする語を見つけます。式を評価中に発生した全ての例外は補足して無視され、*None* を返します。

第

SEVEN

バイナリデータ処理

この章で紹介されているモジュールはバイナリデータを扱うための基本的な処理を提供しています。ファイルフォーマットやネットワークプロトコルなど、その他のバイナリデータ処理については、それぞれの節で解説されています。

テキスト処理サービス で紹介する一部のライブラリには、ASCII 互換のバイナリフォーマットで利用できるもの (例: *re*) や 、すべてのバイナリデータで利用できるもの (例: *difflib*) があります。

加えて、**バイナリシーケンス型** --- *bytes, bytearray, memoryview* に書かれている Python ビルトインデータ型についても参照してください。

7.1 struct --- バイト列をパックされたバイナリデータとして解釈する

ソースコード: Lib/struct.py

このモジュールは、Python の値と Python *bytes* オブジェクトとして表される C の構造体データとの間の変換を実現します。このモジュールは特に、ファイルに保存されたり、ネットワーク接続を経由したバイナリデータを扱うときに使われます。このモジュールでは、C 構造体のレイアウトおよび Python の値との間で行いたい変換をコンパクトに表現するために、**書式文字列** を使います。

注釈: デフォルトでは、与えられた C の構造体をパックする際に、関連する C データ型を適切にアラインメント (alignment) するために数バイトのパディングを行うことがあります。この挙動が選択されたのは、パックされた構造体のバイト表現を対応する C 構造体のメモリレイアウトに正確に対応させるためです。プラットフォーム独立のデータフォーマットを扱ったり、隠れたパディングを排除したりするには、サイズ及びアラインメントとして `native` の代わりに `standard` を使うようにします: 詳しくは **バイトオーダ、サイズ、アラインメント** を参照して下さい。

いくつかの *struct* の関数 (および *Struct* のメソッド) は *buffer* 引数を取ります。これは bufferobjects を実装していて読み取り可能または読み書き可能なバッファを提供するオブジェクトのことです。この目的のために使われる最も一般的な型は *bytes* と *bytearray* ですが、バイトの配列とみなすことができるような他の多くの型がバッファプロトコルを実装しています。そのため、それらは *bytes* オブジェクトから追加のコピーなしで読み出しや書き込みができます。

7.1.1 関数と例外

このモジュールは以下の例外と関数を定義しています:

exception struct.error
様々な状況で送出される例外です。引数は何が問題なのかを記述する文字列です。

struct.pack(*format*, *v1*, *v2*, ...)
フォーマット文字列 *format* に従い値 *v1*, *v2*, ... をパックして、バイト列オブジェクトを返します。引数は指定したフォーマットが要求する型と正確に一致していなければなりません。

struct.pack_into(*format*, *buffer*, *offset*, *v1*, *v2*, ...)
フォーマット文字列 *format* に従い値 *v1*, *v2*, ... をパックしてバイト列にし、書き込み可能な *buffer* のオフセット *offset* 位置より書き込みます。オフセットは省略出来ません。

struct.unpack(*format*, *buffer*)
(`pack(format, ...)` でパックされたであろう) バッファ *buffer* を、書式文字列 *format* に従ってアンパックします。値が一つしかない場合を含め、結果はタプルで返されます。バッファのバイトサイズは、*calcsize()* の返り値である書式文字列が要求するサイズと一致しなければなりません。

struct.unpack_from(*format*, *buffer*, *offset=0*)
バッファ *buffer* を、*offset* の位置から書式文字列 *format* に従ってアンパックします。値が一つしかない場合を含め、結果はタプルで返されます。*offset* を始点とするバッファのバイトサイズは、少なくとも *calcsize()* の返り値である書式文字列が要求するサイズでなければなりません。

struct.iter_unpack(*format*, *buffer*)
Iteratively unpack from the buffer *buffer* according to the format string *format*. This function returns an iterator which will read equally-sized chunks from the buffer until all its contents have been consumed. The buffer's size in bytes must be a multiple of the size required by the format, as reflected by *calcsize()*.

イテレーション毎に書式文字列で指定されたタプルを yield します。

バージョン 3.4 で追加.

struct.calcsize(*format*)
書式文字列 *format* に従って、構造体 (それと `pack(format, ...)` によって作成されるバイト列オブジェクト) のサイズを返します。

7.1.2 書式文字列

書式文字列はデータをパックしたりアンパックしたりするときの期待されるレイアウトを指定するためのメカニズムです。文字列はパック/アンパックされるデータの型を指定する **書式指定文字** から組み立てられます。さらに、**バイトオーダ、サイズ、アラインメント** を制御するための特殊文字もあります。

バイトオーダ、サイズ、アラインメント

デフォルトでは、C での型はマシンのネイティブ (native) の形式およびバイトオーダ (byte order) で表され、適切にアラインメント (alignment) するために、必要に応じて数バイトのパディングを行ってスキップします (これは C コンパイラが用いるルールに従います)。

これに代わって、フォーマット文字列の最初の文字を使って、バイトオーダやサイズ、アラインメントを指定することができます。指定できる文字を以下のテーブルに示します:

文字	バイトオーダ	サイズ	アラインメント
`@`	native	native	native
`=`	native	standard	none
`<`	リトルエンディアン	standard	none
`>`	ビッグエンディアン	standard	none
`!`	ネットワーク (= ビッグエンディアン)	standard	none

フォーマット文字列の最初の文字が上のいずれかでない場合、`'@'` であるとみなされます。

ネイティブのバイトオーダはビッグエンディアンかリトルエンディアンで、ホスト計算機に依存します。例えば、Intel x86 および AMD64 (x86-64) はリトルエンディアンです。Motorola 68000 および PowerPC G5 はビッグエンディアンです。ARM および Intel Itanium はエンディアンを切り替えられる機能を備えています (バイエンディアン)。使っているシステムでのエンディアンは `sys.byteorder` を使って調べて下さい。

ネイティブのサイズおよびアラインメントは C コンパイラの `sizeof` 式で決定されます。ネイティブのサイズおよびアラインメントはネイティブのバイトオーダと同時に使われます。

標準のサイズはフォーマット文字だけで決まります。**書式指定文字** の表を参照して下さい。

`'@'` と `'='` の違いに注意してください: 両方ともネイティブのバイトオーダですが、後者のバイトサイズとアラインメントは標準のものに合わせてあります。

The form `'!'` represents the network byte order which is always big-endian as defined in IETF RFC 1700.

バイトオーダに関して、「(強制的にバイトスワップを行う) ネイティブの逆」を指定する方法はありません。`'<'` または `'>'` のうちふさわしい方を選んでください。

注釈:

(1) パディングは構造体のメンバの並びの中にだけ自動で追加されます。最初や最後にパディングが追加されることはありません。

(2) ネイティブでないサイズおよびアラインメントが使われる場合にはパディングは行われません (たとえば '<', '>', '=', '!' を使った場合です)。

(3) 特定の型によるアラインメント要求に従うように構造体の末端をそろえるには、繰り返し回数をゼロにした特定の型でフォーマットを終端します。**使用例** を参照して下さい。

書式指定文字

フォーマット文字 (format character) は以下の意味を持っています; C と Python の間の変換では、値は正確に以下に指定された型でなくてはなりません:「標準のサイズ」列は standard サイズ使用時にパックされた値が何バイトかを示します。つまり、フォーマット文字列が `'<'`, `'>'`, `'!'`, `'='` のいずれかで始まっている場合のものです。native サイズ使用時にはパックされた値の大きさはプラットフォーム依存です。

フォーマット	C の型	Python の型	標準のサイズ	注釈
x	パディングバイト	値なし		
c	char	長さ 1 のバイト列	1	
b	signed char	整数	1	(1), (2)
B	unsigned char	整数	1	(2)
?	_Bool	真偽値型 (bool)	1	(1)
h	short	整数	2	(2)
H	unsigned short	整数	2	(2)
i	int	整数	4	(2)
I	unsigned int	整数	4	(2)
l	long	整数	4	(2)
L	unsigned long	整数	4	(2)
q	long long	整数	8	(2)
Q	unsigned long long	整数	8	(2)
n	ssize_t	整数		(3)
N	size_t	整数		(3)
e	(6)	浮動小数点数	2	(4)
f	float	浮動小数点数	4	(4)
d	double	浮動小数点数	8	(4)
s	char[]	bytes		
p	char[]	bytes		
P	void *	整数		(5)

バージョン 3.3 で変更: `'n'` および `'N'` フォーマットのサポートが追加されました。

バージョン 3.6 で変更: `'e'` フォーマットのサポートが追加されました。

注釈:

(1) `'?'` 変換コードは C99 で定義された _Bool 型に対応します。その型が利用できない場合は、char で代用されます。標準モードでは常に 1 バイトで表現されます。

(2) 非整数を整数の変換コードを使ってパックしようとすると、非整数が `__index__()` メソッドを持っていた場合は、整数に変換するためにパックする前にそのメソッドが呼ばれます。

バージョン 3.2 で変更: 非整数に対して `__index__()` メソッドが使われるようになったのは 3.2 の新機能です。

(3) `'n'` および `'N'` 変換コードは (デフォルトもしくはバイトオーダ文字 `'@'` 付きで選択される) native

サイズ使用時のみ利用できます。standard サイズ使用時には、自身のアプリケーションに適する他の整数フォーマットを使うことができます。

(4) `'f'` 、`'d'` および `'e'` 変換コードについて、パックされた表現は IEEE 754 binary32 (`'f'` の場合) 、binary64 (`'d'` の場合) 、または binary16(`'e'` の場合) フォーマットが、プラットフォームにおける浮動小数点数のフォーマットに関係なく使われます。

(5) `'P'` フォーマット文字はネイティブバイトオーダでのみ利用可能です (デフォルトのネットワークバイトオーダに設定するか、`'@'` バイトオーダ指定文字を指定しなければなりません)。`'='` を指定した場合、ホスト計算機のバイトオーダに基づいてリトルエンディアンとビッグエンディアンのどちらを使うかを決めます。struct モジュールはこの設定をネイティブのオーダ設定として解釈しないので、`'P'` を使うことはできません。

(6) IEEE 754 の binary16 ” 半精度” 型は、IEEE 754 standard の 2008 年の改訂で導入されました。半精度型は、符号 bit 、5 bit の指数部、11 bit の精度 (明示的には 10 bit が保存される) を持ち、おおよそ `6.1e-05` から `6.5e+04` までの数を完全な精度で表現できます。この型は C コンパイラでは広くはサポートされていません: たいていのマシンでは、保存するのに unsigned short が使えますが、数学の演算には使えません。詳しいことは Wikipedia の half-precision floating-point format のページを参照してください。

フォーマット文字の前に整数をつけ、繰り返し回数 (count) を指定することができます。例えば、フォーマット文字列 `'4h'` は `'hhhh'` と全く同じ意味です。

フォーマット文字間の空白文字は無視されます; count とフォーマット文字の間にはスペースを入れてはいけません。

`'s'` フォーマット文字での繰り返し回数 (count) は、他のフォーマット文字のような繰り返し回数ではなく、バイト長として解釈されます。例えば、`'10s'` は単一の 10 バイトの文字列を意味し、`'10c'` は 10 個の文字を意味します。繰り返し回数が指定されなかった場合は、デフォルト値の 1 とみなされます。パックでは、文字列はサイズに合うように切り詰められたり null バイトで埋められたりします。アンパックでは、返されるバイトオブジェクトのバイト数は、正確に指定した通りになります。特殊な場合として、`'0s'` は単一の空文字列を意味し、`'0c'` は 0 個の文字を意味します。

整数フォーマット (`'b'`, `'B'`, `'h'`, `'H'`, `'i'`, `'I'`, `'l'`, `'L'`, `'q'`, `'Q'`) のいずれかを使って値 `x` をパックするとき `x` がフォーマットの適切な値の範囲に無い場合、*`struct.error`* が送出されます。

バージョン 3.1 で変更: 3.0 では、いくつかの整数フォーマットが適切な範囲にない値を覆い隠して、*`struct.error`* の代わりに *`DeprecationWarning`* を送出していました。

フォーマット文字 `'p'` は ”Pascal 文字列 (pascal string)” をコードします。Pascal 文字列は count で与えられる **固定長のバイト列** に収められた短い可変長の文字列です。このデータの先頭の 1 バイトには文字列の長さか 255 のうち、小さい方の数が収められます。その後に文字列のバイトデータが続きます。*`pack()`* に渡された Pascal 文字列の長さが長すぎた (count-1 よりも長い) 場合、先頭の `count-1` バイトが書き込まれます。文字列が `count-1` よりも短い場合、指定した count バイトに達するまでの残りの部分はヌルで埋められます。*`unpack()`* では、フォーマット文字 `'p'` は指定された `count` バイトだけデータを読み込みますが、返される文字列は決して 255 文字を超えることはないので注意してください。

`'?'` フォーマット文字では、返り値は *`True`* または *`False`* です。パックするときには、引数オブジェクトの

論理値としての値が使われます。0 または 1 のネイティブや標準の真偽値表現でパックされ、アンパックされるときはゼロでない値は True になります。

使用例

注釈: 全ての例は、ビッグエンディアンのマシンで、ネイティブのバイトオーダ、サイズおよびアラインメントを仮定します。

基本的な例として、三つの整数をパック/アンパックします:

```
>>> from struct import *
>>> pack('hhl', 1, 2, 3)
b'\x00\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('hhl', b'\x00\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('hhl')
8
```

アンパックした結果のフィールドは、変数に割り当てるか named tuple でラップすることによって名前を付けることができます:

```
>>> record = b'raymond   \x32\x12\x08\x01\x08'
>>> name, serialnum, school, gradelevel = unpack('<10sHHb', record)

>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name serialnum school gradelevel')
>>> Student._make(unpack('<10sHHb', record))
Student(name=b'raymond   ', serialnum=4658, school=264, gradelevel=8)
```

アラインメントの要求を満たすために必要なパディングが異なるという理由により、フォーマット文字の順番がサイズの違いを生み出すことがあります:

```
>>> pack('ci', b'*', 0x12131415)
b'*\x00\x00\x00\x12\x13\x14\x15'
>>> pack('ic', 0x12131415, b'*')
b'\x12\x13\x14\x15*'
>>> calcsize('ci')
8
>>> calcsize('ic')
5
```

以下のフォーマット `'llh0l'` は、long 型が 4 バイトを境界としてそろえられていると仮定して、末端に 2 バイトをパディングします:

```
>>> pack('llh0l', 1, 2, 3)
b'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

この例はネイティブのサイズとアラインメントが使われているときだけ思った通りに動きます。標準のサイズとアラインメントはアラインメントの設定ではいかなるアラインメントも行いません。

参考:

array **モジュール**　一様なデータ型からなるバイナリ記録データのパック。

xdrlib **モジュール**　XDR データのパックおよびアンパック。

7.1.3 クラス

struct モジュールは次の型を定義します:

class struct.Struct(*format*)

フォーマット文字列 *format* に従ってバイナリデータを読み書きする、新しい Struct オブジェクトを返します。Struct オブジェクトを一度作ってからそのメソッドを使うと、フォーマット文字列のコンパイルが一度で済むので、*struct* モジュールの関数を同じフォーマットで何度も呼び出すよりも効率的です。

注釈:　The compiled versions of the most recent format strings passed to *Struct* and the module-level functions are cached, so programs that use only a few format strings needn't worry about reusing a single *Struct* instance.

コンパイルされた Struct オブジェクトは以下のメソッドと属性をサポートします:

pack(*v1*, *v2*, ...)

pack() 関数と同じ、コンパイルされたフォーマットを利用するメソッドです。(`len(result)` は *size* と等しいでしょう)

pack_into(*buffer*, *offset*, *v1*, *v2*, ...)

pack_into() 関数と同じ、コンパイルされたフォーマットを利用するメソッドです。

unpack(*buffer*)

unpack() 関数と同じ、コンパイルされたフォーマットを利用するメソッドです。(buffer のバイト数は *size* と等しくなければなりません)。

unpack_from(*buffer*, *offset=0*)

unpack_from() 関数と同じ、コンパイルされたフォーマットを利用するメソッドです。(*offset* を始点とする buffer のバイト数は少なくとも *size* 以上でなければなりません)。

iter_unpack(*buffer*)

iter_unpack() 関数と同じ、コンパイルされたフォーマットを利用するメソッドです。(buffer のバイト数は *size* の倍数でなければなりません)。

バージョン 3.4 で追加.

format
この Struct オブジェクトを作成する時に利用されたフォーマット文字列です。

バージョン 3.7 で変更: The format string type is now *str* instead of *bytes*.

size
format 属性に対応する構造体の (従って *pack()* メソッドによって作成されるバイト列オブジェクトの) サイズです。

7.2 codecs --- codec レジストリと基底クラス

ソースコード: Lib/codecs.py

このモジュールは、標準的な Python codec (エンコーダとデコーダ) 用の基底クラスを定義し、codec とエラー処理検索プロセスを管理する内部の Python codec レジストリへのアクセスを提供します。多くの codec はテキストをバイト形式にエンコードする **テキストエンコーディング** ですが、テキストをテキストに、またはバイトをバイトにエンコードする codec も提供されています。カスタムの codec は任意の型間でエンコードとデコードを行えますが、一部のモジュール機能は **テキストエンコーディング** か *bytes* へのエンコードのみに制限されています。

このモジュールでは、任意の codec でエンコードやデコードを行うための、以下の関数が定義されています。

codecs.encode(*obj*, *encoding='utf-8'*, *errors='strict'*)
encoding に記載された codec を使用して *obj* をエンコードします。

希望のエラー処理スキームを *errors* に設定することができます。デフォルトのエラーハンドラ (エラー処理関数) は `'strict'` です。これはエンコードエラーは *ValueError* (もしくは *UnicodeEncodeError* のような、より codec に固有のサブクラス) を送出することを意味します。codec エラー処理についてのより詳しい情報は *Codec* **基底クラス** を参照してください。

codecs.decode(*obj*, *encoding='utf-8'*, *errors='strict'*)
encoding に記載された codec を使用して *obj* をデコードします。

希望のエラー処理スキームを *errors* に設定することができます。デフォルトのエラーハンドラは `'strict'` です。これはデコードエラーは *ValueError* (もしくは *UnicodeDecodeError* のような、より codec に固有のサブクラス) を送出することを意味します。codec エラー処理についてのより詳しい情報は *Codec* **基底クラス** を参照してください。

各 codec についての詳細も、次のようにして直接調べることができます。

codecs.lookup(*encoding*)
Python codec レジストリから codec 情報を探し、以下で定義するような *CodecInfo* オブジェクトを返します。

エンコーディングの検索は、まずレジストリのキャッシュから行います。見つからなければ、登録されている検索関数のリストから探します。*CodecInfo* オブジェクトが一つも見つからなければ

LookupError を送出します。見つかったら、その *CodecInfo* オブジェクトはキャッシュに保存され、呼び出し側に返されます。

class codecs.**CodecInfo**(*encode*, *decode*, *streamreader=None*, *streamwriter=None*, *incrementalencoder=None*, *incrementaldecoder=None*, *name=None*)

codec レジストリ内を検索する場合の、codec の詳細です。コントラクタ引数は、次の同名の属性に保存されます。

name

エンコーディングの名前です。

encode

decode

ステートレスなエンコーディングとデコーディングの関数です。これらは、Codec インスタンスの *encode()* メソッドと *decode()* メソッドと同じインターフェースを持っている必要があります (see *Codec* **のインターフェース** を参照)。この関数またはメソッドは、ステートレスモードで動作することが想定されています。

incrementalencoder

incrementaldecoder

インクリメンタル・エンコーダとデコーダのクラスまたはファクトリ関数です。これらは、基底クラスの *IncrementalEncoder* と *IncrementalDecoder* が定義するインターフェースをそれぞれ提供する必要があります。インクリメンタルな codec は、ステート (内部状態) を保持することができます。

streamwriter

streamreader

ストリームライターとリーダーのクラスまたはファクトリ関数です。これらは、基底クラスの *StreamWriter* と *StreamReader* が定義するインターフェースをそれぞれ提供する必要があります。ストリーム codec は、ステートを保持することができます。

さまざまな codec 構成要素へのアクセスを簡便化するために、このモジュールは以下のような関数を提供しています。これらの関数は、codec の検索に *lookup()* を使います:

codecs.**getencoder**(*encoding*)

与えられたエンコーディングに対する codec を検索し、エンコーダ関数を返します。

エンコーディングが見つからなければ *LookupError* を送出します。

codecs.**getdecoder**(*encoding*)

与えられたエンコーディングに対する codec を検索し、デコーダ関数を返します。

エンコーディングが見つからなければ *LookupError* を送出します。

codecs.**getincrementalencoder**(*encoding*)

与えられたエンコーディングに対する codec を検索し、インクリメンタル・エンコーダクラスまたはファクトリ関数を返します。

エンコーディングが見つからないか、codec がインクリメンタル・エンコーダをサポートしなければ *LookupError* を送出します。

codecs.**getincrementaldecoder**(*encoding*)

与えられたエンコーディングに対する codec を検索し、インクリメンタル・デコーダクラスまたはファクトリ関数を返します。

エンコーディングが見つからないか、codec がインクリメンタル・デコーダをサポートしなければ *LookupError* を送出します。

codecs.**getreader**(*encoding*)

与えられたエンコーディングに対する codec を検索し、*StreamReader* クラスまたはファクトリ関数を返します。

エンコーディングが見つからなければ *LookupError* を送出します。

codecs.**getwriter**(*encoding*)

与えられたエンコーディングに対する codec を検索し、*StreamWriter* クラスまたはファクトリ関数を返します。

エンコーディングが見つからなければ *LookupError* を送出します。

次のように、適切な codec 検索関数を登録することで、カスタムの codecs を利用することができます。

codecs.**register**(*search_function*)

codec 検索関数を登録します。検索関数は第 1 引数にすべてアルファベットの小文字から成るエンコーディング名を取り、*CodecInfo* オブジェクトを返します。検索関数が指定されたエンコーディングを見つけられない場合、`None` を返します。

注釈: 現在、検索関数の登録は不可逆的です。このため、ユニットテストやモジュールの再ロード時などに問題が生じることがあります。

エンコードされたテキストファイルを処理する場合、組み込みの *open()* とそれに関連付けられた *io* モジュールの使用が推奨されていますが、このモジュールは追加のユーティリティ関数とクラスを提供し、バイナリファイルを処理する場合に幅広い codecs を利用できるようにします。

codecs.**open**(*filename*, *mode='r'*, *encoding=None*, *errors='strict'*, *buffering=-1*)

エンコードされたファイルを *mode* を使って開き、透過的なエンコード/デコードを提供する *StreamReaderWriter* のインスタンスを返します。デフォルトのファイルモードは `'r'` 、つまり、読み出しモードでファイルを開きます。

注釈: 下層のエンコードされたファイルは、常にバイナリモードで開きます。読み書き時に、`'\n'` の自動変換は行われません。*mode* 引数は、組み込みの *open()* 関数が受け入れる任意のバイナリモードにすることができます。`'b'` が自動的に付加されます。

---

encoding は、そのファイルに対して使用されるエンコーディングを指定します。バイトにエンコードする、あるいはバイトからデコードするすべてのエンコーディングが許可されます。ファイルメソッドがサポートするデータ型は、使用される codec によって異なります。

エラーハンドリングのために *errors* を渡すことができます。これはデフォルトでは `'strict'` で、エンコード時にエラーがあれば *ValueError* を送出します。

buffering has the same meaning as for the built-in *open()* function. It defaults to -1 which means that the default buffer size will be used.

codecs.EncodedFile(*file*, *data_encoding*, *file_encoding=None*, *errors='strict'*)

透過的なエンコード変換を行うファイルのラップされたバージョンである、*StreamRecoder* インスタンスを返します。元のファイルは、ラップされたバージョンが閉じられる時に、閉じられます。

ラップされたファイルに書き込まれたデータは、指定された *data_encoding* に従ってデコードされ、次に *file_encoding* を使用して元のファイルにバイトとして書き出されます。元のファイルから読み出されたバイトは、*file_encoding* に従ってデコードされ、結果は *data_encoding* を使用してエンコードされます。

file_encoding が与えられなければ、*data_encoding* がデフォルトになります。

エラーハンドリングのために *errors* を渡すことができます。これはデフォルトでは `'strict'` で、エンコード時にエラーがあれば *ValueError* を送出します。

codecs.iterencode(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)

インクリメンタル・エンコーダを使って、*iterator* から供給される入力を反復的にエンコードします。この関数は *generator* です。*errors* 引数は (他のあらゆるキーワード引数と同様に) インクリメンタル・エンコーダにそのまま引き渡されます。

この関数では、コーデックはエンコードするテキストの *str* オブジェクトを受け付ける必要があります。従って、`base64_codec` のようなバイトからバイトへのエンコーダはサポートしていません。

codecs.iterdecode(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)

インクリメンタル・デコーダを使って、*iterator* から供給される入力を反復的にデコードします。この関数は *generator* です。*errors* 引数は (他のあらゆるキーワード引数と同様に) インクリメンタル・デコーダにそのまま引き渡されます。

この関数では、コーデックはエンコードする *bytes* オブジェクトを受け付ける必要があります。従って、`rot_13` のようなテキストからテキストへのエンコーダが *iterencode()* で同等に使えるとしても、この関数ではサポートしていません。

このモジュールは以下のような定数も定義しています。プラットフォーム依存なファイルを読み書きするのに役立ちます:

codecs.BOM
codecs.BOM_BE
codecs.BOM_LE
codecs.BOM_UTF8
codecs.BOM_UTF16
codecs.BOM_UTF16_BE
codecs.BOM_UTF16_LE
codecs.BOM_UTF32
codecs.BOM_UTF32_BE

codecs.BOM_UTF32_LE

これらの定数は、いくつかのエンコーディングの Unicode のバイトオーダマーク (BOM) で、様々なバイトシーケンスを定義します。これらは、UTF-16 と UTF-32 のデータストリームで使用するバイトオーダを指定したり、UTF-8 で Unicode シグネチャとして使われます。*BOM_UTF16* は、プラットフォームのネイティブバイトオーダによって *BOM_UTF16_BE* または *BOM_UTF16_LE* です。*BOM* は *BOM_UTF16* のエイリアスです。同様に、*BOM_LE* は *BOM_UTF16_LE* の、*BOM_BE* は *BOM_UTF16_BE* のエイリアスです。その他の定数は UTF-8 と UTF-32 エンコーディングの BOM を表します。

7.2.1 Codec 基底クラス

codecs モジュールは、codec オブジェクトを操作するインタフェースを定義する一連の基底クラスを定義します。このモジュールは、カスタムの codec の実装の基礎として使用することもできます。

Python で codec として使えるようにするには、ステートレスエンコーダ、ステートレスデコーダ、ストリームリーダ、ストリームライタの 4 つのインタフェースを定義する必要があります。通常は、ストリームリーダとライタはステートレスエンコーダとデコーダを再利用して、ファイルプロトコルを実装します。codec の作者は、codec がエンコードとデコードのエラーの処理方法も定義する必要があります。

エラーハンドラ

To simplify and standardize error handling, codecs may implement different error handling schemes by accepting the *errors* string argument. The following string values are defined and implemented by all standard Python codecs:

値	意味
'strict'	Raise *UnicodeError* (or a subclass); this is the default. Implemented in *strict_errors()*.
'ignore'	不正な形式のデータを無視し、何も通知することなく処理を継続します。*ignore_errors()* で実装されています。

以下のエラーハンドラは、テキストエンコーディング にのみ適用されます。

値	意味
'replace'	Replace with a suitable replacement marker; Python will use the official U+FFFD REPLACEMENT CHARACTER for the built-in codecs on decoding, and '?' on encoding. Implemented in *replace_errors()*.
'xmlcharrefreplace'	Replace with the appropriate XML character reference (only for encoding). Implemented in *xmlcharrefreplace_errors()*.
'backslashreplace'	バックスラッシュつきのエスケープシーケンスで置換します。*backslashreplace_errors()* で実装されています。
'namereplace'	Replace with \N{...} escape sequences (only for encoding). Implemented in *namereplace_errors()*.
'surrogateescape'	デコード時には、バイト列を U+DC80 から U+DCFF の範囲の個々のサロゲートコードで置き換えます。データのエンコード時に 'surrogateescape' エラーハンドラが使用されると、このコードは同じバイト列に戻されます。(詳しくは **PEP 383** を参照。)

さらに、次のエラーハンドラは与えられた codec に特有です:

値	Codecs	意味
'surrogatepass'	utf-8, utf-16, utf-32, utf-16-be, utf-16-le, utf-32-be, utf-32-le	Allow encoding and decoding of surrogate codes. These codecs normally treat the presence of surrogates as an error.

バージョン 3.1 で追加: 'surrogateescape' および 'surrogatepass' エラーハンドラ。

バージョン 3.4 で変更: 'surrogatepass' エラーハンドラは utf-16* コーデックと utf-32* コーデックで動作するようになりました。

バージョン 3.5 で追加: 'namereplace' エラーハンドラです。

バージョン 3.5 で変更: 'backslashreplace' エラーハンドラは、デコード時と翻訳時に動作するようになりました。

次のように、名前付きの新しいエラーハンドラを登録することで、許可される値の集合を拡張することができます。

codecs.**register_error**(*name*, *error_handler*)

エラーハンドラ *error_handler* を名前 *name* で登録します。エンコード中およびデコード中にエラーが送出された場合、*name* が errors 引数として指定されていれば *error_handler* が呼び出されます。

error_handler はエラーの場所に関する情報の入った *UnicodeEncodeError* インスタンスとともに呼び出されます。エラー処理関数はこの例外を送出するか、別の例外を送出するか、入力のエンコードできなかった部分の代替文字列とエンコードを再開する場所が入ったタプルを返す必要があります。代替文字列は *str* または *bytes* のいずれかにすることができます。代替文字列がバイト列である場合、エンコーダは単に出力バッファにそれをコピーします。代替文字列が文字列である場合、エンコーダは代替文字列をエンコードします。元の入力中の指定位置からエンコードが再開されます。位置を負の値

にすると、入力文字列の末端からの相対位置として扱われます。境界の外側にある位置を返した場合には *IndexError* が送出されます。

デコードと翻訳の動作は似ていますが、エラーハンドラに渡されるのが *UnicodeDecodeError* か *UnicodeTranslateError* である点と、エラーハンドラの置換した内容が直接出力されるという点が異なります。

登録済みのエラーハンドラ (標準エラーハンドラを含む) は、次のようにその名前で検索することができます。

codecs.**lookup_error**(*name*)

名前 *name* で登録済みのエラーハンドラを返します。

エラーハンドラが見つからなければ *LookupError* を送出します。

以下の標準エラーハンドラも、モジュールレベルの関数として利用できます。

codecs.**strict_errors**(*exception*)

`strict` エラー処理を実装します。エンコードエラーまたはデコードエラーはそれぞれ *UnicodeError* を送出します。

codecs.**replace_errors**(*exception*)

`'replace'` エラー処理を実装します (**テキストエンコーディング** のみ)。(codec によりエンコードする必要のある) エンコードエラーに対しては `'?'` に、デコードエラーに対しては `'\ufffd'` (Unicode 代替文字) に置き換えます。

codecs.**ignore_errors**(*exception*)

`ignore` エラー処理を実装します。不正な形式のデータは無視され、エンコードまたはデコードは何も通知することなく継続されます。

codecs.**xmlcharrefreplace_errors**(*exception*)

`'xmlcharrefreplace'` エラー処理を実装します (**テキストエンコーディング** のエンコードのみ)。エンコードできない文字は、適切な XML 文字参照に置き換えます。

codecs.**backslashreplace_errors**(*exception*)

`'backslashreplace'` エラー処理を実装します (**テキストエンコーディング** のエンコードのみ)。不正な形式のデータは、バックスラッシュ付きのエスケープシーケンスに置き換えます。

codecs.**namereplace_errors**(*exception*)

`'namereplace'` エラー処理を実装します (**テキストエンコーディング** のエンコードのみ)。エンコードできない文字は、`\N{...}` エスケープシーケンスに置き換えます。

バージョン 3.5 で追加.

ステートレスなエンコードとデコード

基底の Codec クラスは以下のメソッドを定義します。これらのメソッドは、内部状態を持たないエンコーダ／デコーダ関数のインタフェースを定義します:

Codec.**encode**(*input*[, *errors*])

オブジェクト *input* エンコードし、(出力オブジェクト, 消費した長さ) のタプルを返します。例えば、テキストエンコーディング は文字列オブジェクトを特有の文字セット (例えば cp1252 や iso-8859-1) を用いてバイト列オブジェクトに変換します。

errors 引数は適用するエラー処理を定義します。'strict' 処理がデフォルトです。

このメソッドは Codec に内部状態を保存してはなりません。効率よくエンコードするために状態を保持しなければならないような codecs には *StreamWriter* を使ってください。

エンコーダは長さが 0 の入力を処理できなければなりません。この場合、空のオブジェクトを出力オブジェクトとして返さなければなりません。

Codec.**decode**(*input*[, *errors*])

オブジェクト *input* をデコードし、(出力オブジェクト, 消費した長さ) のタプルを返します。例えば、テキストエンコーディング は、特定の文字集合エンコーディングでエンコードされたバイト列オブジェクトを文字列オブジェクトに変換します。

テキストエンコーディングとバイト列からバイト列への codec では、*input* は bytes オブジェクト、または読み出し専用のバッファインタフェースを提供するオブジェクトである必要があります。例えば、buffer オブジェクトやメモリマップドファイルでなければなりません。

errors 引数は適用するエラー処理を定義します。'strict' 処理がデフォルトです。

このメソッドは、Codec インスタンスに内部状態を保存してはなりません。効率よくエンコード／デコードするために状態を保持しなければならないような codecs には *StreamReader* を使ってください。

デコーダは長さが 0 の入力を処理できなければなりません。この場合、空のオブジェクトを出力オブジェクトとして返さなければなりません。

インクリメンタルなエンコードとデコード

IncrementalEncoder クラスおよび *IncrementalDecoder* クラスはそれぞれインクリメンタル・エンコードおよびデコードのための基本的なインタフェースを提供します。エンコード／デコードは内部状態を持たないエンコーダ／デコーダを一度呼び出すことで行なわれるのではなく、インクリメンタル・エンコーダ／デコーダの *encode()*/*decode()* メソッドを複数回呼び出すことで行なわれます。インクリメンタル・エンコーダ／デコーダはメソッド呼び出しの間エンコード／デコード処理の進行を管理します。

encode()/*decode()* メソッド呼び出しの出力結果をまとめたものは、入力をひとまとめにして内部状態を持たないエンコーダ／デコーダでエンコード／デコードしたものと同じになります。

IncrementalEncoder オブジェクト

IncrementalEncoder クラスは入力を複数ステップでエンコードするのに使われます。全てのインクリメンタル・エンコーダが Python codec レジストリと互換性を持つために定義すべきメソッドとして、このクラスには以下のメソッドが定義されています。

class codecs.**IncrementalEncoder**(*errors='strict'*)

IncrementalEncoder インスタンスのコンストラクタ。

全てのインクリメンタル・エンコーダはこのコンストラクタインタフェースを提供しなければなりません。さらにキーワード引数を付け加えるのは構いませんが、Python codec レジストリで利用されるのはここで定義されているものだけです。

IncrementalEncoder は、*errors* キーワード引数を提供することで、様々なエラー取扱方法を実装することができます。取り得る値については **エラーハンドラ** を参照してください。

errors 引数は、同名の属性に代入されます。この属性を変更すると、*IncrementalEncoder* オブジェクトが生きている間に、異なるエラー処理方法に切り替えることができるようになります。

encode(*object*[, *final*])

object を (エンコーダの現在の状態を考慮に入れて) エンコードし、得られたエンコードされたオブジェクトを返します。*encode()* 呼び出しがこれで最後という時には *final* は真でなければなりません (デフォルトは偽です)。

reset()

エンコーダを初期状態にリセットします。出力は破棄されます。`.encode(object, final=True)` を呼び出して、必要に応じて空バイト列またはテキスト文字列を渡すことにより、エンコーダをリセットし、出力を取得します。

getstate()

エンコーダの現在の状態を返します。それは必ず整数でなければなりません。実装は、`0` が最も一般的な状態であることを保証すべきです。(整数より複雑な状態は、状態を marshal/pickle して生じた文字列のバイトを整数にコード化することによって整数に変換することができます。)

setstate(*state*)

エンコーダの状態を *state* にセットします。*state* は *getstate()* によって返されたエンコーダ状態でなければなりません。

IncrementalDecoder オブジェクト

IncrementalDecoder クラスは入力を複数ステップでデコードするのに使われます。全てのインクリメンタル・デコーダが Python codec レジストリと互換性を持つために定義すべきメソッドとして、このクラスには以下のメソッドが定義されています。

class codecs.**IncrementalDecoder**(*errors='strict'*)

IncrementalDecoder インスタンスのコンストラクタ。

全てのインクリメンタル・デコーダはこのコンストラクタインタフェースを提供しなければなりませ

ん。さらにキーワード引数を付け加えるのは構いませんが、Python codec レジストリで利用されるのはここで定義されているものだけです。

IncrementalDecoder は、*errors* キーワード引数を提供することで、様々なエラー取扱方法を実装することができます。取り得る値については **エラーハンドラ** を参照してください。

errors 引数は、同名の属性に代入されます。この属性を変更すると、*IncrementalDecoder* オブジェクトが生きている間に、異なるエラー処理方法に切り替えることができるようになります。

decode(*object*[, *final*])

object を (デコーダの現在の状態を考慮に入れて) デコードし、得られたデコードされたオブジェクトを返します。*decode()* 呼び出しがこれで最後という時には *final* は真でなければなりません (デフォルトは偽です)。もし *final* が真ならばデコーダは入力をデコードし切り全てのバッファをフラッシュしなければなりません。そうできない場合 (たとえば入力の最後に不完全なバイト列があるから)、デコーダは内部状態を持たない場合と同じようにエラーの取り扱いを開始しなければなりません (例外を送出するかもしれません)。

reset()

デコーダを初期状態にリセットします。

getstate()

デコーダの現在の状態を返します。これは 2 要素を含むタプルでなければなりません。1 番目はまだデコードされていない入力を含むバッファです。2 番目は整数で、付加的な状態情報です (実装は 0 が最も一般的な付加的な状態情報であることを保証すべきです)。この付加的な状態情報が 0 である場合、デコーダを入力がバッファされていない状態に戻して、付加的な状態情報を 0 にセットすることが可能でなければなりません。その結果、以前バッファされた入力をデコーダに与えることで、何の出力もせずにデコーダを前の状態に戻します。(整数より複雑な付加的な状態情報は、情報を marshal/pickle して、結果として生じる文字列のバイト列を整数にエンコードすることで、整数に変換することができます。)

setstate(*state*)

デコーダの状態を *state* にセットします。*state* は *getstate()* によって返されたデコーダの状態でなければなりません。

ストリームのエンコードとデコード

StreamWriter と *StreamReader* クラスは、新しいエンコーディングサブモジュールを非常に簡単に実装するのに使用できる、一般的なインターフェイスを提供します。実装例は `encodings.utf_8` をご覧ください。

StreamWriter オブジェクト

StreamWriter クラスは Codec のサブクラスで、以下のメソッドを定義しています。全てのストリームライタは、Python の codec レジストリとの互換性を保つために、これらのメソッドを定義する必要があります。

class codecs.**StreamWriter**(*stream*, *errors='strict'*)

StreamWriter インスタンスのコンストラクタです。

全てのストリームライタはコンストラクタとしてこのインタフェースを提供しなければなりません。キーワード引数を追加しても構いませんが、Python の codec レジストリはここで定義されている引数だけを使います。

stream 引数は、特定の codec に対応して、テキストまたはバイナリデータの書き込みが可能なファイルライクオブジェクトである必要があります。

StreamWriter は、*errors* キーワード引数を提供することで、様々なエラー取扱方法を実装することができます。下層のストリーム codec がサポートできる標準エラーハンドラについては **エラーハンドラ** を参照してください。

errors 引数は、同名の属性に代入されます。この属性を変更すると、*StreamWriter* オブジェクトが生きている間に、異なるエラー処理に変更できます。

write(*object*)

object の内容をエンコードしてストリームに書き出します。

writelines(*list*)

文字列からなるリストを連結して、ストリームに書き出します (可能な場合には *write()* を再利用します) 。バイト列からバイト列への標準 codecs は、このメソッドをサポートしません。

reset()

状態保持に使われていた codec のバッファを強制的に出力してリセットします。

このメソッドが呼び出された場合、出力先データをきれいな状態にし、わざわざストリーム全体を再スキャンして状態を元に戻さなくても新しくデータを追加できるようにしなければなりません。

ここまでで挙げたメソッドの他にも、*StreamWriter* では背後にあるストリームの他の全てのメソッドや属性を継承しなければなりません。

StreamReader オブジェクト

StreamReader クラスは Codec のサブクラスで、以下のメソッドを定義しています。全てのストリームリーダは、Python の codec レジストリとの互換性を保つために、これらのメソッドを定義する必要があります。

class codecs.**StreamReader**(*stream*, *errors='strict'*)

StreamReader インスタンスのコンストラクタです。

全てのストリームリーダはコンストラクタとしてこのインタフェースを提供しなければなりません。キーワード引数を追加しても構いませんが、Python の codec レジストリはここで定義されている引数だけを使います。

stream 引数は、特定の codec に対応して、テキストまたはバイナリデータの読み出しが可能なファイルライクオブジェクトである必要があります。

StreamReader は、*errors* キーワード引数を提供することで、様々なエラー取扱方法を実装することができます。下層のストリーム codec がサポートできる標準エラーハンドラについては **エラーハンドラ** を参照してください。

errors 引数は、同名の属性に代入されます。この属性を変更すると、*StreamReader* オブジェクトが生きている間に、異なるエラー処理に変更できます。

errors 引数に許される値の集合は *register_error()* で拡張できます。

read([*size*[, *chars*[, *firstline*]]])

ストリームからのデータをデコードし、デコード済のオブジェクトを返します。

chars 引数は、いくつのデコードされたコードポイントまたはバイト列を返すかを表します。*read()* メソッドは、要求された数以上のデータを返すことはありませんが、データがそれより少ない場合には要求された数未満のデータを返す場合があります。

size 引数は、デコード用に読み込むエンコードされたバイト列またはコードポイントの、およその最大バイト数を表します。デコーダはこの値を適切な値に変更できます。デフォルト値 -1 の場合、可能な限り多くのデータを読み込みます。この引数の目的は、巨大なファイルの一括デコードを防ぐことにあります。

firstline フラグは、1 行目さえ返せばその後の行でデコードエラーがあっても無視して十分だ、ということを示します。

このメソッドは貪欲な読み込み戦略を取るべきです。すなわち、エンコーディング定義と size の値が許す範囲で、できるだけ多くのデータを読むべきだということです。たとえば、ストリーム上にエンコーディングの終端や状態の目印があれば、それも読み込みます。

readline([*size*[, *keepends*]])

入力ストリームから 1 行読み込み、デコード済みのデータを返します。

size が与えられた場合、ストリームの *read()* メソッドに size 引数として渡されます。

keepends が偽の場合には行末の改行が削除された行が返ります。

readlines([*sizehint*[, *keepends*]])

入力ストリームから全ての行を読み込み、行のリストとして返します。

keepends が真なら、改行は、codec の *decode()* メソッドを使って実装され、リスト要素の中に含まれます。

sizehint が与えられた場合、ストリームの *read()* メソッドに *size* 引数として渡されます。

reset()

状態保持に使われた codec のバッファをリセットします。

ストリームの読み位置を再設定してはならないので注意してください。このメソッドはデコードの際にエラーから復帰できるようにするためのものです。

ここまでで挙げたメソッドの他にも、*StreamReader* では背後にあるストリームの他の全てのメソッドや属性を継承しなければなりません。

StreamReaderWriter オブジェクト

StreamReaderWriter は、読み書き両方に使えるストリームをラップできる便利なクラスです。

lookup() 関数が返すファクトリ関数を使って、インスタンスを生成するという設計です。

class codecs.**StreamReaderWriter**(*stream*, *Reader*, *Writer*, *errors='strict'*)

StreamReaderWriter インスタンスを生成します。*stream* はファイル類似のオブジェクトです。*Reader* と *Writer* は、それぞれ *StreamReader* と *StreamWriter* インタフェースを提供するファクトリ関数かファクトリクラスでなければなりません。エラー処理は、ストリームリーダとライタで定義したものと同じように行われます。

StreamReaderWriter インスタンスは、*StreamReader* クラスと *StreamWriter* クラスを合わせたインタフェースを継承します。元になるストリームからは、他のメソッドや属性を継承します。

StreamRecoder オブジェクト

StreamRecoder はデータをあるエンコーディングから別のエンコーディングに変換します。異なるエンコーディング環境を扱うとき、便利な場合があります。

lookup() 関数が返すファクトリ関数を使って、インスタンスを生成するという設計です。

class codecs.**StreamRecoder**(*stream*, *encode*, *decode*, *Reader*, *Writer*, *errors='strict'*)

双方向変換を実装する *StreamRecoder* インスタンスを生成します。*encode* と *decode* はフロントエンド (`read()` および `write()` を呼び出すコードから見えるデータ) ではたらき、*Reader* と *Writer* はバックエンド (*stream* 内のデータ) ではたらきます。

これらのオブジェクトを使って、たとえば、Latin-1 から UTF-8 への変換、あるいは逆向きの変換を、透過的に行うことができます。

stream 引数はファイルライクオブジェクトでなくてはなりません。

encode 引数と *decode* 引数は `Codec` のインタフェースに忠実でなくてはなりません。*Reader* と *Writer* は、それぞれ *StreamReader* と *StreamWriter* のインターフェースを提供するオブジェクトのファクトリ関数かクラスでなくてはなりません。

エラー処理はストリーム・リーダやライタで定義されている方法と同じように行われます。

StreamRecoder インスタンスは、*StreamReader* と *StreamWriter* クラスを合わせたインタフェースを定義します。また、元のストリームのメソッドと属性も継承します。

7.2.2 エンコーディングと Unicode

Strings are stored internally as sequences of code points in range `0x0--0x10FFFF`. (See **PEP 393** for more details about the implementation.) Once a string object is used outside of CPU and memory, endianness and how these arrays are stored as bytes become an issue. As with other codecs, serialising a string into a sequence of bytes is known as *encoding*, and recreating the string from the sequence of bytes is known as *decoding*. There are a variety of different text serialisation codecs, which are collectivity referred to as *text encodings*.

最も単純なテキストエンコーディング (`'latin-1'` または `'iso-8859-1'`) では、0--255 の範囲にあるコードポイントを `0x0--0xff` のバイトにマップします。つまり、この codec では `U+00FF` 以上のコードポイントを含む文字列オブジェクトをエンコードすることはできません。このようなエンコード処理をしようとすると、次のように *`UnicodeEncodeError`* が送出されます (エラーメッセージの細かところは異なる場合があります。): `UnicodeEncodeError: 'latin-1' codec can't encode character '\u1234' in position 3: ordinal not in range(256)` 。

他のエンコーディングの一群 (charmap エンコーディングと呼ばれます) があり、Unicode コードポイントの別の部分集合と、それらから `0x0--0xff` のバイトへの対応付けを選択したものです。これがどのように行なわれるかを知るには、単にたとえば `encodings/cp1252.py` (主に Windows で使われるエンコーディングです) を開いてみてください。256 文字のひとつの文字列定数があり、どの文字がどのバイト値へ対応付けられるかが示されています。

これらのエンコーディングはすべて、Unicode に定義された 1114112 のコードポイントのうちの 256 だけをエンコードすることができます。Unicode のすべてのコードポイントを格納するための単純で直接的な方法は、各コードポイントを連続する 4 バイトとして格納することです。これには 2 つの可能性があります: ビッグエンディアンまたはリトルエンディアンの順にバイトを格納することです。これら 2 つのエンコーディングはそれぞれ `UTF-32-BE` および `UTF-32-LE` と呼ばれます。それらのデメリットは、例えばリトルエンディアンのマシン上で `UTF-32-BE` を使用すると、エンコードでもデコードでも常にバイト順を交換する必要があることです。`UTF-32` はこの問題を回避します: バイト順は、常に自然なエンディアンに従います。しかし、これらのバイト順が異なるエンディアン性を持つ CPU によって読まれる場合、結局バイト順を交換しなければなりません。`UTF-16` あるいは `UTF-32` バイト列のエンディアン性を検出する目的で、いわゆる BOM (「バイト・オーダー・マーク」) があります。これは Unicode 文字 `U+FEFF` です。この文字はすべての `UTF-16` あるいは `UTF-32` バイト列の前に置くことができます。この文字のバイトが交換されたバージョン (`0xFFFE`) は、Unicode テキストに現われてはならない不正な文字です。したがって、`UTF-16` あるいは `UTF-32` バイト列中の最初の文字が `U+FFFE` であるように見える場合、デコードの際にバイトを交換しなければなりません。不運にも文字 `U+FEFF` は `ZERO WIDTH NO-BREAK SPACE` として別の目的を持っていました: 幅を持たず、単語を分割することを許容しない文字。それは、例えばリガチャアルゴリズムにヒントを与えるために使用することができます。Unicode 4.0 で、`ZERO WIDTH NO-BREAK SPACE` としての `U+FEFF` の使用は廃止予定になりました (この役割は `U+2060` (`WORD JOINER`) によって引き継がれました)。しかしながら、Unicode ソフトウェアは、依然として両方の役割の `U+FEFF` を扱うことができなければなりません: BOM として、エンコードされたバイトのメモリレイアウトを決定する手段であり、一旦バイト列が文字列にデコードされたならば消えます; `ZERO WIDTH NO-BREAK SPACE` として、他の任意の文字のようにデコードされる通常の文字です。

さらにもう一つ Unicode 文字全てをエンコードできるエンコーディングがあり、UTF-8 と呼ばれています。UTF-8 は 8 ビットエンコーディングで、したがって UTF-8 にはバイト順の問題はありません。UTF-8 バイ

ト列の各バイトは二つのパートから成ります。二つはマーカ (上位数ビット) とペイロードです。マーカは 0 ビットから 4 ビットの 1 の列に 0 のビットが一つ続いたものです。Unicode 文字は次のようにエンコードされます (x はペイロードを表わし、連結されると一つの Unicode 文字を表わします):

範囲	エンコーディング
U-00000000 ... U-0000007F	0xxxxxxx
U-00000080 ... U-000007FF	110xxxxx 10xxxxxx
U-00000800 ... U-0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
U-00010000 ... U-0010FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Unicode 文字の最下位ビットとは最も右にある x のビットです。

UTF-8 は 8 ビットエンコーディングなので BOM は必要とせず、デコードされた文字列中の U+FEFF は (たとえ最初の文字であったとしても) ZERO WIDTH NO-BREAK SPACE として扱われます。

外部からの情報無しには、文字列のエンコーディングにどのエンコーディングが使われたのか信頼できる形で決定することは不可能です。どの charmap エンコーディングもどんなランダムなバイト列でもデコードできます。しかし UTF-8 ではそれは可能ではありません。任意のバイト列を許さないような構造を持っているからです。UTF-8 エンコーディングであることを検知する信頼性を向上させるために、Microsoft は Notepad プログラム用に UTF-8 の変種 (Python 2.5 では "utf-8-sig" と呼んでいます) を考案しました。Unicode 文字がファイルに書き込まれる前に UTF-8 でエンコードした BOM (バイト列では 0xef, 0xbb, 0xbf のように見えます) が書き込まれます。このようなバイト値で charmap エンコードされたファイルが始まることはほとんどあり得ない (たとえば iso-8859-1 では

LATIN SMALL LETTER I WITH DIAERESIS
RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
INVERTED QUESTION MARK

のようになる) ので、utf-8-sig エンコーディングがバイト列から正しく推測される確率を高めます。つまりここでは BOM はバイト列を生成する際のバイト順を決定できるように使われているのではなく、エンコーディングを推測する助けになる印として使われているのです。utf-8-sig codec はエンコーディングの際ファイルに最初の 3 文字として 0xef, 0xbb, 0xbf を書き込みます。デコーディングの際はファイルの先頭に現れたこれら 3 バイトはスキップします。UTF-8 では BOM の使用は推奨されておらず、一般的には避けるべきです。

7.2.3 標準エンコーディング

Python には数多くの codec が組み込みで付属します。これらは C 言語の関数、対応付けを行うテーブルの両方で提供されています。以下のテーブルでは codec と、いくつかの良く知られている別名と、エンコーディングが使われる言語を列挙します。別名のリスト、言語のリストともしらみつぶしに網羅されているわけではありません。大文字と小文字、またはアンダースコアの代りにハイフンにしただけの綴りも有効な別名です; そのため、例えば 'utf-8' は 'utf_8' codec の正当な別名です。

CPython implementation detail: いくつかの一般的なエンコーディングは、パフォーマンスを改善するために codec の検索機構を回避することがあります。このような最適化の機会を認識するのは、CPython

の限定された (大文字小文字を区別しない) 別名に対してのみです: utf-8, utf8, latin-1, latin1, iso-8859-1, iso8859-1, mbcs (Windows のみ), ascii, us-ascii, utf-16, utf16, utf-32, utf32 およびダッシュの代わりにアンダースコアを用いたもの。これらのエンコーディングの別のつづりを使用すると実行時間の低下を招くかもしれません。

バージョン 3.6 で変更: us-ascii に対して最適化の機会が認識されるようになりました。

多くの文字セットは同じ言語をサポートしています。これらの文字セットは個々の文字 (例えば、EURO SIGN がサポートされているかどうか) や、文字のコード部分への割り付けが異なります。特に欧州言語では、典型的に以下の変種が存在します:

- ISO 8859 コードセット
- Microsoft Windows コードページで、8859 コード形式から導出されているが、制御文字を追加のグラフィック文字と置き換えたもの
- IBM EBCDIC コードページ
- ASCII 互換の IBM PC コードページ

Codec	別名	言語
ascii	646, us-ascii	英語
big5	big5-tw, csbig5	繁体字中国語
big5hkscs	big5-hkscs, hkscs	繁体字中国語
cp037	IBM037, IBM039	英語
cp273	273, IBM273, csIBM273	ドイツ語 バージョン 3.4 で追加.
cp424	EBCDIC-CP-HE, IBM424	ヘブライ語
cp437	437, IBM437	英語
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	西ヨーロッパ言語
cp720		アラビア語
cp737		ギリシャ語
cp775	IBM775	バルト沿岸国
cp850	850, IBM850	西ヨーロッパ言語
cp852	852, IBM852	中央および東ヨーロッパ
cp855	855, IBM855	ブルガリア、ベラルーシ、マケドニア、ロシア、セルビア
cp856		ヘブライ語
cp857	857, IBM857	トルコ語
cp858	858, IBM858	西ヨーロッパ言語
cp860	860, IBM860	ポルトガル語
cp861	861, CP-IS, IBM861	アイスランド語
cp862	862, IBM862	ヘブライ語
cp863	863, IBM863	カナダ

次のページに続く

表 1 – 前のページからの続き

Codec	別名	言語
cp864	IBM864	アラビア語
cp865	865, IBM865	デンマーク、ノルウェー
cp866	866, IBM866	ロシア語
cp869	869, CP-GR, IBM869	ギリシャ語
cp874		タイ語
cp875		ギリシャ語
cp932	932, ms932, mskanji, ms-kanji	日本語
cp949	949, ms949, uhc	韓国語
cp950	950, ms950	繁体字中国語
cp1006		Urdu
cp1026	ibm1026	トルコ語
cp1125	1125, ibm1125, cp866u, ruscii	ウクライナ語 バージョン 3.4 で追加.
cp1140	ibm1140	西ヨーロッパ言語
cp1250	windows-1250	中央および東ヨーロッパ
cp1251	windows-1251	ブルガリア、ベラルーシ、マケドニア、ロシア、セルビア
cp1252	windows-1252	西ヨーロッパ言語
cp1253	windows-1253	ギリシャ語
cp1254	windows-1254	トルコ語
cp1255	windows-1255	ヘブライ語
cp1256	windows-1256	アラビア語
cp1257	windows-1257	バルト沿岸国
cp1258	windows-1258	ベトナム
euc_jp	eucjp, ujis, u-jis	日本語
euc_jis_2004	jisx0213, eucjis2004	日本語
euc_jisx0213	eucjisx0213	日本語
euc_kr	euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001	韓国語
gb2312	chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58	簡体字中国語
gbk	936, cp936, ms936	Unified Chinese
gb18030	gb18030-2000	Unified Chinese
hz	hzgb, hz-gb, hz-gb-2312	簡体字中国語
iso2022_jp	csiso2022jp, iso2022jp, iso-2022-jp	日本語
iso2022_jp_1	iso2022jp-1, iso-2022-jp-1	日本語

次のページに続く

表 1 – 前のページからの続き

Codec	別名	言語
iso2022_jp_2	iso2022jp-2, iso-2022-jp-2	日本語, 韓国語, 簡体字中国語, 西欧, ギリシャ語
iso2022_jp_2004	iso2022jp-2004, iso-2022-jp-2004	日本語
iso2022_jp_3	iso2022jp-3, iso-2022-jp-3	日本語
iso2022_jp_ext	iso2022jp-ext, iso-2022-jp-ext	日本語
iso2022_kr	csiso2022kr, iso2022kr, iso-2022-kr	韓国語
latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	西ヨーロッパ言語
iso8859_2	iso-8859-2, latin2, L2	中央および東ヨーロッパ
iso8859_3	iso-8859-3, latin3, L3	エスペラント、マルタ
iso8859_4	iso-8859-4, latin4, L4	バルト沿岸国
iso8859_5	iso-8859-5, cyrillic	ブルガリア、ベラルーシ、マケドニア、ロシア、セルビア
iso8859_6	iso-8859-6, arabic	アラビア語
iso8859_7	iso-8859-7, greek, greek8	ギリシャ語
iso8859_8	iso-8859-8, hebrew	ヘブライ語
iso8859_9	iso-8859-9, latin5, L5	トルコ語
iso8859_10	iso-8859-10, latin6, L6	北欧語
iso8859_11	iso-8859-11, thai	タイ語
iso8859_13	iso-8859-13, latin7, L7	バルト沿岸国
iso8859_14	iso-8859-14, latin8, L8	ケルト語
iso8859_15	iso-8859-15, latin9, L9	西ヨーロッパ言語
iso8859_16	iso-8859-16, latin10, L10	南東ヨーロッパ
johab	cp1361, ms1361	韓国語
koi8_r		ロシア語
koi8_t		タジク バージョン 3.5 で追加.
koi8_u		ウクライナ語
kz1048	kz_1048, strk1048_2002, rk1048	カザフ バージョン 3.5 で追加.
mac_cyrillic	maccyrillic	ブルガリア、ベラルーシ、マケドニア、ロシア、セルビア
mac_greek	macgreek	ギリシャ語
mac_iceland	maciceland	アイスランド語
mac_latin2	maclatin2, maccentraleurope	中央および東ヨーロッパ
mac_roman	macroman, macintosh	西ヨーロッパ言語
mac_turkish	macturkish	トルコ語

次のページに続く

表 1 – 前のページからの続き

Codec	別名	言語
ptcp154	csptcp154, pt154, cp154, cyrillic-asian	カザフ
shift_jis	csshiftjis, shiftjis, sjis, s_jis	日本語
shift_jis_2004	shiftjis2004, sjis_2004, sjis2004	日本語
shift_jisx0213	shiftjisx0213, sjisx0213, s_jisx0213	日本語
utf_32	U32, utf32	全ての言語
utf_32_be	UTF-32BE	全ての言語
utf_32_le	UTF-32LE	全ての言語
utf_16	U16, utf16	全ての言語
utf_16_be	UTF-16BE	全ての言語
utf_16_le	UTF-16LE	全ての言語
utf_7	U7, unicode-1-1-utf-7	全ての言語
utf_8	U8, UTF, utf8, cp65001	全ての言語
utf_8_sig		全ての言語

バージョン 3.4 で変更: utf-16* と utf-32* のエンコーダは、サロゲートコードポイント (`U+D800--U+DFFF`) がエンコードされることを許可しなくなりました。utf-32* デコーダは、サロゲートコードポイントに対応するバイト列をデコードしなくなりました。

バージョン 3.8 で変更: `cp65001` is now an alias to `utf_8`.

7.2.4 Python 特有のエンコーディング

予め定義された codec のいくつかは Python 特有のものなので、それらの codec 名は Python の外では無意味なものとなります。以下に、想定されている入出力のタイプに基づいて、それらを表にしました（テキストエンコーディングは codec の最も一般的な使用例ですが、その根底にある codec 基盤は、ただのテキストエンコーディングというよりも、任意のデータの変換をサポートしていることに注意してください)。非対称的な codec については、記載された意味がエンコーディングの方向を説明しています。

テキストエンコーディング

次の codec では、Unicode におけるテキストエンコーディングと同様に、*str* から *bytes* へのエンコードと、*bytes-like object* から *str* へのデコードを提供します。

Codec	別名	意味
idna		RFC 3490 の実装です。*encodings.idna* も参照してください。errors='strict' のみがサポートされています。
mbcs	ansi, dbcs	Windows のみ: 被演算子を ANSI コードページ (CP_ACP) に従ってエンコードします。
oem		Windows のみ: 被演算子を OEM コードページ (CP_OEMCP) に従ってエンコードします。 バージョン 3.6 で追加.
palmos		PalmOS 3.5 のエンコーディングです。
punycode		RFC 3492 の実装です。ステートフル codecs は、サポートされません。
raw_unicode_escape		別のコードポイントに \uXXXX と \UXXXXXXXX を使用する Latin-1 エンコーディングです。既存のバックスラッシュは、いかなる方法でもエスケープされません。Python の pickle プロトコルで使用されます。
undefined		空文字列を含む全ての変換に対して例外を送出します。エラーハンドラは無視されます。
unicode_escape		ASCII でエンコードされた Python ソースコード内の、Unicode リテラルに適したエンコーディングです。ただし、引用符はエスケープされません。Latin-1 ソースコードからデコードします。実際には、Python のソースコードはデフォルトでは UTF-8 を使用することに注意してください。

バージョン 3.8 で変更: "unicode_internal" codec is removed.

バイナリ変換 (Binary Transforms)

以下の codec は、*bytes-like object* から *bytes* マッピングへのバイナリ変換を提供します。*bytes.decode()* はこの変換をサポートしておらず、*str* を出力するだけです。

Codec	別名	意味	エンコーダ / デコーダ
base64_codec[*1]	base64, base_64	被演算子をマルチラインの MIME base64 に変換します (結果は常に末尾の '\n' を含みます)。バージョン 3.4 で変更: 任意の *bytes-like object* をエンコードとデコード用の入力として受け取ります。	*base64.encodebytes()* / *base64.decodebytes()*
bz2_codec	bz2	被演算子を bz2 を使って圧縮します。	*bz2.compress()* / *bz2.decompress()*
hex_codec	hex	被演算子をバイトあたり 2 桁の 16 進数の表現に変換します。	*binascii.b2a_hex()* / *binascii.a2b_hex()*
quopri_codec	quopri, quoted-printable, quoted_printable	被演算子を MIME quoted printable 形式に変換します。	quotetabs=True を指定した *quopri.encode()* / *quopri.decode()*
uu_codec	uu	被演算子を uuencode を用いて変換します。	*uu.encode()* / *uu.decode()*
zlib_codec	zip, zlib	被演算子を gzip を用いて圧縮します。	*zlib.compress()* / *zlib.decompress()*

バージョン 3.2 で追加: バイナリ変換が復活しました。(訳注: 2.x にはあったものが 3.0 で削除されていた。)

バージョン 3.4 で変更: バイナリ変換のエイリアスが復活しました。(訳注: 2.x にはあったエイリアス。3.2 でエイリアスは復活しなかった。)

[*1] バイト様オブジェクト に加えて、'base64_codec' も *str* の ASCII のみのインスタンスをデコード用に受け入れるようになりました

テキスト変換 (Text Transforms)

以下の codec は、*str* から *str* マッピングへのテキスト変換を提供します。*str.encode()* はこの変換をサポートしておらず、*bytes* を出力するだけです。

Codec	別名	意味
rot_13	rot13	被演算子のシーザー暗号 (Caesar-cypher) を返します。

バージョン 3.2 で追加: `rot_13` テキスト変換が復活しました。(訳注: 2.x にはあったものが 3.0 で削除されていた。)

バージョン 3.4 で変更: `rot13` エイリアスが復活しました。(訳注: 2.x にはあったエイリアス。3.2 でエイリアスは復活しなかった。)

7.2.5 `encodings.idna` --- アプリケーションにおける国際化ドメイン名 (IDNA)

このモジュールでは **RFC 3490** (アプリケーションにおける国際化ドメイン名、IDNA: Internationalized Domain Names in Applications) および **RFC 3492** (Nameprep: 国際化ドメイン名 (IDN) のための stringprep プロファイル) を実装しています。このモジュールは `punycode` エンコーディングおよび *stringprep* の上に構築されています。

If you need the IDNA 2008 standard from **RFC 5891** and **RFC 5895**, use the third-party *idna module <https://pypi.org/project/idna/>_*.

これらの RFC はともに、非 ASCII 文字の入ったドメイン名をサポートするためのプロトコルを定義しています。(`www.Alliancefrançaise.nu` のような) 非 ASCII 文字を含むドメイン名は、ASCII と互換性のあるエンコーディング (ACE、`www.xn--alliancefranaise-npb.nu` のような形式) に変換されます。ドメイン名の ACE 形式は、DNS クエリ、HTTP *Host* フィールドなどといった、プロトコル中で任意の文字を使えないような全ての局面で用いられます。この変換はアプリケーション内で行われます; 可能ならユーザからは不可視となります: アプリケーションは Unicode ドメインラベルをネットワークに載せる際に IDNA に、ACE ドメインラベルをユーザに提供する前に Unicode に、それぞれ透過的に変換しなければなりません。

Python ではこの変換をいくつかの方法でサポートします: `idna` codec は Unicode と ACE 間の変換を行い、入力文字列を **RFC 3490 の section 3.1** で定義されている区切り文字に基づいてラベルに分解し、各ラベルを要求通りに ACE に変換します。逆に、入力のバイト文字列を `.` 区切り文字でラベルに分解し、ACE ラベルを Unicode に変換します。さらに、*socket* モジュールは Unicode ホスト名を ACE に透過的に変換するため、アプリケーションはホスト名を *socket* モジュールに渡す際にホスト名の変換に煩わされることがありません。その上で、ホスト名を関数パラメタとして持つ、*http.client* や *ftplib* のようなモジュールでは Unicode ホスト名を受理します (*http.client* でもまた、*Host* フィールドにある IDNA ホスト名を、フィールド全体を送信する場合に透過的に送信します)。

(逆引きなどによって) ネットワーク越しにホスト名を受信する際、Unicode への自動変換は行われません: こうしたホスト名をユーザに提供したいアプリケーションでは、Unicode にデコードしてやる必要があります。

encodings.idna ではまた、nameprep 手続きを実装しています。nameprep はホスト名に対してある正規化を行って、国際化ドメイン名で大小文字を区別しないようにするとともに、類似の文字を一元化します。

nameprep 関数は必要なら直接使うこともできます。

encodings.idna.nameprep(*label*)
: *label* を nameprep したバージョンを返します。現在の実装ではクエリ文字列を仮定しているので、`AllowUnassigned` は真です。

encodings.idna.ToASCII(*label*)
: **RFC 3490** 仕様に従ってラベルを ASCII に変換します。`UseSTD3ASCIIRules` は偽であると仮定します。

encodings.idna.ToUnicode(*label*)
: **RFC 3490** 仕様に従ってラベルを Unicode に変換します。

7.2.6 encodings.mbcs --- Windows ANSI コードページ

This module implements the ANSI codepage (CP_ACP).

利用可能な環境: Windows のみ。

バージョン 3.3 で変更: 任意のエラーハンドラのサポート。

バージョン 3.2 で変更: 3.2 以前は *errors* 引数は無視されました; エンコードには常に `'replace'` が、デコードには `'ignore'` が使われました。

7.2.7 encodings.utf_8_sig --- BOM 印付き UTF-8

このモジュールは UTF-8 codec の変種を実装します。エンコーディング時は、UTF-8 でエンコードしたバイト列の前に UTF-8 でエンコードした BOM を追加します。これは内部状態を持つエンコーダで、この動作は (バイトストリームの最初の書き込み時に) 一度だけ行なわれます。デコーディング時は、データの最初に UTF-8 でエンコードされた BOM があれば、それをスキップします。

第

EIGHT

データ型

この章で解説されるモジュールは日付や時間、型が固定された配列、ヒープキュー、両端キュー、列挙型のような種々の特殊なデータ型を提供します。

Python にはその他にもいくつかの組み込みデータ型があります。特に、*dict* 、*list* 、*set* 、*frozenset* 、そして *tuple* があります。*str* クラスは Unicode データを扱うことができ、*bytes* と *bytearray* クラスはバイナリデータを扱うことができます。

この章では以下のモジュールが記述されています:

8.1 datetime --- 基本的な日付型および時間型

ソースコード: Lib/datetime.py

---

datetime モジュールは、日付や時刻を操作するためのクラスを提供しています。

日付や時刻に対する算術がサポートされている一方、実装では出力のフォーマットや操作のための効率的な属性の抽出に重点を置いています。

参考:

calendar **モジュール**　汎用のカレンダー関連関数。

time **モジュール**　時刻へのアクセスと変換。

dateutil **パッケージ**　拡張タイムゾーンと構文解析サポートのあるサードパーティーライブラリ。

8.1.1 Aware オブジェクトと Naive オブジェクト

日時のオブジェクトは、それらがタイムゾーンの情報を含んでいるかどうかによって "aware" あるいは "naive" に分類されます。

タイムゾーンや夏時間の情報のような、アルゴリズム的で政治的な適用可能な時間調節に関する知識を持っているため、**aware** オブジェクトは他の aware オブジェクトとの相対関係を特定できます。aware オブジェク

トは解釈の余地のない特定の実時刻を表現します。[*1]

naive オブジェクトには他の日付時刻オブジェクトとの相対関係を把握するのに足る情報が含まれません。あるプログラム内の数字がメートルを表わしているのか、マイルなのか、それとも質量なのかがプログラムによって異なるように、naive オブジェクトが協定世界時 (UTC) なのか、現地時間なのか、それとも他のタイムゾーンなのかはそのプログラムに依存します。Naive オブジェクトはいくつかの現実的な側面を無視してしまうというコストを無視すれば、簡単に理解でき、うまく利用することができます。

aware オブジェクトを必要とするアプリケーションのために、*datetime* と *time* オブジェクトは追加のタイムゾーン情報の属性 `tzinfo` を持ちます。*tzinfo* には抽象クラス *tzinfo* のサブクラスのインスタンスを設定できます。これらの *tzinfo* オブジェクトは UTC 時間からのオフセットやタイムゾーンの名前、夏時間が実施されるかどうかの情報を保持しています。

ただ一つの具象 *tzinfo* クラスである *timezone* クラスが *datetime* モジュールで提供されています。*timezone* クラスは、UTC からのオフセットが固定である単純なタイムゾーン（例えば UTC それ自体)、および北アメリカにおける東部標準時（EST）／東部夏時間（EDT）のような単純ではないタイムゾーンの両方を表現できます。より深く詳細までタイムゾーンをサポートするかはアプリケーションに依存します。世界中の時刻の調整を決めるルールは合理的というよりかは政治的なもので、頻繁に変わり、UTC を除くと都合のよい基準というものはありません。

8.1.2 定数

datetime モジュールでは以下の定数を公開しています:

datetime.MINYEAR
: *date* や *datetime* オブジェクトで許されている、年を表現する最小の数字です。*MINYEAR* は `1` です。

datetime.MAXYEAR
: *date* や *datetime* オブジェクトで許されている、年を表現する最大の数字です。*MAXYEAR* は `9999` です。

8.1.3 利用可能なデータ型

class datetime.date
: 理想的な naive な日付で、これまでもこれからも現在のグレゴリオ暦 (Gregorian calender) が有効であることを仮定しています。属性は *year*, *month*, および *day* です。

class datetime.time
: 理想的な時刻で、特定の日から独立しており、毎日が厳密に 24*60*60 秒であると仮定しています。(”うるう秒: leap seconds” の概念はありません。) 属性は *hour*, *minute*, *second*, *microsecond*, および *tzinfo* です。

class datetime.datetime
: 日付と時刻を組み合わせたものです。属性は *year*, *month*, *day*, *hour*, *minute*, *second*, *microsecond*,

[*1] もし相対性理論の効果を無視するならば、ですが

および *tzinfo* です。

class datetime.timedelta

date, *time*, あるいは *datetime* クラスの二つのインスタンス間の時間差をマイクロ秒精度で表す経過時間値です。

class datetime.tzinfo

タイムゾーン情報オブジェクトの抽象基底クラスです。*datetime* および *time* クラスで用いられ、カスタマイズ可能な時刻修正の概念 (たとえばタイムゾーンや夏時間の計算) を提供します。

class datetime.timezone

tzinfo 抽象基底クラスを UTC からの固定オフセットとして実装するクラスです。

バージョン 3.2 で追加.

これらの型のオブジェクトは変更不可能 (immutable) です。

サブクラスの関係は以下のようになります:

```
object
    timedelta
    tzinfo
        timezone
    time
    date
        datetime
```

共通の特徴

date 型、*datetime* 型、*time* 型、*timezone* 型には共通する特徴があります:

- これらの型のオブジェクトは変更不可能 (immutable) です。
- これらの型のオブジェクトはハッシュ可能であり、辞書のキーとして使えることになります。
- これらの型のオブジェクトは *pickle* モジュールを利用して効率的な pickle 化をサポートしています。

オブジェクトが Aware なのか Naive なのかの判断

date 型のオブジェクトは常に naive です。

time 型あるいは *datetime* 型のオブジェクトは aware か naive のどちらかです。

次の条件を両方とも満たす場合、*datetime* オブジェクト *d* は aware です:

1. `d.tzinfo` が `None` でない
2. `d.tzinfo.utcoffset(d)` が `None` を返さない

どちらかを満たさない場合は、*d* は naive です。

次の条件を両方とも満たす場合、*time* オブジェクト *t* は aware です:

1. `t.tzinfo` が `None` でない
2. `t.tzinfo.utcoffset(None)` が `None` を返さない

どちらかを満たさない場合は、*t* は naive です。

aware なオブジェクトと naive なオブジェクトの区別は *timedelta* オブジェクトにはあてはまりません。

8.1.4 timedelta オブジェクト

timedelta オブジェクトは経過時間、すなわち二つの日付や時刻間の差を表します。

class datetime.**timedelta**(*days=0*, *seconds=0*, *microseconds=0*, *milliseconds=0*, *minutes=0*, *hours=0*, *weeks=0*)

全ての引数がオプションで、デフォルト値は `0` です。引数は整数、浮動小数点数でもよく、正でも負でもかまいません。

days, *seconds*, *microseconds* だけが内部的に保持されます。引数は以下のようにして変換されます:

- 1 ミリ秒は 1000 マイクロ秒に変換されます。
- 1 分は 60 秒に変換されます。
- 1 時間は 3600 秒に変換されます。
- 1 週間は 7 日に変換されます。

さらに、値が一意に表されるように days, seconds, microseconds が以下のように正規化されます

- `0 <= microseconds < 1000000`
- `0 <= seconds < 3600*24` (一日中の秒数)
- `-999999999 <= days <= 999999999`

次の例は、*days*, *seconds*, *microseconds* に加えて任意の引数がどう ” 集約” され、最終的に 3 つの属性に正規化されるかの説明をしています:

```
>>> from datetime import timedelta
>>> delta = timedelta(
...     days=50,
...     seconds=27,
...     microseconds=10,
...     milliseconds=29000,
...     minutes=5,
...     hours=8,
...     weeks=2
... )
>>> # Only days, seconds, and microseconds remain
>>> delta
datetime.timedelta(days=64, seconds=29156, microseconds=10)
```

引数のいずれかが浮動小数点であり、小数のマイクロ秒が存在する場合、小数のマイクロ秒は全ての引数から一度取り置かれ、それらの和は最近接偶数のマイクロ秒に丸められます。浮動小数点の引数がない場合、値の変換と正規化の過程は厳密な (失われる情報がない) ものとなります。

日の値を正規化した結果、指定された範囲の外側になった場合には、*OverflowError* が送出されます。

負の値を正規化すると、最初は混乱するような値になります。例えば:

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

以下にクラス属性を示します:

timedelta.min
: 最小の値を表す *timedelta* オブジェクトで、timedelta(-999999999) です。

timedelta.max
: 最大の値を表す *timedelta* オブジェクトで、timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999) です。

timedelta.resolution
: *timedelta* オブジェクトが等しくならない最小の時間差で、timedelta(microseconds=1) です。

正規化のために、timedelta.max > -timedelta.min となるので注意してください。-timedelta.max は *timedelta* オブジェクトとして表現することができません。

インスタンスの属性 (読み出しのみ):

属性	値
days	両端値を含む -999999999 から 999999999 の間
seconds	両端値を含む 0 から 86399 の間
microseconds	両端値を含む 0 から 999999 の間

サポートされている演算を以下に示します:

演算	結果
`t1 = t2 + t3`	*t2* と *t3* の和。演算後、*t1*-*t2* == *t3* および *t1*-*t3* == *t2* は真になります。(1)
`t1 = t2 - t3`	*t2* と *t3* の差分です。演算後、*t1* == *t2* - *t3* および *t2* == *t1* + *t3* は真になります。(1)(6)
`t1 = t2 * i` または `t1 = i * t2`	時間差と整数の積。演算後、*t1* // i == *t2* は `i != 0` であれば真となります。
	一般的に、*t1* * i == *t1* * (i-1) + *t1* は真となります。(1)
`t1 = t2 * f` または `t1 = f * t2`	時間差と浮動小数点の積。結果は最近接偶数への丸めを利用して最も近い timedelta.resolution の倍数に丸められます。
`f = t2 / t3`	*t2* を *t3* で除算 (3) したもの。*float* オブジェクトを返します。
`t1 = t2 / f` または `t1 = t2 / i`	時間差を浮動小数点や整数で除したもの。結果は最近接偶数への丸めを利用して最も近い timedelta.resolution の倍数に丸められます。
`t1 = t2 // i` または `t1 = t2 // t3`	floor が計算され、余りは (もしあれば) 捨てられます。後者の場合、整数が返されます。(3)
`t1 = t2 % t3`	剰余が *timedelta* オブジェクトとして計算されます。(3)
`q, r = divmod(t1, t2)`	商と剰余が計算されます: `q = t1 // t2` (3) と `r = t1 % t2` 。q は整数で r は *timedelta* オブジェクトです。
`+t1`	同じ値を持つ *timedelta* オブジェクトを返します。(2)
`-t1`	*timedelta*(*-t1.days*, *-t1.seconds*, *-t1.microseconds*)、および *t1** -1 と同じです。(1)(4)
`abs(t)`	`t.days >= 0` のときには *+t*, `t.days < 0` のときには *-t* となります。(2)
`str(t)`	`[D day[s], ][H]H:MM:SS[.UUUUUU]` という形式の文字列を返します。`t` が負の値の場合は `D` は負の値となります。(5)
`repr(t)`	*timedelta* オブジェクトの文字列表現を返します。その文字列は、正規の属性値を持つコンストラクタ呼び出しのコードになっています。

注釈:

(1) この演算は正確ですが、オーバフローするかもしれません。

(2) この演算は正確であり、オーバフローし得ません。

(3) 0 による除算は *ZeroDivisionError* を送出します。

(4) *-timedelta.max* は *timedelta* オブジェクトで表現することができません。

(5) *timedelta* オブジェクトの文字列表現は内部表現に類似した形に正規化されます。そのため負の timedelta は少し変な結果になります。例えば:

```
>>> timedelta(hours=-5)
datetime.timedelta(days=-1, seconds=68400)
>>> print(_)
-1 day, 19:00:00
```

(6) t3 が `timedelta.max` のときを除けば、式 `t2 - t3` は常に、式 `t2 + (-t3)` と同等です。t3 が `timedelta.max` の場合、前者の式は結果の値が出ますが、後者はオーバーフローを起こします。

上に列挙した操作に加え *timedelta* オブジェクトは *date* および *datetime* オブジェクトとの間で加減算をサポートしています (下を参照してください)。

バージョン 3.2 で変更: *timedelta* オブジェクトの別の *timedelta* オブジェクトによる、切り捨ての割り算と真の値の割り算、および剰余演算子と *divmod()* 関数がサポートされるようになりました。*timedelta* オブジェクトと *float* オブジェクトの真の値の割り算と掛け算がサポートされるようになりました。

timedelta オブジェクトどうしの比較が、注意書き付きでサポートされました。

== および != の比較は、比較されているオブジェクトの型が何であれ、**常に** *bool* を返します:

```
>>> from datetime import timedelta
>>> delta1 = timedelta(seconds=57)
>>> delta2 = timedelta(hours=25, seconds=2)
>>> delta2 != delta1
True
>>> delta2 == 5
False
```

(< and > などの) それ以外の全ての比較で、*timedelta* オブジェクトを異なる型のオブジェクトと比較したときは *TypeError* が送出されます:

```
>>> delta2 > delta1
True
>>> delta2 > 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'datetime.timedelta' and 'int'
```

ブール演算コンテキストでは、*timedelta* オブジェクトは timedelta(0) に等しくない場合かつそのときに限り真となります。

インスタンスメソッド:

timedelta.total_seconds()

この期間に含まれるトータルの秒数を返します。td / timedelta(seconds=1) と等価です。秒以外の期間の単位では、直接に除算する形式 (例えば td / timedelta(microseconds=1)) が使われます。

非常に長い期間 (多くのプラットフォームでは 270 年以上) については、このメソッドはマイクロ秒の精度を失うことがあることに注意してください。

バージョン 3.2 で追加.

使用例: timedelta

正規化の追加の例です:

```
>>> # Components of another_year add up to exactly 365 days
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                          minutes=50, seconds=600)
>>> year == another_year
True
>>> year.total_seconds()
31536000.0
```

timedelta の計算の例です:

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> ten_years = 10 * year
>>> ten_years
datetime.timedelta(days=3650)
>>> ten_years.days // 365
10
>>> nine_years = ten_years - year
>>> nine_years
datetime.timedelta(days=3285)
>>> three_years = nine_years // 3
>>> three_years, three_years.days // 365
(datetime.timedelta(days=1095), 3)
```

8.1.5 date オブジェクト

date オブジェクトは、両方向に無期限に拡張された現在のグレゴリオ暦という理想化された暦の日付 (年月日) を表します。

1 年 1 月 1 日は日番号 1、1 年 1 月 2 日は日番号 2 と呼ばれ、他も同様です。[*2]

class datetime.**date**(*year*, *month*, *day*)

全ての引数が必須です。引数は整数で、次の範囲に収まっていなければなりません:

- `MINYEAR <= year <= MAXYEAR`
- `1 <= month <= 12`
- `1 <= day <=` **指定された月と年における日数**

範囲を超えた引数を与えた場合、*ValueError* が送出されます。

[*2] この暦法は、全ての計算における基本カレンダーである、Dershowitz と Reingold の書籍 *Calendrical Calculations* における先発グレゴリオ暦 ("proleptic Gregorian") の定義に一致します。先発グレゴリオ暦の序数とその他多くの暦法どうしの変換アルゴリズムについては、この書籍を参照してください。

他のコンストラクタ、および全てのクラスメソッドを以下に示します:

classmethod date.today()
現在のローカルな日付を返します。

date.fromtimestamp(time.time()) と等価です。

classmethod date.fromtimestamp(*timestamp*)
time.time() で返されるような POSIX タイムスタンプに対応するローカルな日付を返します。

timestamp がプラットフォームの C 関数 localtime() がサポートする値の範囲から外れていた場合、*OverflowError* を送出するかもしれません。また localtime() 呼び出しが失敗した場合には *OSError* を送出するかもしれません。この範囲は通常は 1970 年から 2038 年までに制限されています。タイムスタンプの表記にうるう秒を含める非 POSIX なシステムでは、うるう秒は *fromtimestamp()* では無視されます。

バージョン 3.3 で変更: timestamp がプラットフォームの C 関数 localtime() のサポートする値の範囲から外れていた場合、*ValueError* ではなく *OverflowError* を送出するようになりました。localtime() の呼び出し失敗で *ValueError* ではなく *OSError* を送出するようになりました。

classmethod date.fromordinal(*ordinal*)
先発グレゴリオ暦による序数に対応する日付を返します。1 年 1 月 1 日が序数 1 となります。

1 <= ordinal <= date.max.toordinal() でない場合、*ValueError* が送出されます。任意の日付 *d* に対し、date.fromordinal(d.toordinal()) == d となります。

classmethod date.fromisoformat(*date_string*)
YYYY-MM-DD という書式で与えられた *date_string* に対応する *date* を返します

```
>>> from datetime import date
>>> date.fromisoformat('2019-12-04')
datetime.date(2019, 12, 4)
```

この関数は *date.isoformat()* の逆関数です。YYYY-MM-DD という書式のみをサポートしています。

バージョン 3.7 で追加.

classmethod date.fromisocalendar(*year*, *week*, *day*)
年月日で指定された ISO 暦の日付に対応する *date* を返します。この関数は *date.isocalendar()* 関数の逆関数です。

バージョン 3.8 で追加.

以下にクラス属性を示します:

date.min
表現できる最も古い日付で、date(MINYEAR, 1, 1) です。

date.max
表現できる最も新しい日付で、date(MAXYEAR, 12, 31) です。

date.resolution
　等しくない日付オブジェクト間の最小の差で、timedelta(days=1) です。

インスタンスの属性 (読み出しのみ):

date.year
　両端値を含む *MINYEAR* から *MAXYEAR* までの値です。

date.month
　両端値を含む 1 から 12 までの値です。

date.day
　1 から与えられた月と年における日数までの値です。

サポートされている演算を以下に示します:

演算	結果
date2 = date1 + timedelta	*date2* は *date1* から timedelta.days 日だけ移動した日付です。(1)
date2 = date1 - timedelta	date2 + timedelta == date1 であるような日付 *date2* を計算します。(2)
timedelta = date1 - date2	(3)
date1 < date2	*date1* が時刻として *date2* よりも前を表す場合に、*date1* は *date2* よりも小さいと見なされます。(4)

注釈:

(1) *date2* は、timedelta.days > 0 の場合は進む方向に、timedelta.days < 0 の場合は戻る方向に移動します。演算後は date2 - date1 == timedelta.days が成立します。timedelta.seconds および timedelta.microseconds は無視されます。date2.year が *MINYEAR* になってしまったり、*MAXYEAR* より大きくなってしまう場合には *OverflowError* が送出されます。

(2) timedelta.seconds と timedelta.microseconds は無視されます。

(3) この演算は厳密で、オーバフローしません。timedelta.seconds および timedelta.microseconds は 0 で、演算後には date2 + timedelta == date1 となります。

(4) 言い換えると、date1 < date2 は date1.toordinal() < date2.toordinal() と同等です。日付の比較は、比較相手が *date* オブジェクトでない場合には、*TypeError* を送出します。ただし、比較相手に timetuple() 属性がある場合は、NotImplemented が代わりに送出されます。このフックによって、他の種類の日付オブジェクトに、違う型どうしの比較処理を実装できる可能性が生まれます。相手が timetuple() 属性を持っていない場合に *date* と違う型のオブジェクトと比較すると、== または != の比較でない限り *TypeError* が送出されます。後者の場合では、それぞれ *False* および *True* が返されます。

ブール演算コンテキストでは、全ての *time* オブジェクトは真とみなされます。

インスタンスメソッド:

date.**replace**(*year=self.year*, *month=self.month*, *day=self.day*)

キーワード引数で指定されたパラメータが置き換えられることを除き、同じ値を持つ *date* オブジェクトを返します。

以下はプログラム例です:

```
>>> from datetime import date
>>> d = date(2002, 12, 31)
>>> d.replace(day=26)
datetime.date(2002, 12, 26)
```

date.**timetuple**()

time.localtime() が返すような *time.struct_time* を返します。

時分秒が 0 で、DST フラグが -1 です。

`d.timetuple()` は次の式と等価です:

```
time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1))
```

ここで、`yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` は本年の 1 月 1 日を 1 としたときの日付番号です。

date.**toordinal**()

先発グレゴリオ暦における日付序数を返します。1 年の 1 月 1 日が序数 1 となります。任意の *date* オブジェクト *d* について、`date.fromordinal(d.toordinal()) == d` となります。

date.**weekday**()

月曜日を 0、日曜日を 6 として、曜日を整数で返します。例えば、`date(2002, 12, 4).weekday() == 2` であり、水曜日を示します。*isoweekday()* も参照してください。

date.**isoweekday**()

月曜日を 1, 日曜日を 7 として、曜日を整数で返します。例えば、`date(2002, 12, 4).isoweekday() == 3` であり、水曜日を示します。*weekday()*, *isocalendar()* も参照してください。

date.**isocalendar**()

3 要素のタプル (ISO 年、ISO 週番号、ISO 曜日) を返します。

ISO 暦はグレゴリオ暦の変種として広く用いられています。[*3]

ISO 年は完全な週が 52 週または 53 週あり、週は月曜から始まって日曜に終わります。ISO 年でのある年における最初の週は、その年の木曜日を含む最初の (グレゴリオ暦での) 週となります。この週は週番号 1 と呼ばれ、この木曜日での ISO 年はグレゴリオ暦における年と等しくなります。

例えば、2004 年は木曜日から始まるため、ISO 年の最初の週は 2003 年 12 月 29 日、月曜日から始まり、2004 年 1 月 4 日、日曜日に終わります

[*3] 優れた説明は R. H. van Gent の guide to the mathematics of the ISO 8601 calendar を参照してください。

```
>>> from datetime import date
>>> date(2003, 12, 29).isocalendar()
(2004, 1, 1)
>>> date(2004, 1, 4).isocalendar()
(2004, 1, 7)
```

date.isoformat()

日付を ISO 8601 書式の YYYY-MM-DD で表した文字列を返します:

```
>>> from datetime import date
>>> date(2002, 12, 4).isoformat()
'2002-12-04'
```

この関数は *date.fromisoformat()* の逆関数です。

date.__str__()

date オブジェクト *d* において、str(d) は d.isoformat() と等価です。

date.ctime()

日付を表す文字列を返します:

```
>>> from datetime import date
>>> date(2002, 12, 4).ctime()
'Wed Dec  4 00:00:00 2002'
```

d.ctime() は次の式と等価です:

```
time.ctime(time.mktime(d.timetuple()))
```

これが等価になるのは、(*time.ctime()* に呼び出され、*date.ctime()* に呼び出されない) ネイティブの C 関数 ctime() が C 標準に準拠しているプラットフォーム上でです。

date.strftime(*format*)

明示的な書式文字列で制御された、日付を表現する文字列を返します。時間、分、秒を表す書式コードは値 0 になります。完全な書式化指定子のリストについては *strftime() と strptime() の振る舞い* を参照してください。

date.__format__(*format*)

date.strftime() と等価です。これにより、フォーマット済み文字列リテラル の中や *str.format()* を使っているときに *date* オブジェクトの書式文字列を指定できます。書式化コードの完全なリストについては *strftime() と strptime() の振る舞い* を参照してください。

使用例: date

イベントまでの日数を数える例を示します:

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year + 1)
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202
```

さらなる *date* を使う例:

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 0001
>>> d
datetime.date(2002, 3, 11)

>>> # Methods related to formatting string output
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> d.ctime()
'Mon Mar 11 00:00:00 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'

>>> # Methods for to extracting 'components' under different calendars
>>> t = d.timetuple()
>>> for i in t:
...     print(i)
2002                # year
3                   # month
11                  # day
0
0
0
0                   # weekday (0 = Monday)
70                  # 70th day in the year
-1
```

(次のページに続く)

(前のページからの続き)

```
>>> ic = d.isocalendar()
>>> for i in ic:
...     print(i)
2002                # ISO year
11                  # ISO week number
1                   # ISO day number ( 1 = Monday )

>>> # A date object is immutable; all operations produce a new object
>>> d.replace(year=2005)
datetime.date(2005, 3, 11)
```

8.1.6 datetime オブジェクト

datetime オブジェクトは *date* オブジェクトおよび *time* オブジェクトの全ての情報が入っている単一のオブジェクトです。

date オブジェクトと同様に、*datetime* は現在のグレゴリオ暦が両方向に延長されているものと仮定します。また、*time* オブジェクトと同様に、*datetime* は毎日が厳密に 3600*24 秒であると仮定します。

以下にコンストラクタを示します:

class datetime.**datetime**(*year*, *month*, *day*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, ***, *fold=0*)

year, *month*, *day* 引数は必須です。*tzinfo* は None または *tzinfo* サブクラスのインスタンスです。残りの引数は次の範囲の整数でなければなりません:

- MINYEAR <= year <= MAXYEAR,
- 1 <= month <= 12,
- 1 <= day <= **指定された月と年における日数**,
- 0 <= hour < 24,
- 0 <= minute < 60,
- 0 <= second < 60,
- 0 <= microsecond < 1000000,
- fold in [0, 1].

範囲を超えた引数を与えた場合、*ValueError* が送出されます。

バージョン 3.6 で追加: fold 引数が追加されました。

他のコンストラクタ、および全てのクラスメソッドを以下に示します:

classmethod datetime.**today**()

tzinfo を None にして、現在のローカルな日時を返します。

次と等価です:

```
datetime.fromtimestamp(time.time())
```

now(), *fromtimestamp()* も参照してください。

このメソッドの機能は *now()* と等価ですが、tz 引数はありません。

classmethod **datetime.now**(*tz=None*)

現在のローカルな日時を返します。

オプションの引数 *tz* が None であるか指定されていない場合、このメソッドは *today()* と同様ですが、可能ならば *time.time()* タイムスタンプを通じて得ることができる、より高い精度で時刻を提供します (例えば、プラットフォームが C 関数 gettimeofday() をサポートする場合には可能なことがあります)。

tz が None でない場合、*tz* は *tzinfo* のサブクラスのインスタンスでなければならず、現在の日付および時刻は *tz* のタイムゾーンに変換されます。

today() および *utcnow()* よりもこの関数を使う方が好ましいです。

classmethod **datetime.utcnow**()

tzinfo が None である現在の UTC の日付および時刻を返します。

このメソッドは *now()* と似ていますが、naive な *datetime* オブジェクトとして現在の UTC 日付および時刻を返します。aware な現在の UTC datetime は datetime.now(timezone.utc) を呼び出すことで取得できます。*now()* も参照してください。

> **警告:** naive な datetime オブジェクトは多くの datetime メソッドでローカルな時間として扱われるため、aware な datetime を使って UTC の時刻を表すのが好ましいです。そのため、UTC での現在の時刻を表すオブジェクトの作成では datetime.now(timezone.utc) を呼び出す方法が推奨されます。

classmethod **datetime.fromtimestamp**(*timestamp*, *tz=None*)

time.time() が返すような、POSIX タイムスタンプに対応するローカルな日付と時刻を返します。オプションの引数 *tz* が None であるか、指定されていない場合、タイムスタンプはプラットフォームのローカルな日付および時刻に変換され、返される *datetime* オブジェクトは naive なものになります。

tz が None でない場合、*tz* は *tzinfo* のサブクラスのインスタンスでなければならず、タイムスタンプは *tz* のタイムゾーンに変換されます。

タイムスタンプがプラットフォームの C 関数 localtime() や gmtime() でサポートされている範囲を超えた場合、*fromtimestamp()* は *OverflowError* を送出することがあります。この範囲はよく 1970 年から 2038 年に制限されています。また localtime() や gmtime() が失敗した際は *OSError* を送出します。うるう秒がタイムスタンプの概念に含まれている非 POSIX システムでは、*fromtimestamp()* はうるう秒を無視します。このため、秒の異なる二つのタイムスタンプが同一の

datetime オブジェクトとなることが起こり得ます。*utcfromtimestamp()* よりも、このメソッドの方が好ましいです。

バージョン 3.3 で変更: timestamp がプラットフォームの C 関数 `localtime()` もしくは `gmtime()` のサポートする値の範囲から外れていた場合、*ValueError* ではなく *OverflowError* を送出するようになりました。`localtime()` もしくは `gmtime()` の呼び出し失敗で *ValueError* ではなく *OSError* を送出するようになりました。

バージョン 3.6 で変更: *fromtimestamp()* は *fold* を 1 にしてインスタンスを返します。

classmethod datetime.**utcfromtimestamp**(*timestamp*)

POSIX タイムスタンプに対応する、*tzinfo* が None の UTC での *datetime* を返します。(返されるオブジェクトは naive です。)

タイムスタンプがプラットフォームにおける C 関数 `localtime()` でサポートされている範囲を超えている場合には *OverflowError* を、`gmtime()` が失敗した場合には *OSError* を送出します。これはたいてい 1970 年から 2038 年に制限されています。

aware な *datetime* オブジェクトを得るには *fromtimestamp()* を呼んでください:

```
datetime.fromtimestamp(timestamp, timezone.utc)
```

POSIX 互換プラットフォームでは、これは以下の表現と等価です:

```
datetime(1970, 1, 1, tzinfo=timezone.utc) + timedelta(seconds=timestamp)
```

後者を除き、式は常に年の全範囲 (*MINYEAR* から *MAXYEAR* を含みます) をサポートします。

> **警告:** naive な `datetime` オブジェクトは多くの `datetime` メソッドでローカルな時間として扱われるため、aware な datetime を使って UTC の時刻を表すのが好ましいです。そのため、UTC でのある特定のタイムスタンプを表すオブジェクトの作成では `datetime.fromtimestamp(timestamp, tz=timezone.utc)` を呼び出す方法が推奨されます。

バージョン 3.3 で変更: timestamp がプラットフォームの C 関数 `gmtime()` のサポートする値の範囲から外れていた場合、*ValueError* ではなく *OverflowError* を送出するようになりました。`gmtime()` の呼び出し失敗で *ValueError* ではなく *OSError* を送出するようになりました。

classmethod datetime.**fromordinal**(*ordinal*)

1 年 1 月 1 日を序数 1 とする早期グレゴリオ暦序数に対応する *datetime* オブジェクトを返します。`1 <= ordinal <= datetime.max.toordinal()` でなければ *ValueError* が送出されます。返されるオブジェクトの時間、分、秒、およびマイクロ秒はすべて 0 で、*tzinfo* は None となっています。

classmethod datetime.**combine**(*date*, *time*, *tzinfo=self.tzinfo*)

日付部分と与えられた *date* オブジェクトとが等しく、時刻部分と与えられた *time* オブジェクトとが等しい、新しい *datetime* オブジェクトを返します。*tzinfo* 引数が与えられた場合、その値は返り値の *tzinfo* 属性に設定するのに使われます。そうでない場合、*time* 引数の *tzinfo* 属性が使われます。

任意の *datetime* オブジェクト *d* で `d == datetime.combine(d.date(), d.time(), d.tzinfo)` が成立します。date が *datetime* オブジェクトだった場合、その *datetime* オブジェクトの時刻部分と *tzinfo* 属性は無視されます。

バージョン 3.6 で変更: *tzinfo* 引数が追加されました。

classmethod datetime.**fromisoformat**(*date_string*)

date.isoformat() および *datetime.isoformat()* の出力書式で、*date_string* に対応する *datetime* を返します。

具体的には、この関数は次の書式の文字列をサポートしています:

```
YYYY-MM-DD[*HH[:MM[:SS[.fff[fff]]]][+HH:MM[:SS[.ffffff]]]]
```

ここで * は任意の一文字にマッチします。

ご用心: このメソッドは任意の ISO 8601 文字列の構文解析をサポートしてはいません - このメソッドは *datetime.isoformat()* の逆操作をするためだけのものです。より高機能な ISO 8601 構文解析器である `dateutil.parser.isoparse` が、サードパーティーパッケージの dateutil から利用可能です。

例:

```
>>> from datetime import datetime
>>> datetime.fromisoformat('2011-11-04')
datetime.datetime(2011, 11, 4, 0, 0)
>>> datetime.fromisoformat('2011-11-04T00:05:23')
datetime.datetime(2011, 11, 4, 0, 5, 23)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283+00:00')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000, tzinfo=datetime.timezone.utc)
>>> datetime.fromisoformat('2011-11-04T00:05:23+04:00')
datetime.datetime(2011, 11, 4, 0, 5, 23,
    tzinfo=datetime.timezone(datetime.timedelta(seconds=14400)))
```

バージョン 3.7 で追加.

classmethod datetime.**fromisocalendar**(*year*, *week*, *day*)

年月日で指定された ISO 暦の日付に対応する *datetime* を返します。datetime の日付でない部分は、標準のデフォルト値で埋められます。この関数は *datetime.isocalendar()* の逆関数です。

バージョン 3.8 で追加.

classmethod datetime.**strptime**(*date_string*, *format*)

date_string に対応した *datetime* を返します。*format* にしたがって構文解析されます。

これは次と等価です:

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

date_string と format が *time.strptime()* で構文解析できない場合や、この関数が時刻タプルを返してこない場合には *ValueError* を送出します。完全な書式化指定子のリストについては *strftime()* と *strptime()* **の振る舞い** を参照してください。

以下にクラス属性を示します:

datetime.min
表現できる最も古い *datetime* で、datetime(MINYEAR, 1, 1, tzinfo=None) です。

datetime.max
表現できる最も新しい *datetime* で、datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None) です。

datetime.resolution
等しくない *datetime* オブジェクト間の最小の差で、timedelta(microseconds=1) です。

インスタンスの属性 (読み出しのみ):

datetime.year
両端値を含む *MINYEAR* から *MAXYEAR* までの値です。

datetime.month
両端値を含む 1 から 12 までの値です。

datetime.day
1 から与えられた月と年における日数までの値です。

datetime.hour
in range(24) を満たします。。

datetime.minute
in range(60) を満たします。

datetime.second
in range(60) を満たします。

datetime.microsecond
in range(1000000) を満たします。

datetime.tzinfo
datetime コンストラクタに *tzinfo* 引数として与えられたオブジェクトになり、何も渡されなかった場合には None になります。

datetime.fold
[0, 1] のどちらかです。繰り返し期間中の実時間の曖昧さ除去に使われます。(繰り返し期間は、夏時間の終わりに時計が巻き戻るときや、現在のゾーンの UTC オフセットが政治的な理由で減少するときに発生します。) 0 (1) という値は、同じ実時間で表現される 2 つの時刻のうちの早い方 (遅い方) を表します。

バージョン 3.6 で追加.

サポートされている演算を以下に示します:

演算	結果
`datetime2 = datetime1 + timedelta`	(1)
`datetime2 = datetime1 - timedelta`	(2)
`timedelta = datetime1 - datetime2`	(3)
`datetime1 < datetime2`	*datetime* を *datetime* と比較します。(4)

(1) datetime2 は datetime1 から時間 timedelta 移動したもので、`timedelta.days` > 0 の場合未来へ、`timedelta.days` < 0 の場合過去へ移動します。結果は入力の datetime と同じ *tzinfo* 属性を持ち、演算後には datetime2 - datetime1 == timedelta となります。datetime2.year が *MINYEAR* よりも小さいか、*MAXYEAR* より大きい場合には *OverflowError* が送出されます。入力が aware なオブジェクトの場合でもタイムゾーン修正は全く行われません。

(2) datetime2 + timedelta == datetime1 となるような datetime2 を計算します。ちなみに、結果は入力の datetime と同じ *tzinfo* 属性を持ち、入力が aware だとしてもタイムゾーン修正は全く行われません。この操作は date1 + (-timedelta) と等価ではありません。なぜならば、date1 - timedelta がオーバフローしない場合でも、-timedelta 単体がオーバフローする可能性があるからです。

(3) *datetime* から *datetime* の減算は両方の被演算子が naive であるか、両方とも aware である場合にのみ定義されています。片方が aware でもう一方が naive の場合、*TypeError* が送出されます。

両方とも naive か、両方とも aware で同じ *tzinfo* 属性を持つ場合、*tzinfo* 属性は無視され、結果は `datetime2 + t == datetime1` であるような *timedelta* オブジェクト *t* となります。この場合タイムゾーン修正は全く行われません。

両方が aware で異なる *tzinfo* 属性を持つ場合、`a-b` は *a* および *b* をまず naive な UTC datetime オブジェクトに変換したかのようにして行います。演算結果は決してオーバフローを起こさないことを除き、`(a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())` と同じになります。

(4) *datetime1* が時刻として *datetime2* よりも前を表す場合に、*datetime1* は *datetime2* よりも小さいと見なされます。

比較の一方が naive であり、もう一方が aware の場合に、順序比較が行われると *TypeError* が送出されます。等価比較では、naive インスタンスと aware インスタンスは等価になることはありません。

比較対象が両方とも aware で、同じ *tzinfo* 属性を持つ場合、*tzinfo* は無視され datetime だけで比較が行われます。比較対象が両方とも aware であり、異なる *tzinfo* 属性を持つ場合、まず最初に (`self.utcoffset()` で取得できる) それぞれの UTC オフセットを引いて調整します。

バージョン 3.3 で変更: aware な *datetime* インスタンスと naive な *datetime* インスタンスの等価比較では *TypeError* は送出されません。

注釈: 型混合の比較がデフォルトのオブジェクトアドレス比較となってしまうのを抑止するため

に、被演算子のもう一方が *datetime* オブジェクトと異なる型のオブジェクトの場合には *TypeError* が送出されます。しかしながら、被比較演算子のもう一方が `timetuple()` 属性を持つ場合には `NotImplemented` が返されます。このフックにより、他種の日付オブジェクトに型混合比較を実装するチャンスを与えています。そうでない場合, *datetime* オブジェクトと異なる型のオブジェクトが比較されると、比較演算子が `==` または `!=` でないかぎり *TypeError* が送出されます。後者の場合、それぞれ *False* または *True* を返します。

インスタンスメソッド:

datetime.date()
同じ年、月、日の *date* オブジェクトを返します。

datetime.time()
同じ hour、minute、second、microsecond 及び fold を持つ *time* オブジェクトを返します。*tzinfo* は `None` です。*timetz()* も参照してください。

バージョン 3.6 で変更: 値 fold は返される *time* オブジェクトにコピーされます。

datetime.timetz()
同じ hour、minute、second、microsecond、fold および tzinfo 属性を持つ *time* オブジェクトを返します。*time()* メソッドも参照してください。

バージョン 3.6 で変更: 値 fold は返される *time* オブジェクトにコピーされます。

datetime.replace(*year=self.year*, *month=self.month*, *day=self.day*, *hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, ***, *fold=0*)
キーワード引数で指定した属性の値を除き、同じ属性をもつ datetime オブジェクトを返します。メンバに対する変換を行わずに aware な datetime オブジェクトから naive な datetime オブジェクトを生成するために、`tzinfo=None` を指定することもできます。

バージョン 3.6 で追加: `fold` 引数が追加されました。

datetime.astimezone(*tz=None*)
tz を新たに *tzinfo* 属性 として持つ *datetime* オブジェクトを返します。日付および時刻データを調整して、返り値が *self* と同じ UTC 時刻を持ち、*tz* におけるローカルな時刻を表すようにします。

もし与えられた場合、*tz* は *tzinfo* のサブクラスのインスタンスでなければならず、インスタンスの *utcoffset()* および *dst()* メソッドは `None` を返してはなりません。もし *self* が naive ならば、おそらくシステムのタイムゾーンで時間を表現します。

引数無し (もしくは `tz=None` の形) で呼び出された場合、システムのローカルなタイムゾーンが変更先のタイムゾーンだと仮定されます。変換後の datetime インスタンスの `.tzinfo` 属性には、OS から取得したゾーン名とオフセットを持つ *timezone* インスタンスが設定されます。

`self.tzinfo` が *tz* の場合、`self.astimezone(tz)` は *self* に等しくなります。つまり、date および time に対する調整は行われません。そうでない場合、結果はタイムゾーン *tz* におけるローカル時刻で、*self* と同じ UTC 時刻を表すようになります。これは、`astz = dt.astimezone(tz)` とした後、

`astz - astz.utcoffset()` は通常 `dt - dt.utcoffset()` と同じ date および time を持つことを示します。

単にタイムゾーンオブジェクト *tz* を *datetime* オブジェクト *dt* に追加したいだけで、日付や時刻データへの調整を行わないのなら、`dt.replace(tzinfo=tz)` を使ってください。単に aware な *datetime* オブジェクト *dt* からタイムゾーンオブジェクトを除去したいだけで、日付や時刻データの変換を行わないのなら、`dt.replace(tzinfo=None)` を使ってください。

デフォルトの *tzinfo.fromutc()* メソッドを *tzinfo* のサブクラスで上書きして, *astimezone()* が返す結果に影響を及ぼすことができます。エラーの場合を無視すると、*astimezone()* は以下のように動作します:

```
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new time zone object.
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```

バージョン 3.3 で変更: *tz* が省略可能になりました。

バージョン 3.6 で変更: *datetime.datetime.astimezone()* メソッドを naive なインスタンスに対して呼び出せるようになりました。これは、システムのローカルな時間を表現していると想定されます。

datetime.utcoffset()

tzinfo が None の場合、None を返し、そうでない場合には `self.tzinfo.utcoffset(self)` を返します。後者の式が None あるいは 1 日以下の大きさを持つ *timedelta* オブジェクトのいずれかを返さない場合には例外を送出します。

バージョン 3.7 で変更: UTC オフセットが分単位でなければならない制限が無くなりました。

datetime.dst()

tzinfo が None の場合 None を返し、そうでない場合には `self.tzinfo.dst(self)` を返します。後者の式が None もしくは、1 日未満の大きさを持つ *timedelta* オブジェクトのいずれかを返さない場合には例外を送出します。

バージョン 3.7 で変更: DST オフセットが分単位でなければならない制限が無くなりました。

datetime.tzname()

tzinfo が None の場合 None を返し、そうでない場合には `self.tzinfo.tzname(self)` を返します。後者の式が None か文字列オブジェクトのいずれかを返さない場合には例外を送出します。

datetime.timetuple()

time.localtime() が返すような *time.struct_time* を返します。

`d.timetuple()` は次の式と等価です:

```
time.struct_time((d.year, d.month, d.day,
                  d.hour, d.minute, d.second,
                  d.weekday(), yday, dst))
```

ここで `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` はその年の 1 月 1 日を 1 としたときのその日の位置です。返されるタプルの `tm_isdst` フラグは *dst()* メソッドに従って設定されます: *tzinfo* が `None` か *dst()* が `None` を返す場合、`tm_isdst` は `-1` に設定されます; そうでない場合、*dst()* がゼロでない値を返すと `tm_isdst` は `1` となります; それ以外の場合には `tm_isdst` は `0` に設定されます。

datetime.utctimetuple()

datetime インスタンス *d* が naive の場合、このメソッドは `d.timetuple()` と同じであり、`d.dst()` の返す内容にかかわらず `tm_isdst` が 0 に強制される点だけが異なります。DST が UTC 時刻に影響を及ぼすことは決してありません。

d が aware だった場合、*d* は `d.utcoffset()` を引いて UTC 時刻に正規化され、その時刻が *time.struct_time* として返されます。`tm_isdst` は 0 に強制されます。*d*.year が `MINYEAR` もしくは `MAXYEAR` であり、UTC 時刻への調整により適切な年の範囲を越えた場合、*OverflowError* が送出される可能性があることに注意してください。

> **警告:** naive な `datetime` オブジェクトは多くの `datetime` メソッドでローカルな時間として扱われるため、aware な datetime を使って UTC の時刻を表すのが好ましいです。結果として、`utcfromtimetuple` は誤解を招きやすい返り値を返すかもしれません。UTC を表す naive な `datetime` があった場合、*datetime.timetuple()* が使えるところでは `datetime.replace(tzinfo=timezone.utc)` で aware にします。

datetime.toordinal()

先発グレゴリオ暦における日付序数を返します。`self.date().toordinal()` と同じです。

datetime.timestamp()

datetime インスタンスに対応する POSIX タイムスタンプを返します。返り値は *time.time()* で返される値に近い *float* です。

このメソッドでは naive な *datetime* インスタンスはローカル時刻とし、プラットフォームの C 関数 `mktime()` に頼って変換を行います。*datetime* は多くのプラットフォームの `mktime()` より広い範囲の値をサポートしているので、遥か過去の時刻や遥か未来の時刻に対し、このメソッドは *OverflowError* を送出するかもしれません。

aware な *datetime* インスタンスに対しては以下のように返り値が計算されます:

```
(dt - datetime(1970, 1, 1, tzinfo=timezone.utc)).total_seconds()
```

バージョン 3.3 で追加.

バージョン 3.6 で変更: The *timestamp()* method uses the *fold* attribute to disambiguate the times during a repeated interval.

注釈: UTC 時刻を表す naive な *datetime* インスタンスから直接 POSIX タイムスタンプを取得するメソッドはありません。アプリケーションがその変換を使っており、システムのタイムゾーンが UTC に設定されていなかった場合、`tzinfo=timezone.utc` を引数に与えることで POSIX タイムスタンプを取得できます:

```
timestamp = dt.replace(tzinfo=timezone.utc).timestamp()
```

もしくは直接タイムスタンプを計算することもできます:

```
timestamp = (dt - datetime(1970, 1, 1)) / timedelta(seconds=1)
```

datetime.weekday()

月曜日を 0、日曜日を 6 として、曜日を整数で返します。`self.date().weekday()` と同じです。*isoweekday()* も参照してください。

datetime.isoweekday()

月曜日を 1、日曜日を 7 として、曜日を整数で返します。`self.date().isoweekday()` と等価です。*weekday()* 、*isocalendar()* も参照してください。

datetime.isocalendar()

3 要素のタプル (ISO 年、ISO 週番号、ISO 曜日) を返します。`self.date().isocalendar()` と等価です。

datetime.isoformat(*sep='T', timespec='auto'*)

日時を ISO 8601 書式で表した文字列で返します:

- *microsecond* が 0 でない場合は `YYYY-MM-DDTHH:MM:SS.ffffff`
- *microsecond* が 0 の場合は `YYYY-MM-DDTHH:MM:SS`

utcoffset() が `None` を返さない場合は、文字列の後ろに UTC オフセットが追記されます:

- *microsecond* が 0 でない場合は `YYYY-MM-DDTHH:MM:SS.ffffff+HH:MM[:SS[.ffffff]]`
- *microsecond* が 0 の場合は `YYYY-MM-DDTHH:MM:SS+HH:MM[:SS[.ffffff]]`

例:

```
>>> from datetime import datetime, timezone
>>> datetime(2019, 5, 18, 15, 17, 8, 132263).isoformat()
'2019-05-18T15:17:08.132263'
>>> datetime(2019, 5, 18, 15, 17, tzinfo=timezone.utc).isoformat()
'2019-05-18T15:17:00+00:00'
```

オプションの引数 *sep* (デフォルトでは `'T'` です) は 1 文字のセパレータで、結果の文字列の日付と時刻の間に置かれます。例えば:

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     """A time zone with an arbitrary, constant -06:39 offset."""
...     def utcoffset(self, dt):
...         return timedelta(hours=-6, minutes=-39)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
>>> datetime(2009, 11, 27, microsecond=100, tzinfo=TZ()).isoformat()
'2009-11-27T00:00:00.000100-06:39'
```

オプション引数 *timespec* は、含める追加の時間の要素の数を指定します (デフォルトでは 'auto' です)。以下の内一つを指定してください。

- 'auto': *microsecond* が 0 である場合 'seconds' と等しく、そうでない場合は 'microseconds' と等しくなります。
- 'hours': *hour* を 2 桁の HH 書式で含めます。
- 'minutes': *hour* および *minute* を HH:MM の書式で含めます。
- 'seconds': *hour* 、*minute* 、*second* を HH:MM:SS の書式で含めます。
- 'milliseconds': 全ての時刻を含みますが、小数第二位をミリ秒に切り捨てます。HH:MM:SS.sss の書式で表現します。
- 'microseconds': 全ての時刻を HH:MM:SS.mmmmmm の書式で含めます。

注釈: 除外された要素は丸め込みではなく、切り捨てされます。

不正な *timespec* 引数には *ValueError* があげられます:

```
>>> from datetime import datetime
>>> datetime.now().isoformat(timespec='minutes')
'2002-12-25T00:00'
>>> dt = datetime(2015, 1, 1, 12, 30, 59, 0)
>>> dt.isoformat(timespec='microseconds')
'2015-01-01T12:30:59.000000'
```

バージョン 3.6 で追加: *timespec* 引数が追加されました。

datetime.__str__()

datetime オブジェクト *d* において、str(d) は d.isoformat(' ') と等価です。

datetime.ctime()

日付および時刻を表す文字列を返します:

```
>>> from datetime import datetime
>>> datetime(2002, 12, 4, 20, 30, 40).ctime()
'Wed Dec  4 20:30:40 2002'
```

出力文字列は入力が aware であれ naive であれ、タイムゾーン情報を含み **ません**。

`d.ctime()` は次の式と等価です:

```
time.ctime(time.mktime(d.timetuple()))
```

これが等価になるのは、(*time.ctime()* に呼び出され、*datetime.ctime()* に呼び出されない) ネイティブの C 関数 `ctime()` が C 標準に準拠しているプラットフォーム上でです。

datetime.strftime(*format*)

明示的な書式文字列で制御された、日付および時刻を表現する文字列を返します。完全な書式化指定子のリストについては *strftime()* と *strptime()* **の振る舞い** を参照してください。

datetime.__format__(*format*)

datetime.strftime() と等価です。これにより、フォーマット済み文字列リテラル の中や *str.format()* を使っているときに *datetime* オブジェクトの書式文字列を指定できます。書式化指定子の完全なリストについては *strftime()* と *strptime()* **の振る舞い** を参照してください。

使用例: datetime

datetime オブジェクトを使う例:

```
>>> from datetime import datetime, date, time, timezone

>>> # Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)

>>> # Using datetime.now()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)   # GMT +1
>>> datetime.now(timezone.utc)
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060, tzinfo=datetime.timezone.utc)

>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)

>>> # Using datetime.timetuple() to get tuple of all attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print(it)
...
2006    # year
11      # month
21      # day
16      # hour
```

(次のページに続く)

(前のページからの続き)

```
30      # minute
0       # second
1       # weekday (0 = Monday)
325     # number of days since 1st January
-1      # dst - method tzinfo.dst() returned None

>>> # Date in ISO format
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print(it)
...
2006    # ISO year
47      # ISO week
2       # ISO weekday

>>> # Formatting a datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}.'.format(dt, "day", "month",
↪"time")
'The day is 21, the month is November, the time is 04:30PM.'
```

下にある例では、1945 年までは +4 UTC 、それ以降は +4:30 UTC を使用しているアフガニスタンのカブールのタイムゾーン情報を表現する *tzinfo* のサブクラスを定義しています:

```
from datetime import timedelta, datetime, tzinfo, timezone

class KabulTz(tzinfo):
    # Kabul used +4 until 1945, when they moved to +4:30
    UTC_MOVE_DATE = datetime(1944, 12, 31, 20, tzinfo=timezone.utc)

    def utcoffset(self, dt):
        if dt.year < 1945:
            return timedelta(hours=4)
        elif (1945, 1, 1, 0, 0) <= dt.timetuple()[:5] < (1945, 1, 1, 0, 30):
            # An ambiguous ("imaginary") half-hour range representing
            # a 'fold' in time due to the shift from +4 to +4:30.
            # If dt falls in the imaginary range, use fold to decide how
            # to resolve. See PEP495.
            return timedelta(hours=4, minutes=(30 if dt.fold else 0))
        else:
            return timedelta(hours=4, minutes=30)

    def fromutc(self, dt):
        # Follow same validations as in datetime.tzinfo
        if not isinstance(dt, datetime):
            raise TypeError("fromutc() requires a datetime argument")
        if dt.tzinfo is not self:
            raise ValueError("dt.tzinfo is not self")

        # A custom implementation is required for fromutc as
```

(次のページに続く)

(前のページからの続き)

```
        # the input to this function is a datetime with utc values
        # but with a tzinfo set to self.
        # See datetime.astimezone or fromtimestamp.
        if dt.replace(tzinfo=timezone.utc) >= self.UTC_MOVE_DATE:
            return dt + timedelta(hours=4, minutes=30)
        else:
            return dt + timedelta(hours=4)

    def dst(self, dt):
        # Kabul does not observe daylight saving time.
        return timedelta(0)

    def tzname(self, dt):
        if dt >= self.UTC_MOVE_DATE:
            return "+04:30"
        return "+04"
```

上に出てきた KabulTz の使い方:

```
>>> tz1 = KabulTz()

>>> # Datetime before the change
>>> dt1 = datetime(1900, 11, 21, 16, 30, tzinfo=tz1)
>>> print(dt1.utcoffset())
4:00:00

>>> # Datetime after the change
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=tz1)
>>> print(dt2.utcoffset())
4:30:00

>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(timezone.utc)
>>> dt3
datetime.datetime(2006, 6, 14, 8, 30, tzinfo=datetime.timezone.utc)
>>> dt2
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=KabulTz())
>>> dt2 == dt3
True
```

8.1.7 time オブジェクト

time オブジェクトは (ローカルの) 日中時刻を表現します。この時刻表現は特定の日の影響を受けず、*tzinfo* オブジェクトを介した修正の対象となります。

class datetime.**time**(*hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, ***, *fold=0*)

全ての引数はオプションです。*tzinfo* は None または *tzinfo* クラスのサブクラスのインスタンスにすることができます。残りの引数は整数で、以下のような範囲に入らなければなりません:

- `0 <= hour < 24`,
- `0 <= minute < 60`,
- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,
- `fold in [0, 1]`.

引数がこれらの範囲外にある場合、*ValueError* が送出されます。*tzinfo* のデフォルト値が *None* である以外のデフォルト値は 0 です。

以下にクラス属性を示します:

time.min
表現できる最も古い *time* で、`time(0, 0, 0, 0)` です。

time.max
表現できる最も新しい *time* で、`time(23, 59, 59, 999999)` です。

time.resolution
等しくない *time* オブジェクト間の最小の差で、`timedelta(microseconds=1)` ですが, *time* オブジェクト間の四則演算はサポートされていないので注意してください。

インスタンスの属性 (読み出しのみ):

time.hour
in `range(24)` を満たします。。

time.minute
in `range(60)` を満たします。

time.second
in `range(60)` を満たします。

time.microsecond
in `range(1000000)` を満たします。

time.tzinfo
time コンストラクタに *tzinfo* 引数として与えられたオブジェクトになり、何も渡されなかった場合には `None` になります。

time.fold
`[0, 1]` のどちらかです。繰り返し期間中の実時間の曖昧さ除去に使われます。(繰り返し期間は、夏時間の終わりに時計が巻き戻るときや、現在のゾーンの UTC オフセットが政治的な理由で減少するときに発生します。) 0 (1) という値は、同じ実時間で表現される 2 つの時刻のうちの早い方 (遅い方) を表します。

バージョン 3.6 で追加.

time オブジェクトは *time* どうしの比較をサポートしていて、*a* が *b* より前の時刻だった場合 *a* が *b* より小さいとされます。比較対象の片方が naive であり、もう片方が aware の場合に、順序比較が行われると *TypeError* が送出されます。等価比較では、naive インスタンスと aware インスタンスは等価になることはありません。

比較対象が両方とも aware であり、同じ *tzinfo* 属性を持つ場合、*tzinfo* は無視され datetime だけで比較が行われます。比較対象が両方とも aware であり、異なる *tzinfo* 属性を持つ場合、まず最初に (`self.utcoffset()` で取得できる) それぞれの UTC オフセットを引いて調整します。異なる型どうしの比較がデフォルトのオブジェクトアドレス比較となってしまうのを防ぐために、*time* オブジェクトを異なる型のオブジェクトと比較すると、比較演算子が `==` または `!=` でないかぎり *TypeError* が送出されます。比較演算子が `==` または `!=` である場合、それぞれ *False* または *True* を返します。

バージョン 3.3 で変更: aware な *datetime* インスタンスと naive な *time* インスタンスの等価比較では *TypeError* は送出されません。

ブール値の文脈では、*time* オブジェクトは常に真とみなされます。

バージョン 3.5 で変更: Python 3.5 以前は、*time* オブジェクトは UTC で深夜を表すときに偽とみなされていました。この挙動は分かりにくく、エラーの元となると考えられ、Python 3.5 で削除されました。全詳細については bpo-13936 を参照してください。

その他のコンストラクタ:

classmethod **time.fromisoformat**(*time_string*)

time.isoformat() の出力書式のうちの 1 つの書式で、*time_string* に対応する *time* を返します。具体的には、この関数は次の書式の文字列をサポートしています:

```
HH[:MM[:SS[.fff[fff]]]][+HH:MM[:SS[.ffffff]]]
```

> **ご用心:** この関数は、任意の ISO 8601 文字列の構文解析をサポートしているわけでは **ありません** これは *time.isoformat()* の逆演算を意図して実装されています。

例:

```
>>> from datetime import time
>>> time.fromisoformat('04:23:01')
datetime.time(4, 23, 1)
>>> time.fromisoformat('04:23:01.000384')
datetime.time(4, 23, 1, 384)
>>> time.fromisoformat('04:23:01+04:00')
datetime.time(4, 23, 1, tzinfo=datetime.timezone(datetime.timedelta(seconds=14400)))
```

バージョン 3.7 で追加.

インスタンスメソッド:

time.replace(*hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, ***, *fold=0*)

キーワード引数で指定したメンバの値を除き、同じ値をもつ *time* オブジェクトを返します。データに対する変換を行わずに aware な *time* オブジェクトから naive な *time* オブジェクトを生成するために、tzinfo=None を指定することもできます。

バージョン 3.6 で追加: fold 引数が追加されました。

time.isoformat(*timespec='auto'*)

時刻を ISO 8601 書式で表した次の文字列のうち 1 つを返します:

- *microsecond* が 0 でない場合は HH:MM:SS.ffffff
- *microsecond* が 0 の場合は HH:MM:SS
- *utcoffset()* が None を返さない場合、HH:MM:SS.ffffff+HH:MM[:SS[.ffffff]]
- *microsecond* が 0 で *utcoffset()* が None を返さない場合、HH:MM:SS+HH:MM[:SS[.ffffff]]

オプション引数 *timespec* は、含める追加の時間の要素の数を指定します (デフォルトでは 'auto' です)。以下の内一つを指定してください。

- 'auto': *microsecond* が 0 である場合 'seconds' と等しく、そうでない場合は 'microseconds' と等しくなります。
- 'hours': *hour* を 2 桁の HH 書式で含めます。
- 'minutes': *hour* および *minute* を HH:MM の書式で含めます。
- 'seconds': *hour* 、*minute* 、*second* を HH:MM:SS の書式で含めます。
- 'milliseconds': 全ての時刻を含みますが、小数第二位をミリ秒に切り捨てます。HH:MM:SS.sss の書式で表現します。
- 'microseconds': 全ての時刻を HH:MM:SS.mmmmmm の書式で含めます。

注釈: 除外された要素は丸め込みではなく、切り捨てされます。

不正な *timespec* 引数には *ValueError* があげられます。

以下はプログラム例です:

```
>>> from datetime import time
>>> time(hour=12, minute=34, second=56, microsecond=123456).isoformat(timespec='minutes')
'12:34'
>>> dt = time(hour=12, minute=34, second=56, microsecond=0)
>>> dt.isoformat(timespec='microseconds')
'12:34:56.000000'
>>> dt.isoformat(timespec='auto')
'12:34:56'
```

バージョン 3.6 で追加: *timespec* 引数が追加されました。

time.__str__()
: *time* オブジェクト *t* において、str(t) は t.isoformat() と等価です。

time.strftime(*format*)
: 明示的な書式文字列で制御された、時刻を表現する文字列を返します。完全な書式化指定子のリストについては *strftime() と strptime() の振る舞い* を参照してください。

time.__format__(*format*)
: *time.strftime()* と等価です。これにより、フォーマット済み文字列リテラル の中や *str.format()* を使っているときに *time* オブジェクトの書式文字列を指定できます。書式化指定子の完全なリストについては *strftime() と strptime() の振る舞い* を参照してください。

time.utcoffset()
: *tzinfo* が None の場合、None を返し、そうでない場合には self.tzinfo.utcoffset(None) を返します。後者の式が None あるいは 1 日以下の大きさを持つ *timedelta* オブジェクトのいずれかを返さない場合には例外を送出します。

  バージョン 3.7 で変更: UTC オフセットが分単位でなければならない制限が無くなりました。

time.dst()
: *tzinfo* が None の場合 None を返し、そうでない場合には self.tzinfo.dst(None) を返します。後者の式が None もしくは、1 日未満の大きさを持つ *timedelta* オブジェクトのいずれかを返さない場合には例外を送出します。

  バージョン 3.7 で変更: DST オフセットが分単位でなければならない制限が無くなりました。

time.tzname()
: *tzinfo* が None の場合 None を返し、そうでない場合には self.tzinfo.tzname(None) を返します。後者の式が None か文字列オブジェクトのいずれかを返さない場合には例外を送出します。

使用例: time

time オブジェクトを使う例:

```
>>> from datetime import time, tzinfo, timedelta
>>> class TZ1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self,dt):
...         return "+01:00"
...     def  __repr__(self):
...         return f"{self.__class__.__name__}()"
...
>>> t = time(12, 10, 30, tzinfo=TZ1())
>>> t
datetime.time(12, 10, 30, tzinfo=TZ1())
>>> t.isoformat()
```

(次のページに続く)

(前のページからの続き)

```
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
>>> t.tzname()
'+01:00'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 +01:00'
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'
```

8.1.8 tzinfo オブジェクト

class datetime.tzinfo

このクラスは抽象基底クラスで、直接インスタンス化すべきでないことを意味します。*tzinfo* のサブクラスを定義し、ある特定のタイムゾーンに関する情報を保持するようにしてください。

tzinfo (の具体的なサブクラス) のインスタンスは *datetime* および *time* オブジェクトのコンストラクタに渡すことができます。後者のオブジェクトでは、データ属性をローカル時刻におけるものとして見ており、*tzinfo* オブジェクトはローカル時刻の UTC からのオフセット、タイムゾーンの名前、DST オフセットを、渡された日付および時刻オブジェクトからの相対で示すためのメソッドを提供します。

具象サブクラスを作成し、(少なくとも) 使いたい *datetime* のメソッドが必要とする *tzinfo* のメソッドを実装する必要があります。*datetime* モジュールは *tzinfo* のシンプルな具象サブクラス *timezone* を提供します。これは UTC そのものか北アメリカの EST と EDT のような UTC からの固定されたオフセットを持つタイムゾーンを表せます。

pickle 化についての特殊な要求事項: *tzinfo* のサブクラスは引数なしで呼び出すことのできる `__init__()` メソッドを持たなければなりません。そうでなければ、pickle 化することはできますがおそらく unpickle 化することはできないでしょう。これは技術的な側面からの要求であり、将来緩和されるかもしれません。

tzinfo の具体的なサブクラスでは、以下のメソッドを実装する必要があります。厳密にどのメソッドが必要なのかは、aware な *datetime* オブジェクトがこのサブクラスのインスタンスをどのように使うかに依存します。不確かならば、単に全てを実装してください。

tzinfo.utcoffset(*dt*)

ローカル時間の UTC からのオフセットを、UTC から東向きを正とした分で返します。ローカル時間が UTC の西側にある場合、この値は負になります。

このメソッドは UTC からのオフセットの **総計** を表しています。例えば、*tzinfo* オブジェクトがタイムゾーンと DST 修正の両方を表現する場合、*utcoffset()* はそれらの合計を返さなければなりません。UTC オフセットが未知である場合、`None` を返します。そうでない場合には、返される値は `-timedelta(hours=24)` から `timedelta(hours=24)` までの *timedelta* 境界を含まないオブジェクトでなければなりません (オフセットの大きさは 1 日より短くなければなりません)。ほとんどの *utcoffset()* 実装は、おそらく以下の二つのうちの一つに似たものになるでしょう:

```
return CONSTANT                 # fixed-offset class
return CONSTANT + self.dst(dt)  # daylight-aware class
```

utcoffset() が None を返さない場合、*dst()* も None を返してはなりません。

utcoffset() のデフォルトの実装は *NotImplementedError* を送出します。

バージョン 3.7 で変更: UTC オフセットが分単位でなければならない制限が無くなりました。

tzinfo.dst(*dt*)

夏時間 (DST) 修正を、UTC から東向きを正とした分で返します。DST 情報が未知の場合、None が返されます。

DST が有効でない場合には timedelta(0) を返します。DST が有効の場合、オフセットは *timedelta* オブジェクトで返します (詳細は *utcoffset()* を参照してください)。DST オフセットが利用可能な場合、この値は *utcoffset()* が返す UTC からのオフセットには既に加算されているため、DST を個別に取得する必要がない限り *dst()* を使って問い合わせる必要はないので注意してください。例えば、*datetime.timetuple()* は *tzinfo* 属性の *dst()* メソッドを呼んで tm_isdst フラグがセットされているかどうか判断し、*tzinfo.fromutc()* は *dst()* タイムゾーンを移動する際に DST による変更があるかどうかを調べます。

標準および夏時間の両方をモデル化している *tzinfo* サブクラスのインスタンス *tz* は以下の式:

```
tz.utcoffset(dt) - tz.dst(dt)
```

が、dt.tzinfo == tz 全ての *datetime* オブジェクト *dt* について常に同じ結果を返さなければならないという点で、一貫性を持っていなければなりません。正常に実装された *tzinfo* のサブクラスでは、この式はタイムゾーンにおける " 標準オフセット (standard offset)" を表し、特定の日や時刻の事情ではなく地理的な位置にのみ依存していなくてはなりません。*datetime.astimezone()* の実装はこの事実に依存していますが、違反を検出することができません; 正しく実装するのはプログラマの責任です。*tzinfo* のサブクラスでこれを保証することができない場合、*tzinfo.fromutc()* の実装をオーバライドして、astimezone() に関わらず正しく動作するようにしてもかまいません。

ほとんどの *dst()* 実装は、おそらく以下の二つのうちの一つに似たものになるでしょう:

```
def dst(self, dt):
    # a fixed-offset class:  doesn't account for DST
    return timedelta(0)
```

もしくは:

```
def dst(self, dt):
    # Code to set dston and dstoff to the time zone's DST
    # transition times based on the input dt.year, and expressed
    # in standard local time.

    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

デフォルトの *dst()* 実装は *NotImplementedError* を送出します。

バージョン 3.7 で変更: DST オフセットが分単位でなければならない制限が無くなりました。

tzinfo.**tzname**(*dt*)

datetime オブジェクト *dt* に対応するタイムゾーン名を文字列で返します。*datetime* モジュールでは文字列名について何も定義しておらず、特に何かを意味するといった要求仕様もまったくありません。例えば、"GMT", "UTC", "-500", "-5:00", "EDT", "US/Eastern", "America/New York" は全て有効な応答となります。文字列名が未知の場合には None を返してください。*tzinfo* のサブクラスでは、特に, *tzinfo* クラスが夏時間について記述している場合のように、渡された *dt* の特定の値によって異なった名前を返したい場合があるため、文字列値ではなくメソッドとなっていることに注意してください。

デフォルトの *tzname()* 実装は *NotImplementedError* を送出します。

以下のメソッドは *datetime* や *time* オブジェクトにおいて、同名のメソッドが呼び出された際に応じて呼び出されます。*datetime* オブジェクトは自身を引数としてメソッドに渡し、*time* オブジェクトは引数として None をメソッドに渡します。従って、*tzinfo* のサブクラスにおけるメソッドは引数 *dt* が None の場合と、*datetime* の場合を受理するように用意しなければなりません。

None が渡された場合、最良の応答方法を決めるのはクラス設計者次第です。例えば、このクラスが *tzinfo* プロトコルと関係をもたないということを表明させたければ、None が適切です。標準時のオフセットを見つける他の手段がない場合には、標準 UTC オフセットを返すために `utcoffset(None)` を使うともっと便利かもしれません。

datetime オブジェクトが `datetime()` メソッドの応答として返された場合、`dt.tzinfo` は *self* と同じオブジェクトになります。ユーザが直接 *tzinfo* メソッドを呼び出さないかぎり、*tzinfo* メソッドは `dt.tzinfo` と *self* が同じであることに依存します。その結果 *tzinfo* メソッドは *dt* がローカル時間であると解釈するので、他のタイムゾーンでのオブジェクトの振る舞いについて心配する必要がありません。

サブクラスでオーバーライドすると良い、もう 1 つの *tzinfo* のメソッドがあります:

tzinfo.**fromutc**(*dt*)

デフォルトの *datetime.astimezone()* 実装で呼び出されます。*datetime.astimezone()* から呼ばれた場合、`dt.tzinfo` は *self* であり、*dt* の日付および時刻データは UTC 時刻を表しているものとして見えます。*fromutc()* の目的は、*self* のローカル時刻に等しい *datetime* オブジェクトを返すことにより日付と時刻データメンバを修正することにあります。

ほとんどの *tzinfo* サブクラスではデフォルトの *fromutc()* 実装を問題なく継承できます。デフォルトの実装は、固定オフセットのタイムゾーンや、標準時と夏時間の両方について記述しているタイムゾーン、そして DST 移行時刻が年によって異なる場合でさえ、扱えるくらい強力なものです。デフォルトの *fromutc()* 実装が全ての場合に対して正しく扱うことができないような例は、標準時の (UTC からの) オフセットが引数として渡された特定の日や時刻に依存するもので、これは政治的な理由によって起きることがあります。デフォルトの `astimezone()` や *fromutc()* の実装は、結果が標準時オフセットの変化にまたがる何時間かの中にある場合、期待通りの結果を生成しないかもしれません。

エラーの場合のためのコードを除き、デフォルトの *fromutc()* の実装は以下のように動作します:

```
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst  # this is self's standard offset
    if delta:
        dt += delta   # convert to standard local time
        dtdst = dt.dst()
        # raise ValueError if dtdst is None
    if dtdst:
        return dt + dtdst
    else:
        return dt
```

次の tzinfo_examples.py ファイルには、*tzinfo* クラスの例がいくつか載っています:

```
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)
SECOND = timedelta(seconds=1)

# A class capturing the platform's idea of local time.
# (May result in wrong values on historical times in
#  timezones where UTC offset and/or the DST rules had
#  changed in the past.)
import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
if _time.daylight:
    DSTOFFSET = timedelta(seconds = -_time.altzone)
else:
    DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):

    def fromutc(self, dt):
        assert dt.tzinfo is self
        stamp = (dt - datetime(1970, 1, 1, tzinfo=self)) // SECOND
        args = _time.localtime(stamp)[:6]
        dst_diff = DSTDIFF // SECOND
        # Detect fold
        fold = (args == _time.localtime(stamp - dst_diff))
        return datetime(*args, microsecond=dt.microsecond,
                        tzinfo=self, fold=fold)

    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
```

(次のページに続く)

(前のページからの続き)

```
        else:
            return STDOFFSET

    def dst(self, dt):
        if self._isdst(dt):
            return DSTDIFF
        else:
            return ZERO

    def tzname(self, dt):
        return _time.tzname[self._isdst(dt)]

    def _isdst(self, dt):
        tt = (dt.year, dt.month, dt.day,
              dt.hour, dt.minute, dt.second,
              dt.weekday(), 0, 0)
        stamp = _time.mktime(tt)
        tt = _time.localtime(stamp)
        return tt.tm_isdst > 0

Local = LocalTimezone()


# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt


# US DST Rules
#
# This is a simplified (i.e., wrong for a few cases) set of rules for US
# DST start and end times. For a complete and up-to-date set of DST rules
# and timezone definitions, visit the Olson Database (or try pytz):
# http://www.twinsun.com/tz/tz-link.htm
# http://sourceforge.net/projects/pytz/ (might not be up-to-date)
#
# In the US, since 2007, DST starts at 2am (standard time) on the second
# Sunday in March, which is the first Sunday on or after Mar 8.
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 2)
# From 1987 to 2006, DST used to start at 2am (standard time) on the first
# Sunday in April and to end at 2am (DST time) on the last
# Sunday of October, which is the first Sunday on or after Oct 25.
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
DSTEND_1987_2006 = datetime(1, 10, 25, 2)
# From 1967 to 1986, DST used to start at 2am (standard time) on the last
```

(次のページに続く)

(前のページからの続き)

```
# Sunday in April (the one on or after April 24) and to end at 2am (DST time)
# on the last Sunday of October, which is the first Sunday
# on or after Oct 25.
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)
DSTEND_1967_1986 = DSTEND_1987_2006

def us_dst_range(year):
    # Find start and end times for US DST. For years before 1967, return
    # start = end for no DST.
    if 2006 < year:
        dststart, dstend = DSTSTART_2007, DSTEND_2007
    elif 1986 < year < 2007:
        dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
    elif 1966 < year < 1987:
        dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
    else:
        return (datetime(year, 1, 1), ) * 2

    start = first_sunday_on_or_after(dststart.replace(year=year))
    end = first_sunday_on_or_after(dstend.replace(year=year))
    return start, end


class USTimeZone(tzinfo):

    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
        return self.reprname

    def tzname(self, dt):
        if self.dst(dt):
            return self.dstname
        else:
            return self.stdname

    def utcoffset(self, dt):
        return self.stdoffset + self.dst(dt)

    def dst(self, dt):
        if dt is None or dt.tzinfo is None:
            # An exception may be sensible here, in one or both cases.
            # It depends on how you want to treat them.  The default
            # fromutc() implementation (called by the default astimezone()
            # implementation) passes a datetime with dt.tzinfo is self.
            return ZERO
        assert dt.tzinfo is self
```

(次のページに続く)

(前のページからの続き)

```
        start, end = us_dst_range(dt.year)
        # Can't compare naive to aware objects, so strip the timezone from
        # dt first.
        dt = dt.replace(tzinfo=None)
        if start + HOUR <= dt < end - HOUR:
            # DST is in effect.
            return HOUR
        if end - HOUR <= dt < end:
            # Fold (an ambiguous hour): use dt.fold to disambiguate.
            return ZERO if dt.fold else HOUR
        if start <= dt < start + HOUR:
            # Gap (a non-existent hour): reverse the fold rule.
            return HOUR if dt.fold else ZERO
        # DST is off.
        return ZERO

    def fromutc(self, dt):
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        start = start.replace(tzinfo=self)
        end = end.replace(tzinfo=self)
        std_time = dt + self.stdoffset
        dst_time = std_time + HOUR
        if end <= dst_time < end + HOUR:
            # Repeated hour
            return std_time.replace(fold=1)
        if std_time < start or dst_time >= end:
            # Standard time
            return std_time
        if start <= std_time < end - HOUR:
            # Daylight saving time
            return dst_time


Eastern  = USTimeZone(-5, "Eastern",  "EST", "EDT")
Central  = USTimeZone(-6, "Central",  "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific  = USTimeZone(-8, "Pacific",  "PST", "PDT")
```

標準時および夏時間の両方を記述している *tzinfo* のサブクラスでは、夏時間の移行のときに、回避不能の難解な問題が年に 2 度あるので注意してください。具体的な例として、東部アメリカ時刻 (US Eastern, UTC -0500) を考えます。EDT は 3 月の第二日曜日の 1:59 (EST) の 1 分後に開始し、11 月の最初の日曜日の (EDT の) 1:59 に終了します:

```
  UTC    3:MM  4:MM  5:MM  6:MM  7:MM  8:MM
  EST   22:MM 23:MM  0:MM  1:MM  2:MM  3:MM
  EDT   23:MM  0:MM  1:MM  2:MM  3:MM  4:MM

start   22:MM 23:MM  0:MM  1:MM  3:MM  4:MM

  end   23:MM  0:MM  1:MM  1:MM  2:MM  3:MM
```

DST の開始 ("start" ライン) で、ローカルの実時間は 1:59 から 3:00 に飛びます。この日には、2:MM という形式の実時間は意味をなさないので、DST が始まった日に `astimezone(Eastern)` は "hour == 2"となる結果を返すことはありません。例として、2016 年の春方向の移行では、次のような結果になります:

```
>>> from datetime import datetime, timezone
>>> from tzinfo_examples import HOUR, Eastern
>>> u0 = datetime(2016, 3, 13, 5, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname())
...
05:00:00 UTC = 00:00:00 EST
06:00:00 UTC = 01:00:00 EST
07:00:00 UTC = 03:00:00 EDT
08:00:00 UTC = 04:00:00 EDT
```

DST が終了 ("end" ライン) で、更なる問題が潜んでいます: ローカルの実時間で、曖昧さ無しに時を綴れない 1 時間が存在します: それは夏時間の最後の 1 時間です。東部では、夏時間が終了する日の UTC での 5:MM 形式の時間がそれです。ローカルの実時間は (夏時間の) 1:59 から (標準時の) 1:00 に再び巻き戻されます。ローカルの時刻における 1:MM は曖昧です。そして `astimezone()` は 2 つの隣り合う UTC 時間を同じローカルの時間に対応付けて、ローカルの時計の振る舞いを真似ます。東部の例では、5:MM および 6:MM という形式の UTC 時刻は両方とも東部時刻に変換された際に 1:MM に対応付けられますが、それ以前の時間は *fold* 属性を 0 にし、以降の時間では 1 にします。例えば、2016 年での秋方向の移行では、次のような結果になります:

```
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname(), t.fold)
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0
```

fold 属性が異なるだけの *datetime* インスタンスは比較において等しいとみなされることに注意してください。

壁時間に関する曖昧さは、明示的に *fold* 属性を検証するか、*timezone* が使用されたハイブリッドな *tzinfo* サブクラスか、そのほかの絶対時間差を示す *tzinfo* サブクラス (EST (-5 時間の絶対時間差) のみを表すクラスや、EDT (-4 時間の絶対時間差) のみを表すクラス) を使用すると回避できます。このような曖昧さを許容できないアプリケーションは、このような手法によって回避すべきです。

参考:

dateutil.tz *datetime* モジュールには (UTC からの任意の固定オフセットを扱う) 基本的な *timezone* クラスと、(UTC タイムゾーンのインスタンスである) *timezone.utc* 属性があります。

dateutil.tz ライブラリは Python に *IANA* **タイムゾーンデータベース** (オルソンデータベースとしても知られています) を導入するもので、これを使うことが推奨されています。

IANA **タイムゾーンデータベース** (しばしば tz、tzdata や zoneinfo と呼ばれる) タイムゾーンデータベースはコードとデータを保持しており、それらは地球全体にわたる多くの代表的な場所のローカル時刻の履歴を表しています。政治団体によるタイムゾーンの境界、UTC オフセット、夏時間のルールの変更を反映するため、定期的にデータベースが更新されます。

8.1.9 timezone オブジェクト

timezone クラスは *tzinfo* のサブクラスで、各インスタンスは UTC からの固定されたオフセットで定義されたタイムゾーンを表しています。

このクラスのオブジェクトは、一年のうち異なる日に異なるオフセットが使われていたり、常用時 (civil time) に歴史的な変化が起きた場所のタイムゾーン情報を表すのには使えないので注意してください。

class datetime.**timezone**(*offset*, *name=None*)

ローカル時刻と UTC の差分を表す *timedelta* オブジェクトを *offset* 引数に指定しなくてはいけません。これは `-timedelta(hours=24)` から `timedelta(hours=24)` までの両端を含まない範囲に収まっていなくてはなりません。そうでない場合 *ValueError* が送出されます。

The *name* argument is optional. If specified it must be a string that will be used as the value returned by the *datetime.tzname()* method.

バージョン 3.2 で追加.

バージョン 3.7 で変更: UTC オフセットが分単位でなければならない制限が無くなりました。

timezone.**utcoffset**(*dt*)

timezone インスタンスが構築されたときに指定された固定値を返します。

dt 引数は無視されます。返り値は、ローカル時刻と UTC の差分に等しい *timedelta* インスタンスです。

バージョン 3.7 で変更: UTC オフセットが分単位でなければならない制限が無くなりました。

timezone.**tzname**(*dt*)

timezone インスタンスが構築されたときに指定された固定値を返します。

name が構築時に与えられなかった場合、`tzname(dt)` によって返される name は以下の様に `offset` の値から生成されます。*offset* が `timedelta(0)` であった場合、name は "UTC" になります。それ以外の場合、'UTC±HH:MM' という書式の文字列になり、± は `offset` を、HH と MM はそれぞれ二桁の `offset.hours` と `offset.minutes` を表現します。

バージョン 3.6 で変更: `offset=timedelta(0)` によって生成される名前はプレーンな *'UTC'* であり `'UTC+00:00'` ではありません。

timezone.**dst**(*dt*)

常に `None` を返します。

timezone.fromutc(*dt*)

dt + offset を返します。*dt* 引数は tzinfo が self になっている aware な *datetime* インスタンスでなければなりません。

以下にクラス属性を示します:

timezone.utc

UTC タイムゾーン timezone(timedelta(0)) です。

8.1.10 strftime() と strptime() の振る舞い

date, *datetime*, *time* オブジェクトは全て strftime(format) メソッドをサポートし、時刻を表現する文字列を明示的な書式文字列で統制して作成しています。

逆に *datetime.strptime()* クラスメソッドは日付や時刻に対応する書式文字列から *datetime* オブジェクトを生成します。

下の表は strftime() と strptime() との高レベルの対比を表しています。

	strftime	strptime
使用法	オブジェクトを与えられた書式に従って文字列に変換する	指定された対応する書式で文字列を構文解析して *datetime* オブジェクトにする
メソッドの種類	インスタンスメソッド	クラスメソッド
メソッドを持つクラス	*date*; *datetime*; *time*	*datetime*
シグネチャ	strftime(format)	strptime(date_string, format)

strftime() と strptime() の書式コード

以下のリストは 1989 C 標準が要求する全ての書式コードで、標準 C 実装があれば全ての環境で動作します。

指定子	意味	使用例	注釈
%a	ロケールの曜日名を短縮形で表示します。	Sun, Mon, ..., Sat (en_US); So, Mo, ..., Sa (de_DE)	(1)
%A	ロケールの曜日名を表示します。	Sunday, Monday, ..., Saturday (en_US); Sonntag, Montag, ..., Samstag (de_DE)	(1)
%w	曜日を 10 進表記した文字列を表示します。0 が日曜日で、6 が土曜日を表します。	0, 1, ..., 6	
%d	0 埋めした 10 進数で表記した月中の日にち。	01, 02, ..., 31	(9)
%b	ロケールの月名を短縮形で表示します。	Jan, Feb, ..., Dec (en_US); Jan, Feb, ..., Dez (de_DE)	(1)
%B	ロケールの月名を表示します。	January, February, ..., December (en_US); Januar, Februar, ..., Dezember (de_DE)	(1)
%m	0 埋めした 10 進数で表記した月。	01, 02, ..., 12	(9)
%y	0 埋めした 10 進数で表記した世紀無しの年。	00, 01, ..., 99	(9)
%Y	西暦 (4 桁) の 10 進表記を表します。	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(2)
%H	0 埋めした 10 進数で表記した時 (24 時間表記)。	00, 01, ..., 23	(9)
%I	0 埋めした 10 進数で表記した時 (12 時間表記)。	01, 02, ..., 12	(9)
%p	ロケールの AM もしくは PM と等価な文字列になります。	AM, PM (en_US); am, pm (de_DE)	(1), (3)

C89 規格により要求されない幾つかの追加のコードが便宜上含まれています。これらのパラメータはすべて ISO 8601 の日付値に対応しています。

指定子	意味	使用例	注釈
%G	ISO week(%V) の内過半数を含む西暦表記の ISO 8601 year です。	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(8)
%u	1 を月曜日を表す 10 進数表記の ISO 8601 weekday です。	1, 2, ..., 7	
%V	週で最初の月曜日を始めとする ISO 8601 week です。Week 01 は 1 月 4 日を含みます。	01, 02, ..., 53	(8), (9)

これらが `strftime()` メソッドと一緒に使用された場合、すべてのプラットフォームで利用できるわけではありません。ISO 8601 year 指定子および ISO 8601 week 指定子は、上記の year および week number 指定子と互換性がありません。不完全またはあいまいな ISO 8601 指定子で `strptime()` を呼び出すと、*ValueError* が送出されます。

Python はプラットフォームの C ライブラリの `strftime()` 関数を呼び出していて、プラットフォームごとにその実装が異なるのはよくあることなので、サポートされる書式コード全体はプラットフォームごとに様々です。手元のプラットフォームでサポートされているフォーマット記号全体を見るには、*strftime(3)* のドキュメントを参照してください。

バージョン 3.6 で追加: %G, %u および %V が追加されました。

技術詳細

大雑把にいうと、`d.strftime(fmt)` は *time* モジュールの `time.strftime(fmt, d.timetuple())` のように動作します。ただし全てのオブジェクトが `timetuple()` メソッドをサポートしているわけではありません。

datetime.strptime() クラスメソッドでは、デフォルト値は `1900-01-01T00:00:00.000` です。書式文字列で指定されなかった部分はデフォルト値から引っ張ってきます。[*4]

`datetime.strptime(date_string, format)` は次の式と等価です:

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

ただし、`datetime.strptime` はサポートしているが `time.strptime` には無い、秒未満の単位やタイムゾーンオフセットの情報が format に 含まれているときは除きます。

time オブジェクトには、年、月、日の値がないため、それらを書式コードを使うことができません。無理矢理使った場合、年は `1900` に置き換えられ、月と日は `1` に置き換えられます。

date オブジェクトには、時、分、秒、マイクロ秒の値がないため、それらの書式コードを使うことができません。無理矢理使った場合、これらの値は `0` に置き換えられます。

[*4] `1900` は閏年ではないので `datetime.strptime('Feb 29', '%b %d')` を渡すと失敗します。

同じ理由で、現在のロケールの文字集合で表現できない Unicode コードポイントを含む書式文字列の対処もプラットフォーム依存です。あるプラットフォームではそういったコードポイントはそのまま出力に出される一方、他のプラットフォームでは strftime が *UnicodeError* を送出したり、その代わりに空文字列を返したりするかもしれません。

注釈:

(1) 書式は現在のロケールに依存するので、出力値について何か仮定するときは注意すべきです。フィールドの順序は様々で (例えば、”月/日/年” と ”日/月/年”) 、出力はロケールのデフォルトエンコーディングでエンコードされた Unicode 文字列を含むかもしれません (例えば、現在のロケールが ja_JP だった場合、デフォルトエンコーディングは eucJP 、SJIS 、utf-8 のいずれかになりえます。*locale.getlocale()* を使って現在のロケールのエンコーディングを確認します) 。

(2) strptime() メソッドは [1, 9999] の範囲の年数全てを構文解析できますが、year < 1000 の範囲の年数は 0 埋めされた 4 桁の数字でなければなりません。

バージョン 3.2 で変更: 以前のバージョンでは、strftime() メソッドは years >= 1900 の範囲の年数しか扱えませんでした。

バージョン 3.3 で変更: バージョン 3.2 では、strftime() メソッドは years >= 1000 の範囲の年数しか扱えませんでした。

(3) strptime() メソッドと共に使われた場合、%p 指定子は出力の時間フィールドのみに影響し、%I 指定子が使われたかのように振る舞います。

(4) *time* モジュールと違い、*datetime* モジュールはうるう秒をサポートしていません。

(5) strptime() メソッドと共に使われた場合、%f 指定子は 1 桁から 6 桁の数字を受け付け、右側から 0 埋めされます。%f は C 標準規格の書式文字セットの拡張です (とは言え、datetime モジュールのオブジェクトそれぞれに実装されているので、どれででも使えます)。

(6) naive オブジェクトでは、書式コード %z および %Z は空文字列に置き換えられます。

aware オブジェクトでは次のようになります:

%z utcoffset() は ±HHMM[SS[.ffffff]] 形式の文字列に変換されます。ここで、HH は UTC オフセットの時間を表す 2 桁の文字列、MM は UTC オフセットの分数を表す 2 桁の文字列、SS は UTC オフセットの秒数を表す 2 桁の文字列、ffffff は UTC オフセットのマイクロ秒数を表す 6 桁の文字列です。オフセットに秒未満の端数が無いときは ffffff 部分は省略され、オフセットに分未満の端数が無いときは ffffff 部分も SS 部分も省略されます。例えば、utcoffset() が timedelta(hours=-3, minutes=-30) を返す場合、%z は文字列 '-0330' に置き換えられます。

バージョン 3.7 で変更: UTC オフセットが分単位でなければならない制限が無くなりました。

バージョン 3.7 で変更: %z 指定子が strptime() メソッドに渡されたときは、時分秒のセパレータとしてコロンが UTC オフセットで使えます。例えば、'+01:00:00' は 1 時間のオフセットだと構文解析されます。加えて、'Z' を渡すことは '+00:00' を渡すことと同等です。

%Z tzname() が None を返した場合、%Z は空文字列に置き換 わります。そうでない場合、%Z は返された値に置き換わりますが、これは文字列でなければなりません。

バージョン 3.2 で変更: %z 指定子が strptime() メソッドに与えられた場合、aware な *datetime* オブジェクトが作成されます。返り値の tzinfo は *timezone* インスタンスになっています。

(7) strptime() メソッドと共に使われた場合、%U と %W 指定子は、曜日と年 (%Y) が指定された場合の計算でのみ使われます。

(8) %U および %W と同様に、%V は曜日と ISO 年 (%G) が strptime() の書式文字列の中で指定された場合に計算でのみ使われます。%G と %Y は互いに完全な互換性を持たないことにも注意してください。

(9) strptime() メソッドと共に使われたときは、書式 %d, %m, %H, %I, %M, %S, %J, %U, %W, %V の後ろに続ける 0 は任意です。書式 %y では後ろに続ける 0 は必須です。

脚注

8.2 calendar --- 一般的なカレンダーに関する関数群

ソースコード: Lib/calendar.py

このモジュールは Unix の **cal** プログラムのようなカレンダー出力を行い、それに加えてカレンダーに関する有益な関数群を提供します。標準ではこれらのカレンダーは（ヨーロッパの慣例に従って）月曜日を週の始まりとし、日曜日を最後の日としています。*setfirstweekday()* を用いることで、日曜日 (6) や他の曜日を週の始まりに設定することができます。日付を表す引数は整数値で与えます。関連する機能として、*datetime* と *time* モジュールも参照してください。

このモジュールに定義された関数やクラスは理想化されたカレンダー、つまり現在のグレゴリオ暦を過去と未来両方に無限に拡張したものを使っています。これはすべての計算の基礎のカレンダーとなっている、Dershowitz と Reingold の本 "Calendrical Calculations" 中の "proleptic Gregorian" のカレンダーの定義に合致します。ゼロと負の年は ISO 8601 の基準に定められている通りに扱われます。0 年は 1 BC、-1 年は 2 BC、のように続きます。

class calendar.**Calendar**(*firstweekday=0*)

Calendar オブジェクトを作ります。*firstweekday* は整数で週の始まりの曜日を指定するものです。0 が月曜 (デフォルト)、6 なら日曜です。

Calendar オブジェクトは整形されるカレンダーのデータを準備するために使えるいくつかのメソッドを提供しています。しかし整形機能そのものは提供していません。それはサブクラスの仕事なのです。

Calendar インスタンスには以下のメソッドがあります:

iterweekdays()

曜日の数字を一週間分生成するイテレータを返します。イテレータから得られる最初の数字は *firstweekday* が返す数字と同じになります。

itermonthdates(*year*, *month*)

year 年 *month* (1--12) 月に対するイテレータを返します。このイテレータはその月の全ての日、およびその月が始まる前の日とその月が終わった後の日のうち、週の欠けを埋めるために必要な日を (*datetime.date* オブジェクトとして) 返します。

itermonthdays(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*, but not restricted by the *datetime.date* range. Days returned will simply be day of the month numbers. For the days outside of the specified month, the day number is `0`.

itermonthdays2(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*, but not restricted by the *datetime.date* range. Days returned will be tuples consisting of a day of the month number and a week day number.

itermonthdays3(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*, but not restricted by the *datetime.date* range. Days returned will be tuples consisting of a year, a month and a day of the month numbers.

バージョン 3.7 で追加.

itermonthdays4(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*, but not restricted by the *datetime.date* range. Days returned will be tuples consisting of a year, a month, a day of the month, and a day of the week numbers.

バージョン 3.7 で追加.

monthdatescalendar(*year*, *month*)

year 年 *month* 月の週のリストを返します。週は全て七つの *datetime.date* オブジェクトからなるリストです。

monthdays2calendar(*year*, *month*)

year 年 *month* 月の週のリストを返します。週は全て七つの日付の数字と曜日を表す数字のタプルからなるリストです。

monthdayscalendar(*year*, *month*)

year 年 *month* 月の週のリストを返します。週は全て七つの日付の数字からなるリストです。

yeardatescalendar(*year*, *width=3*)

指定された年のデータを整形に向く形で返します。返される値は月の並びのリストです。月の並びは最大で *width* ヶ月 (デフォルトは 3 ヶ月) 分です。各月は 4 ないし 6 週からなり、各週は 1 ないし 7 日からなります。各日は *datetime.date* オブジェクトです。

yeardays2calendar(*year*, *width=3*)

指定された年のデータを整形に向く形で返します (*yeardatescalendar()* と同様です)。週のリストの中が日付の数字と曜日の数字のタプルになります。月の範囲外の部分の日付はゼロです。

yeardayscalendar(*year*, *width=3*)

指定された年のデータを整形に向く形で返します (*yeardatescalendar()* と同様です)。週のリストの中が日付の数字になります。月の範囲外の日付はゼロです。

class calendar.TextCalendar(*firstweekday=0*)

このクラスはプレインテキストのカレンダーを生成するのに使えます。

TextCalendar インスタンスには以下のメソッドがあります:

formatmonth(*theyear*, *themonth*, *w=0*, *l=0*)
ひと月分のカレンダーを複数行の文字列で返します。*w* により日の列幅を変えることができ、それらは中央揃えされます。*l* により各週の表示される行数を変えることができます。*setfirstweekday()* メソッドでセットされた週の最初の曜日に依存します。

prmonth(*theyear*, *themonth*, *w=0*, *l=0*)
formatmonth() で返されるひと月分のカレンダーを出力します。

formatyear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)
m 列からなる一年間のカレンダーを複数行の文字列で返します。任意の引数 *w*, *l*, *c* はそれぞれ、日付列の表示幅、各週の行数及び月と月の間のスペースの数を変更するためのものです。*setfirstweekday()* メソッドでセットされた週の最初の曜日に依存します。カレンダーを出力できる最初の年はプラットフォームに依存します。

pryear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)
formatyear() で返される一年間のカレンダーを出力します。

class calendar.**HTMLCalendar**(*firstweekday=0*)
このクラスは HTML のカレンダーを生成するのに使えます。

HTMLCalendar インスタンスには以下のメソッドがあります:

formatmonth(*theyear*, *themonth*, *withyear=True*)
ひと月分のカレンダーを HTML のテーブルとして返します。*withyear* が真であればヘッダには年も含まれます。そうでなければ月の名前だけが使われます。

formatyear(*theyear*, *width=3*)
一年分のカレンダーを HTML のテーブルとして返します。*width* の値 (デフォルトでは 3 です) は何ヶ月分を一行に収めるかを指定します。

formatyearpage(*theyear*, *width=3*, *css='calendar.css'*, *encoding=None*)
一年分のカレンダーを一つの完全な HTML ページとして返します。*width* の値 (デフォルトでは 3 です) は何ヶ月分を一行に収めるかを指定します。*css* は使われるカスケーディングスタイルシートの名前です。スタイルシートを使わないようにするために *None* を渡すこともできます。*encoding* には出力に使うエンコーディングを指定します (デフォルトではシステムデフォルトのエンコーディングです)。

HTMLCalendar has the following attributes you can override to customize the CSS classes used by the calendar:

cssclasses
A list of CSS classes used for each weekday. The default class list is:

```
cssclasses = ["mon", "tue", "wed", "thu", "fri", "sat", "sun"]
```

more styles can be added for each day:

```
cssclasses = ["mon text-bold", "tue", "wed", "thu", "fri", "sat", "sun red"]
```

Note that the length of this list must be seven items.

cssclass_noday

The CSS class for a weekday occurring in the previous or coming month.

バージョン 3.7 で追加.

cssclasses_weekday_head

A list of CSS classes used for weekday names in the header row. The default is the same as *cssclasses*.

バージョン 3.7 で追加.

cssclass_month_head

The month's head CSS class (used by `formatmonthname()`). The default value is `"month"`.

バージョン 3.7 で追加.

cssclass_month

The CSS class for the whole month's table (used by *formatmonth()*). The default value is `"month"`.

バージョン 3.7 で追加.

cssclass_year

The CSS class for the whole year's table of tables (used by *formatyear()*). The default value is `"year"`.

バージョン 3.7 で追加.

cssclass_year_head

The CSS class for the table head for the whole year (used by *formatyear()*). The default value is `"year"`.

バージョン 3.7 で追加.

Note that although the naming for the above described class attributes is singular (e.g. `cssclass_month cssclass_noday`), one can replace the single CSS class with a space separated list of CSS classes, for example:

```
"text-bold text-red"
```

Here is an example how `HTMLCalendar` can be customized:

```
class CustomHTMLCal(calendar.HTMLCalendar):
    cssclasses = [style + " text-nowrap" for style in
                  calendar.HTMLCalendar.cssclasses]
    cssclass_month_head = "text-center month-head"
```

(次のページに続く)

(前のページからの続き)

```
    cssclass_month = "text-center month"
    cssclass_year = "text-italic lead"
```

class calendar.**LocaleTextCalendar**(*firstweekday=0*, *locale=None*)

この *TextCalendar* のサブクラスではコンストラクタにロケール名を渡すことができ、メソッドの返り値で月や曜日が指定されたロケールのものになります。このロケールがエンコーディングを含む場合には、月や曜日の入った文字列はユニコードとして返されます。

class calendar.**LocaleHTMLCalendar**(*firstweekday=0*, *locale=None*)

この *HTMLCalendar* のサブクラスではコンストラクタにロケール名を渡すことができ、メソッドの返り値で月や曜日が指定されたロケールのものになります。このロケールがエンコーディングを含む場合には、月や曜日の入った文字列はユニコードとして返されます。

注釈: これら 2 つのクラスの `formatweekday()` と `formatmonthname()` メソッドは、一時的に現在の locale を指定された *locale* に変更します。現在の locale はプロセス全体に影響するので、これらはスレッドセーフではありません。

単純なテキストのカレンダーに関して、このモジュールには以下のような関数が提供されています。

calendar.**setfirstweekday**(*weekday*)

週の最初の曜日 (`0` は月曜日, `6` は日曜日) を設定します。定数 `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` 及び `SUNDAY` は便宜上提供されています。例えば、日曜日を週の開始日に設定するときは:

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

calendar.**firstweekday**()

現在設定されている週の最初の曜日を返します。

calendar.**isleap**(*year*)

year が閏年なら *True* を、そうでなければ *False* を返します。

calendar.**leapdays**(*y1*, *y2*)

範囲 (*y1* ... *y2*) 指定された期間の閏年の回数を返します。ここで *y1* や *y2* は年を表します。

この関数は、世紀の境目をまたぐ範囲でも正しく動作します。

calendar.**weekday**(*year*, *month*, *day*)

year (`1970`--...), *month* (`1`--`12`), *day* (`1`--`31`) で与えられた日の曜日 (`0` は月曜日) を返します。

calendar.**weekheader**(*n*)

短縮された曜日名を含むヘッダを返します。*n* は各曜日を何文字で表すかを指定します。

calendar.**monthrange**(*year*, *month*)

year と *month* で指定された月の一日の曜日と日数を返します。

calendar.**monthcalendar**(*year*, *month*)
月のカレンダーを行列で返します。各行が週を表し、月の範囲外の日は 0 になります。それぞれの週は *setfirstweekday()* で設定をしていない限り月曜日から始まります。

calendar.**prmonth**(*theyear*, *themonth*, *w=0*, *l=0*)
month() 関数によって返される月のカレンダーを出力します。

calendar.**month**(*theyear*, *themonth*, *w=0*, *l=0*)
TextCalendar の `formatmonth()` メソッドを利用して、ひと月分のカレンダーを複数行の文字列で返します。

calendar.**prcal**(*year*, *w=0*, *l=0*, *c=6*, *m=3*)
calendar() 関数で返される一年間のカレンダーを出力します。

calendar.**calendar**(*year*, *w=2*, *l=1*, *c=6*, *m=3*)
TextCalendar の `formatyear()` メソッドを利用して、3 列からなる一年間のカレンダーを複数行の文字列で返します。

calendar.**timegm**(*tuple*)
カレンダーと直接は関係無いが、*time* モジュールの *gmtime()* 関数が返す形式の時刻を表すタプルを引数に取り、1970 を基点とするエポック時刻で POSIX エンコーディングであると仮定して、対応する Unix タイムスタンプの値を返します。実際には、*time.gmtime()* と *timegm()* はお互いの逆関数です。

calendar モジュールの以下のデータ属性を利用することができます:

calendar.**day_name**
現在のロケールでの曜日を表す配列です。

calendar.**day_abbr**
現在のロケールでの短縮された曜日を表す配列です。

calendar.**month_name**
現在のロケールでの月の名を表す配列です。この配列は通常の約束事に従って、1 月を数字の 1 で表しますので、長さが 13 ある代わりに `month_name[0]` が空文字列になります。

calendar.**month_abbr**
現在のロケールでの短縮された月の名を表す配列です。この配列は通常の約束事に従って、1 月を数字の 1 で表しますので、長さが 13 ある代わりに `month_abbr[0]` が空文字列になります。

参考:

datetime **モジュール**　*time* モジュールと似た機能を持った日付と時間用のオブジェクト指向インタフェース。

time **モジュール**　時間に関連した低水準の関数群。

8.3 collections --- コンテナデータ型

ソースコード: Lib/collections/___init___.py

このモジュールは、汎用の Python 組み込みコンテナ *dict*, *list*, *set*, および *tuple* に代わる、特殊なコンテナデータ型を実装しています。

namedtuple()	名前付きフィールドを持つタプルのサブクラスを作成するファクトリ関数
deque	両端における append や pop を高速に行えるリスト風のコンテナ
ChainMap	複数のマッピングの一つのビューを作成する辞書風のクラス
Counter	ハッシュ可能なオブジェクトを数え上げる辞書のサブクラス
OrderedDict	項目が追加された順序を記憶する辞書のサブクラス
defaultdict	ファクトリ関数を呼び出して存在しない値を供給する辞書のサブクラス
UserDict	辞書のサブクラス化を簡単にする辞書オブジェクトのラッパ
UserList	リストのサブクラス化を簡単にするリストオブジェクトのラッパ
UserString	文字列のサブクラス化を簡単にする文字列オブジェクトのラッパ

Deprecated since version 3.3, will be removed in version 3.10: コレクション抽象基底クラス が *collections.abc* モジュールに移動されました。後方互換性のため、それらは引き続き Python 3.9 モジュールでも利用できます。

8.3.1 ChainMap オブジェクト

バージョン 3.3 で追加.

ChainMap クラスは、複数のマッピングを素早く連結し、一つの単位として扱うために提供されています。これはたいてい、新しい辞書を作成して *update()* を繰り返すよりも早いです。

このクラスはネストされたスコープをシミュレートするのに使え、テンプレート化に便利です。

class collections.**ChainMap**(**maps*)

ChainMap は、複数の辞書やその他のマッピングをまとめて、一つの、更新可能なビューを作成します。*maps* が指定されないなら、一つの空辞書が与えられますから、新しいチェーンは必ず一つ以上のマッピングをもちます。

根底のマッピングはリストに保存されます。このリストはパブリックで、*maps* 属性を使ってアクセスや更新できます。それ以外に状態はありません。

探索は、根底のマッピングをキーが見つかるまで引き続き探します。対して、書き込み、更新、削除は、最初のマッピングのみ操作します。

ChainMap は、根底のマッピングを参照によって組み込みます。ですから、根底のマッピングの一つが更新されると、その変更は *ChainMap* に反映されます。

通常の辞書のメソッドすべてがサポートされています。さらに、*maps* 属性、新しいサブコンテキストを作成するメソッド、最初のマッピング以外のすべてにアクセスするためのプロパティがあります:

maps

マッピングのユーザがアップデートできるリストです。このリストは最初に探されるものから最後に探されるものの順に並んでいます。これが唯一のソートされた状態であり、変更してマッピングが探される順番を変更できます。このリストは常に一つ以上のマッピングを含んでいなければなりません。

new_child(*m=None*)

新しい辞書の後ろに現在のインスタンスにある全ての辞書が続いたものを持つ、新しい *ChainMap* を返します。m が指定された場合、それがマッピングのリストの先頭の新しい辞書になります; 指定されていない場合、d.new_child() が ChainMap({}, *d.maps) と同等となるように空の辞書が使われます。このメソッドは、親マッピングを変更することなく値を更新できるサブコンテキストを作成するのに使われます。

バージョン 3.4 で変更: オプションの m 引数が追加されました。

parents

現在のインスタンスの最初のマッピング以外のすべてのマッピングを含む新しい *ChainMap* を返すプロパティです。これは最初のマッピングを検索から飛ばすのに便利です。使用例は nonlocal キーワードを **ネストされたスコープ** に使う例と似ています。この使用例はまた、組み込み *super()* 関数にも似ています。d.parents への参照は ChainMap(*d.maps[1:]) と等価です。

ChainMap() の反復順序は、マッピングオブジェクトを末尾から先頭に向かう走査で決まることに注意してください。

```
>>> baseline = {'music': 'bach', 'art': 'rembrandt'}
>>> adjustments = {'art': 'van gogh', 'opera': 'carmen'}
>>> list(ChainMap(adjustments, baseline))
['music', 'art', 'opera']
```

これは、末尾のマッピングオブジェクトから始めた一連の *dict.update()* の呼び出しと同じ順序になります。

```
>>> combined = baseline.copy()
>>> combined.update(adjustments)
>>> list(combined)
['music', 'art', 'opera']
```

参考:

- Enthought 社の CodeTools パッケージ に含まれる MultiContext クラス は、チェーン内のすべてのマッピングへの書き込みをサポートするオプションを持ちます。
- Django のテンプレート用の *Context class* '_ **は、読み出し専用のマッピングのチェーンです。** *:meth:'~collections.ChainMap.new_child* メソッドや *parents* プロパティに似た push や pop の機能もあります。
- Nested Contexts recipe は、書き込みその他の変更が最初のマッピングにのみ適用されるか、チェーンのすべてのマッピングに適用されるか、制御するオプションを持ちます。
- 非常に単純化した読み出し専用バージョンの Chainmap。

ChainMap の例とレシピ

この節では、チェーンされたマッピングを扱う様々な手法を示します。

Python の内部探索チェーンをシミュレートする例:

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

ユーザ指定のコマンドライン引数、環境変数、デフォルト値、の順に優先させる例:

```
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k: v for k, v in vars(namespace).items() if v is not None}

combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

ChainMap を使ってネストされたコンテキストをシミュレートするパターンの例:

```
c = ChainMap()          # Create root context
d = c.new_child()       # Create nested child context
e = c.new_child()       # Child of c, independent from d
e.maps[0]               # Current context dictionary -- like Python's locals()
e.maps[-1]              # Root context -- like Python's globals()
e.parents               # Enclosing context chain -- like Python's nonlocals

d['x'] = 1              # Set value in current context
d['x']                  # Get first key in the chain of contexts
del d['x']              # Delete from current context
list(d)                 # All nested values
k in d                  # Check all nested values
len(d)                  # Number of nested values
d.items()               # All nested items
dict(d)                 # Flatten into a regular dictionary
```

ChainMap クラスは、探索はチェーン全体に対して行いますが、更新 (書き込みと削除) は最初のマッピングに対してのみ行います。しかし、深い書き込みと削除を望むなら、チェーンの深いところで見つかったキーを更新するサブクラスを簡単に作れます:

```
class DeepChainMap(ChainMap):
    'Variant of ChainMap that allows direct updates to inner scopes'

    def __setitem__(self, key, value):
```

(次のページに続く)

(前のページからの続き)

```
        for mapping in self.maps:
            if key in mapping:
                mapping[key] = value
                return
        self.maps[0][key] = value

    def __delitem__(self, key):
        for mapping in self.maps:
            if key in mapping:
                del mapping[key]
                return
        raise KeyError(key)

>>> d = DeepChainMap({'zebra': 'black'}, {'elephant': 'blue'}, {'lion': 'yellow'})
>>> d['lion'] = 'orange'         # update an existing key two levels down
>>> d['snake'] = 'red'           # new keys get added to the topmost dict
>>> del d['elephant']            # remove an existing key one level down
>>> d                            # display result
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {}, {'lion': 'orange'})
```

8.3.2 Counter オブジェクト

便利で迅速な検数をサポートするカウンタツールが提供されています。例えば:

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

class collections.**Counter**(*[iterable-or-mapping]*)

Counter はハッシュ可能なオブジェクトをカウントする *dict* のサブクラスです。これは、要素を辞書のキーとして保存し、そのカウントを辞書の値として保存するコレクションです。カウントは、0 や負のカウントを含む整数値をとれます。*Counter* クラスは、他の言語のバッグや多重集合のようなものです。

要素は、*iterable* から数え上げられたり、他の *mapping* (やカウンタ) から初期化されます:

```
>>> c = Counter()                           # a new, empty counter
>>> c = Counter('gallahad')                 # a new counter from an iterable
```

(次のページに続く)

(前のページからの続き)

```
>>> c = Counter({'red': 4, 'blue': 2})      # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)             # a new counter from keyword args
```

カウンタオブジェクトは辞書のインタフェースを持ちますが、存在しない要素に対して *KeyError* を送出する代わりに 0 を返すという違いがあります:

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                              # count of a missing element is zero
0
```

カウントを 0 に設定しても、要素はカウンタから取り除かれません。完全に取り除くには、`del` を使ってください:

```
>>> c['sausage'] = 0                        # counter entry with a zero count
>>> del c['sausage']                        # del actually removes the entry
```

バージョン 3.1 で追加.

バージョン 3.7 で変更: *Counter* は class:*dict* のサブクラスとして要素の挿入順を維持する機能を継承しました。*Counter* オブジェクトに対する数学演算も順序を維持します。結果は左の被演算子で最初に要素が出現するあとに、右の被演算子で要素が出現する順序になります。

カウンタオブジェクトは、すべての辞書で利用できるメソッドに加えて、次の 3 つのメソッドをサポートしています。

elements()

それぞれの要素を、そのカウント分の回数だけ繰り返すイテレータを返します。要素は挿入した順番で返されます。ある要素のカウントが 1 未満なら、*elements()* はそれを無視します。

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> sorted(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common([*n*]**)**

最も多い *n* 要素を、カウントが多いものから少ないものまで順に並べたリストを返します。*n* が省略されるか None であれば、*most_common()* はカウンタの **すべての** 要素を返します。等しいカウントの要素は挿入順に並べられます:

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('b', 2), ('r', 2)]
```

subtract([*iterable-or-mapping*]**)**

要素から *iterable* の要素または *mapping* の要素が引かれます。*dict.update()* に似ていますが、カウントを置き換えるのではなく引きます。入力も出力も、0 や負になりえます。

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
```

(次のページに続く)

(前のページからの続き)

```
>>> c.subtract(d)
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

バージョン 3.2 で追加.

普通の辞書のメソッドは、以下の 2 つのメソッドがカウンタに対して異なる振る舞いをするのを除き、*Counter* オブジェクトにも利用できます。

fromkeys(*iterable*)

このクラスメソッドは *Counter* オブジェクトには実装されていません。

update(*[iterable-or-mapping]*)

要素が *iterable* からカウントされるか、別の *mapping* (やカウンタ) が追加されます。*dict.update()* に似ていますが、カウントを置き換えるのではなく追加します。また、*iterable* には (key, value) 対のシーケンスではなく、要素のシーケンスが求められます。

Counter オブジェクトを使ったよくあるパターン:

```
sum(c.values())                 # total of all counts
c.clear()                       # reset all counts
list(c)                         # list unique elements
set(c)                          # convert to a set
dict(c)                         # convert to a regular dictionary
c.items()                       # convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs))    # convert from a list of (elem, cnt) pairs
c.most_common()[:-n-1:-1]       # n least common elements
+c                              # remove zero and negative counts
```

Counter オブジェクトを組み合わせて多重集合 (1 以上のカウントをもつカウンタ) を作るために、いくつかの数学演算が提供されています。足し算と引き算は、対応する要素を足したり引いたりすることによってカウンタを組み合わせます。積集合と和集合は、対応するカウントの最大値と最小値を返します。それぞれの演算はカウントに符号がついた入力を受け付けますが、カウントが 0 以下である結果は出力から除かれます。

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                       # add two counters together:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                       # subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d                       # intersection:  min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d                       # union:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
```

単項加算および減算は、空カウンタの加算や空カウンタからの減算へのショートカットです。

```
>>> c = Counter(a=2, b=-4)
>>> +c
```

(次のページに続く)

(前のページからの続き)

```
Counter({'a': 2})
>>> -c
Counter({'b': 4})
```

バージョン 3.3 で追加: 単項加算、単項減算、in-place の多重集合操作のサポートが追加されました。

注釈: カウンタはもともと、推移するカウントを正の整数で表すために設計されました。しかし、他の型や負の値を必要とするユースケースを不必要に排除することがないように配慮されています。このようなユースケースの助けになるように、この節で最低限の範囲と型の制限について記述します。

- *Counter* クラス自体は辞書のサブクラスで、キーと値に制限はありません。値はカウントを表す数であることを意図していますが、値フィールドに任意のものを保存 **できます** 。
- *most_common()* メソッドが要求するのは、値が順序付け可能なことだけです。
- `c[key] += 1` のようなインプレース演算では、値の型に必要なのは 足し算と引き算ができることだけです。よって分数、浮動小数点数、小数も使え、負の値がサポートされています。これと同じことが、負や 0 の値を入力と出力に許す *update()* と *subtract()* メソッド にも言えます。
- 多重集合メソッドは正の値を扱うユースケースに対してのみ設計されています。入力は負や 0 に出来ますが、正の値の出力のみが生成されます。型の制限はありませんが、値の型は足し算、引き算、比較をサポートしている必要があります。
- *elements()* メソッドは整数のカウントを要求します。これは 0 と負のカウントを無視します。

参考:

- Smalltalk の Bag class 。
- Wikipedia の Multisets の項目。
- C++ multisets の例を交えたチュートリアル。
- 数学的な多重集合の演算とそのユースケースは、*Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19* を参照してください。
- 与えられた要素の集まりからなる与えられた大きさの相違なる多重集合をすべて数え上げるには、*itertools.combinations_with_replacement()* を参照してください:

```
map(Counter, combinations_with_replacement('ABC', 2)) # --> AA AB AC BB BC CC
```

8.3.3 deque オブジェクト

class collections.deque([*iterable*[, *maxlen*]])

iterable で与えられるデータから、新しい deque オブジェクトを (*append()* をつかって) 左から右に初期化して返します。*iterable* が指定されない場合、新しい deque オブジェクトは空になります。

Deque とは、スタックとキューを一般化したものです (この名前は「デック」と発音され、これは「double-ended queue」の省略形です)。Deque はどちらの側からも append と pop が可能で、スレッドセーフでメモリ効率がよく、どちらの方向からもおよそ `O(1)` のパフォーマンスで実行できます。

list オブジェクトでも同様の操作を実現できますが、これは高速な固定長の操作に特化されており、内部のデータ表現形式のサイズと位置を両方変えるような `pop(0)` や `insert(0, v)` などの操作ではメモリ移動のために `O(n)` のコストを必要とします。

maxlen が指定されなかったり `None` だった場合、deque は任意のサイズまで大きくなります。そうでない場合、deque のサイズは指定された最大長に制限されます。長さが制限された deque がいっぱいになると、新しい要素を追加するときに追加した要素数分だけ追加したのと反対側から要素が捨てられます。長さが制限された deque は Unix における `tail` フィルタと似た機能を提供します。トランザクションの tracking や最近使った要素だけを残したいデータプール (pool of data) などにも便利です。

Deque オブジェクトは以下のようなメソッドをサポートしています:

append(*x*)

x を deque の右側につけ加えます。

appendleft(*x*)

x を deque の左側につけ加えます。

clear()

deque からすべての要素を削除し、長さを 0 にします。

copy()

deque の浅いコピーを作成します。

バージョン 3.5 で追加.

count(*x*)

deque の *x* に等しい要素を数え上げます。

バージョン 3.2 で追加.

extend(*iterable*)

イテラブルな引数 iterable から得られる要素を deque の右側に追加し拡張します。

extendleft(*iterable*)

イテラブルな引数 iterable から得られる要素を deque の左側に追加し拡張します。注意: 左から追加した結果は、イテラブルな引数の順序とは逆になります。

index(*x*[, *start*[, *stop*]])
deque 内の *x* の位置を返します (インデックス *start* からインデックス *stop* の両端を含む範囲で)。最初のマッチを返すか、見つからない場合には *ValueError* を発生させます。

バージョン 3.5 で追加.

insert(*i*, *x*)
x を deque の位置 *i* に挿入します。

挿入によって、長さに制限のある deque の長さが *maxlen* を超える場合、*IndexError* が発生します。

バージョン 3.5 で追加.

pop()
deque の右側から要素をひとつ削除し、その要素を返します。要素がひとつも存在しない場合は *IndexError* を発生させます。

popleft()
deque の左側から要素をひとつ削除し、その要素を返します。要素がひとつも存在しない場合は *IndexError* を発生させます。

remove(*value*)
value の最初に現れるものを削除します。要素が見付からないない場合は *ValueError* を送出します。

reverse()
deque の要素をインプレースに反転し、None を返します。

バージョン 3.2 で追加.

rotate(*n=1*)
deque の要素を全体で *n* ステップだけ右にローテートします。*n* が負の値の場合は、左にローテートします。

deque が空でないときは、deque をひとつ右にローテートすることは `d.appendleft(d.pop())` と同じで、deque をひとつ左にローテートすることは `d.append(d.popleft())` と同じです。

deque オブジェクトは読み出し専用属性も 1 つ提供しています:

maxlen
deque の最大長で、制限されていなければ None です。

バージョン 3.1 で追加.

上記に加え、deque はイテレーション, pickle 化, `len(d)`, `reversed(d)`, `copy.copy(d)`, `copy.deepcopy(d)`, `in` 演算子による包含の検査, `d[0]` のような添字による参照をサポートしています。添字によるアクセスは、両端の要素では O(1) ですが、中央部分の要素では O(n) と遅くなります。高速なランダムアクセスのためには、代わりにリストを使ってください。

バージョン 3.5 から deque は `__add__()`, `__mul__()`, `__imul__()` をサポートしました。

例:

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...     print(elem.upper())
G
H
I

>>> d.append('j')                    # add a new entry to the right side
>>> d.appendleft('f')                # add a new entry to the left side
>>> d                                # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # return and remove the rightmost item
'j'
>>> d.popleft()                      # return and remove the leftmost item
'f'
>>> list(d)                          # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                             # peek at leftmost item
'g'
>>> d[-1]                            # peek at rightmost item
'i'

>>> list(reversed(d))                # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                         # search the deque
True
>>> d.extend('jkl')                  # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                      # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                     # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))               # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                        # empty the deque
>>> d.pop()                          # cannot pop from an empty deque
Traceback (most recent call last):
    File "<pyshell#6>", line 1, in -toplevel-
        d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')              # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

deque のレシピ

この節では deque を使った様々なアプローチを紹介します。

長さが制限された deque は Unix における `tail` フィルタに相当する機能を提供します:

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return deque(f, n)
```

deque を使用する別のアプローチは、右に要素を追加し左から要素を取り出すことで最近追加した要素のシーケンスを保持することです:

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # http://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / n
```

ラウンドロビンスケジューラ は、入力されたイテレータを *deque* に格納することで実装できます。値は、位置 0 にある選択中のイテレータから取り出されます。そのイテレータが値を出し切った場合は、*popleft()* で除去できます; そうでない場合は、*rotate()* メソッドで末尾に回せます:

```
def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    iterators = deque(map(iter, iterables))
    while iterators:
        try:
            while True:
                yield next(iterators[0])
                iterators.rotate(-1)
        except StopIteration:
            # Remove an exhausted iterator.
            iterators.popleft()
```

rotate() メソッドは、*deque* のスライスや削除の機能を提供します。例えば、純粋な Python 実装の `del d[n]` は `rotate()` メソッドを頼りに、pop される要素の位置を割り出します:

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

deque のスライスの実装でも、同様のアプローチを使います。まず対象となる要素を *rotate()* によって

deque の左端まで移動させてから、*popleft()* で古い要素を削除します。そして、*extend()* で新しい要素を追加したのち、循環を逆にします。このアプローチをやや変えたものとして、Forth スタイルのスタック操作、つまり dup, drop, swap, over, pick, rot, および roll を実装するのも簡単です。

8.3.4 defaultdict オブジェクト

class collections.**defaultdict**([*default_factory*[, ...]])

新しいディクショナリ様のオブジェクトを返します。*defaultdict* は組み込みの *dict* のサブクラスです。メソッドをオーバーライドし、書き込み可能なインスタンス変数を 1 つ追加している以外は *dict* クラスと同じです。同じ部分については以下では省略されています。

1 つ目の引数は *default_factory* 属性の初期値です。デフォルトは None です。残りの引数はキーワード引数も含め、*dict* のコンストラクタに与えられた場合と同様に扱われます。

defaultdict オブジェクトは標準の *dict* に加えて、以下のメソッドを実装しています:

__missing__(*key*)

もし *default_factory* 属性が None であれば、このメソッドは *KeyError* 例外を、*key* を引数として発生させます。

もし *default_factory* 属性が None でない場合、このメソッドは引数なしで呼び出され、与えらえた *key* に対応するデフォルト値を提供します。この値は、辞書内に *key* に対応して登録され、最後に返されます。

もし *default_factory* の呼出が例外を発生させた場合には、変更せずそのまま例外を投げます。

このメソッドは *dict* クラスの __getitem__() メソッドで、キーが存在しなかった場合によびだされます。値を返すか例外を発生させるのどちらにしても、__getitem__() からもそのまま値が返るか例外が発生します。

なお、*__missing__()* は __getitem__() 以外のいかなる演算に対しても呼び出され **ません** 。よって get() は、普通の辞書と同様に、*default_factory* を使うのではなくデフォルトとして None を返します。

defaultdict オブジェクトは以下のインスタンス変数をサポートしています:

default_factory

この属性は *__missing__()* メソッドによって使われます。これは存在すればコンストラクタの第 1 引数によって初期化され、そうでなければ None になります。

defaultdict の使用例

list を *default_factory* とすることで、キー=値ペアのシーケンスをリストの辞書へ簡単にグループ化できます。:

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

それぞれのキーが最初に登場したとき、マッピングにはまだ存在しません。そのためエントリは *default_factory* 関数が返す空の *list* を使って自動的に作成されます。`list.append()` 操作は新しいリストに紐付けられます。キーが再度出現した場合には、通常の参照動作が行われます (そのキーに対応するリストが返ります)。そして `list.append()` 操作で別の値をリストに追加します。このテクニックは *dict.setdefault()* を使った等価なものよりシンプルで速いです:

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

default_factory を *int* にすると、*defaultdict* を (他の言語の bag や multiset のように) 要素の数え上げに便利に使うことができます:

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

最初に文字が出現したときは、マッピングが存在しないので *default_factory* 関数が *int()* を呼んでデフォルトのカウント 0 を生成します。インクリメント操作が各文字を数え上げます。

常に 0 を返す *int()* は特殊な関数でした。定数を生成するより速くて柔軟な方法は、0 に限らず何でも定数を生成するラムダ関数を使うことです:

```
>>> def constant_factory(value):
...     return lambda: value
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

default_factory を *set* に設定することで、*defaultdict* をセットの辞書を作るために利用することがで

きます:

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
...
>>> sorted(d.items())
[('blue', {2, 4}), ('red', {1, 3})]
```

8.3.5 namedtuple() 名前付きフィールドを持つタプルのファクトリ関数

名前付きタプルは、タプルの中のすべての場所に意味を割り当てて、より読みやすく自己解説的なコードを書けるようにします。通常のタプルが利用される場所ならどこでも利用でき、場所に対するインデックスの代わりに名前を使ってフィールドにアクセスできるようになります。

collections.**namedtuple**(*typename*, *field_names*, ***, *rename=False*, *defaults=None*, *module=None*)

typename という名前の tuple の新しいサブクラスを返します。新しいサブクラスは、tuple に似ているけれどもインデックスやイテレータだけでなく属性名によるアクセスもできるオブジェクトを作るのに使います。このサブクラスのインスタンスは、わかりやすい docstring (型名と属性名が入っています) や、tuple の内容を `name=value` という形のリストで返す使いやすい `__repr__()` も持っています。

field_names は `['x', 'y']` のような文字列のシーケンスです。*field_names* には、代わりに各属性名を空白文字 (whitespace) および/またはカンマ (,) で区切った文字列を渡すこともできます。例えば、`'x y'` や `'x, y'` です。

アンダースコア (_) で始まる名前を除いて、Python の正しい識別子 (identifier) ならなんでも属性名として使うことができます。正しい識別子とはアルファベット (letters), 数字 (digits), アンダースコア (_) を含みますが、数字やアンダースコアで始まる名前や、*class*, *for*, *return*, *global*, *pass*, *raise* などといった *keyword* は使えません。

rename が真の場合、不適切なフィールド名は自動的に位置を示す名前に置き換えられます。例えば `['abc', 'def', 'ghi', 'abc']` は、予約語の `def` と、重複しているフィールド名の `abc` が除去され、`['abc', '_1', 'ghi', '_3']` に変換されます。

defaults には `None` あるいはデフォルト値の *iterable* が指定できます。デフォルト値を持つフィールドはデフォルト値を持たないフィールドより後ろに来なければならないので、*defaults* は最も右にある変数に適用されます。例えば、field_names が `['x', 'y', 'z']` で defaults が `(1, 2)` の場合、`x` は必須の引数、`y` は `1` がデフォルト、`z` は `2` がデフォルトとなります。

もし *module* が指定されていれば、名前付きタプルの `__module__` 属性は、指定された値に設定されます

名前付きタプルのインスタンスはインスタンスごとの辞書を持たないので、軽量で、普通のタプル以上のメモリを使用しません。

バージョン 3.1 で変更: *rename* のサポートが追加されました。

バージョン 3.6 で変更: *verbose* と *rename* 引数が **キーワード専用引数** になりました.

バージョン 3.6 で変更: *module* 引数が追加されました。

バージョン 3.7 で変更: *verbose* 引数と `_source` 属性が削除されました。

バージョン 3.7 で変更: *defaults* 引数と `_field_defaults` 属性が追加されました。

```
>>> # Basic example
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22)     # instantiate with positional or keyword arguments
>>> p[0] + p[1]             # indexable like the plain tuple (11, 22)
33
>>> x, y = p                # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y               # fields also accessible by name
33
>>> p                       # readable __repr__ with a name=value style
Point(x=11, y=22)
```

名前付きタプルは *csv* や *sqlite3* モジュールが返すタプルのフィールドに名前を付けるときにとても便利です:

```
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, paygrade')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
    print(emp.name, emp.title)

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print(emp.name, emp.title)
```

タプルから継承したメソッドに加えて、名前付きタプルは 3 つの追加メソッドと 2 つの属性をサポートしています。フィールド名との衝突を避けるために、メソッド名と属性名はアンダースコアで始まります。

classmethod somenamedtuple.**_make**(*iterable*)

既存の sequence や Iterable から新しいインスタンスを作るクラスメソッド.

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

somenamedtuple.**_asdict**()

フィールド名を対応する値にマッピングする新しい *dict* を返します:

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
{'x': 11, 'y': 22}
```

バージョン 3.1 で変更: 通常の *dict* の代わりに *OrderedDict* を返すようになりました。

バージョン 3.8 で変更: *collections.OrderedDict* ではなく *dict* を返すようになりました。Python 3.7 以降は、通常の辞書で順番が保証されています。*OrderedDict* 特有の機能を使いたい場合は、結果を `OrderedDict(nt._asdict())` 型にキャストして使用することを推奨します。

somenamedtuple.**_replace**(***kwargs*)

指定されたフィールドを新しい値で置き換えた、新しい名前付きタプルを作って返します:

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)

>>> for partnum, record in inventory.items():
...     inventory[partnum] = record._replace(price=newprices[partnum], timestamp=time.
↪now())
```

somenamedtuple.**_fields**

フィールド名をリストにしたタプルです。内省 (introspection) したり、既存の名前付きタプルをもとに新しい名前つきタプルを作成する時に便利です。

```
>>> p._fields            # view the field names
('x', 'y')

>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

somenamedtuple.**_field_defaults**

フィールド名からデフォルト値への対応を持つ辞書です。

```
>>> Account = namedtuple('Account', ['type', 'balance'], defaults=[0])
>>> Account._field_defaults
{'balance': 0}
>>> Account('premium')
Account(type='premium', balance=0)
```

文字列に格納された名前を使って名前つきタプルから値を取得するには *getattr()* 関数を使います:

```
>>> getattr(p, 'x')
11
```

辞書を名前付きタプルに変換するには、** 演算子 (double-star-operator, tut-unpacking-arguments で説明しています) を使います。:

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

名前付きタプルは通常の Python クラスなので、継承して機能を追加したり変更するのは容易です。次の例では計算済みフィールドと固定幅の print format を追加しています:

```
>>> class Point(namedtuple('Point', ['x', 'y'])):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
...     def __str__(self):
...         return 'Point: x=%6.3f  y=%6.3f  hypot=%6.3f' % (self.x, self.y, self.hypot)

>>> for p in Point(3, 4), Point(14, 5/7):
...     print(p)
Point: x= 3.000  y= 4.000  hypot= 5.000
Point: x=14.000  y= 0.714  hypot=14.018
```

このサブクラスは `__slots__` に空のタプルをセットしています。これにより、インスタンス辞書の作成を抑制してメモリ使用量を低く保つのに役立ちます。

サブクラス化は新しいフィールドを追加するのには適していません。代わりに、新しい名前付きタプルを *_fields* 属性を元に作成してください:

```
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

`__doc__` フィールドに直接代入することでドックストリングをカスタマイズすることが出来ます:

```
>>> Book = namedtuple('Book', ['id', 'title', 'authors'])
>>> Book.__doc__ += ': Hardcover book in active collection'
>>> Book.id.__doc__ = '13-digit ISBN'
>>> Book.title.__doc__ = 'Title of first printing'
>>> Book.authors.__doc__ = 'List of authors sorted by last name'
```

バージョン 3.5 で変更: 属性ドックストリングが書き込み可能になりました。

参考:

- 名前付きタプルに型ヒントを追加する方法については、*typing.NamedTuple* を参照してください。`class` キーワードを使った洗練された記法も紹介されています:

  ```
  class Component(NamedTuple):
      part_number: int
      weight: float
      description: Optional[str] = None
  ```

- タプルではなく、辞書をもとにした変更可能な名前空間を作成するには *types.SimpleNamespace()* を参照してください。

- *dataclasses* モジュールは、生成される特殊メソッドをユーザー定義クラスに自動的に追加するためのデコレータや関数を提供しています。

8.3.6 OrderedDict オブジェクト

順序付き辞書は普通の辞書のようですが、順序操作に関係する追加の機能があります。組み込みの *dict* クラスが挿入順序を記憶しておく機能 (この新しい振る舞いは Python 3.7 で保証されるようになりました) を獲得した今となっては、順序付き辞書の重要性は薄れました。

いまだ残っている *dict* との差分:

- 通常の *dict* は対応付けに向いているように設計されました。挿入順序の追跡は二の次です。
- *OrderedDict* は並べ替え操作に向いているように設計されました。空間効率、反復処理の速度、更新操作のパフォーマンスは二の次です。
- アルゴリズム的に、*OrderedDict* は高頻度の並べ替え操作を *dict* よりも上手く扱えます。この性質により、*OrderedDict* は直近のアクセスの追跡 (例えば、LRU キャッシュ) に向いています。
- *OrderedDict* に対する等価演算は突き合わせ順序もチェックします。
- *OrderedDict* の popitem() メソッドはシグネチャが異なります。どの要素を取り出すかを指定するオプション引数を受け付けます。
- *OrderedDict* には、効率的に要素を末尾に置き直す move_to_end() メソッドがあります。
- Python 3.8 以前は、*dict* には __reversed__() メソッドが欠けています。

class collections.**OrderedDict**([*items*])

辞書の順序を並べ直すためのメソッドを持つ *dict* のサブクラスのインスタンスを返します。

バージョン 3.1 で追加.

popitem(*last=True*)

順序付き辞書の *popitem()* メソッドは、(key, value) 対を返して消去します。この対は *last* が真なら LIFO で、偽なら FIFO (first-in, first-out, 先入先出) で返されます。

move_to_end(*key*, *last=True*)

既存の *key* を順序付き辞書の両端に移動します。項目は、*last* が真 (デフォルト) なら右端に、*last* が偽なら最初に移動されます。*key* が存在しなければ *KeyError* を送出します:

```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d.keys())
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d.keys())
'bacde'
```

バージョン 3.2 で追加.

通常のマッピングのメソッドに加え、順序付き辞書は *reversed()* による逆順の反復もサポートしています。

OrderedDict 間の等価判定は順序が影響し、`list(od1.items())==list(od2.items())` のように実装されます。*OrderedDict* オブジェクトと他のマッピング (*Mapping*) オブジェクトの等価判定は、順序に影響されず、通常の辞書と同様です。これによって、*OrderedDict* オブジェクトは通常の辞書が使われるところならどこでも使用できます。

バージョン 3.5 で変更: *OrderedDict* の項目、キー、値の **ビュー** が *reversed()* による逆順の反復をサポートするようになりました。

バージョン 3.6 で変更: **PEP 468** の受理によって、*OrderedDict* のコンストラクタと、`update()` メソッドに渡したキーワード引数の順序は保持されます。

OrderedDict の例とレシピ

キーが **最後に** 追加されたときの順序を記憶する、順序付き辞書の変種を作るのは簡単です。新しい値が既存の値を上書きする場合、元々の挿入位置が最後尾へ変更されます:

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        self.move_to_end(key)
```

OrderedDict は *functools.lru_cache()* の変種を実装するのにも役に立ちます:

```
class LRU(OrderedDict):
    'Limit size, evicting the least recently looked-up key when full'

    def __init__(self, maxsize=128, /, *args, **kwds):
        self.maxsize = maxsize
        super().__init__(*args, **kwds)

    def __getitem__(self, key):
        value = super().__getitem__(key)
        self.move_to_end(key)
        return value

    def __setitem__(self, key, value):
        if key in self:
            self.move_to_end(key)
        super().__setitem__(key, value)
        if len(self) > self.maxsize:
            oldest = next(iter(self))
            del self[oldest]
```

8.3.7 UserDict オブジェクト

クラス *UserDict* は、辞書オブジェクトのラッパとしてはたらきます。このクラスの必要性は、*dict* から直接的にサブクラス化できる能力に部分的に取って代わられました; しかし、根底の辞書に属性としてアクセスできるので、このクラスを使った方が簡単になることもあります。

class collections.**UserDict**(*[initialdata]*)

辞書をシミュレートするクラスです。インスタンスの内容は通常の辞書に保存され、*UserDict* インスタンスの *data* 属性を通してアクセスできます。*initialdata* が与えられれば、*data* はその内容で初期化されます。他の目的のために使えるように、*initialdata* への参照が保存されないことがあるということに注意してください。

マッピングのメソッドと演算をサポートするのに加え、*UserDict* インスタンスは以下の属性を提供します:

data

UserDict クラスの内容を保存するために使われる実際の辞書です。

8.3.8 UserList オブジェクト

このクラスはリストオブジェクトのラッパとしてはたらきます。これは独自のリスト風クラスの基底クラスとして便利で、既存のメソッドをオーバーライドしたり新しいメソッドを加えたりできます。こうして、リストに新しい振る舞いを加えられます。

このクラスの必要性は、*list* から直接的にサブクラス化できる能力に部分的に取って代わられました; しかし、根底のリストに属性としてアクセスできるので、このクラスを使った方が簡単になることもあります。

class collections.**UserList**(*[list]*)

リストをシミュレートするクラスです。インスタンスの内容は通常のリストに保存され、*UserList* インスタンスの *data* 属性を通してアクセスできます。インスタンスの内容は最初に *list* のコピーに設定されますが、デフォルトでは空リスト `[]` です。*list* は何らかのイテラブル、例えば通常の Python リストや *UserList* オブジェクト、です。

ミュータブルシーケンスのメソッドと演算をサポートするのに加え、*UserList* インスタンスは以下の属性を提供します:

data

UserList クラスの内容を保存するために使われる実際の *list* オブジェクトです。

サブクラス化の要件: *UserList* のサブクラスは引数なしか、あるいは一つの引数のどちらかとともに呼び出せるコンストラクタを提供することが期待されています。新しいシーケンスを返すリスト演算は現在の実装クラスのインスタンスを作成しようとします。そのために、データ元として使われるシーケンスオブジェクトである一つのパラメータとともにコンストラクタを呼び出せると想定しています。

派生クラスがこの要求に従いたくないならば、このクラスがサポートしているすべての特殊メソッドはオーバーライドされる必要があります。その場合に提供される必要のあるメソッドについての情報は、ソースを参考にしてください。

8.3.9 UserString オブジェクト

クラス *UserString* は、文字列オブジェクトのラッパとしてはたらきます。このクラスの必要性は、*str* から直接的にサブクラス化できる能力に部分的に取って代わられました; しかし、根底の文字列に属性としてアクセスできるので、このクラスを使った方が簡単になることもあります。

class collections.**UserString**(*seq*)

文字列オブジェクトをシミュレートするクラスです。インスタンスの内容は通常の文字列に保存され、*UserString* インスタンスの *data* 属性を通してアクセスできます。インスタンスの内容には最初に *seq* のコピーが設定されます。*seq* 引数は、組み込みの *str()* 関数で文字列に変換できる任意のオブジェクトです。

文字列のメソッドと演算をサポートするのに加え、*UserString* インスタンスは次の属性を提供します:

data

UserString クラスの内容を保存するために使われる実際の *str* オブジェクトです。

バージョン 3.5 で変更: 新たなメソッド `__getnewargs__`, `__rmod__`, `casefold`, `format_map`, `isprintable`, `maketrans`。

8.4 collections.abc --- コレクションの抽象基底クラス

バージョン 3.3 で追加: 以前はこのモジュールは *collections* モジュールの一部でした。

ソースコード: Lib/_collections_abc.py

このモジュールは、**抽象基底クラス** を提供します。抽象基底クラスは、クラスが特定のインタフェースを提供しているか、例えばハッシュ可能であるかやマッピングであるかを判定します。

8.4.1 コレクション抽象基底クラス

collections モジュールは以下の *ABC (抽象基底クラス)* を提供します:

ABC	継承しているクラス	抽象メソッド	mixin メソッド
Container		__contains__	
Hashable		__hash__	
Iterable		__iter__	
Iterator	*Iterable*	__next__	__iter__
Reversible	*Iterable*	__reversed__	
Generator	*Iterator*	send, throw	close, __iter__, __next__
Sized		__len__	
Callable		__call__	
Collection	*Sized*, *Iterable*, *Container*	__contains__, __iter__, __len__	
Sequence	*Reversible*, *Collection*	__getitem__, __len__	__contains__, __iter__, __reversed__, index, count
MutableSequence	*Sequence*	__getitem__, __setitem__, __delitem__, __len__, insert	*Sequence* から継承したメソッドと、append, reverse, extend, pop, remove, __iadd__
ByteString	*Sequence*	__getitem__, __len__	*Sequence* から継承したメソッド
Set	*Collection*	__contains__, __iter__, __len__	__le__, __lt__, __eq__, __ne__, __gt__, __ge__, __and__, __or__, __sub__, __xor__, isdisjoint
MutableSet	*Set*	__contains__, __iter__, __len__, add, discard	*Set* から継承したメソッドと、clear, pop, remove, __ior__, __iand__, __ixor__, __isub__
Mapping	*Collection*	__getitem__, __iter__, __len__	__contains__, keys, items, values, get, __eq__, __ne__
MutableMapping	*Mapping*	__getitem__, __setitem__, __delitem__, __iter__, __len__	*Mapping* から継承したメソッドと、pop, popitem, clear, update, setdefault
MappingView	*Sized*		__len__
ItemsView	*MappingView*, *Set*		__contains__, __iter__
KeysView	*MappingView*, *Set*		__contains__, __iter__
ValuesView	*MappingView*, *Collection*		__contains__, __iter__
Awaitable		__await__	
Coroutine	*Awaitable*	send, throw	close
AsyncIterable		__aiter__	
AsyncIterator	*AsyncIterable*	__anext__	__aiter__
AsyncGenerator	*AsyncIterator*	asend, athrow	aclose, __aiter__, __anext__

class collections.abc.Container
__contains__() メソッドを提供するクラスの ABC です。

class collections.abc.Hashable
__hash__() メソッドを提供するクラスの ABC です。

class collections.abc.Sized
__len__() メソッドを提供するクラスの ABC です。

class collections.abc.Callable
__call__() メソッドを提供するクラスの ABC です。

class collections.abc.Iterable
__iter__() メソッドを提供するクラスの ABC です。

メソッド isinstance(obj, Iterable) で使用すると、*Iterable* や __iter__() メソッドを持っているクラスを検出できます。しかし、__getitem__() メソッドで反復するクラスは検出しません。オブジェクトが *iterable* であるかどうかを判別するにあたって、信頼できる唯一の方法は iter(obj) を呼び出す方法です。

class collections.abc.Collection
サイズ付きのイテラブルなコンテナクラスの ABC です。

バージョン 3.6 で追加.

class collections.abc.Iterator
__iter__() メソッドと *__next__()* メソッドを提供するクラスの ABC です。*iterator* の定義も参照してください。

class collections.abc.Reversible
__reversed__() メソッドを提供するイテラブルクラスの ABC です。

バージョン 3.6 で追加.

class collections.abc.Generator
PEP 342 で定義された、イテレータを send(), throw(), close() の各メソッドに拡張するプロトコルを実装する、ジェネレータクラスの ABC です。*generator* の定義も参照してください。

バージョン 3.5 で追加.

class collections.abc.Sequence
class collections.abc.MutableSequence
class collections.abc.ByteString
読み出し専用の **シーケンス** およびミュータブルな **シーケンス** の ABC です。

実装における注意: __iter__(), __reversed__(), index() など、一部の mixin メソッドは、下層の __getitem__() メソッドを繰り返し呼び出します。その結果、__getitem__() が定数のアクセス速度で実装されている場合、mixin メソッドは線形のパフォーマンスとなります。下層のメソッドが線形 (リンクされたリストの場合など) の場合、mixin は 2 乗のパフォーマンスとなるため、多くの場合上書きする必要があるでしょう。

バージョン 3.5 で変更: index() メソッドは *stop* と *start* 引数をサポートしました。

class collections.abc.**Set**

class collections.abc.**MutableSet**

読み出し専用の集合およびミュータブルな集合の ABC です。

class collections.abc.**Mapping**

class collections.abc.**MutableMapping**

読み出し専用の **マッピング** およびミュータブルな **マッピング** の ABC です。

class collections.abc.**MappingView**

class collections.abc.**ItemsView**

class collections.abc.**KeysView**

class collections.abc.**ValuesView**

マッピング、要素、キー、値の **ビュー** の ABC です。

class collections.abc.**Awaitable**

`await` で使用できる *awaitable* オブジェクトの ABC です。カスタムの実装は、`__await__()` メソッドを提供しなければなりません。

Coroutine ABC の *Coroutine* オブジェクトとインスタンスは、すべてこの ABC のインスタンスです。

注釈: CPython では、ジェネレータベースのコルーチン (*types.coroutine()* または *asyncio.coroutine()* でデコレートされたジェネレータ) は、`__await__()` メソッドを持ちませんが、待機可能 (*awaitables*) です。これらに対して `isinstance(gencoro, Awaitable)` を使用すると、`False` が返されます。これらを検出するには、*inspect.isawaitable()* を使用します。

バージョン 3.5 で追加.

class collections.abc.**Coroutine**

コルーチンと互換性のあるクラスの ABC です。これらは、coroutine-objects で定義された `send()`, `throw()`, `close()` のメソッドを実装します。カスタムの実装は、`__await__()` も実装しなければなりません。*Coroutine* のすべてのインスタンスは、*Awaitable* のインスタンスでもあります。*coroutine* の定義も参照してください。

注釈: CPython では、ジェネレータベースのコルーチン (*types.coroutine()* または *asyncio.coroutine()* でデコレートされたジェネレータ) は、`__await__()` メソッドを持ちませんが、待機可能 (*awaitables*) です。これらに対して `isinstance(gencoro, Coroutine)` を使用すると、`False` が返されます。これらを検出するには、*inspect.isawaitable()* を使用します。

バージョン 3.5 で追加.

class collections.abc.**AsyncIterable**

`__aiter__` メソッドを提供するクラスの ABC です。*asynchronous iterable* の定義も参照してくだ

さい。

バージョン 3.5 で追加.

class collections.abc.AsyncIterator

__aiter__ および __anext__ メソッドを提供するクラスの ABC です。*asynchronous iterator* の定義も参照してください。

バージョン 3.5 で追加.

class collections.abc.AsyncGenerator

PEP 525 と **PEP 492** に定義されているプロトコルを実装した非同期ジェネレータクラスの ABC です。

バージョン 3.6 で追加.

これらの ABC はクラスやインスタンスが特定の機能を提供しているかどうかを調べるのに使えます。例えば:

```
size = None
if isinstance(myvar, collections.abc.Sized):
    size = len(myvar)
```

幾つかの ABC はコンテナ型 API を提供するクラスを開発するのを助ける mixin 型としても使えます。例えば、*Set* API を提供するクラスを作る場合、3 つの基本になる抽象メソッド __contains__(), __iter__(), __len__() だけが必要です。ABC が残りの __and__() や isdisjoint() といったメソッドを提供します:

```
class ListBasedSet(collections.abc.Set):
    ''' Alternate set implementation favoring space over speed
        and not requiring the set elements to be hashable. '''
    def __init__(self, iterable):
        self.elements = lst = []
        for value in iterable:
            if value not in lst:
                lst.append(value)

    def __iter__(self):
        return iter(self.elements)

    def __contains__(self, value):
        return value in self.elements

    def __len__(self):
        return len(self.elements)

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2            # The __and__() method is supported automatically
```

Set と *MutableSet* を mixin 型として利用するときの注意点:

(1) 幾つかの set の操作は新しい set を作るので、デフォルトの mixin メソッドは iterable から新しいインスタンスを作成する方法を必要とします。クラスのコンストラクタは ClassName(iterable)

の形のシグネチャを持つと仮定されます。内部の `_from_iterable()` というクラスメソッドが `cls(iterable)` を呼び出して新しい set を作る部分でこの仮定が使われています。コンストラクタのシグネチャが異なるクラスで *Set* を使う場合は、iterable 引数から新しいインスタンスを生成するように `_from_iterable()` をオーバーライドする必要があります。

(2) (たぶん意味はそのままに速度を向上する目的で) 比較をオーバーライドする場合、`__le__()` と `__ge__()` だけを再定義すれば、その他の演算は自動的に追随します。

(3) *Set* mixin 型は set のハッシュ値を計算する `_hash()` メソッドを提供しますが、すべての set が hashable や immutable とは限らないので、`__hash__()` は提供しません。mixin を使ってハッシュ可能な set を作る場合は、*Set* と *Hashable* の両方を継承して、`__hash__ = Set._hash` と定義してください。

参考:

- *MutableSet* を使った例として OrderedSet recipe。
- ABCs についての詳細は、*abc* モジュールと **PEP 3119** を参照してください。

8.5 heapq --- ヒープキューアルゴリズム

ソースコード: Lib/heapq.py

このモジュールではヒープキューアルゴリズムの一実装を提供しています。優先度キューアルゴリズムとしても知られています。

ヒープとは、全ての親ノードの値が、その全ての子の値以下であるようなバイナリツリーです。この実装は、全ての k に対して、ゼロから要素を数えていった際に、`heap[k] <= heap[2*k+1]` かつ `heap[k] <= heap[2*k+2]` となる配列を使っています。比較のために、存在しない要素は無限大として扱われます。ヒープの興味深い性質は、最小の要素が常にルート、つまり `heap[0]` になることです。

以下の API は教科書におけるヒープアルゴリズムとは 2 つの側面で異なっています: (a) ゼロベースのインデクス化を行っています。これにより、ノードに対するインデクスとその子ノードのインデクスの関係がやや明瞭でなくなりますが、Python はゼロベースのインデクス化を使っているのでよりしっくりきます。(b) われわれの pop メソッドは最大の要素ではなく最小の要素 (教科書では "min heap:最小ヒープ" と呼ばれています; 教科書では並べ替えをインプレースで行うのに適した "max heap:最大ヒープ" が一般的です)。

これらの 2 点によって、ユーザに戸惑いを与えることなく、ヒープを通常の Python リストとして見ることができます: `heap[0]` が最小の要素となり、`heap.sort()` はヒープ不変式を保ちます!

ヒープを作成するには、`[]` に初期化されたリストを使うか、*heapify()* を用いて要素の入ったリストを変換します。

次の関数が用意されています:

heapq.heappush(*heap*, *item*)
item を *heap* に push します。ヒープ不変式を保ちます。

heapq.**heappop**(*heap*)

pop を行い、*heap* から最小の要素を返します。ヒープ不変式は保たれます。ヒープが空の場合、*IndexError* が送出されます。pop せずに最小の要素にアクセスするには、`heap[0]` を使ってください。

heapq.**heappushpop**(*heap*, *item*)

item を *heap* に push した後、pop を行って *heap* から最初の要素を返します。この一続きの動作を *heappush()* に引き続いて *heappop()* を別々に呼び出すよりも効率的に実行します。

heapq.**heapify**(*x*)

リスト *x* をインプレース処理し、線形時間でヒープに変換します。

heapq.**heapreplace**(*heap*, *item*)

heap から最小の要素を pop して返し、新たに *item* を push します。ヒープのサイズは変更されません。ヒープが空の場合、*IndexError* が送出されます。

この一息の演算は *heappop()* に次いで *heappush()* を送出するよりも効率的で、固定サイズのヒープを用いている場合にはより適しています。pop/push の組み合わせは必ずヒープから要素を一つ返し、それを *item* と置き換えます。

返される値は加えられた *item* よりも大きくなるかもしれません。それを望まないなら、代わりに *heappushpop()* を使うことを考えてください。この push/pop の組み合わせは二つの値の小さい方を返し、大きい方の値をヒープに残します。

このモジュールではさらに 3 つのヒープに基く汎用関数を提供します。

heapq.**merge**(**iterables*, *key=None*, *reverse=False*)

複数のソートされた入力をマージ (merge) して一つのソートされた出力にします (たとえば、複数のログファイルの時刻の入ったエントリーをマージします)。ソートされた値にわたる *iterator* を返します。

`sorted(itertools.chain(*iterables))` と似ていますが、イテレータを返し、一度にはデータをメモリに読み込まず、それぞれの入力ストリームが予め (最小から最大へ) ソートされていることを仮定します。

2 つのオプション引数があり、これらはキーワード引数として指定されなければなりません。

key は 1 つの引数からなる *key function* を指定します。この関数は、入力の各要素から比較のキーを取り出すのに使われます。デフォルト値は `None` です (要素を直接比較します)。

reverse は真偽値です。`True` を設定した場合、挿入要素は逆向きに比較されたかのように結合されます。`sorted(itertools.chain(*iterables), reverse=True)` でこれに似た挙動を実現するには、全てのイテラブルは降順で並んでいなければなりません。

バージョン 3.5 で変更: オプションの *key* 引数および *reverse* 引数を追加.

heapq.**nlargest**(*n*, *iterable*, *key=None*)

iterable で定義されるデータセットのうち、最大値から降順に *n* 個の値のリストを返します。(あたえられた場合) *key* は、引数を一つとる、*iterable* のそれぞれの要素から比較キーを生成する関数を指定します (例 `key=str.lower`)。以下のコードと同等です: `sorted(iterable, key=key, reverse=True)[:n]`

heapq.**nsmallest**(*n*, *iterable*, *key=None*)

iterable で定義されるデータセットのうち、最小値から昇順に *n* 個の値のリストを返します。(あたえられた場合) *key* は、引数を一つとる、*iterable* のそれぞれの要素から比較キーを生成する関数を指定します (例 `key=str.lower`)。以下のコードと同等です: `sorted(iterable, key=key)[:n]`

後ろ二つの関数は *n* の値が小さな場合に最適な動作をします。大きな値の時には *sorted()* 関数の方が効率的です。さらに、`n==1` の時には *min()* および *max()* 関数の方が効率的です。この関数を繰り返し使うことが必要なら、iterable を実際のヒープに変えることを考えてください。

8.5.1 基本的な例

すべての値をヒープに push してから最小値を 1 つずつ pop することで、ヒープソート を実装できます:

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

これは `sorted(iterable)` に似ていますが、*sorted()* とは異なり、この実装はステーブルソートではありません。

ヒープの要素はタプルに出来ます。これは、追跡される主レコードとは別に (タスクの優先度のような) 比較値を指定するときに便利です:

```
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

8.5.2 優先度キュー実装の注釈

優先度つきキュー は、ヒープの一般的な使い方で、実装にはいくつか困難な点があります:

- ソート安定性: 優先度が等しい二つのタスクが、もともと追加された順序で返されるためにはどうしたらいいでしょうか？
- (priority, task) ペアに対するタプルの比較は、priority が同じで task がデフォルトの比較順を持たないときに破綻します。
- あるタスクの優先度が変化したら、どうやってそれをヒープの新しい位置に移動させるのでしょうか？

- 未解決のタスクが削除される必要があるとき、どのようにそれをキューから探して削除するのでしょうか？

最初の二つの困難の解決策は、項目を優先度、項目番号、そしてタスクを含む 3 要素のリストとして保存することです。この項目番号は、同じ優先度の二つのタスクが、追加された順序で返されるようにするための同点決勝戦として働きます。そして二つの項目番号が等しくなることはありませんので、タプルの比較が二つのタスクを直接比べようとすることはありえません。

比較できないタスク問題のもう一つの解決法は、タスクアイテムを無視して優先順序フィールドだけで比較するラッパークラスです:

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

残りの困難は主に、未解決のタスクを探して、その優先度を変更したり、完全に削除することです。タスクを探すことは、キュー内の項目を指し示す辞書によってなされます。

項目を削除したり、優先度を変更することは、ヒープ構造の不変関係を壊すことになるので、もっと難しいです。ですから、可能な解決策は、その項目が無効であるものとしてマークし、必要なら変更された優先度の項目を加えることです:

```
pq = []                         # list of entries arranged in a heap
entry_finder = {}               # mapping of tasks to entries
REMOVED = '<removed-task>'      # placeholder for a removed task
counter = itertools.count()     # unique sequence count

def add_task(task, priority=0):
    'Add a new task or update the priority of an existing task'
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
    'Mark an existing task as REMOVED.  Raise KeyError if not found.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    'Remove and return the lowest priority task. Raise KeyError if empty.'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
```

(次のページに続く)

(前のページからの続き)

```
            return task
    raise KeyError('pop from an empty priority queue')
```

8.5.3 理論

ヒープとは、全ての k について、要素を 0 から数えたときに、`a[k] <= a[2*k+1]` かつ `a[k] <= a[2*k+2]` となる配列です。比較のために、存在しない要素を無限大と考えます。ヒープの興味深い属性は `a[0]` が常に最小の要素になることです。

上記の奇妙な不変式は、勝ち抜き戦判定の際に効率的なメモリ表現を行うためのものです。以下の番号は `a[k]` ではなく k とします:

```
                             0

              1                              2

      3               4               5               6

  7       8       9       10      11      12      13      14

15 16   17 18   19 20   21 22   23 24   25 26   27 28   29 30
```

上の木構造では、各セル k は `2*k+1` および `2*k+2` を最大値としています。スポーツに見られるような通常の 2 つ組勝ち抜き戦では、各セルはその下にある二つのセルに対する勝者となっていて、個々のセルの勝者を追跡していくことにより、そのセルに対する全ての相手を見ることができます。しかしながら、このような勝ち抜き戦を使う計算機アプリケーションの多くでは、勝歴を追跡する必要はりません。メモリ効率をより高めるために、勝者が上位に進級した際、下のレベルから持ってきて置き換えることにすると、あるセルとその下位にある二つのセルは異なる三つの要素を含み、かつ上位のセルは二つの下位のセルに対して ” 勝者と” なります。

このヒープ不変式が常に守られれば、インデクス 0 は明らかに最勝者となります。最勝者の要素を除去し、” 次の” 勝者を見つけるための最も単純なアルゴリズム的手法は、ある敗者要素 (ここでは上図のセル 30 とします) を 0 の場所に持っていき、この新しい 0 を濾過するようにしてツリーを下らせて値を交換してゆきます。不変関係が再構築されるまでこれを続けます。この操作は明らかに、ツリー内の全ての要素数に対して対数的な計算量となります。全ての要素について繰り返すと、O(n log n) のソート (並べ替え) になります。

このソートの良い点は、新たに挿入する要素が、最後に取り出された 0 番目の要素よりも ” 良い値” でない限り、ソートを行っている最中に新たな要素を効率的に追加できるというところです。この性質は、シミュレーション的な状況で、ツリーで全ての入力イベントを保持し、” 勝者” の状況を最小のスケジュール時刻にするような場合に特に便利です。あるイベントが他のイベント群の実行をスケジュールする際、それらは未来にスケジュールされることになるので、それらのイベント群を容易にヒープに積むことができます。すなわち、ヒープはスケジューラを実装する上で良いデータ構造であるといえます (私はこれを MIDI シーケンサで使っています :-)。

これまで、スケジューラを実装するための様々なデータ構造が広範に研究されてきました。ヒープは、十分高速で、速度はおおむね一定であり、最悪の場合でも平均的な速度とさほど変わらないため、良いデータ構造と

いえます。しかし、最悪の場合にひどい速度になるとしても、全体的にはより効率の高い他のデータ構造表現も存在します。

ヒープはまた、巨大なディスクのソートでも非常に有用です。おそらくご存知のように、巨大なソートを行うと、複数の ” ラン (run)” (予めソートされた配列で、そのサイズは通常 CPU メモリの量に関係しています) が生成され、続いて統合処理 (merging) がこれらのランを判定します。この統合処理はしばしば非常に巧妙に組織されています[*1]。重要なのは、最初のソートが可能な限り長いランを生成することです。勝ち抜き戦はこれを達成するための良い方法です。もし利用可能な全てのメモリを使って勝ち抜き戦を行い、要素を置換および濾過処理して現在のランに収めれば、ランダムな入力に対してメモリの二倍のサイズのランを生成することになり、大体順序づけがなされている入力に対してはもっと高い効率になります。

さらに、ディスク上の 0 番目の要素を出力して、現在の勝ち抜き戦に (最後に出力した値に ” 勝って” しまうために) 収められない入力を得たなら、ヒープには収まらないため、ヒープのサイズは減少します。解放されたメモリは二つ目のヒープを段階的に構築するために巧妙に再利用することができ、この二つ目のヒープは最初のヒープが崩壊していくのと同じ速度で成長します。最初のヒープが完全に消滅したら、ヒープを切り替えて新たなランを開始します。なんと巧妙で効率的なのでしょう！

一言で言うと、ヒープは知って得するメモリ構造です。私はいくつかのアプリケーションでヒープを使っていて、’ ヒープ’ モジュールを常備するのはいい事だと考えています。:-)

脚注

8.6 bisect --- 配列二分法アルゴリズム

ソースコード: Lib/bisect.py

このモジュールは、挿入の度にリストをソートすることなく、リストをソートされた順序に保つことをサポートします。大量の比較操作を伴うような、アイテムがたくさんあるリストでは、より一般的なアプローチに比べて、パフォーマンスが向上します。動作に基本的な二分法アルゴリズムを使っているので、*bisect* と呼ばれています。ソースコードはこのアルゴリズムの実例として一番役に立つかもしれません (境界条件はすでに正しいです!)。

次の関数が用意されています:

bisect.bisect_left(*a*, *x*, *lo=0*, *hi=len(a)*)

ソートされた順序を保ったまま x を a に挿入できる点を探し当てます。リストの中から検索する部分集合を指定するには、パラメータの *lo* と *hi* を使います。デフォルトでは、リスト全体が使われます。x がすでに a に含まれている場合、挿入点は既存のどのエントリーよりも前 (左) になります。戻り値は、`list.insert()` の第一引数として使うのに適しています。a はすでにソートされているものとします。

[*1] 現在使われているディスクバランス化アルゴリズムは、最近ではもはや巧妙というよりも目障りになっています。これは、ディスクのシーク機能が向上した結果です。巨大な容量を持つテープドライブなど、シーク不能なデバイスでは、事情は全く異なります。テープの 1 つ 1 つの動きが可能な限り効率的に行われるように非常に巧妙な処理を (相当前もって) 確保しなければなりません (統合処理の ” 進行” に最も多く使用させます)。テープによっては逆方向に読むことさえでき、巻き戻しに時間を取られるのを避けるために使うこともできます。正直、本当に良いテープソートは見ていて素晴らしく驚異的なものです！ ソートというのは常に偉大な芸術なのです！:-)

返された挿入点 *i* は、配列 *a* を二つに分け、`all(val < x for val in a[lo:i])` が左側に、`all(val >= x for val in a[i:hi])` が右側になるようにします。

bisect.bisect_right(*a*, *x*, *lo=0*, *hi=len(a)*)
bisect.bisect(*a*, *x*, *lo=0*, *hi=len(a)*)

bisect_left() と似ていますが、*a* に含まれる *x* のうち、どのエントリーよりも後ろ (右) にくるような挿入点を返します。

返された挿入点 *i* は、配列 *a* を二つに分け、`all(val <= x for val in a[lo:i])` が左側に、`all(val > x for val in a[i:hi])` が右側になるようにします。

bisect.insort_left(*a*, *x*, *lo=0*, *hi=len(a)*)

x を *a* にソート順で挿入します。これは、*a* がすでにソートされている場合、`a.insert(bisect.bisect_left(a, x, lo, hi), x)` と等価です。なお、O(log n) の探索に対して、遅い O(n) の挿入の段階が律速となります。

bisect.insort_right(*a*, *x*, *lo=0*, *hi=len(a)*)
bisect.insort(*a*, *x*, *lo=0*, *hi=len(a)*)

insort_left() と似ていますが、*a* に含まれる *x* のうち、どのエントリーよりも後ろに *x* を挿入します。

参考:

bisect を利用して、直接の探索ができ、キー関数をサポートする、完全な機能を持つコレクションクラスを組み立てる SortedCollection recipe。キーは、探索中に不必要な呼び出しをさせないために、予め計算しておきます。

8.6.1 ソート済みリストの探索

上記の *bisect()* 関数群は挿入点を探索するのには便利ですが、普通の探索タスクに使うのはトリッキーだったり不器用だったりします。以下の 5 関数は、これらをどのように標準の探索やソート済みリストに変換するかを説明します:

```
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
```

(次のページに続く)

(前のページからの続き)

```
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

def find_ge(a, x):
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
```

8.6.2 その他の使用例

bisect() 関数は数値テーブルの探索に役に立ちます。この例では、*bisect()* を使って、(たとえば) 順序のついた数値の区切り点の集合に基づいて、試験の成績の等級を表す文字を調べます。区切り点は 90 以上は 'A'、80 から 89 は 'B'、などです:

```
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
...     i = bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

sorted() 関数と違い、*bisect()* 関数に *key* や *reversed* 引数を用意するのは、設計が非効率になるので、非合理的です (連続する bisect 関数の呼び出しは前回の key 参照の結果を " 記憶" しません)。

代わりに、事前に計算しておいたキーのリストから検索して、レコードのインデックスを見つけます:

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])
>>> keys = [r[1] for r in data]         # precomputed list of keys
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```

8.7 array --- 効率のよい数値アレイ

このモジュールでは、基本的な値 (文字、整数、浮動小数点数) のアレイ (array、配列) をコンパクトに表現できるオブジェクト型を定義しています。アレイはシーケンス (sequence) 型であり、中に入れるオブジェクトの型に制限があることを除けば、リストとまったく同じように振る舞います。オブジェクト生成時に一文字の **型コード** を用いて型を指定します。次の型コードが定義されています:

型コード	C の型	Python の型	最小サイズ (バイト単位)	注釈
'b'	signed char	int	1	
'B'	unsigned char	int	1	
'u'	Py_UNICODE	Unicode 文字 (unicode 型)	2	(1)
'h'	signed short	int	2	
'H'	unsigned short	int	2	
'i'	signed int	int	2	
'I'	unsigned int	int	2	
'l'	signed long	int	4	
'L'	unsigned long	int	4	
'q'	signed long long	int	8	
'Q'	unsigned long long	int	8	
'f'	浮動小数点数	浮動小数点数	4	
'd'	double	浮動小数点数	8	

注釈:

(1) タイプコード 'u' は Python の古い Unicode 文字 (Py_UNICODE あるいは wchar_t) を表します。プラットフォームに依存して、これは 16bit か 32bit になります。

'u' は将来的に他の Py_UNICODE API と一緒に削除されるでしょう。

Deprecated since version 3.3, will be removed in version 4.0.

値の実際の表現はマシンアーキテクチャ (厳密に言うと C の実装) によって決まります。値の実際のサイズは itemsize 属性から得られます。

このモジュールでは次の型を定義しています:

class array.array(*typecode*[, *initializer*])

要素のデータ型が *typecode* に限定される新しいアレイで、オプションの値 *initializer* を渡すと初期値になりますが、リスト、*bytes-like object* または適当な型のイテレーション可能オブジェクトでなければなりません。

リストか文字列を渡した場合、initializer は新たに作成されたアレイの *fromlist()* 、*frombytes()* あるいは *fromunicode()* メソッド (以下を参照) に渡され、アレイに初期項目を追加します。それ以外の場合には、イテラブルの *initializer* は *extend()* メソッドに渡されます。

引数 typecode, initializer 付きで 監査イベント array.__new__ を送出します。

array.typecodes
すべての利用可能なタイプコードを含む文字列

アレイオブジェクトでは、インデクス指定、スライス、連結および反復といった、通常のシーケンスの演算をサポートしています。スライス代入を使うときは、代入値は同じ型コードのアレイオブジェクトでなければなりません。それ以外のオブジェクトを指定すると *TypeError* を送出します。アレイオブジェクトはバッファインタフェースを実装しており、*bytes-like objects* をサポートしている場所ならどこでも利用できます。

次のデータ要素やメソッドもサポートされています:

array.typecode
アレイを作るときに使う型コード文字です。

array.itemsize
アレイの要素 1 つの内部表現に使われるバイト長です。

array.append(*x*)
値 *x* の新たな要素をアレイの末尾に追加します。

array.buffer_info()
アレイの内容を記憶するために使っているバッファの、現在のメモリアドレスと要素数の入ったタプル (address, length) を返します。バイト単位で表したメモリバッファの大きさは array.buffer_info()[1] * array.itemsize で計算できます。例えば ioctl() 操作のような、メモリアドレスを必要とする低レベルな (そして、本質的に危険な) I/O インタフェースを使って作業する場合に、ときどき便利です。アレイ自体が存在し、長さを変えるような演算を適用しない限り、有効な値を返します。

注釈: C や C++ で書いたコードからアレイオブジェクトを使う場合 (buffer_info() の情報を使う意味のある唯一の方法です) は、アレイオブジェクトでサポートしているバッファインタフェースを使う方がより理にかなっています。このメソッドは後方互換性のために保守されており、新しいコードでの使用は避けるべきです。バッファインタフェースの説明は bufferobjects にあります。

---

array.byteswap()
アレイのすべての要素に対して「バイトスワップ」(リトルエンディアンとビッグエンディアンの変換) を行います。このメソッドは大きさが 1、2、4 および 8 バイトの値のみをサポートしています。他の種類の値に使うと *RuntimeError* を送出します。異なるバイトオーダを使うマシンで書かれたファイルからデータを読み込むときに役に立ちます。

array.count(*x*)
シーケンス中の *x* の出現回数を返します。

array.extend(*iterable*)
iterable から要素を取り出し、アレイの末尾に要素を追加します。*iterable* が別のアレイ型である場合、二つのアレイは **全く** 同じ型コードでなければなりません。それ以外の場合には *TypeError* を送出します。*iterable* がアレイでない場合、アレイに値を追加できるような正しい型の要素からなるイテレーション可能オブジェクトでなければなりません。

array.**frombytes**(*s*)
: 文字列から要素を追加します。文字列は、(ファイルから *fromfile()* メソッドを使って値を読み込んだときのように) マシンのデータ形式で表された値の配列として解釈されます。

  バージョン 3.2 で追加: 明確化のため *fromstring()* の名前が *frombytes()* に変更されました。

array.**fromfile**(*f*, *n*)
: ファイルオブジェクト *f* から (マシンのデータ形式そのままで) *n* 個の要素を読み出し、アレイの末尾に要素を追加します。*n* 個未満の要素しか読めなかった場合は *EOFError* を送出しますが、それまでに読み出せた値はアレイに追加されます。

array.**fromlist**(*list*)
: リストから要素を追加します。型に関するエラーが発生した場合にアレイが変更されないことを除き、`for x in list: a.append(x)` と同じです。

array.**fromstring**()
: *frombytes()* に対する廃止予定のエイリアス

  Deprecated since version 3.2, will be removed in version 3.9.

array.**fromunicode**(*s*)
: 指定した Unicode 文字列のデータを使ってアレイを拡張します。アレイの型コードは `'u'` でなければなりません。それ以外の場合には、*ValueError* を送出します。他の型のアレイに Unicode 型のデータを追加するには、`array.frombytes(unicodestring.encode(enc))` を使ってください。

array.**index**(*x*)
: アレイ中で *x* が出現するインデクスのうち最小の値 *i* を返します。

array.**insert**(*i*, *x*)
: アレイ中の位置 *i* の前に値 *x* をもつ新しい要素を挿入します。*i* の値が負の場合、アレイの末尾からの相対位置として扱います。

array.**pop**($[i]$)
: アレイからインデクスが *i* の要素を取り除いて返します。オプションの引数はデフォルトで `-1` になっていて、最後の要素を取り除いて返すようになっています。

array.**remove**(*x*)
: アレイ中の *x* のうち、最初に現れたものを取り除きます。

array.**reverse**()
: アレイの要素の順番を逆にします。

array.**tobytes**()
: array をマシンの値の array に変換して、bytes の形で返します (*tofile()* メソッドを使ってファイルに書かれるバイト列と同じです)。

  バージョン 3.2 で追加: 明確化のため *tostring()* の名前が *tobytes()* に変更されました。

array.**tofile**(*f*)
: すべての要素を (マシンの値の形式で) *file object* *f* に書き込みます。

array.tolist()
: アレイを同じ要素を持つ普通のリストに変換します。

array.tostring()
: *tobytes()* に対する廃止予定のエイリアス

  Deprecated since version 3.2, will be removed in version 3.9.

array.tounicode()
: アレイを Unicode 文字列に変換します。アレイの型コードは 'u' でなければなりません。それ以外の場合には *ValueError* を送出します。他の型のアレイから Unicode 文字列を得るには、array.tobytes().decode(enc) を使ってください。

アレイオブジェクトを表示したり文字列に変換したりすると、array(typecode, initializer) という形式で表現されます。アレイが空の場合、*initializer* の表示を省略します。アレイが空でなければ、*typecode* が 'u' の場合には文字列に、それ以外の場合には数値のリストになります。*array* クラスが from array import array というふうにインポートされている限り、変換後の文字列に *eval()* を用いると元のアレイオブジェクトと同じデータ型と値を持つアレイに逆変換できることが保証されています。文字列表現の例を以下に示します:

```
array('l')
array('u', 'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

参考:

struct **モジュール**　異なる種類のバイナリデータのパックおよびアンパック。

xdrlib **モジュール**　遠隔手続き呼び出しシステムで使われる外部データ表現仕様 (External Data Representation, XDR) のデータのパックおよびアンパック。

Numerical Python **ドキュメント**　Numeric Python 拡張モジュール (NumPy) では、別の方法でシーケンス型を定義しています。Numerical Python に関する詳しい情報は http://www.numpy.org/ を参照してください。

8.8 weakref --- 弱参照

ソースコード: Lib/weakref.py

---

weakref モジュールは、Python プログラマがオブジェクトへの弱参照 (*weak refarence*) を作成できるようにします。

以下では、用語リファレント (*referent*) は弱参照が参照するオブジェクトを意味します。

オブジェクトへの弱参照があることは、そのオブジェクトを生かしておくのには不十分です。リファレントへの参照が弱参照しか残っていない場合、*garbage collection* はリファレントを自由に破棄し、メモリを別のも

のに再利用することができます。しかし、オブジェクトへの強参照がなくても、オブジェクトが実際に破棄されるまでは、弱参照はオブジェクトを返す場合があります。

弱参照の主な用途は、巨大なオブジェクトを保持するキャッシュやマッピングを実装することです。ここで、キャッシュやマッピングに保持されているからという理由だけで、巨大なオブジェクトが生き続けることは望ましくありません。

例えば、巨大なバイナリ画像のオブジェクトがたくさんあり、それぞれに名前を関連付けたいとします。Python の辞書型を使って名前を画像に対応付けたり画像を名前に対応付けたりすると、画像オブジェクトは辞書内のキーや値に使われているため存続しつづけることになります。*weakref* モジュールが提供している *WeakKeyDictionary* や *WeakValueDictionary* クラスはその代用で、対応付けを構築するのに弱参照を使い、キャッシュやマッピングに存在するという理由だけでオブジェクトを存続させないようにします。例えば、もしある画像オブジェクトが *WeakValueDictionary* の値になっていた場合、最後に残った画像オブジェクトへの参照を弱参照マッピングが保持していれば、ガーベジコレクションはこのオブジェクトを再利用でき、画像オブジェクトに対する弱参照内の対応付けは削除されます。

WeakKeyDictionary と *WeakValueDictionary* はその実装に弱参照を使用しており、キーや値がガーベジコレクションによって回収されたことを弱参照辞書に通知するコールバック関数を設定しています。*WeakSet* は *set* インターフェースを実装していますが、*WeakKeyDictionary* のように要素への弱参照を持ちます。

finalize は、オブジェクトのガベージコレクションの実行時にクリーンアップ関数が呼び出されるように登録する、単純な方法を提供します。これは、未加工の弱参照上にコールバック関数を設定するよりも簡単です。なぜなら、オブジェクトのコレクションが完了するまでファイナライザが生き続けることを、モジュールが自動的に保証するからです。

ほとんどのプログラムでは弱参照コンテナまたは *finalize* のどれかを使えば必要なものは揃うはずです。通常は直接自前の弱参照を作成する必要はありません。低レベルな機構は、より進んだ使い方をするために *weakref* モジュールとして公開されています。

全てのオブジェクトが弱参照で参照できるわけではありません; 弱参照で参照できるのは、クラスインスタンス、(C ではなく) Python で書かれた関数、インスタンスメソッド、set オブジェクト、frozenset オブジェクト、*file* **オブジェクト** 、*generators* 型のオブジェクト、socket オブジェクト、array オジェクト、deque オブジェクト、正規表現パターンオブジェクト、code オブジェクトです。

バージョン 3.2 で変更: thread.lock, threading.Lock, code オブジェクトのサポートが追加されました。

list や *dict* など、いくつかの組み込み型は弱参照を直接サポートしませんが、以下のようにサブクラス化を行えばサポートを追加できます:

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)   # this object is weak referenceable
```

CPython implementation detail: *tuple* や *int* など、他の組み込み型はサブクラス化しても弱参照をサポートしません。

拡張型は、簡単に弱参照をサポートできます。詳細については、weakref-support を参照してください。

class weakref.ref(*object*[, *callback*])

object への弱参照を返します。リファレントがまだ生きているならば、元のオブジェクトは参照オブジェクトの呼び出しで取り出せす。リファレントがもはや生きていないならば、参照オブジェクトを呼び出したときに *None* を返します。*callback* に *None* 以外の値を与えた場合、オブジェクトをまさに後始末処理しようとするときに呼び出します。このとき弱参照オブジェクトは *callback* の唯一のパラメタとして渡されます。リファレントはもはや利用できません。

同じオブジェクトに対してたくさんの弱参照を作れます。それぞれの弱参照に対して登録されたコールバックは、もっとも新しく登録されたコールバックからもっとも古いものへと呼び出されます。

コールバックが発生させた例外は標準エラー出力に書き込まれますが、伝播されません。それらはオブジェクトの `__del__()` メソッドが発生させる例外と完全に同じ方法で処理されます。

object が **ハッシュ可能** ならば、弱参照はハッシュ可能です。それらは *object* が削除された後でもそれらのハッシュ値を保持します。*object* が削除されてから初めて *hash()* が呼び出された場合に、その呼び出しは *TypeError* を発生させます。

弱参照は等価性のテストをサポートしていますが、順序をサポートしていません。参照がまだ生きているならば、*callback* に関係なく二つの参照はそれらのリファレントと同じ等価関係を持ちます。リファレントのどちらか一方が削除された場合、参照オブジェクトが同一である場合に限り、その参照は等価です。

これはサブクラス化可能な型というよりファクトリ関数です。

`__callback__`

この読み出し専用の属性は、現在弱参照に関連付けられているコールバックを返します。コールバックが存在しないか、弱参照のリファレントが生きていない場合、この属性の値は `None` になります。

バージョン 3.4 で変更: *__callback__* 属性が追加されました。

weakref.proxy(*object*[, *callback*])

弱参照を使う *object* へのプロキシを返します。弱参照オブジェクトを明示的な参照外しをしながら利用する代わりに、多くのケースでプロキシを利用することができます。返されるオブジェクトは、*object* が呼び出し可能かどうかによって、`ProxyType` または `CallableProxyType` のどちらかの型を持ちます。プロキシオブジェクトはリファレントに関係なく **ハッシュ可能** ではありません。これによって、それらの基本的な変更可能という性質による多くの問題を避けています。そして、辞書のキーとしての利用を妨げます。*callback* は *ref()* 関数の同じ名前のパラメータと同じものです。(--- 訳注: リファレントが変更不能型であっても、プロキシはリファレントが消えるという状態の変更があるために、変更可能型です。---)

バージョン 3.8 で変更: Extended the operator support on proxy objects to include the matrix multiplication operators `@` and `@=`.

weakref.getweakrefcount(*object*)

object を参照する弱参照とプロキシの数を返します。

weakref.getweakrefs(*object*)

object を参照するすべての弱参照とプロキシオブジェクトのリストを返します。

class weakref.**WeakKeyDictionary**([*dict*])

キーを弱参照するマッピングクラス。キーへの強参照がなくなったときに、辞書のエントリは捨てられます。アプリケーションの他の部分が所有するオブジェクトへ属性を追加することもなく、それらのオブジェクトに追加データを関連づけるために使うことができます。これは属性へのアクセスをオーバーライドするオブジェクトに特に便利です。

WeakKeyDictionary オブジェクトは、追加のメソッドを持ちます。このメソッドは、内部の参照を直接公開します。その参照は、利用される時に生存しているとは限りません。なので、参照を利用する前に、その参照をチェックする必要があります。これにより、必要なくなったキーの参照が残っているために、ガベージコレクタがそのキーを削除できなくなる事態を避ける事ができます。

WeakKeyDictionary.**keyrefs**()

キーへの弱参照を持つ iterable オブジェクトを返します。

class weakref.**WeakValueDictionary**([*dict*])

値を弱参照するマッピングクラス。値への強参照が存在しなくなったときに、辞書のエントリは捨てられます。

WeakValueDictionary オブジェクトは *WeakKeyDictionary* オブジェクトの **keyrefs()** メソッドと同じ目的を持つ追加のメソッドを持っています。

WeakValueDictionary.**valuerefs**()

値への弱参照を持つ iterable オブジェクトを返します。

class weakref.**WeakSet**([*elements*])

要素への弱参照を持つ集合型。要素への強参照が無くなったときに、その要素は削除されます。

class weakref.**WeakMethod**(*method*)

拡張された *ref* のサブクラスで、束縛されたメソッドへの弱参照をシミュレートします (つまり、クラスで定義され、インスタンスにあるメソッド)。束縛されたメソッドは短命なので、標準の弱参照では保持し続けられません。*WeakMethod* には、オブジェクトと元々の関数が死ぬまで束縛されたメソッドを再作成する特別なコードがあります:

```
>>> class C:
...     def method(self):
...         print("method called!")
...
>>> c = C()
>>> r = weakref.ref(c.method)
>>> r()
>>> r = weakref.WeakMethod(c.method)
>>> r()
<bound method C.method of <__main__.C object at 0x7fc859830220>>
>>> r()()
method called!
>>> del c
>>> gc.collect()
0
>>> r()
>>>
```

バージョン 3.4 で追加.

class weakref.finalize(*obj*, *func*, **args*, ***kwargs*)

obj がガベージコレクションで回収されるときに呼び出される、呼び出し可能なファイナライザオブジェクトを返します。通常の弱参照とは異なり、ファイナライザは参照しているオブジェクトが回収されるまで必ず生き残り、そのおかげでライフサイクル管理が大いに簡単になります。

ファイナライザは (直接もしくはガベージコレクションのときに) 呼び出されるまで **生きている** と見なされ、呼び出された後には **死んでいます** 。生きているファイナライザを呼び出すと、`func(*arg, **kwargs)` を評価した結果を返します。一方、死んでいるファイナライザを呼び出すと *None* を返します。

ガベージコレクション中にファイナライザコールバックが発生させた例外は、標準エラー出力に表示されますが、伝播することはできません。これらの例外は、オブジェクトの `__del__()` メソッドや弱参照のコールバックが発生させる例外と同じ方法で処理されます。

プログラムが終了するとき、生き残ったそれぞれのファイナライザは、自身の *atexit* 属性が偽に設定されるまで呼び出され続けます。ファイナライザは生成された順序の逆順で呼び出されます。

A finalizer will never invoke its callback during the later part of the *interpreter shutdown* when module globals are liable to have been replaced by *None*.

`__call__()`

If *self* is alive then mark it as dead and return the result of calling `func(*args, **kwargs)`. If *self* is dead then return *None*.

`detach()`

If *self* is alive then mark it as dead and return the tuple `(obj, func, args, kwargs)`. If *self* is dead then return *None*.

`peek()`

If *self* is alive then return the tuple `(obj, func, args, kwargs)`. If *self* is dead then return *None*.

`alive`

ファイナライザが生きている場合には真、そうでない場合には偽のプロパティです。

`atexit`

A writable boolean property which by default is true. When the program exits, it calls all remaining live finalizers for which *atexit* is true. They are called in reverse order of creation.

注釈: It is important to ensure that *func*, *args* and *kwargs* do not own any references to *obj*, either directly or indirectly, since otherwise *obj* will never be garbage collected. In particular, *func* should not be a bound method of *obj*.

バージョン 3.4 で追加.

weakref.ReferenceType
弱参照オブジェクトのための型オブジェクト。

weakref.ProxyType
呼び出し可能でないオブジェクトのプロキシのための型オブジェクト。

weakref.CallableProxyType
呼び出し可能なオブジェクトのプロキシのための型オブジェクト。

weakref.ProxyTypes
プロキシのためのすべての型オブジェクトを含むシーケンス。これは両方のプロキシ型の名前付けに依存しないで、オブジェクトがプロキシかどうかのテストをより簡単にできます。

参考:

PEP 205 - 弱参照 この機能の提案と理論的根拠。初期の実装と他の言語における類似の機能についての情報へのリンクを含んでいます。

8.8.1 弱参照オブジェクト

Weak reference objects have no methods and no attributes besides *ref.__callback__*. A weak reference object allows the referent to be obtained, if it still exists, by calling it:

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

リファレントがもはや存在しないならば、参照オブジェクトの呼び出しは *None* を返します:

```
>>> del o, o2
>>> print(r())
None
```

弱参照オブジェクトがまだ生きているかどうかのテストは、式 `ref() is not None` を用いて行われます。通常、参照オブジェクトを使う必要があるアプリケーションコードはこのパターンに従います:

```
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print("Object has been deallocated; can't frobnicate.")
else:
    print("Object is still live!")
    o.do_something_useful()
```

” 生存性 (liveness)” のテストを分割すると、スレッド化されたアプリケーションにおいて競合状態を作り出します。(訳注:`if r() is not None: r().do_something()` では、2 度目の r() が None を返す可能性があります) 弱参照が呼び出される前に、他のスレッドは弱参照が無効になる原因となり得ます。上で示したイディオムは、シングルスレッドのアプリケーションと同じくマルチスレッド化されたアプリケーションにおいても安全です。

サブクラス化を行えば、*ref* オブジェクトの特殊なバージョンを作成できます。これは *WeakValueDictionary* の実装で使われており、マップ内の各エントリによるメモリのオーバヘッドを減らしています。こうした実装は、ある参照に追加情報を関連付けたい場合に便利ですし、リファレントを取り出すための呼び出し時に何らかの追加処理を行いたい場合にも使えます。

以下の例では、*ref* のサブクラスを使って、あるオブジェクトに追加情報を保存し、リファレントがアクセスされたときにその値に作用をできるようにするための方法を示しています:

```
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, /, **annotations):
        super().__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.items():
            setattr(self, k, v)

    def __call__(self):
        """Return a pair containing the referent and the number of
        times the reference has been called.
        """
        ob = super().__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```

8.8.2 使用例

この簡単な例では、アプリケーションが以前に参照したオブジェクトを取り出すためにオブジェクト ID を利用する方法を示します。オブジェクトに生きたままであることを強制することなく、オブジェクトの ID を他のデータ構造の中で使うことができ、必要に応じて ID からオブジェクトを取り出せます。

```
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid
```

(次のページに続く)

(前のページからの続き)

```
def id2obj(oid):
    return _id2obj_dict[oid]
```

8.8.3 ファイナライザオブジェクト

The main benefit of using *finalize* is that it makes it simple to register a callback without needing to preserve the returned finalizer object. For instance

```
>>> import weakref
>>> class Object:
...     pass
...
>>> kenny = Object()
>>> weakref.finalize(kenny, print, "You killed Kenny!")
<finalize object at ...; for 'Object' at ...>
>>> del kenny
You killed Kenny!
```

ファイナライザは直接呼び出すこともできます。ただし、ファイナライザはコールバックを最大でも一度しか呼び出しません。

```
>>> def callback(x, y, z):
...     print("CALLBACK")
...     return x + y + z
...
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> assert f.alive
>>> assert f() == 6
CALLBACK
>>> assert not f.alive
>>> f()                     # callback not called because finalizer dead
>>> del obj                 # callback not called because finalizer dead
```

You can unregister a finalizer using its *detach()* method. This kills the finalizer and returns the arguments passed to the constructor when it was created.

```
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> f.detach()
(<...Object object ...>, <function callback ...>, (1, 2), {'z': 3})
>>> newobj, func, args, kwargs = _
>>> assert not f.alive
>>> assert newobj is obj
>>> assert func(*args, **kwargs) == 6
CALLBACK
```

Unless you set the *atexit* attribute to *False*, a finalizer will be called when the program exits if it is

still alive. For instance

```
>>> obj = Object()
>>> weakref.finalize(obj, print, "obj dead or exiting")
<finalize object at ...; for 'Object' at ...>
>>> exit()
obj dead or exiting
```

8.8.4 ファイナライザと __del__() メソッドとの比較

インスタンスが一時ディレクトリを表す、クラスを作成するとします。そのディレクトリは、次のイベントのいずれかが起きた時に、そのディレクトリの内容とともに削除されるべきです。

- オブジェクトのガベージコレクションが行われた場合
- オブジェクトの remove() メソッドが呼び出された場合
- プログラムが終了した場合

ここでは、__del__() メソッドを使用して次のようにクラスを実装します:

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()

    def remove(self):
        if self.name is not None:
            shutil.rmtree(self.name)
            self.name = None

    @property
    def removed(self):
        return self.name is None

    def __del__(self):
        self.remove()
```

Starting with Python 3.4, __del__() methods no longer prevent reference cycles from being garbage collected, and module globals are no longer forced to *None* during *interpreter shutdown*. So this code should work without any issues on CPython.

However, handling of __del__() methods is notoriously implementation specific, since it depends on internal details of the interpreter's garbage collector implementation.

A more robust alternative can be to define a finalizer which only references the specific functions and objects that it needs, rather than having access to the full state of the object:

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()
```

(次のページに続く)

(前のページからの続き)

```
        self._finalizer = weakref.finalize(self, shutil.rmtree, self.name)

    def remove(self):
        self._finalizer()

    @property
    def removed(self):
        return not self._finalizer.alive
```

Defined like this, our finalizer only receives a reference to the details it needs to clean up the directory appropriately. If the object never gets garbage collected the finalizer will still be called at exit.

The other advantage of weakref based finalizers is that they can be used to register finalizers for classes where the definition is controlled by a third party, such as running code when a module is unloaded:

```
import weakref, sys
def unloading_module():
    # implicit reference to the module globals from the function body
weakref.finalize(sys.modules[__name__], unloading_module)
```

注釈: If you create a finalizer object in a daemonic thread just as the program exits then there is the possibility that the finalizer does not get called at exit. However, in a daemonic thread *atexit.register()*, `try: ... finally: ...` and `with: ...` do not guarantee that cleanup occurs either.

8.9 types --- 動的な型生成と組み込み型に対する名前

ソースコード: Lib/types.py

このモジュールは新しい型の動的な生成を支援するユーティリティ関数を定義します。

さらに、標準の Python インタプリタによって使用されているものの、*int* や *str* のように組み込みとして公開されていないようないくつかのオブジェクト型の名前を定義しています。

最後に、組み込みになるほど基本的でないような追加の型関連のユーティリティと関数をいくつか提供しています。

8.9.1 動的な型生成

types.**new_class**(*name*, *bases=()*, *kwds=None*, *exec_body=None*)

適切なメタクラスを使用して動的にクラスオブジェクトを生成します。

最初の 3 つの引数はクラス定義ヘッダーを構成する—クラス名、基底クラス (順番に)、キーワード引数 (例えば metaclass)—です。

exec_body 引数は、新規に生成されたクラスの名前空間を構築するために使用されるコールバックです。それは唯一の引数としてクラスの名前空間を受け取り、クラスの内容で名前空間を直接更新します。コールバックが渡されない場合、それは `lambda ns: None` を渡すことと同じ効果があります。

バージョン 3.3 で追加.

types.**prepare_class**(*name*, *bases=()*, *kwds=None*)

適切なメタクラスを計算してクラスの名前空間を生成します。

引数はクラス定義ヘッダーを構成する要素—クラス名、基底クラス (順番に)、キーワード引数 (例えば metaclass)—です。

返り値は `metaclass, namespace, kwds` の 3 要素のタプルです

metaclass は適切なメタクラスです。*namespace* は用意されたクラスの名前空間です。また *kwds* は、`'metaclass'` エントリが削除された、渡された *kwds* 引数の更新されたコピーです。*kwds* 引数が渡されなければ、これは空の dict になります。

バージョン 3.3 で追加.

バージョン 3.6 で変更: 返されるタプルの namespace 要素のデフォルト値が変更されました。現在では、メタクラスが `__prepare__` メソッドを持っていないときは、挿入順序を保存するマッピングが使われます。

参考:

metaclasses これらの関数によってサポートされるクラス生成プロセスの完全な詳細

PEP 3115 - Metaclasses in Python 3000 `__prepare__` 名前空間フックの導入

types.**resolve_bases**(*bases*)

Resolve MRO entries dynamically as specified by **PEP 560**.

This function looks for items in *bases* that are not instances of *type*, and returns a tuple where each such object that has an `__mro_entries__` method is replaced with an unpacked result of calling this method. If a *bases* item is an instance of *type*, or it doesn't have an `__mro_entries__` method, then it is included in the return tuple unchanged.

バージョン 3.7 で追加.

参考:

PEP 560 - typing モジュールとジェネリック型に対する言語コアによるサポート

8.9.2 標準的なインタプリタ型

このモジュールは、Python インタプリタを実装するために必要な多くの型に対して名前を提供します。それは、`listiterator` 型のような、単に処理中に付随的に発生するいくつかの型が含まれることを意図的に避けています。

これらの名前は典型的に *isinstance()* や *issubclass()* によるチェックに使われます。

これらのタイプのいずれかをインスタンス化する場合、シグネチャは Python のバージョンによって異なる可能性があることに注意してください。

以下の型に対して標準的な名前が定義されています:

types.FunctionType

types.LambdaType

ユーザ定義の関数と `lambda` 式によって生成された関数の型です。

引数 `code` を指定して **監査イベント** `function.__new__` を送出します。

この監査イベントは、関数オブジェクトを直接インスタンス化した場合にのみ発生し、通常のコンパイル時には発生しません。

types.GeneratorType

ジェネレータ関数によって生成された **ジェネレータ** イテレータオブジェクトの型です。

types.CoroutineType

`async def` 関数に生成される **コルーチン** オブジェクトです。

バージョン 3.5 で追加.

types.AsyncGeneratorType

非同期ジェネレータ関数によって作成された **非同期ジェネレータ** イテレータオブジェクトの型です。

バージョン 3.6 で追加.

class types.CodeType(***kwargs*)

compile() 関数が返すようなコードオブジェクトの型です。

引数 `code`, `filename`, `name`, `argcount`, `posonlyargcount`, `kwonlyargcount`, `nlocals`, `stacksize`, `flags` を指定して **監査イベント** `code.__new__` を送出します。

Note that the audited arguments may not match the names or positions required by the initializer. The audit event only occurs for direct instantiation of code objects, and is not raised for normal compilation.

replace(***kwargs*)

Return a copy of the code object with new values for the specified fields.

バージョン 3.8 で追加.

types.CellType

The type for cell objects: such objects are used as containers for a function's free variables.

バージョン 3.8 で追加.

types.MethodType
ユーザー定義のクラスのインスタンスのメソッドの型です。

types.BuiltinFunctionType
types.BuiltinMethodType
len() や *sys.exit()* のような組み込み関数や、組み込み型のメソッドの型です。(ここでは ”組み込み” という単語を ”C で書かれた” という意味で使っています)

types.WrapperDescriptorType
The type of methods of some built-in data types and base classes such as `object.__init__()` or `object.__lt__()`.

バージョン 3.7 で追加.

types.MethodWrapperType
The type of *bound* methods of some built-in data types and base classes. For example it is the type of `object().__str__`.

バージョン 3.7 で追加.

types.MethodDescriptorType
The type of methods of some built-in data types such as *str.join()*.

バージョン 3.7 で追加.

types.ClassMethodDescriptorType
The type of *unbound* class methods of some built-in data types such as `dict.__dict__['fromkeys']`.

バージョン 3.7 で追加.

class types.ModuleType(*name*, *doc=None*)
module の型です。コンストラクタは生成するモジュールの名前と任意でその *docstring* を受け取ります。

注釈: インポートによりコントロールされる様々な属性を設定する場合、*importlib.util.module_from_spec()* を使用して新しいモジュールを作成してください。

__doc__
モジュールの *docstring* です。デフォルトは None です。

__loader__
モジュールをロードする *loader* です。デフォルトは None です。

This attribute is to match *importlib.machinery.ModuleSpec.loader* as stored in the attr:*__spec__* object.

注釈: A future version of Python may stop setting this attribute by default. To guard against this potential change, preferrably read from the `__spec__` attribute instead or use `getattr(module, "__loader__", None)` if you explicitly need to use this attribute.

バージョン 3.4 で変更: デフォルトが `None` になりました。以前はオプションでした。

__name__
The name of the module. Expected to match *importlib.machinery.ModuleSpec.name*.

__package__
モジュールがどの *package* に属しているかです。モジュールがトップレベルである (すなわち、いかなる特定のパッケージの一部でもない) 場合、この属性は `''` に設定されます。そうでない場合、パッケージ名 (モジュールがパッケージ自身なら `__name__`) に設定されます。デフォルトは `None` です。

This attribute is to match *importlib.machinery.ModuleSpec.parent* as stored in the attr:*__spec__* object.

注釈: A future version of Python may stop setting this attribute by default. To guard against this potential change, preferrably read from the `__spec__` attribute instead or use `getattr(module, "__package__", None)` if you explicitly need to use this attribute.

バージョン 3.4 で変更: デフォルトが `None` になりました。以前はオプションでした。

__spec__
A record of the the module's import-system-related state. Expected to be an instance of *importlib.machinery.ModuleSpec*.

バージョン 3.4 で追加.

class types.TracebackType(*tb_next*, *tb_frame*, *tb_lasti*, *tb_lineno*)
`sys.exc_info()[2]` で得られるようなトレースバックオブジェクトの型です。

See the language reference for details of the available attributes and operations, and guidance on creating tracebacks dynamically.

types.FrameType
フレームオブジェクトの型です。トレースバックオブジェクト `tb` の `tb.tb_frame` などです。

See the language reference for details of the available attributes and operations.

types.GetSetDescriptorType
`FrameType.f_locals` や `array.array.typecode` のような、拡張モジュールにおいて `PyGetSetDef` によって定義されたオブジェクトの型です。この型はオブジェクト属性のディスクリプタとして利用されます。*property* 型と同じ目的を持った型ですが、こちらは拡張モジュールで定義された型のためのものです。

types.MemberDescriptorType

datetime.timedelta.days のような、拡張モジュールにおいて PyMemberDef によって定義されたオブジェクトの型です。この型は、標準の変換関数を利用するような、C のシンプルなデータメンバで利用されます。*property* 型と同じ目的を持った型ですが、こちらは拡張モジュールで定義された型のためのものです。

CPython implementation detail: Python の他の実装では、この型は GetSetDescriptorType と同じかもしれません。

class types.MappingProxyType(*mapping*)

読み出し専用のマッピングのプロキシです。マッピングのエントリーに関する動的なビューを提供します。つまり、マッピングが変わった場合にビューがこれらの変更を反映するということです。

バージョン 3.3 で追加.

key in proxy

元になったマッピングが *key* というキーを持っている場合 True を返します。そうでなければ False を返します。

proxy[key]

元になったマッピングの *key* というキーに対応するアイテムを返します。*key* が存在しなければ、*KeyError* が発生します。

iter(proxy)

元になったマッピングのキーを列挙するイテレータを返します。これは iter(proxy.keys()) のショートカットです。

len(proxy)

元になったマッピングに含まれるアイテムの数を返します。

copy()

元になったマッピングの浅いコピーを返します。

get(*key*[, *default*])

key が元になったマッピングに含まれている場合 *key* に対する値を返し、そうでなければ *default* を返します。もし *default* が与えられない場合は、デフォルト値の None になります。そのため、このメソッドが *KeyError* を発生させることはありません。

items()

元になったマッピングの items ((key, value) ペアの列) に対する新しいビューを返します。

keys()

元になったマッピングの keys に対する新しいビューを返します。

values()

元になったマッピングの values に対する新しいビューを返します。

8.9.3 追加のユーティリティクラスと関数

class types.SimpleNamespace

名前空間への属性アクセスに加えて意味のある repr を提供するための、単純な *object* サブクラスです。

object とは異なり、SimpleNamespace は、属性を追加したり削除したりすることができます。SimpleNamespace オブジェクトがキーワード引数で初期化される場合、それらは元になる名前空間に直接追加されます。

この型は以下のコードとほぼ等価です:

```
class SimpleNamespace:
    def __init__(self, /, **kwargs):
        self.__dict__.update(kwargs)

    def __repr__(self):
        keys = sorted(self.__dict__)
        items = ("{}={!r}".format(k, self.__dict__[k]) for k in keys)
        return "{}({})".format(type(self).__name__, ", ".join(items))

    def __eq__(self, other):
        if isinstance(self, SimpleNamespace) and isinstance(other, SimpleNamespace):
           return self.__dict__ == other.__dict__
        return NotImplemented
```

SimpleNamespace は class NS: pass を置き換えるものとして有用かもしれません。ですが、構造化されたレコード型に対しては、これよりはむしろ *namedtuple()* を使用してください。

バージョン 3.3 で追加.

types.DynamicClassAttribute(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

クラスの属性アクセスを ___getattr___ に振り替えます。

これは記述子で、インスタンス経由のアクセスとクラス経由のアクセスで振る舞いが異なる属性を定義するのに使います。インスタンスアクセスは通常通りですが、クラス経由の属性アクセスはクラスの ___getattr___ メソッドに振り替えられます。これは AttributeError の送出により行われます。

これによって、インスタンス上で有効なプロパティを持ち、クラス上で同名の仮想属性を持つことができます (例については *Enum* を参照してください)。

バージョン 3.4 で追加.

8.9.4 コルーチンユーティリティ関数

types.**coroutine**(*gen_func*)

この関数は、*generator* 関数を、ジェネレータベースのコルーチンを返す *coroutine function* に変換します。返されるジェネレータベースのコルーチンは依然として *generator iterator* ですが、同時に *coroutine* オブジェクトかつ *awaitable* であるとみなされます。ただし、必ずしも `__await__()` メソッドを実装する必要はありません。

gen_func はジェネレータ関数で、インプレースに変更されます。

gen_func がジェネレータ関数でない場合、この関数はラップされます。この関数が *collections.abc.Generator* のインスタンスを返す場合、このインスタンスは *awaitable* なプロキシオブジェクトにラップされます。それ以外のすべての型のオブジェクトは、そのまま返されます。

バージョン 3.5 で追加.

8.10 copy --- 浅いコピーおよび深いコピー操作

ソースコード: Lib/copy.py

Python において代入文はオブジェクトをコピーしません。代入はターゲットとオブジェクトの間に束縛を作ります。ミュータブルなコレクションまたはミュータブルなアイテムを含むコレクションについては、元のオブジェクトを変更せずにコピーを変更できるように、コピーが必要になることが時々あります。このモジュールは、汎用的な浅い (shallow) コピーと深い (deep) コピーの操作 (以下で説明されます) を提供します。

以下にインタフェースをまとめます:

copy.**copy**(*x*)

x の浅い (shallow) コピーを返します。

copy.**deepcopy**(*x*[, *memo*])

x の深い (deep) コピーを返します。

exception copy.**Error**

モジュール特有のエラーを送出します。

浅い (shallow) コピーと深い (deep) コピーの違いが関係するのは、複合オブジェクト (リストやクラスインスタンスのような他のオブジェクトを含むオブジェクト) だけです:

- **浅いコピー** *(shallow copy)* は新たな複合オブジェクトを作成し、その後 (可能な限り) 元のオブジェクト中に見つかったオブジェクトに対する **参照** を挿入します。
- **深いコピー** *(deep copy)* は新たな複合オブジェクトを作成し、その後元のオブジェクト中に見つかったオブジェクトの **コピー** を挿入します。

深いコピー操作には、しばしば浅いコピー操作の時には存在しない 2 つの問題がついてまわります:

- 再帰的なオブジェクト (直接、間接に関わらず、自分自身に対する参照を持つ複合オブジェクト) は再帰ループを引き起こします。
- 深いコピーは何もかもコピーしてしまうため、例えば複数のコピー間で共有するつもりだったデータも余分にコピーしてしまいます。

deepcopy() 関数では、これらの問題を以下のようにして回避しています:

- 現時点でのコピー過程ですでにコピーされたオブジェクトの memo 辞書を保持する。
- ユーザ定義のクラスでコピー操作やコピーされる内容の集合を上書きできるようにする。

このモジュールでは、モジュール、メソッド、スタックトレース、スタックフレーム、ファイル、ソケット、ウィンドウ、アレイ、その他これらに類似の型をコピーしません。このモジュールでは元のオブジェクトを変更せずに返すことで関数とクラスを (浅くまたは深く)「コピー」します。これは *pickle* モジュールでの扱われかたと同じです。

辞書型の浅いコピーは *dict.copy()* で、リストの浅いコピーはリスト全体を指すスライス (例えば `copied_list = original_list[:]`) でできます。

クラスは、コピーを制御するために pickle の制御に使用するのと同じインターフェースを使用することができます。これらのメソッドについての情報はモジュール *pickle* の説明を参照してください。実際、*copy* モジュールは、*copyreg* モジュールによって登録された pickle 関数を使用します。

クラス独自のコピー実装を定義するために、特殊メソッド `__copy__()` および `__deepcopy__()` を定義することができます。前者は浅いコピー操作を実装するために使われます; 追加の引数はありません。後者は深いコピー操作を実現するために呼び出されます; この関数には単一の引数として memo 辞書が渡されます。`__deepcopy__()` の実装で、内容のオブジェクトに対して深いコピーを生成する必要がある場合、*deepcopy()* を呼び出し、最初の引数にそのオブジェクトを、メモ辞書を二つ目の引数に与えなければなりません。

参考:

pickle **モジュール**　オブジェクト状態の取得と復元をサポートするために使われる特殊メソッドについて議論されています。

8.11 pprint --- データ出力の整然化

ソースコード: Lib/pprint.py

---

pprint モジュールを使うと、Python の任意のデータ構造をインタープリタへの入力で使われる形式にして "pretty-print" できます。書式化された構造の中に Python の基本的なタイプではないオブジェクトがあるなら、表示できないかもしれません。表示できないのは、ファイル、ソケット、あるいはクラスのようなオブジェクトや、その他 Python のリテラルとして表現できない様々なオブジェクトが含まれていた場合です。

可能であればオブジェクトを 1 行で整形しますが、与えられた幅に合わないなら複数行に分けて整形します。出力幅を指定したい場合は、*PrettyPrinter* オブジェクトを作成して明示してください。

辞書は表示される前にキーの順でソートされます。

pprint モジュールには 1 つのクラスが定義されています:

class pprint.PrettyPrinter(*indent=1*, *width=80*, *depth=None*, *stream=None*, ***, *compact=False*, *sort_dicts=True*)

PrettyPrinter インスタンスを作ります。このコンストラクタにはいくつかのキーワードパラメータを設定できます。*stream* キーワードで出力ストリームを設定できます；このストリームに対して呼び出されるメソッドはファイルプロトコルの `write()` メソッドだけです。もし設定されなければ、*PrettyPrinter* は `sys.stdout` を使用します。再帰的なレベルごとに加えるインデントの量は *indent* で設定できます；デフォルト値は 1 です。他の値にすると出力が少しおかしく見えますが、ネスト化されたところが見分け易くなります。出力されるレベルは *depth* で設定できます；出力されるデータ構造が深いなら、指定以上の深いレベルのものは `...` で置き換えられて表示されます。デフォルトでは、オブジェクトの深さを制限しません。*width* パラメータを使うと、出力する幅を望みの文字数に設定できます；デフォルトでは 80 文字です。もし指定した幅にフォーマットできない場合は、できるだけ近づけます。*compact* が偽ならば (これがデフォルトです)、長いシーケンスのアイテム一つずつが、一行ずつ分けてフォーマットされます。*compact* を真にすると、*width* 幅に収まるだけの量のアイテムがそれぞれの出力行にフォーマットされます。*sort_dicts* が真ならば (これがデフォルトです)、辞書はキーでソートされた状態にフォーマットされますが、そうでない場合は挿入順で表示されます。

バージョン 3.4 で変更: *compact* 引数が追加されました。

バージョン 3.8 で変更: *sort_dicts* 引数が追加されました。

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[   ['spam', 'eggs', 'lumberjack', 'knights', 'ni'],
    'spam',
    'eggs',
    'lumberjack',
    'knights',
    'ni']
>>> pp = pprint.PrettyPrinter(width=41, compact=True)
>>> pp.pprint(stuff)
[['spam', 'eggs', 'lumberjack',
  'knights', 'ni'],
 'spam', 'eggs', 'lumberjack', 'knights',
 'ni']
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
... ('parrot', ('fresh fruit',)))))))
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead', (...)))))))
```

pprint モジュールは幾つかのショートカット関数も提供しています:

pprint.pformat(*object*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*, *sort_dicts=True*)

object を書式化して文字列として返します。*indent*, *width*, *depth*, *compact*, *sort_dicts* は

PrettyPrinter コンストラクタに書式化引数として渡されます。

バージョン 3.4 で変更: *compact* 引数が追加されました。

バージョン 3.8 で変更: *sort_dicts* 引数が追加されました。

pprint.pp(*object*, **args*, *sort_dicts=False*, ***kwargs*)

object のフォーマットされた表現の後に改行を入れて表示します。もし *sort_dicts* が false (デフォルト) ならば、辞書はキーが挿入順に表示され、そうでなければ、辞書のキーがソートされます。*args* と *kwargs* はフォーマットパラメータとして *pprint()* に渡されます。

バージョン 3.8 で追加.

pprint.pprint(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*, *sort_dicts=True*)

stream 上に *object* の書式化された表現、続いて改行を出力します。*stream* が None の場合、sys.stdout が使用されます。これは、対話型インタプリタの中で値を調査するために *print()* 関数の代わりに使用されることがあります (さらに、スコープ内で使用するために print = pprint.pprint を再代入することができます)。*indent*, *width*, *depth*, *compact*, *sort_dicts* は、書式化引数として *PrettyPrinter* コンストラクタに渡されます。

バージョン 3.4 で変更: *compact* 引数が追加されました。

バージョン 3.8 で変更: *sort_dicts* 引数が追加されました。

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
[<Recursion on list with id=...>,
 'spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
```

pprint.isreadable(*object*)

object を書式化して出力できる ("readable") か、あるいは *eval()* を使って値を再構成できるかを返します。再帰的なオブジェクトに対しては常に False を返します。

```
>>> pprint.isreadable(stuff)
False
```

pprint.isrecursive(*object*)

object が再帰的な表現かどうかを返します。

さらにもう 1 つ、関数が定義されています:

pprint.saferepr(*object*)

object の文字列表現を、再帰的なデータ構造から保護した形式で返します。もし *object* の文字列表現

が再帰的な要素を持っているなら、再帰的な参照は <Recursion on typename with id=number> で表示されます。出力は他と違って書式化されません。

```
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs', 'lumberjack', 'knights', 'ni']"
```

8.11.1 PrettyPrinter オブジェクト

PrettyPrinter インスタンスには以下のメソッドがあります:

PrettyPrinter.**pformat**(*object*)
: *object* の書式化した表現を返します。これは *PrettyPrinter* のコンストラクタに渡されたオプションを考慮して書式化されます。

PrettyPrinter.**pprint**(*object*)
: *object* の書式化した表現を指定したストリームに出力し、最後に改行します。

以下のメソッドは、対応する同じ名前の関数と同じ機能を持っています。以下のメソッドをインスタンスに対して使うと、新たに *PrettyPrinter* オブジェクトを作る必要がないのでちょっぴり効果的です。

PrettyPrinter.**isreadable**(*object*)
: *object* を書式化して出力できる ("readable") か、あるいは *eval()* を使って値を再構成できるかを返します。これは再帰的なオブジェクトに対して False を返すことに注意して下さい。もし *PrettyPrinter* の *depth* 引数が設定されていて、オブジェクトのレベルが設定よりも深かったら、False を返します。

PrettyPrinter.**isrecursive**(*object*)
: オブジェクトが再帰的な表現かどうかを返します。

このメソッドをフックとして、サブクラスがオブジェクトを文字列に変換する方法を修正するのが可能になっています。デフォルトの実装では、内部で *saferepr()* を呼び出しています。

PrettyPrinter.**format**(*object*, *context*, *maxlevels*, *level*)
: 次の 3 つの値を返します。*object* をフォーマット化して文字列にしたもの、その結果が読み込み可能かどうかを示すフラグ、再帰が含まれているかどうかを示すフラグ。最初の引数は表示するオブジェクトです。2 つめの引数はオブジェクトの *id()* をキーとして含むディクショナリで、オブジェクトを含んでいる現在の（直接、間接に *object* のコンテナとして表示に影響を与える）環境です。ディクショナリ *context* の中でどのオブジェクトが表示されたか表示する必要があるなら、3 つめの返り値は True になります。*format()* メソッドの再帰呼び出しではこのディクショナリのコンテナに対してさらにエントリを加えます。3 つめの引数 *maxlevels* で再帰呼び出しのレベルを制限します。制限しない場合、0 になります。この引数は再帰呼び出しでそのまま渡されます。4 つめの引数 *level* で現在のレベルを設定します。再帰呼び出しでは、現在の呼び出しより小さい値が渡されます。

8.11.2 使用例

pprint() 関数のいくつかの用途とそのパラメータを実証するために、PyPI からプロジェクトに関する情報を取って来ましょう:

```
>>> import json
>>> import pprint
>>> from urllib.request import urlopen
>>> with urlopen('https://pypi.org/pypi/sampleproject/json') as resp:
...     project_info = json.load(resp)['info']
```

その基本形式では、*pprint()* はオブジェクト全体を表示します:

```
>>> pprint.pprint(project_info)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': ['Development Status :: 3 - Alpha',
                 'Intended Audience :: Developers',
                 'License :: OSI Approved :: MIT License',
                 'Programming Language :: Python :: 2',
                 'Programming Language :: Python :: 2.6',
                 'Programming Language :: Python :: 2.7',
                 'Programming Language :: Python :: 3',
                 'Programming Language :: Python :: 3.2',
                 'Programming Language :: Python :: 3.3',
                 'Programming Language :: Python :: 3.4',
                 'Topic :: Software Development :: Build Tools'],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {'last_day': -1, 'last_month': -1, 'last_week': -1},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
```

(次のページに続く)

(前のページからの続き)

```
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {'Download': 'UNKNOWN',
                  'Homepage': 'https://github.com/pypa/sampleproject'},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

結果をある深さ *depth* に制限することができます (より深い内容には省略記号が使用されます):

```
>>> pprint.pprint(project_info, depth=1)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
```

(次のページに続く)

(前のページからの続き)

```
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

それに加えて、最大の文字幅 *width* を指示することもできます。長いオブジェクトを分離することができなければ、指定された幅を超過します:

```
>>> pprint.pprint(project_info, depth=1, width=60)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the '
                'project.\n'
                '\n'
                'The file should use UTF-8 encoding and be '
                'written using ReStructured Text. It\n'
                'will be used to generate the project '
                'webpage on PyPI, and should be written '
                'for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would '
                'include an overview of the project, '
                'basic\n'
                'usage examples, etc. Generally, including '
                'the project changelog in here is not\n'
                'a good idea, although a simple "What\'s '
                'New" section for the most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
```

(次のページに続く)

(前のページからの続き)

```
'project_urls': {...},
'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
'requires_dist': None,
'requires_python': None,
'summary': 'A sample Python project',
'version': '1.2.0'}
```

8.12 reprlib --- もう一つの repr() の実装

ソースコード: Lib/reprlib.py

reprlib モジュールは、結果の文字列のサイズに対する制限付きでオブジェクト表現を生成するための手段を提供します。これは Python デバッガの中で使用されており、他の文脈でも同様に役に立つかもしれません。

このモジュールはクラスとインスタンス、それに関数を提供します:

class reprlib.Repr

組み込み関数 *repr()* に似た関数を実装するために役に立つフォーマット用サービスを提供します。過度に長い表現を作り出さないようにするための大きさの制限をオブジェクト型ごとに設定できます。

reprlib.aRepr

これは下で説明される *repr()* 関数を提供するために使われる *Repr* のインスタンスです。このオブジェクトの属性を変更すると、*repr()* と Python デバッガが使うサイズ制限に影響します。

reprlib.repr(*obj*)

これは aRepr の *repr()* メソッドです。同じ名前の組み込み関数が返す文字列と似ていますが、最大サイズに制限のある文字列を返します。

サイズを制限するツールに加えて、このモジュールはさらに __repr__() に対する再帰呼び出しの検出とプレースホルダー文字列による置換のためのデコレータを提供します。

@reprlib.recursive_repr(*fillvalue='...'*)

__repr__() メソッドに対する同一スレッド内の再帰呼び出しを検出するデコレータです。再帰呼び出しが行われている場合 *fillvalue* が返されます。そうでなければ通常の __repr__() 呼び出しが行われます。例えば:

```
>>> from reprlib import recursive_repr
>>> class MyList(list):
...     @recursive_repr()
...     def __repr__(self):
...         return '<' + '|'.join(map(repr, self)) + '>'
...
>>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

バージョン 3.2 で追加.

8.12.1 Repr オブジェクト

Repr インスタンスはオブジェクト型毎に表現する文字列のサイズを制限するために使えるいくつかの属性と、特定のオブジェクト型をフォーマットするメソッドを提供します。

Repr.maxlevel
再帰的な表現を作る場合の深さ制限。デフォルトは 6 です。

Repr.maxdict
Repr.maxlist
Repr.maxtuple
Repr.maxset
Repr.maxfrozenset
Repr.maxdeque
Repr.maxarray
指定されたオブジェクト型に対するエントリ表現の数についての制限。*maxdict* に対するデフォルトは 4 で、*maxarray* は 5 、その他に対しては 6 です。

Repr.maxlong
整数の表現のおける文字数の最大値。中央の数字が抜け落ちます。デフォルトは 40 です。

Repr.maxstring
文字列の表現における文字数の制限。文字列の" 通常の" 表現は文字の「元」として使われることに注意してください。表現にエスケープシーケンスが必要とされる場合、表現が短縮されるときにこれらのエスケープシーケンスの形式は崩れます。デフォルトは 30 です。

Repr.maxother
この制限は *Repr* オブジェクトに利用できる特定のフォーマットメソッドがないオブジェクト型のサイズをコントロールするために使われます。*maxstring* と同じようなやり方で適用されます。デフォルトは 20 です。

Repr.repr(*obj*)
このインスタンスで設定されたフォーマットを使う、組み込み *repr()* と等価なもの。

Repr.repr1(*obj*, *level*)
repr() が使う再帰的な実装。*obj* の型を使ってどのフォーマットメソッドを呼び出すかを決定し、それに *obj* と *level* を渡します。再帰呼び出しにおいて *level* の値に対して `level - 1` を与える再帰的なフォーマットを実行するために、型に固有のメソッドは *repr1()* を呼び出します。

Repr.repr_TYPE(*obj*, *level*)
型名に基づく名前をもつメソッドとして、特定の型に対するフォーマットメソッドは実装されます。メソッド名では、**TYPE** は `'_'.join(type(obj).__name__.split())` に置き換えられます。これらのメソッドへのディスパッチは *repr1()* によって処理されます。再帰的に値をフォーマットする必要がある型固有のメソッドは、`self.repr1(subobj, level - 1)` を呼び出します。

8.12.2 Repr オブジェクトをサブクラス化する

更なる組み込みオブジェクト型へのサポートを追加するため、あるいはすでにサポートされている型の扱いを変更するために、*Repr.repr1()* による動的なディスパッチは *Repr* のサブクラス化に対応しています。この例はファイルオブジェクトのための特別なサポートを追加する方法を示しています:

```
import reprlib
import sys

class MyRepr(reprlib.Repr):

    def repr_TextIOWrapper(self, obj, level):
        if obj.name in {'<stdin>', '<stdout>', '<stderr>'}:
            return obj.name
        return repr(obj)

aRepr = MyRepr()
print(aRepr.repr(sys.stdin))         # prints '<stdin>'
```

8.13 enum --- 列挙型のサポート

バージョン 3.4 で追加.

ソースコード: Lib/enum.py

列挙型は、一意の定数値に束縛された識別名 (メンバー) の集合です。列挙型の中でメンバーの同一性を比較でき、列挙型自身でイテレートが可能です。

注釈: Enum メンバーは大文字/小文字？

Enum クラス群は定数を表現するために使われるため、列挙型のメンバーの名前には UPPER_CASE を使うことを推奨します。本ページのドキュメントのサンプルでもそのスタイルを採用します。

8.13.1 モジュールコンテンツ

このモジュールでは一意の名前と値の集合を定義するのに使用できる 4 つの列挙型クラス *Enum*, *IntEnum*, *Flag*, *IntFlag* を定義しています。このモジュールはデコレータの *unique()* とヘルパークラスの *auto* も定義しています。

class enum.**Enum**

列挙型定数を作成する基底クラスです。もうひとつの構築構文については 機能 *API* を参照してください。

class enum.IntEnum

int のサブクラスでもある列挙型定数を作成する基底クラスです。

class enum.IntFlag

列挙型定数を作成する基底クラスで、ビット演算子を使って組み合わせられ、その結果も *IntFlag* メンバーになります。*IntFlag* は *int* のサブクラスでもあります。

class enum.Flag

列挙型定数を作成する基底クラスで、ビット演算を使って組み合わせられ、その結果も *IntFlag* メンバーになります。

enum.unique()

一つの名前だけがひとつの値に束縛されていることを保証する Enum クラスのデコレーターです。

class enum.auto

インスタンスはそれぞれ、適切な値で置き換えられます。値は、1 からはじまります。

バージョン 3.6 で追加: Flag, IntFlag, auto

8.13.2 Enum の作成

列挙型は読み書きが容易になるよう class 文を使って作成します。もうひとつの作成方法は **機能 *API*** で説明しています。列挙型は以下のように *Enum* のサブクラスとして定義します:

```
>>> from enum import Enum
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...
```

注釈: 列挙型のメンバー値

メンバー値は何であっても構いません: *int*, *str* などなど。正確な値が重要でない場合は、*auto* インスタンスを使っておくと、適切な値が選ばれます。*auto* とそれ以外の値を混ぜて使う場合は注意する必要があります。

注釈: 用語

- クラス Color は **列挙型** (または *Enum*) です
- 属性 Color.RED, Color.GREEN などは **列挙型のメンバー** (または *Enum* **メンバー**) で、機能的には定数です。
- 列挙型のメンバーは **名前** と **値** を持ちます (Color.RED の名前は RED 、Color.BLUE の値は 3 など。)

注釈: Enum の作成に class 文を使用するものの、Enum は通常の Python クラスではありません。詳細は *Enum* **はどう違うのか?** を参照してください。

列挙型のメンバーは人が読める文字列表現を持ちます:

```
>>> print(Color.RED)
Color.RED
```

... その一方でそれらの repr はより多くの情報を持っています:

```
>>> print(repr(Color.RED))
<Color.RED: 1>
```

列挙型メンバーの **データ型** はそれが所属する列挙型になります:

```
>>> type(Color.RED)
<enum 'Color'>
>>> isinstance(Color.GREEN, Color)
True
>>>
```

Enum メンバーは自身の名前を持つだけのプロパティも持っています:

```
>>> print(Color.RED.name)
RED
```

列挙型は定義順でのイテレーションをサポートしています:

```
>>> class Shake(Enum):
...     VANILLA = 7
...     CHOCOLATE = 4
...     COOKIES = 9
...     MINT = 3
...
>>> for shake in Shake:
...     print(shake)
...
Shake.VANILLA
Shake.CHOCOLATE
Shake.COOKIES
Shake.MINT
```

列挙型のメンバーはハッシュ化可能なため、辞書や集合で使用できます:

```
>>> apples = {}
>>> apples[Color.RED] = 'red delicious'
>>> apples[Color.GREEN] = 'granny smith'
>>> apples == {Color.RED: 'red delicious', Color.GREEN: 'granny smith'}
True
```

8.13.3 列挙型メンバーおよびそれらの属性へのプログラム的アクセス

プログラム的にメンバーに番号でアクセスしたほうが便利な場合があります (すなわち、プログラムを書いている時点で正確な色がまだわからなく、Color.RED と書くのが無理な場合など)。Enum ではそのようなアクセスも可能です:

```
>>> Color(1)
<Color.RED: 1>
>>> Color(3)
<Color.BLUE: 3>
```

列挙型メンバーに **名前** でアクセスしたい場合はアイテムとしてアクセスできます:

```
>>> Color['RED']
<Color.RED: 1>
>>> Color['GREEN']
<Color.GREEN: 2>
```

列挙型メンバーの name か value が必要な場合:

```
>>> member = Color.RED
>>> member.name
'RED'
>>> member.value
1
```

8.13.4 列挙型メンバーと値の重複

同じ名前の列挙型メンバーを複数持つことはできません:

```
>>> class Shape(Enum):
...     SQUARE = 2
...     SQUARE = 3
...
Traceback (most recent call last):
...
TypeError: Attempted to reuse key: 'SQUARE'
```

ただし、複数の列挙型メンバーが同じ値を持つことはできます。同じ値を持つ 2 つのメンバー A および B (先に定義したのは A) が与えられたとき、B は A の別名になります。A および B を値で調べたとき、A が返されます。B を名前で調べたとき、A が返されます:

```
>>> class Shape(Enum):
...     SQUARE = 2
...     DIAMOND = 1
...     CIRCLE = 3
...     ALIAS_FOR_SQUARE = 2
...
```

(次のページに続く)

(前のページからの続き)

```
>>> Shape.SQUARE
<Shape.SQUARE: 2>
>>> Shape.ALIAS_FOR_SQUARE
<Shape.SQUARE: 2>
>>> Shape(2)
<Shape.SQUARE: 2>
```

注釈: すでに定義されている属性と同じ名前のメンバー (一方がメンバーでもう一方がメソッド、など) の作成、あるいはメンバーと同じ名前の属性の作成はできません。

8.13.5 番号付けの値が同一であることの確認

デフォルトでは、前述のように複数の名前への同じ値の定義は別名とすることで許されています。この挙動を望まない場合、以下のデコレーターを使用することで各値が列挙型内で一意かどうか確認できます:

@enum.unique

列挙型専用の `class` デコレーターです。列挙型の `__members__` に別名がないかどうか検索します; 見つかった場合、*ValueError* が詳細情報とともに送出されます:

```
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...     FOUR = 3
...
Traceback (most recent call last):
...
ValueError: duplicate values found in <enum 'Mistake'>: FOUR -> THREE
```

8.13.6 値の自動設定を使う

正確な値が重要でない場合、*auto* が使えます:

```
>>> from enum import Enum, auto
>>> class Color(Enum):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

その値は _generate_next_value_() によって選ばれ、この関数はオーバーライドできます:

```
>>> class AutoName(Enum):
...     def _generate_next_value_(name, start, count, last_values):
...         return name
...
>>> class Ordinal(AutoName):
...     NORTH = auto()
...     SOUTH = auto()
...     EAST = auto()
...     WEST = auto()
...
>>> list(Ordinal)
[<Ordinal.NORTH: 'NORTH'>, <Ordinal.SOUTH: 'SOUTH'>, <Ordinal.EAST: 'EAST'>, <Ordinal.WEST:
↪'WEST'>]
```

注釈: デフォルトの _generate_next_value_() メソッドの目的は、最後に提供した *int* の次から順々に *int* を提供することですが、この動作は実装詳細であり変更される可能性があります。

注釈: _generate_next_value_() メソッドは他のメンバーよりも前に定義される必要があります。

8.13.7 イテレーション

列挙型のメンバーのイテレートは別名をサポートしていません:

```
>>> list(Shape)
[<Shape.SQUARE: 2>, <Shape.DIAMOND: 1>, <Shape.CIRCLE: 3>]
```

特殊属性 __members__ は読み出し専用で、順序を保持した、対応する名前と列挙型メンバーのマッピングです。これには別名も含め、列挙されたすべての名前が入っています。

```
>>> for name, member in Shape.__members__.items():
...     name, member
...
('SQUARE', <Shape.SQUARE: 2>)
('DIAMOND', <Shape.DIAMOND: 1>)
('CIRCLE', <Shape.CIRCLE: 3>)
('ALIAS_FOR_SQUARE', <Shape.SQUARE: 2>)
```

属性 __members__ は列挙型メンバーへの詳細なアクセスに使用できます。以下はすべての別名を探す例です:

```
>>> [name for name, member in Shape.__members__.items() if member.name != name]
['ALIAS_FOR_SQUARE']
```

8.13.8 比較

列挙型メンバーは同一性を比較できます:

```
>>> Color.RED is Color.RED
True
>>> Color.RED is Color.BLUE
False
>>> Color.RED is not Color.BLUE
True
```

列挙型の値の順序の比較はサポートされて **いません**。Enum メンバーは整数ではありません (*IntEnum* を参照してください):

```
>>> Color.RED < Color.BLUE
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'Color' and 'Color'
```

ただし等価の比較は定義されています:

```
>>> Color.BLUE == Color.RED
False
>>> Color.BLUE != Color.RED
True
>>> Color.BLUE == Color.BLUE
True
```

非列挙型の値との比較は常に不等となります (繰り返しになりますが、*IntEnum* はこれと異なる挙動になるよう設計されています):

```
>>> Color.BLUE == 2
False
```

8.13.9 列挙型で許されるメンバーと属性

上述の例では列挙型の値に整数を使用しています。整数の使用は短くて使いやすい (そして **機能** *API* でデフォルトで提供されています) のですが、厳密には強制ではありません。ほとんどの事例では列挙型の実際の値が何かを気にしていません。しかし、値が重要で **ある** 場合、列挙型は任意の値を持つことができます。

列挙型は Python のクラスであり、通常どおりメソッドや特殊メソッドを持つことができます:

```
>>> class Mood(Enum):
...     FUNKY = 1
...     HAPPY = 3
...
...     def describe(self):
...         # self is the member here
```

(次のページに続く)

(前のページからの続き)

```
...         return self.name, self.value
...
...     def __str__(self):
...         return 'my custom str! {0}'.format(self.value)
...
...     @classmethod
...     def favorite_mood(cls):
...         # cls here is the enumeration
...         return cls.HAPPY
...
```

上記の結果が以下のようになります:

```
>>> Mood.favorite_mood()
<Mood.HAPPY: 3>
>>> Mood.HAPPY.describe()
('HAPPY', 3)
>>> str(Mood.FUNKY)
'my custom str! 1'
```

何が許されているかのルールは次のとおりです。先頭と末尾が 1 個のアンダースコアの名前は列挙型により予約されているため、使用できません。列挙型内で定義されたその他すべての名前は、その列挙型のメンバーとして使用できます。特殊メソッド (`__str__()`, `__add__()` など) と、メソッドを含むデスクリプタ (記述子)、`_ignore_` に記載されている変数名は例外です。

注意: 列挙型で `__new__()` および/または `__init__()` を定義した場合、列挙型メンバーに与えられた値はすべてこれらのメソッドに渡されます。例 *Planet* を参照してください。

8.13.10 Enum のサブクラス化の制限

新しい *Enum* クラスは、ベースの Enum クラスを 1 つ、具象データ型を 1 つ、複数の *object* ベースのミックスインクラスが許容されます。これらのベースクラスの順序は次の通りです:

```
class EnumName([mix-in, ...,] [data-type,] base-enum):
    pass
```

列挙型のサブクラスの作成はその列挙型にメンバーが一つも定義されていない場合のみ行なえます。従って以下は許されません:

```
>>> class MoreColor(Color):
...     PINK = 17
...
Traceback (most recent call last):
...
TypeError: Cannot extend enumerations
```

以下のような場合は許されます:

```
>>> class Foo(Enum):
...     def some_behavior(self):
...         pass
...
>>> class Bar(Foo):
...     HAPPY = 1
...     SAD = 2
...
```

メンバーが定義された列挙型のサブクラス化を許可すると、いくつかのデータ型およびインスタンスの重要な不変条件の違反を引き起こします。とはいえ、それが許可されると、列挙型のグループ間での共通の挙動を共有するという利点もあります。(*OrderedEnum* の例を参照してください。)

8.13.11 Pickle 化

列挙型は pickle 化と unpickle 化が行えます:

```
>>> from test.test_enum import Fruit
>>> from pickle import dumps, loads
>>> Fruit.TOMATO is loads(dumps(Fruit.TOMATO))
True
```

通常の pickle 化の制限事項が適用されます: pickle 可能な列挙型はモジュールのトップレベルで定義されていなくてはならず、unpickle 化はモジュールからインポート可能でなければなりません。

注釈: pickle プロトコルバージョン 4 では他のクラスで入れ子になった列挙型の pickle 化も容易です。

Enum メンバーをどう pickle 化/unpickle 化するかは、列挙型クラス内の *__reduce_ex__()* で定義することで変更できます。

8.13.12 機能 API

Enum クラスは呼び出し可能で、以下の機能 API を提供しています:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG')
>>> Animal
<enum 'Animal'>
>>> Animal.ANT
<Animal.ANT: 1>
>>> Animal.ANT.value
1
>>> list(Animal)
[<Animal.ANT: 1>, <Animal.BEE: 2>, <Animal.CAT: 3>, <Animal.DOG: 4>]
```

この API の動作は *namedtuple* と似ています。*Enum* 呼び出しの第 1 引数は列挙型の名前です。

第 2 引数は列挙型メンバー名の **ソース** です。空白で区切った名前の文字列、名前のシーケンス、キー/値のペアの 2 要素タプルのシーケンス、あるいは名前と値のマッピング (例: 辞書) を指定できます。最後の 2 個のオプションでは、列挙型へ任意の値を割り当てることができます。前の 2 つのオプションでは、1 から始まり増加していく整数を自動的に割り当てます (別の開始値を指定するには、start 引数を使用します)。*Enum* から派生した新しいクラスが返されます。言い換えれば、上記の Animal への割り当ては以下と等価です:

```
>>> class Animal(Enum):
...     ANT = 1
...     BEE = 2
...     CAT = 3
...     DOG = 4
...
```

デフォルトの開始番号が 0 ではなく 1 である理由は、0 がブール演算子では False になりますが、すべての列挙型メンバーの評価は True でなければならないためです。

機能 API による Enum の pickle 化は、その列挙型がどのモジュールで作成されたかを見つけ出すためにフレームスタックの実装の詳細が使われるので、トリッキーになることがあります (例えば別のモジュールのユーティリティ関数を使うと失敗しますし、IronPython や Jython ではうまくいきません)。解決策は、以下のようにモジュール名を明示的に指定することです:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', module=__name__)
```

警告: module が与えられない場合、Enum はそれがなにか決定できないため、新しい Enum メンバーは unpickle 化できなくなります; エラーをソースの近いところで発生させるため、pickle 化は無効になります。

新しい pickle プロトコルバージョン 4 では、一部の状況において、pickle がクラスを発見するための場所の設定に *__qualname__* を参照します。例えば、そのクラスがグローバルスコープ内のクラス SomeData 内で利用可能とするには以下のように指定します:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', qualname='SomeData.Animal')
```

完全な構文は以下のようになります:

```
Enum(value='NewEnumName', names=<...>, *, module='...', qualname='...', type=<mixed-in class>,␣
↪start=1)
```

value　新しい Enum クラスに記録されるそれ自身の名前です。

名前　Enum のメンバーです。空白またはカンマで区切った文字列でも構いません (特に指定がない限り、値は 1 から始まります):

```
'RED GREEN BLUE' | 'RED,GREEN,BLUE' | 'RED, GREEN, BLUE'
```

または名前のイテレータで指定もできます:

```
['RED', 'GREEN', 'BLUE']
```

または (名前, 値) のペアのイテレータでも指定できます:

```
[('CYAN', 4), ('MAGENTA', 5), ('YELLOW', 6)]
```

またはマッピングでも指定できます:

```
{'CHARTREUSE': 7, 'SEA_GREEN': 11, 'ROSEMARY': 42}
```

module　新しい Enum クラスが属するモジュールの名前です。

qualname　新しい Enum クラスが属するモジュールの場所です。

type　新しい Enum クラスに複合されるデータ型です。

start　names のみが渡されたときにカウントを開始する数です。

バージョン 3.5 で変更: *start* 引数が追加されました。

8.13.13 派生列挙型

IntEnum

提供されている 1 つ目の *Enum* の派生型であり、*int* のサブクラスでもあります。*IntEnum* のメンバーは整数と比較できます; さらに言うと、異なる整数列挙型どうしでも比較できます:

```
>>> from enum import IntEnum
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Request(IntEnum):
...     POST = 1
...     GET = 2
...
>>> Shape == 1
False
>>> Shape.CIRCLE == 1
True
>>> Shape.CIRCLE == Request.POST
True
```

ただし、これらも標準の *Enum* 列挙型とは比較できません:

```
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
```

(次のページに続く)

(前のページからの続き)

```
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...
>>> Shape.CIRCLE == Color.RED
False
```

IntEnum の値は他の用途では整数のように振る舞います:

```
>>> int(Shape.CIRCLE)
1
>>> ['a', 'b', 'c'][Shape.CIRCLE]
'b'
>>> [i for i in range(Shape.SQUARE)]
[0, 1]
```

IntFlag

提供されている 2 つ目の *Enum* の派生型 *IntFlag* も *int* を基底クラスとしています。*IntFlag* メンバーが *Enum* メンバーと異なるのは、ビット演算子 (&, |, ^, ~) を使って組み合わせられ、その結果も *IntFlag* メンバーになることです。しかし、名前が示すように、*IntFlag* は *int* のサブクラスでもあり、*int* が使われるところでもどこでも使えます。*IntFlag* メンバーに対してビット演算以外のどんな演算をしても、その結果は *IntFlag* メンバーではなくなります。

バージョン 3.6 で追加.

IntFlag クラスの例:

```
>>> from enum import IntFlag
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...
>>> Perm.R | Perm.W
<Perm.R|W: 6>
>>> Perm.R + Perm.W
6
>>> RW = Perm.R | Perm.W
>>> Perm.R in RW
True
```

組み合わせにも名前を付けられます:

```
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...     RWX = 7
```

(次のページに続く)

(前のページからの続き)

```
>>> Perm.RWX
<Perm.RWX: 7>
>>> ~Perm.RWX
<Perm.-8: -8>
```

IntFlag と *Enum* のもう 1 つの重要な違いは、フラグが設定されていない (値が 0 である) 場合、その真偽値としての評価は *False* になることです:

```
>>> Perm.R & Perm.X
<Perm.0: 0>
>>> bool(Perm.R & Perm.X)
False
```

IntFlag は *int* のサブクラスでもあるので、その両者を組み合わせられます:

```
>>> Perm.X | 8
<Perm.8|X: 9>
```

Flag

最後の派生型は *Flag* です。*IntFlag* と同様に、*Flag* メンバーもビット演算子 (&, |, ^, ~) を使って組み合わせられます。しかし *IntFlag* とは違い、他のどの *Flag* 列挙型とも *int* とも組み合わせたり、比較したりできません。値を直接指定することも可能ですが、値として *auto* を使い、*Flag* に適切な値を選ばせることが推奨されています。

バージョン 3.6 で追加.

IntFlag と同様に、*Flag* メンバーの組み合わせがどのフラグも設定されていない状態になった場合、その真偽値としての評価は *False* となります:

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.RED & Color.GREEN
<Color.0: 0>
>>> bool(Color.RED & Color.GREEN)
False
```

個別のフラグは 2 のべき乗 (1, 2, 4, 8, ...) の値を持つべきですが、フラグの組み合わせはそうはなりません:

```
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...     WHITE = RED | BLUE | GREEN
```

(次のページに続く)

(前のページからの続き)

```
...
>>> Color.WHITE
<Color.WHITE: 7>
```

” フラグが設定されていない” 状態に名前を付けても、その真偽値は変わりません:

```
>>> class Color(Flag):
...     BLACK = 0
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.BLACK
<Color.BLACK: 0>
>>> bool(Color.BLACK)
False
```

注釈: ほとんどの新しいコードでは、*Enum* と *Flag* が強く推奨されます。というのは、*IntEnum* と *IntFlag* は (整数と比較でき、従って推移的に他の無関係な列挙型と比較できてしまうことにより) 列挙型の意味論的な約束に反するからです。*IntEnum* と *IntFlag* は、*Enum* や *Flag* では上手くいかない場合のみに使うべきです; 例えば、整数定数を列挙型で置き換えるときや、他のシステムとの相互運用性を持たせたいときです。

その他

IntEnum は *enum* モジュールの一部ですが、単独での実装もとても簡単に行なえます:

```
class IntEnum(int, Enum):
    pass
```

ここでは似たような列挙型の派生を定義する方法を紹介します; 例えば、StrEnum は *int* ではなく *str* で複合させたものです。

いくつかのルール:

1. *Enum* のサブクラスを作成するとき、複合させるデータ型は、基底クラスの並びで *Enum* 自身より先に記述しなければなりません (上記 *IntEnum* の例を参照)。

2. *Enum* のメンバーはどんなデータ型でも構いませんが、追加のデータ型 (例えば、上の例の *int*) と複合させてしまうと、すべてのメンバーの値はそのデータ型でなければならなくなります。この制限は、メソッドの追加するだけの、*int* や *str* のような他のデータ型を指定しない複合には適用されません。

3. 他のデータ型と複合された場合、value 属性は、たとえ等価であり等価であると比較が行えても、列挙型メンバー自身としては **同じではありません** 。

4. %-方式の書式: *%s* および *%r* はそれぞれ *Enum* クラスの `__str__()` および `__repr__()` を呼び出します; その他のコード (IntEnum の *%i* や *%h* など) は列挙型のメンバーを複合されたデータ型として

扱います。

5. フォーマット済み文字列リテラル 、*str.format()* 、*format()* は、`__str__()` やサブクラスでオーバーライドされた `__format__()` よりもミックスイン型の `__format__()` を優先して使います。!s や!r フォーマットコードを使うと、*Enum* クラスの `__str__()` や `__repr__()` メソッドが強制的に使われます。

8.13.14 `__init__()` と `__init__()` のどちらを使うべきか

`__new__()` は *Enum* メンバーの実際の値をカスタマイズしたいときに利用します。他の変更を加える場合、`__new__()` と `__init__()` のどちらを利用するかは、`__init__()` の方が望ましいでしょう。

例えば、複数の値をコンストラクタに渡すが、その中の 1 つだけを値として使いたい場合は次のようにします:

```
>>> class Coordinate(bytes, Enum):
...     """
...     Coordinate with binary codes that can be indexed by the int code.
...     """
...     def __new__(cls, value, label, unit):
...         obj = bytes.__new__(cls, [value])
...         obj._value_ = value
...         obj.label = label
...         obj.unit = unit
...         return obj
...     PX = (0, 'P.X', 'km')
...     PY = (1, 'P.Y', 'km')
...     VX = (2, 'V.X', 'km/s')
...     VY = (3, 'V.Y', 'km/s')
...

>>> print(Coordinate['PY'])
Coordinate.PY

>>> print(Coordinate(3))
Coordinate.VY
```

8.13.15 興味深い例

Enum, *IntEnum*, *IntFlag*, *Flag* は用途の大部分をカバーすると予想されますが、そのすべてをカバーできているわけではありません。ここでは、そのまま、あるいは独自の列挙型を作る例として使える、様々なタイプの列挙型を紹介します。

値の省略

多くの用途では、列挙型の実際の値が何かは気にされません。このタイプの単純な列挙型を定義する方法はいくつかあります:

- 値に *auto* インスタンスを使用する
- 値として *object* インスタンスを使用する
- 値として解説文字列を使用する
- 値としてタプルを使用し、独自の `__new__()` を使用してタプルを *int* 値で置き換える

これらのどの方法を使ってもユーザーに対して、値は重要ではなく、他のメンバーの番号の振り直しをする必要無しに、メンバーの追加、削除、並べ替えが行えるということを示せます。

どの方法を選んでも、(重要でない) 値を隠す *repr()* を提供すべきです:

```
>>> class NoValue(Enum):
...     def __repr__(self):
...         return '<%s.%s>' % (self.__class__.__name__, self.name)
...
```

auto を使う

auto を使うと次のようになります:

```
>>> class Color(NoValue):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.GREEN
<Color.GREEN>
```

object を使う

object を使うと次のようになります:

```
>>> class Color(NoValue):
...     RED = object()
...     GREEN = object()
...     BLUE = object()
...
>>> Color.GREEN
<Color.GREEN>
```

解説文字列を使う

値として文字列を使うと次のようになります:

```
>>> class Color(NoValue):
...     RED = 'stop'
...     GREEN = 'go'
...     BLUE = 'too fast!'
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
'go'
```

独自の __new__() を使う

自動で番号を振る __new__() を使うと次のようになります:

```
>>> class AutoNumber(NoValue):
...     def __new__(cls):
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
...
>>> class Color(AutoNumber):
...     RED = ()
...     GREEN = ()
...     BLUE = ()
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
2
```

AutoNumber をより広い用途で使うには、シグニチャに *args を追加します:

```
>>> class AutoNumber(NoValue):
...     def __new__(cls, *args):      # this is the only change from above
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
...
```

AutoNumber を継承すると、追加の引数を取り扱える独自の __init__ が書けます。

```
>>> class Swatch(AutoNumber):
...     def __init__(self, pantone='unknown'):
...         self.pantone = pantone
```

(次のページに続く)

(前のページからの続き)

```
...     AUBURN = '3497'
...     SEA_GREEN = '1246'
...     BLEACHED_CORAL = () # New color, no Pantone code yet!
...
>>> Swatch.SEA_GREEN
<Swatch.SEA_GREEN: 2>
>>> Swatch.SEA_GREEN.pantone
'1246'
>>> Swatch.BLEACHED_CORAL.pantone
'unknown'
```

注釈: `__new__()` メソッドが定義されていれば、Enum 番号の作成時に使用されます; これは Enum の `__new__()` と置き換えられ、クラスが作成された後の既存の番号を取得に使用されます。

OrderedEnum

IntEnum をベースとしないため、通常の *Enum* の不変条件 (他の列挙型と比較できないなど) のままで、メンバーを順序付けできる列挙型です:

```
>>> class OrderedEnum(Enum):
...     def __ge__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value >= other.value
...         return NotImplemented
...     def __gt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value > other.value
...         return NotImplemented
...     def __le__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value <= other.value
...         return NotImplemented
...     def __lt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value < other.value
...         return NotImplemented
...
>>> class Grade(OrderedEnum):
...     A = 5
...     B = 4
...     C = 3
...     D = 2
...     F = 1
...
>>> Grade.C < Grade.A
True
```

DuplicateFreeEnum

値が同じメンバーが見つかった場合、別名を作るのではなく、エラーを送出します:

```
>>> class DuplicateFreeEnum(Enum):
...     def __init__(self, *args):
...         cls = self.__class__
...         if any(self.value == e.value for e in cls):
...             a = self.name
...             e = cls(self.value).name
...             raise ValueError(
...                 "aliases not allowed in DuplicateFreeEnum:  %r --> %r"
...                 % (a, e))
...
>>> class Color(DuplicateFreeEnum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     GRENE = 2
...
Traceback (most recent call last):
...
ValueError: aliases not allowed in DuplicateFreeEnum:  'GRENE' --> 'GREEN'
```

注釈: これは Enum に別名を無効にするのと同様な振る舞いの追加や変更をおこなうためのサブクラス化に役立つ例です。単に別名を無効にしたいだけなら、*unique()* デコレーターを使用して行えます。

Planet

__new__() や __init__() が定義されている場合、列挙型メンバーの値はこれらのメソッドに渡されます:

```
>>> class Planet(Enum):
...     MERCURY = (3.303e+23, 2.4397e6)
...     VENUS   = (4.869e+24, 6.0518e6)
...     EARTH   = (5.976e+24, 6.37814e6)
...     MARS    = (6.421e+23, 3.3972e6)
...     JUPITER = (1.9e+27,   7.1492e7)
...     SATURN  = (5.688e+26, 6.0268e7)
...     URANUS  = (8.686e+25, 2.5559e7)
...     NEPTUNE = (1.024e+26, 2.4746e7)
...     def __init__(self, mass, radius):
...         self.mass = mass       # in kilograms
...         self.radius = radius   # in meters
...     @property
...     def surface_gravity(self):
...         # universal gravitational constant  (m3 kg-1 s-2)
...         G = 6.67300E-11
...         return G * self.mass / (self.radius * self.radius)
...
```

(次のページに続く)

(前のページからの続き)

```
>>> Planet.EARTH.value
(5.976e+24, 6378140.0)
>>> Planet.EARTH.surface_gravity
9.802652743337129
```

TimePeriod

ignore 属性の使用方法のサンプルです:

```
>>> from datetime import timedelta
>>> class Period(timedelta, Enum):
...     "different lengths of time"
...     _ignore_ = 'Period i'
...     Period = vars()
...     for i in range(367):
...         Period['day_%d' % i] = i
...
>>> list(Period)[:2]
[<Period.day_0: datetime.timedelta(0)>, <Period.day_1: datetime.timedelta(days=1)>]
>>> list(Period)[-2:]
[<Period.day_365: datetime.timedelta(days=365)>, <Period.day_366: datetime.timedelta(days=366)>
↪]
```

8.13.16 Enum はどう違うのか?

Enum は Enum 派生クラスやそれらのインスタンス (メンバー) 双方の多くの側面に影響を及ぼすカスタムメタクラスを持っています。

Enum クラス

EnumMeta メタクラスは、__contains__()、__dir__()、__iter__() および標準的なクラスでは失敗するが *Enum* クラスでは動作するその他のメソッド (*list(Color)* や *some_enum_var in Color* など) を責任を持って提供します。EnumMeta は最終的な *Enum* クラスのさまざまなメソッド (__new__()、*__getnewargs__()*、__str__() および __repr__()) が正しいことを責任を持って保証します。

Enum メンバー (インスタンス)

Enum メンバーについて最も興味深いのは、それらがシングルトンであるということです。EnumMeta は *Enum* 自身を作成し、メンバーを作成し、新しいインスタンスが作成されていないかどうかを確認するために既存のメンバーインスタンスだけを返すカスタム __new__() を追加します。

細かい点

__dunder__ 名のサポート

__members__ は読み込み専用の、member_name:member を要素とする順序付きマッピングです。これはクラスでのみ利用可能です。

__new__() が、もし指定されていた場合、列挙型のメンバーを作成し、返します; そのメンバー の _value_ を適切に設定するのも非常によい考えです。いったん全てのメンバーが作成されると、それ以降 __new__() は使われません。

sunder 名のサポート

- _name_ -- メンバー名
- _value_ -- メンバーの値; __new__ で設定したり、変更したりできます
- _missing_ -- 値が見付からなかったときに使われる検索関数; オーバーライドされていることがあります
- _ignore_ -- a list of names, either as a *list()* or a *str()*, that will not be transformed into members, and will be removed from the final class
- _order_ -- Python 2/3 のコードでメンバーの順序を固定化するのに利用されます（クラス属性で、クラス作成時に削除されます）
- _generate_next_value_ -- *Functional API* から利用され、*auto* が列挙型のメンバーの適切な値を取得するのに使われます。オーバーライドされます。

バージョン 3.6 で追加: _missing_, _order_, _generate_next_value_

バージョン 3.7 で追加: _ignore_

Pythono 2 / Python 3 のコードの同期を取りやすくするために _order_ 属性を提供できます。実際の列挙値の順序と比較して一致してなければエラーを送出します:

```
>>> class Color(Enum):
...     _order_ = 'RED GREEN BLUE'
...     RED = 1
...     BLUE = 3
...     GREEN = 2
...
Traceback (most recent call last):
...
TypeError: member order does not match _order_
```

注釈: Python 2 のコードでは _order_ 属性は定義順が記録される前消えてしまうため、重要です。

Enum メンバー型

Enum メンバーは、それらの *Enum* クラスのインスタンスで、通常は EnumClass.member のようにアクセスします。ある状況下では、EnumClass.member.member としてもアクセスできますが、この方法は絶対に使うべきではありません。というのは、この検索は失敗するか、さらに悪い場合には、探している *Enum* メンバー以外のものを返す場合もあるからです (これがメンバーの名前に大文字のみを使うのが良い理由の 1 つでもあります):

```
>>> class FieldTypes(Enum):
...     name = 0
...     value = 1
...     size = 2
...
>>> FieldTypes.value.size
<FieldTypes.size: 2>
>>> FieldTypes.size.value
2
```

バージョン 3.5 で変更.

Enum クラスとメンバーの真偽値

(*int*, *str* などのような) 非 *Enum* 型と複合させた *Enum* のメンバーは、その複合された型の規則に従って評価されます; そうでない場合は、全てのメンバーは *True* と評価されます。メンバーの値に依存する独自の Enum の真偽値評価を行うには、クラスに次のコードを追加してください:

```
def __bool__(self):
    return bool(self.value)
```

Enum クラスは常に *True* と評価されます。

メソッド付きの Enum クラス

Enum サブクラスに追加のメソッドを与えた場合、上述の *Planet* クラスのように、そのメソッドはメンバーの *dir()* に表示されますが、クラスの *dir()* には表示されません:

```
>>> dir(Planet)
['EARTH', 'JUPITER', 'MARS', 'MERCURY', 'NEPTUNE', 'SATURN', 'URANUS', 'VENUS', '__class__', '_
↪_doc__', '__members__', '__module__']
>>> dir(Planet.EARTH)
['__class__', '__doc__', '__module__', 'name', 'surface_gravity', 'value']
```

Flag のメンバーの組み合わせ

Flag メンバーの組み合わせに名前が無い場合、*repr()* の出力には、その値にある全ての名前を持つフラグと全ての名前を持つ組み合わせが含まれます:

```
>>> class Color(Flag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...     MAGENTA = RED | BLUE
...     YELLOW = RED | GREEN
...     CYAN = GREEN | BLUE
...
>>> Color(3)  # named combination
<Color.YELLOW: 3>
>>> Color(7)      # not named combination
<Color.CYAN|MAGENTA|BLUE|YELLOW|GREEN|RED: 7>
```

第

NINE

数値と数学モジュール

この章で記述されているモジュールは、数値に関するあるいは数学関係の関数とデータ型を提供します。*numbers* モジュールは、数値の型の抽象的な階層を定義します。*math* と *cmath* モジュールは、浮動小数点数と複素数のための様々な数学関数を含んでいます。*decimal* モジュールは、任意精度の計算を使用して、10進数の正確な表現をサポートします。

この章では以下のモジュールが記述されています:

9.1 numbers --- 数の抽象基底クラス

ソースコード: Lib/numbers.py

---

numbers モジュール (**PEP 3141**) は数の **抽象基底クラス** の階層を定義します。この階層では、さらに多くの演算が順番に定義されます。このモジュールで定義される型はどれもインスタンス化できません。

class numbers.Number

数の階層の根。引数 *x* が、種類は何であれ、数であるということだけチェックしたい場合、`isinstance(x, Number)` が使えます。

9.1.1 数値塔

class numbers.Complex

この型のサブクラスは複素数を表し、組み込みの *complex* 型を受け付ける演算を含みます。それらは: *complex* および *bool* への変換、*real*, *imag*, `+`, `-`, `*`, `/`, *abs()*, *conjugate()*, `==`, `!=` です。`-` と `!=` 以外の全てのものは抽象メソッドや抽象プロパティです。

real

抽象プロパティ。この数の実部を取り出します。

imag

抽象プロパティ。この数の虚部を取り出します。

abstractmethod conjugate()

抽象プロパティ。複素共役を返します。たとえば、`(1+3j).conjugate() == (1-3j)` です。

class numbers.Real

Real は、*Complex* 上に、実数に対して行える演算を加えます。

簡潔に言うとそれらは: *float* への変換, *math.trunc()*, *round()*, *math.floor()*, *math.ceil()*, *divmod()*, //, %, <, <=, > および >= です。

Real はまた *complex()*, *real*, *imag* および *conjugate()* のデフォルトを提供します。

class numbers.Rational

Real をサブタイプ化し *numerator* と *denominator* のプロパティを加えたものです。これらは既約分数のものでなければなりません。この他に *float()* のデフォルトも提供します。

numerator

抽象プロパティ。

denominator

抽象プロパティ。

class numbers.Integral

Rational をサブタイプ化し *int* への変換が加わります。*float()*, *numerator*, *denominator* のデフォルトを提供します。** に対する抽象メソッドと、ビット列演算 <<, >>, &, ^, |, ~ を追加します。

9.1.2 型実装者のための注意事項

実装する人は等しい数が等しく扱われるように同じハッシュを与えるように気を付けねばなりません。これは二つの異なった実数の拡張があるような場合にはややこしいことになるかもしれません。たとえば、*fractions.Fraction* は *hash()* を以下のように実装しています:

```
def __hash__(self):
    if self.denominator == 1:
        # Get integers right.
        return hash(self.numerator)
    # Expensive check, but definitely correct.
    if self == float(self):
        return hash(float(self))
    else:
        # Use tuple's hash to avoid a high collision rate on
        # simple fractions.
        return hash((self.numerator, self.denominator))
```

さらに数の ABC を追加する

数に対する ABC が他にも多く存在しうることは、言うまでもありません。それらの ABC を階層に追加する可能性が閉ざされるとしたら、その階層は貧相な階層でしかありません。たとえば、MyFoo を *Complex* と *Real* の間に付け加えるには、次のようにします:

```
class MyFoo(Complex): ...
MyFoo.register(Real)
```

算術演算の実装

算術演算を実装する際には、型混合 (mixed-mode) 演算を行うと、作者が両方の引数の型について知っているような実装を呼び出すか、両方の引数をそれぞれ最も似ている組み込み型に変換してその型で演算を行うか、どちらになるのが望ましい実装です。つまり、*Integral* のサブタイプに対しては `__add__()` と `__radd__()` を次のように定義するべきです:

```
class MyIntegral(Integral):

    def __add__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(self, other)
        else:
            return NotImplemented

    def __radd__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(other, self)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(other, self)
        elif isinstance(other, Integral):
            return int(other) + int(self)
        elif isinstance(other, Real):
            return float(other) + float(self)
        elif isinstance(other, Complex):
            return complex(other) + complex(self)
        else:
            return NotImplemented
```

ここには 5 つの異なる *Complex* のサブクラス間の混在型の演算があります。上のコードの中で MyIntegral と OtherTypeIKnowAbout に触れない部分を "ボイラープレート" と呼ぶことにしましょう。a を *Complex* のサブタイプである A のインスタンス (`a : A <: Complex`)、同様に `b : B <: Complex` として、`a + b` を考えます:

1. A が b を受け付ける `__add__()` を定義している場合、何も問題はありません。

2. A でボイラープレート部分に落ち込み、その結果 `__add__()` が値を返すならば、B に良く考えられた `__radd__()` が定義されている可能性を見逃してしまいますので、ボイラープレートは `__add__()` から *NotImplemented* を返すのが良いでしょう。(若しくは、A はまったく `__add__()` を実装すべきで

はなかったかもしれません。)

3. そうすると、B の `__radd__()` にチャンスが巡ってきます。ここで a が受け付けられるならば、結果は上々です。

4. ここでボイラープレートに落ち込むならば、もう他に試すべきメソッドはありませんので、デフォルト実装の出番です。

5. もし `B <: A` ならば、Python は `A.__add__` の前に `B.__radd__` を試します。これで良い理由は、A についての知識を持って実装しており、*Complex* に委ねる前にこれらのインスタンスを扱えるはずだからです。

もし `A <: Complex` かつ `B <: Real` で他に共有された知識が無いならば、適切な共通の演算は組み込みの *complex* を使ったものになり、どちらの `__radd__()` ともそこに着地するでしょうから、`a+b == b+a` です。

ほとんどの演算はどのような型についても非常に良く似ていますので、与えられた演算子について順結合 (forward) および逆結合 (reverse) のメソッドを生成する支援関数を定義することは役に立ちます。たとえば、*fractions.Fraction* では次のようなものを利用しています:

```
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
        else:
            return NotImplemented
    forward.__name__ = '__' + fallback_operator.__name__ + '__'
    forward.__doc__ = monomorphic_operator.__doc__

    def reverse(b, a):
        if isinstance(a, Rational):
            # Includes ints.
            return monomorphic_operator(a, b)
        elif isinstance(a, numbers.Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, numbers.Complex):
            return fallback_operator(complex(a), complex(b))
        else:
            return NotImplemented
    reverse.__name__ = '__r' + fallback_operator.__name__ + '__'
    reverse.__doc__ = monomorphic_operator.__doc__

    return forward, reverse

def _add(a, b):
    """a + b"""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)
```

(次のページに続く)

(前のページからの続き)

```
__add__, __radd__ = _operator_fallbacks(_add, operator.add)

# ...
```

9.2 math --- 数学関数

このモジュールは、C 標準で定義された数学関数へのアクセスを提供します。

これらの関数で複素数を使うことはできません。複素数に対応する必要があるならば、*cmath* モジュールにある同じ名前の関数を使ってください。ほとんどのユーザーは複素数を理解するのに必要なだけの数学を勉強したくないので、複素数に対応した関数と対応していない関数の区別がされています。これらの関数では複素数が利用できないため、引数に複素数を渡されると、複素数の結果が返るのではなく例外が発生します。その結果、どういった理由で例外が送出されたかに早い段階で気づく事ができます。

このモジュールでは次の関数を提供しています。明示的な注記のない限り、戻り値は全て浮動小数点数になります。

9.2.1 数論および数表現の関数

math.ceil(*x*)

x の「天井」(*x* 以上の最小の整数) を返します。*x* が浮動小数点数でなければ、内部的に `x.__ceil__()` が実行され、*Integral* 値が返されます。

math.comb(*n*, *k*)

n 個の中から *k* 個を重複無く順序をつけずに選ぶ方法の数を返します。

`k <= n` のとき `n! / (k! * (n - k)!)` と評価し、`k > n` のとき 0 と評価します。

Also called the binomial coefficient because it is equivalent to the coefficient of k-th term in polynomial expansion of the expression `(1 + x) ** n`.

いずれかの引数が整数でないなら *TypeError* を送出します。いずれかの引数が負であれば *ValueError* を送出します。

バージョン 3.8 で追加.

math.copysign(*x*, *y*)

x の大きさ (絶対値) で *y* と同じ符号の浮動小数点数を返します。符号付きのゼロをサポートしているプラットフォームでは、`copysign(1.0, -0.0)` は *-1.0* を返します。

math.fabs(*x*)

x の絶対値を返します。

math.factorial(*x*)

x の階乗を整数で返します。*x* が整数でないか、負の数の場合は、*ValueError* を送出します。

math.floor(*x*)

x の「床」(*x* 以下の最大の整数) を返します。*x* が浮動小数点数でなければ、内部的に `x.__floor__()` が実行され、*Integral* 値が返されます。

math.fmod(*x*, *y*)

プラットフォームの C ライブラリで定義されている `fmod(x, y)` を返します。Python の `x % y` という式は必ずしも同じ結果を返さないということに注意してください。C 標準の要求では、`fmod()` は除算の結果が *x* と同じ符号になり、大きさが `abs(y)` より小さくなるような整数 *n* については `fmod(x, y)` が厳密に (数学的に、つまり無限の精度で) `x - n*y` と等価であるよう求めています。Python の `x % y` は、*y* と同じ符号の結果を返し、浮動小数点の引数に対して厳密な解を出せないことがあります。例えば、`fmod(-1e-100, 1e100)` は `-1e-100` ですが、Python の `-1e-100 % 1e100` は `1e100-1e-100` になり、浮動小数点型で厳密に表現できず、ややこしいことに `1e100` に丸められます。このため、一般には浮動小数点の場合には関数 *fmod()* 、整数の場合には `x % y` を使う方がよいでしょう。

math.frexp(*x*)

x の仮数と指数を `(m, e)` のペアとして返します。*m* は float 型で、*e* は厳密に `x == m * 2**e` であるような整数型です。*x* がゼロの場合は、`(0.0, 0)` を返し、それ以外の場合は、`0.5 <= abs(m) < 1` を返します。これは浮動小数点型の内部表現を可搬性を保ったまま " 分解 (pick apart)" するためです。

math.fsum(*iterable*)

iterable 中の値の浮動小数点数の正確な和を返します。複数の部分和を追跡することで精度のロスを防ぎます:

```
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>> fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
1.0
```

アルゴリズムの正確性は、IEEE-754 演算の保証と、丸めモードが偶数丸め (half-even) である典型的な場合に依存します。Windows 以外のいくつかのビルドでは、下層の C ライブラリが拡張精度の加算を行い、時々計算途中の和を double 型へ丸めてしまうため、最下位ビットが消失することがあります。

より詳細な議論と代替となる二つのアプローチについては、ASPN cookbook recipes for accurate floating point summation をご覧下さい。

math.gcd(*a*, *b*)

整数 *a* と *b* の最大公約数を返します。*a* と *b* のいずれかがゼロでない場合、`gcd(a, b)` の値は *a* と *b* の両方を割り切ることのできる、最も大きな正の整数です。`gcd(0, 0)` は `0` を返します。

バージョン 3.5 で追加.

math.isclose(*a*, *b*, ***, *rel_tol=1e-09*, *abs_tol=0.0*)

値 *a* と *b* が互いに近い場合 `True` を、そうでない場合は `False` を返します。

2 値が近いと見なされるかどうかは与えられた絶対または相対許容差により決定されます。

rel_tol は相対許容差、すなわち *a* と *b* の絶対値の大きい方に対する *a* と *b* の許容される最大の差です。例えば許容差を 5% に設定する場合 `rel_tol=0.05` を渡します。デフォルトの許容差は `1e-09` で、2 値が 9 桁同じことを保証します。*rel_tol* は 0 より大きくなければなりません。

abs_tol は最小の絶対許容差です。0 に近い値を比較するのに有用です。*abs_tol* は 0 より大きくなければなりません。

エラーが起こらなければ結果は `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)` です。

IEEE 754 特殊値 `NaN`、`inf`、`-inf` は IEEE の規則に従って処理されます。具体的には、`NaN` は自身を含めたあらゆる値に近いとは見なされません。`inf` と `-inf` は自身とのみ近いと見なされます。

バージョン 3.5 で追加.

参考:

PEP 485 -- A function for testing approximate equality

`math.isfinite`(*x*)

x が無限でも NaN でもない場合に `True` を返します。それ以外の時には `False` を返します。(注意: `0.0` は有限数と扱われます。)

バージョン 3.2 で追加.

`math.isinf`(*x*)

x が正ないし負の無限数ならば `True` を返します。それ以外の時には `False` を返します。

`math.isnan`(*x*)

x が NaN (not a number、非数) の時に `True` を返します。それ以外の場合には `False` を返します。

`math.isqrt`(*n*)

非負整数 *n* の整数平方根を返します。これは *n* の正確な平方根の床であり、$a^2 \leq n$ を満たす最大の整数 *a* と等価です。

少し応用すれば、$n \leq a^2$ を満たす最小の整数 *a* 、言い換えれば *n* の正確な平方根の天井を効率的に得られます。正の *n* に対して、これは `a = 1 + isqrt(n - 1)` を使って計算できます。

バージョン 3.8 で追加.

`math.ldexp`(*x*, *i*)

`x * (2**i)` を返します。これは本質的に *frexp()* の逆関数です。

`math.modf`(*x*)

x の小数部分と整数部分を返します。両方の結果は *x* の符号を受け継ぎます。整数部は float 型で返されます。

`math.perm`(*n*, *k=None*)

n 個の中から *k* 個を重複無く順序をつけて選ぶ方法の数を返します。

`k <= n` のとき `n! / (n - k)!` と評価し、`k > n` のとき 0 と評価します。

k が指定されていないか None であれば、k はデフォルトで n となりこの関数は `n!` を返します。

いずれかの引数が整数でないなら *TypeError* を送出します。いずれかの引数が負であれば *ValueError* を送出します。

バージョン 3.8 で追加.

`math.prod`(*iterable*, *, *start=1*)

入力 *iterable* の全ての要素の積を計算します。積のデフォルト *start* 値は `1` です。

イテラブルが空のとき、初期値を返します。この関数は特に数値に使うことを意図されており、非数値を受け付けないことがあります。

バージョン 3.8 で追加.

`math.remainder`(x, y)

IEEE 754 標準方式の x を y で割った剰余を返します。有限な x と有限な y では、分数 `x / y` の厳密な値に最も近い整数を `n` として、`x - n*y` がこの返り値となります。`x / y` が隣り合う 2 つの整数のちょうど真ん中だった場合は、最も近い **偶数** が `n` として使われます。従って、剰余 `r = remainder(x, y)` は常に `abs(r) <= 0.5 * abs(y)` を満たします。

特殊なケースについては IEEE 754 に従います: 任意の有限な x に対する `remainder(x, math.inf)`、および任意の非 NaN の x に対する `remainder(x, 0)` と `remainder(math.inf, x)` は *ValueError* を送出します。剰余演算の結果がゼロの場合、そのゼロは x と同じ符号を持ちます。

IEEE 754 の二進浮動小数点数を使用しているプラットフォームでは、この演算の結果は常に厳密に表現可能です。丸め誤差は発生しません。

バージョン 3.7 で追加.

`math.trunc`(x)

x の *Integral* 値 (たいていは整数) へ切り捨てられた *Real* 値を返します。`x.__trunc__()` に処理を委譲します。

frexp() と *modf()* は C のものとは異なった呼び出し/返しパターンを持っていることに注意してください。引数を 1 つだけ受け取り、1 組のペアになった値を返すので、2 つ目の戻り値を ' 出力用の引数' 経由で返したりはしません (Python には出力用の引数はありません)。

ceil()、*floor()*、および *modf()* 関数については、非常に大きな浮動小数点数が **全て** 整数そのものになるということに注意してください。通常、Python の浮動小数点型は 53 ビット以上の精度をもたない (プラットフォームにおける C double 型と同じ) ので、結果的に `abs(x) >= 2**52` であるような浮動小数点型 x は小数部分を持たなくなるのです。

9.2.2 指数関数と対数関数

math.exp(*x*)
: $e = 2.718281...$ を自然対数の底として、e の x 乗を返します。この値は、通常は `math.e ** x` や `pow(math.e, x)` よりも精度が高いです。

math.expm1(*x*)
: e の x 乗から 1 を引いた値を返します。ここでの e は自然対数の底です。小さい浮動小数点数の x において、減算 `exp(x) - 1` は 桁落ち が発生します。*expm1()* 関数は、この量を最大精度で計算する方法を提供します。

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

バージョン 3.2 で追加.

math.log(*x*[, *base*])
: 引数が 1 つの場合、x の (e を底とする) 自然対数を返します。

引数が 2 つの場合、`log(x)/log(base)` として求められる *base* を底とした x の対数を返します。

math.log1p(*x*)
: *1+x* の自然対数 (つまり底 e の対数) を返します。結果はゼロに近い x に対して正確になるような方法で計算されます。

math.log2(*x*)
: 2 を底とする x の対数を返します。この関数は、一般に `log(x, 2)` よりも正確な値を返します。

バージョン 3.3 で追加.

参考:

int.bit_length() は、その整数を二進法で表すのに何ビット必要かを返す関数です。符号と先頭のゼロは無視されます。

math.log10(*x*)
: x の 10 を底とした対数 (常用対数) を返します。この関数は通常、`log(x, 10)` よりも高精度です。

math.pow(*x*, *y*)
: `x` の `y` 乗を返します。例外的な場合については、C99 標準の付録 'F' に可能な限り従います。特に、`pow(1.0, x)` と `pow(x, 0.0)` は、たとえ `x` が零や NaN でも、常に `1.0` を返します。もし `x` と `y` の両方が有限の値で、`x` が負、`y` が整数でない場合、`pow(x, y)` は未定義で、*ValueError* を送出します。

組み込みの `**` 演算子と違って、*math.pow()* は両方の引数を *float* 型に変換します。正確な整数の冪乗を計算するには `**` もしくは組み込みの *pow()* 関数を使ってください。

math.sqrt(*x*)
: x の平方根を返します。

9.2.3 三角関数

math.acos(*x*)
: *x* の逆余弦を、ラジアンで返します。

math.asin(*x*)
: *x* の逆正弦を、ラジアンで返します。

math.atan(*x*)
: *x* の逆正接を、ラジアンで返します。

math.atan2(*y*, *x*)
: `atan(y / x)` を、ラジアンで返します。戻り値は `-pi` から `pi` の間になります。この角度は、極座標平面において原点から `(x, y)` へのベクトルが X 軸の正の方向となす角です。*atan2()* のポイントは、両方の入力の符号が既知であるために、位相角の正しい象限を計算できることにあります。例えば、`atan(1)` と `atan2(1, 1)` はいずれも `pi/4` ですが、`atan2(-1, -1)` は `-3*pi/4` になります。

math.cos(*x*)
: *x* ラジアンの余弦を返します。

math.dist(*p*, *q*)
: それぞれ座標のシーケンス (またはイテラブル) として与えられる点 *p* と *q* の間のユークリッド距離を返します。二点の次元は同じでなければなりません。

  およそ次と等価です:

```
sqrt(sum((px - qx) ** 2.0 for px, qx in zip(p, q)))
```

  バージョン 3.8 で追加.

math.hypot(**coordinates*)
: ユークリッドノルム `sqrt(sum(x**2 for x in coordinates))` を返します。これは原点から座標で与えられる点までのベクトルの長さです。

  二次元の点 `(x, y)` では、これは直角三角形の斜辺をピタゴラスの定理 `sqrt(x*x + y*y)` を用いて計算することと等価です。

  バージョン 3.8 で変更: n 次元の点のサポートが追加されました。以前は、二次元の場合しかサポートされていませんでした。

math.sin(*x*)
: *x* ラジアンの正弦を返します。

math.tan(*x*)
: *x* ラジアンの正接を返します。

9.2.4 角度変換

math.**degrees**(*x*)
角 *x* をラジアンから度に変換します。

math.**radians**(*x*)
角 *x* を度からラジアンに変換します。

9.2.5 双曲線関数

双曲線関数 は円ではなく双曲線を元にした三角関数のようなものです。

math.**acosh**(*x*)
x の逆双曲線余弦を返します。

math.**asinh**(*x*)
x の逆双曲線正弦を返します。

math.**atanh**(*x*)
x の逆双曲線正接を返します。

math.**cosh**(*x*)
x の双曲線余弦を返します。

math.**sinh**(*x*)
x の双曲線正弦を返します。

math.**tanh**(*x*)
x の双曲線正接を返します。

9.2.6 特殊関数

math.**erf**(*x*)
x の 誤差関数 を返します。

erf() 関数は、伝統的な統計関数を計算するのに使うことができます。例えば、累積標準正規分布 を計算する関数は次のように定義できます:

```
def phi(x):
    'Cumulative distribution function for the standard normal distribution'
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

バージョン 3.2 で追加.

math.**erfc**(*x*)
x の相補誤差関数を返します。相補誤差関数 は `1.0 - erf(x)` と定義されます。この関数は、1 との引き算では 桁落ち をするような大きな *x* に対し使われます。

バージョン 3.2 で追加.

math.gamma(*x*)
x の ガンマ関数 を返します。

バージョン 3.2 で追加.

math.lgamma(*x*)
x のガンマ関数の絶対値の自然対数を返します。

バージョン 3.2 で追加.

9.2.7 定数

math.pi
利用可能なだけの精度の数学定数 π = 3.141592... (円周率)。

math.e
利用可能なだけの精度の数学定数 *e* = 2.718281... (自然対数の底)。

math.tau
利用可能なだけの精度の数学定数 τ = 6.283185... です。タウは 2π に等しい円定数で、円周と半径の比です。タウについて学ぶには Vi Hart のビデオ Pi is (still) Wrong をチェックして、パイを二倍食べて Tau day を祝い始めましょう！

バージョン 3.6 で追加.

math.inf
浮動小数の正の無限大です。(負の無限大には -math.inf を使います。) float('inf') の出力と等価です。

バージョン 3.5 で追加.

math.nan
浮動小数の非数 "not a number" (NaN) です。float('nan') の出力と等価です。

バージョン 3.5 で追加.

CPython implementation detail: *math* モジュールは、ほとんどが実行プラットフォームにおける C 言語の数学ライブラリ関数に対する薄いラッパでできています。例外時の挙動は、適切である限り C99 標準の Annex F に従います。現在の実装では、sqrt(-1.0) や log(0.0) といった (C99 Annex F で不正な演算やゼロ除算を通知することが推奨されている) 不正な操作に対して *ValueError* を送出し、(例えば exp(1000.0) のような) 演算結果がオーバーフローする場合には *OverflowError* を送出します。上記の関数群は、1 つ以上の引数が NaN であった場合を除いて NaN を返しません。引数に NaN が与えられた場合は、殆どの関数は NaN を返しますが、(C99 Annex F に従って) 別の動作をする場合があります。例えば、pow(float('nan'), 0.0) や hypot(float('nan'), float('inf')) といった場合です。訳注: 例外が発生せずに結果が返ると、計算結果がおかしくなった原因が複素数を渡したためだということに気づくのが遅れる可能性があります。

Python は signaling NaN と quiet NaN を区別せず、signaling NaN に対する挙動は未定義とされていることに注意してください。典型的な挙動は、全ての NaN を quiet NaN として扱うことです。

参考:

cmath **モジュール**　これらの多くの関数の複素数版。

9.3 cmath --- 複素数のための数学関数

このモジュールは、複素数を扱う数学関数へのアクセスを提供しています。このモジュール中の関数は整数、浮動小数点数または複素数を引数にとります。また、`__complex__()` または `__float__()` どちらかのメソッドを提供している Python オブジェクトも受け付けます。これらのメソッドはそのオブジェクトを複素数または浮動小数点数に変換するのにそれぞれ使われ、呼び出された関数はそうして変換された結果を利用します。

注釈:　ハードウェア及びシステムレベルでの符号付きゼロのサポートがあるプラットフォームでは、分枝切断 (branch cut) の関わる関数において切断された **両側** の分枝で連続になります。ゼロの符号でどちらの分枝であるかを区別するのです。符号付きゼロがサポートされないプラットフォームでは連続性は以下の仕様で述べるようになります。

9.3.1 極座標変換

Python の複素数 `z` は内部的には **直交座標** もしくは **デカルト座標** と呼ばれる座標を使って格納されています。この座標はその複素数の **実部** `z.real` と **虚部** `z.imag` で決まります。言い換えると:

```
z == z.real + z.imag*1j
```

極座標 は複素数を表現する別の方法です。極座標では、複素数 z は半径 r と位相角 phi で定義されます。半径 r は z から原点までの距離です。位相 phi は x 軸の正の部分から原点と z を結んだ線分までの角度を反時計回りにラジアンで測った値です。

次の関数はネイティブの直交座標を極座標に変換したりその逆を行うのに使えます。

cmath.phase(*x*)

x の位相 (x の **偏角** とも呼びます) を浮動小数点数で返します。`phase(x)` は `math.atan2(x.imag, x.real)` と同等です。返り値は [-π , π] の範囲にあり、この演算の分枝切断は負の実軸に沿って延びていて、上から連続です。(現在のほとんどのシステムはそうですが) 符号付きゼロをサポートしているシステムでは、結果の符号は `x.imag` がゼロであってさえ `x.imag` の符号と等しくなります:

```
>>> phase(complex(-1.0, 0.0))
3.141592653589793
>>> phase(complex(-1.0, -0.0))
-3.141592653589793
```

注釈: 複素数 x のモジュラス (絶対値) は組み込みの *abs()* 関数で計算できます。この演算を行う *cmath* モジュールの関数はありません。

cmath.**polar**(x)
: x の極座標表現を返します。x の半径 r と x の位相 *phi* の組 `(r, phi)` を返します。`polar(x)` は `(abs(x), phase(x))` に等しいです。

cmath.**rect**(r, *phi*)
: 極座標 r, *phi* を持つ複素数 x を返します。値は `r * (math.cos(phi) + math.sin(phi)*1j)` に等しいです。

9.3.2 指数関数と対数関数

cmath.**exp**(x)
: e を自然対数の底として、e の x 乗を返します。

cmath.**log**(x[, *base*])
: *base* を底とする x の対数を返します。もし *base* が指定されていない場合には、x の自然対数を返します。分枝切断を一つもち、0 から負の実数軸に沿って -∞ へと延びており、上から連続しています。

cmath.**log10**(x)
: x の底を 10 とする対数を返します。*log()* と同じ分枝切断を持ちます。

cmath.**sqrt**(x)
: x の平方根を返します。*log()* と同じ分枝切断を持ちます。

9.3.3 三角関数

cmath.**acos**(x)
: x の逆余弦を返します。この関数には二つの分枝切断 (branch cut) があります: 一つは 1 から右側に実数軸に沿って∞へと延びていて、下から連続しています。もう一つは -1 から左側に実数軸に沿って -∞へと延びていて、上から連続しています。

cmath.**asin**(x)
: x の逆正弦を返します。*acos()* と同じ分枝切断を持ちます。

cmath.**atan**(x)
: x の逆正接を返します。二つの分枝切断があります: 一つは `1j` から虚数軸に沿って ∞ `j` へと延びており、右から連続です。もう一つは `-1j` から虚数軸に沿って -∞ `j` へと延びており、左から連続です。

cmath.**cos**(x)
: x の余弦を返します。

cmath.**sin**(x)
: x の正弦を返します。

cmath.**tan**(*x*)
: *x* の正接を返します。

9.3.4 双曲線関数

cmath.**acosh**(*x*)
: *x* の逆双曲線余弦を返します。分枝切断が一つあり、1 の左側に実数軸に沿って -∞へと延びていて、上から連続しています。

cmath.**asinh**(*x*)
: *x* の逆双曲線正弦を返します。二つの分枝切断があります: 一つは 1j から虚数軸に沿って ∞ j へと延びており、右から連続です。もう一つは -1j から虚数軸に沿って -∞ j へと延びており、左から連続です。

cmath.**atanh**(*x*)
: *x* の逆双曲線正接を返します。二つの分枝切断があります: 一つは 1 から実数軸に沿って ∞ へと延びており、下から連続です。もう一つは -1 から実数軸に沿って -∞ へと延びており、上から連続です。

cmath.**cosh**(*x*)
: *x* の双曲線余弦を返します。

cmath.**sinh**(*x*)
: *x* の双曲線正弦を返します。

cmath.**tanh**(*x*)
: *x* の双曲線正接を返します。

9.3.5 類別関数

cmath.**isfinite**(*x*)
: *x* の実部、虚部ともに有限であれば True を返し、それ以外の場合 False を返します。

  バージョン 3.2 で追加.

cmath.**isinf**(*x*)
: *x* の実数部または虚数部が正または負の無限大であれば True を、そうでなければ False を返します。

cmath.**isnan**(*x*)
: *x* の実部と虚部のどちらかが NaN のとき True を返し、それ以外の場合 False を返します。

cmath.**isclose**(*a*, *b*, ***, *rel_tol=1e-09*, *abs_tol=0.0*)
: 値 *a* と *b* が互いに近い場合 True を、そうでない場合は False を返します。

  2 値が近いと見なされるかどうかは与えられた絶対または相対許容差により決定されます。

  rel_tol は相対許容差、すなわち *a* と *b* の絶対値の大きい方に対する *a* と *b* の許容される最大の差です。例えば許容差を 5% に設定する場合 rel_tol=0.05 を渡します。デフォルトの許容差は 1e-09 で、2 値が 9 桁同じことを保証します。*rel_tol* は 0 より大きくなければなりません。

abs_tol は最小の絶対許容差です。0 に近い値を比較するのに有用です。*abs_tol* は 0 より大きくなければなりません。

エラーが起こらなければ結果は `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)` です。

IEEE 754 特殊値 `NaN`、`inf`、`-inf` は IEEE の規則に従って処理されます。具体的には、`NaN` は自身を含めたあらゆる値に近いとは見なされません。`inf` と `-inf` は自身とのみ近いと見なされます。

バージョン 3.5 で追加.

参考:

PEP 485 -- A function for testing approximate equality

9.3.6 定数

cmath.pi
定数 **π** (円周率) で、浮動小数点数です。

cmath.e
定数 *e* (自然対数の底) で、浮動小数点数です。

cmath.tau
数学定数 **τ** で、浮動小数点数です。

バージョン 3.6 で追加.

cmath.inf
浮動小数点数の正の無限大です。`float('inf')` と等価です。

バージョン 3.6 で追加.

cmath.infj
実部がゼロ、虚部が正の無限大の複素数です。`complex(0.0, float('inf'))` と等価です。

バージョン 3.6 で追加.

cmath.nan
浮動小数点数の非数 "not a number" (NaN) です。`float('nan')` と等価です。

バージョン 3.6 で追加.

cmath.nanj
実部がゼロ、虚部が NaN の複素数です。`complex(0.0, float('nan'))` と等価です。

バージョン 3.6 で追加.

math と同じような関数が選ばれていますが、全く同じではないので注意してください。機能を二つのモジュールに分けているのは、複素数に興味がなかったり、もしかすると複素数とは何かすら知らないようなユーザがいるからです。そういった人たちはむしろ、`math.sqrt(-1)` が複素数を返すよりも例外を送出してほしいと

考えます。また、*cmath* で定義されている関数は、たとえ結果が実数で表現可能な場合 (虚数部がゼロの複素数) でも、常に複素数を返すので注意してください。

分枝切断 (branch cut) に関する注釈: 分枝切断を持つ曲線上では、与えられた関数は連続ではなくなります。これらは多くの複素関数における必然的な特性です。複素関数を計算する必要がある場合、これらの分枝に関して理解しているものと仮定しています。悟りに至るために何らかの (到底基礎的とはいえない) 複素数に関する書をひもといてください。数値計算を目的とした分枝切断の正しい選択方法についての情報としては、以下がよい参考文献となります:

参考:

Kahan, W: Branch cuts for complex elementary functions; or, Much ado about nothings's sign bit. In Iserles, A., and Powell, M. (eds.), The state of the art in numerical analysis. Clarendon Press (1987) pp165--211.

9.4 decimal --- 十進固定及び浮動小数点数の算術演算

ソースコード: Lib/decimal.py

---

decimal モジュールは正確に丸められた十進浮動小数点算術をサポートします。*decimal* には、*float* データ型に比べて、以下のような利点があります:

- 「(Decimal は) 人々を念頭にデザインされた浮動小数点モデルを元にしており、必然的に最も重要な指針があります -- コンピュータは人々が学校で習った算術と同じように動作する算術を提供しなければならない」-- 十進数演算仕様より。

- 十進数を正確に表現できます。`1.1` や `2.2` のような数は、二進数の浮動小数点型では正しく表現できません。エンドユーザは普通、二進数における `1.1 + 2.2` の近似値が `3.3000000000000003` だからといって、そのように表示してほしいとは考えないものです。

- 値の正確さは算術にも及びます。十進の浮動小数点による計算では、`0.1 + 0.1 + 0.1 - 0.3` は厳密にゼロに等しくなります。二進浮動小数点では `5.5511151231257827e-017` になってしまいます。ゼロに近い値とはいえ、この誤差は数値間の等価性テストの信頼性を阻害します。また、誤差が蓄積されることもあります。こうした理由から、数値間の等価性を厳しく保たなければならないようなアプリケーションを考えるなら、十進数による数値表現が望ましいということになります。

- *decimal* モジュールでは、有効桁数の表記が取り入れられており、例えば `1.30 + 1.20` は `2.50` になります。すなわち、末尾のゼロは有効数字を示すために残されます。こうした仕様は通貨計算を行うアプリケーションでは慣例です。乗算の場合、「教科書的な」アプローチでは、乗算の被演算子すべての桁数を使います。例えば、`1.3 * 1.2` は `1.56` になり、`1.30 * 1.20` は `1.5600` になります。

- ハードウェアによる 2 進浮動小数点表現と違い、*decimal* モジュールでは計算精度をユーザが変更できます (デフォルトでは 28 桁です)。この桁数はほとんどの問題解決に十分な大きさです:

```
>>> from decimal import *
>>> getcontext().prec = 6
```

(次のページに続く)

(前のページからの続き)

```
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

- 二進と十進の浮動小数点は、いずれも広く公開されている標準仕様のもとに実装されています。組み込みの浮動小数点型では、標準仕様で提唱されている機能のほんのささやかな部分を利用できるにすぎませんが、*decimal* では標準仕様が要求している全ての機能を利用できます。必要に応じて、プログラマは値の丸めやシグナル処理を完全に制御できます。この中には全ての不正確な操作を例外でブロックして正確な算術を遵守させるオプションもあります。

- decimal モジュールは「偏見なく、正確な丸めなしの十進算術 (固定小数点算術と呼ばれることもある) と丸めありの浮動小数点数算術」(十進数演算仕様より引用) をサポートするようにデザインされました。

このモジュールは、十進数型、算術コンテキスト (context for arithmetic)、そしてシグナル (signal) という三つの概念を中心に設計されています。

十進数型は変更不能です。これは符号、係数部、そして指数を持ちます。有効桁数を残すために、仮数部の末尾にあるゼロは切り詰められません。*decimal* では、`Infinity`, `-Infinity`, および `NaN` といった特殊な値も定義されています。標準仕様では `-0` と `+0` も区別します。

算術コンテキストとは、精度や値丸めの規則、指数部の制限を決めている環境です。この環境では、演算結果を表すためのフラグや、演算上発生した特定のシグナルを例外として扱うかどうかを決めるトラップイネーブラも定義しています。丸め規則には *ROUND_CEILING*, *ROUND_DOWN*, *ROUND_FLOOR*, *ROUND_HALF_DOWN*, *ROUND_HALF_EVEN*, *ROUND_HALF_UP*, *ROUND_UP*, および *ROUND_05UP* があります。

シグナルとは、演算の過程で生じる例外的条件です。個々のシグナルは、アプリケーションそれぞれの要求に従って、無視されたり、単なる情報とみなされたり、例外として扱われたりします。*decimal* モジュールには、*Clamped*, *InvalidOperation*, *DivisionByZero*, *Inexact*, *Rounded*, *Subnormal*, *Overflow*, *Underflow*, および *FloatOperation* といったシグナルがあります。

各シグナルには、フラグとトラップイネーブラがあります。演算上何らかのシグナルに遭遇すると、フラグは 1 にセットされます。このとき、もしトラップイネーブラが 1 にセットされていれば、例外を送出します。フラグの値は膠着型 (sticky) なので、演算によるフラグの変化をモニタしたければ、予めフラグをリセットしておかなければなりません。

参考:

- IBM による汎用十進演算仕様、The General Decimal Arithmetic Specification。

9.4.1 クイックスタートチュートリアル

普通、*decimal* を使うときには、モジュールをインポートし、現在の演算コンテキストを *getcontext()* で調べ、必要なら、精度、丸め、有効なトラップを設定します:

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[Overflow, DivisionByZero,
        InvalidOperation])

>>> getcontext().prec = 7       # Set a new precision
```

Decimal インスタンスは、整数、文字列、浮動小数点数、またはタプルから構成できます。整数や浮動小数点数からの構成は、整数や浮動小数点数の値を正確に変換します。*Decimal* は " 非数 (Not a Number)" を表す `NaN` や正負の `Infinity` (無限大)、`-0` といった特殊な値も表現できます:

```
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.140000000000000124344978758017532527446746826171875')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.4142135623730951')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.4142135623730950488016887242')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

FloatOperation シグナルがトラップされる場合、コンストラクタや順序比較において誤って decimal と float が混ざると、例外が送出されます:

```
>>> c = getcontext()
>>> c.traps[FloatOperation] = True
>>> Decimal(3.14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') < 3.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') == 3.5
True
```

バージョン 3.3 で追加.

新たな *Decimal* の有効桁数は入力の桁数だけで決まります。演算コンテキストにおける精度や値丸めの設定が影響するのは算術演算の間だけです。

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

C バージョンの内部制限を超えた場合、decimal の構成は *InvalidOperation* を送出します:

```
>>> Decimal("1e9999999999999999999")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.InvalidOperation: [<class 'decimal.InvalidOperation'>]
```

バージョン 3.3 で変更.

decimal はほとんどの場面で Python の他の機能とうまくやりとりできます。decimal 浮動小数点の空飛ぶサーカス (flying circus) をお見せしましょう:

```
>>> data = list(map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split()))
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
 Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
>>> sum(data)
Decimal('19.29')
>>> a,b,c = data[:3]
>>> str(a)
'1.34'
>>> float(a)
1.34
>>> round(a, 1)
Decimal('1.3')
>>> int(a)
1
>>> a * 5
Decimal('6.70')
>>> a * b
Decimal('2.5058')
>>> c % a
```

(次のページに続く)

(前のページからの続き)

```
Decimal('0.77')
```

いくつかの数学的関数も Decimal には用意されています:

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.414213562373095048801688724')
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal('10').ln()
Decimal('2.302585092994045684017991455')
>>> Decimal('10').log10()
Decimal('1')
```

quantize() メソッドは位を固定して数値を丸めます。このメソッドは、結果を固定の桁数で丸めることがよくある、金融アプリケーションで便利です:

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

前述のように、*getcontext()* 関数を使うと現在の演算コンテキストにアクセスでき、設定を変更できます。ほとんどのアプリケーションはこのアプローチで十分です。

より高度な作業を行う場合、*Context()* コンストラクタを使って別の演算コンテキストを作っておくと便利なことがあります。別の演算コンテキストをアクティブにしたければ、*setcontext()* を使います。

decimal モジュールでは、標準仕様に従って、すぐ利用できる二つの標準コンテキスト、*BasicContext* および *ExtendedContext* を提供しています。前者はほとんどのトラップが有効になっており、とりわけデバッグの際に便利です:

```
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')

>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
```

(次のページに続く)

(前のページからの続き)

```
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

演算コンテキストには、演算中に遭遇した例外的状況をモニタするためのシグナルフラグがあります。フラグが一度セットされると、明示的にクリアするまで残り続けます。そのため、フラグのモニタを行いたいような演算の前には clear_flags() メソッドでフラグをクリアしておくのがベストです。

```
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal('3.14159292')
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[Inexact, Rounded], traps=[])
```

flags エントリから、Pi の有理数による近似値が丸められた (コンテキスト内で決められた精度を超えた桁数が捨てられた) ことと、計算結果が厳密でない (無視された桁の値に非ゼロのものがあった) ことがわかります。

コンテキストの traps フィールドに入っている辞書を使うと、個々のトラップをセットできます:

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

ほとんどのプログラムでは、開始時に一度だけ現在の演算コンテキストを修正します。また、多くのアプリケーションでは、データから *Decimal* への変換はループ内で一度だけキャストして行います。コンテキストを設定し、*Decimal* オブジェクトを生成できたら、ほとんどのプログラムは他の Python 数値型と全く変わらないかのように *Decimal* を操作できます。

9.4.2 Decimal オブジェクト

class decimal.**Decimal**(*value="0"*, *context=None*)

value に基づいて新たな *Decimal* オブジェクトを構築します。

value は整数、文字列、タプル、*float* および他の *Decimal* オブジェクトにできます。*value* を指定しない場合、Decimal('0') を返します。*value* が文字列の場合、先頭と末尾の空白および全てのアンダースコアを取り除いた後には以下の 10 進数文字列の文法に従わなければなりません:

```
sign           ::=  '+' | '-'
digit          ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
indicator      ::=  'e' | 'E'
digits         ::=  digit [digit]...
decimal-part   ::=  digits '.' [digits] | ['.'] digits
exponent-part  ::=  indicator [sign] digits
infinity       ::=  'Infinity' | 'Inf'
nan            ::=  'NaN' [digits] | 'sNaN' [digits]
numeric-value  ::=  decimal-part [exponent-part] | infinity
numeric-string ::=  [sign] numeric-value | [sign] nan
```

他の Unicode 数字も上の `digit` の場所に使うことができます。つまり各書記体系における (アラビア-インド系やデーヴァナーガリーなど) の数字や、全角数字 0 (`'\uff10'`) から 9 (`'\uff19'`) までなどです。

value を *tuple* にする場合、タプルは三つの要素を持ち、それぞれ符号 (正なら `0` 、負なら `1`)、仮数部を表す数字の *tuple* 、そして指数を表す整数でなければなりません。例えば、`Decimal((0, (1, 4, 1, 4), -3))` は `Decimal('1.414')` を返します。

value を *float* にする場合、二進浮動小数点数値が損失なく正確に等価な Decimal に変換されます。この変換はしばしば 53 桁以上の精度を要求します。例えば、`Decimal(float('1.1'))` は `Decimal('1.100000000000000088817841970012523233890533447265625')` に変換されます。

context の精度 (precision) は、記憶される桁数には影響しません。桁数は *value* に指定した桁数だけから決定されます。例えば、演算コンテキストに指定された精度が 3 桁しかなくても、`Decimal('3.00000')` は 5 つのゼロを全て記憶します。

context 引数の目的は、*value* が正しくない形式の文字列であった場合に行う処理を決めることにあります; 演算コンテキストが *InvalidOperation* をトラップするようになっていれば、例外を送出します。それ以外の場合には、コンストラクタは値が `NaN` の *Decimal* を返します。

一度生成すると、*Decimal* オブジェクトは変更不能 (immutable) になります。

バージョン 3.2 で変更: コンストラクタに対する引数に *float* インスタンスも許されるようになりました。

バージョン 3.3 で変更: *FloatOperation* トラップがセットされていた場合 *float* 引数は例外を送出します。デフォルトでトラップはオフです。

バージョン 3.6 で変更: コード中の整数リテラルや浮動小数点リテラルと同様に、アンダースコアを用いて桁をグルーピングできます。

十進浮動小数点オブジェクトは、*float* や *int* のような他の組み込み型と多くの点で似ています。通常の数学演算や特殊メソッドを適用できます。また、*Decimal* オブジェクトはコピーでき、pickle 化でき、print で出力でき、辞書のキーにでき、集合の要素にでき、比較、保存、他の型 (*float* や *int*) への型強制を行えます。

十進オブジェクトの算術演算と整数や浮動小数点数の算術演算には少々違いがあります。十進オブジェクトに対して剰余演算を適用すると、計算結果の符号は除数の符号ではなく **被除数** の符号と一致します:

```
>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
```

整数除算演算子 `//` も同様に、実際の商の切り捨てではなく (0 に近付くように丸めた) 整数部分を返します。そうすることで通常の恒等式 `x == (x // y) * y + x % y` が保持されます:

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

演算子 `%` と演算子 `//` は (それぞれ) 仕様にあるような **剰余** 操作と **整数除算** 操作を実装しています。

Decimal オブジェクトは一般に、算術演算で浮動小数点数や *fractions.Fraction* オブジェクトと組み合わせることができません。例えば、*Decimal* に *float* を足そうとすると、*TypeError* が送出されます。ただし、Python の比較演算子を使って *Decimal* インスタンス `x` と別の数 `y` を比較することができます。これにより、異なる型の数間の等価比較の際に、紛らわしい結果を避けます。

バージョン 3.2 で変更: *Decimal* インスタンスと他の数値型が混在する比較が完全にサポートされるようになりました。

こうした標準的な数値型の特性の他に、十進浮動小数点オブジェクトには様々な特殊メソッドがあります:

adjusted()
: 仮数の先頭の一桁だけが残るように右側の数字を追い出す桁シフトを行い、その結果の指数部を返します: `Decimal('321e+5').adjusted()` は 7 を返します。最上桁の小数点からの相対位置を調べる際に使います。

as_integer_ratio()
: 与えられた *Decimal* インスタンスを、既約分数で分母が正数の分数として表現した整数のペア `(n, d)` を返します。

  ```
  >>> Decimal('-3.14').as_integer_ratio()
  (-157, 50)
  ```

  変換は正確に行われます。無限大に対しては OverflowError を、NaN に対しては ValueError を送出します。

バージョン 3.6 で追加.

as_tuple()
: 数値を表現するための **名前付きタプル**: `DecimalTuple(sign, digittuple, exponent)` を返します。

canonical()
: 引数の標準的 (canonical) エンコーディングを返します。現在のところ、*Decimal* インスタンスの

エンコーディングは常に標準的なので、この操作は引数に手を加えずに返します。

compare(*other*, *context=None*)

二つの Decimal インスタンスの値を比較します。*compare()* は Decimal インスタンスを返し、被演算子のどちらかが NaN ならば結果は NaN です:

```
a or b is a NaN  ==> Decimal('NaN')
a < b            ==> Decimal('-1')
a == b           ==> Decimal('0')
a > b            ==> Decimal('1')
```

compare_signal(*other*, *context=None*)

この演算は *compare()* とほとんど同じですが、全ての NaN がシグナルを送るところが異なります。すなわち、どちらの比較対象も発信 (signaling) NaN でないならば無言 (quiet) NaN である比較対象があたかも発信 NaN であるかのように扱われます。

compare_total(*other*, *context=None*)

二つの対象を数値によらず抽象表現によって比較します。*compare()* に似ていますが、結果は *Decimal* に全順序を与えます。この順序づけによると、数値的に等しくても異なった表現を持つ二つの *Decimal* インスタンスの比較は等しくなりません:

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

無言 NaN と発信 NaN もこの全順序に位置付けられます。この関数の結果は、もし比較対象が同じ表現を持つならば `Decimal('0')` であり、一つめの比較対象が二つめより下位にあれば `Decimal('-1')`、上位にあれば `Decimal('1')` です。全順序の詳細については仕様を参照してください。

この演算はコンテキストに影響されず、静かです。すなわち、フラグは変更されず、丸めは行われません。例外として、2 番目の比較対象の厳密な変換ができない場合、C バージョンのライブラリでは InvalidOperation 例外を送出するかもしれません。

compare_total_mag(*other*, *context=None*)

二つの対象を *compare_total()* のように数値によらず抽象表現によって比較しますが、両者の符号を無視します。`x.compare_total_mag(y)` は `x.copy_abs().compare_total(y.copy_abs())` と等価です。

この演算はコンテキストに影響されず、静かです。すなわち、フラグは変更されず、丸めは行われません。例外として、2 番目の比較対象の厳密な変換ができない場合、C バージョンのライブラリでは InvalidOperation 例外を送出するかもしれません。

conjugate()

self を返すだけです。このメソッドは十進演算仕様に適合するためだけのものです。

copy_abs()

引数の絶対値を返します。この演算はコンテキストに影響されず、静かです。すなわち、フラグは変更されず、丸めは行われません。

copy_negate()
引数の符号を変えて返します。この演算はコンテキストに影響されず、静かです。すなわち、フラグは変更されず、丸めは行われません。

copy_sign(*other*, *context=None*)
最初の演算対象のコピーに二つめと同じ符号を付けて返します。たとえば:

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

この演算はコンテキストに影響されず、静かです。すなわち、フラグは変更されず、丸めは行われません。例外として、2 番目の比較対象の厳密な変換ができない場合、C バージョンのライブラリでは InvalidOperation 例外を送出するかもしれません。

exp(*context=None*)
与えられた数での (自然) 指数関数 `e**x` の値を返します。結果は *ROUND_HALF_EVEN* 丸めモードで正しく丸められます。

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

from_float(*f*)
浮動小数点数を正確に小数に変換するクラスメソッドです。

なお、*Decimal.from_float(0.1)* は *Decimal('0.1')* と同じではありません。0.1 は二進浮動小数点数で正確に表せないので、その値は表現できる最も近い値、*0x1.999999999999ap-4* として記憶されます。浮動小数点数での等価な値は *0.1000000000000000055511151231257827021181583404541015625* です。

注釈: Python 3.2 以降では、*Decimal* インスタンスは *float* から直接構成できるようになりました。

```
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

バージョン 3.1 で追加.

fma(*other*, *third*, *context=None*)
融合積和 (fused multiply-add) です。self*other+third を途中結果の積 self*other で丸めを行わずに計算して返します。

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

is_canonical()
引数が標準的 (canonical) ならば *True* を返し、そうでなければ *False* を返します。現在のところ、*Decimal* のインスタンスは常に標準的なのでこのメソッドの結果はいつでも *True* です。

is_finite()
引数が有限の数値ならば *True* を、無限大か NaN ならば *False* を返します。

is_infinite()
引数が正または負の無限大ならば *True* を、そうでなければ *False* を返します。

is_nan()
引数が (無言か発信かは問わず) NaN であれば *True* を、そうでなければ *False* を返します。

is_normal(*context=None*)
引数が **正規** *(normal)* の有限数値ならば *True* を返します。引数がゼロ、非正規 (subnormal)、無限大または NaN であれば *False* を返します。

is_qnan()
引数が無言 NaN であれば *True* を、そうでなければ *False* を返します。

is_signed()
引数に負の符号がついていれば *True* を、そうでなければ *False* を返します。注意すべきはゼロや NaN なども符号を持ち得ることです。

is_snan()
引数が発信 NaN であれば *True* を、そうでなければ *False* を返します。

is_subnormal(*context=None*)
引数が非正規数 (subnormal) であれば *True* を、そうでなければ *False* を返します。

is_zero()
引数が (正または負の) ゼロであれば *True* を、そうでなければ *False* を返します。

ln(*context=None*)
演算対象の自然対数 (底 e の対数) を返します。結果は *ROUND_HALF_EVEN* 丸めモードで正しく丸められます。

log10(*context=None*)
演算対象の底 10 の対数を返します。結果は *ROUND_HALF_EVEN* 丸めモードで正しく丸められます。

logb(*context=None*)
非零の数値については、*Decimal* インスタンスとして調整された指数を返します。演算対象がゼロだった場合、`Decimal('-Infinity')` が返され *DivisionByZero* フラグが送出されます。演算対象が無限大だった場合、`Decimal('Infinity')` が返されます。

logical_and(*other*, *context=None*)

logical_and() は二つの **論理引数** (**論理引数** 参照) を取る論理演算です。結果は二つの引数の数字ごとの and です。

logical_invert(*context=None*)

logical_invert() は論理演算です。結果は引数の数字ごとの反転です。

logical_or(*other*, *context=None*)

logical_or() は二つの **論理引数** (**論理引数** 参照) を取る論理演算です。結果は二つの引数の数字ごとの or です。

logical_xor(*other*, *context=None*)

logical_xor() は二つの **論理引数** (**論理引数** 参照) を取る論理演算です。結果は二つの引数の数字ごとの排他的論理和です。

max(*other*, *context=None*)

max(self, other) と同じですが、値を返す前に現在のコンテキストに即した丸め規則を適用します。また、NaN に対して、(コンテキストの設定と、発信か無言どちらのタイプであるかに応じて) シグナルを発行するか無視します。

max_mag(*other*, *context=None*)

max() メソッドに似ていますが、比較は絶対値で行われます。

min(*other*, *context=None*)

min(self, other) と同じですが、値を返す前に現在のコンテキストに即した丸め規則を適用します。また、NaN に対して、(コンテキストの設定と、発信か無言どちらのタイプであるかに応じて) シグナルを発行するか無視します。

min_mag(*other*, *context=None*)

min() メソッドに似ていますが、比較は絶対値で行われます。

next_minus(*context=None*)

与えられたコンテキスト (またはコンテキストが渡されなければ現スレッドのコンテキスト) において表現可能な、操作対象より小さい最大の数を返します。

next_plus(*context=None*)

与えられたコンテキスト (またはコンテキストが渡されなければ現スレッドのコンテキスト) において表現可能な、操作対象より大きい最小の数を返します。

next_toward(*other*, *context=None*)

二つの比較対象が等しくなければ、一つめの対象に最も近く二つめの対象へ近付く方向の数を返します。もし両者が数値的に等しければ、二つめの対象の符号を採った一つめの対象のコピーを返します。

normalize(*context=None*)

数値を正規化 (normalize) して、右端に連続しているゼロを除去し、Decimal('0') と同じ結果はすべて Decimal('0e0') に変換します。等価クラスの属性から基準表現を生成する際に用います。たとえば、Decimal('32.100') と Decimal('0.321000e+2') の正規化は、いずれも同じ値 Decimal('32.1') になります。

number_class(*context=None*)

操作対象の **クラス** を表す文字列を返します。返されるのは以下の 10 種類のいずれかです。

- "-Infinity", 負の無限大であることを示します。
- "-Normal", 負の通常数であることを示します。
- "-Subnormal", 負の非正規数であることを示します。
- "-Zero", 負のゼロであることを示します。
- "+Zero", 正のゼロであることを示します。
- "+Subnormal", 正の非正規数であることを示します。
- "+Normal", 正の通常数であることを示します。
- "+Infinity", 正の無限大であることを示します。
- "NaN", 無言 (quiet) NaN (Not a Number) であることを示します。
- "sNaN", 発信 (signaling) NaN であることを示します。

quantize(*exp*, *rounding=None*, *context=None*)

二つ目の操作対象と同じ指数を持つように丸めを行った、一つめの操作対象と等しい値を返します。

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

他の操作と違い、打ち切り (quantize) 操作後の係数の長さが精度を越えた場合には、*InvalidOperation* がシグナルされます。これによりエラー条件がない限り打ち切られた指数が常に右側の引数と同じになることが保証されます。

同様に、他の操作と違い、quantize は Underflow を、たとえ結果が非正規になったり不正確になったとしても、シグナルしません。

二つ目の演算対象の指数が一つ目のそれよりも大きければ丸めが必要かもしれません。この場合、丸めモードは以下のように決められます。rounding 引数が与えられていればそれが使われます。そうでなければ context 引数で決まります。どちらの引数も渡されなければ現在のスレッドのコンテキストの丸めモードが使われます。

処理結果の指数が Emax よりも大きい場合や Etiny よりも小さい場合にエラーが返されます。

radix()

Decimal(10) つまり *Decimal* クラスがその全ての算術を実行する基数を返します。仕様との互換性のために取り入れられています。

remainder_near(*other*, *context=None*)

self を *other* で割った剰余を返します。これは self % other とは違って、剰余の絶対値を小さくするように符号が選ばれます。より詳しく言うと、n を self / other の正確な値に最も近い

整数としたときの `self - n * other` が返り値になります。最も近い整数が 2 つある場合には偶数のものが選ばれます。

結果が 0 になる場合の符号は *self* の符号と同じになります。

```
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

rotate(*other*, *context=None*)

一つ目の演算対象の数字を二つ目で指定された量だけ巡回 (rotate) した結果を返します。二つめの演算対象は -precision から precision までの範囲の整数でなければなりません。この二つ目の演算対象の絶対値を何桁ずらすかを決めます。そしてもし正の数ならば巡回の方向は左に、そうでなければ右になります。一つ目の演算対象の仮数部は必要ならば精度いっぱいまでゼロで埋められます。符号と指数は変えられません。

same_quantum(*other*, *context=None*)

self と *other* が同じ指数を持っているか、あるいは双方とも `NaN` である場合に真を返します。

この演算はコンテキストに影響されず、静かです。すなわち、フラグは変更されず、丸めは行われません。例外として、2 番目の比較対象の厳密な変換ができない場合、C バージョンのライブラリでは InvalidOperation 例外を送出するかもしれません。

scaleb(*other*, *context=None*)

二つ目の演算対象で調整された指数の一つ目の演算対象を返します。同じことですが、一つめの演算対象を `10**other` 倍したものを返します。二つ目の演算対象は整数でなければなりません。

shift(*other*, *context=None*)

一つ目の演算対象の数字を二つ目で指定された量だけシフトした結果を返します。二つ目の演算対象は -precision から precision までの範囲の整数でなければなりません。この二つ目の演算対象の絶対値が何桁ずらすかを決めます。そしてもし正の数ならばシフトの方向は左に、そうでなければ右になります。一つ目の演算対象の係数は必要ならば精度いっぱいまでゼロで埋められます。符号と指数は変えられません。

sqrt(*context=None*)

引数の平方根を最大精度で求めます。

to_eng_string(*context=None*)

文字列に変換します。指数が必要なら工学表記が使われます。

工学表記法では指数は 3 の倍数になります。これにより、基数の小数部には最大で 3 桁までの数字が残されるとともに、末尾に 1 つまたは 2 つの 0 の付加が必要とされるかもしれません。

たとえば、`Decimal('123E+1')` は `Decimal('1.23E+3')` に変換されます。

to_integral(*rounding=None*, *context=None*)

to_integral_value() メソッドと同じです。`to_integral` の名前は古いバージョンとの互換性

のために残されています。

to_integral_exact(*rounding=None*, *context=None*)
最近傍の整数に値を丸め、丸めが起こった場合には *Inexact* または *Rounded* のシグナルを適切に出します。丸めモードは以下のように決められます。rounding 引数が与えられていればそれが使われます。そうでなければ context 引数で決まります。どちらの引数も渡されなければ現在のスレッドのコンテキストの丸めモードが使われます。

to_integral_value(*rounding=None*, *context=None*)
Inexact や *Rounded* といったシグナルを出さずに最近傍の整数に値を丸めます。*rounding* が指定されていれば適用されます; それ以外の場合、値丸めの方法は *context* の設定か現在のコンテキストの設定になります。

論理引数

logical_and(), logical_invert(), logical_or(), および logical_xor() メソッドはその引数が **論理引数** であると想定しています。**論理引数** とは *Decimal* インスタンスで指数と符号は共にゼロであり、各桁の数字が 0 か 1 であるものです。

9.4.3 Context オブジェクト

コンテキスト (context) とは、算術演算における環境設定です。コンテキストは計算精度を決定し、値丸めの方法を設定し、シグナルのどれが例外になるかを決め、指数の範囲を制限しています。

多重スレッドで処理を行う場合には各スレッドごとに現在のコンテキストがあり、*getcontext()* や *setcontext()* といった関数でアクセスしたり設定変更できます:

decimal.**getcontext**()
アクティブなスレッドの現在のコンテキストを返します。

decimal.**setcontext**(*c*)
アクティブなスレッドのコンテキストを *c* に設定します。

with 文と *localcontext()* 関数を使って実行するコンテキストを一時的に変更することもできます。

decimal.**localcontext**(*ctx=None*)
with 文の入口でアクティブなスレッドのコンテキストを *ctx* のコピーに設定し、with 文を抜ける時に元のコンテキストに復旧する、コンテキストマネージャを返します。コンテキストが指定されなければ、現在のコンテキストのコピーが使われます。

たとえば、以下のコードでは精度を 42 桁に設定し、計算を実行し、そして元のコンテキストに復帰します:

```
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42   # Perform a high precision calculation
```

(次のページに続く)

(前のページからの続き)

```
    s = calculate_something()
s = +s  # Round the final result back to the default precision
```

新たなコンテキストは、以下で説明する *Context* コンストラクタを使って生成できます。その他にも、*decimal* モジュールでは作成済みのコンテキストを提供しています:

class decimal.BasicContext

汎用十進演算仕様で定義されている標準コンテキストの一つです。精度は 9 桁に設定されています。丸め規則は *ROUND_HALF_UP* です。すべての演算結果フラグはクリアされています。*Inexact*, *Rounded*, *Subnormal* を除く全ての演算エラートラップが有効 (例外として扱う) になっています。

多くのトラップが有効になっているので、デバッグの際に便利なコンテキストです。

class decimal.ExtendedContext

汎用十進演算仕様で定義されている標準コンテキストの一つです。精度は 9 桁に設定されています。丸め規則は *ROUND_HALF_EVEN* です。すべての演算結果フラグはクリアされています。トラップは全て無効 (演算中に一切例外を送出しない) になっています。

トラップが無効になっているので、エラーの伴う演算結果を NaN や Infinity にし、例外を送出しないようにしたいアプリケーションに向いたコンテキストです。このコンテキストを使うと、他の場合にはプログラムが停止してしまうような状況があっても実行を完了させられます。

class decimal.DefaultContext

Context コンストラクタが新たなコンテキストを作成するさいに雛形にするコンテキストです。このコンテキストのフィールド (精度の設定など) を変更すると、*Context* コンストラクタが生成する新たなコンテキストに影響を及ぼします。

このコンテキストは、主に多重スレッド環境で便利です。スレッドを開始する前に何らかのフィールドを変更しておくと、システム全体のデフォルト設定に効果を及ぼすことができます。スレッドを開始した後にフィールドを変更すると、競合条件を抑制するためにスレッドを同期化しなければならないので、推奨しません。

単一スレッドの環境では、このコンテキストを使わないよう薦めます。下で述べるように明示的にコンテキストを作成してください。

デフォルトの値は、prec=28, rounding=*ROUND_HALF_EVEN* で、トラップ *Overflow*, *InvalidOperation*, および *DivisionByZero* が有効になっています。

上に挙げた三つのコンテキストに加え、*Context* コンストラクタを使って新たなコンテキストを生成できます。

class decimal.Context(*prec=None*, *rounding=None*, *Emin=None*, *Emax=None*, *capitals=None*, *clamp=None*, *flags=None*, *traps=None*)

新たなコンテキストを生成します。あるフィールドが定義されていないか *None* であれば、*DefaultContext* からデフォルト値をコピーします。*flags* フィールドが設定されていないか *None* の場合には、全てのフラグがクリアされます。

prec フィールドは範囲 [1, *MAX_PREC*] 内の整数で、コンテキストにおける算術演算の計算精度を設定します。

rounding オプションは、節 **丸めモード** で挙げられる定数の一つです。

traps および *flags* フィールドには、セットしたいシグナルを列挙します。一般的に、新たなコンテキストを作成するときにはトラップだけを設定し、フラグはクリアしておきます。

Emin および *Emax* フィールドは、許容する指数の外限を指定する整数です。*Emin* は範囲 [*MIN_EMIN*, 0] 内で、*Emax* は範囲 [0, *MAX_EMAX*] 内でなければなりません。

capitals フィールドは 0 または 1 (デフォルト) にします。1 に設定すると、指数記号を大文字 E で出力します。それ以外の場合には Decimal('6.02e+23') のように e を使います。

clamp フィールドは、0 (デフォルト) または 1 です。1 に設定されると、このコンテキストにおける *Decimal* インスタンスの指数 e は厳密に範囲 Emin - prec + 1 <= e <= Emax - prec + 1 に制限されます。*clamp* が 0 なら、それより弱い条件が支配します: 調整された *Decimal* インスタンスの指数は最大で Emax です。*clamp* が 1 なら、大きな正規数は、可能なら、指数が減らされ、対応する数の 0 が係数に加えられ、指数の制約に合わせられます; これは数の値を保存しますが、有効な末尾の 0 に関する情報を失います。例えば:

```
>>> Context(prec=6, Emax=999, clamp=1).create_decimal('1.23e999')
Decimal('1.23000E+999')
```

clamp の値 1 は、IEEE 754 で規定された固定幅十進交換形式と互換にできます。

Context クラスでは、いくつかの汎用のメソッドの他、現在のコンテキストで算術演算を直接行うためのメソッドを数多く定義しています。加えて、*Decimal* の各メソッドについて (adjusted() および as_tuple() メソッドを例外として) 対応する *Context* のメソッドが存在します。たとえば、*Context* インスタンス C と *Decimal* インスタンス x に対して、C.exp(x) は x.exp(context=C) と等価です。それぞれの *Context* メソッドは、Decimal インスタンスが受け付けられるところならどこでも、Python の整数 (*int* のインスタンス) を受け付けます。

clear_flags()
フラグを全て 0 にリセットします。

clear_traps()
トラップを全て 0 にリセットします。

バージョン 3.3 で追加.

copy()
コンテキストの複製を返します。

copy_decimal(*num*)
Decimal インスタンス num のコピーを返します。

create_decimal(*num*)
self をコンテキストとする新たな Decimal インスタンスを *num* から生成します。*Decimal* コンストラクタと違い、数値を変換する際にコンテキストの精度、値丸め方法、フラグ、トラップを適用します。

定数値はしばしばアプリケーションの要求よりも高い精度を持っているため、このメソッドが役に

立ちます。また、値丸めを即座に行うため、例えば以下のように、入力値に値丸めを行わないために合計値にゼロの加算を追加するだけで結果が変わってしまうといった、現在の精度よりも細かい値の影響が紛れ込む問題を防げるという恩恵もあります。以下の例は、丸められていない入力を使うということは和にゼロを加えると結果が変わり得るという見本です:

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

このメソッドは IBM 仕様の to-number 演算を実装したものです。引数が文字列の場合、前や後ろに余計な空白を付けたり、アンダースコアを含めたりすることは許されません。

create_decimal_from_float(*f*)

浮動小数点数 *f* から新しい Decimal インスタンスを生成しますが、*self* をコンテキストとして丸めます。*Decimal.from_float()* クラスメソッドとは違い、変換にコンテキストの精度、丸めメソッド、フラグ、そしてトラップが適用されます。

```
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(math.pi)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create_decimal_from_float(math.pi)
Traceback (most recent call last):
    ...
decimal.Inexact: None
```

バージョン 3.1 で追加.

Etiny()

Emin - prec + 1 に等しい値を返します。演算結果の劣化が起こる桁の最小値です。アンダーフローが起きた場合、指数は *Etiny* に設定されます。

Etop()

Emax - prec + 1 に等しい値を返します。

Decimal を使った処理を行う場合、通常は *Decimal* インスタンスを生成して、算術演算を適用するというアプローチをとります。演算はアクティブなスレッドにおける現在のコンテキストの下で行われます。もう一つのアプローチは、コンテキストのメソッドを使った特定のコンテキスト下での計算です。コンテキストのメソッドは *Decimal* クラスのメソッドに似ているので、ここでは簡単な説明にとどめます。

abs(*x*)

x の絶対値を返します。

add(*x*, *y*)

x と *y* の和を返します。

canonical(*x*)

同じ Decimal オブジェクト *x* を返します。

compare(*x*, *y*)
x と *y* を数値として比較します。

compare_signal(*x*, *y*)
二つの演算対象の値を数値として比較します。

compare_total(*x*, *y*)
二つの演算対象を抽象的な表現を使って比較します。

compare_total_mag(*x*, *y*)
二つの演算対象を抽象的な表現を使い符号を無視して比較します。

copy_abs(*x*)
x のコピーの符号を 0 にセットして返します。

copy_negate(*x*)
x のコピーの符号を反転して返します。

copy_sign(*x*, *y*)
y から *x* に符号をコピーします。

divide(*x*, *y*)
x を *y* で除算した値を返します。

divide_int(*x*, *y*)
x を *y* で除算した値を整数に切り捨てて返します。

divmod(*x*, *y*)
二つの数値間の除算を行い、結果の整数部を返します。

exp(*x*)
e ** *x* を返します。

fma(*x*, *y*, *z*)
x を *y* 倍したものに *z* を加えて返します。

is_canonical(*x*)
x が標準的 (canonical) ならば True を返します。そうでなければ False です。

is_finite(*x*)
x が有限ならば True を返します。そうでなければ False です。

is_infinite(*x*)
x が無限ならば True を返します。そうでなければ False です。

is_nan(*x*)
x が qNaN か sNaN であれば True を返します。そうでなければ False です。

is_normal(*x*)
x が通常の数ならば True を返します。そうでなければ False です。

is_qnan(*x*)
x が無言 NaN であれば True を返します。そうでなければ False です。

is_signed(*x*)
x が負の数であれば True を返します。そうでなければ False です。

is_snan(*x*)
x が発信 NaN であれば True を返します。そうでなければ False です。

is_subnormal(*x*)
x が非正規数であれば True を返します。そうでなければ False です。

is_zero(*x*)
x がゼロであれば True を返します。そうでなければ False です。

ln(*x*)
x の自然対数 (底 e の対数) を返します。

log10(*x*)
x の底 10 の対数を返します。

logb(*x*)
演算対象の MSD の大きさの指数部を返します。

logical_and(*x*, *y*)
それぞれの桁に論理演算 *and* を当てはめます。

logical_invert(*x*)
x の全ての桁を反転させます。

logical_or(*x*, *y*)
それぞれの桁に論理演算 *or* を当てはめます。

logical_xor(*x*, *y*)
それぞれの桁に論理演算 *xor* を当てはめます。

max(*x*, *y*)
二つの値を数値として比較し、大きいほうを返します。

max_mag(*x*, *y*)
値を符号を無視して数値として比較します。

min(*x*, *y*)
二つの値を数値として比較し、小さいほうを返します。

min_mag(*x*, *y*)
値を符号を無視して数値として比較します。

minus(*x*)
Python における単項マイナス演算子に対応する演算です。

multiply(*x*, *y*)
x と *y* の積を返します。

next_minus(*x*)
x より小さい最大の表現可能な数を返します。

next_plus(*x*)
x より大きい最小の表現可能な数を返します。

next_toward(*x*, *y*)
x に *y* の方向に向かって最も近い数を返します。

normalize(*x*)
x をもっとも単純な形にします。

number_class(*x*)
x のクラスを指し示すものを返します。

plus(*x*)
Python における単項のプラス演算子に対応する演算です。コンテキストにおける精度や値丸めを適用するので、等値 (identity) 演算とは **違います**。

power(*x*, *y*, *modulo=None*)
x の y 乗を計算します。modulo が指定されていればモジュロを取ります。

引数が 2 つの場合、x**y を計算します。x が負の場合、y は整数でなければなりません。y が整数、結果が有限、結果が 'precision' 桁で正確に表現できる、という条件をすべて満たさない場合、結果は不正確になります。結果はコンテキストの丸めモードを使って丸められます。結果は常に、Python バージョンにおいて正しく丸められます。

Decimal(0) ** Decimal(0) results in InvalidOperation, and if InvalidOperation is not trapped, then results in Decimal('NaN').

バージョン 3.3 で変更: C モジュールは *power()* を適切に丸められた *exp()* および *ln()* 関数によって計算します。結果は well-defined ですが、「ほとんどの場合には適切に丸められる」だけです。

引数が 3 つの場合、(x**y) % modulo を計算します。この 3 引数の形式の場合、引数には以下の制限が課せられます。

- 全ての引数は整数
- y は非負でなければならない
- x と y の少なくともどちらかはゼロでない
- modulo は非零で大きくても 'precision' 桁

Context.power(x, y, modulo) で得られる値は (x**y) % modulo を精度無制限で計算して得られるものと同じ値ですが、より効率的に計算されます。結果の指数は x, y, modulo の指数に関係なくゼロです。この計算は常に正確です。

quantize(*x*, *y*)
x に値丸めを適用し、指数を *y* にした値を返します。

radix()
単に 10 を返します。何せ十進ですから :)

remainder(*x*, *y*)
整数除算の剰余を返します。

剰余がゼロでない場合、符号は割られる数の符号と同じになります。

remainder_near(*x*, *y*)
`x - y * n` を返します。ここで *n* は `x / y` の正確な値に一番近い整数です (この結果が 0 ならばその符号は *x* の符号と同じです)。

rotate(*x*, *y*)
x の *y* 回巡回したコピーを返します。

same_quantum(*x*, *y*)
2 つの演算対象が同じ指数を持っている場合に `True` を返します。

scaleb(*x*, *y*)
一つめの演算対象の指数部に二つめの値を加えたものを返します。

shift(*x*, *y*)
x を *y* 回シフトしたコピーを返します。

sqrt(*x*)
x の平方根を精度いっぱいまで求めます。

subtract(*x*, *y*)
x と *y* の間の差を返します。

to_eng_string(*x*)
文字列に変換します。指数が必要なら工学表記が使われます。

工学表記法では指数は 3 の倍数になります。これにより、基数の小数部には最大で 3 桁までの数字が残されるとともに、末尾に 1 つまたは 2 つの 0 の付加が必要とされるかもしれません。

to_integral_exact(*x*)
最近傍の整数に値を丸めます。

to_sci_string(*x*)
数値を科学表記で文字列に変換します。

9.4.4 定数

この節の定数は C モジュールにのみ意味があります。互換性のために、pure Python 版も含まれます。

	32-bit	64-bit
decimal.MAX_PREC	425000000	999999999999999999
decimal.MAX_EMAX	425000000	999999999999999999
decimal.MIN_EMIN	-425000000	-999999999999999999
decimal.MIN_ETINY	-849999999	-1999999999999999997

decimal.HAVE_THREADS
: 値は True です。現在の Python は常にスレッドを持っているので、非推奨になりました。

バージョン 3.9 で非推奨.

decimal.HAVE_CONTEXTVAR
: The default value is True. If Python is compiled --without-decimal-contextvar, the C version uses a thread-local rather than a coroutine-local context and the value is False. This is slightly faster in some nested context scenarios.

バージョン 3.9 で追加: backported to 3.7 and 3.8

9.4.5 丸めモード

decimal.ROUND_CEILING
: Infinity 方向に丸めます。

decimal.ROUND_DOWN
: ゼロ方向に丸めます。

decimal.ROUND_FLOOR
: -Infinity 方向に丸めます。

decimal.ROUND_HALF_DOWN
: 近い方に、引き分けはゼロ方向に向けて丸めます。

decimal.ROUND_HALF_EVEN
: 近い方に、引き分けは偶数整数方向に向けて丸めます。

decimal.ROUND_HALF_UP
: 近い方に、引き分けはゼロから遠い方向に向けて丸めます。

decimal.ROUND_UP
ゼロから遠い方向に丸めます。

decimal.ROUND_05UP
ゼロ方向に丸めた後の最後の桁が 0 または 5 ならばゼロから遠い方向に、そうでなければゼロ方向に丸めます。

9.4.6 シグナル

シグナルは、計算中に生じた様々なエラー条件を表現します。各々のシグナルは一つのコンテキストフラグと一つのトラップイネーブラに対応しています。

コンテキストフラグは、該当するエラー条件に遭遇するたびにセットされます。演算後にフラグを調べれば、演算に関する情報 (例えば計算が厳密だったかどうか) がわかります。フラグを調べたら、次の計算を始める前にフラグを全てクリアするようにしてください。

あるコンテキストのトラップイネーブラがあるシグナルに対してセットされている場合、該当するエラー条件が生じると Python の例外を送出します。例えば、*DivisionByZero* が設定されていると、エラー条件が生じた際に *DivisionByZero* 例外を送出します。

class decimal.Clamped
値の表現上の制限に沿わせるために指数部が変更されたことを通知します。

通常、クランプ (clamp) は、指数部がコンテキストにおける指数桁の制限値 Emin および Emax を越えた場合に発生します。可能な場合には、係数部にゼロを加えた表現に合わせて指数部を減らします。

class decimal.DecimalException
他のシグナルの基底クラスで、*ArithmeticError* のサブクラスです。

class decimal.DivisionByZero
有限値をゼロで除算したときのシグナルです。

除算やモジュロ除算、数を負の値で累乗した場合に起きることがあります。このシグナルをトラップしない場合、演算結果は Infinity または -Infinity になり、その符号は演算に使った入力に基づいて決まります。

class decimal.Inexact
値の丸めによって演算結果から厳密さが失われたことを通知します。

このシグナルは値丸め操作中にゼロでない桁を無視した際に生じます。演算結果は値丸め後の値です。シグナルのフラグやトラップは、演算結果の厳密さが失われたことを検出するために使えるだけです。

class decimal.InvalidOperation
無効な演算が実行されたことを通知します。

ユーザが有意な演算結果にならないような操作を要求したことを示します。このシグナルをトラップしない場合、NaN を返します。このシグナルの発生原因として考えられるのは、以下のような状況です:

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

class decimal.Overflow

数値オーバフローを示すシグナルです。

このシグナルは、値丸めを行った後の指数部が Emax より大きいことを示します。シグナルをトラップしない場合、演算結果は値丸めのモードにより、表現可能な最大の数値になるように内側へ引き込んで丸めを行った値か、Infinity になるように外側に丸めた値のいずれかになります。いずれの場合も、*Inexact* および *Rounded* が同時にシグナルされます。

class decimal.Rounded

情報が全く失われていない場合も含み、値丸めが起きたときのシグナルです。

このシグナルは、値丸めによって桁がなくなると常に発生します。なくなった桁がゼロ (例えば 5.00 を丸めて 5.0 になった場合) であってもです。このシグナルをトラップしなければ、演算結果をそのまま返します。このシグナルは有効桁数の減少を検出する際に使います。

class decimal.Subnormal

値丸めを行う前に指数部が Emin より小さかったことを示すシグナルです。

演算結果が微小である場合 (指数が小さすぎる場合) に発生します。このシグナルをトラップしなければ、演算結果をそのまま返します。

class decimal.Underflow

演算結果が値丸めによってゼロになった場合に生じる数値アンダフローです。

演算結果が微小なため、値丸めによってゼロになった場合に発生します。*Inexact* および *Subnormal* シグナルも同時に発生します。

class decimal.FloatOperation

float と Decimal の混合の厳密なセマンティクスを有効にします。

シグナルがトラップされなかった場合 (デフォルト)、*Decimal* コンストラクタ、*create_decimal()* 、およびすべての比較演算子において float と Decimal の混合が許されます。変換も比較も正確です。コンテキストフラグ内に *FloatOperation* を設定することで、混合操作は現れるたびに暗黙に記録されます。*from_float()* や *create_decimal_from_float()* による明示的な変換はフラグを設定しません。

そうでなければ (シグナルがトラップされれば)、等価性比較および明示的な変換のみが静かにに行われ、その他の混合演算は *FloatOperation* を送出します。

これらのシグナルの階層構造をまとめると、以下の表のようになります:

```
exceptions.ArithmeticError(exceptions.Exception)
    DecimalException
        Clamped
        DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
        Inexact
            Overflow(Inexact, Rounded)
            Underflow(Inexact, Rounded, Subnormal)
        InvalidOperation
        Rounded
        Subnormal
        FloatOperation(DecimalException, exceptions.TypeError)
```

9.4.7 浮動小数点数に関する注意

精度を上げて丸め誤差を抑制する

十進浮動小数点数を使うと、十進数表現による誤差を抑制できます (0.1 を正確に表現できるようになります); しかし、ゼロでない桁が一定の精度を越えている場合には、演算によっては依然として値丸めによる誤差を引き起こします。

値丸めによる誤差の影響は、桁落ちを生じるような、ほとんど相殺される量での加算や減算によって増幅されます。Knuth は、十分でない計算精度の下で値丸めを伴う浮動小数点演算を行った結果、加算の結合則や分配則における恒等性が崩れてしまう例を二つ示しています:

```
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.5111111')
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
>>> u * (v+w)
Decimal('0.0060000')
```

decimal モジュールでは、最下桁を失わないように十分に計算精度を広げることで、上で問題にしたような恒等性をとりもどせます:

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
```

(次のページに続く)

(前のページからの続き)

```
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
>>> u * (v+w)
Decimal('0.0060000')
```

特殊値

decimal モジュールの数体系では、NaN, sNaN, -Infinity, Infinity, および二つのゼロ、+0 と -0 といった特殊な値を提供しています。

無限大 (Infinity) は Decimal('Infinity') で直接構築できます。また、*DivisionByZero* をトラップせずにゼロで除算を行った場合にも出てきます。同様に、*Overflow* シグナルをトラップしなければ、表現可能な最大の数値の制限を越えた値を丸めたときに出てきます。

無限大には符号があり (アフィン: affine であり)、算術演算に使用でき、非常に巨大で不確定の (indeterminate) 値として扱われます。例えば、無限大に何らかの定数を加算すると、演算結果は別の無限大になります。

演算によっては結果が不確定になるものがあり、NaN を返します。ただし、*InvalidOperation* シグナルをトラップするようになっていれば例外を送出します。例えば、0/0 は NaN を返します。NaN は「非数値 (not a number)」を表します。このような NaN は暗黙のうちに生成され、一度生成されるとそれを他の計算にも流れてゆき、関係する個々の演算全てが個別の NaN を返すようになります。この挙動は、たまに入力値が欠けるような状況で一連の計算を行う際に便利です --- 特定の計算に対しては無効な結果を示すフラグを立てつつ計算を進められるからです。

一方、NaN の変種である sNaN は関係する全ての演算で演算後にシグナルを送出します。sNaN は、無効な演算結果に対して特別な処理を行うために計算を停止する必要がある場合に便利です。

Python の比較演算は NaN が関わってくると少し驚くようなことがあります。等価性のテストの一方の対象が無言または発信 NaN である場合いつでも *False* を返し (たとえ Decimal('NaN')==Decimal('NaN') でも)、一方で不等価をテストするといつでも *True* を返します。二つの Decimal を <, <=, > または >= を使って比較する試みは一方が NaN である場合には *InvalidOperation* シグナルを送出し、このシグナルをトラップしなければ結果は *False* に終わります。汎用十進演算仕様は直接の比較の振る舞いについて定めていないことに注意しておきましょう。ここでの NaN が関係する比較ルールは IEEE 854 標準から持ってきました (section 5.7 の Table 3 を見て下さい)。厳格に標準遵守を貫くなら、compare() および compare-signal() メソッドを代わりに使いましょう。

アンダフローの起きた計算は、符号付きのゼロ (signed zero) を返すことがあります。符号は、より高い精度で計算を行った結果の符号と同じになります。符号付きゼロの大きさはやはりゼロなので、正のゼロと負のゼロは等しいとみなされ、符号は単なる参考にすぎません。

二つの符号付きゼロが区別されているのに等価であることに加えて、異なる精度におけるゼロの表現はまちまちなのに、値は等価とみなされるということがあります。これに慣れるには多少時間がかかります。正規化浮動小数点表現に目が慣れてしまうと、以下の計算でゼロに等しい値が返っているとは即座に分かりません:

```
>>> 1 / Decimal('Infinity')
Decimal('0E-1000026')
```

9.4.8 スレッドを使った処理

関数 *getcontext()* は、スレッド毎に別々の *Context* オブジェクトにアクセスします。別のスレッドコンテキストを持つということは、複数のスレッドが互いに影響を及ぼさずに (getcontext().prec=10 のような) 変更を適用できるということです。

同様に、*setcontext()* 関数は自動的に引数のコンテキストを現在のスレッドのコンテキストに設定します。

getcontext() を呼び出す前に *setcontext()* が呼び出されていなければ、現在のスレッドで使うための新たなコンテキストを生成するために *getcontext()* が自動的に呼び出されます。

新たなコンテキストは、*DefaultContext* と呼ばれる雛形からコピーされます。アプリケーションを通じて全てのスレッドに同じ値を使うようにデフォルトを設定したければ、*DefaultContext* オブジェクトを直接変更します。*getcontext()* を呼び出すスレッド間で競合条件が生じないようにするため、*DefaultContext* への変更はいかなるスレッドを開始するよりも **前に** 行わなければなりません。以下に例を示します:

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
 . . .
```

9.4.9 レシピ

Decimal クラスの利用を実演している例をいくつか示します。これらはユーティリティ関数としても利用できます:

```
def moneyfmt(value, places=2, curr='', sep=',', dp='.',
             pos='', neg='-', trailneg=''):
    """Convert Decimal to a money formatted string.

    places:  required number of places after the decimal point
    curr:    optional currency symbol before the sign (may be blank)
    sep:     optional grouping separator (comma, period, space, or blank)
    dp:      decimal point indicator (comma or period)
             only specify as blank when places is zero
    pos:     optional sign for positive numbers: '+', space or blank
```

(次のページに続く)

(前のページからの続き)

```
    neg:     optional sign for negative numbers: '-', '(', space or blank
    trailneg:optional trailing minus indicator:  '-', ')', space or blank

    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
    '-$1,234,567.89'
    >>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
    '1.234.568-'
    >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
    '($1,234,567.89)'
    >>> moneyfmt(Decimal(123456789), sep=' ')
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
    '<0.02>'

    """
    q = Decimal(10) ** -places      # 2 places --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = list(map(str, digits))
    build, next = result.append, digits.pop
    if sign:
        build(trailneg)
    for i in range(places):
        build(next() if digits else '0')
    if places:
        build(dp)
    if not digits:
        build('0')
    i = 0
    while digits:
        build(next())
        i += 1
        if i == 3 and digits:
            i = 0
            build(sep)
    build(curr)
    build(neg if sign else pos)
    return ''.join(reversed(result))

def pi():
    """Compute Pi to the current precision.

    >>> print(pi())
    3.141592653589793238462643383

    """
    getcontext().prec += 2  # extra digits for intermediate steps
    three = Decimal(3)      # substitute "three=3.0" for regular floats
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
```

(次のページに続く)

(前のページからの続き)

```
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2
    return +s               # unary plus applies the new precision

def exp(x):
    """Return e raised to the power of x.  Result type matches input type.

    >>> print(exp(Decimal(1)))
    2.718281828459045235360287471
    >>> print(exp(Decimal(2)))
    7.389056098930650227230427461
    >>> print(exp(2.0))
    7.38905609893
    >>> print(exp(2+0j))
    (7.38905609893+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 1
        fact *= i
        num *= x
        s += num / fact
    getcontext().prec -= 2
    return +s

def cos(x):
    """Return the cosine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(cos(Decimal('0.5')))
    0.8775825618903727161162815826
    >>> print(cos(0.5))
    0.87758256189
    >>> print(cos(0.5+0j))
    (0.87758256189+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 2
```

(次のページに続く)

(前のページからの続き)

```
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

def sin(x):
    """Return the sine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(sin(Decimal('0.5')))
    0.47942553860420300027328793521
    >>> print(sin(0.5))
    0.479425538604
    >>> print(sin(0.5+0j))
    (0.479425538604+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s
```

9.4.10 Decimal FAQ

Q. `decimal.Decimal('1234.5')` などと打ち込むのは煩わしいのですが、対話式インタプリタを使う際にタイプ量を少なくする方法はありませんか?

A. コンストラクタを 1 文字に縮める人もいるようです:

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

Q. 小数点以下 2 桁の固定小数点数のアプリケーションの中で、いくつかの入力が余計な桁を保持しているのでこれを丸めなければなりません。その他のものに余計な桁はなくそのまま使えます。どのメソッドを使うのがいいでしょうか?

A. `quantize()` メソッドで固定した桁に丸められます。*Inexact* トラップを設定しておけば、確認にも有用

です:

```
>>> TWOPLACES = Decimal(10) ** -2       # same as Decimal('0.01')
```

```
>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')
```

```
>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')
```

```
>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
   ...
Inexact: None
```

Q. 正当な 2 桁の入力が得られたとして、その正当性をアプリケーション実行中も変わらず保ち続けるにはどうすればいいでしょうか?

A. 加減算あるいは整数との乗算のような演算は自動的に固定小数点を守ります。その他の除算や整数以外の乗算などは小数点以下の桁を変えてしまいますので実行後は quantize() ステップが必要です:

```
>>> a = Decimal('102.72')           # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                           # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42                          # So does integer multiplication
Decimal('4314.24')
>>> (a * b).quantize(TWOPLACES)     # Must quantize non-integer multiplication
Decimal('325.62')
>>> (b / a).quantize(TWOPLACES)     # And quantize division
Decimal('0.03')
```

固定小数点のアプリケーションを開発する際は、quantize() の段階を扱う関数を定義しておくと便利です:

```
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)
```

```
>>> mul(a, b)                       # Automatically preserve fixed-point
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')
```

Q. 一つの値に対して多くの表現方法があります。200 と 200.000 と 2E2 と 02E+4 は全て同じ値で違った精度の数です。これらをただ一つの正規化された値に変換することはできますか?

A. `normalize()` メソッドは全ての等しい値をただ一つの表現に直します:

```
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

Q. ある種の十進数値はいつも指数表記で表示されます。指数表記以外の表示にする方法はありますか?

A. 値によっては、指数表記だけが有効桁数を表せる表記法なのです。たとえば、`5.0E+3` を `5000` と表してしまうと、値は変わりませんが元々の 2 桁という有効数字が反映されません。

もしアプリケーションが有効数字の追跡を等閑視するならば、指数部や末尾のゼロを取り除き、有効数字を忘れ、しかし値を変えずにおくことは容易です:

```
>>> def remove_exponent(d):
...     return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()
```

```
>>> remove_exponent(Decimal('5E+3'))
Decimal('5000')
```

Q. 普通の float を *Decimal* に変換できますか?

A. はい。どんな 2 進浮動小数点数も Decimal として正確に表現できます。ただし、正確な変換は直感的に考えたよりも多い桁になることがあります:

```
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

Q. 複雑な計算の中で、精度不足や丸めの異常で間違った結果になっていないことをどうやって保証すれば良いでしょうか。

A. decimal モジュールでは検算は容易です。一番良い方法は、大きめの精度や様々な丸めモードで再計算してみることです。大きく異なった結果が出てきたら、精度不足や丸めの問題や悪条件の入力、または数値計算的に不安定なアルゴリズムを示唆しています。

Q. コンテキストの精度は計算結果には適用されていますが入力には適用されていないようです。様々に異なる精度の入力値を混ぜて計算する時に注意すべきことはありますか?

A. はい。原則として入力値は正確であると見做しておりそれらの値を使った計算も同様です。結果だけが丸められます。入力の強みは "what you type is what you get" (打ち込んだ値が得られる値) という点にあります。入力が丸められないということを忘れていると結果が奇妙に見えるというのは弱点です:

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

解決策は、精度を増やすか、単項プラス演算子を使って入力の丸めを強制することです:

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')      # unary plus triggers rounding
Decimal('1.23')
```

もしくは、入力を *Context.create_decimal()* を使って生成時に丸めてしまうこともできます:

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

Q. CPython 実装は大きな数に対しても速いでしょうか?

A. はい。CPython 実装と PyPy3 実装では、C/CFFI 版の decimal モジュールは、任意精度の正しい丸めを行う 10 進浮動小数点演算のための高速な libmpdec ライブラリを統合しています。`libmpdec` は Karatsuba multiplication を中程度のサイズの数に対して使い、Number Theoretic Transform を非常に大きな数に対して使います。ただし、このパフォーマンス向上を得るためには、計算で丸めが発生しないように context を設定する必要があります。

```
>>> c = getcontext()
>>> c.prec = MAX_PREC
>>> c.Emax = MAX_EMAX
>>> c.Emin = MIN_EMIN
```

バージョン 3.3 で追加.

9.5 fractions --- 有理数

ソースコード: Lib/fractions.py

---

fractions モジュールは有理数計算のサポートを提供します。

Fraction インスタンスは一対の整数、他の有理数、または文字列から生成されます。

class fractions.Fraction(*numerator=0*, *denominator=1*)
class fractions.Fraction(*other_fraction*)
class fractions.Fraction(*float*)
class fractions.Fraction(*decimal*)
class fractions.Fraction(*string*)

最初のバージョンは *numerator* と *denominator* が *numbers.Rational* のインスタンスであることを要求し、`numerator/denominator` の値を持つ新しい *Fraction* インスタンスを返します。*denominator* が `0` ならば、*ZeroDivisionError* を送出します。二番目のバージョンは *other_fraction* が *numbers.Rational* のインスタンスであることを要求し、同じ値を持つ新しい *Fraction* インスタンスを返します。その次の二つのバージョンは、*float* と *decimal.Decimal* インスタンスを受け付け、それとちょうど同じ値を持つ *Fraction* インスタンスを返します。なお、二進浮動小数点数にお決まりの問題 (tut-fp-issues 参照) のため、`Fraction(1.1)` の引数は 11/10 と正確に等しいとは言えないので、`Fraction(1.1)` は予期した通りの `Fraction(11, 10)` を返し **ません** 。(ただし、以下の

limit_denominator() メソッドのドキュメントを参照してください。) 最後のバージョンは、文字列またはユニコードのインスタンスを渡されることを想定します。このインスタンスは、通常、次のような形式です:

```
[sign] numerator ['/' denominator]
```

ここで、オプションの sign は '+' か '-' のどちらかであり、numerator および (存在する場合) denominator は十進数の数字の文字列です。さらに、*float* コンストラクタで受け付けられる有限の値を表す文字列は、*Fraction* コンストラクタでも受け付けられます。どちらの形式でも、入力される文字列は前後に空白があって構いません。以下に、いくつかの例を示します:

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)
```

Fraction クラスは抽象基底クラス *numbers.Rational* を継承し、その全てのメソッドと演算を実装します。*Fraction* インスタンスはハッシュ可能で、不変 (immutable) であるものとして扱われます。加えて、*Fraction* には以下のプロパティとメソッドがあります:

バージョン 3.2 で変更: *Fraction* のコンストラクタが *float* および *decimal.Decimal* インスタンスを受け付けるようになりました。

numerator

有理数を既約分数で表したときの分子。

denominator

有理数を既約分数で表したときの分母。

as_integer_ratio()

2 つの整数からなるタプルで、比が Fraction インスタンスと等しく、分母が正になるものを返し

ます。

バージョン 3.8 で追加.

from_float(*flt*)

このクラスメソッドは *float* である *flt* の正確な値を表す *Fraction* を構築します。Fraction.from_float(0.3) と Fraction(3, 10) の値は同じでないことに注意してください 。

注釈: Python 3.2 以降では、*float* から直接 *Fraction* インスタンスを構築できるようになりました。

---

from_decimal(*dec*)

このクラスメソッドは *decimal.Decimal* インスタンスである *dec* の正確な値を表す *Fraction* を構築します。

注釈: Python 3.2 以降では、*decimal.Decimal* インスタンスから直接 *Fraction* インスタンスを構築できるようになりました。

---

limit_denominator(*max_denominator=1000000*)

分母が高々 max_denominator である、self に最も近い *Fraction* を見付けて返します。このメソッドは与えられた浮動小数点数の有理数近似を見つけるのに役立ちます:

```
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

あるいは float で表された有理数を元に戻すのにも使えます:

```
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
Fraction(4503599627370497, 9007199254740992)
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
>>> Fraction(1.1).limit_denominator()
Fraction(11, 10)
```

__floor__()

最大の *int* <= self を返します。このメソッドは *math.floor()* 関数からでもアクセスできます:

```
>>> from math import floor
>>> floor(Fraction(355, 113))
3
```

__ceil__()

最小の *int* >= self を返します。このメソッドは *math.ceil()* 関数からでもアクセスできます。

__round__()
__round__(*ndigits*)
第一のバージョンは、self に最も近い *int* を偶数丸めで返します。第二のバージョンは、このメソッドは self に最も近い Fraction(1, 10**ndigits) の倍数 (論理的に、ndigits が負なら) を、これも偶数丸めで丸めます。*round()* 関数からでもアクセスできます。

fractions.gcd(*a*, *b*)
整数 a と b の最大公約数を返します。a も b もゼロでないとすると、gcd(a, b) の絶対値は a と b の両方を割り切る最も大きな整数です。gcd(a, b) は b がゼロでなければ b と同じ符号になります。そうでなければ a の符号を取ります。gcd(0, 0) は 0 を返します。

バージョン 3.5 で非推奨: 代わりに *math.gcd()* を使用してください。

参考:

numbers **モジュール**　数値の塔を作り上げる抽象基底クラス。

9.6 random --- 擬似乱数を生成する

ソースコード: Lib/random.py

このモジュールでは様々な分布をもつ擬似乱数生成器を実装しています。

整数用に、ある範囲からの一様な選択があります。シーケンス用には、シーケンスからのランダムな要素の一様な選択、リストのランダムな置換をインプレースに生成する関数、順列を置換せずにランダムサンプリングする関数があります。

実数用としては、一様分布、正規分布 (ガウス分布)、対数正規分布、負の指数分布、ガンマおよびベータ分布を計算する関数があります。角度の分布を生成するにはフォン・ミーゼス分布が利用できます。

ほとんど全てのモジュール関数は、基礎となる関数 *random()* に依存します。この関数はランダムな浮動小数点数を半開区間 [0.0, 1.0) 内に一様に生成します。Python は中心となる乱数生成器としてメルセンヌツイスタを使います。これは 53 ビット精度の浮動小数点を生成し、周期は 2**19937-1 です。本体は C で実装されていて、高速でスレッドセーフです。メルセンヌツイスタは、現存する中で最も広範囲にテストされた乱数生成器のひとつです。しかしながら、メルセンヌツイスタは完全に決定論的であるため、全ての目的に合致しているわけではなく、暗号化の目的には全く向いていません。

このモジュールで提供されている関数は、実際には *random.Random* クラスの隠蔽されたインスタンスのメソッドに束縛されています。内部状態を共有しない生成器を取得するため、自分で *Random* のインスタンスを生成することができます。

自分で考案した基本乱数生成器を使いたい場合、クラス *Random* をサブクラス化することもできます。この場合、メソッド random()、seed()、getstate()、setstate() をオーバライドしてください。オプションとして、新しいジェネレータは getrandbits() メソッドを提供することができます。これにより *randrange()* メソッドが任意に大きな範囲から選択を行えるようになります。

random モジュールは *SystemRandom* クラスも提供していて、このクラスは OS が提供している乱数発生源を利用して乱数を生成するシステム関数 *os.urandom()* を使うものです。

警告: このモジュールの擬似乱数生成器をセキュリティ目的に使用してはいけません。セキュリティや暗号学的な用途については *secrets* モジュールを参照してください。

参考:

M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3--30 1998.

Complementary-Multiply-with-Carry recipe 長い周期と比較的シンプルな更新操作を備えた互換性のある別の乱数生成器。

9.6.1 保守 (bookkeeping) 関数

random.seed(*a=None*, *version=2*)

乱数生成器を初期化します。

a が省略されるか None の場合、現在のシステム時刻が使用されます。乱数のソースがオペレーティングシステムによって提供される場合、システム時刻の代わりにそれが使用されます (利用可能性についての詳細は *os.urandom()* 関数を参照)。

a が int の場合、それが直接使われます。

バージョン 2 (デフォルト) では、*str*, *bytes*, *bytearray* オブジェクトは *int* に変換され、そのビットがすべて使用されます。

バージョン 1 (Python の古いバージョンでのランダムなシーケンスを再現するために提供される) では、*str* と *bytes* に対して適用されるアルゴリズムは、より狭い範囲のシードを生成します。

バージョン 3.2 で変更: 文字列シードのすべてのビットを使うバージョン 2 スキームに移行。

random.getstate()

乱数生成器の現在の内部状態を記憶したオブジェクトを返します。このオブジェクトを *setstate()* に渡して内部状態を復元することができます。

random.setstate(*state*)

state は予め *getstate()* を呼び出して得ておかなくてはなりません。*setstate()* は *getstate()* が呼び出された時の乱数生成器の内部状態を復元します。

random.getrandbits(*k*)

Returns a Python integer with *k* random bits. This method is supplied with the Mersenne Twister generator and some other generators may also provide it as an optional part of the API. When available, *getrandbits()* enables *randrange()* to handle arbitrarily large ranges.

9.6.2 整数用の関数

random.**randrange**(*stop*)
random.**randrange**(*start*, *stop*[, *step*])

range(start, stop, step) の要素からランダムに選ばれた要素を返します。この関数は choice(range(start, stop, step)) と等価ですが、実際には range オブジェクトを生成しません。

位置引数のパターンは *range()* のそれと一致します。キーワード引数は、この関数に望まれない方法で使われるかもしれないので、使うべきではありません。

バージョン 3.2 で変更: 一様に分布した値の生成に関して *randrange()* がより洗練されました。以前は int(random()*n) のようなやや一様でない分布を生成するスタイルを使用していました。

random.**randint**(*a*, *b*)

a <= N <= b であるようなランダムな整数 *N* を返します。randrange(a, b+1) のエイリアスです。

9.6.3 シーケンス用の関数

random.**choice**(*seq*)

空でないシーケンス *seq* からランダムに要素を返します。*seq* が空のときは、*IndexError* が送出されます。

random.**choices**(*population*, *weights=None*, ***, *cum_weights=None*, *k=1*)

population から重複ありで選んだ要素からなる大きさ *k* のリストを返します。*population* が空の場合 *IndexError* を送出します。

weights シーケンスが与えられた場合、相対的な重みに基づいて要素が選ばれます。あるいは、*cum_weights* シーケンスが与えられた場合、累積的な重み (*itertools.accumulate()* を用いて計算されるかもしれません) で要素が選ばれます。例えば、相対的な重み [10, 5, 30, 5] は累積的な重み [10, 15, 45, 50] と等価です。内部的には、相対的な重みは要素選択の前に累積的な重みに変換されるため、累積的な重みを渡すと手間を省けます。

weights および *cum_weights* が与えられなかった場合、要素は同じ確率で選択されます。重みのシーケンスが与えられた場合、その長さは *population* シーケンスと同じでなければなりません。*weights* と *cum_weights* を同時に与えると *TypeError* が送出されます。

weights や *cum_weights* には *random()* が返す *float* と相互に変換できるような、あらゆる型を使用できます (int、float、fraction を含みますが、decimal は除きます)。重みは非負だと想定されています。

与えられた種に対して、同じ重みを持つ *choices()* 関数は、一般に *choice()* を繰り返し呼び出す場合とは異なるシーケンスを生成します。*choices()* で使用されるアルゴリズムは、内部の一貫性とスピードのために浮動小数点演算を使用します。*choice()* で使われるアルゴリズムは、丸め誤差による小さな偏りを避けるために、デフォルトでは選択を繰り返す整数演算になっています。

バージョン 3.6 で追加.

random.**shuffle**(*x*[, *random*])

シーケンス *x* をインプレースにシャッフルします。

オプション引数 *random* は [0.0, 1.0) の範囲のランダムな浮動小数を返す引数なしの関数です。デフォルトでは *random()* 関数です。

イミュータブルなシーケンスをシャッフルしてシャッフルされたリストを新たに返すには、代わりに `sample(x, k=len(x))` を使用してください。

たとえ `len(x)` が小さくても、*x* の並べ替えの総数 (訳注: 要素数の階乗) は大半の乱数生成器の周期よりもすぐに大きくなることに注意してください。つまり、長いシーケンスの大半の並べ替えは決して生成されないだろう、ということです。例えば、長さ 2080 のシーケンスがメルセンヌツイスタ生成器の周期に収まる中で最大のものになります。

random.**sample**(*population*, *k*)

母集団のシーケンスまたは集合から選ばれた長さ *k* の一意な要素からなるリストを返します。重複無しのランダムサンプリングに用いられます。

母集団自体を変更せずに、母集団内の要素を含む新たなリストを返します。返されたリストは選択された順に並んでいるので、このリストの部分スライスもランダムなサンプルになります。これにより、くじの当選者 (サンプル) を 1 等賞と 2 等賞（の部分スライス）に分けることも可能です。

母集団の要素は **ハッシュ可能** でなくても、ユニークでなくてもかまいません。母集団が繰り返しを含む場合、出現するそれぞれがサンプルに選択されえます。

ある範囲の整数からサンプルを取る場合、引数に *range()* オブジェクトを使用してください。大きな母集団の場合、これは特に速く、メモリ効率が良いです: `sample(range(10000000), k=60)`。

サンプルの大きさが母集団の大きさより大きい場合 *ValueError* が送出されます。

9.6.4 実数分布

以下の関数は特定の実数値分布を生成します。関数の引数の名前は、一般的な数学の慣例で使われている分布の公式の対応する変数から取られています; これらの公式のほとんどはどんな統計学のテキストにも載っています。

random.**random**()

次のランダムな浮動小数点数（範囲は [0.0, 1.0) ）を返します。

random.**uniform**(*a*, *b*)

`a <= b` であれば `a <= N <= b` 、`b < a` であれば `b <= N <= a` であるようなランダムな浮動小数点数 *N* を返します。

端点の値 `b` が範囲に含まれるかどうかは、等式 `a + (b-a) * random()` における浮動小数点の丸めに依存します。

random.**triangular**(*low*, *high*, *mode*)

`low <= N <= high` でありこれら境界値の間に指定された最頻値 *mode* を持つようなランダムな浮動

小数点数 *N* を返します。境界 *low* と *high* のデフォルトは 0 と 1 です。最頻値 *mode* 引数のデフォルトは両境界値の中点になり、対称な分布を与えます。

random.betavariate(*alpha*, *beta*)

ベータ分布です。引数の満たすべき条件は `alpha > 0` および `beta > 0` です。0 から 1 の範囲の値を返します。

random.expovariate(*lambd*)

指数分布です。*lambd* は平均にしたい値の逆数です。(この引数は "lambda" と呼ぶべきなのですが、Python の予約語なので使えません。) 返す値の範囲は *lambd* が正なら 0 から正の無限大、*lambd* が負なら負の無限大から 0 です。

random.gammavariate(*alpha*, *beta*)

ガンマ分布です (ガンマ関数 **ではありません** ！)。引数の満たすべき条件は `alpha > 0` および `beta > 0` です。

確率分布関数は:

```
          x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) =  --------------------------------------
            math.gamma(alpha) * beta ** alpha
```

random.gauss(*mu*, *sigma*)

ガウス分布です。*mu* は平均であり、*sigma* は標準偏差です。この関数は後で定義する関数 *normalvariate()* より少しだけ高速です。

random.lognormvariate(*mu*, *sigma*)

対数正規分布です。この分布を自然対数を用いた分布にした場合、平均 *mu* で標準偏差 *sigma* の正規分布になります。*mu* は任意の値を取ることができ、*sigma* はゼロより大きくなければなりません。

random.normalvariate(*mu*, *sigma*)

正規分布です。*mu* は平均で、*sigma* は標準偏差です。

random.vonmisesvariate(*mu*, *kappa*)

mu は平均の角度で、0 から 2**pi* までのラジアンで表されます。*kappa* は濃度パラメータで、ゼロ以上でなければなりません。*kappa* がゼロに等しい場合、この分布は範囲 0 から 2**pi* の一様でランダムな角度の分布に退化します。

random.paretovariate(*alpha*)

パレート分布です。*alpha* は形状パラメータです。

random.weibullvariate(*alpha*, *beta*)

ワイブル分布です。*alpha* は尺度パラメタで、*beta* は形状パラメータです。

9.6.5 他の生成器

class random.Random([seed])
: デフォルトの疑似乱数生成器を *random* を使って実装したクラスです。

class random.SystemRandom([seed])
: オペレーティングシステムの提供する発生源によって乱数を生成する *os.urandom()* 関数を使うクラスです。すべてのシステムで使えるメソッドではありません。ソフトウェアの状態に依存してはいけませんし、一連の操作は再現不能です。従って、*seed()* メソッドは何の影響も及ぼさず、無視されます。*getstate()* と *setstate()* メソッドが呼び出されると、例外 *NotImplementedError* が送出されます。

9.6.6 再現性について

疑似乱数生成器から与えられたシーケンスを再現できると便利なことがあります。シード値を再利用することで、複数のスレッドが実行されていない限り、実行ごとに同じシーケンスが再現できます。

random モジュールのアルゴリズムやシード処理関数のほとんどは、Python バージョン間で変更される対象となりますが、次の二点は変更されないことが保証されています:

- 新しいシード処理メソッドが追加されたら、後方互換なシード処理器が提供されます。
- 生成器の `random()` メソッドは、互換なシード処理器に同じシードが与えられた場合、引き続き同じシーケンスを生成します。

9.6.7 例とレシピ

基礎的な例:

```
>>> random()                            # Random float:  0.0 <= x < 1.0
0.37444887175646646

>>> uniform(2.5, 10.0)                  # Random float:  2.5 <= x < 10.0
3.1800146073117523

>>> expovariate(1 / 5)                  # Interval between arrivals averaging 5 seconds
5.148957571865031

>>> randrange(10)                       # Integer from 0 to 9 inclusive
7

>>> randrange(0, 101, 2)                # Even integer from 0 to 100 inclusive
26

>>> choice(['win', 'lose', 'draw'])     # Single random element from a sequence
'draw'
```

(次のページに続く)

(前のページからの続き)

```
>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                          # Shuffle a list
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4)     # Four samples without replacement
[40, 10, 50, 30]
```

シミュレーション:

```
>>> # Six roulette wheel spins (weighted sampling with replacement)
>>> choices(['red', 'black', 'green'], [18, 18, 2], k=6)
['red', 'green', 'black', 'black', 'red', 'black']

>>> # Deal 20 cards without replacement from a deck of 52 playing cards
>>> # and determine the proportion of cards with a ten-value
>>> # (a ten, jack, queen, or king).
>>> deck = collections.Counter(tens=16, low_cards=36)
>>> seen = sample(list(deck.elements()), k=20)
>>> seen.count('tens') / 20
0.15

>>> # Estimate the probability of getting 5 or more heads from 7 spins
>>> # of a biased coin that settles on heads 60% of the time.
>>> def trial():
...     return choices('HT', cum_weights=(0.60, 1.00), k=7).count('H') >= 5
...
>>> sum(trial() for i in range(10_000)) / 10_000
0.4169

>>> # Probability of the median of 5 samples being in middle two quartiles
>>> def trial():
...     return 2_500 <= sorted(choices(range(10_000), k=5))[2] < 7_500
...
>>> sum(trial() for i in range(10_000)) / 10_000
0.7958
```

サンプルの平均の信頼区間を推定するのに、重複ありでリサンプリングして ‘統計的ブートストラップ‘_ を行う例:

```
# http://statistics.about.com/od/Applications/a/Example-Of-Bootstrapping.htm
from statistics import fmean as mean
from random import choices

data = [41, 50, 29, 37, 81, 30, 73, 63, 20, 35, 68, 22, 60, 31, 95]
means = sorted(mean(choices(data, k=len(data))) for i in range(100))
print(f'The sample mean of {mean(data):.1f} has a 90% confidence '
      f'interval from {means[5]:.1f} to {means[94]:.1f}')
```

薬と偽薬の間に観察された効果の違いについて、統計的有意性、すなわち p 値 を決定するために、リサンプリング順列試験 を行う例:

```
# Example from "Statistics is Easy" by Dennis Shasha and Manda Wilson
from statistics import fmean as mean
from random import shuffle

drug = [54, 73, 53, 70, 73, 68, 52, 65, 65]
placebo = [54, 51, 58, 44, 55, 52, 42, 47, 58, 46]
observed_diff = mean(drug) - mean(placebo)

n = 10_000
count = 0
combined = drug + placebo
for i in range(n):
    shuffle(combined)
    new_diff = mean(combined[:len(drug)]) - mean(combined[len(drug):])
    count += (new_diff >= observed_diff)

print(f'{n} label reshufflings produced only {count} instances with a difference')
print(f'at least as extreme as the observed difference of {observed_diff:.1f}.')
print(f'The one-sided p-value of {count / n:.4f} leads us to reject the null')
print(f'hypothesis that there is no difference between the drug and the placebo.')
```

マルチサーバーキューにおける到達時間とサービス提供のシミュレーション:

```
from heapq import heappush, heappop
from random import expovariate, gauss
from statistics import mean, median, stdev

average_arrival_interval = 5.6
average_service_time = 15.0
stdev_service_time = 3.5
num_servers = 3

waits = []
arrival_time = 0.0
servers = [0.0] * num_servers  # time when each server becomes available
for i in range(100_000):
    arrival_time += expovariate(1.0 / average_arrival_interval)
    next_server_available = heappop(servers)
    wait = max(0.0, next_server_available - arrival_time)
    waits.append(wait)
    service_duration = gauss(average_service_time, stdev_service_time)
    service_completed = arrival_time + wait + service_duration
    heappush(servers, service_completed)

print(f'Mean wait: {mean(waits):.1f}.  Stdev wait: {stdev(waits):.1f}.')
print(f'Median wait: {median(waits):.1f}.  Max wait: {max(waits):.1f}.')
```

参考:

Statistics for Hackers Jake Vanderplas による統計解析のビデオ。シミュレーション、サンプリング、シャッフル、交差検定といった基本的な概念のみを用いています。

Economics Simulation Peter Norvig による市場価格のシミュレーション。このモジュールが提供する多くのツールや分布 (gauss, uniform, sample, betavariate, choice, triangular, randrange) の活用法を示しています。

A Concrete Introduction to Probability (using Python) Peter Norvig によるチュートリアル。確率論の基礎、シミュレーションの書き方、Python を使用したデータ解析法をカバーしています。

9.7 statistics --- 数理統計関数

バージョン 3.4 で追加.

ソースコード: Lib/statistics.py

このモジュールは、数値 (*Real* 型) データを数学的に統計計算するための関数を提供します。

The module is not intended to be a competitor to third-party libraries such as NumPy, SciPy, or proprietary full-featured statistics packages aimed at professional statisticians such as Minitab, SAS and Matlab. It is aimed at the level of graphing and scientific calculators.

特に明記しない限り、これらの関数は *int*, *float*, *Decimal* そして *Fraction* をサポートします。他の型 (算術型及びそれ以外) は現在サポートされていません。型が混ざったコレクションも未定義で実装依存です。入力データが複数の型からなる場合、*map()* を使用すると正しい結果が得られるでしょう。例: `map(float, input_data)`。

9.7.1 平均及び中心位置の測度

これらの関数は母集団または標本の平均値や標準値を計算します。

mean()	データの算術平均。
fmean()	Fast, floating point arithmetic mean.
geometric_mean()	データの幾何平均。
harmonic_mean()	データの調和平均。
median()	データの中央値。
median_low()	データの low median。
median_high()	データの high median。
median_grouped()	grouped data の中央値、すなわち 50 パーセンタイル。
mode()	離散/名義尺度データの最頻値 (single mode) 。
multimode()	List of modes (most common values) of discrete or nomimal data.
quantiles()	データの等確率での分割。

9.7.2 分散の測度

これらの関数は母集団または標本が標準値や平均値からどれくらい離れているかについて計算します。

pstdev()	データの母標準偏差。
pvariance()	データの母分散。
stdev()	データの標本標準偏差。
variance()	データの標本標準分散。

9.7.3 関数の詳細

註釈: 関数の引数となるデータをソートしておく必要はありません。例の多くがソートされているのは見やすさのためです。

statistics.mean(*data*)

シーケンス型またはイテラブルになり得る *data* の標本算術平均を返します。

算術平均はデータの総和をデータ数で除したものです。単に「平均」と呼ばれることも多いですが、数学における平均の一種に過ぎません。データの中心位置の測度の一つです。

data が空の場合 *StatisticsError* を送出します。

使用例:

```
>>> mean([1, 2, 3, 4, 4])
2.8
>>> mean([-1.0, 2.5, 3.25, 5.75])
2.625

>>> from fractions import Fraction as F
>>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
Fraction(13, 21)

>>> from decimal import Decimal as D
>>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
Decimal('0.5625')
```

注釈: 平均値は外れ値に大きく影響を受けるため、中心位置のロバストな推定量ではありません。すなわち、平均値はデータ点の代表例では必ずしもありません。よりロバストな中心位置の測度には *median()* および *mode()* があります。

標本平均は真の母平均の不偏推定量のため、出来る限り多くの標本から平均を求めると、mean(sample) は真の母平均に収束します (訳注: 大数の法則)。*data* が標本ではなく母集団全体の場合、mean(data) は真の母平均 μ を計算することと等価です。

---

statistics.**fmean**(*data*)

data を float に変換し、算術平均を計算します。

This runs faster than the *mean()* function and it always returns a *float*. The *data* may be a sequence or iterable. If the input dataset is empty, raises a *StatisticsError*.

```
>>> fmean([3.5, 4.0, 5.25])
4.25
```

バージョン 3.8 で追加.

statistics.**geometric_mean**(*data*)

Convert *data* to floats and compute the geometric mean.

The geometric mean indicates the central tendency or typical value of the *data* using the product of the values (as opposed to the arithmetic mean which uses their sum).

Raises a *StatisticsError* if the input dataset is empty, if it contains a zero, or if it contains a negative value. The *data* may be a sequence or iterable.

No special efforts are made to achieve exact results. (However, this may change in the future.)

```
>>> round(geometric_mean([54, 24, 36]), 1)
36.0
```

バージョン 3.8 で追加.

statistics.**harmonic_mean**(*data*)

実数値のシーケンス型またはイテラブルの *data* の調和平均を返します。

調和平均（harmonic mean, subcontrary mean）は、データの逆数の算術平均の逆数です。例えば、3 つの値 *a*, *b*, *c* の調和平均は “3/(1/a + 1/b + 1/c)“ になります。いずれかの値が 0 の場合、結果は 0 になります。

調和平均は平均の一種で、データの中心位置の測度です。速度のような率 (rates) や比 (ratios) を平均するときにしばしば適切です。

Suppose a car travels 10 km at 40 km/hr, then another 10 km at 60 km/hr. What is the average speed?

```
>>> harmonic_mean([40, 60])
48.0
```

投資家が P / E（価格/収益）の比率が 2.5,3,10 という 3 つの会社のそれぞれに等しい価値の株式を購入したとします。投資家のポートフォリオの平均 P / E の比率はいくつでしょうか？

```
>>> harmonic_mean([2.5, 3, 10])  # For an equal investment portfolio.
3.6
```

もし *data* が空の場合、またはいずれの要素が 0 より小さい場合、例外 *StatisticsError* が送出されます。

The current algorithm has an early-out when it encounters a zero in the input. This means that the subsequent inputs are not tested for validity. (This behavior may change in the future.)

バージョン 3.6 で追加.

statistics.**median**(*data*)

一般的な「中央 2 つの平均をとる」方法を使用して、数値データの中央値（中間値）を返します。もし *data* が空の場合、例外 *StatisticsError* が送出されます。*data* はシーケンス型またはイテラブルにもなれます。

中央値は外れ値の影響を受けにくいため、中心位置のロバストな測度です。データ数が奇数の場合は中央の値を返します:

```
>>> median([1, 3, 5])
3
```

データ数が偶数の場合は、中央値は中央に最も近い 2 値の算術平均で補間されます:

```
>>> median([1, 3, 5, 7])
4.0
```

データが離散的で、中央値がデータ点にない値でも構わない場合に適しています。

もしあなたのデータが（注文操作をサポートする）序数で、（追加操作をサポートしない）数値でないならば、代わりに *median_low()* または *median_high()* の使用を検討してください。

statistics.**median_low**(*data*)

数値データの小さい方の中央値 (low median) を返します。もし *data* が空の場合、:exc:‘StatisticsError‘が送出されます。*data* はシーケンス型またはイテラブルにもなれます。

low median は必ずデータに含まれています。データ数が奇数の場合は中央の値を返し、偶数の場合は中央の 2 値の小さい方を返します。

```
>>> median_low([1, 3, 5])
3
>>> median_low([1, 3, 5, 7])
3
```

データが離散的で、中央値が補間値よりもデータ点にある値の方が良い場合に用いてください。

statistics.**median_high**(*data*)

データの大きい方の中央値 (high median) を返します。もし *data* が空の場合、*StatisticsError* が送出されます。*data* はシーケンス型やイテラブルにもなれます。

high median は必ずデータに含まれています。データ数が奇数の場合は中央の値を返し、偶数の場合は中央の 2 値の大きい方を返します。

```
>>> median_high([1, 3, 5])
3
```

(次のページに続く)

(前のページからの続き)

```
>>> median_high([1, 3, 5, 7])
5
```

データが離散的で、中央値が補間値よりもデータ点にある値の方が良い場合に用いてください。

statistics.**median_grouped**(*data*, *interval=1*)

補間を使用して、50 番目のパーセンタイルとして計算されたグループ化された連続データの中央値を返します。もし *data* が空の場合、*StatisticsError* が送出されます。*data* はシーケンス型やイテラブルにもなれます。

```
>>> median_grouped([52, 52, 53, 54])
52.5
```

次の例ではデータは丸められているため、各値はデータの中間です。例えば 1 は 0.5 と 1.5 の中間、2 は 1.5 と 2.5 の中間、3 は 2.5 と 3.5 の中間です。例では中央の値は 3.5 から 4.5 で、補間により推定されます:

```
>>> median_grouped([1, 2, 2, 3, 4, 4, 4, 4, 4, 5])
3.7
```

interval は間隔を表します。デフォルトは 1 です。間隔を変えると当然補間値は変わります:

```
>>> median_grouped([1, 3, 3, 5, 7], interval=1)
3.25
>>> median_grouped([1, 3, 3, 5, 7], interval=2)
3.5
```

この関数はデータ点が少なくとも *interval* だけ差があるかどうかチェックしません。

CPython implementation detail: 環境によっては *median_grouped()* はデータ点を強制的に浮動小数点に変換します。この挙動はいずれ変更されるでしょう。

参考:

- ”Statistics for the Behavioral Sciences”, Frederick J Gravetter and Larry B Wallnau (8th Edition).
- The SSMEDIAN function in the Gnome Gnumeric spreadsheet, including this discussion.

statistics.**mode**(*data*)

Return the single most common data point from discrete or nominal *data*. The mode (when it exists) is the most typical value and serves as a measure of central location.

If there are multiple modes with the same frequency, returns the first one encountered in the *data*. If the smallest or largest of those is desired instead, use `min(multimode(data))` or `max(multimode(data))`. If the input *data* is empty, *StatisticsError* is raised.

mode は離散データであることを想定していて、1 つの値を返します。これは学校で教わるような最頻値の標準的な取扱いです:

```
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

The mode is unique in that it is the only statistic in this package that also applies to nominal (non-numeric) data:

```
>>> mode(["red", "blue", "blue", "red", "green", "red", "red"])
'red'
```

バージョン 3.8 で変更: Now handles multimodal datasets by returning the first mode encountered. Formerly, it raised *StatisticsError* when more than one mode was found.

statistics.multimode(*data*)

Return a list of the most frequently occurring values in the order they were first encountered in the *data*. Will return more than one result if there are multiple modes or an empty list if the *data* is empty:

```
>>> multimode('aabbbbccddddeeffffgg')
['b', 'd', 'f']
>>> multimode('')
[]
```

バージョン 3.8 で追加.

statistics.pstdev(*data*, *mu=None*)

母標準偏差 (母分散の平方根) を返します。引数や詳細は *pvariance()* を参照してください。

```
>>> pstdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
0.986893273527251
```

statistics.pvariance(*data*, *mu=None*)

data の母分散を返します。*data* は実数の空でないシーケンスまたはイテラブルです。分散、すなわち 2 次の中心化モーメントはデータの散らばり具合の測度です。分散が大きいデータはばらつきが大きく、分散が小さいデータは平均値のまわりに固まっています。

If the optional second argument *mu* is given, it is typically the mean of the *data*. It can also be used to compute the second moment around a point that is not the mean. If it is missing or `None` (the default), the arithmetic mean is automatically calculated.

母集団全体から分散を計算する場合に用いてください。標本から分散を推定する場合は *variance()* を使いましょう。

data が空の場合 *StatisticsError* を送出します。

例:

```
>>> data = [0.0, 0.25, 0.25, 1.25, 1.5, 1.75, 2.75, 3.25]
>>> pvariance(data)
1.25
```

既にデータの平均値を計算している場合、それを第 2 引数 *mu* に渡して再計算を避けることが出来ます:

```
>>> mu = mean(data)
>>> pvariance(data, mu)
1.25
```

Decimal と Fraction がサポートされています:

```
>>> from decimal import Decimal as D
>>> pvariance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('24.815')

>>> from fractions import Fraction as F
>>> pvariance([F(1, 4), F(5, 4), F(1, 2)])
Fraction(13, 72)
```

注釈: 母集団全体で呼んだ場合は母分散 σ² を返します。代わりに標本で呼んだ場合は biased variance s^2、すなわち自由度 N の分散を返します。

If you somehow know the true population mean μ , you may use this function to calculate the variance of a sample, giving the known population mean as the second argument. Provided the data points are a random sample of the population, the result will be an unbiased estimate of the population variance.

---

statistics.stdev(*data*, *xbar=None*)

標本標準偏差 (標本分散の平方根) を返します。引数や詳細は *variance()* を参照してください。

```
>>> stdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
1.0810874155219827
```

statistics.variance(*data*, *xbar=None*)

data の標本分散を返します。*data* は少なくとも 2 つの実数の iterable です。分散、すなわち 2 次の中心化モーメントはデータの散らばり具合の測度です。分散が大きいデータはばらつきが大きく、分散が小さいデータは平均値のまわりに固まっています。

第 2 引数 *xbar* に値を渡す場合は *data* の平均値でなくてはなりません。*xbar* が与えられない場合や None の場合 (デフォルト)、平均値は自動的に計算されます。

データが母集団の標本であるときに用いてください。母集団全体から分散を計算するには *pvariance()* を参照してください。

data の値が 2 より少ない場合 *StatisticsError* を送出します。

例:

```
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> variance(data)
1.3720238095238095
```

既にデータの平均値を計算している場合、それを第 2 引数 *xbar* に渡して再計算を避けることが出来ます:

```
>>> m = mean(data)
>>> variance(data, m)
1.3720238095238095
```

この関数は引数として渡した *xbar* が実際の平均値かどうかチェックしません。任意の値を *xbar* に渡すと無効な結果やありえない結果が返ることがあります。

Decimal と Fraction がサポートされています:

```
>>> from decimal import Decimal as D
>>> variance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('31.01875')

>>> from fractions import Fraction as F
>>> variance([F(1, 6), F(1, 2), F(5, 3)])
Fraction(67, 108)
```

注釈: Bessel 補正済みの標本分散 s^2、すなわち自由度 N-1 の分散です。与えられたデータ点が代表的 (たとえば独立で均等に分布) な場合、戻り値は母分散の不偏推定量になります。

何らかの方法で真の母平均 μ を知っている場合、それを *pvariance()* の引数 *mu* に渡して標本の分散を計算することが出来ます。

---

statistics.quantiles(*data*, ***, *n=4*, *method='exclusive'*)

Divide *data* into *n* continuous intervals with equal probability. Returns a list of `n - 1` cut points separating the intervals.

Set *n* to 4 for quartiles (the default). Set *n* to 10 for deciles. Set *n* to 100 for percentiles which gives the 99 cuts points that separate *data* into 100 equal sized groups. Raises *StatisticsError* if *n* is not least 1.

The *data* can be any iterable containing sample data. For meaningful results, the number of data points in *data* should be larger than *n*. Raises *StatisticsError* if there are not at least two data points.

The cut points are linearly interpolated from the two nearest data points. For example, if a cut point falls one-third of the distance between two sample values, `100` and `112`, the cut-point will evaluate to `104`.

The *method* for computing quantiles can be varied depending on whether the *data* includes or excludes the lowest and highest possible values from the population.

The default *method* is "exclusive" and is used for data sampled from a population that can have more extreme values than found in the samples. The portion of the population falling below the *i-th* of *m* sorted data points is computed as `i / (m + 1)`. Given nine sample values, the method sorts them and assigns the following percentiles: 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%.

Setting the *method* to "inclusive" is used for describing population data or for samples that are known to include the most extreme values from the population. The minimum value in *data* is treated as the 0th percentile and the maximum value is treated as the 100th percentile. The portion of the population falling below the *i-th* of *m* sorted data points is computed as `(i - 1) / (m - 1)`. Given 11 sample values, the method sorts them and assigns the following percentiles: 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%.

```
# Decile cut points for empirically sampled data
>>> data = [105, 129, 87, 86, 111, 111, 89, 81, 108, 92, 110,
...         100, 75, 105, 103, 109, 76, 119, 99, 91, 103, 129,
...         106, 101, 84, 111, 74, 87, 86, 103, 103, 106, 86,
...         111, 75, 87, 102, 121, 111, 88, 89, 101, 106, 95,
...         103, 107, 101, 81, 109, 104]
>>> [round(q, 1) for q in quantiles(data, n=10)]
[81.0, 86.2, 89.0, 99.4, 102.5, 103.6, 106.0, 109.8, 111.0]
```

バージョン 3.8 で追加.

9.7.4 例外

例外が 1 つ定義されています:

exception statistics.StatisticsError

統計関係の例外。*ValueError* の派生クラス。

9.7.5 NormalDist オブジェクト

NormalDist is a tool for creating and manipulating normal distributions of a random variable. It is a class that treats the mean and standard deviation of data measurements as a single entity.

Normal distributions arise from the Central Limit Theorem and have a wide range of applications in statistics.

class statistics.NormalDist(*mu=0.0*, *sigma=1.0*)

Returns a new *NormalDist* object where *mu* represents the arithmetic mean and *sigma* represents the standard deviation.

sigma がマイナスの場合 *StatisticsError* を送出します。

mean

A read-only property for the arithmetic mean of a normal distribution.

median

A read-only property for the median of a normal distribution.

mode

A read-only property for the mode of a normal distribution.

stdev

A read-only property for the standard deviation of a normal distribution.

variance

A read-only property for the variance of a normal distribution. Equal to the square of the standard deviation.

classmethod from_samples(*data*)

Makes a normal distribution instance with *mu* and *sigma* parameters estimated from the *data* using *fmean()* and *stdev()*.

The *data* can be any *iterable* and should consist of values that can be converted to type *float*. If *data* does not contain at least two elements, raises *StatisticsError* because it takes at least one point to estimate a central value and at least two points to estimate dispersion.

samples(*n*, ***, *seed=None*)

Generates *n* random samples for a given mean and standard deviation. Returns a *list* of *float* values.

If *seed* is given, creates a new instance of the underlying random number generator. This is useful for creating reproducible results, even in a multi-threading context.

pdf(*x*)

Using a probability density function (pdf), compute the relative likelihood that a random variable *X* will be near the given value *x*. Mathematically, it is the limit of the ratio `P(x <= X < x+dx) / dx` as *dx* approaches zero.

The relative likelihood is computed as the probability of a sample occurring in a narrow range divided by the width of the range (hence the word "density"). Since the likelihood is relative to other points, its value can be greater than *1.0*.

cdf(*x*)

Using a cumulative distribution function (cdf), compute the probability that a random variable *X* will be less than or equal to *x*. Mathematically, it is written `P(X <= x)`.

inv_cdf(*p*)

Compute the inverse cumulative distribution function, also known as the quantile function or the percent-point function. Mathematically, it is written `x : P(X <= x) = p`.

Finds the value *x* of the random variable *X* such that the probability of the variable being less than or equal to that value equals the given probability *p*.

overlap(*other*)

Measures the agreement between two normal probability distributions. Returns a value between 0.0 and 1.0 giving the overlapping area for the two probability density functions.

quantiles(*n=4*)

Divide the normal distribution into *n* continuous intervals with equal probability. Returns a list of (n - 1) cut points separating the intervals.

Set *n* to 4 for quartiles (the default). Set *n* to 10 for deciles. Set *n* to 100 for percentiles which gives the 99 cuts points that separate the normal distribution into 100 equal sized groups.

Instances of *NormalDist* support addition, subtraction, multiplication and division by a constant. These operations are used for translation and scaling. For example:

```
>>> temperature_february = NormalDist(5, 2.5)             # Celsius
>>> temperature_february * (9/5) + 32                     # Fahrenheit
NormalDist(mu=41.0, sigma=4.5)
```

Dividing a constant by an instance of *NormalDist* is not supported because the result wouldn't be normally distributed.

Since normal distributions arise from additive effects of independent variables, it is possible to add and subtract two independent normally distributed random variables represented as instances of *NormalDist*. For example:

```
>>> birth_weights = NormalDist.from_samples([2.5, 3.1, 2.1, 2.4, 2.7, 3.5])
>>> drug_effects = NormalDist(0.4, 0.15)
>>> combined = birth_weights + drug_effects
>>> round(combined.mean, 1)
3.1
>>> round(combined.stdev, 1)
0.5
```

バージョン 3.8 で追加.

NormalDist の例とレシピ

NormalDist readily solves classic probability problems.

For example, given historical data for SAT exams showing that scores are normally distributed with a mean of 1060 and a standard deviation of 195, determine the percentage of students with test scores between 1100 and 1200, after rounding to the nearest whole number:

```
>>> sat = NormalDist(1060, 195)
>>> fraction = sat.cdf(1200 + 0.5) - sat.cdf(1100 - 0.5)
>>> round(fraction * 100.0, 1)
18.4
```

Find the quartiles and deciles for the SAT scores:

```
>>> list(map(round, sat.quantiles()))
[928, 1060, 1192]
>>> list(map(round, sat.quantiles(n=10)))
[810, 896, 958, 1011, 1060, 1109, 1162, 1224, 1310]
```

To estimate the distribution for a model than isn't easy to solve analytically, *NormalDist* can generate input samples for a Monte Carlo simulation:

```
>>> def model(x, y, z):
...     return (3*x + 7*x*y - 5*y) / (11 * z)
...
>>> n = 100_000
>>> X = NormalDist(10, 2.5).samples(n, seed=3652260728)
>>> Y = NormalDist(15, 1.75).samples(n, seed=4582495471)
>>> Z = NormalDist(50, 1.25).samples(n, seed=6582483453)
>>> quantiles(map(model, X, Y, Z))
[1.4591308524824727, 1.8035946855390597, 2.175091447274739]
```

Normal distributions can be used to approximate Binomial distributions when the sample size is large and when the probability of a successful trial is near 50%.

For example, an open source conference has 750 attendees and two rooms with a 500 person capacity. There is a talk about Python and another about Ruby. In previous conferences, 65% of the attendees preferred to listen to Python talks. Assuming the population preferences haven't changed, what is the probability that the Python room will stay within its capacity limits?

```
>>> n = 750             # Sample size
>>> p = 0.65            # Preference for Python
>>> q = 1.0 - p         # Preference for Ruby
>>> k = 500             # Room capacity

>>> # Approximation using the cumulative normal distribution
>>> from math import sqrt
>>> round(NormalDist(mu=n*p, sigma=sqrt(n*p*q)).cdf(k + 0.5), 4)
0.8402

>>> # Solution using the cumulative binomial distribution
>>> from math import comb, fsum
>>> round(fsum(comb(n, r) * p**r * q**(n-r) for r in range(k+1)), 4)
0.8402

>>> # Approximation using a simulation
>>> from random import seed, choices
>>> seed(8675309)
>>> def trial():
...     return choices(('Python', 'Ruby'), (p, q), k=n).count('Python')
>>> mean(trial() <= k for i in range(10_000))
0.8398
```

Normal distributions commonly arise in machine learning problems.

Wikipedia has a nice example of a Naive Bayesian Classifier. The challenge is to predict a person's gender from measurements of normally distributed features including height, weight, and foot size.

We're given a training dataset with measurements for eight people. The measurements are assumed to be normally distributed, so we summarize the data with *NormalDist*:

```
>>> height_male = NormalDist.from_samples([6, 5.92, 5.58, 5.92])
>>> height_female = NormalDist.from_samples([5, 5.5, 5.42, 5.75])
>>> weight_male = NormalDist.from_samples([180, 190, 170, 165])
>>> weight_female = NormalDist.from_samples([100, 150, 130, 150])
>>> foot_size_male = NormalDist.from_samples([12, 11, 12, 10])
>>> foot_size_female = NormalDist.from_samples([6, 8, 7, 9])
```

Next, we encounter a new person whose feature measurements are known but whose gender is unknown:

```
>>> ht = 6.0        # height
>>> wt = 130        # weight
>>> fs = 8          # foot size
```

Starting with a 50% prior probability of being male or female, we compute the posterior as the prior times the product of likelihoods for the feature measurements given the gender:

```
>>> prior_male = 0.5
>>> prior_female = 0.5
>>> posterior_male = (prior_male * height_male.pdf(ht) *
...                   weight_male.pdf(wt) * foot_size_male.pdf(fs))

>>> posterior_female = (prior_female * height_female.pdf(ht) *
...                     weight_female.pdf(wt) * foot_size_female.pdf(fs))
```

The final prediction goes to the largest posterior. This is known as the maximum a posteriori or MAP:

```
>>> 'male' if posterior_male > posterior_female else 'female'
'female'
```

第

TEN

関数型プログラミング用モジュール

この章では、関数型プログラミングスタイルをサポートする呼び出し可能で汎用な操作を実現する関数やクラスについて説明します。

この章では以下のモジュールが解説されています:

10.1 itertools --- 効率的なループ実行のためのイテレータ生成関数

このモジュールは **イテレータ** を構築する部品を実装しています。プログラム言語 APL, Haskell, SML からアイデアを得ていますが、Python に適した形に修正されています。

このモジュールは、高速でメモリ効率に優れ、単独でも組合せても使用することのできるツールを標準化したものです。同時に、このツール群は ” イテレータの代数” を構成していて、pure Python で簡潔かつ効率的なツールを作れるようにしています。

例えば、SML の作表ツール `tabulate(f)` は `f(0), f(1), ...` のシーケンスを作成します。同じことを Python では *`map()`* と *`count()`* を組合せて `map(f, count())` という形で実現できます。

これらのツールと組み込み関数は *`operator`* モジュール内の高速な関数とともに使うことで見事に動作します。例えば、乗算演算子を 2 つのベクトルにわたってマップすることで効率的な内積計算を実現できます: `sum(map(operator.mul, vector1, vector2))` 。

無限イテレータ:

イテレータ	引数	結果	使用例
`count()`	start, [step]	start, start+step, start+2*step, ...	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, ... plast, p0, p1, ...	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, ... 無限もしくは n 回	`repeat(10, 3) --> 10 10 10`

一番短い入力シーケンスで止まるイテレータ:

イテレータ	引数	結果	使用例
accumulate()	p [,func]	p0, p0+p1, p0+p1+p2, ...	`accumulate([1,2,3,4,5]) --> 1 3 6 10 15`
chain()	p, q, ...	p0, p1, ... plast, q0, q1, ...	`chain('ABC', 'DEF') --> A B C D E F`
chain.from_iterable()	iterable	p0, p1, ... plast, q0, q1, ...	`chain.from_iterable(['ABC', 'DEF']) --> A B C D E F`
compress()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ...	`compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F`
dropwhile()	pred, seq	seq[n], seq[n+1], pred が偽の場所から始まる	`dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1`
filterfalse()	pred, seq	pred(elem) が偽になる seq の要素	`filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8`
groupby()	iterable[, key]	key(v) の値でグループ化したサブイテレータ	
islice()	seq, [start,] stop [, step]	seq[start:stop:step]	`islice('ABCDEFG', 2, None) --> C D E F G`
starmap()	func, seq	func(*seq[0]), func(*seq[1]), ...	`starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000`
takewhile()	pred, seq	seq[0], seq[1], pred が偽になるまで	`takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4`
tee()	it, n	it1, it2 , ... itn 一つのイテレータを n 個に分ける	
zip_longest()	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	`zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-`

組合せイテレータ:

イテレータ	引数	結果
product()	p, q, ... [repeat=1]	デカルト積、ネストした for ループと等価
permutations()	p[, r]	長さ r のタプル列、重複なしのあらゆる並び
combinations()	p, r	長さ r のタプル列、ソートされた順で重複なし
combinations_with_replacement()	p, r	長さ r のタプル列、ソートされた順で重複あり

使用例	結果
product('ABCD', repeat=2)	AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
permutations('ABCD', 2)	AB AC AD BA BC BD CA CB CD DA DB DC
combinations('ABCD', 2)	AB AC AD BC BD CD
combinations_with_replacement('ABCD', 2)	AA AB AC AD BB BC BD CC CD DD

10.1.1 Itertool 関数

以下の関数は全て、イテレータを作成して返します。無限長のストリームのイテレータを返す関数もあり、この場合にはストリームを中断するような関数かループ処理から使用しなければなりません。

itertools.**accumulate**(*iterable*[, *func*, *, *initial=None*])

累計や加算ではない (*func* オプション引数で指定される) 2 変数関数による累積結果を返すイテレータを作成します。

func が与えられた場合、2 つの引数を取る関数でなければなりません。入力 *iterable* の要素は、*func* が引数として取れる型を持ちます。(例えば、デフォルトの演算の加算では、要素は *Decimal* や *Fraction* を含む、加算ができる型を持ちます。)

通常、出力される要素の数は入力 *iterable* の要素数と一致します。ただし、キーワード引数 *initial* が与えられたときは、*initial* を先頭値としたイテレーションが行われ、出力される *iterable* は入力 *iterable* より要素を 1 つ多く持ちます。

およそ次と等価です:

```
def accumulate(iterable, func=operator.add, *, initial=None):
    'Return running totals'
    # accumulate([1,2,3,4,5]) --> 1 3 6 10 15
    # accumulate([1,2,3,4,5], initial=100) --> 100 101 103 106 110 115
    # accumulate([1,2,3,4,5], operator.mul) --> 1 2 6 24 120
    it = iter(iterable)
    total = initial
    if initial is None:
        try:
            total = next(it)
        except StopIteration:
            return
    yield total
    for element in it:
        total = func(total, element)
        yield total
```

func 引数の利用法はたくさんあります。最小値にするために *min()* を、最大値にするために *max()* を、積にするために *operator.mul()* を使うことができます。金利を累積し支払いを適用して償還表を作成することもできます。初期値をイテラブルに与えて *func* 引数で累積和を利用するだけで一階の

漸化式 をモデル化できます:

```
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, operator.mul))     # running product
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]
>>> list(accumulate(data, max))              # running maximum
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]

# Amortize a 5% loan of 1000 with 4 annual payments of 90
>>> cashflows = [1000, -90, -90, -90, -90]
>>> list(accumulate(cashflows, lambda bal, pmt: bal*1.05 + pmt))
[1000, 960.0, 918.0, 873.9000000000001, 827.5950000000001]

# Chaotic recurrence relation https://en.wikipedia.org/wiki/Logistic_map
>>> logistic_map = lambda x, _:  r * x * (1 - x)
>>> r = 3.8
>>> x0 = 0.4
>>> inputs = repeat(x0, 36)     # only the initial value is used
>>> [format(x, '.2f') for x in accumulate(inputs, logistic_map)]
['0.40', '0.91', '0.30', '0.81', '0.60', '0.92', '0.29', '0.79', '0.63',
 '0.88', '0.39', '0.90', '0.33', '0.84', '0.52', '0.95', '0.18', '0.57',
 '0.93', '0.25', '0.71', '0.79', '0.63', '0.88', '0.39', '0.91', '0.32',
 '0.83', '0.54', '0.95', '0.20', '0.60', '0.91', '0.30', '0.80', '0.60']
```

最終的な累積値だけを返す類似の関数については *functools.reduce()* を見てください。

バージョン 3.2 で追加.

バージョン 3.3 で変更: オプションの *func* 引数が追加されました。

バージョン 3.8 で変更: オプションの *initial* パラメータが追加されました。

itertools.**chain**(**iterables*)

先頭の iterable の全要素を返し、次に 2 番目の iterable の全要素を返し、と全 iterable の要素を返すイテレータを作成します。連続したシーケンスを一つのシーケンスとして扱う場合に使用します。およそ次と等価です:

```
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

classmethod chain.**from_iterable**(*iterable*)

chain() のためのもう一つのコンストラクタです。遅延評価される iterable 引数一つから連鎖した入力を受け取ります。この関数は、以下のコードとほぼ等価です:

```
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

itertools.**combinations**(*iterable*, *r*)

入力 *iterable* の要素からなる長さ *r* の部分列を返します。

組合せ (combination) は入力 *iterable* に応じた辞書式順序で出力されます。したがって、入力 *iterable* がソートされていれば、出力される組合わせ (combination) タプルもソートされた順番で生成されます。

各要素は場所に基づいて一意に取り扱われ、値にはよりません。入力された要素がバラバラなら各組合せの中に重複した値は現れません。

およそ次と等価です:

```
def combinations(iterable, r):
    # combinations('ABCD', 2) --> AB AC AD BC BD CD
    # combinations(range(4), 3) --> 012 013 023 123
    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = list(range(r))
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

combinations() のコードは *permutations()* のシーケンスから (入力プールでの位置に応じた順序で) 要素がソートされていないものをフィルターしたようにも表現できます:

```
def combinations(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in permutations(range(n), r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

返される要素の数は、`0 <= r <= n` の場合は、`n! / r! / (n-r)!` で、`r > n` の場合は 0 です。

itertools.**combinations_with_replacement**(*iterable*, *r*)

入力 *iterable* から、それぞれの要素が複数回現れることを許して、長さ *r* の要素の部分列を返します。

組合せ (combination) は入力 *iterable* に応じた辞書式順序で出力されます。したがって、入力 *iterable* がソートされていれば、出力される組合わせ (combination) タプルもソートされた順番で生成されます。

要素は、値ではなく位置に基づいて一意に扱われます。ですから、入力の要素が一意であれば、生成さ

れた組合せも一意になります。

およそ次と等価です:

```
def combinations_with_replacement(iterable, r):
    # combinations_with_replacement('ABC', 2) --> AA AB AC BB BC CC
    pool = tuple(iterable)
    n = len(pool)
    if not n and r:
        return
    indices = [0] * r
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != n - 1:
                break
        else:
            return
        indices[i:] = [indices[i] + 1] * (r - i)
        yield tuple(pool[i] for i in indices)
```

combinations_with_replacement() のコードは、*product()* の部分列から、要素が (入力プールの位置に従って) ソートされた順になっていない項目をフィルタリングしたものとしても表せます:

```
def combinations_with_replacement(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in product(range(n), repeat=r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

返される要素の数は、`n > 0` のとき `(n+r-1)! / r! / (n-1)!` です。

バージョン 3.1 で追加.

itertools.**compress**(*data*, *selectors*)

data の要素から *selectors* の対応する要素が `True` と評価されるものだけをフィルタしたイテレータを作ります。*data* と *selectors* のいずれかが尽きたときに止まります。およそ次と等価です:

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
    return (d for d, s in zip(data, selectors) if s)
```

バージョン 3.1 で追加.

itertools.**count**(*start=0*, *step=1*)

数 *start* で始まる等間隔の値を返すイテレータを作成します。*map()* に渡して連続したデータを生成するのによく使われます。また、*zip()* に連続した番号を追加するのにも使われます。およそ次と等価です:

```
def count(start=0, step=1):
    # count(10) --> 10 11 12 13 14 ...
    # count(2.5, 0.5) -> 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
```

浮動小数点数でカウントするときは (`start + step * i for i in count()`) のように掛け算を使ったコードに置き換えたほうが正確にできることがあります。

バージョン 3.1 で変更: *step* 引数が追加され、非整数の引数が許されるようになりました。

itertools.**cycle**(*iterable*)

iterable から要素を取得し、そのコピーを保存するイテレータを作成します。iterable の全要素を返すと、セーブされたコピーから要素を返します。これを無限に繰り返します。およそ次と等価です:

```
def cycle(iterable):
    # cycle('ABCD') --> A B C D A B C D A B C D ...
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
    while saved:
        for element in saved:
              yield element
```

cycle() は大きなメモリ領域を使用します。使用するメモリ量は iterable の大きさに依存します。

itertools.**dropwhile**(*predicate*, *iterable*)

predicate (述語) が真である間は要素を飛ばし、その後は全ての要素を返すイテレータを作成します。このイテレータは、predicate が最初に偽になるまで **全く** 要素を返さないため、要素を返し始めるまでに長い時間がかかる場合があります。およそ次と等価です:

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

itertools.**filterfalse**(*predicate*, *iterable*)

iterable から predicate が `False` となる要素だけを返すイテレータを作成します。*predicate* が `None` の場合、偽の要素だけを返します。およそ次と等価です:

```
def filterfalse(predicate, iterable):
    # filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
```

(次のページに続く)

(前のページからの続き)

```
    if predicate is None:
        predicate = bool
    for x in iterable:
        if not predicate(x):
            yield x
```

itertools.**groupby**(*iterable*, *key=None*)

同じキーをもつような要素からなる *iterable* 中のグループに対して、キーとグループを返すようなイテレータを作成します。*key* は各要素に対するキー値を計算する関数です。キーを指定しない場合や None にした場合、*key* 関数のデフォルトは恒等関数になり要素をそのまま返します。通常、*iterable* は同じキー関数でソート済みである必要があります。

groupby() の操作は Unix の uniq フィルターと似ています。key 関数の値が変わるたびに休止または新しいグループを生成します (このために通常同じ key 関数でソートしておく必要があるのです)。この動作は SQL の入力順に関係なく共通の要素を集約する GROUP BY とは違います。

返されるグループはそれ自体がイテレータで、*groupby()* と *iterable* を共有しています。もととなる *iterable* を共有しているため、*groupby()* オブジェクトの要素取り出しを先に進めると、それ以前の要素であるグループは見えなくなってしまいます。従って、データが後で必要な場合にはリストの形で保存しておく必要があります:

```
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))      # Store group iterator as a list
    uniquekeys.append(k)
```

groupby() はおよそ次と等価です:

```
class groupby:
    # [k for k, g in groupby('AAAABBBCCDAABBB')] --> A B C D A B
    # [list(g) for k, g in groupby('AAAABBBCCD')] --> AAAA BBB CC D
    def __init__(self, iterable, key=None):
        if key is None:
            key = lambda x: x
        self.keyfunc = key
        self.it = iter(iterable)
        self.tgtkey = self.currkey = self.currvalue = object()
    def __iter__(self):
        return self
    def __next__(self):
        self.id = object()
        while self.currkey == self.tgtkey:
            self.currvalue = next(self.it)    # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)
        self.tgtkey = self.currkey
        return (self.currkey, self._grouper(self.tgtkey, self.id))
    def _grouper(self, tgtkey, id):
```

(次のページに続く)

(前のページからの続き)

```
        while self.id is id and self.currkey == tgtkey:
            yield self.currvalue
            try:
                self.currvalue = next(self.it)
            except StopIteration:
                return
            self.currkey = self.keyfunc(self.currvalue)
```

itertools.**islice**(*iterable*, *stop*)
itertools.**islice**(*iterable*, *start*, *stop*[, *step*])

iterable から要素を選択して返すイテレータを作成します。*start* が 0 でない場合、iterable の要素は start に達するまでスキップされます。その後、要素が順に返されます。

```
def islice(iterable, *args):
    # islice('ABCDEFG', 2) --> A B
    # islice('ABCDEFG', 2, 4) --> C D
    # islice('ABCDEFG', 2, None) --> C D E F G
    # islice('ABCDEFG', 0, None, 2) --> A C E G
    s = slice(*args)
    start, stop, step = s.start or 0, s.stop or sys.maxsize, s.step or 1
    it = iter(range(start, stop, step))
    try:
        nexti = next(it)
    except StopIteration:
        # Consume *iterable* up to the *start* position.
        for i, element in zip(range(start), iterable):
            pass
        return
    try:
        for i, element in enumerate(iterable):
            if i == nexti:
                yield element
                nexti = next(it)
    except StopIteration:
        # Consume to *stop*.
        for i, element in zip(range(i + 1, stop), iterable):
            pass
```

start が None の場合、イテレーションは 0 から始まります。*step* が None の場合、ステップはデフォルトの 1 になります。

itertools.**permutations**(*iterable*, *r=None*)

iterable の要素からなる長さ *r* の順列 (permutation) を連続的に返します。

r が指定されない場合や None の場合、*r* はデフォルトで *iterable* の長さとなり、可能な最長の順列の全てが生成されます。

順列 (permutation) は入力 *iterable* に応じた辞書式順序で出力されます。したがって、入力 *iterable* がソートされていれば、出力される組合わせタプルもソートされた順番で生成されます。

要素は値ではなく位置に基づいて一意的に扱われます。したがって入力された要素が全て異なっている場合、それぞれの順列に重複した要素が現れないことになります。

およそ次と等価です:

```
def permutations(iterable, r=None):
    # permutations('ABCD', 2) --> AB AC AD BA BC BD CA CB CD DA DB DC
    # permutations(range(3)) --> 012 021 102 120 201 210
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return
    indices = list(range(n))
    cycles = list(range(n, n-r, -1))
    yield tuple(pool[i] for i in indices[:r])
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
```

permutations() のコードは *product()* の列から重複 (それらは入力プールの同じ位置から取られたものです) を除くようフィルタしたものとしても表現できます:

```
def permutations(iterable, r=None):
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    for indices in product(range(n), repeat=r):
        if len(set(indices)) == r:
            yield tuple(pool[i] for i in indices)
```

返される要素の数は、`0 <= r <= n` の場合 `n! / (n-r)!` で、`r > n` の場合は 0 です。

itertools.**product**(**iterables*, *repeat=1*)

入力イテラブルのデカルト積です。

ジェネレータ式の入れ子になった for ループとおよそ等価です。たとえば `product(A, B)` は `((x,y) for x in A for y in B)` と同じものを返します。

入れ子ループは走行距離計と同じように右端の要素がイテレーションごとに更新されていきます。このパターンは辞書式順序を作り出し、入力のイテレート可能オブジェクトたちがソートされていれば、直積タプルもソートされた順に出てきます。

イテラブル自身との直積を計算するためには、オプションの *repeat* キーワード引数に繰り返し回数を指定します。たとえば `product(A, repeat=4)` は `product(A, A, A, A)` と同じ意味です。

この関数は以下のコードとおよそ等価ですが、実際の実装ではメモリ中に中間結果を作りません:

```
def product(*args, repeat=1):
    # product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) --> 000 001 010 011 100 101 110 111
    pools = [tuple(pool) for pool in args] * repeat
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    for prod in result:
        yield tuple(prod)
```

itertools.**repeat**(*object*[, *times*])

繰り返し *object* を返すイテレータを作成します。*times* 引数を指定しなければ、無限に値を返し続けます。*map()* の引数にして、呼び出された関数に同じ引数を渡すのに使います。また *zip()* と使って、タプルの変わらない部分を作ります。

およそ次と等価です:

```
def repeat(object, times=None):
    # repeat(10, 3) --> 10 10 10
    if times is None:
        while True:
            yield object
    else:
        for i in range(times):
            yield object
```

repeat は *map* や *zip* に定数のストリームを与えるためによく利用されます:

```
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

itertools.**starmap**(*function*, *iterable*)

iterable の要素を引数として funtion を計算するイテレータを作成します。function の引数が一つの iterable からタプルに既にグループ化されている (データが "zip 済み") 場合、*map()* の代わりに使用します。*map()* と *starmap()* の違いは `function(a,b)` と `function(*c)` の差に似ています。およそ次と等価です:

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
    for args in iterable:
        yield function(*args)
```

itertools.**takewhile**(*predicate*, *iterable*)

predicate が真である限り iterable から要素を返すイテレータを作成します。およそ次と等価です:

```
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

itertools.**tee**(*iterable*, *n=2*)

一つの iterable から *n* 個の独立したイテレータを返します。

以下の Python コードは *tee* がすることについての理解を助けるでしょう (ただし実際の実装はより複雑で、下層の FIFO キューを一つしか使いませんが)。

およそ次と等価です:

```
def tee(iterable, n=2):
    it = iter(iterable)
    deques = [collections.deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:             # when the local deque is empty
                try:
                    newval = next(it)   # fetch a new value and
                except StopIteration:
                    return
                for d in deques:        # load it to all the deques
                    d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in deques)
```

一度 *tee()* でイテレータを分割すると、もとの *iterable* を他で使ってはいけません。さもなければ、*tee()* オブジェクトの知らない間に *iterable* が先の要素に進んでしまうことになります。

tee() イテレーターはスレッドセーフではありません。同じ *tee()* から生じた複数のイテレータを同時に使用すると、元のイテラブルがスレッドセーフであっても、*RuntimeError* が発生することがあります。

tee() はかなり大きなメモリ領域を使用するかもしれません (使用するメモリ量は iterable の大きさに依存します)。一般には、一つのイテレータが他のイテレータよりも先にほとんどまたは全ての要素を消費するような場合には、*tee()* よりも *list()* を使った方が高速です。

itertools.**zip_longest**(**iterables*, *fillvalue=None*)

各 iterable の要素をまとめるイテレータを作成します。iterable の長さが違う場合、足りない値は *fillvalue* で埋められます。最も長い itarable が尽きるまでイテレーションします。およそ次と等価です:

```
def zip_longest(*args, fillvalue=None):
    # zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-
    iterators = [iter(it) for it in args]
    num_active = len(iterators)
```

(次のページに続く)

(前のページからの続き)

```
    if not num_active:
        return
    while True:
        values = []
        for i, it in enumerate(iterators):
            try:
                value = next(it)
            except StopIteration:
                num_active -= 1
                if not num_active:
                    return
                iterators[i] = repeat(fillvalue)
                value = fillvalue
            values.append(value)
        yield tuple(values)
```

iterables の 1 つが無限になりうる場合 *zip_longest()* は呼び出し回数を制限するような何かでラップしなければいけません (例えば *islice()* or *takewhile()*)。*fillvalue* は指定しない場合のデフォルトは None です。

10.1.2 Itertools レシピ

この節では、既存の itertools を素材としてツールセットを拡張するためのレシピを示します。

下記のレシピをはじめ、さまざまなレシピが Python Package Index 上の more-itertools project からインストールできます。

```
pip install more-itertools
```

iterable 全体を一度にメモリ上に置くよりも、要素を一つづつ処理する方がメモリ効率上の有利さを保てます。関数形式のままツールをリンクしてゆくと、コードのサイズを減らし、一時変数を減らす助けになります。インタプリタのオーバヘッドをもたらす for ループや **ジェネレータ** を使わずに、" ベクトル化された" ビルディングブロックを使うと、高速な処理を実現できます。

```
def take(n, iterable):
    "Return first n items of the iterable as a list"
    return list(islice(iterable, n))

def prepend(value, iterator):
    "Prepend a single value in front of an iterator"
    # prepend(1, [2, 3, 4]) -> 1 2 3 4
    return chain([value], iterator)

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return map(function, count(start))

def tail(n, iterable):
```

(次のページに続く)

(前のページからの続き)

```
    "Return an iterator over the last n items"
    # tail(3, 'ABCDEFG') --> E F G
    return iter(collections.deque(iterable, maxlen=n))

def consume(iterator, n=None):
    "Advance the iterator n-steps ahead. If n is None, consume entirely."
    # Use functions that consume iterators at C speed.
    if n is None:
        # feed the entire iterator into a zero-length deque
        collections.deque(iterator, maxlen=0)
    else:
        # advance to the empty slice starting at position n
        next(islice(iterator, n, n), None)

def nth(iterable, n, default=None):
    "Returns the nth item or a default value"
    return next(islice(iterable, n, None), default)

def all_equal(iterable):
    "Returns True if all the elements are equal to each other"
    g = groupby(iterable)
    return next(g, True) and not next(g, False)

def quantify(iterable, pred=bool):
    "Count how many times the predicate is true"
    return sum(map(pred, iterable))

def padnone(iterable):
    """Returns the sequence elements and then returns None indefinitely.

    Useful for emulating the behavior of the built-in map() function.
    """
    return chain(iterable, repeat(None))

def ncycles(iterable, n):
    "Returns the sequence elements n times"
    return chain.from_iterable(repeat(tuple(iterable), n))

def dotproduct(vec1, vec2):
    return sum(map(operator.mul, vec1, vec2))

def flatten(list_of_lists):
    "Flatten one level of nesting"
    return chain.from_iterable(list_of_lists)

def repeatfunc(func, times=None, *args):
    """Repeat calls to func with specified arguments.

    Example:  repeatfunc(random.random)
    """
    if times is None:
```

(次のページに続く)

(前のページからの続き)

```
        return starmap(func, repeat(args))
    return starmap(func, repeat(args, times))

def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    next(b, None)
    return zip(a, b)

def grouper(iterable, n, fillvalue=None):
    "Collect data into fixed-length chunks or blocks"
    # grouper('ABCDEFG', 3, 'x') --> ABC DEF Gxx"
    args = [iter(iterable)] * n
    return zip_longest(*args, fillvalue=fillvalue)

def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    # Recipe credited to George Sakkis
    num_active = len(iterables)
    nexts = cycle(iter(it).__next__ for it in iterables)
    while num_active:
        try:
            for next in nexts:
                yield next()
        except StopIteration:
            # Remove the iterator we just exhausted from the cycle.
            num_active -= 1
            nexts = cycle(islice(nexts, num_active))

def partition(pred, iterable):
    'Use a predicate to partition entries into false entries and true entries'
    # partition(is_odd, range(10)) --> 0 2 4 6 8   and  1 3 5 7 9
    t1, t2 = tee(iterable)
    return filterfalse(pred, t1), filter(pred, t2)

def powerset(iterable):
    "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

def unique_everseen(iterable, key=None):
    "List unique elements, preserving order. Remember all elements ever seen."
    # unique_everseen('AAAABBBCCDAABBB') --> A B C D
    # unique_everseen('ABBCcAD', str.lower) --> A B C D
    seen = set()
    seen_add = seen.add
    if key is None:
        for element in filterfalse(seen.__contains__, iterable):
            seen_add(element)
            yield element
    else:
```

(次のページに続く)

(前のページからの続き)

```
        for element in iterable:
            k = key(element)
            if k not in seen:
                seen_add(k)
                yield element

def unique_justseen(iterable, key=None):
    "List unique elements, preserving order. Remember only the element just seen."
    # unique_justseen('AAAABBBCCDAABBB') --> A B C D A B
    # unique_justseen('ABBCcAD', str.lower) --> A B C A D
    return map(next, map(operator.itemgetter(1), groupby(iterable, key)))

def iter_except(func, exception, first=None):
    """ Call a function repeatedly until an exception is raised.

    Converts a call-until-exception interface to an iterator interface.
    Like builtins.iter(func, sentinel) but uses an exception instead
    of a sentinel to end the loop.

    Examples:
        iter_except(functools.partial(heappop, h), IndexError)   # priority queue iterator
        iter_except(d.popitem, KeyError)                         # non-blocking dict iterator
        iter_except(d.popleft, IndexError)                       # non-blocking deque iterator
        iter_except(q.get_nowait, Queue.Empty)                   # loop over a producer Queue
        iter_except(s.pop, KeyError)                             # non-blocking set iterator

    """
    try:
        if first is not None:
            yield first()            # For database APIs needing an initial cast to db.first()
        while True:
            yield func()
    except exception:
        pass

def first_true(iterable, default=False, pred=None):
    """Returns the first true value in the iterable.

    If no true value is found, returns *default*

    If *pred* is not None, returns the first item
    for which pred(item) is true.

    """
    # first_true([a,b,c], x) --> a or b or c or x
    # first_true([a,b], x, f) --> a if f(a) else b if f(b) else x
    return next(filter(pred, iterable), default)

def random_product(*args, repeat=1):
    "Random selection from itertools.product(*args, **kwds)"
    pools = [tuple(pool) for pool in args] * repeat
```

(次のページに続く)

(前のページからの続き)

```
    return tuple(random.choice(pool) for pool in pools)

def random_permutation(iterable, r=None):
    "Random selection from itertools.permutations(iterable, r)"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "Random selection from itertools.combinations(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with_replacement(iterable, r):
    "Random selection from itertools.combinations_with_replacement(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.randrange(n) for i in range(r))
    return tuple(pool[i] for i in indices)

def nth_combination(iterable, r, index):
    'Equivalent to list(combinations(iterable, r))[index]'
    pool = tuple(iterable)
    n = len(pool)
    if r < 0 or r > n:
        raise ValueError
    c = 1
    k = min(r, n-r)
    for i in range(1, k+1):
        c = c * (n - k + i) // i
    if index < 0:
        index += c
    if index < 0 or index >= c:
        raise IndexError
    result = []
    while r:
        c, n, r = c*r//n, n-1, r-1
        while index >= c:
            index -= c
            c, n = c*(n-r)//n, n-1
        result.append(pool[-1-n])
    return tuple(result)
```

10.2 functools --- 高階関数と呼び出し可能オブジェクトの操作

ソースコード: Lib/functools.py

---

functools モジュールは高階関数、つまり関数に影響を及ぼしたり他の関数を返したりする関数のためのものです。一般に、どんな呼び出し可能オブジェクトでもこのモジュールの目的には関数として扱えます。

モジュール *functools* は以下の関数を定義します:

@functools.cached_property(*func*)

クラスのメソッドを、値を一度だけ計算して通常の属性としてキャッシュするプロパティに変換します。キャッシュはインスタンスの生存期間にわたって有効です。計算コストが高く、一度計算すればその後は不変であるようなインスタンスのプロパティに対して有用です。

以下はプログラム例です:

```
class DataSet:
    def __init__(self, sequence_of_numbers):
        self._data = sequence_of_numbers

    @cached_property
    def stdev(self):
        return statistics.stdev(self._data)

    @cached_property
    def variance(self):
        return statistics.variance(self._data)
```

バージョン 3.8 で追加.

注釈: This decorator requires that the `__dict__` attribute on each instance be a mutable mapping. This means it will not work with some types, such as metaclasses (since the `__dict__` attributes on type instances are read-only proxies for the class namespace), and those that specify `__slots__` without including `__dict__` as one of the defined slots (as such classes don't provide a `__dict__` attribute at all).

---

functools.cmp_to_key(*func*)

古いスタイルの比較関数を *key function* に変換します。key 関数を受け取るツール (*sorted()*, *min()*, *max()*, *heapq.nlargest()*, *heapq.nsmallest()*, *itertools.groupby()* など) と共に使用します。この関数は、主に比較関数を使っていた Python 2 からプログラムの移行のための変換ツールとして使われます。

比較関数は 2 つの引数を受け取り、それらを比較し、" より小さい" 場合は負の数を、同値の場合には 0 を、" より大きい" 場合には正の数を返す、あらゆる呼び出し可能オブジェクトです。key 関数は呼び出し可能オブジェクトで、1 つの引数を受け取り、ソートキーとして使われる値を返します。

以下はプログラム例です:

```
sorted(iterable, key=cmp_to_key(locale.strcoll))  # locale-aware sort order
```

ソートの例と簡単なチュートリアルは sortinghowto を参照して下さい。

バージョン 3.2 で追加.

@functools.lru_cache(*user_function*)

@functools.lru_cache(*maxsize=128*, *typed=False*)

関数をメモ化用の呼び出し可能オブジェクトでラップし、最近の呼び出し最大 *maxsize* 回まで保存するするデコレータです。高価な関数や I/O に束縛されている関数を定期的に同じ引数で呼び出すときに、時間を節約できます。

結果のキャッシュには辞書が使われるので、関数の位置引数およびキーワード引数はハッシュ可能でなくてはなりません。

引数のパターンが異なる場合は、異なる呼び出しと見なされ別々のキャッシュエントリーとなります。例えば、*f(a=1, b=2)* と *f(b=2, a=1)* はキーワード引数の順序が異なっているので、2 つの別個のキャッシュエントリーになります。

user_function が指定された場合、それは呼び出し可能でなければなりません。これにより *lru_cache* デコレータがユーザー関数に直接適用できるようになります。このとき *maxsize* の値はデフォルトの 128 となります:

```
@lru_cache
def count_vowels(sentence):
    sentence = sentence.casefold()
    return sum(sentence.count(vowel) for vowel in 'aeiou')
```

maxsize が `None` に設定された場合は、LRU 機能は無効化され、キャッシュは際限無く大きくなります。

typed が真に設定された場合は、関数の異なる型の引数が別々にキャッシュされます。例えば、`f(3)` と `f(3.0)` は別の結果をもたらす別の呼び出しとして扱われます。

キャッシュ効率の測定や *maxsize* パラメータの調整をしやすくするため、ラップされた関数には `cache_info()` 関数が追加されます。この関数は *hits*, *misses*, *maxsize*, *currsize* を示す *named tuple* を返します。マルチスレッド環境では、hits と misses は概算です。

このデコレータは、キャッシュの削除と無効化のための `cache_clear()` 関数も提供します。

元々の基底の関数には、`__wrapped__` 属性を通してアクセスできます。これはキャッシュを回避して、または関数を別のキャッシュでラップして、内観するのに便利です。

LRU (least recently used) キャッシュ は、最新の呼び出しが次も呼び出される可能性が最も高い場合 (例えば、ニュースサーバーの最も人気のある記事は、毎日変わる傾向にあります) に最も効率が良くなります。キャッシュのサイズ制限は、キャッシュがウェブサーバーなどの長期間に渡るプロセスにおける限界を超えては大きくならないことを保証します。

一般的には、LRU キャッシュは前回計算した値を再利用したいときにのみ使うべきです。そのため、副作用のある関数、呼び出すごとに個別の可変なオブジェクトを作成する必要がある関数、time() や

random() のような純粋でない関数をキャッシュする意味はありません。

静的 web コンテンツ の LRU キャッシュの例:

```
@lru_cache(maxsize=32)
def get_pep(num):
    'Retrieve text of a Python Enhancement Proposal'
    resource = 'http://www.python.org/dev/peps/pep-%04d/' % num
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
...     pep = get_pep(n)
...     print(n, len(pep))

>>> get_pep.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)
```

キャッシュを使って 動的計画法 の技法を実装し、フィボナッチ数 を効率よく計算する例:

```
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)
```

バージョン 3.2 で追加.

バージョン 3.3 で変更: *typed* オプションが追加されました。

バージョン 3.8 で変更: *user_function* オプションが追加されました。

@functools.total_ordering

ひとつ以上の拡張順序比較メソッド (rich comparison ordering methods) を定義したクラスを受け取り、残りを実装するクラスデコレータです。このデコレータは全ての拡張順序比較演算をサポートするための労力を軽減します:

引数のクラスは、__lt__(), __le__(), __gt__(), __ge__() の中からどれか 1 つと、__eq__() メソッドを定義する必要があります。

例えば:

```
@total_ordering
class Student:
    def _is_valid_operand(self, other):
        return (hasattr(other, "lastname") and
                hasattr(other, "firstname"))
    def __eq__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
```

注釈: このデコレータにより、完全に順序の付いた振る舞いの良い型を簡単に作ることができますが、実行速度は遅くなり、派生した比較メソッドのスタックトレースは複雑になります。性能ベンチマークにより、これがアプリケーションのボトルネックになっていることがわかった場合は、代わりに 6 つの拡張比較メソッドをすべて実装すれば、簡単にスピードアップを図れるでしょう。

バージョン 3.2 で追加.

バージョン 3.4 で変更: 認識できない型に対して下層の比較関数から NotImplemented を返すことがサポートされるようになりました。

functools.**partial**(*func*, */*, **args*, ***keywords*)

新しい *partial* **オブジェクト** を返します。このオブジェクトは呼び出されると位置引数 *args* とキーワード引数 *keywords* 付きで呼び出された *func* のように振る舞います。呼び出しに際してさらなる引数が渡された場合、それらは *args* に付け加えられます。追加のキーワード引数が渡された場合には、それらで *keywords* を拡張または上書きします。おおよそ次のコードと等価です:

```
def partial(func, /, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = {**keywords, **fkeywords}
        return func(*args, *fargs, **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

関数 *partial()* は、関数の位置引数・キーワード引数の一部を「凍結」した部分適用として使われ、簡素化された引数形式をもった新たなオブジェクトを作り出します。例えば、*partial()* を使って *base* 引数のデフォルトが 2 である *int()* 関数のように振る舞う呼び出し可能オブジェクトを作ることができます:

```
>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
```

class functools.partialmethod(*func*, */*, **args*, ***keywords*)

partial と似た動作をする新しい *partialmethod* 記述子 (デスクリプタ) を返します。直接呼び出しではなく、メソッド定義としての使用が目的であることのみが、partial とは異なります。

func は、*descriptor* または呼び出し可能オブジェクトである必要があります (通常の関数など、両方の性質を持つオブジェクトは記述子として扱われます。)

func が記述子 (Python の通常の関数、*classmethod()*、*staticmethod()*、abstractmethod() または別の *partialmethod* のインスタンスなど) の場合、__get__ への呼び出しは下層の記述子に委譲され、返り値として適切な *partial* **オブジェクト** が返されます。

func が記述子以外の呼び出し可能オブジェクトである場合、適切な束縛メソッドが動的に作成されます。この func は、メソッドとして使用された場合、Python の通常の関数と同様に動作します。*partialmethod* コンストラクタに *args* と *keywords* が渡されるよりも前に、*self* 引数が最初の位置引数として挿入されます。

以下はプログラム例です:

```
>>> class Cell(object):
...     def __init__(self):
...         self._alive = False
...     @property
...     def alive(self):
...         return self._alive
...     def set_state(self, state):
...         self._alive = bool(state)
...     set_alive = partialmethod(set_state, True)
...     set_dead = partialmethod(set_state, False)
...
>>> c = Cell()
>>> c.alive
False
>>> c.set_alive()
>>> c.alive
True
```

バージョン 3.4 で追加.

functools.reduce(*function*, *iterable*[, *initializer*])

iterable の要素に対して、iterable を単一の値に短縮するような形で 2 つの引数をもつ *function* を左から右に累積的に適用します。例えば、reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) は (((((1+2)+3)+4)+5) を計算します。左引数 *x* は累計の値になり、右引数 *y* は iterable から取り出した更新値になります。オプションの *initializer* が存在する場合、計算の際に iterable の先頭に置かれます。また、iterable が空の場合には標準の値になります。*initializer* が与えられておらず、*iterable*

が単一の要素しか持っていない場合、最初の要素が返されます。

およそ次と等価です:

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value
```

全ての中間値を返すイテレータについては *itertools.accumulate()* を参照してください。

@functools.singledispatch

関数を **シングルディスパッチ ジェネリック関数** に変換します。

ジェネリック関数を定義するには、@singledispatch デコレータを付けます。ディスパッチは 1 つ目の引数の型で行われることに注意して、関数を次のように作成してください:

```
>>> from functools import singledispatch
>>> @singledispatch
... def fun(arg, verbose=False):
...     if verbose:
...         print("Let me just say,", end=" ")
...     print(arg)
```

関数にオーバーロード実装を追加するには、ジェネリック関数の register() 属性を使用します。この属性はデコレータです。型アノテーションが付いている関数については、このデコレータは 1 つ目の引数の型を自動的に推測します。

```
>>> @fun.register
... def _(arg: int, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> @fun.register
... def _(arg: list, verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

型アノテーションを使っていないコードについては、デコレータに適切な型引数を明示的に渡せます:

```
>>> @fun.register(complex)
... def _(arg, verbose=False):
...     if verbose:
```

(次のページに続く)

(前のページからの続き)

```
...         print("Better than complicated.", end=" ")
...     print(arg.real, arg.imag)
...
```

register() 属性を関数形式で使用すると、lambda 関数と既存の関数の登録を有効にできます:

```
>>> def nothing(arg, verbose=False):
...     print("Nothing.")
...
>>> fun.register(type(None), nothing)
```

The register() attribute returns the undecorated function which enables decorator stacking, pickling, as well as creating unit tests for each variant independently:

```
>>> @fun.register(float)
... @fun.register(Decimal)
... def fun_num(arg, verbose=False):
...     if verbose:
...         print("Half of your number:", end=" ")
...     print(arg / 2)
...
>>> fun_num is fun
False
```

汎用関数は、呼び出されると 1 つ目の引数の型でディスパッチします:

```
>>> fun("Hello, world.")
Hello, world.
>>> fun("test.", verbose=True)
Let me just say, test.
>>> fun(42, verbose=True)
Strength in numbers, eh? 42
>>> fun(['spam', 'spam', 'eggs', 'spam'], verbose=True)
Enumerate this:
0 spam
1 spam
2 eggs
3 spam
>>> fun(None)
Nothing.
>>> fun(1.23)
0.615
```

特定の型について登録された実装が存在しない場合、その型のメソッド解決順序が、汎用の実装をさらに検索するために使用されます。@singledispatch でデコレートされた元の関数は基底の object 型に登録されます。これは、他によりよい実装が見つからないことを意味します。

指定された型に対して、汎用関数がどの実装を選択するかを確認するには、dispatch() 属性を使用します:

```
>>> fun.dispatch(float)
<function fun_num at 0x1035a2840>
>>> fun.dispatch(dict)    # note: default implementation
<function fun at 0x103fe0000>
```

登録されたすべての実装にアクセスするには、読み出し専用の registry 属性を使用します:

```
>>> fun.registry.keys()
dict_keys([<class 'NoneType'>, <class 'int'>, <class 'object'>,
           <class 'decimal.Decimal'>, <class 'list'>,
           <class 'float'>])
>>> fun.registry[float]
<function fun_num at 0x1035a2840>
>>> fun.registry[object]
<function fun at 0x103fe0000>
```

バージョン 3.4 で追加.

バージョン 3.7 で変更: register() 属性が型アノテーションの使用をサポートするようになりました。

class functools.singledispatchmethod(*func*)

メソッドを **シングルディスパッチ ジェネリック関数** に変換します。

ジェネリックメソッドを定義するには、@singledispatchmethod デコレータを付けます。ディスパッチは 1 つ目の引数の型で行われることに注意して、関数を次のように作成してください:

```
class Negator:
    @singledispatchmethod
    def neg(self, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
    def _(self, arg: int):
        return -arg

    @neg.register
    def _(self, arg: bool):
        return not arg
```

@singledispatchmethod supports nesting with other decorators such as @classmethod. Note that to allow for dispatcher.register, singledispatchmethod must be the *outer most* decorator. Here is the Negator class with the neg methods being class bound:

```
class Negator:
    @singledispatchmethod
    @classmethod
    def neg(cls, arg):
        raise NotImplementedError("Cannot negate a")
```

(次のページに続く)

(前のページからの続き)

```
    @neg.register
    @classmethod
    def _(cls, arg: int):
        return -arg

    @neg.register
    @classmethod
    def _(cls, arg: bool):
        return not arg
```

The same pattern can be used for other similar decorators: `staticmethod`, `abstractmethod`, and others.

バージョン 3.8 で追加.

functools.update_wrapper(*wrapper*, *wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

wrapper 関数を *wrapped* 関数に見えるようにアップデートします。オプション引数はタプルで、元の関数のどの属性がラッパー関数の対応する属性に直接代入されるか、またラッパー関数のどの属性が元の関数の対応する属性でアップデートされるか、を指定します。これらの引数のデフォルト値は、モジュールレベル定数 `WRAPPER_ASSIGNMENTS` (これはラッパー関数の `__module__`, `__name__`, `__qualname__`, `__annotations__` そしてドキュメンテーション文字列 `__doc__` に代入する) と `WRAPPER_UPDATES` (これはラッパー関数の `__dict__` すなわちインスタンス辞書をアップデートする) です。

内観や別の目的 (例えば、*lru_cache()* のようなキャッシュするデコレータの回避) のために元の関数にアクセスできるように、この関数はラップされている関数を参照するラッパーに自動的に `__wrapped__` 属性を追加します。

この関数は主に関数を包んでラッパーを返す **デコレータ** 関数の中で使われるよう意図されています。もしラッパー関数がアップデートされないとすると、返される関数のメタデータは元の関数の定義ではなくラッパー関数の定義を反映してしまい、これは通常あまり有益ではありません。

update_wrapper() は、関数以外の呼び出し可能オブジェクトにも使えます。*assigned* または *updated* で指名され、ラップされるオブジェクトに存在しない属性は、すべて無視されます (すなわち、ラッパー関数にそれらの属性を設定しようとは試みられません)。しかし、*updated* で指名された属性がラッパー関数自身に存在しないなら *AttributeError* が送出されます。

バージョン 3.2 で追加: `__wrapped__` 属性の自動的な追加。

バージョン 3.2 で追加: デフォルトで `__annotations__` 属性がコピーされます。

バージョン 3.2 で変更: 存在しない属性によって *AttributeError* を発生しなくなりました。

バージョン 3.4 で変更: ラップされた関数が `__wrapped__` を定義していない場合でも、`__wrapped__` が常にラップされた関数を参照するようになりました。(bpo-17482 を参照)

@functools.wraps(*wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

これはラッパー関数を定義するときに *update_wrapper()* を関数デコレータとして呼び出

す便宜関数です。これは partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated) と等価です。例えば:

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print('Calling decorated function')
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print('Called example function')
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

このデコレータ・ファクトリを使用しないと、上の例中の関数の名前は 'wrapper' となり、元の example() のドキュメンテーション文字列は失われてしまいます。

10.2.1 partial オブジェクト

partial オブジェクトは、*partial()* 関数によって作られる呼び出し可能オブジェクトです。オブジェクトには読み出し専用の属性が三つあります:

partial.func
呼び出し可能オブジェクトまたは関数です。*partial* オブジェクトの呼び出しは新しい引数とキーワードと共に *func* に転送されます。

partial.args
最左の位置引数で、*partial* オブジェクトの呼び出し時にその呼び出しの際の位置引数の前に追加されます。

partial.keywords
partial オブジェクトの呼び出し時に渡されるキーワード引数です。

partial オブジェクトは function オブジェクトのように呼び出し可能で、弱参照可能で、属性を持つことができます。重要な相違点もあります。例えば、*__name__* と __doc__ 両属性は自動では作られません。また、クラス中で定義された *partial* オブジェクトはスタティックメソッドのように振る舞い、インスタンスの属性問い合わせの中で束縛メソッドに変換されません。

10.3 operator --- 関数形式の標準演算子

ソースコード: Lib/operator.py

---

operator モジュールは、Python の組み込み演算子に対応する効率的な関数群を提供します。例えば、operator.add(x, y) は式 x+y と等価です。多くの関数名は、特殊メソッドに使われている名前から前後の二重アンダースコアを除いたものと同じです。後方互換性のため、ほとんどの関数に二重アンダースコアを付けたままのバージョンがあります。簡潔さのために、二重アンダースコアが無いバージョンの方が好まれます。

これらの関数は、オブジェクト比較、論理演算、数学演算、シーケンス演算をするものに分類されます。

オブジェクト比較関数は全てのオブジェクトで有効で、関数の名前はサポートする拡張比較演算子からとられています:

operator.lt(*a*, *b*)
operator.le(*a*, *b*)
operator.eq(*a*, *b*)
operator.ne(*a*, *b*)
operator.ge(*a*, *b*)
operator.gt(*a*, *b*)
operator.__lt__(*a*, *b*)
operator.__le__(*a*, *b*)
operator.__eq__(*a*, *b*)
operator.__ne__(*a*, *b*)
operator.__ge__(*a*, *b*)
operator.__gt__(*a*, *b*)

a と *b* の ” 拡張比較 (rich comparisons)” を行います。具体的には、lt(a, b) は a < b 、le(a, b) は a <= b 、eq(a, b) は a == b 、ne(a, b) は a != b 、gt(a, b) は a > b 、そして ge(a, b) は a >= b と等価です。これらの関数はどのような値を返してもよく、ブール値として解釈できてもできなくてもかまいません。拡張比較の詳細については comparisons を参照してください。

論理演算もまた全てのオブジェクトに対して適用でき、真理値判定、同一性判定およびブール演算をサポートします:

operator.not_(*obj*)
operator.__not__(*obj*)

not *obj* の結果を返します。(オブジェクトインスタンスには *__not__()* メソッドは無いので注意してください; インタプリタコアがこの演算を定義しているだけです。結果は __bool__() および __len__() メソッドに影響されます。)

operator.truth(*obj*)

obj が真の場合 *True* を返し、そうでない場合 *False* を返します。この関数は *bool* のコンストラクタ呼び出しと同等です。

operator.is_(*a*, *b*)

a is b を返します。オブジェクトの同一性を判定します。

operator.**is_not**(*a*, *b*)
　a is not b を返します。オブジェクトの同一性を判定します。

演算子で最も多いのは数学演算およびビット単位の演算です:

operator.**abs**(*obj*)
operator.**__abs__**(*obj*)
　obj の絶対値を返します。

operator.**add**(*a*, *b*)
operator.**__add__**(*a*, *b*)
　数値 *a* および *b* について a + b を返します。

operator.**and_**(*a*, *b*)
operator.**__and__**(*a*, *b*)
　a と *b* のビット単位論理積を返します。

operator.**floordiv**(*a*, *b*)
operator.**__floordiv__**(*a*, *b*)
　a // b を返します。

operator.**index**(*a*)
operator.**__index__**(*a*)
　整数に変換された *a* を返します。a.__index__() と同等です。

operator.**inv**(*obj*)
operator.**invert**(*obj*)
operator.**__inv__**(*obj*)
operator.**__invert__**(*obj*)
　obj のビット単位反転を返します。~obj と同じです。

operator.**lshift**(*a*, *b*)
operator.**__lshift__**(*a*, *b*)
　a の *b* ビット左シフトを返します。

operator.**mod**(*a*, *b*)
operator.**__mod__**(*a*, *b*)
　a % b を返します。

operator.**mul**(*a*, *b*)
operator.**__mul__**(*a*, *b*)
　数値 *a* および *b* について a * b を返します。

operator.**matmul**(*a*, *b*)
operator.**__matmul__**(*a*, *b*)
　a @ b を返します。

　バージョン 3.5 で追加.

operator.**neg**(*obj*)

operator.__neg__(*obj*)
負の *obj* (-obj) を返します。

operator.or_(*a*, *b*)
operator.__or__(*a*, *b*)
a と *b* のビット単位論理和を返します。

operator.pos(*obj*)
operator.__pos__(*obj*)
正の *obj* (+obj) を返します。

operator.pow(*a*, *b*)
operator.__pow__(*a*, *b*)
数値 *a* および *b* について a ** b を返します。

operator.rshift(*a*, *b*)
operator.__rshift__(*a*, *b*)
a の *b* ビット右シフトを返します。

operator.sub(*a*, *b*)
operator.__sub__(*a*, *b*)
a - b を返します。

operator.truediv(*a*, *b*)
operator.__truediv__(*a*, *b*)
2/3 が 0 ではなく 0.66 となるような a / b を返します。" 真の" 除算としても知られています。

operator.xor(*a*, *b*)
operator.__xor__(*a*, *b*)
a および *b* のビット単位排他的論理和を返します。

シーケンスを扱う演算子（いくつかの演算子はマッピングも扱います）には以下のようなものがあります:

operator.concat(*a*, *b*)
operator.__concat__(*a*, *b*)
シーケンス *a* および *b* について a + b を返します。

operator.contains(*a*, *b*)
operator.__contains__(*a*, *b*)
b in a の判定結果を返します。被演算子が左右反転しているので注意してください。

operator.countOf(*a*, *b*)
a の中に *b* が出現する回数を返します。

operator.delitem(*a*, *b*)
operator.__delitem__(*a*, *b*)
a でインデクスが *b* の値を削除します。

operator.getitem(*a*, *b*)

operator.**__getitem__**(*a*, *b*)
: *a* でインデクスが *b* の値を返します。

operator.**indexOf**(*a*, *b*)
: *a* で最初に *b* が出現する場所のインデクスを返します。

operator.**setitem**(*a*, *b*, *c*)
operator.**__setitem__**(*a*, *b*, *c*)
: *a* でインデクスが *b* の値を *c* に設定します。

operator.**length_hint**(*obj*, *default=0*)
: オブジェクト *o* の概算の長さを返します。最初に実際の長さを、次に `object.__length_hint__()` を使って概算の長さを、そして最後にデフォルトの値を返そうとします。

  バージョン 3.4 で追加.

operator モジュールは属性とアイテムの汎用的な検索のための道具も定義しています。*map()*, *sorted()*, *itertools.groupby()*, や関数を引数に取るその他の関数に対して高速にフィールドを抽出する際に引数として使うと便利です。

operator.**attrgetter**(*attr*)
operator.**attrgetter**(**attrs*)
: 演算対象から *attr* を取得する呼び出し可能なオブジェクトを返します。二つ以上の属性を要求された場合には、属性のタプルを返します。属性名はドットを含むこともできます。例えば:

  - `f = attrgetter('name')` とした後で、`f(b)` を呼び出すと `b.name` を返します。
  - `f = attrgetter('name', 'date')` とした後で、`f(b)` を呼び出すと `(b.name, b.date)` を返します。
  - `f = attrgetter('name.first', 'name.last')` とした後で、`f(b)` を呼び出すと `(b.name.first, b.name.last)` を返します。

  次と等価です:

```
def attrgetter(*items):
    if any(not isinstance(item, str) for item in items):
        raise TypeError('attribute name must be a string')
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
        def g(obj):
            return tuple(resolve_attr(obj, attr) for attr in items)
    return g

def resolve_attr(obj, attr):
    for name in attr.split("."):
        obj = getattr(obj, name)
    return obj
```

operator.**itemgetter**(*item*)
operator.**itemgetter**(**items*)

演算対象からその *__getitem__()* メソッドを使って *item* を取得する呼び出し可能なオブジェクトを返します。二つ以上のアイテムを要求された場合には、アイテムのタプルを返します。例えば:

- `f = itemgetter(2)` とした後で、`f(r)` を呼び出すと `r[2]` を返します。
- `g = itemgetter(2, 5, 3)` とした後で、`g(r)` を呼び出すと `(r[2], r[5], r[3])` を返します。

次と等価です:

```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

アイテムは被演算子の *__getitem__()* メソッドが受け付けるどんな型でも構いません。辞書ならば任意のハッシュ可能な値を受け付けます。リスト、タプル、文字列などはインデクスかスライスを受け付けます:

```
>>> itemgetter(1)('ABCDEFG')
'B'
>>> itemgetter(1, 3, 5)('ABCDEFG')
('B', 'D', 'F')
>>> itemgetter(slice(2, None))('ABCDEFG')
'CDEFG'
>>> soldier = dict(rank='captain', name='dotterbart')
>>> itemgetter('rank')(soldier)
'captain'
```

itemgetter() を使って特定のフィールドをタプルレコードから取り出す例:

```
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> list(map(getcount, inventory))
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

operator.**methodcaller**(*name*, */*, **args*, ***kwargs*)

引数の *name* メソッドを呼び出す呼び出し可能オブジェクトを返します。追加の引数および/またはキーワード引数が与えられると、これらもそのメソッドに引き渡されます。例えば:

- `f = methodcaller('name')` とした後で、`f(b)` を呼び出すと `b.name()` を返します。
- `f = methodcaller('name', 'foo', bar=1)` とした後で、`f(b)` を呼び出すと `b.name('foo',`

bar=1) を返します。

次と等価です:

```
def methodcaller(name, /, *args, **kwargs):
    def caller(obj):
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

10.3.1 演算子から関数への対応表

下のテーブルでは、個々の抽象的な操作が、どのように Python 構文上の各演算子や *operator* モジュールの関数に対応しているかを示しています。

演算	操作	関数
加算	a + b	add(a, b)
結合	seq1 + seq2	concat(seq1, seq2)
包含判定	obj in seq	contains(seq, obj)
除算	a / b	truediv(a, b)
除算	a // b	floordiv(a, b)
ビット単位論理積	a & b	and_(a, b)
ビット単位排他的論理和	a ^ b	xor(a, b)
ビット単位反転	~ a	invert(a)
ビット単位論理和	a | b	or_(a, b)
冪乗	a ** b	pow(a, b)
同一性	a is b	is_(a, b)
同一性	a is not b	is_not(a, b)
インデクス指定の代入	obj[k] = v	setitem(obj, k, v)
インデクス指定の削除	del obj[k]	delitem(obj, k)
インデクス指定	obj[k]	getitem(obj, k)
左シフト	a << b	lshift(a, b)
剰余	a % b	mod(a, b)
乗算	a * b	mul(a, b)
行列の乗算	a @ b	matmul(a, b)
(算術) 負	- a	neg(a)
(論理) 否	not a	not_(a)
正	+ a	pos(a)
右シフト	a >> b	rshift(a, b)
スライス指定の代入	seq[i:j] = values	setitem(seq, slice(i, j), values)
スライス指定の削除	del seq[i:j]	delitem(seq, slice(i, j))
スライス指定	seq[i:j]	getitem(seq, slice(i, j))
文字列書式化	s % obj	mod(s, obj)

次のページに続く

表 1 – 前のページからの続き

演算	操作	関数
減算	a - b	sub(a, b)
真理値判定	obj	truth(obj)
順序付け	a < b	lt(a, b)
順序付け	a <= b	le(a, b)
等価性	a == b	eq(a, b)
不等性	a != b	ne(a, b)
順序付け	a >= b	ge(a, b)
順序付け	a > b	gt(a, b)

10.3.2 インプレース (in-place) 演算子

多くの演算に「インプレース」版があります。以下の関数はそうした演算子の通常の文法に比べてより素朴な呼び出し方を提供します。たとえば、文 x += y は x = operator.iadd(x, y) と等価です。別の言い方をすると、z = operator.iadd(x, y) は複合文 z = x; z += y と等価です。

なお、これらの例では、インプレースメソッドが呼び出されたとき、計算と代入は二段階に分けて行われます。以下に挙げるインプレース関数は、インプレースメソッドを呼び出してその第一段階だけを行います。第二段階の代入は扱われません。

文字列、数、タプルのようなイミュータブルなターゲットでは、更新された値が計算されますが、入力変数に代入し返されはしません。

```
>>> a = 'hello'
>>> iadd(a, ' world')
'hello world'
>>> a
'hello'
```

リストや辞書のようなミュータブルなターゲットでは、インプレースメソッドは更新を行うので、その後に代入をする必要はありません。

```
>>> s = ['h', 'e', 'l', 'l', 'o']
>>> iadd(s, [' ', 'w', 'o', 'r', 'l', 'd'])
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> s
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

operator.**iadd**(*a*, *b*)

operator.**__iadd__**(*a*, *b*)

a = iadd(a, b) は a += b と等価です。

operator.**iand**(*a*, *b*)

operator.**__iand__**(*a*, *b*)

a = iand(a, b) は a &= b と等価です。

operator.**iconcat**(*a*, *b*)

operator.**__iconcat__**(*a*, *b*)

a = iconcat(a, b) は二つのシーケンス *a* と *b* に対し a += b と等価です。

operator.**ifloordiv**(*a*, *b*)

operator.**__ifloordiv__**(*a*, *b*)

a = ifloordiv(a, b) は a //= b と等価です。

operator.**ilshift**(*a*, *b*)

operator.**__ilshift__**(*a*, *b*)

a = ilshift(a, b) は a <<= b と等価です。

operator.**imod**(*a*, *b*)

operator.**__imod__**(*a*, *b*)

a = imod(a, b) は a %= b と等価です。

operator.**imul**(*a*, *b*)

operator.**__imul__**(*a*, *b*)

a = imul(a, b) は a *= b と等価です。

operator.**imatmul**(*a*, *b*)

operator.**__imatmul__**(*a*, *b*)

a = imatmul(a, b) は a @= b と等価です。

バージョン 3.5 で追加.

operator.**ior**(*a*, *b*)

operator.**__ior__**(*a*, *b*)

a = ior(a, b) は a |= b と等価です。

operator.**ipow**(*a*, *b*)

operator.**__ipow__**(*a*, *b*)

a = ipow(a, b) は a **= b と等価です。

operator.**irshift**(*a*, *b*)

operator.**__irshift__**(*a*, *b*)

a = irshift(a, b) は a >>= b と等価です。

operator.**isub**(*a*, *b*)

operator.**__isub__**(*a*, *b*)

a = isub(a, b) は a -= b と等価です。

operator.**itruediv**(*a*, *b*)

operator.**__itruediv__**(*a*, *b*)

a = itruediv(a, b) は a /= b と等価です。

operator.**ixor**(*a*, *b*)

operator.**__ixor__**(*a*, *b*)

a = ixor(a, b) は a ^= b と等価です。

第

ELEVEN

ファイルとディレクトリへのアクセス

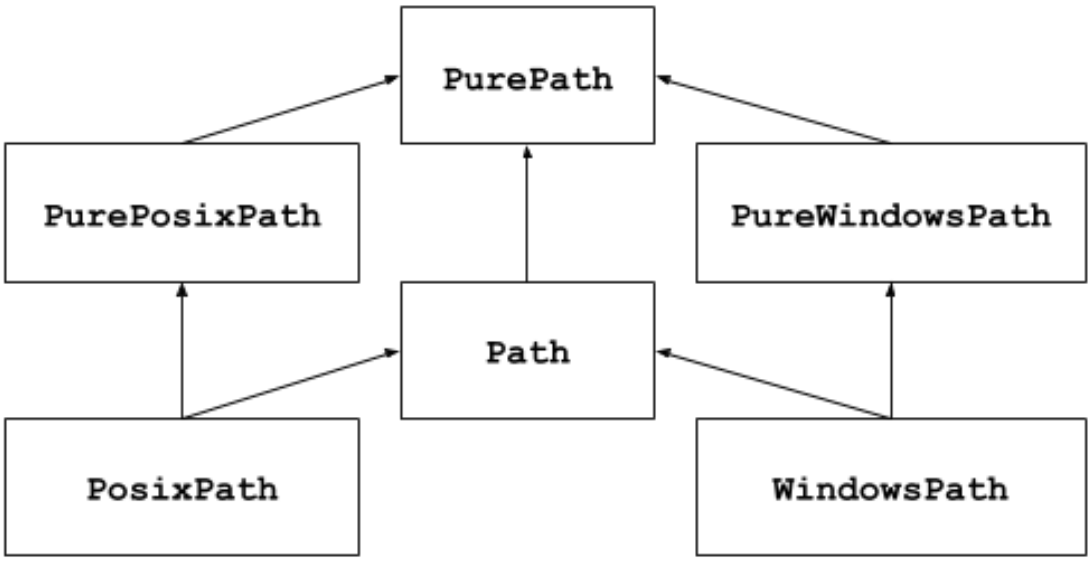
この章で説明されるモジュールはディスクのファイルやディレクトリを扱います。たとえば、ファイルの属性を読むためのモジュール、ファイルパスを移植可能な方式で操作する、テンポラリファイルを作成するためのモジュールです。この章の完全な一覧は:

11.1 pathlib --- オブジェクト指向のファイルシステムパス

バージョン 3.4 で追加.

ソースコード: Lib/pathlib.py

このモジュールはファイルシステムのパスを表すクラスを提供していて、様々なオペレーティングシステムについての適切な意味論をそれらのクラスに持たせています。Path クラスは **純粋パス** と **具象パス** からなります。純粋パスは I/O を伴わない純粋な計算操作を提供します。具象パスは純粋パスを継承していますが、I/O 操作も提供しています。



あなたが今までこのモジュールを使用したことがない場合や、タスクに適しているのがどのクラスかわからない場合は、*Path* はきっとあなたに必要なものでしょう。*Path* はコードが実行されているプラットフォーム用の **具象パス** のインスタンスを作成します。

純粋パスは、以下のようないくつかの特殊なケースで有用です:

1. Unix マシン上で Windows のパスを扱いたいとき (またはその逆)。Unix 上で実行しているときに *WindowsPath* のインスタンスを作成することはできませんが、*PureWindowsPath* なら可能になります。
2. 実際に OS にアクセスすることなしにパスを操作するだけのコードを確認したいとき。この場合、純粋クラスのインスタンスを一つ作成すれば、それが OS にアクセスすることはないので便利です。

参考:

PEP 428: The pathlib module -- オブジェクト指向のファイルシステムパス。

参考:

文字列による低水準のパス操作の場合は *os.path* も使用できます。

11.1.1 基本的な使い方

メインクラスをインポートします:

```
>>> from pathlib import Path
```

サブディレクトリの一覧を取得します:

```
>>> p = Path('.')
>>> [x for x in p.iterdir() if x.is_dir()]
[PosixPath('.hg'), PosixPath('docs'), PosixPath('dist'),
 PosixPath('__pycache__'), PosixPath('build')]
```

このディレクトリツリー内の Python ソースファイルの一覧を取得します:

```
>>> list(p.glob('**/*.py'))
[PosixPath('test_pathlib.py'), PosixPath('setup.py'),
 PosixPath('pathlib.py'), PosixPath('docs/conf.py'),
 PosixPath('build/lib/pathlib.py')]
```

ディレクトリツリー内を移動します:

```
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
>>> q.resolve()
PosixPath('/etc/rc.d/init.d/halt')
```

パスのプロパティを問い合わせます:

```
>>> q.exists()
True
>>> q.is_dir()
False
```

ファイルを開きます:

```
>>> with q.open() as f: f.readline()
...
'#!/bin/bash\n'
```

11.1.2 純粋パス

純粋パスオブジェクトは実際にファイルシステムにアクセスしないパス操作処理を提供します。これらのクラスにアクセスするには 3 つの方法があり、それらを **フレーバー** と呼んでいます:

class pathlib.**PurePath**(**pathsegments*)

システムのパスのフレーバーを表すジェネリッククラスです (インスタンスを作成することで *PurePosixPath* または *PureWindowsPath* のどちらかが作成されます):

```
>>> PurePath('setup.py')      # Running on a Unix machine
PurePosixPath('setup.py')
```

pathsegments の各要素は、部分パスの文字列表現か、文字列を返す *os.PathLike* インターフェイスを実装しているオブジェクトか、その他の path オブジェクトです。

```
>>> PurePath('foo', 'some/path', 'bar')
PurePosixPath('foo/some/path/bar')
>>> PurePath(Path('foo'), Path('bar'))
PurePosixPath('foo/bar')
```

pathsegments が空のとき、現在のディレクトリとみなされます:

```
>>> PurePath()
PurePosixPath('.')
```

絶対パスが複数与えられた場合、最後の要素がアンカーとして取られます (*os.path.join()* の挙動を真似ています):

```
>>> PurePath('/etc', '/usr', 'lib64')
PurePosixPath('/usr/lib64')
>>> PureWindowsPath('c:/Windows', 'd:bar')
PureWindowsPath('d:bar')
```

ただし、Windows のパスでは、ローカルルートを変更してもそれまでのドライブ設定は破棄されません:

```
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

誤ったスラッシュおよび単一ドットは無視されますが、2 個のドット (`'..'`) は、シンボリックリンクのときにパスの意味の変更を意味するため受け付けられます:

```
>>> PurePath('foo//bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/./bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/../bar')
PurePosixPath('foo/../bar')
```

(通常 `PurePosixPath('foo/../bar')` は `PurePosixPath('bar')` と等価になりますが、`foo` が他のディレクトリへのシンボリックリンクの場合は等価になりません)

純粋パスオブジェクトは *os.PathLike* インターフェースを実装しており、そのインタフェースを受理する箇所ならどこでも使用することができます。

バージョン 3.6 で変更: *os.PathLike* インターフェースがサポートされました。

class pathlib.**PurePosixPath**(**pathsegments*)

PurePath のサブクラスです。このパスフレーバーは非 Windows パスを表します:

```
>>> PurePosixPath('/etc')
PurePosixPath('/etc')
```

pathsegments の指定は *PurePath* と同じです。

class pathlib.**PureWindowsPath**(**pathsegments*)

PurePath のサブクラスです。このパスフレーバーは Windows ファイルシステムパスを表します:

```
>>> PureWindowsPath('c:/Program Files/')
PureWindowsPath('c:/Program Files')
```

pathsegments の指定は *PurePath* と同じです。

これらクラスはあらゆるシステムコールを行わないため、起動しているシステムにかかわらずインスタンスを作成できます。

全般的な性質

パスオブジェクトはイミュータブルでハッシュ可能です。同じフレーバーのパスオブジェクトは比較ならびに順序付け可能です。これらのプロパティは、フレーバーのケースフォールディング (訳注: 比較のために正規化すること、例えば全て大文字にする) のセマンティクスに従います。

```
>>> PurePosixPath('foo') == PurePosixPath('FOO')
False
>>> PureWindowsPath('foo') == PureWindowsPath('FOO')
```

(次のページに続く)

(前のページからの続き)

```
True
>>> PureWindowsPath('FOO') in { PureWindowsPath('foo') }
True
>>> PureWindowsPath('C:') < PureWindowsPath('d:')
True
```

異なるフレーバーのパスオブジェクト同士の比較は等価になることはなく、順序付けもできません:

```
>>> PureWindowsPath('foo') == PurePosixPath('foo')
False
>>> PureWindowsPath('foo') < PurePosixPath('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'PureWindowsPath' and 'PurePosixPath'
```

演算子

演算子スラッシュ "/" はパスの追加を行います。*os.path.join()* と似ています:

```
>>> p = PurePath('/etc')
>>> p
PurePosixPath('/etc')
>>> p / 'init.d' / 'apache2'
PurePosixPath('/etc/init.d/apache2')
>>> q = PurePath('bin')
>>> '/usr' / q
PurePosixPath('/usr/bin')
```

os.PathLike を実装したオブジェクトが受理できる箇所ならどこでも、パスオブジェクトが使用できます:

```
>>> import os
>>> p = PurePath('/etc')
>>> os.fspath(p)
'/etc'
```

パスオブジェクトの文字列表現はそのシステム自身の Raw ファイルシステムパス (ネイティブの形式、例えば Windows では区切り文字がバックスラッシュ) になり、文字列としてファイルパスを取るあらゆる関数に渡すことができます:

```
>>> p = PurePath('/etc')
>>> str(p)
'/etc'
>>> p = PureWindowsPath('c:/Program Files')
>>> str(p)
'c:\\Program Files'
```

同様に、パスオブジェクトを *bytes* で呼び出すと、Raw ファイルシステムパスを *os.fsencode()* でエンコードされたバイト列オブジェクトで返します:

```
>>> bytes(p)
b'/etc'
```

注釈: *bytes* での呼び出しは Unix 上での使用のみ推奨します。Windows では Unicode 形式が標準的なファイルシステムパス表現になります。

個別の構成要素へのアクセス

パスの個別の ” 構成要素” へアクセスするには、以下のプロパティを使用します:

PurePath.parts

パスのさまざまな構成要素へのアクセス手段を提供するタプルになります:

```
>>> p = PurePath('/usr/bin/python3')
>>> p.parts
('/', 'usr', 'bin', 'python3')

>>> p = PureWindowsPath('c:/Program Files/PSF')
>>> p.parts
('c:\\', 'Program Files', 'PSF')
```

(ドライブ名とローカルルートは単一要素にまとめられます)

メソッドとプロパティ

純粋パスは以下のメソッドとプロパティを提供します:

PurePath.drive

ドライブ文字または名前を表す文字列があればそれになります:

```
>>> PureWindowsPath('c:/Program Files/').drive
'c:'
>>> PureWindowsPath('/Program Files/').drive
''
>>> PurePosixPath('/etc').drive
''
```

UNC 共有名もドライブとみなされます:

```
>>> PureWindowsPath('//host/share/foo.txt').drive
'\\\\host\\share'
```

PurePath.root

ローカルまたはグローバルルートを表す文字列があればそれになります:

```
>>> PureWindowsPath('c:/Program Files/').root
'\\'
>>> PureWindowsPath('c:Program Files/').root
''
>>> PurePosixPath('/etc').root
'/'
```

UNC 共有名は常にルートを持ちます:

```
>>> PureWindowsPath('//host/share').root
'\\'
```

PurePath.**anchor**

ドライブとルートを結合した文字列になります:

```
>>> PureWindowsPath('c:/Program Files/').anchor
'c:\\'
>>> PureWindowsPath('c:Program Files/').anchor
'c:'
>>> PurePosixPath('/etc').anchor
'/'
>>> PureWindowsPath('//host/share').anchor
'\\\\host\\share\\'
```

PurePath.**parents**

パスの論理的な上位パスにアクセスできるイミュータブルなシーケンスになります:

```
>>> p = PureWindowsPath('c:/foo/bar/setup.py')
>>> p.parents[0]
PureWindowsPath('c:/foo/bar')
>>> p.parents[1]
PureWindowsPath('c:/foo')
>>> p.parents[2]
PureWindowsPath('c:/')
```

PurePath.**parent**

パスの論理的な上位パスになります:

```
>>> p = PurePosixPath('/a/b/c/d')
>>> p.parent
PurePosixPath('/a/b/c')
```

アンカーの位置を超えることや空のパスになる位置には対応していません:

```
>>> p = PurePosixPath('/')
>>> p.parent
PurePosixPath('/')
>>> p = PurePosixPath('.')
>>> p.parent
PurePosixPath('.')
```

注釈: これは純粋な字句操作であるため、以下のような挙動になります:

```
>>> p = PurePosixPath('foo/..')
>>> p.parent
PurePosixPath('foo')
```

任意のファイルシステムパスを上位方向に移動したい場合、シンボリックリンクの解決や ".." 要素の除去のため、最初に *Path.resolve()* を呼ぶことを推奨します。

PurePath.**name**

パス要素の末尾を表す文字列があればそれになります。ドライブやルートは含まれません:

```
>>> PurePosixPath('my/library/setup.py').name
'setup.py'
```

UNC ドライブ名は考慮されません:

```
>>> PureWindowsPath('//some/share/setup.py').name
'setup.py'
>>> PureWindowsPath('//some/share').name
''
```

PurePath.**suffix**

末尾の要素に拡張子があればそれになります:

```
>>> PurePosixPath('my/library/setup.py').suffix
'.py'
>>> PurePosixPath('my/library.tar.gz').suffix
'.gz'
>>> PurePosixPath('my/library').suffix
''
```

PurePath.**suffixes**

パスのファイル拡張子のリストになります:

```
>>> PurePosixPath('my/library.tar.gar').suffixes
['.tar', '.gar']
>>> PurePosixPath('my/library.tar.gz').suffixes
['.tar', '.gz']
>>> PurePosixPath('my/library').suffixes
[]
```

PurePath.**stem**

パス要素の末尾から拡張子を除いたものになります:

```
>>> PurePosixPath('my/library.tar.gz').stem
'library.tar'
```

(次のページに続く)

(前のページからの続き)

```
>>> PurePosixPath('my/library.tar').stem
'library'
>>> PurePosixPath('my/library').stem
'library'
```

PurePath.as_posix()

フォワードスラッシュ (/) を使用したパスを表す文字列を返します:

```
>>> p = PureWindowsPath('c:\\windows')
>>> str(p)
'c:\\windows'
>>> p.as_posix()
'c:/windows'
```

PurePath.as_uri()

file URI で表したパスを返します。絶対パスではない場合に *ValueError* を送出します。

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.as_uri()
'file:///etc/passwd'
>>> p = PureWindowsPath('c:/Windows')
>>> p.as_uri()
'file:///c:/Windows'
```

PurePath.is_absolute()

パスが絶対パスかどうかを返します。パスが絶対パスとみなされるのは、ルートと (フレーバーが許す場合) ドライブとの両方が含まれる場合です:

```
>>> PurePosixPath('/a/b').is_absolute()
True
>>> PurePosixPath('a/b').is_absolute()
False

>>> PureWindowsPath('c:/a/b').is_absolute()
True
>>> PureWindowsPath('/a/b').is_absolute()
False
>>> PureWindowsPath('c:').is_absolute()
False
>>> PureWindowsPath('//some/share').is_absolute()
True
```

PurePath.is_reserved()

PureWindowsPath の場合はパスが Windows 上で予約されていれば True を返し、そうでなければ False を返します。*PurePosixPath* の場合は常に False を返します。

```
>>> PureWindowsPath('nul').is_reserved()
True
```

(次のページに続く)

(前のページからの続き)

```
>>> PurePosixPath('nul').is_reserved()
False
```

ファイルシステムで予約されたパスを呼び出すと、原因不明で失敗したり、予期せぬ結果になります。

PurePath.**joinpath**(**other*)

このメソッドの呼び出しは引数 *other* を順々に繋げることと等価になります:

```
>>> PurePosixPath('/etc').joinpath('passwd')
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath(PurePosixPath('passwd'))
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath('init.d', 'apache2')
PurePosixPath('/etc/init.d/apache2')
>>> PureWindowsPath('c:').joinpath('/Program Files')
PureWindowsPath('c:/Program Files')
```

PurePath.**match**(*pattern*)

現在のパスが glob 形式で与えられたパターンと一致したら `True` を、一致しなければ `False` を返します。

pattern が相対表記であればパスは相対および絶対パスを取ることができ、右から一致を調べます:

```
>>> PurePath('a/b.py').match('*.py')
True
>>> PurePath('/a/b/c.py').match('b/*.py')
True
>>> PurePath('/a/b/c.py').match('a/*.py')
False
```

pattern が絶対表記であれば、パスは絶対パスでなければならず、パス全体が一致しなければなりません:

```
>>> PurePath('/a.py').match('/*.py')
True
>>> PurePath('a/b.py').match('/*.py')
False
```

他のメソッドと同様に、大文字小文字の区別はプラットフォームの設定に従います:

```
>>> PurePosixPath('b.py').match('*.PY')
False
>>> PureWindowsPath('b.py').match('*.PY')
True
```

PurePath.**relative_to**(**other*)

other で表されたパスから現在のパスへの相対パスを返します。それが不可能だった場合は ValueError が送出されます:

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.relative_to('/')
PurePosixPath('etc/passwd')
>>> p.relative_to('/etc')
PurePosixPath('passwd')
>>> p.relative_to('/usr')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 694, in relative_to
    .format(str(self), str(formatted)))
ValueError: '/etc/passwd' does not start with '/usr'
```

PurePath.**with_name**(*name*)

現在のパスの *name* 部分を変更したパスを返します。オリジナルパスに *name* 部分がない場合は ValueError が送出されます:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_name('setup.py')
PureWindowsPath('c:/Downloads/setup.py')
>>> p = PureWindowsPath('c:/')
>>> p.with_name('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 751, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

PurePath.**with_suffix**(*suffix*)

suffix を変更した新しいパスを返します。元のパスに suffix が無かった場合、代わりに新しい *suffix* が追加されます。*suffix* が空文字列だった場合、元の suffix は除去されます:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_suffix('.bz2')
PureWindowsPath('c:/Downloads/pathlib.tar.bz2')
>>> p = PureWindowsPath('README')
>>> p.with_suffix('.txt')
PureWindowsPath('README.txt')
>>> p = PureWindowsPath('README.txt')
>>> p.with_suffix('')
PureWindowsPath('README')
```

11.1.3 具象パス

具象パスは純粋パスクラスのサブクラスです。純粋パスが提供する操作に加え、パスオブジェクト上でシステムコールを呼ぶメソッドも提供しています。具象パスのインスタンスを作成するには 3 つの方法があります:

class pathlib.Path(**pathsegments*)

PurePath のサブクラスであり、システムのパスフレーバーの具象パスを表します (このインスタンスの作成で *PosixPath* か *WindowsPath* のどちらかが作成されます):

```
>>> Path('setup.py')
PosixPath('setup.py')
```

pathsegments の指定は *PurePath* と同じです。

class pathlib.PosixPath(**pathsegments*)

Path および *PurePosixPath* のサブクラスで、非 Windows ファイルシステムの具象パスを表します:

```
>>> PosixPath('/etc')
PosixPath('/etc')
```

pathsegments の指定は *PurePath* と同じです。

class pathlib.WindowsPath(**pathsegments*)

Path および *PureWindowsPath* のサブクラスで、Windows ファイルシステムの具象パスを表します:

```
>>> WindowsPath('c:/Program Files/')
WindowsPath('c:/Program Files')
```

pathsegments の指定は *PurePath* と同じです。

インスタンスを作成できるのはシステムと一致するフレーバーのみです (互換性のないパスフレーバーでのシステムコールの許可はバグやアプリケーションの異常終了の原因になります):

```
>>> import os
>>> os.name
'posix'
>>> Path('setup.py')
PosixPath('setup.py')
>>> PosixPath('setup.py')
PosixPath('setup.py')
>>> WindowsPath('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 798, in __new__
    % (cls.__name__,))
NotImplementedError: cannot instantiate 'WindowsPath' on your system
```

メソッド

具象パスは純粋パスに加え、以下のメソッドを提供します。これらメソッドの多くはシステムコールが失敗すると *OSError* を送出します。(例えばパスが存在しない場合)

バージョン 3.8 で変更: *exists()*, *is_dir()*, *is_file()*, *is_mount()*, *is_symlink()*, *is_block_device()*, *is_char_device()*, *is_fifo()*, *is_socket()* は、OS レベルで表現不能な文字を含むパスに対して、例外を送出する代わりに False を返すようになりました。

classmethod Path.cwd()

(*os.getcwd()* が返す) 現在のディレクトリを表す新しいパスオブジェクトを返します:

```
>>> Path.cwd()
PosixPath('/home/antoine/pathlib')
```

classmethod Path.home()

ユーザーのホームディレクトリ (*os.path.expanduser()* での ~ の返り値) を表す新しいパスオブジェクトを返します:

```
>>> Path.home()
PosixPath('/home/antoine')
```

バージョン 3.5 で追加.

Path.stat()

(*os.stat()* と同様の) 現在のパスについて *os.stat_result* オブジェクトが含む情報を返します。値はそれぞれのメソッドを呼び出した時点のものになります。

```
>>> p = Path('setup.py')
>>> p.stat().st_size
956
>>> p.stat().st_mtime
1327883547.852554
```

Path.chmod(*mode*)

os.chmod() のようにファイルのモードとアクセス権限を変更します:

```
>>> p = Path('setup.py')
>>> p.stat().st_mode
33277
>>> p.chmod(0o444)
>>> p.stat().st_mode
33060
```

Path.exists()

パスが既存のファイルかディレクトリを指しているかどうかを返します:

```
>>> Path('.').exists()
True
>>> Path('setup.py').exists()
```

(次のページに続く)

(前のページからの続き)

```
True
>>> Path('/etc').exists()
True
>>> Path('nonexistentfile').exists()
False
```

注釈: パスがシンボリックリンクを指している場合、*exists()* はシンボリックリンクが既存のファイルかディレクトリを指しているかどうかを返します。

Path.**expanduser**()

パス要素 ~ および ~user を *os.path.expanduser()* が返すように展開した新しいパスオブジェクトを返します:

```
>>> p = PosixPath('~/films/Monty Python')
>>> p.expanduser()
PosixPath('/home/eric/films/Monty Python')
```

バージョン 3.5 で追加.

Path.**glob**(*pattern*)

現在のパスが表すディレクトリ内で相対 *pattern* に一致する (あらゆる種類の) すべてのファイルを yield します:

```
>>> sorted(Path('.').glob('*.py'))
[PosixPath('pathlib.py'), PosixPath('setup.py'), PosixPath('test_pathlib.py')]
>>> sorted(Path('.').glob('*/*.py'))
[PosixPath('docs/conf.py')]
```

パターン "**" は " このディレクトリおよびすべてのサブディレクトリを再帰的に走査" を意味します。言い換えれば、再帰的な Glob 走査が可能という意味です:

```
>>> sorted(Path('.').glob('**/*.py'))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

注釈: パターン "**" を大きなディレクトリツリーで使用するととてつもなく時間がかかるかもしれません。

Path.**group**()

ファイルを所有するグループ名を返します。ファイルの GID がシステムのデータベースに見つからなかった場合は *KeyError* が送出されます。

Path.**is_dir**()
パスがディレクトリ (またはディレクトリへのシンボリックリンク) を指していた場合 True を返し、その他の種類のファイルだった場合 False を返します。

パスが存在しないか壊れたシンボリックリンクだった場合にも False が返されます; (パーミッションエラーのような) その他のエラーは伝搬されます。

Path.**is_file**()
パスが一般ファイル (または一般ファイルへのシンボリックリンク) を指していた場合 True を返します。その他の種類のファイルを指していた場合 False を返します。

パスが存在しないか壊れたシンボリックリンクだった場合にも False が返されます; (パーミッションエラーのような) その他のエラーは伝搬されます。

Path.**is_mount**()
パス名 *path* がマウントポイント *mount point* (ファイルシステムの中で異なるファイルシステムがマウントされているところ) なら True を返します。POSIX では、この関数は *path* の親ディレクトリである path/.. が *path* と異なるデバイス上にあるか、あるいは path/.. と *path* が同じデバイス上の同じ i-node を指しているかをチェックします --- これによってすべての Unix と POSIX 系システムでマウントポイントが検出できます。この関数は Windows では実装されていません。

バージョン 3.7 で追加.

Path.**is_symlink**()
パスがシンボリックリンクを指していた場合 True を返し、その他の場合は False を返します。

パスが存在しない場合も False を返します; (パーミッションエラーのような) その他のエラーは伝搬されます。

Path.**is_socket**()
パスが Unix ソケット (または Unix ソケットへのシンボリックリンク) を指していた場合 True を返します。その他の種類のファイルの場合 False を返します。

パスが存在しないか壊れたシンボリックリンクだった場合にも False が返されます; (パーミッションエラーのような) その他のエラーは伝搬されます。

Path.**is_fifo**()
パスが FIFO (または FIFO へのシンボリックリンク) を指していた場合 True を返します。その他の種類のファイルの場合は False を返します。

パスが存在しないか壊れたシンボリックリンクだった場合にも False が返されます; (パーミッションエラーのような) その他のエラーは伝搬されます。

Path.**is_block_device**()
パスがブロックデバイス (またはブロックデバイスへのシンボリックリンク) を指していた場合 True を返します。その他の種類のファイルの場合は False を返します。

パスが存在しないか壊れたシンボリックリンクだった場合にも False が返されます; (パーミッションエラーのような) その他のエラーは伝搬されます。

Path.**is_char_device**()
パスがキャラクターデバイス (またはキャラクターデバイスへのシンボリックリンク) を指していた場合、True を返します。その他の種類のファイルの場合 False を返します。

パスが存在しないか壊れたシンボリックリンクだった場合にも False が返されます; (パーミッションエラーのような) その他のエラーは伝搬されます。

Path.**iterdir**()
パスがディレクトリを指していた場合、ディレクトリの内容のパスオブジェクトを yield します:

```
>>> p = Path('docs')
>>> for child in p.iterdir(): child
...
PosixPath('docs/conf.py')
PosixPath('docs/_templates')
PosixPath('docs/make.bat')
PosixPath('docs/index.rst')
PosixPath('docs/_build')
PosixPath('docs/_static')
PosixPath('docs/Makefile')
```

The children are yielded in arbitrary order, and the special entries '.' and '..' are not included. If a file is removed from or added to the directory after creating the iterator, whether an path object for that file be included is unspecified.

Path.**lchmod**(*mode*)
Path.chmod() のように振る舞いますが、パスがシンボリックリンクを指していた場合、リンク先ではなくシンボリックリンク自身のモードが変更されます。

Path.**lstat**()
Path.stat() のように振る舞いますが、パスがシンボリックリンクを指していた場合、リンク先ではなくシンボリックリンク自身の情報を返します。

Path.**mkdir**(*mode=0o777*, *parents=False*, *exist_ok=False*)
与えられたパスに新しくディレクトリを作成します。*mode* が与えられていた場合、プロセスの umask 値と組み合わせてファイルのモードとアクセスフラグを決定します。パスがすでに存在していた場合 *FileExistsError* が送出されます。

parents の値が真の場合、このパスの親ディレクトリを必要に応じて作成します; それらのアクセス制限はデフォルト値が取られ、*mode* は使用されません (POSIX の mkdir -p コマンドを真似ています)。

parents の値が偽の場合 (デフォルト)、親ディレクトリがないと *FileNotFoundError* を送出します。

exist_ok の値が (デフォルトの) 偽の場合、対象のディレクトリがすでに存在すると *FileExistsError* を送出します。

exist_ok の値が真の場合、パス要素の末尾がすでに存在するがディレクトリではないときは *FileExistsError* 例外を送出しますが、ディレクトリであれば送出しません (POSIX の mkdir -p コマンドの挙動と同じ)。

バージョン 3.5 で変更: *exist_ok* 引数が追加されました。

Path.**open**(*mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)

組み込み関数 *open()* のようにパスが指しているファイルを開きます:

```
>>> p = Path('setup.py')
>>> with p.open() as f:
...     f.readline()
...
'#!/usr/bin/env python3\n'
```

Path.**owner**()

ファイルの所有者のユーザー名を返します。ファイルの UID がシステムのデータベースに見つからない場合 *KeyError* が送出されます。

Path.**read_bytes**()

指定されたファイルの内容をバイナリオブジェクトで返します:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

バージョン 3.5 で追加.

Path.**read_text**(*encoding=None*, *errors=None*)

指定されたファイルの内容を文字列としてデコードして返します:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

ファイルを開いた後に閉じます。オプションのパラメーターの意味は *open()* と同じです。

バージョン 3.5 で追加.

Path.**rename**(*target*)

このファイルかディレクトリを与えられた *target* にリネームし、*target* を指すパスのインスタンスを返します。Unix では *target* が存在するファイルの場合、ユーザにパーミッションがあれば静かに置換されます。*target* は文字列か別のパスオブジェクトです:

```
>>> p = Path('foo')
>>> p.open('w').write('some text')
9
>>> target = Path('bar')
>>> p.rename(target)
PosixPath('bar')
>>> target.open().read()
'some text'
```

target パスは絶対または相対で指定できます。相対パスは現在の作業ディレクトリからの相対パスとして解釈し、Path オブジェクトのディレクトリ **ではありません**。

バージョン 3.8 で変更: 戻り値を追加し、新しい Path インスタンスを返します。

Path.**replace**(*target*)

現在のファイルまたはディレクトリの名前を *target* に変更し、*target* を指すパスのインスタンスを返します。*target* が既存のファイルかディレクトリを指していた場合、無条件に置き換えられます。

target パスは絶対または相対で指定できます。相対パスは現在の作業ディレクトリからの相対パスとして解釈し、Path オブジェクトのディレクトリ **ではありません**。

バージョン 3.8 で変更: 戻り値を追加し、新しい Path インスタンスを返します。

Path.**resolve**(*strict=False*)

パスを絶対パスにし、あらゆるシンボリックリンクを解決します。新しいパスオブジェクトが返されます:

```
>>> p = Path()
>>> p
PosixPath('.')
>>> p.resolve()
PosixPath('/home/antoine/pathlib')
```

"`..`" 要素は除去されます (このような挙動を示すのはこのメソッドだけです):

```
>>> p = Path('docs/../setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
```

パスが存在せず *strict* が `True` の場合、*FileNotFoundError* が送出されます。*strict* が `False` の場合は、パスは可能な限り解決され、残りの部分は存在するかのチェックをせずに追加されます。もしパスの解決にあたって無限ループする場合は、*RuntimeError* が送出されます。

バージョン 3.6 で追加: *strict* 引数 (3.6 以前の挙動は strict です。)

Path.**rglob**(*pattern*)

これは *Path.glob()* を *pattern* の先頭に "`**/`" を追加して呼び出した場合と似ています:

```
>>> sorted(Path().rglob("*.py"))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

Path.**rmdir**()

現在のディレクトリを削除します。ディレクトリは空でなければなりません。

Path.**samefile**(*other_path*)

このパスが参照するファイルが *other_path* (Path オブジェクトか文字列) と同じであれば `True` を、

異なるファイルであれば False を返します。意味的には *os.path.samefile()* および *os.path.samestat()* と同じです。

なんらかの理由でどちらかのファイルにアクセスできない場合は *OSError* が送出されます。

```
>>> p = Path('spam')
>>> q = Path('eggs')
>>> p.samefile(q)
False
>>> p.samefile('spam')
True
```

バージョン 3.5 で追加.

Path.symlink_to(*target*, *target_is_directory=False*)

現在のパスに *target* へのシンボリックリンクを作成します。Windows では、リンク対象がディレクトリの場合 *target_is_directory* が真でなければなりません (デフォルトは False)。POSIX では、*target_is_directory* の値は無視されます。

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
>>> p.stat().st_size
956
>>> p.lstat().st_size
8
```

注釈: 引数の並び (link, target) は *os.symlink()* とは逆です。

Path.link_to(*target*)

Make *target* a hard link to this path.

警告: This function does not make this path a hard link to *target*, despite the implication of the function and argument names. The argument order (target, link) is the reverse of *Path.symlink_to()*, but matches that of *os.link()*.

バージョン 3.8 で追加.

Path.touch(*mode=0o666*, *exist_ok=True*)

与えられたパスにファイルを作成します。*mode* が与えられた場合、プロセスの umask 値と組み合わせてファイルのモードとアクセスフラグが決定されます。ファイルがすでに存在した場合、*exist_ok* が真ならばこの関数は正常に終了します (そしてファイルの更新日付が現在の日時に変更されます)。その他の場合は *FileExistsError* が送出されます。

Path.unlink(*missing_ok=False*)

このファイルまたはシンボリックリンクを削除します。パスがディレクトリを指している場合は *Path.rmdir()* を使用してください。

missing_ok の値が (デフォルトの) 偽の場合、対象のファイルが存在しないと *FileNotFoundError* を送出します。

missing_ok の値が真の場合、*FileExistsError* 例外を送出しません (POSIX の `rm -f` コマンドの挙動と同じ)。

バージョン 3.8 で変更: *missing_ok* 引数が追加されました。

Path.write_bytes(*data*)

指定されたファイルをバイトモードで開き、*data* を書き込み、ファイルを閉じます:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

同じ名前のファイルがすでにあれば上書きされます。

バージョン 3.5 で追加.

Path.write_text(*data*, *encoding=None*, *errors=None*)

指定されたファイルをテキストモードで開き、*data* を書き込み、ファイルを閉じます:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

同じ名前のファイルが存在する場合は上書きされます。オプションのパラメーターの意味は *open()* と同じです。

バージョン 3.5 で追加.

11.1.4 os モジュールにあるツールとの対応付け

下にあるのは、様々な *os* 関数とそれに相当する *PurePath* あるいは *Path* の同等のものとの対応表です。

注釈: *os.path.relpath()* および *PurePath.relative_to()* は使い道が重なるところもありますが、それらの意味論は同等だと見なすには違い過ぎています。

---

os および os.path	pathlib
os.path.abspath()	*Path.resolve()*
os.chmod()	*Path.chmod()*
os.mkdir()	*Path.mkdir()*
os.rename()	*Path.rename()*
os.replace()	*Path.replace()*
os.rmdir()	*Path.rmdir()*
os.remove(), *os.unlink()*	*Path.unlink()*
os.getcwd()	*Path.cwd()*
os.path.exists()	*Path.exists()*
os.path.expanduser()	*Path.expanduser()* および *Path.home()*
os.listdir()	*Path.iterdir()*
os.path.isdir()	*Path.is_dir()*
os.path.isfile()	*Path.is_file()*
os.path.islink()	*Path.is_symlink()*
os.link()	*Path.link_to()*
os.symlink()	*Path.symlink_to()*
os.stat()	*Path.stat()*, *Path.owner()*, *Path.group()*
os.path.isabs()	*PurePath.is_absolute()*
os.path.join()	*PurePath.joinpath()*
os.path.basename()	*PurePath.name*
os.path.dirname()	*PurePath.parent*
os.path.samefile()	*Path.samefile()*
os.path.splitext()	*PurePath.suffix*

11.2 os.path --- 共通のパス名操作

ソースコード: Lib/posixpath.py (POSIX), Lib/ntpath.py (Windows NT)

このモジュールには、パス名を操作する便利な関数が実装されています。ファイルの読み書きに関しては *open()* を、ファイルシステムへのアクセスに関しては *os* モジュールを参照してください。パスパラメータは文字列またはバイト列で渡すことができます。アプリケーションは、ファイル名を Unicode 文字列で表すことが推奨されています。残念ながら、Unix では文字列で表すことのできないファイル名があるため、Unix 上で任意のファイル名をサポートする必要のあるアプリケーションは、そのパス名にバイト列を使用すべきです。逆に、バイト列オブジェクトを使用すると Windows (標準の `mbcs` エンコーディング) 上ではすべてのファイル名を表すことができないため、Windows アプリケーションはファイルアクセスのために文字列オブジェクトを使用するべきです。

Unix シェルとは異なり、Python はあらゆるパス展開を **自動的には** 行いません。アプリケーションがシェルのようなパス展開を必要とした場合は、*expanduser()* や *expandvars()* といった関数を明示的に呼び出すことで行えます。(*glob* モジュールも参照してください)

参考:

pathlib モジュールは高水準のパスオブジェクトを提供します。

注釈: 以下のすべての関数は、そのパラメータにバイト列のみ、あるいは文字列のみ受け付けます。パスまたはファイル名を返す場合、返り値は同じ型のオブジェクトになります。

注釈: OS によって異なるパス名の決まりがあるため、標準ライブラリにはこのモジュールのいくつかのバージョンが含まれています。*os.path* モジュールは常に現在 Python が動作している OS に適したパスモジュールであるため、ローカルのパスを扱うのに適しています。各々のモジュールをインポートして **常に** 一つのフォーマットを利用することも可能です。これらはすべて同じインタフェースを持っています:

- `posixpath` UNIX スタイルのパス用
- `ntpath` Windows パス用

バージョン 3.8 で変更: *exists()*、*lexists()*、*isdir()*、*isfile()*、*islink()*、および *ismount()* は、OS レベルで表現できない文字列を含む可能性がある例外を送出する代わりに `False` を返すようになりました。

`os.path.abspath(`*path*`)`
: パス名 *path* の正規化された絶対パスを返します。ほとんどのプラットフォームでは、これは関数 *normpath()* を次のように呼び出した時と等価です: `normpath(join(os.getcwd(), path))`。

  バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

`os.path.basename(`*path*`)`
: パス名 *path* の末尾のファイル名部分を返します。これは関数 *split()* に *path* を渡した時に返されるペアの 2 番めの要素です。この関数が返すのは Unix の **basename** とは異なります; Unix の **basename** は `'/foo/bar/'` に対して `'bar'` を返しますが、関数 *basename()* は空文字列 (`''`) を返します。

  バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

`os.path.commonpath(`*paths*`)`
: シーケンス *paths* 中の各パス名に共通するサブパスのうち、最も長いものを返します。*paths* に絶対パス名と相対パス名の両方が含まれている、*paths* が異なるドライブ上にある、または *paths* が空の場合、*ValueError* を送出します。*commonprefix()* とは異なり、有効なパスを返します。

  Availability: Unix, Windows.

  バージョン 3.5 で追加.

  バージョン 3.6 で変更: *path-like objects* のシーケンスを受け入れるようになりました。

`os.path.commonprefix(`*list*`)`
: *list* 内のすべてのパスに共通する接頭辞のうち、最も長いものを (パス名の 1 文字 1 文字を判断して) 返します。*list* が空の場合、空文字列 (`''`) を返します。

注釈: この関数は一度に 1 文字ずつ処理するため、不正なパスを返す場合があります。有効なパスを取得するためには、*commonpath()* を参照してください。

```
>>> os.path.commonprefix(['/usr/lib', '/usr/local/lib'])
'/usr/l'

>>> os.path.commonpath(['/usr/lib', '/usr/local/lib'])
'/usr'
```

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.path.**dirname**(*path*)

パス名 *path* のディレクトリ名を返します。これは関数 *split()* に *path* を渡した時に返されるペアの 1 番めの要素です。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.path.**exists**(*path*)

path が実在するパスかオープンしているファイル記述子を参照している場合 `True` を返します。壊れたシンボリックリンクについては `False` を返します。一部のプラットフォームでは、たとえ *path* が物理的に存在していたとしても、要求されたファイルに対する *os.stat()* の実行権がなければこの関数が `False` を返すことがあります。

バージョン 3.3 で変更: *path* は整数でも可能になりました: それがオープンしているファイル記述子なら `True` が返り、それ以外なら `False` が返ります。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.path.**lexists**(*path*)

path が実在するパスなら `True` を返します。壊れたシンボリックリンクについては `True` を返します。*os.lstat()* がない環境では *exists()* と等価です。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.path.**expanduser**(*path*)

Unix および Windows では、与えられた引数の先頭のパス要素 `~` 、または `~user` を、*user* のホームディレクトリのパスに置き換えて返します。

Unix では、先頭の `~` は、環境変数 `HOME` が設定されているならその値に置き換えられます。設定されていない場合は、現在のユーザのホームディレクトリをビルトインモジュール *pwd* を使ってパスワードディレクトリから探して置き換えます。先頭の `~user` については、直接パスワードディレクトリから探します。

Windows では、`USERPROFILE` が設定されていればそれを使用します。設定されていない場合は、環境変数 `HOMEPATH` と `HOMEDRIVE` の組み合わせで置き換えられます。先頭の `~user` は `~` で得られるユーザパスの最後のディレクトリ要素を除去したものを利用します。

置き換えに失敗したり、引数のパスがチルダで始まっていなかった場合は、パスをそのまま返します。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

バージョン 3.8 で変更: Windows で `HOME` は参照しなくなりました。

`os.path.expandvars`(*path*)
引数のパスの環境変数を展開して返します。引数の中の `$name` または `${name}` のような形式の文字列は環境変数、*name* の値に置き換えられます。不正な変数名や存在しない変数名の場合には変換されず、そのまま返します。

Windows では、`$name` や `${name}` の形式に加えて、`%name%` の形式もサポートされています。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

`os.path.getatime`(*path*)
path に最後にアクセスした時刻を返します。返り値は、エポック (*time* モジュールを参照) からの経過秒数を与える浮動小数点数です。ファイルが存在しない、あるいはアクセスできなかった場合は *OSError* を送出します。

`os.path.getmtime`(*path*)
path に最後に更新した時刻を返します。返り値は、エポック (*time* モジュールを参照) からの経過秒数を与える浮動小数点数です。ファイルが存在しない、あるいはアクセスできなかった場合は *OSError* を送出します。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

`os.path.getctime`(*path*)
システムの ctime、Unix 系など一部のシステムでは最後にメタデータが変更された時刻、Windows などその他のシステムでは *path* の作成時刻を返します。返り値はエポック (*time* モジュールを参照) からの経過時間を示す秒数になります。ファイルが存在しない、あるいはアクセスできなかった場合は *OSError* を送出します。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

`os.path.getsize`(*path*)
path のサイズをバイト数で返します。ファイルが存在しない、あるいはアクセスできなかった場合は *OSError* を送出します。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

`os.path.isabs`(*path*)
path が絶対パスなら `True` を返します。すなわち、Unix ではスラッシュで始まり、Windows ではドライブレターに続く (バック) スラッシュで始まる場合です。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

`os.path.isfile`(*path*)
path が **存在する** 一般ファイルなら `True` を返します。この関数はシンボリックリンクの先を辿るので、同じパスに対して *islink()* と *isfile()* の両方が真を返すことがあります。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.path.isdir(*path*)

path が **存在する** ディレクトリなら True を返します。この関数はシンボリックリンクの先を辿るので、同じパスに対して *islink()* と *isdir()* の両方が真を返すことがあります。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.path.islink(*path*)

path が **存在する** ディレクトリを指すシンボリックリンクなら True を返します。Python ランタイムがシンボリックリンクをサポートしていないプラットフォームでは、常に False を返します。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.path.ismount(*path*)

パス名 *path* がマウントポイント *mount point* (ファイルシステムの中で異なるファイルシステムがマウントされているところ) なら、True を返します。POSIX では、この関数は *path* の親ディレクトリである *path*/.. が *path* と異なるデバイス上にあるか、あるいは *path*/.. と *path* が同じデバイス上の同じ i-node を指しているかをチェックします --- これによって全ての Unix 系システムと POSIX 標準でマウントポイントが検出できます。だたし、同じファイルシステムの bind mount の信頼できる検出はできません。Windows では、ドライブレターを持つルートと共有 UNC は常にマウントポイントであり、また他のパスでは、入力のパスが異なるデバイスからのものか見るために GetVolumePathName が呼び出されます。

バージョン 3.4 で追加: Windows での、ルートでないマウントポイントの検出をサポートするようになっています。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.path.join(*path*, **paths*)

1 つあるいはそれ以上のパスの要素を賢く結合します。戻り値は *path*、ディレクトリの区切り文字を **paths* の各パートの (末尾でない場合の空文字列を除いて) 頭に付けたもの、これらの結合になります。最後の部分が空文字列の場合に限り区切り文字で終わる文字列になります。付け加える要素に絶対パスがあれば、それより前の要素は全て破棄され、以降の要素を結合します。

Windows の場合は、絶対パスの要素 (たとえば r'\foo') が見つかった場合はドライブレターはリセットされません。要素にドライブレターが含まれていれば、それより前の要素は全て破棄され、ドライブレターがリセットされます。各ドライブに対してカレントディレクトリがあるので、os.path.join("c:", "foo") によって、c:\foo ではなく、ドライブ C: 上のカレントディレクトリからの相対パス (c:foo) が返されることに注意してください。

バージョン 3.6 で変更: *path* と *paths* が *path-like object* を受け付けるようになりました。

os.path.normcase(*path*)

パス名の大文字・小文字を正規化します。Windows では、パス名にある文字を全て小文字に、スラッシュをバックスラッシュに変換します。他のオペレーティングシステムでは、パスを変更せずに返します。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.path.normpath(*path*)

パスを正規化します。余分な区切り文字や上位レベル参照を除去し、A//B、A/B/、A/./B や A/foo/../B などはすべて A/B になります。この文字列操作は、シンボリックリンクを含むパスの意味を変えてしまう場合があります。Windows では、スラッシュをバックスラッシュに変換します。大文字小文字の正規化には *normcase()* を使用してください。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.path.**realpath**(*path*)

パスの中のシンボリックリンク (もしそれが当該オペレーティングシステムでサポートされていれば) を取り除いて、指定されたファイル名を正規化したパスを返します。

注釈: シンボリックリンクが循環している場合、循環したリンクのうちの一つのパスが返されます。ただし、どのパスが返されるかは保証されません。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

バージョン 3.8 で変更: Windows においてシンボリックリンクとジャンクションが解決されるようになりました。

os.path.**relpath**(*path*, *start=os.curdir*)

カレントディレクトリあるいはオプションの *start* ディレクトリからの *path* への相対パスを返します。これはパス計算で行っており、ファイルシステムにアクセスして *path* や *start* の存在や性質を確認することはありません。Windows では、*path* と *start* が異なるドライブの場合、*ValueError* を送出します。

start のデフォルト値は *os.curdir* です。

Availability: Unix, Windows.

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.path.**samefile**(*path1*, *path2*)

引数の両パス名が同じファイルまたはディレクトリを参照している場合、True を返します。これは、デバイス番号と i-node 番号で決定されます。どちらかのパス名への *os.stat()* 呼び出しが失敗した場合、例外が送出されます。

Availability: Unix, Windows.

バージョン 3.2 で変更: Windows サポートを追加しました。

バージョン 3.4 で変更: Windows が他のプラットフォームと同じ実装を使用するようになりました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.path.**sameopenfile**(*fp1*, *fp2*)

ファイル記述子 *fp1* と *fp2* が同じファイルを参照していたら True を返します。

Availability: Unix, Windows.

バージョン 3.2 で変更: Windows サポートを追加しました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.path.**samestat**(*stat1*, *stat2*)

stat タプル *stat1* と *stat2* が同じファイルを参照していれば True を返します。これらのタプルは *os.fstat()* 、*os.lstat()* あるいは *os.stat()* の返り値で構いません。この関数は *samefile()* と *sameopenfile()* を使用した比較に基いて実装しています。

Availability: Unix, Windows.

バージョン 3.4 で変更: Windows サポートを追加しました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.path.**split**(*path*)

パス名 *path* を (head, tail) のペアに分割します。*tail* はパス名の構成要素の末尾で、*head* はそれより前の部分です。*tail* はスラッシュを含みません; もし *path* がスラッシュで終わっていれば *tail* は空文字列になります。もし *path* にスラッシュがなければ、*head* は空文字になります。*path* が空文字なら、*head* と *tail* の両方が空文字になります。*head* の末尾のスラッシュは *head* がルートディレクトリ (または 1 個以上のスラッシュだけ) でない限り取り除かれます。join(head, tail) は常に *path* と同じ場所を返しますが、文字列としては異なるかもしれません。関数 *dirname()*, *basename()* も参照してください。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.path.**splitdrive**(*path*)

パス名 *path* を (drive, tail) のペアに分割します。*drive* はマウントポイントか空文字列になります。ドライブ指定をサポートしていないシステムでは、*drive* は常に空文字になります。どの場合でも、drive + tail は *path* と等しくなります。

Windows では、パス名はドライブ名/UNC 共有ポイントと相対パスに分割されます。

パスがドライブレターを含む場合、ドライブレターにはコロンまでが含まれます。例えば、splitdrive("c:/dir") は ("c:", "/dir") を返します。

パスが UNC パスを含む場合、ドライブレターにはホスト名と共有名までが含まれますが、共有名の後の区切り文字は含まれません。例えば、splitdrive("//host/computer/dir") は ("//host/computer", "/dir") を返します。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.path.**splitext**(*path*)

パス名 *path* を (root, ext) のペアに分割します。root + ext == path になります。*ext* は空文字列か 1 つのピリオドで始まり、多くても 1 つのピリオドを含みます。ベースネームを導出するピリオドは無視されます; splitext('.cshrc') は ('.cshrc', '') を返します。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.path.**supports_unicode_filenames**

ファイル名に任意の Unicode 文字列を (システムの制限内で) 使用できる場合は True になります。

11.3 fileinput --- 複数の入力ストリームをまたいだ行の繰り返し処理をサポートする

ソースコード: Lib/fileinput.py

このモジュールは標準入力やファイルの並びにまたがるループを素早く書くためのヘルパークラスと関数を提供しています。単一のファイルを読み書きしたいだけなら、*open()* を参照してください。

典型的な使い方は以下の通りです:

```
import fileinput
for line in fileinput.input():
    process(line)
```

このスクリプトは sys.argv[1:] に列挙されている全てのファイルの行に渡って反復処理を行います。もし列挙されているものがなければ、sys.stdin がデフォルトとして扱われます。ファイル名として '-' が与えられた場合も、sys.stdin に置き換えられ、*mode* と *openhook* は無視されます。別のファイル名リストを使いたい時には、そのリストを *input()* の最初の引数に与えます。ファイル名が 1 つでも受け付けます。

全てのファイルはデフォルトでテキストモードでオープンされます。しかし、*input()* や *FileInput* をコールする際に *mode* パラメータを指定すれば、これをオーバーライドすることができます。オープン中あるいは読み込み中に I/O エラーが発生した場合には、*OSError* が発生します。

バージョン 3.3 で変更: 以前は *IOError* が送出されました; それは現在 *OSError* のエイリアスです。

sys.stdin が 2 回以上使われた場合は、2 回目以降は行を返しません。ただしインタラクティブに利用している時や明示的にリセット (sys.stdin.seek(0) を使う) を行った場合はその限りではありません。

空のファイルは開いた後すぐ閉じられます。空のファイルはファイル名リストの最後にある場合にしか外部に影響を与えません。

ファイルの各行は、各種改行文字まで含めて返されます。ファイルの最後が改行文字で終っていない場合には、改行文字で終わらない行が返されます。

ファイルのオープン方法を制御するためのオープン時フックは、*fileinput.input()* あるいは *FileInput()* の *openhook* パラメータで設定します。このフックは、ふたつの引数 *filename* と *mode* をとる関数でなければなりません。そしてその関数の返り値はオープンしたファイルオブジェクトとなります。このモジュールには、便利なフックが既に用意されています。

以下の関数がこのモジュールの基本的なインタフェースです:

fileinput.input(*files=None*, *inplace=False*, *backup=''*, ***, *mode='r'*, *openhook=None*)

FileInput クラスのインスタンスを作ります。生成されたインスタンスは、このモジュールの関数群が利用するグローバルな状態として利用されます。この関数への引数は *FileInput* クラスのコンストラクタへ渡されます。

FileInput のインスタンスは with 文の中でコンテキストマネージャーとして使用できます。次の例では、仮に例外が生じたとしても with 文から抜けた後で *input* は閉じられます:

```
with fileinput.input(files=('spam.txt', 'eggs.txt')) as f:
    for line in f:
        process(line)
```

バージョン 3.2 で変更: コンテキストマネージャとして使うことができるようになりました。

バージョン 3.8 で変更: キーワード引数 *mode* と *openhook* は、キーワード専用引数になりました。

以下の関数は *fileinput.input()* 関数によって作られたグローバルな状態を利用します。アクティブな状態が無い場合には、*RuntimeError* が発生します。

fileinput.filename()
現在読み込み中のファイル名を返します。一行目が読み込まれる前は None を返します。

fileinput.fileno()
現在のファイルの ” ファイル記述子” を整数値で返します。ファイルがオープンされていない場合 (最初の行の前、ファイルとファイルの間) は -1 を返します。

fileinput.lineno()
最後に読み込まれた行の、累積した行番号を返します。1 行目が読み込まれる前は 0 を返します。最後のファイルの最終行が読み込まれた後には、その行の行番号を返します。

fileinput.filelineno()
現在のファイル中での行番号を返します。1 行目が読み込まれる前は 0 を返します。最後のファイルの最終行が読み込まれた後には、その行のファイル中での行番号を返します。

fileinput.isfirstline()
最後に読み込まれた行がファイルの 1 行目なら True 、そうでなければ False を返します。

fileinput.isstdin()
最後に読み込まれた行が sys.stdin から読み込まれていれば True 、そうでなければ False を返します。

fileinput.nextfile()
現在のファイルを閉じます。次の繰り返しでは (存在すれば) 次のファイルの最初の行が読み込まれます。閉じたファイルの読み込まれなかった行は、累積の行数にカウントされません。ファイル名は次のファイルの最初の行が読み込まれるまで変更されません。最初の行の読み込みが行われるまでは、この関数は呼び出されても何もしませんので、最初のファイルをスキップするために利用することはできません。最後のファイルの最終行が読み込まれた後にも、この関数は呼び出されても何もしません。

fileinput.close()
シーケンスを閉じます。

このモジュールのシーケンスの振舞いを実装しているクラスのサブクラスを作ることもできます:

class fileinput.FileInput(*files=None*, *inplace=False*, *backup=”*, ***, *mode='r'*, *openhook=None*)
FileInput クラスはモジュールの関数に対応するメソッド *filename()* 、*fileno()* 、*lineno()* 、*filelineno()* 、*isfirstline()* 、*isstdin()* 、*nextfile()* および *close()* を実装しています。それに加えて、次の入力行を返す *readline()* メソッドと、シーケンスの振舞いの実装をしている

__getitem__() メソッドがあります。シーケンスはシーケンシャルに読み込むことしかできません。つまりランダムアクセスと *readline()* を混在させることはできません。

mode を使用すると、*open()* に渡すファイルモードを指定することができます。これは 'r' 、'rU' 、'U' および 'rb' のうちのいずれかとなります。

openhook を指定する場合は、ふたつの引数 *filename* と *mode* をとる関数でなければなりません。この関数の返り値は、オープンしたファイルオブジェクトとなります。*inplace* と *openhook* を同時に使うことはできません。

FileInput のインスタンスは with 文の中でコンテキストマネージャーとして使用できます。次の例では、仮に例外が生じたとしても with 文から抜けた後で *input* は閉じられます:

```
with FileInput(files=('spam.txt', 'eggs.txt')) as input:
    process(input)
```

バージョン 3.2 で変更: コンテキストマネージャとして使うことができるようになりました。

バージョン 3.4 で非推奨: 'rU' および 'U' モード。

バージョン 3.8 で非推奨: meth:*___getitem___* メソッドのサポートは非推奨になりました。

バージョン 3.8 で変更: キーワード引数 *mode* と *openhook* は、キーワード専用引数になりました。

インプレース (in-place) フィルタオプション: キーワード引数 inplace=True が *fileinput.input()* か *FileInput* クラスのコンストラクタに渡された場合には、入力ファイルはバックアップファイルに移動され、標準出力が入力ファイルに設定されます (バックアップファイルと同じ名前のファイルが既に存在していた場合には、警告無しに置き替えられます)。これによって入力ファイルをその場で書き替えるフィルタを書くことができます。キーワード引数 *backup* (通常は backup='.<拡張子>' という形で利用します) が与えられていた場合、バックアップファイルの拡張子として利用され、バックアップファイルは削除されずに残ります。デフォルトでは、拡張子は '.bak' になっていて、出力先のファイルが閉じられればバックアップファイルも消されます。インプレースフィルタ機能は、標準入力を読み込んでいる間は無効にされます。

このモジュールには、次のふたつのオープン時フックが用意されています:

fileinput.**hook_compressed**(*filename*, *mode*)

gzip や bzip2 で圧縮された (拡張子が '.gz' や '.bz2' の) ファイルを、*gzip* モジュールや *bz2* モジュールを使って透過的にオープンします。ファイルの拡張子が '.gz' や '.bz2' でない場合は、通常通りファイルをオープンします (つまり、*open()* をコールする際に伸長を行いません)。

使用例: fi = fileinput.FileInput(openhook=fileinput.hook_compressed)

fileinput.**hook_encoded**(*encoding*, *errors=None*)

各ファイルを *open()* でオープンするフックを返します。指定した *encoding* および *errors* でファイルを読み込みます。

使用例: fi = fileinput.FileInput(openhook=fileinput.hook_encoded("utf-8", "surrogateescape"))

バージョン 3.6 で変更: オプションの *errors* 引数が追加されました。

11.4 stat --- stat() の結果を解釈する

ソースコード: Lib/stat.py

---

stat モジュールでは、*os.stat()* 、*os.lstat()* 、および *os.fstat()* が存在する場合に、これらの関数が返す内容を解釈するための定数や関数を定義しています。stat() 、fstat() 、および lstat() の関数呼び出しについての完全な記述はシステムのドキュメントを参照してください。

バージョン 3.4 で変更: stat モジュールは、C 実装に裏付けされるようになりました。

stat モジュールでは、特殊なファイル型を判別するための以下の関数を定義しています:

stat.S_ISDIR(*mode*)
: ファイルのモードがディレクトリの場合にゼロでない値を返します。

stat.S_ISCHR(*mode*)
: ファイルのモードがキャラクタ型の特殊デバイスファイルの場合にゼロでない値を返します。

stat.S_ISBLK(*mode*)
: ファイルのモードがブロック型の特殊デバイスファイルの場合にゼロでない値を返します。

stat.S_ISREG(*mode*)
: ファイルのモードが通常ファイルの場合にゼロでない値を返します。

stat.S_ISFIFO(*mode*)
: ファイルのモードが FIFO (名前つきパイプ) の場合にゼロでない値を返します。

stat.S_ISLNK(*mode*)
: ファイルのモードがシンボリックリンクの場合にゼロでない値を返します。

stat.S_ISSOCK(*mode*)
: ファイルのモードがソケットの場合にゼロでない値を返します。

stat.S_ISDOOR(*mode*)
: ファイルのモードがドアの場合にゼロでない値を返します。

  バージョン 3.4 で追加.

stat.S_ISPORT(*mode*)
: ファイルのモードがイベントポートの場合にゼロでない値を返します。

  バージョン 3.4 で追加.

stat.S_ISWHT(*mode*)
: ファイルのモードがホワイトアウトの場合にゼロでない値を返します。

  バージョン 3.4 で追加.

より一般的なファイルのモードを操作するための二つの関数が定義されています:

stat.S_IMODE(*mode*)

os.chmod() で設定することのできる一部のファイルモード --- すなわち、ファイルの許可ビット (permission bits) に加え、(サポートされているシステムでは) スティッキービット (sticky bit)、実行グループ ID 設定 (set-group-id) および実行ユーザ ID 設定 (set-user-id) ビット --- を返します。

stat.S_IFMT(*mode*)

ファイルの形式を記述しているファイルモードの一部 (上記の S_IS*() 関数で使われます) を返します。

通常、ファイルの形式を調べる場合には os.path.is*() 関数を使うことになります; ここで挙げた関数は同じファイルに対して複数のテストを同時に行いたいが、stat() システムコールを何度も呼び出してオーバヘッドが生じるのを避けたい場合に便利です。これらはまた、ブロック型およびキャラクタ型デバイスに対するテストのように、*os.path* で扱うことのできないファイルの情報を調べる際にも便利です。

以下はプログラム例です:

```
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted at top,
       calling the callback function for each regular file'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.stat(pathname).st_mode
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print('Skipping %s' % pathname)

def visitfile(file):
    print('visiting', file)

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

ファイルのモードを人間が可読な文字列に変換するために、追加のユーティリティ関数が提供されています。

stat.filemode(*mode*)

ファイルのモードを '-rwxrwxrwx' 形式の文字列に変換します。

バージョン 3.3 で追加.

バージョン 3.4 で変更: この関数は、*S_IFDOOR* 、*S_IFPORT* 、*S_IFWHT* をサポートしています。

以下の全ての変数は、*os.stat()* 、*os.fstat()* 、または *os.lstat()* が返す 10 要素のタプルにおけるイ

ンデクスを単にシンボル定数化したものです。

stat.ST_MODE
　I ノードの保護モード。

stat.ST_INO
　I ノード番号。

stat.ST_DEV
　I ノードが存在するデバイス。

stat.ST_NLINK
　該当する I ノードへのリンク数。

stat.ST_UID
　ファイルの所持者のユーザ ID。

stat.ST_GID
　ファイルの所持者のグループ ID。

stat.ST_SIZE
　通常ファイルではバイトサイズ; いくつかの特殊ファイルでは処理待ちのデータ量。

stat.ST_ATIME
　最後にアクセスした時刻。

stat.ST_MTIME
　最後に変更された時刻。

stat.ST_CTIME
　オペレーティングシステムから返される”ctime”。ある OS(Unix など) では最後にメタデータが更新された時間となり、別の OS(Windows など) では作成時間となります (詳細については各プラットフォームのドキュメントを参照してください)。

” ファイルサイズ” の解釈はファイルの型によって異なります。通常のファイルの場合、サイズはファイルの大きさをバイトで表したものです。ほとんどの Unix 系 (特に Linux) における FIFO やソケットの場合、” サイズ” は *os.stat()* 、*os.fstat()* 、あるいは *os.lstat()* を呼び出した時点で読み出し待ちであったデータのバイト数になります; この値は時に有用で、特に上記の特殊なファイルを非ブロックモードで開いた後にポーリングを行いたいといった場合に便利です。他のキャラクタ型およびブロック型デバイスにおけるサイズフィールドの意味はさらに異なっていて、背後のシステムコールの実装によります。

以下の変数は、*ST_MODE* フィールドで使用されるフラグを定義しています。

最初に挙げる、以下のフラグを使うよりは、上記の関数を使うほうがポータブルです:

stat.S_IFSOCK
　ソケット。

stat.S_IFLNK
　シンボリックリンク。

stat.S_IFREG
通常のファイル。

stat.S_IFBLK
ブロックデバイス。

stat.S_IFDIR
ディレクトリ。

stat.S_IFCHR
キャラクターデバイス。

stat.S_IFIFO
FIFO。

stat.S_IFDOOR
ドア。

バージョン 3.4 で追加.

stat.S_IFPORT
イベントポート。

バージョン 3.4 で追加.

stat.S_IFWHT
ホワイトアウト。

バージョン 3.4 で追加.

注釈: *S_IFDOOR* 、*S_IFPORT* 、または *S_IFWHT* は、プラットフォームがこれらのファイルタイプをサポートしていない場合、0 として定義されます。

以下のフラグは、*os.chmod()* の *mode* 引数に使うこともできます:

stat.S_ISUID
UID ビットを設定する。

stat.S_ISGID
グループ ID ビットを設定する。このビットには幾つかの特殊ケースがあります。ディレクトリに対して設定されていた場合、BSD のセマンティクスが利用される事を示しています。すなわち、そこに作成されるファイルは、作成したプロセスの有効グループ ID (effective group ID) ではなくそのディレクトリのグループ ID を継承し、そこに作成されるディレクトリにも *S_ISGID* ビットが設定されます。グループ実行ビット (*S_IXGRP*) が設定されていないファイルに対してこのビットが設定されていた場合、強制ファイル/レコードロックを意味します (*S_ENFMT* も参照してください)。

stat.S_ISVTX
スティッキービット。このビットがディレクトリに対して設定されているとき、そのディレクトリ内の

ファイルは、そのファイルのオーナー、あるいはそのディレクトリのオーナーか特権プロセスのみが、リネームや削除をすることが出来ることを意味しています。

stat.S_IRWXU
ファイルオーナーの権限に対するマスク。

stat.S_IRUSR
オーナーがリード権限を持っている。

stat.S_IWUSR
オーナーがライト権限を持っている。

stat.S_IXUSR
オーナーが実行権限を持っている。

stat.S_IRWXG
グループの権限に対するマスク。

stat.S_IRGRP
グループがリード権限を持っている。

stat.S_IWGRP
グループがライト権限を持っている。

stat.S_IXGRP
グループが実行権限を持っている。

stat.S_IRWXO
その他 (グループ外) の権限に対するマスク。

stat.S_IROTH
その他はリード権限を持っている。

stat.S_IWOTH
その他はライト権限を持っている。

stat.S_IXOTH
その他は実行権限を持っている。

stat.S_ENFMT
System V ファイルロック強制。このフラグは *S_ISGID* と共有されています。グループ実行ビット (*S_IXGRP*) が設定されていないファイルでは、ファイル/レコードのロックが強制されます。

stat.S_IREAD
S_IRUSR の、Unix V7 のシノニム。

stat.S_IWRITE
S_IWUSR の、Unix V7 のシノニム。

stat.S_IEXEC
S_IXUSR の、Unix V7 のシノニム。

以下のフラグを *os.chflags()* の *flags* 引数として利用できます:

stat.UF_NODUMP
ファイルをダンプしない。

stat.UF_IMMUTABLE
ファイルは変更されない。

stat.UF_APPEND
ファイルは追記しかされない。

stat.UF_OPAQUE
ユニオンファイルシステムのスタックを通したとき、このディレクトリは不透明です。

stat.UF_NOUNLINK
ファイルはリネームや削除されない。

stat.UF_COMPRESSED
ファイルは圧縮して保存される (Mac OS X 10.6+)。

stat.UF_HIDDEN
ファイルは GUI で表示されるべきでない (Mac OS X 10.5+)。

stat.SF_ARCHIVED
ファイルはアーカイブされているかもしれません。

stat.SF_IMMUTABLE
ファイルは変更されない。

stat.SF_APPEND
ファイルは追記しかされない。

stat.SF_NOUNLINK
ファイルはリネームや削除されない。

stat.SF_SNAPSHOT
このファイルはスナップショットファイルです。

詳しい情報は *BSD か Mac OS システムの man page *chflags(2)* を参照してください。

Windows では、*os.stat()* が返す st_file_attributes メンバー内のビットを検証する際に、以下のファイル属性定数を使用できます。これらの定数の意味について詳しくは、Windows API documentation を参照してください。

stat.FILE_ATTRIBUTE_ARCHIVE
stat.FILE_ATTRIBUTE_COMPRESSED
stat.FILE_ATTRIBUTE_DEVICE
stat.FILE_ATTRIBUTE_DIRECTORY
stat.FILE_ATTRIBUTE_ENCRYPTED
stat.FILE_ATTRIBUTE_HIDDEN
stat.FILE_ATTRIBUTE_INTEGRITY_STREAM

stat.FILE_ATTRIBUTE_NORMAL
stat.FILE_ATTRIBUTE_NOT_CONTENT_INDEXED
stat.FILE_ATTRIBUTE_NO_SCRUB_DATA
stat.FILE_ATTRIBUTE_OFFLINE
stat.FILE_ATTRIBUTE_READONLY
stat.FILE_ATTRIBUTE_REPARSE_POINT
stat.FILE_ATTRIBUTE_SPARSE_FILE
stat.FILE_ATTRIBUTE_SYSTEM
stat.FILE_ATTRIBUTE_TEMPORARY
stat.FILE_ATTRIBUTE_VIRTUAL
バージョン 3.5 で追加.

Windows では、*os.lstat()* が返す st_reparse_tag メンバーとの比較に次の定数が 使えます。これらはよく知られている定数ですが、全てを網羅したリストではありません。

stat.IO_REPARSE_TAG_SYMLINK
stat.IO_REPARSE_TAG_MOUNT_POINT
stat.IO_REPARSE_TAG_APPEXECLINK
バージョン 3.8 で追加.

11.5 filecmp --- ファイルおよびディレクトリの比較

ソースコード: Lib/filecmp.py

---

filecmp モジュールでは、ファイルおよびディレクトリを比較するため、様々な時間／正確性のトレードオフに関するオプションを備えた関数を定義しています。ファイルの比較については、*difflib* モジュールも参照してください。

filecmp モジュールでは以下の関数を定義しています:

filecmp.cmp(*f1*, *f2*, *shallow=True*)
名前が *f1* および *f2* のファイルを比較し、二つのファイルが同じらしければ True を返し、そうでなければ False を返します。

shallow が真の場合、同一の *os.stat()* シグニチャを持つファイルは等しいとみなされます。そうでなければ、ファイルの内容が比較されます。

可搬性と効率のために、この関数は外部プログラムを一切呼び出さないので注意してください。

この関数は過去の比較と結果のキャッシュを使用します。ファイルの *os.stat()* 情報が変更された場合、キャッシュの項目は無効化されます。*clear_cache()* を使用して全キャッシュを削除することが出来ます。

filecmp.cmpfiles(*dir1*, *dir2*, *common*, *shallow=True*)
dir1 と *dir2* ディレクトリの中の、*common* で指定されたファイルを比較します。

ファイル名からなる 3 つのリスト: *match*, *mismatch*, *errors* を返します。*match* には双方のディレクトリで一致したファイルのリストが含まれ、*mismatch* にはそうでないファイル名のリストが入ります。そして *errors* は比較されなかったファイルが列挙されます。*errors* になるのは、片方あるいは両方のディレクトリに存在しなかった、ユーザーにそのファイルを読む権限がなかった、その他何らかの理由で比較を完了することができなかった場合です。

引数 *shallow* はその意味も標準の設定も *filecmp.cmp()* と同じです。

例えば、`cmpfiles('a', 'b', ['c', 'd/e'])` は `a/c` を `b/c` と、`a/d/e` を `b/d/e` と、それぞれ比較します。`'c'` と `'d/e'` はそれぞれ、返される 3 つのリストのいずれかに登録されます。

filecmp.clear_cache()

filecmp のキャッシュをクリアします。背後のファイルシステムの mtime 分解能未満でのファイル変更後にすぐに比較するような場合に有用です。

バージョン 3.4 で追加.

11.5.1 dircmp クラス

class filecmp.dircmp(*a*, *b*, *ignore=None*, *hide=None*)

ディレクトリ *a* および *b* を比較するための新しいディレクトリ比較オブジェクトを生成します。*ignore* は比較の際に無視するファイル名のリストで、標準の設定では *filecmp.DEFAULT_IGNORES* です。*hide* は表示しない名前のリストで、標準の設定では `[os.curdir, os.pardir]` です。

dircmp クラスは、*filecmp.cmp()* で説明されているような **浅い** 比較を行うことによりファイルを比較します。

dircmp クラスは以下のメソッドを提供しています:

report()

a と *b* の比較を (*sys.stdout* に) 表示します。

report_partial_closure()

a および *b* およびそれらの直下にある共通のサブディレクトリ間での比較結果を出力します。

report_full_closure()

a および *b* およびそれらの共通のサブディレクトリ間での比較結果を (再帰的に比較して) 出力します。

dircmp クラスは、比較されているディレクトリ階層に関する様々な情報のビットを得るために使用することのできる、興味深い属性を数多く提供しています。

すべての属性は `__getattr__()` フックによって遅延評価されるので、計算が軽い属性のみを使用した場合は、属性の計算による速度の低下は起こりません。

left

ディレクトリ *a* です。

right
ディレクトリ *b* です。

left_list
a にあるファイルおよびサブディレクトリです。*hide* および *ignore* でフィルタされています。

right_list
b にあるファイルおよびサブディレクトリです。*hide* および *ignore* でフィルタされています。

common
a および *b* の両方にあるファイルおよびサブディレクトリです。

left_only
a だけにあるファイルおよびサブディレクトリです。

right_only
b だけにあるファイルおよびサブディレクトリです。

common_dirs
a および *b* の両方にあるサブディレクトリです。

common_files
a および *b* の両方にあるファイルです。

common_funny
a および *b* の両方にあり、ディレクトリ間でタイプが異なるか、*os.stat()* がエラーを報告するような名前です。

same_files
クラスのファイル比較オペレータを用いて *a* と *b* の両方において同一のファイルです。

diff_files
a と *b* の両方に存在し、クラスのファイル比較オペレータに基づいて内容が異なるファイルです。

funny_files
a および *b* 両方にあるが、比較されなかったファイルです。

subdirs
common_dirs のファイル名を *dircmp* オブジェクトに対応付けた辞書です。

filecmp.**DEFAULT_IGNORES**
バージョン 3.4 で追加.

デフォルトで *dircmp* に無視されるディレクトリのリストです。

これは subdirs 属性を使用して 2 つのディレクトリを再帰的に探索して、共通の異なるファイルを示すための単純化された例です:

```
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...     for name in dcmp.diff_files:
```

(次のページに続く)

(前のページからの続き)

```
...         print("diff_file %s found in %s and %s" % (name, dcmp.left,
...               dcmp.right))
...     for sub_dcmp in dcmp.subdirs.values():
...         print_diff_files(sub_dcmp)
...
>>> dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)
```

11.6 tempfile --- 一時ファイルやディレクトリの作成

ソースコード: Lib/tempfile.py

このモジュールは一時ファイルやディレクトリを作成します。サポートされている全てのプラットフォームで動作します。*TemporaryFile*、*NamedTemporaryFile*、*TemporaryDirectory*、*SpooledTemporaryFile* は自動的に後始末をし、コンテキストマネージャとして使うことの出来る高水準のインターフェイスです。*mkstemp()* と *mkdtemp()* は手動で後始末をしなければならない低水準の関数です。

ユーザが呼び出し可能な全ての関数とコンストラクタは追加の引数を受け取ります。その引数によって一時ファイルやディレクトリの場所と名前を直接操作することが出来ます。このモジュールに使用されるファイル名はランダムな文字を含みます。そのためファイルは共有された一時ディレクトリに安全に作成されます。後方互換性を保つために引数の順序は若干奇妙です。分かりやすさのためにキーワード引数を使用してください。

このモジュールではユーザが呼び出し可能な以下の項目を定義しています:

tempfile.TemporaryFile(*mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, ***, *errors=None*)

一時的な記憶領域として使うことの出来る *file-like object* を返します。ファイルは *mkstemp()* と同じルールにより安全に作成されます。オブジェクトは閉じられる (オブジェクトのガベージコレクションによる暗黙的なものも含みます) とすぐに破壊されます。Unix では、そのファイルのディレクトリエントリは全く作成されないか、ファイル作成後すぐに削除されます。これは他のプラットフォームではサポートされません。よって、この関数で作成された一時ファイルがファイルシステムで可視な名前を持つかどうかをコードで当てにすべきではありません。

返されたオブジェクトをコンテキストマネージャとして使うことが出来ます (**使用例** を参照してください)。コンテキストの完了やファイルオブジェクトの破壊で、一時ファイルはファイルシステムから削除されます。

作成されたファイルを閉じることなく読み書きできるように、*mode* 引数のデフォルトは `'w+b'` です。保存されるデータに関わらず全てのプラットフォーム上で一貫して動作するようにバイナリモードが使用されます。*buffering*、*encoding*、*errors*、*newline* は、*open()* に対する引数として解釈されます。

dir、*prefix*、*suffix* 引数の意味とデフォルトは *mkstemp()* のものと同じです。

返されたオブジェクトは、POSIX プラットフォームでは本物のファイルオブジェクトです。それ以外のプラットフォームでは、`file` 属性が下層の本物のファイルであるファイル様オブジェクトです。

os.O_TMPFILE フラグは、利用可能で動作する場合に用いられます (Linux 固有で、Linux kernel 3.11 以降)。

引数 `fullpath` を指定して **監査イベント** `tempfile.mkstemp` を送出します。

バージョン 3.5 で変更: 利用可能であれば *os.O_TMPFILE* フラグが使用されます。

バージョン 3.8 で変更: パラメーターに *errors* を追加しました。

tempfile.NamedTemporaryFile(*mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, *delete=True*, ***, *errors=None*)

この関数は、ファイルシステム上でファイルが可視の名前を持つことが保証される (Unix においてはディレクトリエントリが unlink されない) 点以外は *TemporaryFile()* と正確に同じことを行います。その名前は、返されたファイル様オブジェクトの `name` 属性から取得することができます。名前付き一時ファイルがまだ開かれている間にこの名前を使って再度ファイルを開くことができるかどうかは、プラットフォームによって異なります (Unix 上では可能ですが、Windows NT 以降ではできません)。*delete* が真の場合 (デフォルト)、ファイルは閉じられたら即座に削除されます。返されたオブジェクトは常にファイル様オブジェクトで、その `file` 属性は元になった本物のファイルオブジェクトです。このファイルライクオブジェクトは、通常のファイルのように `with` 文の中で使用することができます。

引数 `fullpath` を指定して **監査イベント** `tempfile.mkstemp` を送出します。

バージョン 3.8 で変更: パラメーターに *errors* を追加しました。

tempfile.SpooledTemporaryFile(*max_size=0*, *mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, ***, *errors=None*)

この関数はファイルサイズが *max_size* を超えるかファイルの `fileno()` メソッドが呼ばれるまで、データがメモリにスプールされる点以外は *TemporaryFile()* と正確に同じことを行います。上記条件を満たすと内容はディスクに書き込まれ、操作は *TemporaryFile()* と同様に進みます。

この関数が返すファイルは、追加で 1 つのメソッド `rollover()` を持っています。このメソッドが呼ばれると、(サイズに関係なく) メモリからディスクへのロールオーバーが実行されます。

返されたオブジェクトはファイル様オブジェクトで、その `_file` 属性は (バイナリかテキスト *mode* が指定されたかどうかに依存して) *io.BytesIO* か *io.TextIOWrapper* オブジェクト、あるいは `rollover()` が呼ばれたかどうかに依存して本物のファイルオブジェクトになります。このファイル様オブジェクトは、通常のファイルオブジェクトと同じように `with` 文中で使用することが出来ます。

バージョン 3.3 で変更: truncate メソッドが `size` 引数を受け取るようになりました。

バージョン 3.8 で変更: パラメーターに *errors* を追加しました。

tempfile.TemporaryDirectory(*suffix=None*, *prefix=None*, *dir=None*)

この関数は *mkdtemp()* と同じルールを使用して安全に一時ディレクトリを作成します。返されたオブジェクトは、コンテキストマネージャとして使用することができます (context-managers を参照)。コンテキストの完了や一時ディレクトリの破壊で新規作成された一時ディレクトリとその中身はファイルシステムから削除されます。

ディレクトリ名は返されたオブジェクトの `name` 属性から取得することが出来ます。返されたオブジェ

クトがコンテクスト管理者として使用された場合、1 つなら name は with 文内の as のターゲットに割り当てられます。

cleanup() メソッドを呼んでディレクトリを明示的に片付けることができます。

引数 fullpath を指定して 監査イベント tempfile.mkdtemp を送出します。

バージョン 3.2 で追加.

tempfile.mkstemp(*suffix=None*, *prefix=None*, *dir=None*, *text=False*)

可能な限り最も安全な手段で一時ファイルを生成します。プラットフォームが *os.open()* の *os.O_EXCL* フラグを正しく実装している限り、ファイルの作成で競合が起こることはありません。作成したユーザのユーザ ID からのみファイルを読み書き出来ます。プラットフォームがファイルが実行可能かどうかを示す許可ビットを使用している場合、ファイルは誰からも実行不可です。このファイルのファイル記述子は子プロセスに継承されません。

TemporaryFile() と違って、*mkstemp()* のユーザは用済みになった時に一時ファイルを削除しなければなりません。

suffix が None でない場合、ファイル名はその接尾辞で終わります。そうでない場合、接尾辞はありません。*mkstemp()* はファイル名と接尾辞の間にドットを追加しません。必要であれば *suffix* の先頭につけてください。

prefix が None でない場合、ファイル名はその接頭辞で始まります。そうでない場合、デフォルトの接頭辞が使われます。必要に応じ、デフォルトは *gettempprefix()* または *gettempprefixb()* の返り値です。

dir が None でない場合、ファイルはそのディレクトリ下に作成されます。None の場合、デフォルトのディレクトリが使われます。デフォルトのディレクトリはプラットフォームに依存するリストから選ばれますが、アプリケーションのユーザは *TMPDIR*、*TEMP*、または *TMP* 環境変数を設定することでディレクトリの場所を管理することができます。そのため、生成されるファイル名が、os.popen() で外部コマンドにクォーティング無しで渡すことができるなどといった、扱いやすい性質を持つ保証はありません。

suffix、*prefix*、*dir* のいずれかが None でない場合、それらは同じ型でなければなりません。bytes の場合、返された名前は str でなはく bytes です。他の挙動はデフォルトで返り値を bytes に強制的にしたい場合は suffix=b'' を渡してください。

text に真を指定した場合は、ファイルはテキストモードで開かれます。そうでない場合 (デフォルト) は、ファイルはバイナリモードで開かれます。

mkstemp() は開かれたファイルを扱うための OS レベルのハンドル (*os.open()* が返すものと同じ) とファイルの絶対パス名が順番に並んだタプルを返します。

引数 fullpath を指定して 監査イベント tempfile.mkstemp を送出します。

バージョン 3.5 で変更: *suffix*、*prefix*、*dir* は bytes の返り値を得るために bytes で渡すことが出来ます。それ以前は str のみ許されていました。適切なデフォルト値を使用するよう、*suffix* と *prefix* は None を受け入れ、デフォルトにするようになりました。

バージョン 3.6 で変更: *dir* パラメタが *path-like object* を受け付けるようになりました。

tempfile.mkdtemp(*suffix=None*, *prefix=None*, *dir=None*)

可能な限り安全な方法で一時ディレクトリを作成します。ディレクトリの生成で競合は発生しません。ディレクトリを作成したユーザ ID だけが、このディレクトリに対して内容を読み出したり、書き込んだり、検索したりすることができます。

mkdtemp() のユーザは用済みになった時に一時ディレクトリとその中身を削除しなければなりません。

prefix, *suffix*, *dir* 引数は *mkstemp()* 関数のものと同じです。

mkdtemp() は新たに生成されたディレクトリの絶対パス名を返します。

引数 `fullpath` を指定して **監査イベント** `tempfile.mkdtemp` を送出します。

バージョン 3.5 で変更: *suffix*、*prefix*、*dir* は bytes の返り値を得るために bytes で渡すことが出来ます。それ以前は str のみ許されていました。適切なデフォルト値を使用するよう、*suffix* と *prefix* は `None` を受け入れ、デフォルトにするようになりました。

バージョン 3.6 で変更: *dir* パラメタが *path-like object* を受け付けるようになりました。

tempfile.gettempdir()

一時ファイルに用いられるディレクトリの名前を返します。これはモジュール内の全ての関数の *dir* 引数のデフォルト値を定義します。

Python は呼び出したユーザがファイルを作ることの出来るディレクトリを検索するのに標準的なリストを使用します。そのリストは:

1. 環境変数 `TMPDIR` で与えられているディレクトリ名。
2. 環境変数 `TEMP` で与えられているディレクトリ名。
3. 環境変数 `TMP` で与えられているディレクトリ名。
4. プラットフォーム依存の場所:
   - Windows ではディレクトリ `C:\TEMP` 、`C:\TMP` 、`\TEMP` 、および `\TMP` の順。
   - その他の全てのプラットフォームでは、`/tmp` 、`/var/tmp` 、および `/usr/tmp` の順。
5. 最後の手段として、現在の作業ディレクトリ。

この検索の結果はキャッシュされます。以下の *tempdir* の記述を参照してください。

tempfile.gettempdirb()

gettempdir() と同じですが返り値は bytes です。

バージョン 3.5 で追加.

tempfile.gettempprefix()

一時ファイルを生成する際に使われるファイル名の接頭辞を返します。これにはディレクトリ部は含まれません。

tempfile.gettempprefixb()
: *gettempprefix()* と同じですが返り値は bytes です。

  バージョン 3.5 で追加.

モジュールはグローバル変数を使用して、*gettempdir()* が返す、一時ファイルに用いられるディレクトリ名を記憶します。直接設定して選考過程を上書き出来ますが、推奨されません。このモジュールの全ての関数はディレクトリを指定する *dir* 引数を受け取ります。この方法が推奨されます。

tempfile.tempdir
: None 以外の値に設定された場合、このモジュールで定義されている全ての関数の *dir* 引数のデフォルト値として定義されます。

  tempdir が (デフォルトの) None の場合、*gettempprefix()* を除く上記のいずれかの関数を呼び出す際は常に *gettempdir()* で述べられているアルゴリズムによって初期化されます。

11.6.1 使用例

tempfile モジュールの典型的な使用法のいくつかの例を挙げます:

```
>>> import tempfile

# create a temporary file and write some data to it
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# read data from file
>>> fp.seek(0)
>>> fp.read()
b'Hello world!'
# close the file, it will be removed
>>> fp.close()

# create a temporary file using a context manager
>>> with tempfile.TemporaryFile() as fp:
...     fp.write(b'Hello world!')
...     fp.seek(0)
...     fp.read()
b'Hello world!'
>>>
# file is now closed and removed

# create a temporary directory using the context manager
>>> with tempfile.TemporaryDirectory() as tmpdirname:
...     print('created temporary directory', tmpdirname)
>>>
# directory and contents have been removed
```

11.6.2 非推奨の関数と変数

一時ファイルを作成する歴史的な手法は、まず *mktemp()* 関数でファイル名を作り、その名前を使ってファイルを作成するというものでした。残念ながらこの方法は安全ではありません。なぜなら、*mktemp()* の呼び出しと最初のプロセスが続いてファイル作成を試みる間に、異なるプロセスがその名前でファイルを同時に作成するかもしれないからです。解決策は二つのステップを同時に行い、ファイルをすぐに作成するというものです。この方法は *mkstemp()* や上述している他の関数で使用されています。

tempfile.**mktemp**(*suffix=''*, *prefix='tmp'*, *dir=None*)

バージョン 2.3 で非推奨: 代わりに *mkstemp()* を使って下さい。

呼び出し時には存在しなかった、ファイルの絶対パス名を返します。*prefix*、*suffix*、*dir* 引数は *mkstemp()* のものと似ていますが、bytes のファイル名、`suffix=None`、そして `prefix=None` がサポートされていない点で異なります。

警告: この関数を使うとプログラムのセキュリティホールになる可能性があります。この関数がファイル名を返した後、あなたがそのファイル名を使って次に何かをしようとする段階に至る前に、誰か他の人間があなたを出し抜くことができてしまいます。*mktemp()* の利用は、*NamedTemporaryFile()* に `delete=False` 引数を渡すことで、簡単に置き換えることができます:

```
>>> f = NamedTemporaryFile(delete=False)
>>> f.name
'/tmp/tmptjujjt'
>>> f.write(b"Hello World!\n")
13
>>> f.close()
>>> os.unlink(f.name)
>>> os.path.exists(f.name)
False
```

11.7 glob --- Unix 形式のパス名のパターン展開

ソースコード: Lib/glob.py

---

glob モジュールは Unix シェルで使われているルールに従い指定されたパターンに一致するすべてのパス名を見つけ出します。返される結果の順序は不定です。チルダ展開は行われませんが、`*`, `?`, および `[]` で表現される文字範囲については正しくマッチされます。これは、関数 *os.scandir()* および *fnmatch.fnmatch()* を使用して行われており、実際にサブシェルを呼び出しているわけではありません。*fnmatch.fnmatch()* と異なり、*glob* はドット (`.`) で始まるファイル名は特別扱いする点に注意してください。(チルダおよびシェル変数の展開を利用したい場合は *os.path.expanduser()* および *os.path.expandvars()* を使用してください。)

リテラルにマッチさせるには、メタ文字を括弧に入れてください。例えば、`'[?]'` は文字 `'?'` にマッチします。

参考:

pathlib モジュールは高水準のパスオブジェクトを提供します。

glob.**glob**(*pathname*, ***, *recursive=False*)

pathname (パスの指定を含んだ文字列でなければいけません) にマッチする、空の可能性のあるパス名のリストを返します。*pathname* は (`/usr/src/Python-1.5/Makefile` のように) 絶対パスでも、(`../../Tools/*/*.gif` のように) 相対パスでもよく、シェル形式のワイルドカードを含んでいてもかまいません。結果には (シェルと同じく) 壊れたシンボリックリンクも含まれます。結果がソートされるかどうかは、ファイルシステムによって異なります。この関数の呼び出し中に条件を満たすファイルが移動や追加された場合、そのファイルのパス名を含むかどうかは指定されていません。

recursive が真の場合、パターン "`**`" はあらゆるファイルや 0 個以上のディレクトリ、サブディレクトリおよびディレクトリへのシンボリックリンクにマッチします。パターンの末尾が *os.sep* または *os.altsep* の場合、ファイルは一致しません。

引数 `pathname`, `recursive` を指定して **監査イベント** `glob.glob` を送出します。

注釈: パターン "`**`" を大きなディレクトリツリーで使用するととてつもなく時間がかかるかもしれません。

バージョン 3.5 で変更: "`**`" を使った再帰的な glob がサポートされました。

glob.**iglob**(*pathname*, ***, *recursive=False*)

実際には一度にすべてを格納せずに、*glob()* と同じ値を順に生成する **イテレーター** を返します。

引数 `pathname`, `recursive` を指定して **監査イベント** `glob.glob` を送出します。

glob.**escape**(*pathname*)

すべての特殊文字 (`'?'` 、`'*'` 、`'['`) をエスケープします。特殊文字を含んでいる可能性のある任意のリテラル文字列をマッチさせたいときに便利です。drive/UNC sharepoints の特殊文字はエスケープされません。たとえば Windows では `escape('//?/c:/Quo vadis?.txt')` は `'//?/c:/Quo vadis[?].txt'` を返します。

バージョン 3.4 で追加.

たとえば、次の 3 個のファイル `1.gif`, `2.txt`, `card.gif` と、ファイル `3.txt` だけを含んだサブディレクトリ `sub` があった場合、*glob()* は以下の結果を返します。パスに接頭する要素がどう維持されるかに注意してください。:

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
>>> glob.glob('**/*.txt', recursive=True)
```

(次のページに続く)

(前のページからの続き)

```
['2.txt', 'sub/3.txt']
>>> glob.glob('./**/', recursive=True)
['./', './sub/']
```

ディレクトリが . で始まるファイルを含んでいる場合、デフォルトでそれらはマッチしません。例えば、card.gif と .card.gif を含むディレクトリを考えてください:

```
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

参考:

fnmatch **モジュール**　シェル形式の (パスではない) ファイル名展開

11.8 fnmatch --- Unix ファイル名のパターンマッチ

ソースコード: Lib/fnmatch.py

このモジュールは Unix のシェル形式のワイルドカードに対応しています。これらは、(*re* モジュールでドキュメント化されている) 正規表現とは **異なります** 。シェル形式のワイルドカードで使われる特殊文字は、次のとおりです。

Pattern	**意味**
*	すべてにマッチします
?	任意の一文字にマッチします
[seq]	*seq* にある任意の文字にマッチします
[!seq]	*seq* にない任意の文字にマッチします

リテラルにマッチさせるには、メタ文字を括弧に入れてください。例えば、'[?]' は文字 '?' にマッチします。

ファイル名の区切り文字 (Unix では '/') はこのモジュールに固有なものでは **ない** ことに注意してください。パス名展開については、*glob* モジュールを参照してください (*glob* はパス名の部分にマッチさせるのに *filter()* を使っています)。同様に、ピリオドで始まるファイル名はこのモジュールに固有ではなくて、* と ? のパターンでマッチします。

fnmatch.**fnmatch**(*filename*, *pattern*)

filename の文字列が *pattern* の文字列にマッチするかテストして、*True* 、*False* のいずれかを返します。どちらの引数とも *os.path.normcase()* を使って、大文字、小文字が正規化されます。オペレーティングシステムが標準でどうなっているかに関係なく、大文字、小文字を区別して比較する場合には、*fnmatchcase()* が使えます。

次の例では、カレントディレクトリにある、拡張子が .txt である全てのファイルを表示しています:

```
import fnmatch
import os

for file in os.listdir('.'):
    if fnmatch.fnmatch(file, '*.txt'):
        print(file)
```

fnmatch.**fnmatchcase**(*filename*, *pattern*)

filename が *pattern* にマッチするかテストして、*True* 、*False* を返します。比較は大文字、小文字を区別し、*os.path.normcase()* は適用しません。

fnmatch.**filter**(*names*, *pattern*)

pattern にマッチするイテラブルの *names* を要素とするリストを構築します。`[n for n in names if fnmatch(n, pattern)]` と同じですが、もっと効率よく実装されています。

fnmatch.**translate**(*pattern*)

シェルスタイルの *pattern* を、*re.match()* で使用するための正規表現に変換して返します。

例:

```
>>> import fnmatch, re
>>>
>>> regex = fnmatch.translate('*.txt')
>>> regex
'(?s:.*\\.txt)\\Z'
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<re.Match object; span=(0, 10), match='foobar.txt'>
```

参考:

glob **モジュール** Unix シェル形式のパス展開。

11.9 linecache --- テキストラインにランダムアクセスする

ソースコード: Lib/linecache.py

---

linecache モジュールは、キャッシュ (一つのファイルから何行も読んでおくのが一般的です) を使って、内部で最適化を図りつつ、Python ソースファイルの任意の行を取得するのを可能にします。*traceback* モジュールは、整形されたトレースバックにソースコードを含めるためにこのモジュールを利用しています。

tokenize.open() 関数は、ファイルを開くために使用されます。この関数は、*tokenize.detect_encoding()* を使用してファイルのエンコーディングを取得します。エンコーディングトークンが存在しない場合、デフォルトの UTF-8 になります。

linecache モジュールでは次の関数が定義されています:

linecache.**getline**(*filename*, *lineno*, *module_globals=None*)

filename という名前のファイルから *lineno* 行目を取得します。この関数は決して例外を発生させません --- エラーの際には `''` を返します (行末の改行文字は、見つかった行に含まれます)。

filename という名前のファイルが見付からなかった場合、この関数は最初に *module_globals* にある **PEP 302** `__loader__` を確認します。ローダーが存在していて、`get_source` メソッドが実装されていた場合、ソースコードの行を決定します (`get_source()` が `None` を返した場合は、`''` が返ります)。最後に、*filename* が相対ファイル名だった場合、モジュール検索パス `sys.path` のエントリからの相対パスを探します。

linecache.**clearcache**()

キャッシュをクリアします。それまでに *getline()* を使って読み込んだファイルの行が必要でなくなったら、この関数を使ってください。

linecache.**checkcache**(*filename=None*)

キャッシュが有効かどうかを確認します。キャッシュしたファイルがディスク上で変更された可能性があり、更新後のバージョンが必要な場合にこの関数を使用します。*filename* が与えられない場合、全てのキャッシュエントリを確認します。

linecache.**lazycache**(*filename*, *module_globals*)

後々の呼び出しで *module_globals* が `None` となっていても、ファイルの形式でないモジュールの行を後から *getline()* で取得するのに十分な詳細を把握しておきます。この関数により、モジュールの globals を無限に持ち運ぶ必要無しに、実際に必要な行まで

バージョン 3.5 で追加.

以下はプログラム例です:

```
>>> import linecache
>>> linecache.getline(linecache.__file__, 8)
'import sys\n'
```

11.10 shutil --- 高水準のファイル操作

ソースコード: Lib/shutil.py

---

shutil モジュールはファイルやファイルの集まりに対する高水準の操作方法を多数提供します。特にファイルのコピーや削除のための関数が用意されています。個別のファイルに対する操作については、*os* モジュールも参照してください。

警告: 高水準のファイルコピー関数 (*shutil.copy()*, *shutil.copy2()*) でも、ファイルのメタデータの全てをコピーすることはできません。

POSIX プラットフォームでは、これは ACL やファイルのオーナー、グループが失われることを意味しています。Mac OS では、リソースフォーク (resource fork) やその他のメタデータが利用されません。こ

れは、リソースが失われ、ファイルタイプや生成者コード (creator code) が正しくなくなることを意味しています。Windows では、ファイルオーナー、ACL、代替データストリームがコピーされません。

11.10.1 ディレクトリとファイルの操作

shutil.copyfileobj(*fsrc*, *fdst*[, *length*])

ファイル形式のオブジェクト *fsrc* の内容を *fdst* へコピーします。整数値 *length* は与えられた場合バッファサイズを表します。特に *length* が負の場合、チャンク内のソースデータを繰り返し操作することなくデータをコピーします。デフォルトでは、制御不能なメモリ消費を避けるためにデータはチャンク内に読み込まれます。*fsrc* オブジェクトの現在のファイル位置が 0 でない場合、現在の位置からファイル終端までの内容のみがコピーされることに注意してください。

shutil.copyfile(*src*, *dst*, ***, *follow_symlinks=True*)

src という名前のファイルの内容 (メタデータを含まない) を *dst* という名前のファイルにコピーし、最も効率的な方法で *dst* を返します。*src* と *dst* は path-like object または文字列でパス名を指定します。

dst は完全な対象ファイル名でなければなりません。対象としてディレクトリ名を指定したい場合は *shutil.copy()* を参照してください。*src* と *dst* が同じファイルだった場合、*SameFileError* を送出します。

dst は書き込み可能でなければなりません。そうでない場合、*OSError* 例外を送出します。*dst* がすでに存在する場合、そのファイルは置き換えられます。キャラクタデバイスやブロックデバイスなどの特殊なファイルとパイプをこの関数でコピーすることはできません。

follow_symlinks が偽で *src* がシンボリックリンクの場合、*src* のリンク先をコピーする代わりに新しいシンボリックリンクを作成します。

引数 src 、dst` を指定して :ref:`監査イベント <auditing>` ``shutil.copyfile を送出します。

バージョン 3.3 で変更: 以前は *OSError* の代わりに *IOError* が送出されていました。*follow_symlinks* 引数が追加されました。*dst* を返すようになりました。

バージョン 3.4 で変更: *Error* の代わりに *SameFileError* を送出します。後者は前者のサブクラスなのでこの変更は後方互換です。

バージョン 3.8 で変更: ファイルのコピーをより効率的に行うため、プラットフォーム特有の高速なコピーを行うシステムコールが利用されることがあります。**プラットフォーム依存の効率的なコピー操作**を参照してください。

exception shutil.SameFileError

copyfile() のコピー元と先が同じファイルの場合送出されます。

バージョン 3.4 で追加.

shutil.copymode(*src*, *dst*, ***, *follow_symlinks=True*)

パーミッションを *src* から *dst* にコピーします。ファイルの内容、オーナー、グループには影響しません。*src* と *dst* は path-like object または文字列でファイルのパス名を指定します。*follow_symlinks* が偽で、*src* および *dst* がシンボリックリンクの場合、*copymode()* は (リンク先ではなく) *dst* 自体のモードを変更します。この機能は全てのプラットフォームで使えるわけではありません。詳しくは *copystat()* を参照してください。シンボリックリンクの変更をしようとした時、*copymode()* がローカルのプラットフォームでシンボリックリンクを変更できない場合は何もしません。

引数 src 、dst` を指定して :ref:`監査イベント <auditing>` ``shutil.copymode を送出します。

バージョン 3.3 で変更: *follow_symlinks* 引数が追加されました。

shutil.copystat(*src*, *dst*, ***, *follow_symlinks=True*)

パーミッション、最終アクセス時間、最終変更時間、その他のフラグを *src* から *dst* にコピーします。Linux では、*copystat()* は可能なら ” 拡張属性” もコピーします。ファイルの内容、オーナー、グループには影響しません。*src* と *dst* は path-like object または文字列でファイルのパス名を指定します。

follow_symlinks が偽の場合、*src* と *dst* の両方がシンボリックリンクであれば、*copystat()* はリンク先ではなくてシンボリックリンク自体を操作します。*src* からシンボリックリンクの情報を読み込み、*dst* のシンボリックリンクにその情報を書き込みます。

注釈: すべてのプラットフォームでシンボリックリンクの検査と変更ができるわけではありません。Python はその機能が利用かどうかを調べる方法を用意しています。

- os.chmod in os.supports_follow_symlinks が True の場合 *copystat()* はシンボリックリンクのパーミッションを変更できます。
- os.utime in os.supports_follow_symlinks が True の場合 *copystat()* はシンボリックリンクの最終アクセス時間と最終変更時間を変更できます。
- os.chflags in os.supports_follow_symlinks が True の場合 *copystat()* はシンボリックリンクのフラグを変更できます。(os.chflags がないプラットフォームもあります。)

機能の幾つか、もしくは全てが利用できないプラットフォームでシンボリックリンクを変更しようとした場合、*copystat()* は可能な限り全てをコピーします。*copystat()* が失敗を返すことはありません。

より詳しい情報は *os.supports_follow_symlinks* を参照して下さい。

引数 src 、dst` を指定して :ref:`監査イベント <auditing>` ``shutil.copystat を送出します。

バージョン 3.3 で変更: *follow_symlinks* 引数と Linux の拡張属性がサポートされました。

shutil.copy(*src*, *dst*, ***, *follow_symlinks=True*)

ファイル *src* をファイルまたはディレクトリ *dst* にコピーします。*src* と *dst* は両方共 term:*path-like object* <*path-like object*> または文字列でなければなりません。*dst* がディレクトリを指定している場

合、ファイルは *dst* の中に、*src* のベースファイル名を使ってコピーされます。新しく作成したファイルのパスを返します。

follow_symlinks が偽で、*src* がシンボリックリンクの場合、*dst* はシンボリックリンクとして作成されます。*follow_symlinks* が真で *src* がシンボリックリンクの場合、*dst* には *src* のリンク先のファイルがコピーされます。

copy() はファイルのデータとパーミッションをコピーします。(*os.chmod()* を参照) その他の、ファイルの作成時間や変更時間などのメタデータはコピーしません。コピー元のファイルのメタデータを保存したい場合は、copy2() を利用してください。

引数 src 、dst` を指定して :ref:`監査イベント <auditing>` ``shutil.copyfile を送出します。

引数 src 、dst` を指定して :ref:`監査イベント <auditing>` ``shutil.copymode を送出します。

バージョン 3.3 で変更: *follow_symlinks* 引数が追加されました。新しく作成されたファイルのパスを返すようになりました。

バージョン 3.8 で変更: ファイルのコピーをより効率的に行うため、プラットフォーム特有の高速なコピーを行うシステムコールが利用されることがあります。**プラットフォーム依存の効率的なコピー操作**を参照してください。

shutil.copy2(*src*, *dst*, ***, *follow_symlinks=True*)

copy2() はファイルのメタデータを保持しようとすることを除けば *copy()* と等価です。

follow_symlinks が偽でかつ *src* がシンボリックリンクの場合、*copy2()* は *src* シンボリックリンク全てのメタデータを新たに作成される *dst* シンボリックリンクにコピーしようとします。しかし、この機能は全てのプラットフォームで利用可能なわけではありません。この機能の全部または一部が利用可能でないプラットフォームにおいては、*copy2()* は可能な限りのメタデータを保存しようとしますが、一方で *copy2()* はメタデータを保存できないことを理由とする例外を送出しません。

copy2() はファイルのメタデータをコピーするために *copystat()* を利用します。シンボリックリンクのメタデータを変更するためのプラットフォームサポートについては *copystat()* を参照して下さい。

引数 src 、dst` を指定して :ref:`監査イベント <auditing>` ``shutil.copyfile を送出します。

引数 src 、dst` を指定して :ref:`監査イベント <auditing>` ``shutil.copystat を送出します。

バージョン 3.3 で変更: *follow_symlinks* 引数が追加されました。拡張ファイルシステム属性もコピーしようと試みます (現在は Linux のみ)。新しく作成されたファイルへのパスを返すようになりました。

バージョン 3.8 で変更: ファイルのコピーをより効率的に行うため、プラットフォーム特有の高速なコピーを行うシステムコールが利用されることがあります。**プラットフォーム依存の効率的なコピー操作**を参照してください。

shutil.ignore_patterns(*patterns)

このファクトリ関数は、*copytree()* 関数の *ignore* 引数に渡すための呼び出し可能オブジェクトを作成します。glob 形式の *patterns* にマッチするファイルやディレクトリが無視されます。下の例を参照してください。

shutil.copytree(*src*, *dst*, *symlinks=False*, *ignore=None*, *copy_function=copy2*, *ignore_dangling_symlinks=False*, *dirs_exist_ok=False*)

Recursively copy an entire directory tree rooted at *src* to a directory named *dst* and return the destination directory. *dirs_exist_ok* dictates whether to raise an exception in case *dst* or any missing parent directory already exists.

各ディレクトリのパーミッション、最終アクセス時間、最終変更時間は *copystat()* でコピーされます。それぞれのファイルは *copy2()* でコピーされます。

symlinks が真の場合、ソースツリー内のシンボリックリンクは新しいツリーでもシンボリックになり、元のシンボリックリンクのメタデータはプラットフォームが許す限りコピーされます。偽の場合や省略された場合、リンク先のファイルの内容とメタデータが新しいツリーにコピーされます。

symlinks が偽の場合、リンク先のファイルが存在しなければ、コピー処理終了時に送出される *Error* 例外のエラーリストに例外が追加されます。オプションの *ignore_dangling_symlinks* フラグを真に設定してこのエラーを送出させないこともできます。このオプションは *os.symlink()* をサポートしていないプラットフォーム上では効果がないことに注意してください。

ignore は *copytree()* が走査しているディレクトリと *os.listdir()* が返すその内容のリストを引数として受け取ることのできる呼び出し可能オブジェクトでなければなりません。*copytree()* は再帰的に呼び出されるので、*ignore* はコピーされる各ディレクトリ毎に呼び出されます。*ignore* の戻り値はカレントディレクトリに相対的なディレクトリ名およびファイル名のシーケンス（すなわち第二引数の項目のサブセット）でなければなりません。それらの名前はコピー中に無視されます。*ignore_patterns()* を用いて glob 形式のパターンによって無視する呼び出し可能オブジェクトを作成することが出来ます。

例外が発生した場合、理由のリストとともに *Error* を送出します。

copy_function は各ファイルをコピーするために利用される呼び出し可能オブジェクトでなければなりません。*copy_function* はコピー元のパスとコピー先のパスを引数に呼び出されます。デフォルトでは *copy2()* が利用されますが、同じ特徴を持つ関数 (*shutil.copy()* など) ならどれでも利用可能です。

引数 src 、dst` を指定して :ref:`監査イベント <auditing>` ``shutil.copytree を送出します。

バージョン 3.3 で変更: *symlinks* が偽の場合メタデータをコピーします。*dst* を返すようになりました。

バージョン 3.2 で変更: カスタムコピー機能を提供できるように *copy_function* 引数が追加されました。*symlinks* が偽の時にダングリング (宙ぶらりんの) シンボリックリンクエラーを送出させないために *ignore_dangling_symlinks* 引数が追加されました。

バージョン 3.8 で変更: ファイルのコピーをより効率的に行うため、プラットフォーム特有の高速なコピーを行うシステムコールが利用されることがあります。**プラットフォーム依存の効率的なコピー操作**を参照してください。

バージョン 3.8 で追加: 引数 *dirs_exist_ok* が追加されました。

shutil.**rmtree**(*path*, *ignore_errors=False*, *onerror=None*)

ディレクトリツリー全体を削除します。*path* はディレクトリを指していなければなりません (ただしディレクトリに対するシンボリックリンクではいけません)。*ignore_errors* が真である場合、削除に失敗したことによるエラーは無視されます。偽や省略された場合はこれらのエラーは *onerror* で与えられたハンドラを呼び出して処理され、*onerror* が省略された場合は例外を送出します。

注釈: 必要な fd ベースの関数をサポートしているプラットフォームでは、シンボリックリンク攻撃に耐性のあるバージョンの *rmtree()* がデフォルトで利用されます。それ以外のプラットフォームでは、*rmtree()* の実装はシンボリックリンク攻撃の影響を受けます。適当なタイミングと環境で攻撃者はファイルシステム上のシンボリックリンクを操作して、それ以外の方法ではアクセス不可能なファイルを削除することが出来ます。アプリケーションは、どちらのバージョンの *rmtree()* が利用されているかを知るために関数のデータ属性 *rmtree.avoids_symlink_attacks* を利用することができます。

---

onerror を指定する場合、*function*, *path*, *excinfo* の 3 つの引数を受け取る呼び出し可能オブジェクトでなければなりません。

最初の引数 *function* は例外を送出した関数で、プラットフォームや実装に依存します。第二引数 *path* は *function* に渡されたパス名です。第三引数 *excinfo* は *sys.exc_info()* が返した例外の情報です。*onerror* が送出した例外は捕捉されません。

引数 `path` を指定して **監査イベント** `shutil.rmtree` を送出します。

バージョン 3.3 で変更: プラットフォームが fd ベースの関数をサポートする場合に自動的に使用されるシンボリックリンク攻撃に耐性のあるバージョンが追加されました。

バージョン 3.8 で変更: Windows では、ディレクトリへのジャンクションを削除する際にリンク先ディレクトリにあるファイルを削除しなくなりました。

rmtree.avoids_symlink_attacks

プラットフォームと実装がシンボリックリンク攻撃に耐性のあるバージョンの *rmtree()* を提供しているかどうかを示します。現在のところ、この属性は fd ベースのディレクトリアクセス関数をサポートしているプラットフォームでのみ真になります。

バージョン 3.3 で追加.

shutil.**move**(*src*, *dst*, *copy_function=copy2*)

ファイルまたはディレクトリ (*src*) を再帰的に別の場所 (*dst*) に移動して、移動先を返します。

移動先が存在するディレクトリの場合、*src* はそのディレクトリの中へ移動します。移動先が存在していてそれがディレクトリでない場合、*os.rename()* の動作によっては上書きされることがあります。

ターゲットが現在のファイルシステム上にある場合、*os.rename()* が使用されます。それ以外の場合 *copy_function* を使用して *src* を *dst* にコピーし、その後削除します。シンボリックリンクの場合には、*src* のターゲットを指す新しいシンボリックリンクが、*dst* の中または *dst* として作成され、*src* が削除されます。

copy_function を指定する場合、*src* と *dst* の 2 つの引数を持つ呼び出し可能オブジェクトを指定する必要があります。このオブジェクトは、*os.rename()* を使用できない場合に *src* を *dest* にコピーするために使用されます。ソースがディレクトリの場合、*copytree()* が呼び出され、そのディレクトリを copy_function() に渡します。デフォルトの *copy_function* は *copy2()* です。*copy()* を *copy_function* として使用すると、メタデータをともにコピーすることができない場合に移動を成功させることができます。この場合、メタデータはまったくコピーされません。

引数 src 、dst` を指定して :ref:`監査イベント <auditing>` ``shutil.move を送出します。

バージョン 3.3 で変更: 異なるファイルシステムに対する明示的なシンボリックリンク処理が追加されました。これにより GNU mv の振る舞いに適応するようになります。*dst* を返すようになりました。

バージョン 3.5 で変更: キーワード引数 *copy_function* が追加されました。

バージョン 3.8 で変更: ファイルのコピーをより効率的に行うため、プラットフォーム特有の高速なコピーを行うシステムコールが利用されることがあります。**プラットフォーム依存の効率的なコピー操作**を参照してください。

shutil.disk_usage(*path*)

指定されたパスについて、ディスクの利用状況を、名前付きタプル (*named tuple*) で返します。このタプルには *total*, *used*, *free* という属性があり、それぞれトータル、使用中、空きの容量をバイト単位で示します。*path* はファイルまたはディレクトリです。

バージョン 3.3 で追加.

バージョン 3.8 で変更: Windows においても *path* にディレクトリだけでなくファイルを指定できるようになりました。

Availability: Unix, Windows.

shutil.chown(*path*, *user=None*, *group=None*)

指定された *path* のオーナー *user* と/または *group* を変更します。

user はシステムのユーザー名か uid です。*group* も同じです。少なくともどちらかの引数を指定する必要があります。

内部で利用している *os.chown()* も参照してください。

引数 path 、user 、group を指定して **監査イベント** shutil.chown を送出します。

利用可能な環境: Unix。

バージョン 3.3 で追加.

shutil.which(*cmd*, *mode=os.F_OK | os.X_OK*, *path=None*)

cmd を実行しようとした時に実行される実行ファイルのパスを返します。*cmd* を呼び出せない場合は None を返します。

mode は *os.access()* に渡すパーミッションマスクで、デフォルトではファイルが存在して実行可能であることを確認します。

path が指定されなかった場合、*os.environ()* が利用され、"PATH" の値を返すか *os.defpath* にフォールバックします。

Windows では、*path* を指定した場合もデフォルト値を使った場合も、カレントディレクトリが最初に探されます。これはコマンドシェルが実行ファイルを探すときの動作です。また、*cmd* を *path* から検索するときに、PATHEXT 環境変数も利用します。例えば、shutil.which("python") を実行した場合、*which()* は PATHEXT を参照して *path* ディレクトリから python.exe を探すべきだということを把握します。例えば、Windows では:

```
>>> shutil.which("python")
'C:\\Python33\\python.EXE'
```

バージョン 3.3 で追加.

バージョン 3.8 で変更: *bytes* 型も使用できるようになりました。*cmd* が *bytes* 型の場合、戻り値も *bytes* 型です。

exception shutil.Error

この例外は複数ファイルの操作を行っているときに生じる例外をまとめたものです。*copytree()* に対しては例外の引数は 3 つのタプル (*srcname*, *dstname*, *exception*) からなるリストです。

プラットフォーム依存の効率的なコピー操作

Python 3.8 から、ファイルのコピーを伴う全ての関数 (*copyfile()*, *copy()*, *copy2()*, *copytree()*, および *move()*) はより効率的なファイルのコピーのためにプラットフォーム特有の " 高速なコピー" を行うことがあります (bpo-33671 を参照してください). ここで " 高速なコピー" とは、"outfd.write(infd.read())" のように Python が管理するユーザー空間のバッファを利用することを避け、コピー操作がカーネル空間内で行われることを意味します。

macOS では fcopyfile がファイルの内容（メタデータを除く）をコピーするために利用されます。

Linux では *os.sendfile()* が利用されます。

Windows では *shutil.copyfile()* はより大きなバッファサイズをデフォルトとして使います (6 KiB の代わりに 1 MiB が使われます)。また、*memoryview()* ベースの変形である *shutil.copyfileobj()* が使われます。

高速なコピー操作が失敗して出力ファイルにデータが書き込まれなかった場合、shutil はユーザーへの通知なしでより効率の低い *copyfileobj()* 関数にフォールバックします。

バージョン 3.8 で変更.

copytree の例

以下は前述の *copytree()* 関数のドキュメント文字列を省略した実装例です。本モジュールで提供される他の関数の使い方を示しています。

```
def copytree(src, dst, symlinks=False):
    names = os.listdir(src)
    os.makedirs(dst)
    errors = []
    for name in names:
        srcname = os.path.join(src, name)
        dstname = os.path.join(dst, name)
        try:
            if symlinks and os.path.islink(srcname):
                linkto = os.readlink(srcname)
                os.symlink(linkto, dstname)
            elif os.path.isdir(srcname):
                copytree(srcname, dstname, symlinks)
            else:
                copy2(srcname, dstname)
            # XXX What about devices, sockets etc.?
        except OSError as why:
            errors.append((srcname, dstname, str(why)))
        # catch the Error from the recursive copytree so that we can
        # continue with other files
        except Error as err:
            errors.extend(err.args[0])
    try:
        copystat(src, dst)
    except OSError as why:
        # can't copy file access times on Windows
        if why.winerror is None:
            errors.extend((src, dst, str(why)))
    if errors:
        raise Error(errors)
```

ignore_patterns() ヘルパ関数を利用する、もう 1 つの例です。

```
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*'))
```

この例では、.pyc ファイルと、tmp で始まる全てのファイルやディレクトリを除いて、全てをコピーします。

ignore 引数にロギングさせる別の例です。

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return []   # nothing will be ignored
```

(次のページに続く)

(前のページからの続き)

```
copytree(source, destination, ignore=_logpath)
```

rmtree の例

次の例は、Windows で一部のファイルが読み取り専用のビットセットを含む場合に、ディレクトリツリーを削除する方法を示します。onerror コールバックを使用して、読み取り専用のビットを消去し、削除を再試行します。結果として失敗が発生した場合、それらは伝搬されます:

```
import os, stat
import shutil

def remove_readonly(func, path, _):
    "Clear the readonly bit and reattempt the removal"
    os.chmod(path, stat.S_IWRITE)
    func(path)

shutil.rmtree(directory, onerror=remove_readonly)
```

11.10.2 アーカイブ化操作

バージョン 3.2 で追加.

バージョン 3.5 で変更: *xztar* 形式のサポートが追加されました。

圧縮とアーカイブ化されているファイルの読み書きの高水準なユーティリティも提供されています。これらは *zipfile* 、*tarfile* モジュールに依拠しています。

shutil.make_archive(*base_name*, *format*[, *root_dir*[, *base_dir*[, *verbose*[, *dry_run*[, *owner*[, *group*[, *logger*]]]]]]])

アーカイブファイル (zip や tar) を作成してその名前を返します。

base_name は、作成するファイルの、パスを含み、フォーマットごとの拡張子を抜いた名前です。*format* はアーカイブフォーマットで "zip" (*zlib* モジュールが利用可能な場合), "tar", "gztar" (*zlib* モジュールが利用可能な場合), "bztar" (*bz2* モジュールが利用可能な場合), "xztar" (*lzma* モジュールが利用可能な場合) のいずれかです。

root_dir はアーカイブファイルのルートとなるディレクトリです。アーカイブに含まれる全てのパスは *root_dir* からの相対パスになります。これは、アーカイブファイルを生成する前に *root_dir* へ移動することに相当します。

base_dir はアーカイブを開始するディレクトリです。すなわち、*base_dir* アーカイブに含まれるファイルとディレクトリに対する共通のプレフィックスになります。*base_dir* は *root_dir* からの相対パスでなければなりません。*base_dir* と *root_dir* を組み合わせて使う方法については *base_dir* を使ったアーカイブ化の例 を参照してください。

root_dir と *base_dir* のどちらも、デフォルトはカレントディレクトリです。

dry_run が真の場合、アーカイブは作成されませんが実行される操作は *logger* に記録されます。

owner と *group* は、tar アーカイブを作成するときに使われます。デフォルトでは、カレントのオーナーとグループを使います。

logger は **PEP 282** に互換なオブジェクトでなければなりません。これは普通は *logging.Logger* のインスタンスです。

verbose 引数は使用されず、非推奨です。

引数 base_name 、format 、root_dir 、base_dir``を指定して :ref:`監査イベント <auditing>` ``shutil.make_archive を送出します。

注釈: この関数はスレッドセーフではありません。

バージョン 3.8 で変更: format="tar" で作成されたアーカイブでは、レガシーな GNU 形式に代わってモダンな pax (POSIX.1-2001) 形式が使われます。

shutil.**get_archive_formats**()
アーカイブ化をサポートしているフォーマットのリストを返します。返されるシーケンスのそれぞれの要素は、タプル (name, description) です。

デフォルトでは、*shutil* は次のフォーマットを提供しています。

- *zip*: ZIP ファイル (*zlib* モジュールが利用可能な場合)。
- *tar*: 非圧縮の tar ファイル。POSIX.1-2001 pax 形式が使われます。
- *gztar*: gzip で圧縮された tar ファイル (*zlib* モジュールが利用可能な場合)。
- *bztar*: bzip2 で圧縮された tar ファイル (*bz2* モジュールが利用可能な場合)。
- *xztar*: xz で圧縮された tar ファイル (*lzma* モジュールが利用可能な場合)。

register_archive_format() を使って、新しいフォーマットを登録したり、既存のフォーマットに独自のアーカイバを提供したりできます。

shutil.**register_archive_format**(*name*, *function*[, *extra_args*[, *description*]])
アーカイバをフォーマット *name* に登録します。

function はアーカイブのアンパックに使用される呼び出し可能オブジェクトです。*funciton* は作成するファイルの *base_name*、続いてアーカイブを開始する元の *base_dir* (デフォルトは *os.curdir*) を受け取ります。さらなる引数は、次のキーワード引数として渡されます: *owner*, *group*, *dry_run* ならびに *logger* (*make_archive()* に渡されます)。

extra_args は、与えられた場合、(name, value) の対のシーケンスで、アーカイバ呼び出し可能オブジェクトが使われるときに追加のキーワード引数として使われます。

description は、アーカイバのリストを返す *get_archive_formats()* で使われます。デフォルトでは空の文字列です。

shutil.unregister_archive_format(*name*)
アーカイブフォーマット *name* を、サポートされているフォーマットのリストから取り除きます。

shutil.unpack_archive(*filename*[, *extract_dir*[, *format*[, *filter*]]])
アーカイブをアンパックします。*filename* はアーカイブのフルパスです。

extract_dir はアーカイブをアンパックする先のディレクトリ名です。指定されなかった場合は現在の作業ディレクトリを利用します。

format はアーカイブフォーマットで、"zip", "tar", "gztar", "bztar", "xztar" あるいは *register_unpack_format()* で登録したその他のフォーマットのどれかです。指定されなかった場合、*unpack_archive()* はアーカイブファイル名の拡張子に対して登録されたアンパッカーを利用します。アンパッカーが見つからなかった場合、*ValueError* を発生させます。

The keyword-only *filter* argument, which was added in Python 3.8.17, is passed to the underlying unpacking function. For zip files, *filter* is not accepted. For tar files, it is recommended to set it to `'data'`, unless using features specific to tar and UNIX-like filesystems. (See *Extraction filters* for details.) The `'data'` filter will become the default for tar files in Python 3.14.

引数 `filename`, `extract_dir`, `format` を指定して **監査イベント** `shutil.unpack_archive` を送出します。

> **警告:** Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of the path specified in the *extract_dir* argument, e.g. members that have absolute filenames starting with "/" or filenames with two dots "..".

バージョン 3.7 で変更: *filename* と *extract_dir* が *path-like object* を受け付けるようになりました。

バージョン 3.8.17 で変更: Added the *filter* argument.

shutil.register_unpack_format(*name*, *extensions*, *function*[, *extra_args*[, *description*]])
アンパック用のフォーマットを登録します。*name* はフォーマット名で、*extensions* はそのフォーマットに対応する拡張子 (例えば Zip ファイルに対して `.zip`) のリストです。

function is the callable that will be used to unpack archives. The callable will receive:

- the path of the archive, as a positional argument;
- the directory the archive must be extracted to, as a positional argument;
- possibly a *filter* keyword argument, if it was given to *unpack_archive()*;
- additional keyword arguments, specified by *extra_args* as a sequence of `(name, value)` tuples.

フォーマットの説明として *description* を指定することができます。これは *get_unpack_formats()* 関数によって返されます。

shutil.unregister_unpack_format(*name*)
アンパックフォーマットを登録解除します。*name* はフォーマットの名前です。

shutil.get_unpack_formats()

登録されているすべてのアンパックフォーマットをリストで返します。戻り値のリストの各要素は (name, extensions, description) の形のタプルです。

デフォルトでは、*shutil* は次のフォーマットを提供しています。

- *zip*: ZIP ファイル (対応するモジュールが利用可能な場合にのみ圧縮ファイルはアンパックされます)。
- *tar*: 圧縮されていない tar ファイル。
- *gztar*: gzip で圧縮された tar ファイル (*zlib* モジュールが利用可能な場合)。
- *bztar*: bzip2 で圧縮された tar ファイル (*bz2* モジュールが利用可能な場合)。
- *xztar*: xz で圧縮された tar ファイル (*lzma* モジュールが利用可能な場合)。

register_unpack_format() を使って新しいフォーマットや既存のフォーマットに対する別のアンパッカーを登録することができます。

アーカイブ化の例

この例では、ユーザの .ssh ディレクトリにあるすべてのファイルを含む、gzip された tar ファイルアーカイブを作成します:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
>>> make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

結果のアーカイブは、以下のものを含みます:

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
drwx------ tarek/staff       0 2010-02-01 16:23:40 ./
-rw-r--r-- tarek/staff     609 2008-06-09 13:26:54 ./authorized_keys
-rwxr-xr-x tarek/staff      65 2008-06-09 13:26:54 ./config
-rwx------ tarek/staff     668 2008-06-09 13:26:54 ./id_dsa
-rwxr-xr-x tarek/staff     609 2008-06-09 13:26:54 ./id_dsa.pub
-rw------- tarek/staff    1675 2008-06-09 13:26:54 ./id_rsa
-rw-r--r-- tarek/staff     397 2008-06-09 13:26:54 ./id_rsa.pub
-rw-r--r-- tarek/staff   37192 2010-02-06 18:23:10 ./known_hosts
```

base_dir を使ったアーカイブ化の例

この例では、上記の例 と同じく *make_archive()* の使い方を示しますが、ここでは特に *base_dir* の使い方を説明します。以下のようなディレクトリ構造があるとします。

```
$ tree tmp
tmp
└── root
    └── structure
        ├── content
        │   └── please_add.txt
        └── do_not_add.txt
```

作成するアーカイブには `please_add.txt` が含まれますが、いっぽう `do_not_add.txt` は含まないようにします。この場合以下のようにします。

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> make_archive(
...     archive_name,
...     'tar',
...     root_dir='tmp/root',
...     base_dir='structure/content',
... )
'/Users/tarek/my_archive.tar'
```

アーカイブに含まれるファイルをリストすると、以下のようになります。

```
$ python -m tarfile -l /Users/tarek/myarchive.tar
structure/content/
structure/content/please_add.txt
```

11.10.3 出力ターミナルのサイズの取得

shutil.get_terminal_size(*fallback=(columns, lines)*)

ターミナルウィンドウのサイズを取得します。

幅と高さについて、それぞれ `COLUMNS` と `LINES` という環境変数をチェックします。その変数が定義されていて値が正の整数であればそれを利用します。

典型的な `COLUMNS` や `LINES` が定義されていない場合には、*sys.__stdout__* に接続されているターミナルに *os.get_terminal_size()* を呼び出して問い合わせます。

システムが対応していない場合やターミナルに接続していないなどの理由でターミナルサイズの問い合わせに失敗した場合、`fallback` 引数に与えられた値を利用します。`fallback` のデフォルト値は `(80, 24)` で、これは多くのターミナルエミュレーターが利用しているデフォルトサイズです。

戻り値は *os.terminal_size* 型の名前付きタプルです。

参考: The Single UNIX Specification, Version 2, Other Environment Variables.

バージョン 3.3 で追加.

参考:

os **モジュール** オペレーティングシステムのインタフェース、Python の **ファイルオブジェクト** より低レベルでのファイル操作を含みます。

io **モジュール** Python 組み込みの I/O ライブラリで、抽象クラスとファイル I/O のようないくつかの具象クラスを含みます。

open() **組込み関数** Python で読み書きのためにファイルを開く標準的な方法です。

第

TWELVE

データの永続化

この章で解説されるモジュール群は Python データをディスクに永続的な形式で保存します。モジュール *pickle* とモジュール *marshal* は多くの Python データ型をバイト列に変換し、バイト列から再生成します。様々な DBM 関連モジュールはハッシュを基にした、文字列から他の文字列へのマップを保存するファイルフォーマット群をサポートします。

この章で解説されるモジュールのリスト:

12.1 pickle --- Python オブジェクトの直列化

ソースコード: Lib/pickle.py

---

pickle モジュールは Python オブジェクトの直列化および直列化されたオブジェクトの復元のためのバイナリプロトコルを実装しています。*"Pickle* **化***"* は Python オブジェクト階層をバイトストリームに変換する処理、*"***非** *pickle* **化***"* は (**バイナリファイル** または **バイトライクオブジェクト** から) バイトストリームをオブジェクト階層に復元する処理を意味します。pickle 化 (および非 pickle 化) は " 直列化 (serialization)"、" 整列化 (marshalling)"、あるいは[*1] " 平坦化 (flattening)" とも呼ばれますが、混乱を避けるため、ここでは "Pickle 化"、" 非 pickle 化" で統一します。

警告: "pickle"モジュールは **安全ではありません** 。信頼できるデータのみを非 pickle 化してください。

非 pickle 化の過程で任意のコードを実行する ような、悪意ある pickle オブジェクトを生成することが可能です。信頼できない提供元からのデータや、改竄された可能性のあるデータの非 pickle 化は絶対に行わないでください。

データが改竄されていないことを保証したい場合は、*hmac* による鍵付きハッシュ化を検討してください。

信頼できないデータを処理する場合 *json* のようなより安全な直列化形式の方が適切でしょう。*json* **との比較** を参照してください。

[*1] *marshal* モジュールと間違えないように注意してください。

12.1.1 他の Python モジュールとの関係

marshal との比較

Python には *marshal* と呼ばれるより原始的な直列化モジュールがありますが、一般的に Python オブジェクトを直列化する方法としては *pickle* を選ぶべきです。*marshal* は基本的に .pyc ファイルをサポートするために存在しています。

pickle モジュールはいくつかの点で *marshal* と明確に異なります:

- *pickle* モジュールでは、同じオブジェクトが再度直列化されることのないよう、すでに直列化されたオブジェクトについて追跡情報を保持します。*marshal* はこれを行いません。

  この機能は再帰的オブジェクトと共有オブジェクトの両方に重要な関わりをもっています。再帰的オブジェクトとは自分自身に対する参照を持っているオブジェクトです。再帰的オブジェクトは marshal で扱うことができず、実際、再帰的オブジェクトを marshal 化しようとすると Python インタプリタをクラッシュさせてしまいます。共有オブジェクトは、直列化しようとするオブジェクト階層の異なる複数の場所で同じオブジェクトに対する参照が存在する場合に生じます。共有オブジェクトを共有のままにしておくことは、変更可能なオブジェクトの場合には非常に重要です。

- *marshal* はユーザ定義クラスやそのインスタンスを直列化するために使うことができません。*pickle* はクラスインスタンスを透過的に保存したり復元したりすることができますが、クラス定義をインポートすることが可能で、かつオブジェクトが保存された際と同じモジュールで定義されていなければなりません。

- *marshal* の直列化形式は Python のバージョン間での移植性を保証していません。.pyc ファイルをサポートすることが主な役割であるため、Python 開発者は必要があれば直列化形式に非互換な変更を加える権利を有しています。いっぽう *pickle* の直列化形式は、互換性のあるプロトコルを選ぶという条件のもとで Python リリース間の後方互換性が保証されます。また処理すべきデータが Python 2 と Python 3 の間で非互換な型を含む場合も、pickle 化および非 pickle 化のコードはそのような互換性を破る言語の境界を適切に取り扱います。

json との比較

pickle プロトコルと JSON (JavaScript Object Notation) との基本的な違いは以下のとおりです:

- JSON はテキストの直列化フォーマット (大抵の場合 utf-8 にエンコードされますが、その出力は Unicode 文字列です) で、pickle はバイナリの直列化フォーマットです;

- JSON は人間が読める形式ですが、pickle はそうではありません;

- JSON は相互運用可能で Python 以外でも広く使用されていますが、pickle は Python 固有です;

- JSON は、デフォルトでは Python の組み込み型の一部しか表現することができず、カスタムクラスに対しても行えません; pickle は極めて多くの Python 組み込み型を表現できます (その多くは賢い Python 内省機構によって自動的に行われます; 複雑なケースでは **固有のオブジェクト** *API* によって対応できます)。

- pickle とは異なり、信頼できない JSON を復元するだけでは、任意のコードを実行できる脆弱性は発生しません。

参考:

json モジュール: JSON への直列化および復元を行うための標準ライブラリモジュール。

12.1.2 データストリームの形式

pickle によって使用されるデータフォーマットは Python 固有です。これは、JSON や XDR のような外部標準によって (例えばポインター共有を表わすことができないといったような) 制限を受けることがないという利点があります; ただし、これは非 Python プログラムが pickle された Python オブジェクトを再構成することができないということも意味します。

デフォルトでは、*pickle* データフォーマットは比較的コンパクトなバイナリ表現を使用します。サイズの抑制目的の最適化が必要なら、pickel されたデータを効率的に **圧縮する** ことができます。

pickletools モジュールには *pickle* によって生成されたデータストリームを解析するためのツールが含まれます。*pickletools* のソースコードには、pickle プロトコルで使用される命令コードに関する詳細なコメントがあります。

現在 pickle 化には 6 種類のプロトコルを使用できます。より高いプロトコルを使用するほど、作成された pickle を読み込むためにより高い Python のバージョンが必要になります。

- プロトコルバージョン 0 はオリジナルの「人間に判読可能な」プロトコルで、Python の初期のバージョンとの後方互換性を持ちます。
- プロトコルバージョン 1 は旧形式のバイナリフォーマットで、これも Python の初期バージョンと互換性があります。
- プロトコルバージョン 2 は Python 2.3 で導入されました。このバージョンでは **新方式のクラス** のより効率的な pickle 化を提供しました。プロトコル 2 による改良に関する情報は **PEP 307** を参照してください。
- プロトコルバージョン 3 は Python 3 で追加されました。*bytes* オブジェクトを明示的にサポートしており、Python 2.x で unpickle することはできません。これは Python 3.0 から 3.7 のデフォルトプロトコルでした。
- プロトコルバージョン 4 は Python 3.4 で追加されました。このバージョンでは巨大なオブジェクトのサポート、より多くの種類のオブジェクトの pickle 化、および一部のデータ形式の最適化が行われました。Python 3.8 からのデフォルトプロトコルです。プロトコル 4 による改良に関する情報は **PEP 3154** を参照してください。
- プロトコルバージョン 5 は Python 3.8 で追加されました。このバージョンでは帯域外データのサポートが追加され、また帯域内データに対するパフォーマンスが向上します。プロトコルバージョン 5 によってもたらされる改善についての情報は **PEP 574** を参照してください。

注釈: 直列化は永続性より原始的な概念です。*pickle* はファイルオブジェクトの読み書きを行いますが、永

続オブジェクトの命名に関する問題にも、(さらに困難な) 永続オブジェクトへの並列アクセスに関する問題にも対応しません。*pickle* モジュールは複雑なオブジェクトをバイトストリームに変換し、バイトストリームから同じ内部構造のオブジェクトに復元することができます。これらのバイトストリームはファイルに出力されることが多いでしょうが、ネットワークを介して送信したり、データベースに格納することもありえます。*shelve* モジュールは、オブジェクトを DBM 方式のデータベースファイル上で pickle 化および非 pickle 化するシンプルなインターフェースを提供します。

12.1.3 モジュールインタフェース

オブジェクト階層を直列化するには、*dumps()* 関数を呼ぶだけです。同様に、データストリームを復元するには、*loads()* 関数を呼びます。しかし、直列化および復元に対してより多くのコントロールを行いたい場合、それぞれ *Pickler* または *Unpickler* オブジェクトを作成することができます。

pickle モジュールは以下の定数を提供しています:

pickle.**HIGHEST_PROTOCOL**
: 利用可能なうち最も高い **プロトコルバージョン** (整数)。この値は *protocol* 値として関数 *dump()* および *dumps()* と、*Pickler* コンストラクターに渡すことができます。

pickle.**DEFAULT_PROTOCOL**
: pickle 化に使われるデフォルトの **プロトコルバージョン** (整数)。'HIGHEST_PROTOCOL'よりも小さい場合があります。現在のデフォルトプロトコルは 4 です。このプロトコルは Python3.4 で初めて導入され、その前のバージョンとは互換性がありません。

  バージョン 3.0 で変更: デフォルトプロトコルは 3 です。

  バージョン 3.8 で変更: デフォルトプロトコルは 4 です。

この pickle 化の手続きを便利にするために、*pickle* モジュールでは以下の関数を提供しています:

pickle.**dump**(*obj*, *file*, *protocol=None*, ***, *fix_imports=True*, *buffer_callback=None*)
: オブジェクト *obj* を pickle 化し、すでにオープンしている **ファイルオブジェクト** *file* に書き込みます。`Pickler(file, protocol).dump(obj)` と等価です。

  引数 *file*, *protocol*, *fix_imports* および *buffer_callback* は *Pickler* のコンストラクタと同じ意味になります。

  バージョン 3.8 で変更: *buffer_callback* 引数が追加されました。

pickle.**dumps**(*obj*, *protocol=None*, ***, *fix_imports=True*, *buffer_callback=None*)
: ファイルに書く代わりに、*bytes* オブジェクトとしてオブジェクト *obj* の pickle 表現を返します。

  引数 *protocol*, *fix_imports* および *buffer_callback* は *Pickler* のコンストラクタと同じ意味になります。

  バージョン 3.8 で変更: *buffer_callback* 引数が追加されました。

pickle.load(*file*, *, *fix_imports=True*, *encoding="ASCII"*, *errors="strict"*, *buffers=None*)
あるオブジェクトの pickle 表現を、オープンしている **ファイルオブジェクト** *file* から読み込み、その中で指定されているオブジェクト階層に再構成して返します。これは `Unpickler(file).load()` と等価です。

pickle のプロトコルバージョンは自動的に検出されます。したがって protocol 引数は必要ありません。pickle 化オブジェクト表現より後のバイト列は無視されます。

引数 *file*, *fix_imports*, *encoding*, *errors*, *strict* および *buffers* は *Unpickler* のコンストラクタと同じ意味になります。

バージョン 3.8 で変更: *buffers* 引数が追加されました。

pickle.loads(*data*, *, *fix_imports=True*, *encoding="ASCII"*, *errors="strict"*, *buffers=None*)
オブジェクトのピックル化表現 *data* から再構成されたオブジェクト階層を返します。*data* は バイトライクオブジェクト (*bytes-like object*) でなければなりません。

pickle のプロトコルバージョンは自動的に検出されます。したがって protocol 引数は必要ありません。pickle 化オブジェクト表現より後のバイト列は無視されます。

引数 *file*, *fix_imports*, *encoding*, *errors*, *strict* および *buffers* は *Unpickler* のコンストラクタと同じ意味になります。

バージョン 3.8 で変更: *buffers* 引数が追加されました。

pickle モジュールでは 3 つの例外を定義しています:

exception pickle.PickleError
他の pickle 化例外の共通基底クラス。*Exception* を継承しています。

exception pickle.PicklingError
Pickler が pickle 化不可能なオブジェクトに遭遇したときに送出されるエラー。*PickleError* を継承しています。

どんな種類のオブジェクトが pickle 化できるのか確認するには *pickle* **化、非** *pickle* **化できるもの** を参照してください。

exception pickle.UnpicklingError
データ破損やセキュリティ違反のような、オブジェクトを非 pickle 化するのに問題がある場合に送出されるエラー。*PickleError* を継承します。

非 picke 化の最中に他の例外が送出されることもあるので注意してください。これには AttributeError, EOFError, ImportError, IndexError が含まれます (ただし必ずしもこれらに限定されません)。

pickle モジュールでは、3 つのクラス *Pickler*, *Unpickler* および *Unpickler* を提供しています:

class pickle.Pickler(*file*, *protocol=None*, *, *fix_imports=True*, *buffer_callback=None*)
pickle 化されたオブジェクトのデータストリームを書き込むためのバイナリファイルを引数にとります。

任意の引数 *protocol* は、整数で、pickle 化で使用するプロトコルを指定します; サポートされているプ

ロトコルは 0 から *HIGHEST_PROTOCOL* までになります。指定されない場合、*DEFAULT_PROTOCOL* が使用されます。負数が与えられた場合、*HIGHEST_PROTOCOL* が使用されます。

引数 *file* は、1 バイトの引数一つを受け付ける write() メソッドを持たなければなりません。すなわち、*file* には、バイナリの書き込み用にオープンされたファイルオブジェクト、*io.BytesIO* オブジェクト、このインタフェースに適合するその他のカスタムオブジェクトをとることができます。

fix_imports が真であり、かつ、*protocol* が 3 未満の場合、pickle は新しい Python 3 の名前と Python 2 で使用されていた古いモジュール名との対応付けを試みるので、pickle データストリームは Python 2 でも読み込み可能です。

buffer_callback が None (デフォルト) の場合、バッファビューはストリームの一部として*file* 中に直列化されます。

buffer_callback が None でない場合、バッファビューを引数として何度でも呼び出すことができる関数です。コールバック関数が偽値 (None など) を返すと、与えられたバッファは **アウトオブバウンド管理** *(out-of-band)* となります; そうでない場合はインバンドで、すなわち pickle ストリーム内で、直列化されます。

buffer_callback が None でなく、かつ *protocol* が None または 5 より小さい場合はエラーとなります。

バージョン 3.8 で変更: *buffer_callback* 引数が追加されました。

dump(*obj*)

obj の pickle 化表現を、コンストラクターで与えられた、すでにオープンしているファイルオブジェクトに書き込みます。

persistent_id(*obj*)

デフォルトでは何もしません。このメソッドはサブクラスがオーバーライドできるように存在します。

persistent_id() が `None` を返す場合、通常通り *obj* が pickle 化されます。それ以外の値を返した場合、*Pickler* がその値を *obj* のために永続的な ID として出力するようになります。この永続的な ID の意味は *Unpickler.persistent_load()* によって定義されています。*persistent_id()* によって返された値自身は永続的な ID を持つことができないことに注意してください。

詳細および使用例については **外部オブジェクトの永続化** を参照してください。

dispatch_table

pickler オブジェクトのディスパッチテーブルは *copyreg.pickle()* を使用して宣言できる種類の *reduction functions* のレジストリです。これはキーがクラスでその値が減少関数のマッピング型オブジェクトです。減少関数は関連するクラスの引数を 1 個とり、*__reduce__()* メソッドと同じインタフェースでなければなりません。

デフォルトでは、pickler オブジェクトは *dispatch_table* 属性を持たず、代わりに *copyreg* モジュールによって管理されるグローバルなディスパッチテーブルを使用します。しかし、特定の pickler オブジェクトによる pickle 化をカスタマイズするために *dispatch_table* 属性に dict-like オブジェクトを設定することができます。あるいは、*Pickler* のサブクラスが *dispatch_table*

属性を持てば、そのクラスのインスタンスに対するデフォルトのディスパッチテーブルとして使用されます。

使用例については **ディスパッチテーブル** を参照してください。

バージョン 3.3 で追加.

reducer_override(*self*, *obj*)

Pickler のサブクラスで定義可能な特殊なリデューサ (reducer) です。このメソッドは *dispatch_table* 内のいかなるリデューサよりも優先されます。このメソッドは *__reduce__()* メソッドのインターフェースと適合していなければなりません。また、メソッドが `NotImplemented` を返すことにより、*dispatch_table* に登録されたリデューサにフォールバックして `obj` を直列化することもできます。

詳細な例については、**型、関数、その他のオブジェクトに対するリダクションのカスタマイズ** を参照してください。

バージョン 3.8 で追加.

fast

廃止予定です。真値が設定されれば高速モードを有効にします。高速モードは、メモの使用を無効にします。それにより余分な PUT 命令コードを生成しなくなるので pickle 化処理が高速化します。自己参照オブジェクトに対しては使用すべきではありません。さもなければ *Pickler* に無限再帰を起こさせるでしょう。

よりコンパクトな pickle 化を必要とする場合は、*pickletools.optimize()* を使用してください。

class pickle.Unpickler(*file*, ***, *fix_imports=True*, *encoding="ASCII"*, *errors="strict"*, *buffers=None*)

これは pickle データストリームの読み込みのためにバイナリファイルをとります。

pickle のプロトコルバージョンは自動的に検出されます。したがって protocol 引数は必要ありません。

引数 *file* は *io.BufferedIOBase* のインターフェースと同様に、整数を引数にとる read() 、バッファを引数にとる readinto() 、引数を取らない readline() の 3 つのメソッドを持たなければなりません。したがって、*file* はバイナリ読み込みモードでオープンされたディスク上のファイル、*io.BytesIO* オブジェクト、または上記インターフェース要件を満たす任意のカスタムオブジェクトのいずれかです。

オプション引数 *fix_imports*, *encoding* および *errors* は Python 2 で生成された pickle ストリームに対する互換性サポートを制御するために使われます。*fix_imports* が真の場合、pickle は古い Python 2 の名前を Python 3 の新しい名前に対応づけようとします。*encoding* と *errors* は pickle に Python 2 で pickle 化された 8 ビット文字列をデコードする方法を指定します; これらの引数のデフォルト値はそれぞれ 'ASCII' と 'strict' です。*encoding* は 8 ビット文字列インスタンスをバイトオブジェクトとして読み込む場合は 'bytes' を指定します。Python 2 で pickle 化された NumPy 配列および *datetime*, *date*, *time* の各インスタンスを非 pickle 化するためには `encoding='latin1'` を使う必要があります。

buffers が None (デフォルト値) の場合、非直列化に必要な全てのデータは pickle ストリームに含まれている必要があります。これは *Pickler* がインスタンス化されたとき (または *dump()* や *dumps()*

が呼び出されたとき) に *buffer_callback* 引数に None を指定したことに相当します。

buffers が None でない場合、各イテレーションで **アウトオブバウンド** *(out-of-band)* のバッファビューを参照する pickle ストリームを消費する、バッファ対応のイテラブルでなければなりません。ここに指定するバッファは Pickler オブジェクトの *buffer_callback* に順番に渡されたものです。

バージョン 3.8 で変更: *buffers* 引数が追加されました。

load()

コンストラクターで与えられたオープンしたファイルオブジェクトからオブジェクトの pickle 化表現を読み込み、その中で指定されたオブジェクト階層に再構成して返します。オブジェクトの pickle 化表現より後のバイト列は無視されます。

persistent_load(*pid*)

デフォルトで *UnpicklingError* を送出します。

もし定義されていれば、*persistent_load()* は永続的な ID *pid* によって指定されたオブジェクトを返す必要があります。永続的な ID が無効な場合、*UnpicklingError* を送出しなければなりません。

詳細および使用例については **外部オブジェクトの永続化** を参照してください。

find_class(*module*, *name*)

必要なら *module* をインポートして、そこから *name* という名前のオブジェクトを返します。ここで *module* および *name* 引数は *str* オブジェクトです。その名前が示唆することに反して *find_class()* は関数を探すためにも使われることに注意してください。

サブクラスは、どんな型のオブジェクトを、どのようにロードするか (潜在的にはセキュリティリスクの減少) に関する制御を得るためにこれをオーバーライドすることができます。詳細に関しては **グローバル変数を制限する** を参照してください。

引数 `modulet`, `name` を指定して **監査イベント** `pickle.find_class` を送出します。

class pickle.**PickleBuffer**(*buffer*)

pickle 可能なデータをあらわすバッファのラッパーです。*buffer* はバッファライクなオブジェクト (*bytes-like object*) や N 次元配列のような バッファ機能を提供する オブジェクトでなければなりません。

PickleBuffer はそれ自身バッファ機能を提供します。したがってこのクラスのインスタンスを、バッファ機能を提供するオブジェクトを期待する *memoryview* など他の API に渡すことが可能です。

PickleBuffer オブジェクトはプロトコル 5 以上でのみ直列化可能で、**アウトオブバウンド** *(out-of-band)* **の直列化** に対応しています。

バージョン 3.8 で追加.

raw()

このバッファの背後にあるメモリ領域への *memoryview* を返します。戻り値のオブジェクトはフォーマット `B` (符号なしバイト) の C-連続な 1 次元のメモリビューです。バッファが C-連続でも Fortran-連続でもない場合 *BufferError* 例外が送出されます。

`release()`
PickleBuffer オブジェクトを通じてアクセスされる背後のバッファを解放します。

12.1.4 pickle 化、非 pickle 化できるもの

以下の型は pickle 化できます:

- `None` 、`True` 、および `False`
- 整数、浮動小数点数、複素数
- 文字列、バイト列、バイト配列
- pickle 化可能なオブジェクトからなるタプル、リスト、集合および辞書
- モジュールのトップレベルで定義された関数 (`def` で定義されたもののみで `lambda` で定義されたものは含まない)
- モジュールのトップレベルで定義されている組込み関数
- モジュールのトップレベルで定義されているクラス
- *`__dict__`* 属性を持つクラス、あるいは *`__getstate__()`* メソッドの返り値が pickle 化可能なクラス (詳細は **クラスインスタンスの** *pickle* **化** を参照)。

pickle 化できないオブジェクトを pickle 化しようとすると、*`PicklingError`* 例外が送出されます。この例外が起きたとき、すでに元のファイルには未知の長さのバイト列が書き込まれている場合があります。極端に再帰的なデータ構造を pickle 化しようとした場合には再帰の深さ制限を越えてしまうかもしれず、この場合には *`RecursionError`* が送出されます。この制限は、*`sys.setrecursionlimit()`* で慎重に上げていくことは可能です。

関数 (組込みおよびユーザー定義) は、値ではなく、" 完全修飾" された名前参照で pickle 化されます。[*2] これは関数が定義されたモジュールをともにした関数名のみが pickle 化されることを意味します。関数のコードやその属性は pickle 化されません。すなわち、非 pickle 化する環境で定義したモジュールがインポート可能な状態になっており、そのモジュール内に関数名のオブジェクトが含まれていなければなりません。この条件を満たさなかった場合は例外が送出されます。[*3]

クラスも同様に名前参照で pickle 化されるので、unpickle 化環境には同じ制限が課せられます。クラス中のコードやデータは何も pickle 化されないので、以下の例ではクラス属性 `attr` が unpickle 化環境で復元されないことに注意してください

```
class Foo:
    attr = 'A class attribute'

picklestring = pickle.dumps(Foo)
```

[*2] なぜ `lambda` 関数を pickle 化できないかというと、すべての `lambda` 関数は同じ名前: `<lambda>` を共有しているからです。
[*3] 送出される例外は *`ImportError`* や *`AttributeError`* になるはずですが、他の例外も起こりえます。

pickle 化可能な関数やクラスがモジュールのトップレベルで定義されていなければならないのはこれらの制限のためです。

同様に、クラスのインスタンスが pickle 化された際、そのクラスのコードおよびデータはオブジェクトと一緒に pickle 化されることはありません。インスタンスのデータのみが pickle 化されます。この仕様は、クラス内のバグを修正したりメソッドを追加した後でも、そのクラスの以前のバージョンで作られたオブジェクトを読み出せるように意図的に行われています。あるクラスの多くのバージョンで使われるような長命なオブジェクトを作ろうと計画しているなら、そのクラスの *__setstate__()* メソッドによって適切な変換が行われるようにオブジェクトのバージョン番号を入れておくとよいかもしれません。

12.1.5 クラスインスタンスの pickle 化

この節では、クラスインスタンスがどのように pickle 化または非 pickle 化されるのかを定義したり、カスタマイズしたり、コントロールしたりするのに利用可能な一般的機構について説明します。

ほとんどの場合、インスタンスを pickle 化できるようにするために追加のコードは必要ありません。デフォルトで、pickle はインスタンスのクラスと属性を内省によって検索します。クラスインスタンスが非 pickle 化される場合、通常その `__init__()` メソッドは実行 **されません** 。デフォルトの振る舞いは、最初に初期化されていないインスタンスを作成して、次に保存された属性を復元します。次のコードはこの振る舞いの実装を示しています:

```
def save(obj):
    return (obj.__class__, obj.__dict__)

def load(cls, attributes):
    obj = cls.__new__(cls)
    obj.__dict__.update(attributes)
    return obj
```

クラスは、いくつかの特殊メソッドを提供することによって、デフォルトの振る舞いを変更することができます:

object.__getnewargs_ex__()

プロトコル 2 以上の場合、*__getnewargs_ex__()* メソッドを実装したクラスは `__new__()` メソッドに渡された値の非 pickle 化の方法を指示することができます。このメソッドは、オブジェクトの生成に必要な位置引数のタプル *args* と名前付き引数の辞書 *kwargs* のペア (`args, kwargs`) を返さなければなりません。これらは非 pickle 化に際して `__new__()` メソッドに渡されます。

クラスの `__new__()` メソッドがキーワード専用引数を求める場合はこのメソッドを実装すべきです。そうしない場合、互換性のため *__getnewargs__()* メソッドの実装を推奨します。

バージョン 3.6 で変更: *__getnewargs_ex__()* がプロトコル 2 と 3 でも使われるようになりました。

object.__getnewargs__()

このメソッドは *__getnewargs_ex__()* と同じような機能を提供しますが、位置引数のみをサポートします。このメソッドは引数のタプル `args` を返さなければならず、戻り値は非 pickle 化に際して `__new__()` メソッドに渡されます。

__getnewargs_ex__() が定義されていると *__getnewargs__()* は呼び出しません。

バージョン 3.6 で変更: Python 3.6 以前のプロトコル 2 と 3 では、*__getnewargs_ex__()* の代わりに *__getnewargs__()* が呼び出されていました。

object.__getstate__()

クラスはそのインスタンスをどう pickle 化するかについてさらに影響を与えることができます; クラスに *__getstate__()* メソッドが定義されていた場合それが呼ばれ、返り値のオブジェクトはインスタンスの辞書ではなく、インスタンスの内容が pickle 化されたものになります。*__getstate__()* がないときは通常通りインスタンスの *__dict__* が pickle 化されます。

object.__setstate__(*state*)

非 pickle 化に際して、クラスが *__setstate__()* を定義している場合、それは非 pickle 化された状態とともに呼び出されます。その場合、状態オブジェクトが辞書でなければならないという要求はありません。そうでなければ、pickle された状態は辞書で、その要素は新しいインスタンスの辞書に割り当てられます。

注釈: *__getstate__()* が偽値を返す場合、非 pickle 化時に *__setstate__()* メソッドは呼ばれません。

__getstate__() および *__setstate__()* メソッドの使い方に関する詳細な情報については **状態を持つオブジェクトの扱い** 節を参照してください。

注釈: 非 pickle 化に際しては、__getattr__(), __getattribute__(), または __setattr__() といったメソッドがインスタンスに対して呼ばれることがあります。これらのメソッドが何らかの内部の不変な条件が真であることを必要とする場合、その型は __new__() メソッドを実装してそのような不変な条件を構築すべきです。なぜならばインスタンスの非 pickle 化においては __init__() メソッドは呼ばれないからです。

これらから見るように、pickle は上記のメソッドを直接使用しません。実際には、これらのメソッドは *__reduce__()* 特殊メソッドを実装するコピープロトコルの一部です。コピープロトコルは、pickle 化とオブジェクトのコピーに必要な、データを取得するための統一されたインタフェースを提供します。[*4]

強力ですが、クラスに *__reduce__()* メソッドを直接実装することはエラーを起こしやすくなります。この理由のため、クラスの設計者は可能なかぎり高レベルインタフェース (*__getnewargs_ex__()*、*__getstate__()* および *__setstate__()*) を使用するべきです。公開はしているものの、*__reduce__()* の使用は、あくまでオプションとして、より効果的な pickle 化につながる場合、あるいはその両方の場合のみにしてください。

object.__reduce__()

このインタフェースは現在、以下のように定義されています。*__reduce__()* メソッドは引数を取らず、文字列あるいは (こちらの方が好まれますが) タプルのいずれかを返すべきです (返されたオブジェクトは、しばしば ”reduce value” と呼ばれます)。

文字列が返された場合、その文字列はグローバル変数の名前として解釈されます。それはオブジェクト

[*4] *copy* モジュールは、浅いコピーと深いコピーの操作にこのプロトコルを使用します。

のモジュールから見たローカル名であるべきです; pickle モジュールは、オブジェクトのモジュールを決定するためにモジュールの名前空間を検索します。この振る舞いは、典型的にシングルトンで便利です。

タプルが返された場合、それは 2〜6 要素長でなければなりません。オプションのアイテムは省略することができます。あるいはそれらの値として `None` を渡すことができます。各要素の意味は順に:

- オブジェクトの初期バージョンを作成するために呼ばれる呼び出し可能オブジェクト。
- 呼出し可能オブジェクトに対する引数のタプル。呼出し可能オブジェクトが引数を受け取らない場合、空のタプルが与えられなければなりません。
- 任意で、前述のオブジェクトの *__setstate__()* メソッドに渡されるオブジェクトの状態。オブジェクトがそのようなメソッドを持たない場合、値は辞書でなければならず、それはオブジェクトの *__dict__* 属性に追加されます。
- 任意で、連続した要素を yield する (シーケンスではなく) イテレーター。これらの要素は `obj.append(item)` を使用して、あるいはバッチでは `obj.extend(list_of_items)` を使用して、オブジェクトに追加されます。これは主としてリストのサブクラスに対して使用されますが、適切なシグネチャを持つ `append()` および `extend()` メソッドがあるかぎり、他のクラスで使用することもできます。(`append()` または `extend()` のどちらが使用されるかは、どの pickle プロトコルバージョンが使われるかに加えて追加されるアイテムの数にも依存します。したがって、両方をサポートする必要があります)
- 任意で、連続する key-value ペアを yield する (シーケンスでなく) イテレーター。これらの要素は `obj[key] = value` を使用して、オブジェクトに格納されます。これは主として辞書のサブクラスに対して使用されますが、`__setitem__()` を実装しているかぎり他のクラスで使用することもできます。
- 任意で、シグネチャが `(obj, state)` である呼び出し可能オブジェクト。このオブジェクトは、`obj` のスタティックな *__setstate__()* メソッドの代わりに、ユーザーがオブジェクトの状態を更新する方法をプログラム的に制御することを許します。`None` 以外の場合、この呼び出し可能オブジェクトは `obj` の *__setstate__()* メソッドに優先します。

  バージョン 3.8 で追加: 任意の 6 番目のタプル要素 `(obj, state)` が追加されました。

`object.__reduce_ex__`(*protocol*)

別の方法として、*__reduce_ex__()* メソッドを定義することもできます。唯一の違いは、このメソッドは単一の整数引数、プロトコルバージョンを取る必要があるということです。もし定義された場合、pickle は *__reduce__()* メソッドよりもこのメソッドを優先します。さらに、*__reduce__()* は自動的に拡張版の同義語になります。このメソッドの主な用途は、古い Python リリースに対して後方互換性のある reduce value を提供することです。

外部オブジェクトの永続化

オブジェクトの永続化のために、*pickle* モジュールは、pickle データストリーム外のオブジェクトに対する参照の概念をサポートしています。そのようなオブジェクトは永続的 ID によって参照されます。それは、英数文字の文字列 (プロトコル 0 に対して)[*5] あるいは単に任意のオブジェクト (より新しい任意のプロトコルに対して) のいずれかです。

そのような永続的 ID の分解能は *pickle* モジュールでは定義されていません; これはこの分解能を pickler および unpickler のそれぞれ *persistent_id()* および *persistent_load()* 上でのユーザー定義メソッドに移譲します。

外部の永続的 ID を持つ pickle オブジェクトの pickler は、引数にオブジェクトを取り、None かオブジェクトの永続的 ID を返すカスタム *persistent_id()* メソッドを持たなくてはなりません。None を返す場合、pickler は通常通りマーカーとともにオブジェクトを pickle 化するため、unpickler はそれを永続的 ID として認識します。

外部オブジェクトを非 pickle 化するには、unpickler は永続的 ID オブジェクトを取り被参照オブジェクトを返すカスタム *persistent_load()* メソッドを持たなくてはなりません。

これは、外部のオブジェクトを参照によって pickle 化するために永続的 ID をどのように使用するかを示す包括的な例です。

```
# Simple example presenting how persistent ID can be used to pickle
# external objects by reference.

import pickle
import sqlite3
from collections import namedtuple

# Simple class representing a record in our database.
MemoRecord = namedtuple("MemoRecord", "key, task")

class DBPickler(pickle.Pickler):

    def persistent_id(self, obj):
        # Instead of pickling MemoRecord as a regular class instance, we emit a
        # persistent ID.
        if isinstance(obj, MemoRecord):
            # Here, our persistent ID is simply a tuple, containing a tag and a
            # key, which refers to a specific record in the database.
            return ("MemoRecord", obj.key)
        else:
            # If obj does not have a persistent ID, return None. This means obj
            # needs to be pickled as usual.
            return None


class DBUnpickler(pickle.Unpickler):
```

(次のページに続く)

[*5] 英数文字に関する制限は、プロトコル 0 では永続的な ID が改行文字によって区切られるという事実によります。そのため、永続的な ID に何らかの改行文字が含まれると、結果として生じる pickle は判読不能になります。

(前のページからの続き)

```
    def __init__(self, file, connection):
        super().__init__(file)
        self.connection = connection

    def persistent_load(self, pid):
        # This method is invoked whenever a persistent ID is encountered.
        # Here, pid is the tuple returned by DBPickler.
        cursor = self.connection.cursor()
        type_tag, key_id = pid
        if type_tag == "MemoRecord":
            # Fetch the referenced record from the database and return it.
            cursor.execute("SELECT * FROM memos WHERE key=?", (str(key_id),))
            key, task = cursor.fetchone()
            return MemoRecord(key, task)
        else:
            # Always raises an error if you cannot return the correct object.
            # Otherwise, the unpickler will think None is the object referenced
            # by the persistent ID.
            raise pickle.UnpicklingError("unsupported persistent object")


def main():
    import io
    import pprint

    # Initialize and populate our database.
    conn = sqlite3.connect(":memory:")
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE memos(key INTEGER PRIMARY KEY, task TEXT)")
    tasks = (
        'give food to fish',
        'prepare group meeting',
        'fight with a zebra',
        )
    for task in tasks:
        cursor.execute("INSERT INTO memos VALUES(NULL, ?)", (task,))

    # Fetch the records to be pickled.
    cursor.execute("SELECT * FROM memos")
    memos = [MemoRecord(key, task) for key, task in cursor]
    # Save the records using our custom DBPickler.
    file = io.BytesIO()
    DBPickler(file).dump(memos)

    print("Pickled records:")
    pprint.pprint(memos)

    # Update a record, just for good measure.
    cursor.execute("UPDATE memos SET task='learn italian' WHERE key=1")
```

(次のページに続く)

(前のページからの続き)

```
    # Load the records from the pickle data stream.
    file.seek(0)
    memos = DBUnpickler(file, conn).load()

    print("Unpickled records:")
    pprint.pprint(memos)


if __name__ == '__main__':
    main()
```

ディスパッチテーブル

pickle 化に依存する他のコードの邪魔をせずに、一部のクラスの pickle 化だけをカスタマイズしたい場合、プライベートのディスパッチテーブルを持つ pickler を作成することができます。

copyreg モジュールによって管理されるグローバルなディスパッチテーブルは copyreg.dispatch_table として利用可能です。したがって、copyreg.dispatch_table の修正済のコピーをプライベートのディスパッチテーブルとして使用することを選択できます。

例えば

```
f = io.BytesIO()
p = pickle.Pickler(f)
p.dispatch_table = copyreg.dispatch_table.copy()
p.dispatch_table[SomeClass] = reduce_SomeClass
```

これは SomeClass クラスを特別に扱うプライベートのディスパッチテーブルを持つ *pickle.Pickler* のインスタンスを作成します。あるいは、次のコード

```
class MyPickler(pickle.Pickler):
    dispatch_table = copyreg.dispatch_table.copy()
    dispatch_table[SomeClass] = reduce_SomeClass
f = io.BytesIO()
p = MyPickler(f)
```

も同じことをしますが、MyPickler のすべてのインスタンスはデフォルトで同じディスパッチテーブルを共有します。*copyreg* モジュールを使用する等価なコードは

```
copyreg.pickle(SomeClass, reduce_SomeClass)
f = io.BytesIO()
p = pickle.Pickler(f)
```

状態を持つオブジェクトの扱い

ここでは、クラスを pickle 化する振る舞いの変更手順を紹介しています。TextReader クラスはテキストファイルをオープンし、readline() メソッドが呼ばれると、その度に行番号と行の内容を返します。TextReader インスタンスが pickle 化されるとき、ファイルオブジェクトメンバーを **除く** すべての属性が保存されます。インスタンスが非 pickle 化されるとき、ファイルは再びオープンされ、最後に読み込んだ位置から読み込みを再開します。このような振る舞いを実装するには *__setstate__()* および *__getstate__()* メソッドを使用します。

```
class TextReader:
    """Print and number lines in a text file."""

    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename)
        self.lineno = 0

    def readline(self):
        self.lineno += 1
        line = self.file.readline()
        if not line:
            return None
        if line.endswith('\n'):
            line = line[:-1]
        return "%i: %s" % (self.lineno, line)

    def __getstate__(self):
        # Copy the object's state from self.__dict__ which contains
        # all our instance attributes. Always use the dict.copy()
        # method to avoid modifying the original state.
        state = self.__dict__.copy()
        # Remove the unpicklable entries.
        del state['file']
        return state

    def __setstate__(self, state):
        # Restore instance attributes (i.e., filename and lineno).
        self.__dict__.update(state)
        # Restore the previously opened file's state. To do so, we need to
        # reopen it and read from it until the line count is restored.
        file = open(self.filename)
        for _ in range(self.lineno):
            file.readline()
        # Finally, save the file.
        self.file = file
```

使用例は以下のようになるでしょう:

```
>>> reader = TextReader("hello.txt")
>>> reader.readline()
'1: Hello world!'
```

(次のページに続く)

(前のページからの続き)

```
>>> reader.readline()
'2: I am line number two.'
>>> new_reader = pickle.loads(pickle.dumps(reader))
>>> new_reader.readline()
'3: Goodbye!'
```

12.1.6 型、関数、その他のオブジェクトに対するリダクションのカスタマイズ

バージョン 3.8 で追加.

dispatch_table は、ときにその柔軟性が十分でないがことあります。特に、オブジェクトの型以外の別の条件で pickle 化をカスタマイズしたい場合や、関数やクラスを使って pickle 化をカスタマイズしたい場合などです。

そのような場合、*Pickler* クラスから派生したサブクラスで *reducer_override()* メソッドを実装することができます。このメソッドは任意のリダクション用タプルを返すことができます (*__reduce__()* を参照してください)。もしくは、従来の振る舞いにフォールバックするために NotImplemented を返すこともできます。

dispatch_table と *reducer_override()* の両方が定義されている場合、*reducer_override()* メソッドが優先されます。

注釈: パフォーマンス上の理由により、次に挙げるオブジェクトに対しては *reducer_override()* が呼ばれないことがあります: None, True, False, および *int*, *float*, *bytes*, *str*, *dict*, *set*, *frozenset*, *list*, *tuple* の厳密なインスタンス。

以下は特定のクラスを pickle 化して再構成する単純な例です:

```
import io
import pickle

class MyClass:
    my_attribute = 1

class MyPickler(pickle.Pickler):
    def reducer_override(self, obj):
        """Custom reducer for MyClass."""
        if getattr(obj, "__name__", None) == "MyClass":
            return type, (obj.__name__, obj.__bases__,
                          {'my_attribute': obj.my_attribute})
        else:
            # For any other object, fallback to usual reduction
            return NotImplemented

f = io.BytesIO()
p = MyPickler(f)
```

(次のページに続く)

(前のページからの続き)

```
p.dump(MyClass)

del MyClass

unpickled_class = pickle.loads(f.getvalue())

assert isinstance(unpickled_class, type)
assert unpickled_class.__name__ == "MyClass"
assert unpickled_class.my_attribute == 1
```

12.1.7 アウトオブバウンドバッファ

バージョン 3.8 で追加.

ある状況では、*pickle* モジュールは大量のデータを転送するために使われます。そのため、メモリのコピーを最小限に抑えてパフォーマンスとリソースの消費を良好な状態に保つことが重要になることがあります。しかし、オブジェクトのグラフ的構造をシーケンシャルなバイトストリームに変換する *pickle* モジュールの通常の処理は、本質的に pickle ストリームへの、または pickle ストリームからのデータのコピーを伴います。

この制約は、生産者 *provider* (変換されるオブジェクトの型の実装) と消費者 *consumer* (通信システムの実装) が pickle プロトコル 5 以上で提供されるアウトオブバウンドのデータ転送機能をサポートしていれば回避できます。

生産者 API

pickle 化される大きなサイズのデータオブジェクトは、プロトコル 5 以上でサポートされた *__reduce_ex__()* メソッドを実装しなければなりません。このメソッドは大きなデータに対して (*bytes* オブジェクトなどの代わりに) *PickleBuffer* インスタンスを返します。

PickleBuffer オブジェクトは背後にあるバッファがアウトオブバウンドのデータ転送に適合していることを **知らせます** 。これらのオブジェクトは *pickle* モジュールの通常の使い方との互換性を保っています。しかし、消費者側で *pickle* モジュールに対してそれらのバッファを自身で処理することを事前に知らせることもできます。

消費者 API

通信システムは、オブジェクトグラフを直列化するときに生成された *PickleBuffer* オブジェクトのカスタマイズされた処理を有効化することができます。

送信側は *buffer_callback* 引数を *Pickler* (または *dump()* や *dumps()* 関数) に渡す必要があります。この関数はオブジェクトグラフを pickle 化するときに生成されるそれぞれの *PickleBuffer* を引数として呼ばれます。*buffer_callback* によって蓄積されたバッファは、それが保持するデータのコピーを pickle ストリームに送らず、軽量なマーカーが挿入されるだけです。

受信側は *buffers* 引数を *Unpickler* (または *load()* や *loads()* 関数) に渡す必要があります。これは *buffer_callback* に渡されたバッファのイテラブルです。このイテラブルは *buffer_callback* に渡されたのと

同じ順番でバッファを返さなければなりません。これらのバッファは、pickle 化処理によって *PickleBuffer* オブジェクトを生成したオブジェクトの再構築処理で期待されるデータを提供します。

送信側と受信側の間で、通信システムはアウトオブバウンドバッファの独自の転送メカニズムを自由に実装することができます。見込みのある最適化としては、共有メモリの利用や、データタイプ依存のデータ圧縮などが考えられます。

使用例

以下は、アウトオブバウンドのバッファを使った pickle 処理に関与することができるサブクラス *bytearray* を実装したささいな例です:

```
class ZeroCopyByteArray(bytearray):

    def __reduce_ex__(self, protocol):
        if protocol >= 5:
            return type(self)._reconstruct, (PickleBuffer(self),), None
        else:
            # PickleBuffer is forbidden with pickle protocols <= 4.
            return type(self)._reconstruct, (bytearray(self),)

    @classmethod
    def _reconstruct(cls, obj):
        with memoryview(obj) as m:
            # Get a handle over the original buffer object
            obj = m.obj
            if type(obj) is cls:
                # Original buffer object is a ZeroCopyByteArray, return it
                # as-is.
                return obj
            else:
                return cls(obj)
```

再構成関数 (`_reconstruct` クラスメソッド) は、受け取ったバッファが持っているオブジェクトを、それが正しい型であれば、そのまま返します。これは、このおもちゃのような例において、ゼロコピーの挙動を模擬的に行う簡単な方法です。

消費者側では、これらのオブジェクトを通常の方法で pickle 化することができます。この場合非直列化処理は元のオブジェクトのコピーを返します:

```
b = ZeroCopyByteArray(b"abc")
data = pickle.dumps(b, protocol=5)
new_b = pickle.loads(data)
print(b == new_b)  # True
print(b is new_b)  # False: a copy was made
```

いっぽう直列化において *buffer_callback* を設定し、非直列化において蓄積されたバッファを渡した場合、コピーではなく元のオブジェクトを得ることができます:

```
b = ZeroCopyByteArray(b"abc")
buffers = []
data = pickle.dumps(b, protocol=5, buffer_callback=buffers.append)
new_b = pickle.loads(data, buffers=buffers)
print(b == new_b)  # True
print(b is new_b)  # True: no copy was made
```

この例では *bytearray* がそれ自身メモリを割り当てるという性質による制限があります: すなわち、他のオブジェクトのメモリを参照する *bytearray* を生成することはできません。しかし、NumPy 配列のようなサードパーティのデータ型ではそのような制限はなく、異なるプロセス間または異なるシステム間で、ゼロコピー (または最小限のコピー) での pickle 処理の利用が可能です。

参考:

PEP 574 -- Pickle プロトコルバージョン 5 による帯域外データ

12.1.8 グローバル変数を制限する

デフォルトで、非 pickle 化は pickle データ内で見つけたあらゆるクラスや関数をインポートします。多くのアプリケーションでは、この振る舞いは受け入れられません。なぜなら、それによって unpickler が任意のコードをインポートして実行することが可能になるからです。この手の巧妙に作られた pickle データストリームがロードされたときに何を行うかをちょっと考えてみてください:

```
>>> import pickle
>>> pickle.loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
hello world
0
```

この例において、unpickler は *os.system()* 関数をインポートして、次に文字列の引数 "echo hello world" を適用しています。この例は無害ですが、システムを破壊する例を想像するのは難しくありません。

この理由のため、*Unpickler.find_class()* をカスタマイズすることで非 pickle 化で何を得るかを制御したくなるかもしれません。その名前が示唆するのと異なり、*Unpickler.find_class()* はグローバル (クラスや関数) が必要とした時にはいつでも呼びだされます。したがって、グローバルを完全に禁止することも安全なサブセットに制限することも可能です。

これは、一部の安全なクラスについてのみ *builtins* モジュールからロードすることを許可する unpickler の例です:

```
import builtins
import io
import pickle

safe_builtins = {
    'range',
    'complex',
    'set',
    'frozenset',
```

(次のページに続く)

(前のページからの続き)

```
    'slice',
}

class RestrictedUnpickler(pickle.Unpickler):

    def find_class(self, module, name):
        # Only allow safe classes from builtins.
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # Forbid everything else.
        raise pickle.UnpicklingError("global '%s.%s' is forbidden" %
                                     (module, name))

def restricted_loads(s):
    """Helper function analogous to pickle.loads()."""
    return RestrictedUnpickler(io.BytesIO(s)).load()
```

この unpickler が働く使用例は次のように意図されます:

```
>>> restricted_loads(pickle.dumps([1, 2, range(15)]))
[1, 2, range(0, 15)]
>>> restricted_loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'os.system' is forbidden
>>> restricted_loads(b'cbuiltins\neval\n'
...                  b'(S\'getattr(__import__("os"), "system")'
...                  b'("echo hello world")\'\ntR.')
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'builtins.eval' is forbidden
```

この例が示すように、非 pickle 化を認めるものに注意しなければなりません。したがって、セキュリティが重要な場合は *xmlrpc.client* の marshal API や、サードパーティのソリューションのような別の選択肢を考慮した方がよいでしょう。

12.1.9 性能

pickle プロトコルの最近のバージョン (プロトコル 2 以降) は一部の一般的な機能と組み込みデータ型を効率的にバイナリにエンコードするよう考慮されています。また、*pickle* モジュールは C 言語で書かれた透過的オプティマイザーを持っています。

12.1.10 使用例

最も単純なコードでは、*dump()* および *load()* 関数を使用してください。

```
import pickle

# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

次の例は、pickle 化されたデータを読み込みます。

```
import pickle

with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so we do not
    # have to specify it.
    data = pickle.load(f)
```

参考:

copyreg **モジュール**　拡張型を登録するための Pickle インタフェース構成機構。

pickletools **モジュール**　pickle データの処理や分析を行うためのツール。

shelve **モジュール**　オブジェクトのインデクス付きデータベース; *pickle* を使います。

copy **モジュール**　オブジェクトの浅いコピーおよび深いコピー。

marshal **モジュール**　組み込み型の高性能な直列化。

脚注

12.2 copyreg --- pickle サポート関数を登録する

ソースコード: Lib/copyreg.py

---

copyreg モジュールは、特定のオブジェクトを pickle する際に使われる関数を定義する手段を提供します。*pickle* モジュールと *copy* モジュールは、オブジェクトを pickle/コピーする場合にそれらの関数を使用します。このモジュールは、クラスでないオブジェクトコンストラクタに関する設定情報を提供します。そのようなコンストラクタは、ファクトリ関数か、クラスインスタンスかもしれません。

copyreg.**constructor**(*object*)

object を有効なコンストラクタであると宣言します。*object* が呼び出し可能でなければ (したがってコンストラクタとして有効でなければ)、*TypeError* を発生します。

copyreg.**pickle**(*type*, *function*, *constructor=None*)

function が型 *type* のオブジェクトに対する" リダクション" 関数として使われるように宣言します。*function* は文字列か、2 要素または 3 要素を含んだタプルを返さなければなりません。

オプションの *constructor* パラメータが与えられた場合、それは呼び出し可能オブジェクトで、*function* が返した引数のタプルとともに pickle 化時に呼ばれてオブジェクトを再構築するために使われます。*object* がクラスの場合、または *constructor* が呼び出し可能でない場合には *TypeError* が発生します。

function と *constructor* に期待されるインタフェースについての詳細については *pickle* モジュールを参照してください。pickler オブジェクトまたは *pickle.Pickler* のサブクラスの *dispatch_table* 属性を、リダクション関数の宣言のために使うこともできるということは覚えておいてください。

12.2.1 使用例

下記の例は、pickle 関数を登録する方法と、それがどのように使用されるかを示そうとしています:

```
>>> import copyreg, copy, pickle
>>> class C(object):
...     def __init__(self, a):
...         self.a = a
...
>>> def pickle_c(c):
...     print("pickling a C instance...")
...     return C, (c.a,)
...
>>> copyreg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)
pickling a C instance...
>>> p = pickle.dumps(c)
pickling a C instance...
```

12.3 shelve --- Python オブジェクトの永続化

ソースコード: Lib/shelve.py

" シェルフ (shelf, 棚)" は辞書に似た永続性を持つオブジェクトです。"dbm" データベースとの違いは、シェルフの値 (キーではありません！) は実質上どんな Python オブジェクトにも --- *pickle* モジュールが扱えるなら何でも --- できるということです。これにはほとんどのクラスインスタンス、再帰的なデータ型、沢山の共有されたサブオブジェクトを含むオブジェクトが含まれます。キーは通常の文字列です。

shelve.**open**(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

永続的な辞書を開きます。指定された *filename* は、根底にあるデータベースの基本ファイル名となり

ます。副作用として、*filename* には拡張子がつけられる場合があり、ひとつ以上のファイルが生成される可能性もあります。デフォルトでは、根底にあるデータベースファイルは読み書き可能なように開かれます。オプションの *flag* パラメータは *dbm.open()* における *flag* パラメータと同様に解釈されます。

デフォルトでは、値を整列化する際にはバージョン 3 の pickle 化が用いられます。pickle 化プロトコルのバージョンは *protocol* パラメータで指定することができます。

Python の意味論により、シェルフには永続的な辞書の可変エントリがいつ変更されたかを知る術がありません。デフォルトでは、変更されたオブジェクトはシェルフに代入されたとき **だけ** 書き込まれます (**使用例** 参照)。オプションの *writeback* パラメータが `True` に設定されている場合は、アクセスされたすべてのエントリはメモリ上にキャッシュされ、*sync()* および *close()* を呼び出した際に書き戻されます; この機能は永続的な辞書上の可変の要素に対する変更を容易にしますが、多数のエントリがアクセスされた場合、膨大な量のメモリがキャッシュのために消費され、アクセスされた全てのエントリを書き戻す (アクセスされたエントリが可変であるか、あるいは実際に変更されたかを決定する方法は存在しないのです) ために、ファイルを閉じる操作が非常に低速になります。

注釈: シェルフが自動的に閉じることに依存しないでください; それがもう必要ない場合は常に *close()* を明示的に呼ぶか、*shelve.open()* をコンテキストマネージャとして使用してください:

```
with shelve.open('spam') as db:
    db['eggs'] = 'eggs'
```

警告: *shelve* モジュールは裏で *pickle* を使っているので、信頼できないソースからシェルフを読み込むのは危険です。pickle と同じく、shelf の読み込みでも任意のコードを実行できるからです。

シェルフオブジェクトは辞書がサポートする全てのメソッドをサポートしています。これにより、辞書ベースのスクリプトから永続的な記憶媒体を必要とするスクリプトに容易に移行できるようになります。

追加でサポートされるメソッドが二つあります:

`Shelf.sync()`
: シェルフが *writeback* を *True* にセットして開かれている場合に、キャッシュ中の全てのエントリを書き戻します。また可能な場合は、キャッシュを空にしてディスク上の永続的な辞書を同期します。このメソッドはシェルフを *close()* によって閉じるとき自動的に呼び出されます。

`Shelf.close()`
: 永続的な **辞書** オブジェクトを同期して閉じます。既に閉じられているシェルフに対して呼び出すと *ValueError* を出し失敗します。

参考:

通常の辞書に近い速度をもち、いろいろなストレージフォーマットに対応した、永続化辞書のレシピ 。

12.3.1 制限事項

- どのデータベースパッケージが使われるか (例えば *dbm.ndbm* 、*dbm.gnu*) は、どのインタフェースが利用可能かに依存します。従って、データベースを *dbm* を使って直接開く方法は安全ではありません。データベースはまた、*dbm* が使われた場合 (不幸なことに) その制約に縛られます --- これはデータベースに記録されたオブジェクト (の pickle 化された表現) はかなり小さくなければならず、キー衝突が生じた場合に、稀にデータベースを更新することができなくなることを意味します。

- *shelve* モジュールは、シェルフに置かれたオブジェクトの **並列した** 読み出し/書き込みアクセスをサポートしません (複数の同時読み出しアクセスは安全です)。あるプログラムが書き込みのために開かれたシェルフを持っているとき、他のプログラムはそのシェルフを読み書きのために開いてはいけません。この問題を解決するために Unix のファイルロック機構を使うことができますが、この機構は Unix のバージョン間で異なり、使われているデータベースの実装について知識が必要となります。

class shelve.Shelf(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)

collections.abc.MutableMapping のサブクラスで、*dict* オブジェクト内に pickle 化された値を保持します。

デフォルトでは、値を整列化する際にはバージョン 3 の pickle 化が用いられます。pickle 化プロトコルのバージョンは *protocol* パラメータで指定することができます。pickle 化プロトコルについては *pickle* のドキュメントを参照してください。

writeback パラメータが `True` に設定されていれば、アクセスされたすべてのエントリはメモリ上にキャッシュされ、ファイルを閉じる際に *dict* に書き戻されます; この機能により、可変のエントリに対して自然な操作が可能になりますが、さらに多くのメモリを消費し、辞書をファイルと同期して閉じる際に長い時間がかかるようになります。

keyencoding パラメータは、shelf の背後にある dict に対して使われる前にキーをエンコードするのに使用されるエンコーディングです。

Shelf オブジェクトは、コンテキストマネージャとしても使用できます。この場合、`with` ブロックが終了する際に、自動的に閉じられます。

バージョン 3.2 で変更: *keyencoding* パラメータを追加; 以前はキーは常に UTF-8 でエンコードされていました。

バージョン 3.4 で変更: コンテキストマネージャーサポートが追加されました。

class shelve.BsdDbShelf(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)

Shelf のサブクラスで、`first()`, `next()`, `previous()`, `last()`, `set_location()` メソッドを外部に提供しています。これらのメソッドは pybsddb にあるサードパーティの `bsddb` モジュールでは利用可能ですが、他のデータベースモジュールでは利用できません。コンストラクタに渡される *dict* オブジェクトは上記のメソッドをサポートしていなくてはなりません。通常は、`bsddb.hashopen()`, `bsddb.btopen()` または `bsddb.rnopen()` のいずれかを呼び出して得られるオブジェクトが条件を満たしています。オプションの *protocol*, *writeback* および *keyencoding* パラメータは *Shelf* クラスにおけるパラメータと同様に解釈されます。

class shelve.DbfilenameShelf(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

Shelf のサブクラスで、辞書に似たオブジェクトの代わりに *filename* を受理します。根底にあるファイルは *dbm.open()* を使って開かれます。デフォルトでは、ファイルは読み書き可能な状態で開かれます。オプションの *flag* パラメータは *open()* 関数におけるパラメータと同様に解釈されます。オプションの *protocol* および *writeback* パラメータは *Shelf* クラスにおけるパラメータと同様に解釈されます。

12.3.2 使用例

インタフェースは以下のコードに集約されています (key は文字列で、data は任意のオブジェクトです):

```
import shelve

d = shelve.open(filename)  # open -- file may get suffix added by low-level
                           # library

d[key] = data              # store data at key (overwrites old data if
                           # using an existing key)
data = d[key]              # retrieve a COPY of data at key (raise KeyError
                           # if no such key)
del d[key]                 # delete data stored at key (raises KeyError
                           # if no such key)

flag = key in d            # true if the key exists
klist = list(d.keys())     # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = [0, 1, 2]        # this works as expected, but...
d['xx'].append(3)          # *this doesn't!* -- d['xx'] is STILL [0, 1, 2]!

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']             # extracts the copy
temp.append(5)             # mutates the copy
d['xx'] = temp             # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()                  # close it
```

参考:

dbm **モジュール**　dbm スタイルのデータベースに対する共通インタフェース。

pickle **モジュール**　*shelve* によって使われるオブジェクト整列化機構。

12.4 marshal --- 内部使用向けの Python オブジェクト整列化

このモジュールには Python 値をバイナリ形式で読み書きできるような関数が含まれています。このバイナリ形式は Python 特有のものですが、マシンアーキテクチャ非依存のものです (つまり、Python の値を PC 上でファイルに書き込み、Sun に転送し、そこで読み戻すことができます)。バイナリ形式の詳細は意図的にドキュメント化されていません; この形式は (稀にしかないことですが) Python のバージョン間で変更される可能性があるからです。[*1]

このモジュールは汎用の ” 永続化 (persistence)” モジュールではありません。汎用的な永続化や、RPC 呼び出しを通じた Python オブジェクトの転送については、モジュール *pickle* および *shelve* を参照してください。*marshal* モジュールは主に、” 擬似コンパイルされた (pseudo-compiled)” コードの .pyc ファイルへの読み書きをサポートするために存在します。したがって、Python のメンテナンス担当者は、必要が生じれば marshal 形式を後方互換性のないものに変更する権利を有しています。Python オブジェクトを直列化 (シリアライズ) および非直列化 (デシリアライズ) する場合には、*pickle* モジュールを使ってください。*pickle* は速度は同等で、バージョン間の互換性が保証されていて、marshal より広範囲のオブジェクトをサポートしています。

> **警告:** *marshal* モジュールは、誤ったデータや悪意を持って作成されたデータに対する安全性を考慮していません。信頼できない、もしくは認証されていない出所からのデータを非整列化してはなりません。

すべての Python オブジェクト型がサポートされているわけではありません。一般に、このモジュールによって読み書きすることができるオブジェクトは、その値が Python の特定の起動に依存していないオブジェクトに限ります。次の型がサポートされています。真偽値、整数、浮動小数点数、複素数、文字列、byte 、bytearray 、タプル、リスト、set 、frozenset 、辞書、コードオブジェクト。ここで、タプル、リスト、set 、frozenset 、辞書は、その中に含まれる値がそれ自身サポートされる場合に限りサポートされます。シングルトン *None* 、*Ellipsis* 、*StopIteration* も読み書き (marshalled and unmarshalled) できます。3 未満のフォーマット *version* では、再帰的なリスト、set 、辞書を書き出すことはできません (下記参照)。

bytes-like オブジェクトを操作する関数と同様に、ファイルの読み書きを行う関数が提供されています。

このモジュールでは、以下の関数が定義されています。

marshal.**dump**(*value*, *file*[, *version*])

開かれたファイルに値を書き込みます。値はサポートされている型でなくてはなりません。ファイルは書き込み可能な **バイナリファイル** である必要があります。

値 (または値に含まれるオブジェクト) がサポートされていない型の場合、*ValueError* 例外が送出されます --- しかし、同時にごみのデータがファイルに書き込まれます。このオブジェクトは *load()* で適切に読み出されることはありません。

version 引数は dump が使用するデータフォーマットを指定します (下記を参照してください)。

[*1] このモジュールの名前は (特に) Modula-3 の設計者の間で使われていた用語の一つに由来しています。彼らはデータを自己充足的な形式で輸送する操作に ” 整列化 (marshalling)” という用語を使いました。厳密に言えば、” 整列させる (to marshal)” とは、あるデータを (例えば RPC バッファのように) 内部表現形式から外部表現形式に変換することを意味し、” 非整列化 (unmarshalling)” とはその逆を意味します。

marshal.load(*file*)

開かれたファイルから値を一つ読んで返します。(異なるバージョンの Python の互換性のない marshal フォーマットだったなどの理由で) 有効な値が読み出せなかった場合、*EOFError*、*ValueError*、または *TypeError* を送出します。file は読み込み可能な **バイナリファイル** でなければなりません.

注釈: サポートされていない型を含むオブジェクトが *dump()* で整列化されている場合、*load()* は整列化不能な値を None で置き換えます。

---

marshal.dumps(*value*[, *version*])

dump(value, file) でファイルに書き込まれるような bytes オブジェクトを返します。値はサポートされている型でなければなりません。値がサポートされていない型のオブジェクト (またはサポートされていない型のオブジェクトを含むようなオブジェクト) の場合、*ValueError* 例外が送出されます。

version 引数は dumps が使用するデータフォーマットを指定します (下記を参照してください)。

marshal.loads(*bytes*)

bytes-like object を値に変換します。有効な値が見つからなかった場合、*EOFError*、*ValueError*、または *TypeError* が送出されます。入力中の余分な bytes は無視されます。

これに加えて、以下の定数が定義されています:

marshal.version

このモジュールが利用するバージョンを表します。バージョン 0 は歴史的なフォーマットです。バージョン 1 は文字列の再利用をします。バージョン 2 は浮動小数点数にバイナリフォーマットを使用します。バージョン 3 はオブジェクトのインスタンス化と再帰をサポートします。現在のバージョンは 4 です。

脚注

12.5 dbm --- Unix "データベース" へのインタフェース

ソースコード: Lib/dbm/___init___.py

---

dbm は DBM データベースのいくつかの種類 (*dbm.gnu* または *dbm.ndbm*) に対する汎用的なインタフェースです。これらのモジュールのどれもインストールされていなければ、モジュール *dbm.dumb* に含まれる低速だが単純な実装が使用されます。Oracle Berkeley DB に対する サードパーティのインタフェース があります。

exception dbm.error

サポートされているモジュールそれぞれによって送出される可能性のある例外を含むタプル。これにはユニークな例外があり、最初の要素として同じく *dbm.error* という名前の例外が含まれます --- *dbm.error* が送出される場合、後者 (訳注:タプルの dbm.error ではなく例外 *dbm.error*) が使用されます。

dbm.**whichdb**(*filename*)

この関数は、与えられたファイルを開くために、利用可能ないくつかの単純なデータベースモジュール --- *dbm.gnu*, *dbm.ndbm*, *dbm.dumb* --- のどれを使用すべきか推測を試みます。

次の値のうち 1 つを返します: ファイルが読み取れないか存在しないために開くことができない場合は None; ファイルのフォーマットを推測することができない場合は空文字列 (''); それ以外は 'dbm.ndbm' や 'dbm.gnu' のような、必要なモジュール名を含む文字列。

dbm.**open**(*file*, *flag='r'*, *mode=0o666*)

データベースファイル *file* を開いて対応するオブジェクトを返します。

データベースファイルが既に存在する場合、その種類を決定するために *whichdb()* 関数が使用され、適切なモジュールが使用されます; データベースファイルが存在しない場合、上記のリストの中でインポート可能な最初のモジュールが使用されます。

オプションの *flag* は:

値	意味
'r'	既存のデータベースを読み込み専用で開く (デフォルト)
'w'	既存のデータベースを読み書き用に開く
'c'	データベースを読み書き用に開く。ただし存在しない場合には新たに作成する
'n'	常に新たに読み書き用の新規のデータベースを作成する

オプションの *mode* 引数は、新たにデータベースを作成しなければならない場合に使われる Unix のファイルモードです。標準の値は 8 進数の 0o666 です (この値は現在有効な umask で修飾されます)。

open() によって返されたオブジェクトは辞書とほとんど同じ機能をサポートします; キーとそれに対応付けられた値を記憶し、取り出し、削除することができ、in 演算子や keys() メソッド、また get() や setdefault() を使うことができます。

バージョン 3.2 で変更: get() と setdefault() がすべてのデータベースモジュールで利用できるようになりました。

バージョン 3.8 で変更: 読み出し専用のデータベースからキーを削除しようとすると、*KeyError* ではなくデータベースモジュール専用のエラーが送出されるようになりました。

キーと値は常に byte 列として格納されます。これは、文字列が使用された場合は格納される前に暗黙的にデフォルトエンコーディングに変換されるということを意味します。

これらのオブジェクトは、with 文での使用にも対応しています。with 文を使用した場合、終了時に自動的に閉じられます。

バージョン 3.4 で変更: *open()* が返すオブジェクトに対するコンテキスト管理のプロトコルがネイティブにサポートされました。

以下の例ではホスト名と対応するタイトルをいくつか記録し、データベースの内容を出力します:

```
import dbm

# Open database, creating it if necessary.
with dbm.open('cache', 'c') as db:

    # Record some values
    db[b'hello'] = b'there'
    db['www.python.org'] = 'Python Website'
    db['www.cnn.com'] = 'Cable News Network'

    # Note that the keys are considered bytes now.
    assert db[b'www.python.org'] == b'Python Website'
    # Notice how the value is now in bytes.
    assert db['www.cnn.com'] == b'Cable News Network'

    # Often-used methods of the dict interface work too.
    print(db.get('python.org', b'not present'))

    # Storing a non-string key or value will raise an exception (most
    # likely a TypeError).
    db['www.yahoo.com'] = 4

# db is automatically closed when leaving the with statement.
```

参考:

shelve **モジュール**　非文字列データを記録する永続化モジュール。

個々のサブモジュールは以降の節で説明されます。

12.5.1 dbm.gnu --- GNU による dbm 拡張

ソースコード: Lib/dbm/gnu.py

このモジュールは *dbm* モジュールによく似ていますが、GNU ライブラリ gdbm を使っていくつかの追加機能を提供しています。*dbm.gnu* と *dbm.ndbm* では生成されるファイル形式に互換性がないので注意してください。

dbm.gnu モジュールでは GNU DBM ライブラリへのインタフェースを提供します。dbm.gnu.gdbm オブジェクトはキーと値が必ず保存の前にバイト列に変換されることを除き、マップ型 (辞書型) と同じように動作します。gdbm オブジェクトに対して *print()* を適用してもキーや値を印字することはなく、items() 及び values() メソッドはサポートされていません。

exception dbm.gnu.error

I/O エラーのような *dbm.gnu* 特有のエラーで送出されます。誤ったキーの指定のように、一般的なマップ型のエラーに対しては *KeyError* が送出されます。

dbm.gnu.**open**(*filename*[, *flag*[, *mode*]])

gdbm データベースを開いて gdbm オブジェクトを返します。*filename* 引数はデータベースファイルの

名前です。

オプションの *flag* は:

値	意味
`'r'`	既存のデータベースを読み込み専用で開く (デフォルト)
`'w'`	既存のデータベースを読み書き用に開く
`'c'`	データベースを読み書き用に開く。ただし存在しない場合には新たに作成する
`'n'`	常に新たに読み書き用の新規のデータベースを作成する

以下の追加の文字を flag に追加して、データベースの開きかたを制御することができます:

値	意味
`'f'`	データベースを高速モードで開きます。書き込みが同期されません。
`'s'`	同期モード。データベースへの変更がすぐにファイルに書き込まれます。
`'u'`	データベースをロックしません。

全てのバージョンの gdbm で全てのフラグが有効とは限りません。モジュール定数 `open_flags` はサポートされているフラグ文字からなる文字列です。無効なフラグが指定された場合、例外 *error* が送出されます。

オプションの *mode* 引数は、新たにデータベースを作成しなければならない場合に使われる Unix のファイルモードです。標準の値は 8 進数の `0o666` です。

辞書型形式のメソッドに加えて、gdbm オブジェクトには以下のメソッドがあります:

gdbm.firstkey()

このメソッドと *nextkey()* メソッドを使って、データベースの全てのキーにわたってループ処理を行うことができます。探索は gdbm の内部ハッシュ値の順番に行われ、キーの値に順に並んでいるとは限りません。このメソッドは最初のキーを返します。

gdbm.nextkey(*key*)

データベースの順方向探索において、*key* よりも後に来るキーを返します。以下のコードはデータベース db について、キー全てを含むリストをメモリ上に生成することなく全てのキーを出力します:

```
k = db.firstkey()
while k != None:
    print(k)
    k = db.nextkey(k)
```

gdbm.reorganize()

大量の削除を実行した後、gdbm ファイルの占めるスペースを削減したい場合、このルーチンはデータベースを再組織化します。この再組織化を使用する方法以外に gdbm オブジェクトがデータベースファイルの大きさを短くすることはありません。サイズを縮小しない場合、削除された部分のファイルスペースは保持され、新たな (キー、値の) ペアが追加される際に再利用されます。

gdbm.sync()
データベースが高速モードで開かれていた場合、このメソッドはディスクにまだ書き込まれていないデータを全て書き込ませます。

gdbm.close()
gdbm データベースをクローズします。

12.5.2 dbm.ndbm --- ndbm に基づくインタフェース

ソースコード: Lib/dbm/ndbm.py

---

dbm.ndbm モジュールは Unix の"(n)dbm" ライブラリのインタフェースを提供します。dbm オブジェクトは、キーと値が必ずバイト列である以外は辞書オブジェクトのようなふるまいをします。print 関数などで dbm オブジェクトを出力してもキーと値は出力されません。また、items() と values() メソッドはサポートされません。

このモジュールは、GNU GDBM 互換インタフェースを持った " クラシックな" ndbm インタフェースを使うことができます。Unix 上のビルド時に **configure** スクリプトで適切なヘッダファイルが割り当られます。

exception dbm.ndbm.error
I/O エラーのような *dbm.ndbm* 特有のエラーで送出されます。誤ったキーの指定のように、一般的なマップ型のエラーに対しては *KeyError* が送出されます。

dbm.ndbm.library
ndbm が使用している実装ライブラリ名です。

dbm.ndbm.open(*filename*[, *flag*[, *mode*]])
dbm データベースを開いて ndbm オブジェクトを返します。引数 *filename* はデータベースのファイル名を指定します。(拡張子 .dir や .pag は付けません)。

オプションの *flag* は以下の値のいずれかです:

値	意味
'r'	既存のデータベースを読み込み専用で開く (デフォルト)
'w'	既存のデータベースを読み書き用に開く
'c'	データベースを読み書き用に開く。ただし存在しない場合には新たに作成する
'n'	常に新たに読み書き用の新規のデータベースを作成する

オプションの *mode* 引数は、新たにデータベースを作成しなければならない場合に使われる Unix のファイルモードです。標準の値は 8 進数の 0o666 です (この値は現在有効な umask で修飾されます)。

辞書型様のメソッドに加えて、ndbm オブジェクトには以下のメソッドがあります。

ndbm.close()
ndbm データベースをクローズします。

12.5.3 dbm.dumb --- 可搬性のある DBM 実装

ソースコード: Lib/dbm/dumb.py

注釈: *dbm.dumb* モジュールは、*dbm* が頑健なモジュールを他に見つけることができなかった際の最後の手段とされています。*dbm.dumb* モジュールは速度を重視して書かれているわけではなく、他のデータベースモジュールのように重い使い方をするためのものではありません。

---

dbm.dumb モジュールは永続性辞書に類似したインタフェースを提供し、全て Python で書かれています。*dbm.gnu* のようなモジュールと異なり、外部ライブラリは必要ありません。他の永続性マップ型のように、キーおよび値は常にバイト列として保存されます。

このモジュールは以下を定義します:

exception dbm.dumb.error
: I/O エラーのような *dbm.dumb* 特有のエラーの際に送出されます。不正なキーを指定したときのような、一般的な対応付けエラーの際には *KeyError* が送出されます。

dbm.dumb.open(*filename*[, *flag*[, *mode*]])
: dumbdbm データベースを開き、dubmdbm オブジェクトを返します。*filename* 引数はデータベースファイル名の雛型 (特定の拡張子をもたないもの) です。dumbdbm データベースが生成される際、.dat および .dir の拡張子を持ったファイルが生成されます。

  オプションの *flag* は:

値	意味
'r'	既存のデータベースを読み込み専用で開く (デフォルト)
'w'	既存のデータベースを読み書き用に開く
'c'	データベースを読み書き用に開く。ただし存在しない場合には新たに作成する
'n'	常に新たに読み書き用の新規のデータベースを作成する

  オプションの *mode* 引数は、新たにデータベースを作成しなければならない場合に使われる Unix のファイルモードです。標準の値は 8 進数の 0o666 です (この値は現在有効な umask で修飾されます)。

  警告: 十分に大きかったり複雑だったりするエントリーのあるデータベースを読み込んでいるときに、Python の抽象構文木コンパイラのスタックの深さの限界を越えるせいで、Python インタプリタをクラッシュさせることができます。

  バージョン 3.5 で変更: フラグに値 'n' を与えると、*open()* が常に新しいデータベースを作成するようになりました。

  バージョン 3.8 で変更: フラグ 'r' で開いたデータベースは読み出し専用となりました。データベースが存在していない場合にフラグ 'r' と 'w' で開いても、データベースを作成しなくなりました。

collections.abc.MutableMapping クラスによって提供されるメソッドに加えて、dumbdbm オブジェクトは以下のメソッドを提供します:

dumbdbm.sync()
: ディスク上の辞書とデータファイルを同期します。このメソッドは Shelve.sync() メソッドから呼び出されます。

dumbdbm.close()
: dumbdbm データベースをクローズします。

12.6 sqlite3 --- SQLite データベースに対する DB-API 2.0 インタフェース

ソースコード: Lib/sqlite3/

SQLite は、軽量なディスク上のデータベースを提供する C ライブラリです。別のサーバプロセスを用意する必要なく、SQL クエリー言語の非標準的な一種を使用してデータベースにアクセスできます。一部のアプリケーションは内部データ保存に SQLite を使えます。また、SQLite を使ってアプリケーションのプロトタイプを作り、その後そのコードを PostgreSQL や Oracle のような大規模データベースに移植するということも可能です。

sqlite3 モジュールの著者は Gerhard Häring です。**PEP 249** で記述されている DB-API 2.0 に準拠した SQL インターフェイスを提供します。

このモジュールを使うには、最初にデータベースを表す *Connection* オブジェクトを作ります。ここではデータはファイル example.db に格納されているものとします:

```
import sqlite3
con = sqlite3.connect('example.db')
```

特別な名前である :memory: を使うと RAM 上にデータベースを作ることもできます。

Connection があれば、*Cursor* オブジェクトを作りその *execute()* メソッドを呼んで SQL コマンドを実行することができます:

```
cur = con.cursor()

# Create table
cur.execute('''CREATE TABLE stocks
               (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
cur.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
con.commit()
```

(次のページに続く)

(前のページからの続き)

```
# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
con.close()
```

保存されたデータは永続的であり、次回のセッションでもそのまま使用できます:

```
import sqlite3
con = sqlite3.connect('example.db')
cur = con.cursor()
```

たいてい、SQL 操作では Python 変数の値を使う必要があります。この時、クエリーを Python の文字列操作を使って構築することは安全とは言えないので、すべきではありません。そのようなことをするとプログラムが SQL インジェクション攻撃に対し脆弱になります (https://xkcd.com/327/ ではどうなってしまうかをユーモラスに描いています)。

代わりに、DB-API のパラメータ割り当てを使います。? を変数の値を使いたいところに埋めておきます。その上で、値のタプルをカーソルの *execute()* メソッドの第 2 引数として引き渡します。(他のデータベースモジュールでは変数の場所を示すのに `%s` や `:1` などの異なった表記を用いることがあります。) 例を示します:

```
# Never do this -- insecure!
symbol = 'RHAT'
cur.execute("SELECT * FROM stocks WHERE symbol = '%s'" % symbol)

# Do this instead
t = ('RHAT',)
cur.execute('SELECT * FROM stocks WHERE symbol=?', t)
print(cur.fetchone())

# Larger example that inserts many records at a time
purchases = [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
             ('2006-04-05', 'BUY', 'MSFT', 1000, 72.00),
             ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
            ]
cur.executemany('INSERT INTO stocks VALUES (?,?,?,?,?)', purchases)
```

SELECT 文を実行した後データを取得する方法は 3 つありどれを使っても構いません。一つはカーソルを **イテレータ** として扱う、一つはカーソルの *fetchone()* メソッドを呼んで一致した内の一行を取得する、もう一つは *fetchall()* メソッドを呼んで一致した全ての行のリストとして受け取る、という 3 つです。

以下の例ではイテレータの形を使います:

```
>>> for row in cur.execute('SELECT * FROM stocks ORDER BY price'):
        print(row)

('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
('2006-04-06', 'SELL', 'IBM', 500, 53.0)
('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

参考:

https://www.sqlite.org　SQLite のウェブページ。ここの文書ではサポートされる SQL 方言の文法と使えるデータ型を説明しています。

https://www.w3schools.com/sql/　SQL 学習に効くチュートリアル、リファレンス、実例集。

PEP 249 - Database API Specification 2.0　Marc-Andre Lemburg により書かれた PEP。

12.6.1 モジュールの関数と定数

sqlite3.version
文字列で表現されたモジュールのバージョン番号です。これは SQLite ライブラリのバージョンではありません。

sqlite3.version_info
整数のタプルで表現されたモジュールのバージョン番号です。これは SQLite ライブラリのバージョンではありません。

sqlite3.sqlite_version
文字列で表現された SQLite ランタイムライブラリのバージョン番号です。

sqlite3.sqlite_version_info
整数のタプルで表現された SQLite ランタイムライブラリのバージョン番号です。

sqlite3.PARSE_DECLTYPES
この定数は *connect()* 関数の *detect_types* パラメータとして使われます。

この定数を設定すると *sqlite3* モジュールは戻り値のカラムの宣言された型を読み取るようになります。意味を持つのは宣言の最初の単語です。すなわち、"integer primary key" においては "integer" が読み取られます。また、"number(10)" では、"number" が読み取られます。そして、そのカラムに対して、変換関数の辞書を探してその型に対して登録された関数を使うようにします。

sqlite3.PARSE_COLNAMES
この定数は *connect()* 関数の *detect_types* パラメータとして使われます。

Setting this makes the SQLite interface parse the column name for each column it returns. It will look for a string formed [mytype] in there, and then decide that 'mytype' is the type of the column. It will try to find an entry of 'mytype' in the converters dictionary and then use the converter function found there to return the value. The column name found in *Cursor.description* does not include the type, i. e. if you use something like `'as "Expiration date [datetime]"'` in your SQL, then we will parse out everything until the first `'['` for the column name and strip the preceeding space: the column name would simply be "Expiration date".

sqlite3.connect(*database*[, *timeout*, *detect_types*, *isolation_level*, *check_same_thread*, *factory*, *cached_statements*, *uri*])
Opens a connection to the SQLite database file *database*. By default returns a *Connection* object, unless a custom *factory* is given.

database is a *path-like object* giving the pathname (absolute or relative to the current working directory) of the database file to be opened. You can use `":memory:"` to open a database connection

to a database that resides in RAM instead of on disk.

データベースが複数の接続からアクセスされている状況で、その内の一つがデータベースに変更を加えたとき、SQLite データベースはそのトランザクションがコミットされるまでロックされます。*timeout* パラメータで、例外を送出するまで接続がロックが解除されるのをどれだけ待つかを決めます。デフォルトは 5.0 (5 秒) です。

isolation_level パラメータについては、*Connection* オブジェクトの、*isolation_level* プロパティを参照してください。

SQLite はネイティブで TEXT、INTEGER、REAL、BLOB および NULL のみをサポートしています。その他のタイプを使用したい場合はあなた自身で追加しなければなりません。*detect_types* パラメータおよび、*register_converter()* 関数でモジュールレベルで登録できるカスタム **変換関数** を使用することで簡単に追加できます。

detect_types defaults to 0 (i. e. off, no type detection), you can set it to any combination of *PARSE_DECLTYPES* and *PARSE_COLNAMES* to turn type detection on. Due to SQLite behaviour, types can't be detected for generated fields (for example `max(data)`), even when *detect_types* parameter is set. In such case, the returned type is *str*.

By default, *check_same_thread* is *True* and only the creating thread may use the connection. If set *False*, the returned connection may be shared across multiple threads. When using multiple threads with the same connection writing operations should be serialized by the user to avoid data corruption.

デフォルトでは、*sqlite3* モジュールは connect の呼び出しの際にモジュールの *Connection* クラスを使います。しかし、*Connection* クラスを継承したクラスを *factory* パラメータに渡して *connect()* にそのクラスを使わせることもできます。

詳しくはこのマニュアルの *SQLite と Python の型* 節を参考にしてください。

sqlite3 モジュールは SQL 解析のオーバーヘッドを避けるために内部で文キャッシュを使っています。接続に対してキャッシュされる文の数を自分で指定したいならば、*cached_statements* パラメータに設定してください。現在の実装ではデフォルトでキャッシュされる SQL 文の数を 100 にしています。

uri が真の場合、*database* は URI として解釈されます。これにより、オプションを指定することができます。例えば、データベースを読み出し専用モードで使用できるように開くには、次のようにします:

```
db = sqlite3.connect('file:path/to/database?mode=ro', uri=True)
```

認識されるオプションのリストを含むこの機能についての詳細については、'SQLite URI documentation '_ を参照してください。

引数 `database` を指定して **監査イベント** `sqlite3.connect` を送出します。

バージョン 3.4 で変更: `uri` パラメータが追加されました。

バージョン 3.7 で変更: *database* は、文字列だけでなく、*path-like object* にすることもできるようになりました。

sqlite3.**register_converter**(*typename*, *callable*)

Registers a callable to convert a bytestring from the database into a custom Python type. The callable will be invoked for all database values that are of the type *typename*. Confer the parameter *detect_types* of the *connect()* function for how the type detection works. Note that *typename* and the name of the type in your query are matched in case-insensitive manner.

sqlite3.**register_adapter**(*type*, *callable*)

自分が使いたい Python の型 *type* を SQLite がサポートしている型に変換する呼び出し可能オブジェクト (callable) を登録します。その呼び出し可能オブジェクト *callable* はただ一つの引数に Python の値を受け取り、int, float, str または bytes のいずれかの型の値を返さなければなりません。

sqlite3.**complete_statement**(*sql*)

文字列 *sql* がセミコロンで終端された一つ以上の完全な SQL 文を含んでいる場合、*True* を返します。判定は SQL 文として文法的に正しいかではなく、閉じられていない文字列リテラルが無いことおよびセミコロンで終端されていることだけで行われます。

この関数は以下の例にあるような SQLite のシェルを作る際に使われます:

```
# A minimal SQLite shell for experiments

import sqlite3

con = sqlite3.connect(":memory:")
con.isolation_level = None
cur = con.cursor()

buffer = ""

print("Enter your SQL commands to execute in sqlite3.")
print("Enter a blank line to exit.")

while True:
    line = input()
    if line == "":
        break
    buffer += line
    if sqlite3.complete_statement(buffer):
        try:
            buffer = buffer.strip()
            cur.execute(buffer)

            if buffer.lstrip().upper().startswith("SELECT"):
                print(cur.fetchall())
        except sqlite3.Error as e:
            print("An error occurred:", e.args[0])
        buffer = ""

con.close()
```

sqlite3.**enable_callback_tracebacks**(*flag*)

デフォルトでは、ユーザ定義の関数、集計関数、変換関数、認可コールバックなどはトレースバック

を出力しません。デバッグの際にはこの関数を *flag* に `True` を指定して呼び出します。そうした後は先に述べたような関数のトレースバックが `sys.stderr` に出力されます。元に戻すには *False* を使います。

12.6.2 Connection オブジェクト

class sqlite3.Connection

SQLite データベースコネクション。以下の属性やメソッドを持ちます:

isolation_level

現在のデフォルト分離レベルを取得または設定します。*None* で自動コミットモードまたは "DEFERRED", "IMMEDIATE", "EXLUSIVE" のどれかです。より詳しい説明は **トランザクション制御** 節を参照してください。

in_transaction

トランザクションがアクティブなら (未コミットの変更があるなら) *True*、そうでなければ *False*。リードオンリー属性です。

バージョン 3.2 で追加.

cursor(*factory=Cursor*)

cursor メソッドはオション引数 *factory* を 1 つだけ受け付けます。渡された場合は、*Cursor* またはそのサブクラスのインスタンスを返す呼び出し可能オブジェクトでなければなりません。

commit()

このメソッドは現在のトランザクションをコミットします。このメソッドを呼ばないと、前回 `commit()` を呼び出してから行ったすべての変更は、他のデータベースコネクションから見ることができません。もし、データベースに書き込んだはずのデータが見えなくて悩んでいる場合は、このメソッドの呼び出しを忘れていないかチェックしてください。

rollback()

このメソッドは最後に行った *commit()* 後の全ての変更をロールバックします。

close()

このメソッドはデータベースコネクションを閉じます。このメソッドが自動的に *commit()* を呼び出さないことに注意してください。*commit()* をせずにコネクションを閉じると、変更が消えてしまいます！

execute(*sql*[, *parameters*])

This is a nonstandard shortcut that creates a cursor object by calling the *cursor()* method, calls the cursor's *execute()* method with the *parameters* given, and returns the cursor.

executemany(*sql*[, *parameters*])

This is a nonstandard shortcut that creates a cursor object by calling the *cursor()* method, calls the cursor's *executemany()* method with the *parameters* given, and returns the cursor.

executescript(*sql_script*)

This is a nonstandard shortcut that creates a cursor object by calling the *cursor()* method,

calls the cursor's *executescript()* method with the given *sql_script*, and returns the cursor.

create_function(*name*, *num_params*, *func*, ***, *deterministic=False*)

Creates a user-defined function that you can later use from within SQL statements under the function name *name*. *num_params* is the number of parameters the function accepts (if *num_params* is -1, the function may take any number of arguments), and *func* is a Python callable that is called as the SQL function. If *deterministic* is true, the created function is marked as deterministic, which allows SQLite to perform additional optimizations. This flag is supported by SQLite 3.8.3 or higher, *NotSupportedError* will be raised if used with older versions.

関数は SQLite でサポートされている任意の型を返すことができます。具体的には bytes, str, int, float および None です。

バージョン 3.8 で変更: *deterministic* 引数が追加されました。

例:

```
import sqlite3
import hashlib

def md5sum(t):
    return hashlib.md5(t).hexdigest()

con = sqlite3.connect(":memory:")
con.create_function("md5", 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", (b"foo",))
print(cur.fetchone()[0])

con.close()
```

create_aggregate(*name*, *num_params*, *aggregate_class*)

ユーザ定義の集計関数を作成します。

The aggregate class must implement a **step** method, which accepts the number of parameters *num_params* (if *num_params* is -1, the function may take any number of arguments), and a **finalize** method which will return the final result of the aggregate.

finalize メソッドは SQLite でサポートされている任意の型を返すことができます。具体的には bytes, str, int, float および None です。

例:

```
import sqlite3

class MySum:
    def __init__(self):
        self.count = 0

    def step(self, value):
```

(次のページに続く)

(前のページからの続き)

```
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
cur = con.cursor()
cur.execute("create table test(i)")
cur.execute("insert into test(i) values (1)")
cur.execute("insert into test(i) values (2)")
cur.execute("select mysum(i) from test")
print(cur.fetchone()[0])

con.close()
```

create_collation(*name*, *callable*)

name と *callable* で指定される照合順序を作成します。呼び出し可能オブジェクトには二つの文字列が渡されます。一つめのものが二つめのものより低く順序付けられるならば -1 を返し、等しければ 0 を返し、一つめのものが二つめのものより高く順序付けられるならば 1 を返すようにしなければなりません。この関数はソート (SQL での ORDER BY) をコントロールするもので、比較を行なうことは他の SQL 操作には影響を与えないことに注意しましょう。

また、呼び出し可能オブジェクトに渡される引数は Python のバイト文字列として渡されますが、それは通常 UTF-8 で符号化されたものになります。

以下の例は「間違った方法で」ソートする自作の照合順序です:

```
import sqlite3

def collate_reverse(string1, string2):
    if string1 == string2:
        return 0
    elif string1 < string2:
        return 1
    else:
        return -1

con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)

cur = con.cursor()
cur.execute("create table test(x)")
cur.executemany("insert into test(x) values (?)", [("a",), ("b",)])
cur.execute("select x from test order by x collate reverse")
for row in cur:
    print(row)
con.close()
```

照合順序を取り除くには callable に None を指定して create_collation を呼び出します:

```
con.create_collation("reverse", None)
```

interrupt()

このメソッドを別スレッドから呼び出して接続上で現在実行中であろうクエリを中断させられます。クエリが中断されると呼び出し元は例外を受け取ります。

set_authorizer(*authorizer_callback*)

このルーチンはコールバックを登録します。コールバックはデータベースのテーブルのカラムにアクセスしようとするたびに呼び出されます。コールバックはアクセスが許可されるならば SQLITE_OK を、SQL 文全体がエラーとともに中断されるべきならば SQLITE_DENY を、カラムが NULL 値として扱われるべきなら SQLITE_IGNORE を返さなければなりません。これらの定数は *sqlite3* モジュールに用意されています。

コールバックの第一引数はどの種類の操作が許可されるかを決めます。第二第三引数には第一引数に依存して本当に使われる引数か *None* かが渡されます。第四引数はもし適用されるならばデータベースの名前 ("main", "temp", etc.) です。第五引数はアクセスを試みる要因となった最も内側のトリガまたはビューの名前、またはアクセスの試みが入力された SQL コードに直接起因するものならば *None* です。

第一引数に与えることができる値や、その第一引数によって決まる第二第三引数の意味については、SQLite の文書を参考にしてください。必要な定数は全て *sqlite3* モジュールに用意されています。

set_progress_handler(*handler*, *n*)

このメソッドはコールバックを登録します。コールバックは SQLite 仮想マシン上の *n* 個の命令を実行するごとに呼び出されます。これは、GUI 更新などのために、長時間かかる処理中に SQLite からの呼び出しが欲しい場合に便利です。

以前登録した progress handler をクリアしたい場合は、このメソッドを、*handler* 引数に *None* を渡して呼び出してください。

ハンドラー関数からゼロ以外の値を返すと、現在実行中のクエリが終了 (terminate) し、*OperationalError* 例外を送出します。

set_trace_callback(*trace_callback*)

各 SQL 文が SQLite バックエンドによって実際に実行されるたびに呼び出される *trace_callback* を登録します。

コールバックに渡される唯一の引数は、実行されている SQL 文 (の文字列) です。コールバックの戻り値は無視されます。バックエンドは *Cursor.execute()* メソッドに渡された SQL 文だけを実行するわけではないことに注意してください。他のソースには、Python モジュールのトランザクション管理や、現在のデータベースに定義されたトリガーの実行が含まれます。

trace_callback として *None* を渡すと、トレースコールバックを無効にできます。

バージョン 3.3 で追加.

enable_load_extension(*enabled*)

このメソッドは SQLite エンジンが共有ライブラリから SQLite 拡張を読み込むのを許可したり、

禁止したりします。SQLite 拡張は新しい関数や集計関数や仮想テーブルの実装を定義できます。1 つの有名な拡張は SQLite によって頒布されている全テキスト検索拡張です。

SQLite 拡張はデフォルトで無効にされています。[*1] を見てください。

バージョン 3.2 で追加.

```
import sqlite3

con = sqlite3.connect(":memory:")

# enable extension loading
con.enable_load_extension(True)

# Load the fulltext search extension
con.execute("select load_extension('./fts3.so')")

# alternatively you can load the extension using an API call:
# con.load_extension("./fts3.so")

# disable extension loading again
con.enable_load_extension(False)

# example from SQLite wiki
con.execute("create virtual table recipe using fts3(name, ingredients)")
con.executescript("""
    insert into recipe (name, ingredients) values ('broccoli stew', 'broccoli peppers␣
↪cheese tomatoes');
    insert into recipe (name, ingredients) values ('pumpkin stew', 'pumpkin onions␣
↪garlic celery');
    insert into recipe (name, ingredients) values ('broccoli pie', 'broccoli cheese␣
↪onions flour');
    insert into recipe (name, ingredients) values ('pumpkin pie', 'pumpkin sugar␣
↪flour butter');
    """)
for row in con.execute("select rowid, name, ingredients from recipe where name match
↪'pie'"):
    print(row)

con.close()
```

load_extension(*path*)

このメソッドは共有ライブラリから SQLite 拡張を読み込みます。このメソッドを使う前に *enable_load_extension()* で拡張の読み込みを許可しておかなくてはなりません。

SQLite 拡張はデフォルトで無効にされています。[*1] を見てください。

バージョン 3.2 で追加.

[*1] いくつかのプラットフォームでこの機能なしでコンパイルされる SQLite ライブラリがあるので (特に Mac OS X)、sqlite3 モジュールはデフォルトで SQLite 拡張サポートなしで構築されます。SQLite 拡張サポートを有効にするには、configure に `--enable-loadable-sqlite-extensions` を渡す必要があります。

row_factory
この属性を変更して、カーソルと元の行をタプル形式で受け取り、本当の結果の行を返す呼び出し可能オブジェクトにすることができます。これによって、より進んだ結果の返し方を実装することができます。例えば、各列に列名でもアクセスできるようなオブジェクトを返すことができます。

例:

```
import sqlite3

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d

con = sqlite3.connect(":memory:")
con.row_factory = dict_factory
cur = con.cursor()
cur.execute("select 1 as a")
print(cur.fetchone()["a"])

con.close()
```

タプルを返すのでは物足りず、名前に基づいて列へアクセスしたい場合は、*row_factory* に高度に最適化された *sqlite3.Row* 型を設定することを検討してください。*Row* クラスではインデックスでも大文字小文字を無視した名前でも列にアクセスでき、しかもほとんどメモリーを浪費しません。おそらく独自実装の辞書を使うアプローチよりも良いもので、もしかすると db の行に基づいた解法よりも優れているかもしれません。

text_factory
この属性を使って TEXT データ型をどのオブジェクトで返すかを制御できます。デフォルトではこの属性は *str* に設定されており、*sqlite3* モジュールは TEXT を Unicode オブジェクトで返します。もしバイト列で返したいならば、*bytes* に設定してください。

バイト列を受け取って望みの型のオブジェクトを返すような呼び出し可能オブジェクトを何でも設定して構いません。

以下の説明用のコード例を参照してください:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()

AUSTRIA = "\xd6sterreich"

# by default, rows are returned as Unicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert row[0] == AUSTRIA
```

(次のページに続く)

(前のページからの続き)

```
# but we can make sqlite3 always return bytestrings ...
con.text_factory = bytes
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is bytes
# the bytestrings will be encoded in UTF-8, unless you stored garbage in the
# database ...
assert row[0] == AUSTRIA.encode("utf-8")

# we can also implement a custom text_factory ...
# here we implement one that appends "foo" to all strings
con.text_factory = lambda x: x.decode("utf-8") + "foo"
cur.execute("select ?", ("bar",))
row = cur.fetchone()
assert row[0] == "barfoo"

con.close()
```

total_changes

データベース接続が開始されて以来の行の変更・挿入・削除がなされた行の総数を返します。

iterdump()

データベースを SQL test フォーマットでダンプするためのイテレータを返します。メモリ内のデータベースの内容を、後で復元するために保存する場合に便利です。この関数には、**sqlite3** シェルの中の `.dump` コマンドと同じ機能があります。

以下はプログラム例です:

```
# Convert file existing_db.db to SQL dump file dump.sql
import sqlite3

con = sqlite3.connect('existing_db.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
con.close()
```

backup(*target*, ***, *pages=-1*, *progress=None*, *name="main"*, *sleep=0.250*)

This method makes a backup of a SQLite database even while it's being accessed by other clients, or concurrently by the same connection. The copy will be written into the mandatory argument *target*, that must be another *Connection* instance.

By default, or when *pages* is either `0` or a negative integer, the entire database is copied in a single step; otherwise the method performs a loop copying up to *pages* pages at a time.

If *progress* is specified, it must either be `None` or a callable object that will be executed at each iteration with three integer arguments, respectively the *status* of the last iteration, the *remaining* number of pages still to be copied and the *total* number of pages.

The *name* argument specifies the database name that will be copied: it must be a string containing either `"main"`, the default, to indicate the main database, `"temp"` to indicate the temporary database or the name specified after the `AS` keyword in an `ATTACH DATABASE` statement for an attached database.

The *sleep* argument specifies the number of seconds to sleep by between successive attempts to backup remaining pages, can be specified either as an integer or a floating point value.

Example 1, copy an existing database into another:

```
import sqlite3

def progress(status, remaining, total):
    print(f'Copied {total-remaining} of {total} pages...')

con = sqlite3.connect('existing_db.db')
bck = sqlite3.connect('backup.db')
with bck:
    con.backup(bck, pages=1, progress=progress)
bck.close()
con.close()
```

Example 2, copy an existing database into a transient copy:

```
import sqlite3

source = sqlite3.connect('existing_db.db')
dest = sqlite3.connect(':memory:')
source.backup(dest)
```

Availability: SQLite 3.6.11 or higher

バージョン 3.7 で追加.

12.6.3 カーソルオブジェクト

class sqlite3.Cursor

Cursor インスタンスは以下の属性やメソッドを持ちます。

execute(*sql*[, *parameters*])

SQL 文を実行します。SQL 文はパラメータ化できます (すなわち SQL リテラルの代わりの場所確保文字 (placeholder) を入れておけます)。*sqlite3* モジュールは 2 種類の場所確保記法をサポートします。一つは疑問符 (qmark スタイル)、もう一つは名前 (named スタイル) です。

両方のスタイルの例です:

```
import sqlite3

con = sqlite3.connect(":memory:")
```

(次のページに続く)

(前のページからの続き)

```
cur = con.cursor()
cur.execute("create table people (name_last, age)")

who = "Yeltsin"
age = 72

# This is the qmark style:
cur.execute("insert into people values (?, ?)", (who, age))

# And this is the named style:
cur.execute("select * from people where name_last=:who and age=:age", {"who": who,
↪"age": age})

print(cur.fetchone())

con.close()
```

execute() は一つの SQL 文しか実行しません。二つ以上の文を実行しようとすると、*Warning* を送出します。複数の SQL 文を一つの呼び出しで実行したい場合は *executescript()* を使ってください。

executemany(*sql*, *seq_of_parameters*)

Executes an SQL command against all parameter sequences or mappings found in the sequence *seq_of_parameters*. The *sqlite3* module also allows using an *iterator* yielding parameters instead of a sequence.

```
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')

    def __iter__(self):
        return self

    def __next__(self):
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),) # this is a 1-tuple

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)

cur.execute("select c from characters")
print(cur.fetchall())
```

(次のページに続く)

(前のページからの続き)

```
con.close()
```

もう少し短い ジェネレータ を使った例です:

```
import sqlite3
import string

def char_generator():
    for c in string.ascii_lowercase:
        yield (c,)

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

cur.executemany("insert into characters(c) values (?)", char_generator())

cur.execute("select c from characters")
print(cur.fetchall())

con.close()
```

executescript(*sql_script*)

これは非標準の便宜メソッドで、一度に複数の SQL 文を実行することができます。メソッドは最初に COMMIT 文を発行し、次いで引数として渡された SQL スクリプトを実行します。

sql_script は *str* のインスタンスです。

例:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript("""
    create table person(
        firstname,
        lastname,
        age
    );

    create table book(
        title,
        author,
        published
    );

    insert into book(title, author, published)
    values (
```

(次のページに続く)

(前のページからの続き)

```
        'Dirk Gently''s Holistic Detective Agency',
        'Douglas Adams',
        1987
    );
    """)
con.close()
```

fetchone()

クエリ結果から次の row をフェッチして、1 つのシーケンスを返します。これ以上データがない場合は *None* を返します。

fetchmany(*size=cursor.arraysize*)

クエリ結果から次の幾つかの row をフェッチして、リストを返します。これ以上データがない場合は空のリストを返します。

一回の呼び出しで返される row の数は、*size* 引数で指定できます。この引数が与えられない場合、cursor の arraysize 属性が利用されます。このメソッドは可能な限り指定された *size* の数の row を fetch しようとするべきです。もし、指定された数の row が利用可能でない場合、それより少ない数の row が返されます。

size 引数とパフォーマンスの関係についての注意です。パフォーマンスを最適化するためには、大抵、arraysize 属性を利用するのがベストです。*size* 引数を利用したのであれば、次の *fetchmany()* の呼び出しでも同じ数を利用するのがベストです。

fetchall()

全ての (残りの) クエリ結果の row をフェッチして、リストを返します。cursor の arraysize 属性がこの操作のパフォーマンスに影響することに気をつけてください。これ以上の row がない場合は、空のリストが返されます。

close()

(`__del__` が呼び出される時ではなく、) 今すぐカーソルを閉じます。

この時点から、このカーソルは使用できなくなります。今後、このカーソルで何らかの操作を試みると、*ProgrammingError* 例外が送出されます。

rowcount

一応 *sqlite3* モジュールの *Cursor* クラスはこの属性を実装していますが、データベースエンジン自身の「影響を受けた行」/「選択された行」の決定方法は少し風変わりです。

executemany() では、変更数が *rowcount* に合計されます。

Python DB API 仕様で要求されるように、*rowcount* 属性は「カーソルに対して `executeXX()` が行なわれていないか、最後の操作の rowcount がインターフェースによって決定できなかった場合は -1 」です。これには `SELECT` 文も含まれます。すべての列を取得するまでクエリによって生じた列の数を決定できないからです。

SQLite のバージョン 3.6.5 以前は、条件なしで `DELETE FROM table` を実行すると *rowcount* が 0 にセットされます。

lastrowid

この読み込み専用の属性は、最後に変更した row の rowid を提供します。この属性は、*execute()* メソッドを利用して INSERT 文または REPLACE 文を実行したときのみ設定されます。INSERT と REPLACE 以外の操作や、*executemany()* メソッドを呼び出した場合は、*lastrowid* は *None* に設定されます。

If the INSERT or REPLACE statement failed to insert the previous successful rowid is returned.

バージョン 3.6 で変更: REPLACE 文のサポートが追加されました。

arraysize

fetchmany によって返される行 (row) 数を制御する、読み取りと書き込みが可能な属性。デフォルト値は 1 で、これは呼び出しごとに 1 行が取得されることを意味します。

description

この読み出し専用の属性は、最後のクエリの結果のカラム名を提供します。Python DB API との互換性を維持するために、各カラムに対して 7 つのタプルを返しますが、タプルの後ろ 6 つの要素は全て *None* です。

この属性は SELECT 文にマッチする row が 1 つもなかった場合でもセットされます。

connection

この読み出し専用の属性は、*Cursor* オブジェクトが使用する SQLite データベースの *Connection* を提供します。*con.cursor()* を呼び出すことにより作成される *Cursor* オブジェクトは、*con* を参照する *connection* 属性を持ちます:

```
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
```

12.6.4 Row オブジェクト

class sqlite3.Row

Row インスタンスは、*Connection* オブジェクトの *row_factory* として高度に最適化されています。タプルによく似た機能を持つ row を作成します。

カラム名とインデックスによる要素へのアクセス, イテレーション, repr(), 同値テスト, *len()* をサポートしています。

もし、2 つの *Row* オブジェクトが完全に同じカラムと値を持っていた場合、それらは同値になります。

keys()

このメソッドはカラム名のリストを返します。クエリ直後から、これは *Cursor.description* の各タプルの最初のメンバになります。

バージョン 3.5 で変更: スライスがサポートされました。

Row の例のために、まずサンプルのテーブルを初期化します:

```
con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute('''create table stocks
(date text, trans text, symbol text,
 qty real, price real)''')
cur.execute("""insert into stocks
            values ('2006-01-05','BUY','RHAT',100,35.14)""")
con.commit()
cur.close()
```

そして、*Row* を使ってみます:

```
>>> con.row_factory = sqlite3.Row
>>> cur = con.cursor()
>>> cur.execute('select * from stocks')
<sqlite3.Cursor object at 0x7f4e7dd8fa80>
>>> r = cur.fetchone()
>>> type(r)
<class 'sqlite3.Row'>
>>> tuple(r)
('2006-01-05', 'BUY', 'RHAT', 100.0, 35.14)
>>> len(r)
5
>>> r[2]
'RHAT'
>>> r.keys()
['date', 'trans', 'symbol', 'qty', 'price']
>>> r['qty']
100.0
>>> for member in r:
...     print(member)
...
2006-01-05
BUY
RHAT
100.0
35.14
```

12.6.5 例外

exception sqlite3.Warning

Exception のサブクラスです。

exception sqlite3.Error

このモジュールにおける他の例外クラスの基底クラスです。*Exception* のサブクラスです。

exception sqlite3.DatabaseError

Exception raised for errors that are related to the database.

exception sqlite3.IntegrityError

データベースの参照整合性が影響を受ける場合に発生する例外。たとえば外部キーのチェック (foreign key check) が失敗したとき。*DatabaseError* のサブクラスです。

exception sqlite3.ProgrammingError

Exception raised for programming errors, e.g. table not found or already exists, syntax error in the SQL statement, wrong number of parameters specified, etc. It is a subclass of *DatabaseError*.

exception sqlite3.OperationalError

Exception raised for errors that are related to the database's operation and not necessarily under the control of the programmer, e.g. an unexpected disconnect occurs, the data source name is not found, a transaction could not be processed, etc. It is a subclass of *DatabaseError*.

exception sqlite3.NotSupportedError

Exception raised in case a method or database API was used which is not supported by the database, e.g. calling the *rollback()* method on a connection that does not support transaction or has transactions turned off. It is a subclass of *DatabaseError*.

12.6.6 SQLite と Python の型

はじめに

SQLite は以下の型をネイティブにサポートします: NULL, INTEGER, REAL, TEXT, BLOB。

したがって、次の Python の型は問題なく SQLite に送り込めます:

Python の型	SQLite の型
None	NULL
int	INTEGER
float	REAL
str	TEXT
bytes	BLOB

SQLite の型から Python の型へのデフォルトでの変換は以下の通りです:

SQLite の型	Python の型
NULL	*None*
INTEGER	*int*
REAL	*float*
TEXT	*text_factory* に依存する。デフォルトでは *str* 。
BLOB	*bytes*

sqlite3 モジュールの型システムは二つの方法で拡張できます。一つはオブジェクト適合 (adaptation) を通じて追加された Python の型を SQLite に格納することです。もう一つは変換関数 (converter) を通じて *sqlite3* モジュールに SQLite の型を違った Python の型に変換させることです。

追加された Python の型を SQLite データベースに格納するために適合関数を使う

既に述べたように、SQLite が最初からサポートする型は限られたものだけです。それ以外の Python の型を SQLite で使うには、その型を *sqlite3* モジュールがサポートしている型の一つに **適合** させなくてはなりません。サポートしている型というのは、NoneType, int, float, str, bytes です。

sqlite3 モジュールで望みの Python の型をサポートされている型の一つに適合させる方法は二つあります。

オブジェクト自身で適合するようにする

自分でクラスを書いているならばこの方法が良いでしょう。次のようなクラスがあるとします:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
```

さてこの点を SQLite の一つのカラムに収めたいと考えたとしましょう。最初にしなければならないのはサポートされている型の中から点を表現するのに使えるものを選ぶことです。ここでは単純に文字列を使うことにして、座標を区切るのにはセミコロンを使いましょう。次に必要なのはクラスに変換された値を返す `__conform__(self, protocol)` メソッドを追加することです。引数 *protocol* は `PrepareProtocol` になります。

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return "%f;%f" % (self.x, self.y)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])

con.close()
```

適合関数を登録する

もう一つの可能性は型を文字列表現に変換する関数を作り *register_adapter()* でその関数を登録することです。

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)

sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])

con.close()
```

sqlite3 モジュールには二つの Python 標準型 *datetime.date* と *datetime.datetime* に対するデフォルト適合関数があります。いま *datetime.datetime* オブジェクトを ISO 表現でなく Unix タイムスタンプとして格納したいとしましょう。

```
import sqlite3
import datetime
import time

def adapt_datetime(ts):
    return time.mktime(ts.timetuple())

sqlite3.register_adapter(datetime.datetime, adapt_datetime)

con = sqlite3.connect(":memory:")
cur = con.cursor()

now = datetime.datetime.now()
cur.execute("select ?", (now,))
print(cur.fetchone()[0])

con.close()
```

SQLite の値を好きな Python 型に変換する

適合関数を書くことで好きな Python 型を SQLite に送り込めるようになりました。しかし、本当に使い物になるようにするには Python から SQLite さらに Python へという往還 (roundtrip) の変換ができる必要があります。

そこで変換関数 (converter) です。

`Point` クラスの例に戻りましょう。x, y 座標をセミコロンで区切った文字列として SQLite に格納したのでした。

まず、文字列を引数として取り `Point` オブジェクトをそれから構築する変換関数を定義します。

注釈: 変換関数は SQLite に送り込んだデータ型に関係なく **常に** *bytes* オブジェクトを渡されます。

```
def convert_point(s):
    x, y = map(float, s.split(b";"))
    return Point(x, y)
```

次に *sqlite3* モジュールにデータベースから取得したものが本当に点であることを教えなければなりません。二つの方法があります:

- 宣言された型を通じて暗黙的に
- カラム名を通じて明示的に

どちらの方法も **モジュールの関数と定数** 節の中で説明されています。それぞれ *PARSE_DECLTYPES* 定数と *PARSE_COLNAMES* 定数の項目です。

以下の例で両方のアプローチを紹介します。

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "(%f;%f)" % (self.x, self.y)

def adapt_point(point):
    return ("%f;%f" % (point.x, point.y)).encode('ascii')

def convert_point(s):
    x, y = list(map(float, s.split(b";")))
    return Point(x, y)

# Register the adapter
sqlite3.register_adapter(Point, adapt_point)
```

(次のページに続く)

(前のページからの続き)

```
# Register the converter
sqlite3.register_converter("point", convert_point)

p = Point(4.0, -3.2)

########################
# 1) Using declared types
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(p point)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute("select p from test")
print("with declared types:", cur.fetchone()[0])
cur.close()
con.close()

######################
# 1) Using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(p)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute('select p as "p [point]" from test')
print("with column names:", cur.fetchone()[0])
cur.close()
con.close()
```

デフォルトの適合関数と変換関数

datetime モジュールの date 型および datetime 型のためのデフォルト適合関数があります。これらの型は ISO 日付 / ISO タイムスタンプとして SQLite に送られます。

デフォルトの変換関数は *datetime.date* 用が "date" という名前で、*datetime.datetime* 用が "timestamp" という名前で登録されています。

これにより、多くの場合特別な細工無しに Python の日付 / タイムスタンプを使えます。適合関数の書式は実験的な SQLite の date/time 関数とも互換性があります。

以下の例でこのことを確かめます。

```
import sqlite3
import datetime

con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(d date, ts timestamp)")

today = datetime.date.today()
```

(次のページに続く)

(前のページからの続き)

```
now = datetime.datetime.now()

cur.execute("insert into test(d, ts) values (?, ?)", (today, now))
cur.execute("select d, ts from test")
row = cur.fetchone()
print(today, "=>", row[0], type(row[0]))
print(now, "=>", row[1], type(row[1]))

cur.execute('select current_date as "d [date]", current_timestamp as "ts [timestamp]"')
row = cur.fetchone()
print("current_date", row[0], type(row[0]))
print("current_timestamp", row[1], type(row[1]))

con.close()
```

SQLite に格納されているタイムスタンプが 6 桁より長い小数部を持っている場合、タイムスタンプの変換関数によってマイクロ秒精度に丸められます。

12.6.7 トランザクション制御

The underlying `sqlite3` library operates in `autocommit` mode by default, but the Python *sqlite3* module by default does not.

`autocommit` mode means that statements that modify the database take effect immediately. A `BEGIN` or `SAVEPOINT` statement disables `autocommit` mode, and a `COMMIT`, a `ROLLBACK`, or a `RELEASE` that ends the outermost transaction, turns `autocommit` mode back on.

The Python *sqlite3* module by default issues a `BEGIN` statement implicitly before a Data Modification Language (DML) statement (i.e. `INSERT`/`UPDATE`/`DELETE`/`REPLACE`).

You can control which kind of `BEGIN` statements *sqlite3* implicitly executes via the *isolation_level* parameter to the *connect()* call, or via the `isolation_level` property of connections. If you specify no *isolation_level*, a plain `BEGIN` is used, which is equivalent to specifying `DEFERRED`. Other possible values are `IMMEDIATE` and `EXCLUSIVE`.

You can disable the *sqlite3* module's implicit transaction management by setting `isolation_level` to `None`. This will leave the underlying `sqlite3` library operating in `autocommit` mode. You can then completely control the transaction state by explicitly issuing `BEGIN`, `ROLLBACK`, `SAVEPOINT`, and `RELEASE` statements in your code.

バージョン 3.6 で変更: *sqlite3* used to implicitly commit an open transaction before DDL statements. This is no longer the case.

12.6.8 sqlite3 の効率的な使い方

ショートカットメソッドを使う

Connection オブジェクトの非標準的なメソッド execute(), executemany(), executescript() を使うことで、(しばしば余計な) *Cursor* オブジェクトをわざわざ作り出さずに済むので、コードをより簡潔に書くことができます。*Cursor* オブジェクトは暗黙裡に生成されショートカットメソッドの戻り値として受け取ることができます。この方法を使えば、SELECT 文を実行してその結果について反復することが、*Connection* オブジェクトに対する呼び出し一つで行なえます。

```
import sqlite3

persons = [
    ("Hugo", "Boss"),
    ("Calvin", "Klein")
    ]

con = sqlite3.connect(":memory:")

# Create the table
con.execute("create table person(firstname, lastname)")

# Fill the table
con.executemany("insert into person(firstname, lastname) values (?, ?)", persons)

# Print the table contents
for row in con.execute("select firstname, lastname from person"):
    print(row)

print("I just deleted", con.execute("delete from person").rowcount, "rows")

# close is not a shortcut method and it's not called automatically,
# so the connection object should be closed manually
con.close()
```

位置ではなく名前でカラムにアクセスする

sqlite3 モジュールの有用な機能の一つに、行生成関数として使われるための *sqlite3.Row* クラスがあります。

このクラスでラップされた行は、位置インデクス (タプルのような) でも大文字小文字を区別しない名前でもアクセスできます:

```
import sqlite3

con = sqlite3.connect(":memory:")
con.row_factory = sqlite3.Row

cur = con.cursor()
cur.execute("select 'John' as name, 42 as age")
```

(次のページに続く)

(前のページからの続き)

```
for row in cur:
    assert row[0] == row["name"]
    assert row["name"] == row["nAmE"]
    assert row[1] == row["age"]
    assert row[1] == row["AgE"]

con.close()
```

コネクションをコンテキストマネージャーとして利用する

Connection オブジェクトはコンテキストマネージャーとして利用して、トランザクションを自動的にコミットしたりロールバックすることができます。例外が発生したときにトランザクションはロールバックされ、それ以外の場合、トランザクションはコミットされます:

```
import sqlite3

con = sqlite3.connect(":memory:")
con.execute("create table person (id integer primary key, firstname varchar unique)")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("insert into person(firstname) values (?)", ("Joe",))

# con.rollback() is called after the with block finishes with an exception, the
# exception is still raised and must be caught
try:
    with con:
        con.execute("insert into person(firstname) values (?)", ("Joe",))
except sqlite3.IntegrityError:
    print("couldn't add Joe twice")

# Connection object used as context manager only commits or rollbacks transactions,
# so the connection object should be closed manually
con.close()
```

脚注

第

THIRTEEN

データ圧縮とアーカイブ

この章で説明されるモジュールは zlib, gzip, bzip2, lzma アルゴリズムによるデータの圧縮と、ZIP, tar フォーマットのアーカイブ作成をサポートします。*shutil* モジュールで提供される **アーカイブ化操作** も参照してください。

13.1 zlib --- gzip 互換の圧縮

このモジュールは、データ圧縮を必要とするアプリケーションが zlib ライブラリを使って圧縮および展開を行えるようにします。zlib ライブラリ自身の Web ページは http://www.zlib.net です。Python モジュールと zlib ライブラリの 1.1.3 より前のバージョンには互換性のない部分があることが知られています。1.1.3 にはセキュリティホールが存在するため、1.1.4 以降のバージョンを利用することを推奨します。

zlib の関数にはたくさんのオプションがあり、場合によっては特定の順番で使わなければなりません。このドキュメントではそれら順番についてすべてを説明しようとはしていません。詳細は公式サイト http://www.zlib.net/manual.html にある zlib のマニュアルを参照してください。

`.gz` ファイルの読み書きのためには、*gzip* モジュールを参照してください。

このモジュールで利用可能な例外と関数を以下に示します:

exception zlib.error
: 圧縮および展開時のエラーによって送出される例外です。

zlib.adler32(*data*[, *value*])
: *data* の Adler-32 チェックサムを計算します (Adler-32 チェックサムは、おおむね CRC32 と同等の信頼性を持ちながら、はるかに高速に計算できます)。結果は、符号のない 32 ビットの整数です。*value* が与えられている場合、チェックサム計算の初期値として使われます。与えられていない場合、デフォルト値の 1 が使われます。*value* を与えることで、複数の入力を結合したデータ全体にわたり、通しのチェックサムを計算できます。このアルゴリズムは暗号論的には強力ではなく、認証やデジタル署名などに用いるべきではありません。また、チェックサムアルゴリズムとして設計されているため、汎用のハッシュアルゴリズムには向きません。

  バージョン 3.0 で変更: 常に符号のない値を返します。すべてのバージョンとプラットフォームの Python に渡って同一の数値を生成するには、`adler32(data) & 0xffffffff` を使用します。

zlib.compress(*data*, *level=-1*)

Compresses the bytes in *data*, returning a bytes object containing compressed data. *level* is an integer from 0 to 9 or -1 controlling the level of compression; 1 (Z_BEST_SPEED) is fastest and produces the least compression, 9 (Z_BEST_COMPRESSION) is slowest and produces the most. 0 (Z_NO_COMPRESSION) is no compression. The default value is -1 (Z_DEFAULT_COMPRESSION). Z_DEFAULT_COMPRESSION represents a default compromise between speed and compression (currently equivalent to level 6). Raises the *error* exception if any error occurs.

バージョン 3.6 で変更: *level* can now be used as a keyword parameter.

zlib.compressobj(*level=-1*, *method=DEFLATED*, *wbits=MAX_WBITS*, *memLevel=DEF_MEM_LEVEL*, *strategy=Z_DEFAULT_STRATEGY*[, *zdict*])

一度にメモリ上に置くことができないようなデータストリームを圧縮するための圧縮オブジェクトを返します。

level は圧縮レベルです。0 から 9 、または -1 の整数を取り、1 (Z_BEST_SPEED) は最も高速で最小限の圧縮を行い、9 (Z_BEST_COMPRESSION) は最も低速で最大限の圧縮を行います。0 (Z_NO_COMPRESSION) は圧縮しません。デフォルトは -1 です (Z_DEFAULT_COMPRESSION)。Z_DEFAULT_COMPRESSION は、速度と圧縮の間のデフォルトの妥協点 (現在、レベル 6 に対応します) を表します。

method は圧縮アルゴリズムです。現在、DEFLATED のみサポートされています。

The *wbits* argument controls the size of the history buffer (or the "window size") used when compressing data, and whether a header and trailer is included in the output. It can take several ranges of values, defaulting to 15 (MAX_WBITS):

- +9 to +15: The base-two logarithm of the window size, which therefore ranges between 512 and 32768. Larger values produce better compression at the expense of greater memory usage. The resulting output will include a zlib-specific header and trailer.
- − 9 to − 15: Uses the absolute value of *wbits* as the window size logarithm, while producing a raw output stream with no header or trailing checksum.
- +25 to +31 = 16 + (9 to 15): Uses the low 4 bits of the value as the window size logarithm, while including a basic **gzip** header and trailing checksum in the output.

memLevel 引数は内部圧縮状態用に使用されるメモリ量を制御します。有効な値は 1 から 9 です。大きい値ほど多くのメモリを消費しますが、より速く、より小さな出力を作成します。

strategy は圧縮アルゴリズムの調整に使用されます。指定可能な値は、Z_DEFAULT_STRATEGY, Z_FILTERED, Z_HUFFMAN_ONLY, Z_RLE (zlib 1.2.0.1) および Z_FIXED (zlib 1.2.2.2) です。

zdict は定義済み圧縮辞書です。これは圧縮されるデータ内で繰り返し現れると予想されるサブシーケンスを含む (*bytes* オブジェクトのような) バイト列のシーケンスです。最も一般的と思われるサブシーケンスは辞書の末尾に来なければなりません。

バージョン 3.3 で変更: *zdict* パラメータとキーワード引数のサポートが追加されました。

zlib.**crc32**(*data*[, *value*])

data の CRC (Cyclic Redundancy Check, 巡回冗長検査) チェックサムを計算します。結果は、符号のない 32 ビットの整数です。*value* が与えられている場合、チェックサム計算の初期値として使われます。与えられていない場合、デフォルト値の 1 が使われます。*value* を与えることで、複数の入力を結合したデータ全体にわたり、通しのチェックサムを計算できます。このアルゴリズムは暗号論的には強力ではなく、認証やデジタル署名などに用いるべきではありません。また、チェックサムアルゴリズムとして設計されているため、汎用のハッシュアルゴリズムには向きません。

バージョン 3.0 で変更: 常に符号のない値を返します。すべてのバージョンとプラットフォームの Python に渡って同一の数値を生成するには、`crc32(data) & 0xffffffff` を使用します。

zlib.**decompress**(*data*, *wbits=MAX_WBITS*, *bufsize=DEF_BUF_SIZE*)

Decompresses the bytes in *data*, returning a bytes object containing the uncompressed data. The *wbits* parameter depends on the format of *data*, and is discussed further below. If *bufsize* is given, it is used as the initial size of the output buffer. Raises the *error* exception if any error occurs.

The *wbits* parameter controls the size of the history buffer (or "window size"), and what header and trailer format is expected. It is similar to the parameter for *compressobj()*, but accepts more ranges of values:

- +8 to +15: The base-two logarithm of the window size. The input must include a zlib header and trailer.
- 0: Automatically determine the window size from the zlib header. Only supported since zlib 1.2.3.5.
- −8 to −15: Uses the absolute value of *wbits* as the window size logarithm. The input must be a raw stream with no header or trailer.
- +24 to +31 = 16 + (8 to 15): Uses the low 4 bits of the value as the window size logarithm. The input must include a gzip header and trailer.
- +40 to +47 = 32 + (8 to 15): Uses the low 4 bits of the value as the window size logarithm, and automatically accepts either the zlib or gzip format.

When decompressing a stream, the window size must not be smaller than the size originally used to compress the stream; using a too-small value may result in an *error* exception. The default *wbits* value corresponds to the largest window size and requires a zlib header and trailer to be included.

bufsize は展開されたデータを保持するためのバッファサイズの初期値です。バッファの空きは必要に応じて必要なだけ増加するので、必ずしも正確な値を指定する必要はありません。この値のチューニングでできることは、`malloc()` が呼ばれる回数を数回減らすことぐらいです。

バージョン 3.6 で変更: *wbits* and *bufsize* can be used as keyword arguments.

zlib.**decompressobj**(*wbits=MAX_WBITS*[, *zdict*])

一度にメモリ上に置くことができないようなデータストリームを展開するための展開オブジェクトを返します。

The *wbits* parameter controls the size of the history buffer (or the "window size"), and what header and trailer format is expected. It has the same meaning as *described for decompress()*.

zdict パラメータには定義済み圧縮辞書を指定します。このパラメータを指定する場合、展開するデータを圧縮した際に使用した辞書と同じものでなければなりません。

注釈: *zdict* が (*bytearray* のような) 変更可能オブジェクトの場合、*decompressobj()* の呼び出しとデコンプレッサの `decompress()` メソッドの最初の呼び出しの間に辞書の内容を変更してはいけません。

バージョン 3.3 で変更: パラメータに *zdict* を追加しました。

圧縮オブジェクトは以下のメソッドをサポートしています:

`Compress.compress(`*data*`)`

data を圧縮し、圧縮されたデータを含むバイト列オブジェクトを返します。この文字列は少なくとも *data* の一部分のデータに対する圧縮データを含みます。このデータは以前に呼んだ *compress()* が返した出力と結合することができます。入力の一部は以後の処理のために内部バッファに保存されることもあります。

`Compress.flush(`[*mode*]`)`

未処理の全入力データが処理され、この未処理部分を圧縮したデータを含むバイト列オブジェクトが返されます。*mode* は定数 `Z_NO_FLUSH`、`Z_PARTIAL_FLUSH`、`Z_SYNC_FLUSH`、`Z_FULL_FLUSH`、`Z_BLOCK` (zlib 1.2.3.4)、または `Z_FINISH` のいずれかをとり、デフォルト値は `Z_FINISH` です。`Z_FINISH` を除く全ての定数はこれ以後にもデータバイト文字列を圧縮できるモードです。一方、`Z_FINISH` は圧縮ストリームを閉じ、これ以後のデータの圧縮を停止します。*mode* に `Z_FINISH` を指定して *flush()* メソッドを呼び出した後は、*compress()* メソッドを再び呼ぶべきではありません。唯一の現実的な操作はこのオブジェクトを削除することだけです。

`Compress.copy()`

圧縮オブジェクトのコピーを返します。これを使うと先頭部分が共通している複数のデータを効率的に圧縮することができます。

バージョン 3.8 で変更: Added *copy.copy()* and *copy.deepcopy()* support to compression objects.

展開オブジェクトは以下のメソッドと属性をサポートしています:

`Decompress.unused_data`

圧縮データの末尾より後のバイト列が入ったバイト列オブジェクトです。すなわち、この値は圧縮データの入っているバイト列の最後の文字が利用可能になるまでは `b""` のままとなります。入力バイト文字列すべてが圧縮データを含んでいた場合、この属性は `b""` 、すなわち空バイト列になります。

`Decompress.unconsumed_tail`

展開されたデータを収めるバッファの長さ制限を超えたために、直近の *decompress()* 呼び出しで処理しきれなかったデータを含むバイト列オブジェクトです。このデータはまだ zlib 側からは見えていないので、正しい展開出力を得るには以降の *decompress()* メソッド呼び出しに (場合によっては後続のデータが追加された) データを差し戻さなければなりません。

Decompress.**eof**
: 圧縮データストリームの終了に達したかどうかを示すブール値です。

  これは、正常な形式の圧縮ストリームと、不完全あるいは切り詰められたストリームとを区別することを可能にします。

  バージョン 3.3 で追加.

Decompress.**decompress**(*data*, *max_length=0*)
: *data* を展開し、少なくとも *string* の一部分に対応する展開されたデータを含むバイト列オブジェクトを返します。このデータは以前に *decompress()* メソッドを呼んだ時に返された出力と結合することができます。入力データの一部分が以後の処理のために内部バッファに保存されることもあります。

  オプションパラメータ *max_length* が非ゼロの場合、返される展開データの長さが *max_length* 以下に制限されます。このことは入力した圧縮データの全てが処理されるとは限らないことを意味し、処理されなかったデータは *unconsumed_tail* 属性に保存されます。展開処理を継続する場合、この保存されたバイト文字列を以降の *decompress()* 呼び出しに渡さなくてはなりません。*max_length* が 0 の場合、全ての入力が展開され、*unconsumed_tail* 属性は空になります。

  バージョン 3.6 で変更: *max_length* can be used as a keyword argument.

Decompress.**flush**(*length*)
: 未処理の入力データをすべて処理し、最終的に圧縮されなかった残りの出力バイト列オブジェクトを返します。*flush()* を呼んだ後、*decompress()* を再度呼ぶべきではありません。このときできる唯一の現実的な操作はオブジェクトの削除だけです。

  オプション引数 *length* には出力バッファの初期サイズを指定します。

Decompress.**copy**()
: 展開オブジェクトのコピーを返します。これを使うとデータストリームの途中にある展開オブジェクトの状態を保存でき、未来のある時点で行なわれるストリームのランダムなシークをスピードアップするのに利用できます。

バージョン 3.8 で変更: Added *copy.copy()* and *copy.deepcopy()* support to decompression objects.

使用している zlib ライブラリのバージョン情報を以下の定数で確認できます:

zlib.**ZLIB_VERSION**
: モジュールのビルド時に使用された zlib ライブラリのバージョン文字列です。これは *ZLIB_RUNTIME_VERSION* で確認できる、実行時に使用している実際の zlib ライブラリのバージョンとは異なる場合があります。

zlib.**ZLIB_RUNTIME_VERSION**
: インタプリタが読み込んだ実際の zlib ライブラリのバージョン文字列です。

  バージョン 3.3 で追加.

参考:

gzip **モジュール** gzip 形式ファイルへの読み書きを行うモジュール。

http://www.zlib.net　zlib ライブラリホームページ。

http://www.zlib.net/manual.html　zlib ライブラリの多くの関数の意味と使い方を解説したマニュアル。

13.2 gzip --- gzip ファイルのサポート

ソースコード: Lib/gzip.py

このモジュールは、GNU の **gzip** や **gunzip** のようにファイルを圧縮、展開するシンプルなインターフェイスを提供しています。

データ圧縮は *zlib* モジュールで提供されています。

gzip は *GzipFile* クラスと、簡易関数 *open()*、*compress()*、および *decompress()* を提供しています。*GzipFile* クラスは通常の **ファイルオブジェクト** と同様に **gzip** 形式のファイルを読み書きし、データを自動的に圧縮または展開します。

compress や **pack** 等によって作成され、**gzip** や **gunzip** が展開できる他のファイル形式についてはこのモジュールは対応していないので注意してください。

このモジュールは以下の項目を定義しています:

gzip.open(*filename*, *mode='rb'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)

gzip 圧縮ファイルをバイナリまたはテキストモードで開き、**ファイルオブジェクト** を返します。

引数 *filename* には実際のファイル名 (*str* または *bytes* オブジェクト) か、既存のファイルオブジェクトを指定します。

引数 *mode* には、バイナリモード用に `'r'`、`'rb'`、`'a'`、`'ab'`、`'w'`、`'wb'`、`'x'`、または `'xb'`、テキストモード用に `'rt'`、`'at'`、`'wt'`、または `'xt'` を指定できます。デフォルトは `'rb'` です。

引数 *compresslevel* は *GzipFile* コンストラクタと同様に 0 から 9 の整数を取ります。

バイナリモードでは、この関数は *GzipFile* コンストラクタ `GzipFile(filename, mode, compresslevel)` と等価です。この時、引数 *encoding*、*errors*、および *newline* を指定してはいけません。

テキストモードでは、*GzipFile* オブジェクトが作成され、指定されたエンコーディング、エラーハンドラの挙動、および改行文字で *io.TextIOWrapper* インスタンスにラップされます。

バージョン 3.3 で変更: *filename* にファイルオブジェクト指定のサポート、テキストモードのサポート、および引数に *encoding*、*errors*、および *newline* を追加しました。

バージョン 3.4 で変更: Added support for the `'x'`, `'xb'` and `'xt'` modes.

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

exception gzip.BadGzipFile

An exception raised for invalid gzip files. It inherits *OSError*. *EOFError* and *zlib.error* can also be raised for invalid gzip files.

バージョン 3.8 で追加.

class gzip.GzipFile(*filename=None*, *mode=None*, *compresslevel=9*, *fileobj=None*, *mtime=None*)

GzipFile クラスのコンストラクタです。*GzipFile* オブジェクトは *truncate()* メソッドを除くほとんどの **ファイルオブジェクト** のメソッドをシミュレートします。少なくとも *fileobj* および *filename* は有効な値でなければなりません。

クラスの新しいインスタンスは、*fileobj* に基づいて作成されます。*fileobj* は通常のファイル、*io.BytesIO* オブジェクト、そしてその他ファイルをシミュレートできるオブジェクトでかまいません。値はデフォルトでは *None* で、その場合ファイルオブジェクトを生成するために *filename* を開きます。

fileobj が None でない場合、*filename* 引数は **gzip** ファイルヘッダにインクルードされることのみに使用されます。**gzip** ファイルヘッダは圧縮されていないファイルの元の名前をインクルードするかもしれません。認識可能な場合、規定値は *fileobj* のファイル名です。そうでない場合、規定値は空の文字列で、元のファイル名はヘッダにはインクルードされません。

引数 *mode* は、ファイルを読み込むのか書き出すのかによって、'r'、'rb'、'a'、'ab'、'w'、'wb'、'x'、および 'xb' のいずれかになります。*fileobj* のファイルモードが認識可能な場合、*mode* はデフォルトで *fileobj* のモードと同じになります。そうでない場合、デフォルトのモードは 'rb' です。

ファイルは常にバイナリモードで開かれることに注意してください。圧縮ファイルをテキストモードで開く場合、*open()* (または *GzipFile* を *io.TextIOWrapper* でラップしたオブジェクト) を使ってください。

引数 *compresslevel* は 0 から 9 の整数を取り、圧縮レベルを制御します; 1 は最も高速で最小限の圧縮を行い、9 は最も低速ですが最大限の圧縮を行います。0 は圧縮しません。デフォルトは 9 です。

mtime 引数は、圧縮時にストリームの最終更新日時フィールドに書き込まれるオプションの数値のタイムスタンプです。これは、圧縮モードでのみ提供することができます。省略された場合か None である場合、現在時刻が使用されます。詳細については、*mtime* 属性を参照してください。

圧縮したデータの後ろにさらに何か追加したい場合もあるので、*GzipFile* オブジェクトの close() メソッド呼び出しは *fileobj* を閉じません。このため、書き込みのためにオープンした *io.BytesIO* オブジェクトを *fileobj* として渡し、(*GzipFile* を close() した後に) *io.BytesIO* オブジェクトの *getvalue()* メソッドを使って書き込んだデータの入っているメモリバッファを取得することができます。

GzipFile は、イテレーションと with 文を含む *io.BufferedIOBase* インターフェイスをサポートしています。*truncate()* メソッドのみ実装されていません。

GzipFile は以下のメソッドと属性も提供しています:

peek(*n*)

ファイル内の位置を移動せずに展開した *n* バイトを読み込みます。呼び出し要求を満たすために、圧縮ストリームに対して最大 1 回の単一読み込みが行われます。返されるバイト数はほぼ要求した値になります。

注釈: *peek()* の呼び出しでは *GzipFile* のファイル位置は変わりませんが、下層のファイルオ

ブジェクトの位置が変わる恐れがあります。(e.g. *GzipFile* が *fileobj* 引数で作成された場合)

バージョン 3.2 で追加.

mtime

展開時に、最後に読み取られたヘッダーの最終更新日時フィールドの値は、この属性から整数として読み取ることができます。ヘッダーを読み取る前の初期値は None です。

gzip で圧縮されたすべてのストリームは、このタイムスタンプフィールドを含む必要があります。**gunzip** などの一部のプログラムがこのタイムスタンプを使用します。形式は、*time.time()* の返り値や、*os.stat()* が返すオブジェクトの *st_mtime* 属性と同一です。

バージョン 3.1 で変更: with 文がサポートされました。*mtime* コンストラクタ引数と *mtime* 属性が追加されました。

バージョン 3.2 で変更: ゼロパディングされたファイルやシーク出来ないファイルがサポートされました。

バージョン 3.3 で変更: *io.BufferedIOBase.read1()* メソッドを実装しました。

バージョン 3.4 で変更: 'x' ならびに 'xb' モードがサポートされました。

バージョン 3.5 で変更: 任意の **バイトライクオブジェクト** の書き込みがサポートされました。*read()* メソッドが None を引数として受け取るようになりました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

gzip.**compress**(*data*, *compresslevel=9*, ***, *mtime=None*)

data を圧縮し、圧縮データを含む *bytes* オブジェクトを返します。*compresslevel* と *mtime* の意味は上記 *GzipFile* コンストラクタと同じです。

バージョン 3.2 で追加.

バージョン 3.8 で変更: Added the *mtime* parameter for reproducible output.

gzip.**decompress**(*data*)

data を展開し、展開データを含む *bytes* オブジェクトを返します。

バージョン 3.2 で追加.

13.2.1 使い方の例

圧縮されたファイルを読み込む例:

```
import gzip
with gzip.open('/home/joe/file.txt.gz', 'rb') as f:
    file_content = f.read()
```

GZIP 圧縮されたファイルを作成する例:

```
import gzip
content = b"Lots of content here"
with gzip.open('/home/joe/file.txt.gz', 'wb') as f:
    f.write(content)
```

既存のファイルを GZIP 圧縮する例:

```
import gzip
import shutil
with open('/home/joe/file.txt', 'rb') as f_in:
    with gzip.open('/home/joe/file.txt.gz', 'wb') as f_out:
        shutil.copyfileobj(f_in, f_out)
```

バイナリ文字列を GZIP 圧縮する例:

```
import gzip
s_in = b"Lots of content here"
s_out = gzip.compress(s_in)
```

参考:

zlib **モジュール** `gzip` ファイル形式のサポートを行うために必要な基本ライブラリモジュール。

13.2.2 コマンドラインインターフェイス

gzip モジュールは、ファイルを圧縮、展開するための簡単なコマンドラインインターフェースを提供しています。

Once executed the *gzip* module keeps the input file(s).

バージョン 3.8 で変更: Add a new command line interface with a usage. By default, when you will execute the CLI, the default compression level is 6.

コマンドラインオプション

file

file を指定しない場合、*sys.stdin* から読み込みます。

--fast

Indicates the fastest compression method (less compression).

--best

Indicates the slowest compression method (best compression).

-d, --decompress

Decompress the given file.

-h, --help

ヘルプメッセージを出力します

13.3 bz2 --- bzip2 圧縮のサポート

ソースコード: Lib/bz2.py

このモジュールは、bzip2 アルゴリズムを用いて圧縮・展開を行う包括的なインターフェイスを提供します。

bz2 モジュールには以下のクラスや関数があります:

- 圧縮ファイルを読み書きするための *open()* 関数と *BZ2File* クラス。
- インクリメンタルにデータを圧縮・展開するための *BZ2Compressor* および *BZ2Decompressor* クラス。
- 一度に圧縮・展開を行う *compress()* および *decompress()* 関数。

このモジュールのクラスはすべて、複数のスレッドから安全にアクセスできます。

13.3.1 ファイルの圧縮/解凍

bz2.open(*filename*, *mode='r'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)

bzip2 圧縮されたファイルを、バイナリモードかテキストモードでオープンし、ファイルオブジェクトを返します。

BZ2File のコンストラクタと同様に、引数 *filename* には実際のファイル名 (*str* または *bytes* オブジェクト) か、読み書きする既存のファイルオブジェクトを指定します。

引数 *mode* には、バイナリモード用に `'r'`、`'rb'`、`'w'`、`'wb'`、`'x'`、`'xb'`、`'a'`、あるいは `'ab'`、テキストモード用に `'rt'`、`'wt'`、`'xt'`、あるいは `'at'` を指定できます。デフォルトは `'rb'` です。

引数 *compresslevel* には *BZ2File* コンストラクタと同様に 1 から 9 の整数を指定します。

バイナリモードでは、この関数は *BZ2File* コンストラクタ `BZ2File(filename, mode, compresslevel=compresslevel)` と等価です。この時、引数 *encoding*、*errors*、および *newline* を指定してはいけません。

テキストモードでは、*BZ2File* オブジェクトが作成され、指定されたエンコーディング、エラーハンドラの挙動、および改行文字で *io.TextIOWrapper* にラップされます。

バージョン 3.3 で追加.

バージョン 3.4 で変更: `'x'` (排他的作成) モードが追加されました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

class bz2.BZ2File(*filename*, *mode='r'*, *buffering=None*, *compresslevel=9*)

bzip2 圧縮ファイルをバイナリモードでオープンします。

filename が *str* あるいは *bytes* オブジェクトの場合、それを名前とするファイルを直接開きます。そうでない場合、*filename* は圧縮データを読み書きする ファイルオブジェクト でなくてはなりません。

引数 *mode* は読み込みモードの `'r'` (デフォルト)、上書きモードの `'w'`、排他的作成モードの `'x'`、あるいは追記モードの `'a'` のいずれかを指定できます。これらはそれぞれ `'rb'`、`'wb'`、`'xb'` および `'ab'` と等価です。

filename が (実際のファイル名でなく) ファイルオブジェクトの場合、`'w'` はファイルを上書きせず、`'a'` と等価になります。

引数 *buffering* は無視されます。Python 3.0 以降、この引数の使用は非推奨です。

mode が `'w'` あるいは `'a'` の場合、*compresslevel* に圧縮レベルを `1` から `9` の整数で指定できます。圧縮率は `1` が最低で、`9` (デフォルト値) が最高です。

mode の値が `'r'` の場合、入力ファイルは複数の圧縮ストリームでも構いません。

BZ2File には、*io.BufferedIOBase* で規定されているメソッドや属性のうち、`detach()` と `truncate()` を除くすべてが備わっています。イテレーションと `with` 文をサポートしています。

BZ2File は以下のメソッドも提供しています:

peek([*n*])

ファイル上の現在位置を変更せずにバッファのデータを返します。このメソッドは少なくとも 1 バイトのデータを返します (EOF の場合を除く)。返される正確なバイト数は規定されていません。

注釈: *peek()* の呼び出しでは *BZ2File* のファイル位置は変わりませんが、下層のファイルオブジェクトの位置が変わる惧れがあります (e.g. *BZ2File* を *filename* にファイルオブジェクトを渡して作成した場合)。

バージョン 3.3 で追加.

バージョン 3.0 で非推奨: キーワード引数 *buffering* は非推奨で無視されます。

バージョン 3.1 で変更: `with` 構文のサポートが追加されました。

バージョン 3.3 で変更: `fileno()`、`readable()`、`seekable()`、`writable()`、`read1()`、`readinto()` メソッドが追加されました。

バージョン 3.3 で変更: *filename* が実際のファイル名でなく **ファイルオブジェクト** だった場合のサポートが追加されました。

バージョン 3.3 で変更: `'a'` (追記) モードが追加され、複数のストリームの読み込みがサポートされました。

バージョン 3.4 で変更: `'x'` (排他的作成) モードが追加されました。

バージョン 3.5 で変更: *read()* メソッドが `None` を引数として受け取るようになりました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

13.3.2 逐次圧縮および展開

class bz2.BZ2Compressor(*compresslevel=9*)

新しくコンプレッサオブジェクトを作成します。このオブジェクトはデータの逐次的な圧縮に使用できます。一度に圧縮したい場合は、*compress()* 関数を使ってください。

引数 *compresslevel* を指定する場合は、1 から 9 までの整数を与えてください。デフォルト値は 9 です。

compress(*data*)

データをコンプレッサオブジェクトに渡します。戻り値は圧縮されたデータですが、圧縮データを返すことができない場合は空のバイト文字列を返します。

コンプレッサオブジェクトにデータをすべて渡し終えたら、*flush()* メソッドを呼び出し、圧縮プロセスを完了させてください。

flush()

圧縮プロセスを完了させ、内部バッファに残っている圧縮済みデータを返します。

このメソッドを呼び出すと、それ以後コンプレッサオブジェクトは使用できなくなります。

class bz2.BZ2Decompressor

新しくデコンプレッサオブジェクトを作成します。このオブジェクトは逐次的なデータ展開に使用できます。一度に展開したい場合は、*decompress()* 関数を使ってください。

注釈: このクラスは、*decompress()* や *BZ2File* とは異なり、複数の圧縮レベルが混在しているデータを透過的に扱うことができません。*BZ2Decompressor* クラスを用いて、複数のストリームからなるデータを展開する場合は、それぞれのストリームについてデコンプレッサオブジェクトを用意してください。

---

decompress(*data*, *max_length=-1*)

data (*bytes-like object*) を展開し、未圧縮のデータを bytes で返します。*data* の一部は、後で *decompress()* の呼び出しに使用するため内部でバッファされている場合があります。返すデータは以前の *decompress()* 呼び出しの出力を全て連結したものです。

max_length が非負の場合、最大 *max_length* バイトの展開データを返します。この制限に達して、出力がさらに生成できる場合、*needs_input* が False に設定されます。この場合、*decompress()* を次に呼び出すと、*data* を `b''` として提供し、出力をさらに取得することができます。

入力データの全てが圧縮され返された (*max_length* バイトより少ないためか *max_length* が負のため) 場合、*needs_input* 属性は True になります。

ストリームの終端に到達した後にデータを展開しようとすると *EOFError* が送出されます。ストリームの終端の後ろの全てのデータは無視され、その部分は *unused_data* 属性に保存されます。

バージョン 3.5 で変更: max_length パラメータが追加されました。

eof

ストリーム終端記号に到達した場合 True を返します。

バージョン 3.3 で追加.

unused_data
圧縮ストリームの末尾以降に存在したデータを表します。

ストリームの末尾に達する前には、この属性には `b''` という値が収められています。

needs_input
decompress() メソッドが、新しい非圧縮入力を必要とせずにさらに展開データを提供できる場合、`False` です。

バージョン 3.5 で追加.

13.3.3 一括圧縮/解凍

bz2.compress(*data*, *compresslevel=9*)
バイト類オブジェクト の *data* を圧縮します。

引数 *compresslevel* を指定する場合は、1 から 9 までの整数を与えてください。デフォルト値は 9 です。

逐次的にデータを圧縮したい場合は、*BZ2Compressor* を使ってください。

bz2.decompress(*data*)
バイト類オブジェクト の *data* を展開します。

data が複数の圧縮ストリームから成る場合、そのすべてを展開します。

逐次的に展開を行う場合は、*BZ2Decompressor* を使ってください。

バージョン 3.3 で変更: 複数ストリームの入力をサポートしました。

13.3.4 使い方の例

以下は、典型的な *bz2* モジュールの利用方法です。

compress() と *decompress()* を使い、圧縮して展開する実演をしています:

```
>>> import bz2
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
>>> c = bz2.compress(data)
>>> len(data) / len(c)  # Data compression ratio
1.513595166163142
>>> d = bz2.decompress(c)
```

(次のページに続く)

(前のページからの続き)

```
>>> data == d  # Check equality to original object after round-trip
True
```

BZ2Compressor を使い、逐次圧縮をしています:

```
>>> import bz2
>>> def gen_data(chunks=10, chunksize=1000):
...     """Yield incremental blocks of chunksize bytes."""
...     for _ in range(chunks):
...         yield b"z" * chunksize
...
>>> comp = bz2.BZ2Compressor()
>>> out = b""
>>> for chunk in gen_data():
...     # Provide data to the compressor object
...     out = out + comp.compress(chunk)
...
>>> # Finish the compression process.  Call this once you have
>>> # finished providing data to the compressor.
>>> out = out + comp.flush()
```

上の例は、非常に ” ランダムでない” データストリーム (チャンク *b”z”* のストリーム) です。ランダムなデータは圧縮率が低い傾向にある一方、揃っていて、繰り返しのあるデータは通常は高い圧縮率を叩き出します。

bzip2 圧縮されたファイルをバイナリモードで読み書きしています:

```
>>> import bz2
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
>>> with bz2.open("myfile.bz2", "wb") as f:
...     # Write compressed data to file
...     unused = f.write(data)
>>> with bz2.open("myfile.bz2", "rb") as f:
...     # Decompress data from file
...     content = f.read()
>>> content == data  # Check equality to original object after round-trip
True
```

13.4 lzma --- LZMA アルゴリズムを使用した圧縮

バージョン 3.3 で追加.

ソースコード: Lib/lzma.py

このモジュールは LZMA 圧縮アルゴリズムを使用したデータ圧縮および展開のためのクラスや便利な関数を提供しています。また、**xz** ユーティリティを使用した .xz およびレガシーな .lzma ファイル形式へのファイルインターフェイスの他、RAW 圧縮ストリームもサポートしています。

このモジュールが提供するインターフェイスは *bz2* モジュールと非常によく似ています。ただし、*LZMAFile* は (*bz2.BZ2File* と異なり) スレッドセーフではない点に注意してください。単一の *LZMAFile* インスタンスを複数スレッドから使用する場合は、ロックで保護する必要があります。

exception lzma.LZMAError

この例外は圧縮あるいは展開中にエラーが発生した場合、または圧縮/展開状態の初期化中に送出されます。

13.4.1 圧縮ファイルへの読み書き

lzma.open(*filename*, *mode="rb"*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*, *encoding=None*, *errors=None*, *newline=None*)

LZMA 圧縮ファイルをバイナリまたはテキストモードでオープンし、**ファイルオブジェクト** を返します。

filename 引数には、ファイルをオープンする際には実際のファイル名 (*str*、*bytes*、または *path-like* オブジェクトとして指定します) か、読み込みまたは書き込むためであれば、すでに存在するファイルオブジェクトを指定できます。

引数 *mode* は、バイナリモードでは "r"、"rb"、"w"、"wb"、"x"、"xb"、"a"、あるいは "ab" の、テキストモードでは "rt"、"wt"、"xt"、あるいは "at" のいずれかになります。デフォルトは "rb" です。

読み込み用にファイルをオープンした場合、引数 *format* および *filters* は *LZMADecompressor* と同じ意味になります。この時、引数 *check* および *preset* は使用しないでください。

書き出し用にオープンした場合、引数 *format*、*check*、*preset*、および *filters* は *LZMACompressor* と同じ意味になります。

バイナリモードでは、この関数は *LZMAFile* コンストラクタと等価になります (LZMAFile(filename, mode, ...))。この場合、引数 *encoding*、*errors*、および *newline* を指定しなければなりません。

テキストモードでは、*LZMAFile* オブジェクトが生成され、指定したエンコーディング、エラーハンドラの挙動、および改行コードで *io.TextIOWrapper* にラップされます。

バージョン 3.4 で変更: "x", "xb", "xt" モードのサポートが追加されました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

class lzma.LZMAFile(*filename=None*, *mode="r"*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*)

LZMA 圧縮ファイルをバイナリモードで開きます。

LZMAFile はすでにオープンしている *file object* をラップ、または名前付きファイルを直接操作できます。引数 *filename* にはラップするファイルオブジェクトかオープンするファイル名 (*str* オブジェクトま、*bytes* オブジェクト、または *path-like* オブジェクト) を指定します。既存のファイルオブジェクトをラップした場合、*LZMAFile* をクローズしてもラップしたファイルはクローズされません。

引数 *mode* は読み込みモードの `"r"` (デフォルト)、上書きモードの `"w"`、排他的生成モードの `"x"`、あるいは追記モードの `"a"` のいずれかを指定できます。これらはそれぞれ `"rb"`、`"wb"`、`"xb"`、および `"ab"` と等価です。

filename が (実際のファイル名でなく) ファイルオブジェクトの場合、`"w"` モードはファイルを上書きせず、`"a"` と等価になります。

読み込みモードでファイルをオープンした時、入力ファイルは複数に分割された圧縮ストリームを連結したものでもかまいません。これらは透過的に単一論理ストリームとしてデコードされます。

読み込み用にファイルをオープンした場合、引数 *format* および *filters* は *LZMADecompressor* と同じ意味になります。この時、引数 *check* および *preset* は使用しないでください。

書き出し用にオープンした場合、引数 *format*、*check*、*preset*、および *filters* は *LZMACompressor* と同じ意味になります。

LZMAFile は *io.BufferedIOBase* で規定されているメンバのうち、`detach()` と `truncate()` を除くすべてをサポートします。イテレーションと `with` 文をサポートしています。

次のメソッドを提供しています:

peek(*size=-1*)

ファイル上の現在位置を変更せずにバッファのデータを返します。EOF に達しない限り、少なくとも 1 バイトが返されます。返される正確なバイト数は規定されていません (引数 *size* は無視されます)。

注釈: *peek()* の呼び出しでは *LZMAFile* のファイル位置は変わりませんが、下層のファイルオブジェクトの位置が変わる惧れがあります。(e.g. *LZMAFile* を *filename* にファイルオブジェクトを渡して作成した場合)

バージョン 3.4 で変更: `'x'`, `'xb'` モードがサポートが追加されました。

バージョン 3.5 で変更: *read()* メソッドが `None` を引数として受け取るようになりました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

13.4.2 メモリ上での圧縮と展開

class lzma.LZMACompressor(*format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)

データをインクリメンタルに圧縮する圧縮オブジェクトを作成します。

大量の単一データを圧縮する、より便利な方法については *compress()* を参照してください。

引数 *format* には使用するコンテナフォーマットを指定します。以下の値が指定できます:

- FORMAT_XZ: The .xz コンテナフォーマット。 デフォルトのフォーマットです。
- FORMAT_ALONE: レガシーな .lzma コンテナフォーマット。 このフォーマットは .xz より制限があります -- インテグリティチェックや複数フィルタをサポートしていません。
- FORMAT_RAW: 特定のコンテナフォーマットを使わない、生のデータストリーム。 このフォーマット指定子はインテグリティチェックをサポートしておらず、(圧縮および展開双方のために) 常にカスタムフィルタチェインを指定する必要があります。さらに、この方法で圧縮されたデータは FORMAT_AUTO を使っても展開できません (*LZMADecompressor* を参照)。

引数 *check* には圧縮データに組み込むインテグリティチェックのタイプを指定します。このチェックは展開時に使用され、データが破損していないことを保証します。以下の値が指定できます:

- CHECK_NONE: インテグリティチェックなし。FORMAT_ALONE および FORMAT_RAW のデフォルト (かつ唯一指定可能な値) です。
- CHECK_CRC32: 32-bit 巡回冗長検査。
- CHECK_CRC64: 64-bit 巡回冗長検査。FORMAT_XZ のデフォルトです。
- CHECK_SHA256: 256-bit セキュアハッシュアルゴリズム (SHA)。

指定したチェック方法がサポートされていない場合、*LZMAError* が送出されます。

圧縮設定はプリセット圧縮レベル (引数 *preset* で指定) またはカスタムフィルタチェイン (引数 *filters* で指定) のどちらかを指定します。

引数 *preset* を指定する場合、0 から 9 までの整数値でなければならず、オプションで定数 PRESET_EXTREME を論理和指定できます。*preset* も *filters* も指定されなかった場合、デフォルトの挙動として PRESET_DEFAULT (プリセットレベル 6) が使用されます。高いプリセット値を指定すると圧縮率が上がりますが、圧縮にかかる時間が長くなります。

注釈: CPU の使用量が多いのに加えて、高いプリセットで圧縮を行うには、メモリをずっと多く必要とします (さらに、生成される出力も展開により多くのメモリを必要とします)。例えば、プリセットが 9 の場合、*LZMACompressor* オブジェクトのオーバーヘッドは 800 MiB にまで高くなる場合があります。このため、通常はデフォルトのプリセットを使用するのがよいでしょう。

引数 *filters* を指定する場合、フィルタチェイン指定子でなければなりません。詳しくは **カスタムフィルタチェインの指定** を参照してください。

compress(*data*)

data (*bytes* オブジェクト) を圧縮し、少なくともその一部が圧縮されたデータを格納する *bytes* オブジェクトを返します。*data* の一部は、後で *compress()* および *flush()* の呼び出しに使用するため内部でバッファされている場合があります。返すデータは以前の *compress()* 呼び出しの出力を連結したものです。

flush()

圧縮処理を終了し、コンプレッサの内部バッファにあるあらゆるデータを格納する *bytes* オブジェクトを返します。

コンプレッサはこのメソッドが呼び出された後は使用できません。

class lzma.LZMADecompressor(*format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)

データをインクリメンタルに展開するために使用できる展開オブジェクトを作成します。

圧縮されたストリーム全体を一度に展開にする、より便利な方法については、*decompress()* を参照してください。

引数 *format* には使用するコンテナフォーマットを指定します。デフォルトは FORMAT_AUTO で、.xz および .lzma ファイルを展開できます。その他に指定できる値は、FORMAT_XZ、FORMAT_ALONE、および FORMAT_RAW です。

引数 *memlimit* にはデコンプレッサが使用できるメモリ量をバイトで指定します。この引数を指定した場合、そのメモリ量で展開ができないと *LZMAError* を送出します。

引数 *filters* には展開されるストリームの作成に使用するフィルタチェインを指定します。この引数を使用する際は、引数 *format* に FORMAT_RAW を指定しなければなりません。フィルタチェインについての詳細は **カスタムフィルタチェインの指定** を参照してください。

注釈: このクラスは *decompress()* および *LZMAFile* と異なり、複数の圧縮ストリームを含む入力を透過的に扱いません。*LZMADecompressor* で複数ストリーム入力を展開するには、各ストリームごとに新しいデコンプレッサを作成しなければなりません。

---

decompress(*data*, *max_length=-1*)

data (*bytes-like object*) を展開し、未圧縮のデータを bytes で返します。*data* の一部は、後で *decompress()* の呼び出しに使用するため内部でバッファされている場合があります。返すデータは以前の *decompress()* 呼び出しの出力を全て連結したものです。

max_length が非負の場合、最大 *max_length* バイトの展開データを返します。この制限に達して、出力がさらに生成できる場合、*needs_input* が False に設定されます。この場合、*decompress()* を次に呼び出すと、*data* を b'' として提供し、出力をさらに取得することができます。

入力データの全てが圧縮され返された (*max_length* バイトより少ないためか *max_length* が負のため) 場合、*needs_input* 属性は True になります。

ストリームの終端に到達した後にデータを展開しようとすると *EOFError* が送出されます。ストリームの終端の後ろの全てのデータは無視され、その部分は *unused_data* 属性に保存されます。

バージョン 3.5 で変更: max_length パラメータが追加されました。

check
入力ストリームに使用されるインテグリティチェックの ID です。これは何のインテグリティチェックが使用されているか決定するために十分な入力がデコードされるまでは CHECK_UNKNOWN になることがあります。

eof
ストリーム終端記号に到達した場合 True を返します。

unused_data
圧縮ストリームの末尾以降に存在したデータを表します。

ストリームの末尾に達する前は、これは b"" になります。

needs_input
decompress() メソッドが、新しい非圧縮入力を必要とせずにさらに展開データを提供できる場合、False です。

バージョン 3.5 で追加.

lzma.**compress**(*data*, *format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)
data (*bytes* オブジェクト) を圧縮し、圧縮データを *bytes* オブジェクトとして返します。

引数 *format*、*check*、*preset*、および *filters* についての説明は上記の *LZMACompressor* を参照してください。

lzma.**decompress**(*data*, *format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)
data (*bytes* オブジェクト) を展開し、展開データを *bytes* オブジェクトとして返します。

data が複数の明確な圧縮ストリームの連結だった場合、すべてのストリームを展開し、結果の連結を返します。

引数 *format*、*memlimit*、および *filters* の説明は、上記 *LZMADecompressor* を参照してください。

13.4.3 その他

lzma.**is_check_supported**(*check*)
指定したインテグリティチェックがシステムでサポートされていれば True を返します。

CHECK_NONE および CHECK_CRC32 は常にサポートされています。CHECK_CRC64 および CHECK_SHA256 は **liblzma** が機能制限セットでコンパイルされている場合利用できないことがあります。

13.4.4 カスタムフィルタチェインの指定

フィルタチェイン指定子は、辞書のシーケンスで、各辞書は ID と単一フィルタのオプションからなります。各辞書はキー "id" を持たなければならず、フィルタ依存のオプションを指定する追加キーを持つ場合もあります。有効なフィルタ ID は以下のとおりです:

- **圧縮フィルタ**:
  - FILTER_LZMA1 (FORMAT_ALONE と共に使用)
  - FILTER_LZMA2 (FORMAT_XZ および FORMAT_RAW と共に使用)
- **デルタフィルター**:
  - FILTER_DELTA
- **ブランチコールジャンプ** (BCJ) **フィルター**:
  - FILTER_X86
  - FILTER_IA64
  - FILTER_ARM
  - FILTER_ARMTHUMB
  - FILTER_POWERPC
  - FILTER_SPARC

一つのフィルタチェインは 4 個までのフィルタを定義することができ、空にはできません。チェインの最後は圧縮フィルタでなくてはならず、その他のフィルタはデルタまたは BCJ フィルタでなければなりません。

圧縮フィルタは以下のオプション (追加エントリとしてフィルタを表す辞書に指定) をサポートしています:

- preset: 明示されていないオプションのデフォルト値のソースとして使用する圧縮プリセット。
- dict_size: 辞書のサイズのバイト数。これは、4 KiB から 1.5 GiB の間にしてください (両端を含みます)。
- lc: リテラルコンテキストビットの数。
- lp: リテラル位置ビットの数。lc + lp で最大 4 までです。
- pb: 位置ビットの数。最大で 4 までです。
- mode: MODE_FAST または MODE_NORMAL。
- nice_len: マッチに ”良い” とみなす長さ。273 以下でなければなりません。
- mf: 使用するマッチファインダ -- MF_HC3、MF_HC4、MF_BT2、MF_BT3、または MF_BT4。
- depth: マッチファインダが使用する検索の最大深度。0 (デフォルト) では他のフィルタオプションをベースに自動選択します。

デルタフィルターは、バイト間の差異を保存し、特定の状況で、コンプレッサーに対してさらに反復的な入力を生成します。デルタフィルターは、1 つのオプション `dist` のみをサポートします。これは差し引くバイトどうしの距離を示します。デフォルトは 1 で、隣接するバイトの差異を扱います。

The BCJ filters are intended to be applied to machine code. They convert relative branches, calls and jumps in the code to use absolute addressing, with the aim of increasing the redundancy that can be exploited by the compressor. These filters support one option, `start_offset`. This specifies the address that should be mapped to the beginning of the input data. The default is 0.

13.4.5 使用例

圧縮ファイルからの読み込み:

```
import lzma
with lzma.open("file.xz") as f:
    file_content = f.read()
```

圧縮ファイルの作成:

```
import lzma
data = b"Insert Data Here"
with lzma.open("file.xz", "w") as f:
    f.write(data)
```

メモリ上でデータを圧縮:

```
import lzma
data_in = b"Insert Data Here"
data_out = lzma.compress(data_in)
```

逐次圧縮:

```
import lzma
lzc = lzma.LZMACompressor()
out1 = lzc.compress(b"Some data\n")
out2 = lzc.compress(b"Another piece of data\n")
out3 = lzc.compress(b"Even more data\n")
out4 = lzc.flush()
# Concatenate all the partial results:
result = b"".join([out1, out2, out3, out4])
```

すでにオープンしているファイルへの圧縮データの書き出し:

```
import lzma
with open("file.xz", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with lzma.open(f, "w") as lzf:
        lzf.write(b"This *will* be compressed\n")
    f.write(b"Not compressed\n")
```

カスタムフィルタチェインを使った圧縮ファイルの作成:

```
import lzma
my_filters = [
    {"id": lzma.FILTER_DELTA, "dist": 5},
    {"id": lzma.FILTER_LZMA2, "preset": 7 | lzma.PRESET_EXTREME},
]
with lzma.open("file.xz", "w", filters=my_filters) as f:
    f.write(b"blah blah blah")
```

13.5 zipfile --- ZIP アーカイブの処理

ソースコード: Lib/zipfile.py

ZIP は一般によく知られているアーカイブ (書庫化) および圧縮の標準ファイルフォーマットです。このモジュールでは ZIP 形式のファイルの作成、読み書き、追記、書庫内のファイル一覧の作成を行うためのツールを提供します。より高度な使い方でこのモジュールを利用したいのであれば、PKZIP Application Note に定義されている ZIP ファイルフォーマットの理解が必要になるでしょう。

このモジュールは現在マルチディスク ZIP ファイルを扱うことはできません。ZIP64 拡張を利用する ZIP ファイル (サイズが 4 GiB を超えるような ZIP ファイル) は扱えます。このモジュールは暗号化されたアーカイブの復号をサポートしますが、現在暗号化ファイルを作成することはできません。C 言語ではなく、Python で実装されているため、復号は非常に遅くなっています。

このモジュールは以下の項目を定義しています:

exception zipfile.BadZipFile

正常ではない ZIP ファイルに対して送出されるエラーです。

バージョン 3.2 で追加.

exception zipfile.BadZipfile

BadZipFile の別名です。過去のバージョンの Python との互換性のために用意されています。

バージョン 3.2 で非推奨.

exception zipfile.LargeZipFile

ZIP ファイルが ZIP64 の機能を必要としているが、その機能が有効化されていない場合に送出されるエラーです。

class zipfile.ZipFile

ZIP ファイルの読み書きのためのクラスです。コンストラクタの詳細については、*ZipFile* **オブジェクト** 節を参照してください。

class zipfile.Path

A pathlib-compatible wrapper for zip files. See section *Path* **オブジェクト** for details.

バージョン 3.8 で追加.

class zipfile.PyZipFile

Python ライブラリを含む、ZIP アーカイブを作成するためのクラスです。

class zipfile.ZipInfo(*filename='NoName'*, *date_time=(1980, 1, 1, 0, 0, 0)*)

アーカイブ内の 1 個のメンバの情報を取得するために使うクラスです。このクラスのインスタンスは *ZipFile* オブジェクトの *getinfo()* および *infolist()* メソッドによって返されます。ほとんどの *zipfile* モジュールの利用者はこのクラスのインスタンスを作成する必要はなく、このモジュールによって作成されたものを使用できます。*filename* はアーカイブメンバのフルネームでなければならず、*date_time* はファイルが最後に変更された日時を表す 6 個のフィールドのタプルでなければなりません; フィールドは *ZipInfo* **オブジェクト** 節で説明されています。

zipfile.is_zipfile(*filename*)

filename が正しいマジックナンバをもつ ZIP ファイルの時に `True` を返し、そうでない場合 `False` を返します。*filename* にはファイルやファイルライクオブジェクトを渡すこともできます。

バージョン 3.1 で変更: ファイルおよびファイルライクオブジェクトをサポートしました。

zipfile.ZIP_STORED

アーカイブメンバを圧縮しない (複数ファイルを一つにまとめるだけ) ことを表す数値定数です。

zipfile.ZIP_DEFLATED

通常の ZIP 圧縮方法を表す数値定数です。これには *zlib* モジュールが必要です。

zipfile.ZIP_BZIP2

BZIP2 圧縮方法を表す数値定数です。これには *bz2* モジュールが必要です。

バージョン 3.3 で追加.

zipfile.ZIP_LZMA

LZMA 圧縮方法を表す数値定数です。これには *lzma* モジュールが必要です。

バージョン 3.3 で追加.

注釈: ZIP ファイルフォーマット仕様は 2001 年より bzip2 圧縮を、2006 年より LZMA 圧縮をサポートしていますが、(過去の Python リリースを含む) 一部のツールはこれら圧縮方式をサポートしていないため、ZIP ファイルの処理を全く受け付けないか、あるいは個々のファイルの抽出に失敗する場合があります。

参考:

PKZIP Application Note ZIP ファイルフォーマットおよびアルゴリズムを作成した Phil Katz によるドキュメント。

Info-ZIP Home Page Info-ZIP プロジェクトによる ZIP アーカイブプログラムおよびプログラム開発ライブラリに関する情報。

13.5.1 ZipFile オブジェクト

class zipfile.ZipFile(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *compresslevel=None*, ***, *strict_timestamps=True*)

ZIP ファイルを開きます。*file* はファイルのパス (文字列) か、ファイルライクオブジェクト、*path-like object* のいずれかです。

mode パラメータには、既存のファイルを読み込む場合は `'r'` 、内容を消去して新しいファイルに書き込む場合は `'w'` 、既存のファイルの末尾に追加する場合は `'a'` 、ファイルが存在しない場合にのみファイルを作成して書き込む場合は `'x'` を指定します。*mode* が `'x'` で *file* が既存のファイルを指している場合、*FileExistsError* が発生します。*mode* が `'a'` で *file* が既存の ZIP ファイルを指している場合、新しい ZIP アーカイブがそのファイルに追加されます。*file* が ZIP ファイルでない場合は、ファイルの末尾にあたらしい ZIP アーカイブが追加されます。これは、既存のファイル (例えば `python.exe`) に ZIP アーカイブを付け加える用途を想定したものです。*mode* が `'a'` で *file* が存在しない場合は、ファイルが作成されます。*mode* が `'r'` か `'a'` の場合、ファイルはシーク可能である必要があります。

compression is the ZIP compression method to use when writing the archive, and should be *ZIP_STORED*, *ZIP_DEFLATED*, *ZIP_BZIP2* or *ZIP_LZMA*; unrecognized values will cause *NotImplementedError* to be raised. If *ZIP_DEFLATED*, *ZIP_BZIP2* or *ZIP_LZMA* is specified but the corresponding module (*zlib*, *bz2* or *lzma*) is not available, *RuntimeError* is raised. The default is *ZIP_STORED*.

If *allowZip64* is `True` (the default) zipfile will create ZIP files that use the ZIP64 extensions when the zipfile is larger than 4 GiB. If it is `false` *zipfile* will raise an exception when the ZIP file would require ZIP64 extensions.

The *compresslevel* parameter controls the compression level to use when writing files to the archive. When using *ZIP_STORED* or *ZIP_LZMA* it has no effect. When using *ZIP_DEFLATED* integers `0` through `9` are accepted (see *zlib* for more information). When using *ZIP_BZIP2* integers `1` through `9` are accepted (see *bz2* for more information).

The *strict_timestamps* argument, when set to `False`, allows to zip files older than 1980-01-01 at the cost of setting the timestamp to 1980-01-01. Similar behavior occurs with files newer than 2107-12-31, the timestamp is also set to the limit.

ファイルがモード `'w'`、`'x'` または `'a'` で作成され、その後そのアーカイブにファイルを追加することなく **クローズ** された場合、空のアーカイブのための適切な ZIP 構造がファイルに書き込まれます。

ZipFile はコンテキストマネージャにもなっているので、`with` 文をサポートしています。次の例では、*myzip* は `with` 文のブロックが終了したときに、(たとえ例外が発生したとしても) クローズされます:

```
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

バージョン 3.2 で追加: *ZipFile* をコンテキストマネージャとして使用できるようになりました。

バージョン 3.3 で変更: *bzip2* および *lzma* 圧縮をサポートしました。

バージョン 3.4 で変更: ZIP64 拡張がデフォルトで有効になりました。

バージョン 3.5 で変更: seek 出来ないストリームのサポートが追加されました。'x' モードのサポートが追加されました。

バージョン 3.6 で変更: Previously, a plain *RuntimeError* was raised for unrecognized compression values.

バージョン 3.6.2 で変更: *file* 引数が *path-like object* を受け入れるようになりました。

バージョン 3.7 で変更: *compresslevel* 引数が追加されました。

バージョン 3.8 で追加: *strict_timestamps* キーワード専用引数。

ZipFile.close()
アーカイブファイルをクローズします。*close()* はプログラムを終了する前に必ず呼び出さなければなりません。さもないとアーカイブ上の重要なレコードが書き込まれません。

ZipFile.getinfo(*name*)
アーカイブメンバ *name* に関する情報を持つ *ZipInfo* オブジェクトを返します。アーカイブに含まれないファイル名に対して *getinfo()* を呼び出すと、*KeyError* が送出されます。

ZipFile.infolist()
アーカイブに含まれる各メンバの *ZipInfo* オブジェクトからなるリストを返します。既存のアーカイブファイルを開いている場合、リストの順番は実際の ZIP ファイル中のメンバの順番と同じになります。

ZipFile.namelist()
アーカイブメンバの名前のリストを返します。

ZipFile.open(*name*, *mode='r'*, *pwd=None*, ***, *force_zip64=False*)
Access a member of the archive as a binary file-like object. *name* can be either the name of a file within the archive or a *ZipInfo* object. The *mode* parameter, if included, must be 'r' (the default) or 'w'. *pwd* is the password used to decrypt encrypted ZIP files.

open() はコンテキストマネージャでもあるので with 文をサポートしています:

```
with ZipFile('spam.zip') as myzip:
    with myzip.open('eggs.txt') as myfile:
        print(myfile.read())
```

With *mode* 'r' the file-like object (ZipExtFile) is read-only and provides the following methods: *read()*, *readline()*, *readlines()*, *seek()*, *tell()*, __iter__(), *__next__()*. These objects can operate independently of the ZipFile.

With mode='w', a writable file handle is returned, which supports the *write()* method. While a writable file handle is open, attempting to read or write other files in the ZIP file will raise a *ValueError*.

When writing a file, if the file size is not known in advance but may exceed 2 GiB, pass force_zip64=True to ensure that the header format is capable of supporting large files. If the

file size is known in advance, construct a *ZipInfo* object with *file_size* set, and use that as the *name* parameter.

注釈: *open()*、*read()*、および *extract()* メソッドには、ファイル名または *ZipInfo* オブジェクトを指定できます。これは重複する名前のメンバを含む ZIP ファイルを読み込むときにそのメリットを享受できるでしょう。

バージョン 3.6 で変更: Removed support of `mode='U'`. Use *io.TextIOWrapper* for reading compressed text files in *universal newlines* mode.

バージョン 3.6 で変更: *open()* can now be used to write files into the archive with the `mode='w'` option.

バージョン 3.6 で変更: Calling *open()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

`ZipFile.extract`(*member*, *path=None*, *pwd=None*)

メンバをアーカイブから現在の作業ディレクトリに展開します。*member* は展開するファイルのフルネームまたは *ZipInfo* オブジェクトでなければなりません。ファイル情報は可能な限り正確に展開されます。*path* は展開先のディレクトリを指定します。*member* はファイル名または *ZipInfo* オブジェクトです。*pwd* は暗号化ファイルに使われるパスワードです。

作成された (ディレクトリか新ファイルの) 正規化されたパスを返します。

注釈: メンバのファイル名が絶対パスなら、ドライブ/UNC sharepoint および先頭の (バック) スラッシュは取り除かれます。例えば、Unix で `///foo/bar` は `foo/bar` となり、Window で `C:\foo\bar` は `foo\bar` となります。また、メンバのファイル名に含まれる全ての `".."` は取り除かれます。例えば、`../../foo../../ba..r` は `foo../ba..r` となります。Windows では、不正な文字 (`:`, `<`, `>`, `|`, `"`, `?`, および `*`) はアンダースコア (`_`) で置き換えられます。

バージョン 3.6 で変更: Calling *extract()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

バージョン 3.6.2 で変更: *path* パラメタが *path-like object* を受け付けるようになりました。

`ZipFile.extractall`(*path=None*, *members=None*, *pwd=None*)

すべてのメンバをアーカイブから現在の作業ディレクトリに展開します。*path* は展開先のディレクトリを指定します。*members* は、オプションで、*namelist()* で返されるリストの部分集合でなければなりません。*pwd* は、暗号化ファイルに使われるパスワードです。

警告: 信頼できないソースからきた Zip ファイルを、事前に中身をチェックせずに展開してはいけません。ファイルを *path* の外側に作成することができるからです。例えば、`"/"` で始まる絶対パスを持ったメンバーや、2 つのドット `".."` を持つファイル名などの場合です。このモジュール

はそれを避けようとします。*extract()* の注釈を参照してください。

バージョン 3.6 で変更: Calling *extractall()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

バージョン 3.6.2 で変更: *path* パラメタが *path-like object* を受け付けるようになりました。

ZipFile.printdir()
アーカイブの内容の一覧を `sys.stdout` に出力します。

ZipFile.setpassword(*pwd*)
pwd を展開する圧縮ファイルのデフォルトパスワードとして指定します。

ZipFile.read(*name*, *pwd=None*)
Return the bytes of the file *name* in the archive. *name* is the name of the file in the archive, or a *ZipInfo* object. The archive must be open for read or append. *pwd* is the password used for encrypted files and, if specified, it will override the default password set with *setpassword()*. Calling *read()* on a ZipFile that uses a compression method other than *ZIP_STORED*, *ZIP_DEFLATED*, *ZIP_BZIP2* or *ZIP_LZMA* will raise a *NotImplementedError*. An error will also be raised if the corresponding compression module is not available.

バージョン 3.6 で変更: Calling *read()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

ZipFile.testzip()
Read all the files in the archive and check their CRC's and file headers. Return the name of the first bad file, or else return `None`.

バージョン 3.6 で変更: Calling *testzip()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

ZipFile.write(*filename*, *arcname=None*, *compress_type=None*, *compresslevel=None*)
Write the file named *filename* to the archive, giving it the archive name *arcname* (by default, this will be the same as *filename*, but without a drive letter and with leading path separators removed). If given, *compress_type* overrides the value given for the *compression* parameter to the constructor for the new entry. Similarly, *compresslevel* will override the constructor if given. The archive must be open with mode `'w'`, `'x'` or `'a'`.

注釈: アーカイブ名はアーカイブルートに対する相対パスでなければなりません。言い換えると、アーカイブ名はパスセパレータで始まってはいけません。

注釈: もし、`arcname` (`arcname` が与えられない場合は、`filename`) が null byte を含むなら、アーカイブ中のファイルのファイル名は、null byte までで切り詰められます。

バージョン 3.6 で変更: Calling *write()* on a ZipFile created with mode `'r'` or a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

ZipFile.**writestr**(*zinfo_or_arcname*, *data*, *compress_type=None*, *compresslevel=None*)

Write a file into the archive. The contents is *data*, which may be either a *str* or a *bytes* instance; if it is a *str*, it is encoded as UTF-8 first. *zinfo_or_arcname* is either the file name it will be given in the archive, or a *ZipInfo* instance. If it's an instance, at least the filename, date, and time must be given. If it's a name, the date and time is set to the current date and time. The archive must be opened with mode `'w'`, `'x'` or `'a'`.

If given, *compress_type* overrides the value given for the *compression* parameter to the constructor for the new entry, or in the *zinfo_or_arcname* (if that is a *ZipInfo* instance). Similarly, *compresslevel* will override the constructor if given.

注釈: *ZipInfo* インスタンスを引数 *zinfo_or_arcname* として与えた場合、与えられた *ZipInfo* インスタンスのメンバーである *compress_type* で指定された圧縮方法が使われます。デフォルトでは、*ZipInfo* コンストラクターが、このメンバーを *ZIP_STORED* に設定します。

バージョン 3.2 で変更: 引数 *compress_type* を追加しました。

バージョン 3.6 で変更: Calling *writestr()* on a ZipFile created with mode `'r'` or a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

以下のデータ属性も利用することができます:

ZipFile.**filename**

ZIP ファイルの名前です。

ZipFile.**debug**

使用するデバッグ出力レベルです。この属性は 0 (デフォルト、何も出力しない) から 3 (最も多く出力する) までの値に設定することができます。デバッグ情報は `sys.stdout` に出力されます。

ZipFile.**comment**

ZIP ファイルに *bytes* オブジェクトとして関連付けられたコメントです。モード `'w'` 、`'x'` または `'a'` で作成された *ZipFile* インスタンスへコメントを割り当てる場合、文字列長は 65535 バイトまでにしてください。その長さを超えたコメントは切り捨てられます。

13.5.2 Path オブジェクト

class zipfile.**Path**(*root*, *at=''*)

Construct a Path object from a `root` zipfile (which may be a *ZipFile* instance or `file` suitable for passing to the *ZipFile* constructor).

`at` specifies the location of this Path within the zipfile, e.g. 'dir/file.txt', 'dir/', or ''. Defaults to the empty string, indicating the root.

Path objects expose the following features of *pathlib.Path* objects:

Path objects are traversable using the / operator.

Path.name
The final path component.

Path.open(*, **)
Invoke *ZipFile.open()* on the current path. Accepts the same arguments as *ZipFile.open()*.

> **ご用心:** The signature on this function changes in an incompatible way in Python 3.9. For a future-compatible version, consider using the third-party zipp.Path package (3.0 or later).

Path.iterdir()
Enumerate the children of the current directory.

Path.is_dir()
Return **True** if the current context references a directory.

Path.is_file()
Return **True** if the current context references a file.

Path.exists()
Return **True** if the current context references a file or directory in the zip file.

Path.read_text(*, **)
Read the current file as unicode text. Positional and keyword arguments are passed through to *io.TextIOWrapper* (except **buffer**, which is implied by the context).

Path.read_bytes()
Read the current file as bytes.

13.5.3 PyZipFile オブジェクト

PyZipFile コンストラクタは *ZipFile* コンストラクタと同じパラメータに加え、*optimize* パラメータをとります。

class zipfile.PyZipFile(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *optimize=-1*)
バージョン 3.2 で追加: パラメータに *optimize* を追加しました。

バージョン 3.4 で変更: ZIP64 拡張がデフォルトで有効になりました。

インスタンスは *ZipFile* オブジェクトのメソッドの他に、追加のメソッドを 1 個持ちます:

writepy(*pathname*, *basename=''*, *filterfunc=None*)
*.py ファイルを探し、一致するファイルをアーカイブに追加します。

PyZipFile に *optimize* 引数が与えられない場合、あるいは -1 が指定された場合、対応するファイルは *.pyc ファイルで、必要に応じてコンパイルします。

PyZipFile の *optimize* パラメータが 0、1、あるいは 2 の場合、それを最適化レベル (*compile()* 参照) とするファイルのみが、必要に応じてコンパイルされアーカイブに追加されます。

If *pathname* is a file, the filename must end with .py, and just the (corresponding *.pyc) file is added at the top level (no path information). If *pathname* is a file that does not end with .py, a *RuntimeError* will be raised. If it is a directory, and the directory is not a package directory, then all the files *.pyc are added at the top level. If the directory is a package directory, then all *.pyc are added under the package name as a file path, and if any subdirectories are package directories, all of these are added recursively in sorted order.

basename は内部が使用するためだけのものです。

filterfunc を与える場合、単一の文字列引数を取る関数を渡してください。これには (個々のフルパスを含む) それぞれのパスがアーカイブに加えられる前に渡されます。*filterfunc* が偽を返せば、そのパスはアーカイブに追加されず、ディレクトリだった場合はその中身が無視されます。例として、私たちのテストファイルが全て test ディレクトリの中にあるか、test 文字列で始まるとしましょう。*filterfunc* を使ってそれらを除外出来ます:

```
>>> zf = PyZipFile('myprog.zip')
>>> def notests(s):
...     fn = os.path.basename(s)
...     return (not (fn == 'test' or fn.startswith('test_')))
>>> zf.writepy('myprog', filterfunc=notests)
```

writepy() メソッドは以下のようなファイル名でアーカイブを作成します:

```
string.pyc                  # Top level name
test/__init__.pyc           # Package directory
test/testall.pyc            # Module test.testall
test/bogus/__init__.pyc     # Subpackage directory
test/bogus/myfile.pyc       # Submodule test.bogus.myfile
```

バージョン 3.4 で追加: *filterfunc* パラメータ。

バージョン 3.6.2 で変更: *pathname* 引数が *path-like object* を受け付けるようになりました。

バージョン 3.7 で変更: Recursion sorts directory entries.

13.5.4 ZipInfo オブジェクト

ZipInfo クラスのインスタンスは、*ZipFile* オブジェクトの *getinfo()* および *infolist()* メソッドによって返されます。各オブジェクトは ZIP アーカイブ内の 1 個のメンバに関する情報を格納します。

There is one classmethod to make a *ZipInfo* instance for a filesystem file:

classmethod ZipInfo.**from_file**(*filename*, *arcname=None*, ***, *strict_timestamps=True*)

Construct a *ZipInfo* instance for a file on the filesystem, in preparation for adding it to a zip file.

filename should be the path to a file or directory on the filesystem.

If *arcname* is specified, it is used as the name within the archive. If *arcname* is not specified, the name will be the same as *filename*, but with any drive letter and leading path separators removed.

The *strict_timestamps* argument, when set to `False`, allows to zip files older than 1980-01-01 at the cost of setting the timestamp to 1980-01-01. Similar behavior occurs with files newer than 2107-12-31, the timestamp is also set to the limit.

バージョン 3.6 で追加.

バージョン 3.6.2 で変更: *filename* 引数が *path-like object* を受け入れるようになりました。

バージョン 3.8 で追加: *strict_timestamps* キーワード専用引数。

インスタンスは以下のメソッドと属性を持ちます:

ZipInfo.**is_dir**()

アーカイブのメンバーがディレクトリの場合に “True“を返します。

This uses the entry's name: directories should always end with `/`.

バージョン 3.6 で追加.

ZipInfo.**filename**

アーカイブ中のファイル名。

ZipInfo.**date_time**

アーカイブメンバの最終更新日時。6 つの値からなるタプルになります:

インデックス	値
0	西暦年 (>= 1980)
1	月 (1 から始まる)
2	日 (1 から始まる)
3	時 (0 から始まる)
4	分 (0 から始まる)
5	秒 (0 から始まる)

注釈: ZIP ファイルフォーマットは 1980 年より前のタイムスタンプをサポートしていません。

ZipInfo.compress_type
アーカイブメンバの圧縮形式。

ZipInfo.comment
Comment for the individual archive member as a *bytes* object.

ZipInfo.extra
拡張フィールドデータ。この *bytes* オブジェクトに含まれているデータの内部構成については、PKZIP Application Note でコメントされています。

ZipInfo.create_system
ZIP アーカイブを作成したシステムを記述する文字列。

ZipInfo.create_version
このアーカイブを作成した PKZIP のバージョン。

ZipInfo.extract_version
このアーカイブを展開する際に必要な PKZIP のバージョン。

ZipInfo.reserved
予約領域。ゼロでなくてはなりません。

ZipInfo.flag_bits
ZIP フラグビット列。

ZipInfo.volume
ファイルヘッダのボリューム番号。

ZipInfo.internal_attr
内部属性。

ZipInfo.external_attr
外部ファイル属性。

ZipInfo.header_offset
ファイルヘッダへのバイトオフセット。

ZipInfo.CRC
圧縮前のファイルの CRC-32 チェックサム。

ZipInfo.compress_size
圧縮後のデータのサイズ。

ZipInfo.file_size
圧縮前のファイルのサイズ。

13.5.5 コマンドラインインターフェイス

zipfile モジュールは、ZIP アーカイブを操作するための簡単なコマンドラインインターフェースを提供しています。

ZIP アーカイブを新規に作成したい場合、*-c* オプションの後にまとめたいファイルを列挙してください:

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

ディレクトリを渡すこともできます:

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

ZIP アーカイブを特定のディレクトリに展開したい場合、*-e* オプションを使用してください:

```
$ python -m zipfile -e monty.zip target-dir/
```

ZIP アーカイブ内のファイル一覧を表示するには *-l* を使用してください:

```
$ python -m zipfile -l monty.zip
```

コマンドラインオプション

-l <zipfile>
--list <zipfile>
　zipfile 内のファイル一覧を表示します。

-c <zipfile> <source1> ... <sourceN>
--create <zipfile> <source1> ... <sourceN>
　ソースファイルから zipfile を作成します。

-e <zipfile> <output_dir>
--extract <zipfile> <output_dir>
　zipfile を対象となるディレクトリに展開します。

-t <zipfile>
--test <zipfile>
　zipfile が有効かどうか調べます。

13.5.6 Decompression pitfalls

The extraction in zipfile module might fail due to some pitfalls listed below.

From file itself

Decompression may fail due to incorrect password / CRC checksum / ZIP format or unsupported compression method / decryption.

File System limitations

Exceeding limitations on different file systems can cause decompression failed. Such as allowable characters in the directory entries, length of the file name, length of the pathname, size of a single file, and number of files, etc.

Resources limitations

The lack of memory or disk volume would lead to decompression failed. For example, decompression bombs (aka ZIP bomb) apply to zipfile library that can cause disk volume exhaustion.

Interruption

Interruption during the decompression, such as pressing control-C or killing the decompression process may result in incomplete decompression of the archive.

Default behaviors of extraction

Not knowing the default extraction behaviors can cause unexpected decompression results. For example, when extracting the same archive twice, it overwrites files without asking.

13.6 `tarfile` --- tar アーカイブファイルの読み書き

ソースコード: Lib/tarfile.py

---

tarfile モジュールは、gzip、bz2、および lzma 圧縮されたものを含む、tar アーカイブを読み書きできます。`.zip` ファイルの読み書きには *zipfile* モジュールか、あるいは *shutil* の高水準関数を使用してください。

いくつかの事実と形態:

- モジュールが利用可能な場合、*gzip*、*bz2* ならびに *lzma* で圧縮されたアーカイブを読み書きします。
- POSIX.1-1988 (ustar) フォーマットの読み書きをサポートしています。

- *longname* および *longlink* 拡張を含む GNU tar フォーマットの読み書きをサポートしています。スパースファイルの復元を含む *sparse* 拡張は読み込みのみサポートしています。
- POSIX.1-2001 (pax) フォーマットの読み書きをサポートしています。
- ディレクトリ、一般ファイル、ハードリンク、シンボリックリンク、fifo、キャラクターデバイスおよびブロックデバイスを処理します。また、タイムスタンプ、アクセス許可や所有者のようなファイル情報の取得および保存が可能です。

バージョン 3.3 で変更: *lzma* 圧縮をサポートしました。

`tarfile.open`(*name=None*, *mode='r'*, *fileobj=None*, *bufsize=10240*, ***kwargs*)

パス名 *name* の *TarFile* オブジェクトを返します。*TarFile* オブジェクトと、利用できるキーワード引数に関する詳細な情報については、*TarFile* **オブジェクト** 節を参照してください。

mode は `'filemode[:compression]'` の形式をとる文字列でなければなりません。デフォルトの値は `'r'` です。以下に *mode* のとりうる組み合わせすべてを示します:

mode	action
`'r'` または `'r:*'`	圧縮方法に関して透過的に、読み込み用にオープンします (推奨)。
`'r:'`	非圧縮で読み込み用に排他的にオープンします。
`'r:gz'`	gzip 圧縮で読み込み用にオープンします。
`'r:bz2'`	bzip2 圧縮で読み込み用にオープンします。
`'r:xz'`	lzma 圧縮で読み込み用にオープンします。
`'x'` or `'x:'`	圧縮せずに tarfile を排他的に作成します。tarfile が既存の場合 *FileExistsError* 例外を送出します。
`'x:gz'`	gzip 圧縮で tarfile を作成します。tarfile が既存の場合 *FileExistsError* 例外を送出します。
`'x:bz2'`	bzip2 圧縮で tarfile を作成します。tarfile が既存の場合 *FileExistsError* 例外を送出します。
`'x:xz'`	lzma 圧縮で tarfile を作成します。tarfile が既存の場合 *FileExistsError* 例外を送出します。
`'a'` または `'a:'`	非圧縮で追記用にオープンします。ファイルが存在しない場合は新たに作成されます。
`'w'` または `'w:'`	非圧縮で書き込み用にオープンします。
`'w:gz'`	gzip 圧縮で書き込み用にオープンします。
`'w:bz2'`	bzip2 圧縮で書き込み用にオープンします。
`'w:xz'`	lzma 圧縮で書き込み用にオープンします。

`'a:gz'`、`'a:bz2'`、`'a:xz'` は利用できないことに注意して下さい。もし *mode* が、ある (圧縮した) ファイルを読み込み用にオープンするのに適していないなら、*ReadError* が送出されます。これを防ぐには *mode* `'r'` を使って下さい。もし圧縮方式がサポートされていなければ、*CompressionError* が送出されます。

もし *fileobj* が指定されていれば、それは *name* でバイナリモードでオープンされた **ファイルオブジェクト** の代替として使うことができます。そのファイルオブジェクトの位置が 0 であることを前提に動作します。

`'w:gz'`、`'r:gz'`、`'w:bz2'`、`'r:bz2'`, `'x:gz'`, `'x:bz2'` モードの場合、*tarfile.open()* はファイルの圧縮レベルを指定するキーワード引数 *compresslevel* (デフォルトは 9) を受け付けます。

特別な目的のために、*mode* には 2 番目の形式: '**ファイルモード**|[**圧縮**]' があります。この形式を使うと、*tarfile.open()* が返すのは、データをブロックからなるストリームとして扱う *TarFile* オブジェクトになります。この場合、ファイルに対してランダムなシークが行えなくなります。*fileobj* を指定する場合、`read()` および `write()` メソッドを持つ (*mode* に依存した) 任意のオブジェクトにできます。*bufsize* にはブロックサイズを指定します。デフォルトは `20 * 512` バイトです。`sys.stdin`、ソケット *file object*、あるいはテープデバイスと組み合わせる場合にはこの形式を使ってください。ただし、このような *TarFile* オブジェクトにはランダムアクセスを行えないという制限があります。**使用例** 節を参照してください。現在可能なモードは以下のとおりです。

モード	動作
`'r|*'`	tar ブロックの *stream* を圧縮方法に関して透過的に読み込み用にオープンします。
`'r|'`	非圧縮 tar ブロックの *stream* を読み込み用にオープンします。
`'r|gz'`	gzip 圧縮の *stream* を読み込み用にオープンします。
`'r|bz2'`	bzip2 圧縮の *stream* を読み込み用にオープンします。
`'r|xz'`	lzma 圧縮の *stream* を読み込み用にオープンします。
`'w|'`	非圧縮の *stream* を書き込み用にオープンします。
`'w|gz'`	gzip 圧縮の *stream* を書き込み用にオープンします。
`'w|bz2'`	bzip2 圧縮の *stream* を書き込み用にオープンします。
`'w|xz'`	lzma 圧縮の *stream* を書き込み用にオープンします。

バージョン 3.5 で変更: `'x'` (排他的作成) モードが追加されました。

バージョン 3.6 で変更: *name* パラメタが *path-like object* を受け付けるようになりました。

class tarfile.TarFile

tar アーカイブを読み書きするためのクラスです。このクラスを直接使わないこと: 代わりに *tarfile.open()* を使ってください。*TarFile* **オブジェクト** を参照してください。

tarfile.is_tarfile(*name*)

もし *name* が tar アーカイブファイルであり、*tarfile* モジュールで読み込める場合に *True* を返します。

tarfile モジュールは以下の例外を定義しています:

exception tarfile.TarError

すべての *tarfile* 例外のための基本クラスです。

exception tarfile.ReadError

tar アーカイブがオープンされた時、*tarfile* モジュールで操作できないか、あるいは何か無効であるとき送出されます。

exception tarfile.CompressionError
圧縮方法がサポートされていないか、あるいはデータを正しくデコードできない時に送出されます。

exception tarfile.StreamError
ストリームのような *TarFile* オブジェクトで典型的な制限のために送出されます。

exception tarfile.ExtractError
TarFile.extract() を使った時に **致命的でない** エラーに対して送出されます。ただし *TarFile.errorlevel*== 2 の場合に限ります。

exception tarfile.HeaderError
TarInfo.frombuf() メソッドが取得したバッファーが不正だったときに送出されます。

exception tarfile.FilterError
Base class for members *refused* by filters.

tarinfo
Information about the member that the filter refused to extract, as *TarInfo*.

exception tarfile.AbsolutePathError
Raised to refuse extracting a member with an absolute path.

exception tarfile.OutsideDestinationError
Raised to refuse extracting a member outside the destination directory.

exception tarfile.SpecialFileError
Raised to refuse extracting a special file (e.g. a device or pipe).

exception tarfile.AbsoluteLinkError
Raised to refuse extracting a symbolic link with an absolute path.

exception tarfile.LinkOutsideDestinationError
Raised to refuse extracting a symbolic link pointing outside the destination directory.

モジュールレベルで以下の定数が利用できます。

tarfile.ENCODING
既定の文字エンコーディング。Windows では `'utf-8'` 、それ以外では *sys.getfilesystemencoding()* の返り値です。

以下の各定数は、*tarfile* モジュールが作成できる tar アーカイブフォーマットを定義しています。詳細は、**サポートしている** *tar* **フォーマット** を参照してください。

tarfile.USTAR_FORMAT
POSIX.1-1988 (ustar) フォーマット。

tarfile.GNU_FORMAT
GNU tar フォーマット。

tarfile.PAX_FORMAT
POSIX.1-2001 (pax) フォーマット。

tarfile.DEFAULT_FORMAT
: アーカイブを作成する際のデフォルトのフォーマット。現在は *PAX_FORMAT* です。

  バージョン 3.8 で変更: 新しいアーカイブのデフォルトフォーマットが *GNU_FORMAT* から *PAX_FORMAT* に変更されました。

参考:

zipfile **モジュール** *zipfile* 標準モジュールのドキュメント。

アーカイブ化操作 *shutil* が提供するより高水準のアーカイブ機能についてのドキュメント。

GNU tar manual, Basic Tar Format GNU tar 拡張機能を含む、tar アーカイブファイルのためのドキュメント。

13.6.1 TarFile オブジェクト

TarFile オブジェクトは、tar アーカイブへのインターフェースを提供します。tar アーカイブは一連のブロックです。アーカイブメンバー (保存されたファイル) は、ヘッダーブロックとそれに続くデータブロックで構成されています。一つの tar アーカイブにファイルを何回も保存することができます。各アーカイブメンバーは、*TarInfo* オブジェクトで確認できます。詳細については *TarInfo* **オブジェクト** を参照してください。

TarFile オブジェクトは `with` 文のコンテキストマネージャーとして利用できます。with ブロックが終了したときにオブジェクトはクローズされます。例外が発生した時、内部で利用されているファイルオブジェクトのみがクローズされ、書き込み用にオープンされたアーカイブのファイナライズは行われないことに注意してください。**使用例** 節のユースケースを参照してください。

バージョン 3.2 で追加: コンテキスト管理のプロトコルがサポートされました。

class tarfile.TarFile(*name=None*, *mode='r'*, *fileobj=None*, *format=DEFAULT_FORMAT*, *tarinfo=TarInfo*, *dereference=False*, *ignore_zeros=False*, *encoding=ENCODING*, *errors='surrogateescape'*, *pax_headers=None*, *debug=0*, *errorlevel=1*)
: 以下のすべての引数はオプションで、インスタンス属性としてもアクセスできます。

  name はアーカイブのパス名です。*name* は *path-like object* でも構いません。*fileobj* が渡された場合は省略可能です。その場合、ファイルオブジェクトに `name` 属性があれば、それを利用します。

  mode は、既存のアーカイブから読み込むための `'r'`、既存のアーカイブに追記するための `'a'`、既存のファイルがあれば上書きして新しいファイルを作成する `'w'` 、あるいは存在しない場合にのみ新しいファイルを作成する `'x'` のいずれかです。

  fileobj が与えられていれば、それを使ってデータを読み書きします。もしそれが決定できれば、*mode* は *fileobj* のモードで上書きされます。*fileobj* は位置 0 から利用されます。

  注釈: *TarFile* をクローズした時、*fileobj* はクローズされません。

format はアーカイブの書き込みフォーマットを制御します。モジュールレベルで定義されている、*USTAR_FORMAT*、*GNU_FORMAT*、あるいは *PAX_FORMAT* のいずれかである必要があります。読み出しのときは、1 つのアーカイブに異なるフォーマットが混在していたとしても、フォーマットは自動的に検知されます。

tarinfo 引数を利用して、デフォルトの *TarInfo* クラスを別のクラスで置き換えることができます。

dereference が *False* だった場合、シンボリックリンクやハードリンクがアーカイブに追加されます。*True* だった場合、リンクのターゲットとなるファイルの内容がアーカイブに追加されます。シンボリックリンクをサポートしていないシステムでは効果がありません。

ignore_zeros が *False* だった場合、空ブロックをアーカイブの終端として扱います。*True* だった場合、空の (無効な) ブロックをスキップして、可能な限り多くのメンバーを取得しようとします。このオプションは、連結されたり、壊れたアーカイブファイルを扱うときにのみ、意味があります。

debug は 0 (デバッグメッセージ無し) から 3 (全デバッグメッセージ) まで設定できます。このメッセージは `sys.stderr` に書き込まれます。

errorlevel controls how extraction errors are handled, see `the corresponding attribute`.

引数 *encoding* および *errors* にはアーカイブの読み書きやエラー文字列の変換に使用する文字エンコーディングを指定します。ほとんどのユーザーはデフォルト設定のままで動作します。詳細に関しては *Unicode* **に関する問題** 節を参照してください。

引数 *pax_headers* は、オプションの文字列辞書で、*format* が *PAX_FORMAT* だった場合に pax グローバルヘッダーに追加されます。

バージョン 3.2 で変更: 引数 *errors* のデフォルトが `'surrogateescape'` になりました。

バージョン 3.5 で変更: `'x'` (排他的作成) モードが追加されました。

バージョン 3.6 で変更: *name* パラメタが *path-like object* を受け付けるようになりました。

`classmethod TarFile.open(...)`
代替コンストラクターです。モジュールレベルでの *tarfile.open()* 関数は、実際はこのクラスメソッドへのショートカットです。

`TarFile.getmember(name)`
メンバー *name* に対する *TarInfo* オブジェクトを返します。*name* がアーカイブに見つからなければ、*KeyError* が送出されます。

注釈: アーカイブ内にメンバーが複数ある場合は、最後に出現するものが最新のバージョンとみなされます。

`TarFile.getmembers()`
TarInfo アーカイブのメンバーをオブジェクトのリストとして返します。このリストはアーカイブ内のメンバーと同じ順番です。

TarFile.**getnames**()
メンバーをその名前のリストを返します。これは *getmembers()* で返されるリストと同じ順番です。

TarFile.**list**(*verbose=True*, *, *members=None*)
内容の一覧を sys.stdout に出力します。*verbose* が *False* の場合、メンバー名のみ表示します。*True* の場合、**ls -l** に似た出力を生成します。オプションの *members* を与える場合、*getmembers()* が返すリストのサブセットである必要があります。

バージョン 3.5 で変更: *members* 引数が追加されました。.

TarFile.**next**()
TarFile が読み込み用にオープンされている時、アーカイブの次のメンバーを *TarInfo* オブジェクトとして返します。もしそれ以上利用可能なものがなければ、*None* を返します。

TarFile.**extractall**(*path="."*, *members=None*, *, *numeric_owner=False*, *filter=None*)
すべてのメンバーをアーカイブから現在の作業ディレクトリまたは *path* に抽出します。オプションの *members* が与えられるときには、*getmembers()* で返されるリストの一部でなければなりません。所有者、変更時刻、アクセス権限のようなディレクトリ情報はすべてのメンバーが抽出された後にセットされます。これは二つの問題を回避するためです。一つはディレクトリの変更時刻はその中にファイルが作成されるたびにリセットされるということ、もう一つはディレクトリに書き込み許可がなければその中のファイル抽出は失敗してしまうということです。

numeric_owner が *True* の場合、tarfile の uid と gid 数値が抽出されたファイルのオーナー/グループを設定するために使用されます。False の場合、tarfile の名前付きの値が使用されます。

The *filter* argument, which was added in Python 3.8.17, specifies how **members** are modified or rejected before extraction. See *Extraction filters* for details. It is recommended to set this explicitly depending on which *tar* features you need to support.

> **警告:** 内容を信頼できない tar アーカイブを、事前の内部チェック前に展開してはいけません。ファイルが *path* の外側に作られる可能性があります。例えば、"/" で始まる絶対パスのファイル名や、2 重ドット ".." で始まるパスのファイル名です。
>
> Set `filter='data'` to prevent the most dangerous security issues, and read the *Extraction filters* section for details.

バージョン 3.5 で変更: *numeric_owner* 引数が追加されました。

バージョン 3.6 で変更: *path* パラメタが *path-like object* を受け付けるようになりました。

バージョン 3.8.17 で変更: *filter* パラメータが追加されました。

TarFile.**extract**(*member*, *path=""*, *set_attrs=True*, *, *numeric_owner=False*, *filter=None*)
アーカイブからメンバーの完全な名前を使って、現在のディレクトリに展開します。ファイル情報はできる限り正確に展開されます。*member* はファイル名もしくは *TarInfo* オブジェクトです。*path* を使って別のディレクトリを指定することもできます。*path* は *path-like object* でも構いません。*set_attrs* が false でない限り、ファイルの属性 (所有者、最終更新時刻、モード) は設定されます。

The *numeric_owner* and *filter* arguments are the same as for *extractall()*.

注釈: *extract()* メソッドはいくつかの展開に関する問題を扱いません。ほとんどの場合、*extractall()* メソッドの利用を考慮するべきです。

警告: *extractall()* の警告を参してください。

Set `filter='data'` to prevent the most dangerous security issues, and read the *Extraction filters* section for details.

バージョン 3.2 で変更: パラメーターに *set_attrs* を追加しました。

バージョン 3.5 で変更: *numeric_owner* 引数が追加されました。

バージョン 3.6 で変更: *path* パラメタが *path-like object* を受け付けるようになりました。

バージョン 3.8.17 で変更: *filter* パラメータが追加されました。

`TarFile.extractfile(`*member*`)`

アーカイブからメンバーをファイルオブジェクトとして抽出します。*member* はファイル名でも *TarInfo* オブジェクトでも構いません。*member* が一般ファイルまたはリンクの場合、*io.BufferedReader* オブジェクトが返されます。それ以外の場合、*None* が返されます。

バージョン 3.3 で変更: 戻り値が *io.BufferedReader* オブジェクトになりました。

`TarFile.errorlevel: int`

If *errorlevel* is `0`, errors are ignored when using *TarFile.extract()* and *TarFile.extractall()*. Nevertheless, they appear as error messages in the debug output when *debug* is greater than 0. If `1` (the default), all *fatal* errors are raised as *OSError* or *FilterError* exceptions. If `2`, all *non-fatal* errors are raised as *TarError* exceptions as well.

Some exceptions, e.g. ones caused by wrong argument types or data corruption, are always raised.

Custom *extraction filters* should raise *FilterError* for *fatal* errors and *ExtractError* for *non-fatal* ones.

Note that when an exception is raised, the archive may be partially extracted. It is the user's responsibility to clean up.

`TarFile.extraction_filter`

バージョン 3.8.17 で追加.

The *extraction filter* used as a default for the *filter* argument of *extract()* and *extractall()*.

The attribute may be `None` or a callable. String names are not allowed for this attribute, unlike the *filter* argument to *extract()*.

If `extraction_filter` is `None` (the default), calling an extraction method without a *filter* argument will use the *fully_trusted* filter for compatibility with previous Python versions.

In Python 3.12+, leaving `extraction_filter=None` will emit a `DeprecationWarning`.

In Python 3.14+, leaving `extraction_filter=None` will cause extraction methods to use the *data* filter by default.

The attribute may be set on instances or overridden in subclasses. It also is possible to set it on the `TarFile` class itself to set a global default, although, since it affects all uses of *tarfile*, it is best practice to only do so in top-level applications or *site configuration*. To set a global default this way, a filter function needs to be wrapped in *staticmethod()* to prevent injection of a `self` argument.

`TarFile.add`(*name*, *arcname=None*, *recursive=True*, ***, *filter=None*)

ファイル *name* をアーカイブに追加します。*name* は、任意のファイルタイプ (ディレクトリ、fifo、シンボリックリンク等) です。*arcname* が与えられている場合は、それはアーカイブ内のファイルの代替名を指定します。デフォルトではディレクトリは再帰的に追加されます。これは、*recursive* を *False* に設定すると避けられます。再帰処理はソートされた順序でエントリーを追加します。*filter* が与えられた場合、それは *TarInfo* オブジェクトを引数として受け取り、操作した *TarInfo* オブジェクトを返す関数でなければなりません。代わりに *None* を返した場合、*TarInfo* オブジェクトはアーカイブから除外されます。**使用例** にある例を参照してください。

バージョン 3.2 で変更: *filter* パラメータが追加されました。

バージョン 3.7 で変更: 再帰処理はソートされた順序でエントリーを追加するようになりました。

`TarFile.addfile`(*tarinfo*, *fileobj=None*)

TarInfo オブジェクト *tarinfo* をアーカイブに追加します。*fileobj* を与える場合、*binary file* にしなければならず、`tarinfo.size` バイトがそれから読まれ、アーカイブに追加されます。*TarInfo* オブジェクトを直接作成するか、*gettarinfo()* を使って作成することができます。

`TarFile.gettarinfo`(*name=None*, *arcname=None*, *fileobj=None*)

os.stat() の結果か、既存のファイルに相当するものから、*TarInfo* オブジェクトを作成します。このファイルは、*name* で名付けられるか、ファイル記述子を持つ *file object* *fileobj* として指定されます。*name* は:term:*path-like object* でも構いません。*arcname* が与えられた場合、アーカイブ内のファイルに対して代替名を指定します。与えられない場合、名前は *fileobj* の *name* 属性 *name* 属性から取られます。名前はテキスト文字列にしてください。

TarInfo の属性の一部は、*addfile()* を使用して追加する前に修正できます。ファイルオブジェクトが、ファイルの先頭にある通常のファイルオブジェクトでない場合、*size* などの属性は修正が必要かもしれません。これは、*GzipFile* などの属性に当てはまります。*name* も修正できるかもしれず、この場合、*arcname* はダミーの文字列にすることができます。

バージョン 3.6 で変更: *name* パラメタが *path-like object* を受け付けるようになりました。

`TarFile.close`()

TarFile をクローズします。書き込みモードでは、完了ゼロブロックが 2 個アーカイブに追加されます。

TarFile.**pax_headers**
pax グローバルヘッダーに含まれる key-value ペアの辞書です。

13.6.2 TarInfo オブジェクト

TarInfo オブジェクトは *TarFile* の一つのメンバーを表します。ファイルに必要なすべての属性 (ファイルタイプ、ファイルサイズ、時刻、アクセス権限、所有者等のような) を保存する他に、そのタイプを決定するのに役に立ついくつかのメソッドを提供します。これにはファイルのデータそのものは **含まれません** 。

TarInfo objects are returned by *TarFile*'s methods *getmember()*, *getmembers()* and *gettarinfo()*.

Modifying the objects returned by `getmember()` or `getmembers()` will affect all subsequent operations on the archive. For cases where this is unwanted, you can use *copy.copy()* or call the `replace()` method to create a modified copy in one step.

Several attributes can be set to `None` to indicate that a piece of metadata is unused or unknown. Different *TarInfo* methods handle `None` differently:

- The *extract()* or *extractall()* methods will ignore the corresponding metadata, leaving it set to a default.
- *addfile()* will fail.
- *list()* will print a placeholder string.

バージョン 3.8.17 で変更: Added `replace()` and handling of `None`.

class tarfile.**TarInfo**(*name=""*)
TarInfo オブジェクトを作成します。

classmethod TarInfo.**frombuf**(*buf*, *encoding*, *errors*)
TarInfo オブジェクトを文字列バッファー *buf* から作成して返します。

バッファーが不正な場合 *HeaderError* を送出します。

classmethod TarInfo.**fromtarfile**(*tarfile*)
TarFile オブジェクトの *tarfile* から、次のメンバーを読み込んで、それを *TarInfo* オブジェクトとして返します。

TarInfo.**tobuf**(*format=DEFAULT_FORMAT*, *encoding=ENCODING*, *errors='surrogateescape'*)
TarInfo オブジェクトから文字列バッファーを作成します。引数についての情報は、*TarFile* クラスのコンストラクターを参照してください。

バージョン 3.2 で変更: 引数 *errors* のデフォルトが `'surrogateescape'` になりました。

`TarInfo` オブジェクトには以下のデータ属性があります:

TarInfo.**name**: **str**
アーカイブメンバーの名前。

TarInfo.size: int
バイト単位でのサイズ。

TarInfo.mtime: int | float
Time of last modification in seconds since the *epoch*, as in *os.stat_result.st_mtime*.

バージョン 3.8.17 で変更: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.mode: int
Permission bits, as for *os.chmod()*.

バージョン 3.8.17 で変更: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.type
ファイルタイプ。*type* は通常、定数 REGTYPE、AREGTYPE、LNKTYPE、SYMTYPE、DIRTYPE、FIFOTYPE、CONTTYPE、CHRTYPE、BLKTYPE、あるいは GNUTYPE_SPARSE のいずれかです。*TarInfo* オブジェクトのタイプをもっと簡単に解決するには、下記の is*() メソッドを使って下さい。

TarInfo.linkname: str
リンク先ファイルの名前。これはタイプ LNKTYPE と SYMTYPE の *TarInfo* オブジェクトにだけ存在します。

For symbolic links (SYMTYPE), the *linkname* is relative to the directory that contains the link. For hard links (LNKTYPE), the *linkname* is relative to the root of the archive.

TarInfo.uid: int
ファイルメンバーを保存した元のユーザーのユーザー ID。

バージョン 3.8.17 で変更: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.gid: int
ファイルメンバーを保存した元のユーザーのグループ ID。

バージョン 3.8.17 で変更: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.uname: str
ファイルメンバーを保存した元のユーザーのユーザー名。

バージョン 3.8.17 で変更: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.gname: str
ファイルメンバーを保存した元のユーザーのグループ名。

バージョン 3.8.17 で変更: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.pax_headers: dict

pax 拡張ヘッダーに関連付けられた、key-value ペアの辞書。

TarInfo.replace(name=..., mtime=..., mode=..., linkname=..., uid=..., gid=..., uname=..., gname=..., deep=True)

バージョン 3.8.17 で追加.

Return a *new* copy of the `TarInfo` object with the given attributes changed. For example, to return a `TarInfo` with the group name set to `'staff'`, use:

```
new_tarinfo = old_tarinfo.replace(gname='staff')
```

By default, a deep copy is made. If *deep* is false, the copy is shallow, i.e. `pax_headers` and any custom attributes are shared with the original `TarInfo` object.

TarInfo オブジェクトは便利な照会用のメソッドもいくつか提供しています:

TarInfo.isfile()

Tarinfo オブジェクトが一般ファイルの場合に、*True* を返します。

TarInfo.isreg()

isfile() と同じです。

TarInfo.isdir()

ディレクトリの場合に *True* を返します。

TarInfo.issym()

シンボリックリンクの場合に *True* を返します。

TarInfo.islnk()

ハードリンクの場合に *True* を返します。

TarInfo.ischr()

キャラクターデバイスの場合に *True* を返します。

TarInfo.isblk()

ブロックデバイスの場合に *True* を返します。

TarInfo.isfifo()

FIFO の場合に *True* を返します。

TarInfo.isdev()

キャラクターデバイス、ブロックデバイスあるいは FIFO のいずれかの場合に *True* を返します。

13.6.3 Extraction filters

バージョン 3.8.17 で追加.

The *tar* format is designed to capture all details of a UNIX-like filesystem, which makes it very powerful. Unfortunately, the features make it easy to create tar files that have unintended -- and possibly malicious -- effects when extracted. For example, extracting a tar file can overwrite arbitrary files in various ways (e.g. by using absolute paths, `..` path components, or symlinks that affect later members).

In most cases, the full functionality is not needed. Therefore, *tarfile* supports extraction filters: a mechanism to limit functionality, and thus mitigate some of the security issues.

参考:

PEP 706 Contains further motivation and rationale behind the design.

The *filter* argument to *TarFile.extract()* or *extractall()* can be:

- the string `'fully_trusted'`: Honor all metadata as specified in the archive. Should be used if the user trusts the archive completely, or implements their own complex verification.
- the string `'tar'`: Honor most *tar*-specific features (i.e. features of UNIX-like filesystems), but block features that are very likely to be surprising or malicious. See *tar_filter()* for details.
- the string `'data'`: Ignore or block most features specific to UNIX-like filesystems. Intended for extracting cross-platform data archives. See *data_filter()* for details.
- `None` (default): Use *TarFile.extraction_filter*.

  If that is also `None` (the default), the `'fully_trusted'` filter will be used (for compatibility with earlier versions of Python).

  In Python 3.12, the default will emit a `DeprecationWarning`.

  In Python 3.14, the `'data'` filter will become the default instead. It's possible to switch earlier; see *TarFile.extraction_filter*.

- A callable which will be called for each extracted member with a *TarInfo* describing the member and the destination path to where the archive is extracted (i.e. the same path is used for all members):

  ```
  filter(/, member: TarInfo, path: str) -> TarInfo | None
  ```

  The callable is called just before each member is extracted, so it can take the current state of the disk into account. It can:

  – return a *TarInfo* object which will be used instead of the metadata in the archive, or

  – return `None`, in which case the member will be skipped, or

  – raise an exception to abort the operation or skip the member, depending on *errorlevel*. Note that when extraction is aborted, *extractall()* may leave the archive partially extracted. It

does not attempt to clean up.

Default named filters

The pre-defined, named filters are available as functions, so they can be reused in custom filters:

`tarfile.fully_trusted_filter`(*/*, *member*, *path*)

Return *member* unchanged.

This implements the `'fully_trusted'` filter.

`tarfile.tar_filter`(*/*, *member*, *path*)

Implements the `'tar'` filter.

- Strip leading slashes (`/` and *os.sep*) from filenames.
- *Refuse* to extract files with absolute paths (in case the name is absolute even after stripping slashes, e.g. `C:/foo` on Windows). This raises *AbsolutePathError*.
- *Refuse* to extract files whose absolute path (after following symlinks) would end up outside the destination. This raises *OutsideDestinationError*.
- Clear high mode bits (setuid, setgid, sticky) and group/other write bits (`S_IWOTH`).

Return the modified `TarInfo` member.

`tarfile.data_filter`(*/*, *member*, *path*)

Implements the `'data'` filter. In addition to what `tar_filter` does:

- *Refuse* to extract links (hard or soft) that link to absolute paths, or ones that link outside the destination.

  This raises *AbsoluteLinkError* or *LinkOutsideDestinationError*.

  Note that such files are refused even on platforms that do not support symbolic links.

- *Refuse* to extract device files (including pipes). This raises *SpecialFileError*.
- For regular files, including hard links:
  - Set the owner read and write permissions (`S_IWUSR`).
  - Remove the group & other executable permission (`S_IXOTH`) if the owner doesn't have it (*S_IXUSR*).
- For other files (directories), set `mode` to `None`, so that extraction methods skip applying permission bits.
- Set user and group info (`uid`, `gid`, `uname`, `gname`) to `None`, so that extraction methods skip setting it.

Return the modified `TarInfo` member.

Filter errors

When a filter refuses to extract a file, it will raise an appropriate exception, a subclass of *FilterError*. This will abort the extraction if *TarFile.errorlevel* is 1 or more. With `errorlevel=0` the error will be logged and the member will be skipped, but extraction will continue.

Hints for further verification

Even with `filter='data'`, *tarfile* is not suited for extracting untrusted files without prior inspection. Among other issues, the pre-defined filters do not prevent denial-of-service attacks. Users should do additional checks.

Here is an incomplete list of things to consider:

- Extract to a *new temporary directory* to prevent e.g. exploiting pre-existing links, and to make it easier to clean up after a failed extraction.
- When working with untrusted data, use external (e.g. OS-level) limits on disk, memory and CPU usage.
- Check filenames against an allow-list of characters (to filter out control characters, confusables, foreign path separators, etc.).
- Check that filenames have expected extensions (discouraging files that execute when you "click on them", or extension-less files like Windows special device names).
- Limit the number of extracted files, total size of extracted data, filename length (including symlink length), and size of individual files.
- Check for files that would be shadowed on case-insensitive filesystems.

Also note that:

- Tar files may contain multiple versions of the same file. Later ones are expected to overwrite any earlier ones. This feature is crucial to allow updating tape archives, but can be abused maliciously.
- *tarfile* does not protect against issues with "live" data, e.g. an attacker tinkering with the destination (or source) directory while extraction (or archiving) is in progress.

Supporting older Python versions

Extraction filters were added to Python 3.12, and are backported to older versions as security updates. To check whether the feature is available, use e.g. `hasattr(tarfile, 'data_filter')` rather than checking the Python version.

The following examples show how to support Python versions with and without the feature. Note that setting `extraction_filter` will affect any subsequent operations.

- Fully trusted archive:

```
my_tarfile.extraction_filter = (lambda member, path: member)
my_tarfile.extractall()
```

- Use the `'data'` filter if available, but revert to Python 3.11 behavior (`'fully_trusted'`) if this feature is not available:

```
my_tarfile.extraction_filter = getattr(tarfile, 'data_filter',
                                       (lambda member, path: member))
my_tarfile.extractall()
```

- Use the `'data'` filter; *fail* if it is not available:

```
my_tarfile.extractall(filter=tarfile.data_filter)
```

  or:

```
my_tarfile.extraction_filter = tarfile.data_filter
my_tarfile.extractall()
```

- Use the `'data'` filter; *warn* if it is not available:

```
if hasattr(tarfile, 'data_filter'):
    my_tarfile.extractall(filter='data')
else:
    # remove this when no longer needed
    warn_the_user('Extracting may be unsafe; consider updating Python')
    my_tarfile.extractall()
```

Stateful extraction filter example

While *tarfile*'s extraction methods take a simple *filter* callable, custom filters may be more complex objects with an internal state. It may be useful to write these as context managers, to be used like this:

```
with StatefulFilter() as filter_func:
    tar.extractall(path, filter=filter_func)
```

Such a filter can be written as, for example:

```
class StatefulFilter:
    def __init__(self):
        self.file_count = 0

    def __enter__(self):
        return self

    def __call__(self, member, path):
        self.file_count += 1
        return member
```

(次のページに続く)

(前のページからの続き)

```
    def __exit__(self, *exc_info):
        print(f'{self.file_count} files extracted')
```

13.6.4 コマンドラインインターフェイス

バージョン 3.4 で追加.

tarfile モジュールは、tar アーカイブを操作するための簡単なコマンドラインインターフェースを提供しています。

tar アーカイブを新規に作成したい場合、*-c* オプションの後にまとめたいファイル名のリストを指定してください:

```
$ python -m tarfile -c monty.tar  spam.txt eggs.txt
```

ディレクトリを渡すこともできます:

```
$ python -m tarfile -c monty.tar life-of-brian_1979/
```

tar アーカイブをカレントディレクトリに展開したい場合、*-e* オプションを使用してください:

```
$ python -m tarfile -e monty.tar
```

ディレクトリ名を渡すことで tar アーカイブを別のディレクトリに取り出すこともできます:

```
$ python -m tarfile -e monty.tar  other-dir/
```

tar アーカイブ内のファイル一覧を表示するには *-l* を使用してください:

```
$ python -m tarfile -l monty.tar
```

コマンドラインオプション

-l <tarfile>
--list <tarfile>
tarfile 内のファイル一覧を表示します。

-c <tarfile> <source1> ... <sourceN>
--create <tarfile> <source1> ... <sourceN>
ソースファイルから tarfile を作成します。

-e <tarfile> [<output_dir>]
--extract <tarfile> [<output_dir>]
output_dir が指定されていない場合、カレントディレクトリに tarfile を展開します。

-t <tarfile>

--test <tarfile>

tarfile が有効かどうか調べます。

-v, --verbose

詳細も出力します。

--filter <filtername>

Specifies the *filter* for `--extract`. See *Extraction filters* for details. Only string names are accepted (that is, `fully_trusted`, `tar`, and `data`).

バージョン 3.8.17 で追加.

13.6.5 使用例

tar アーカイブから現在のディレクトリにすべて抽出する方法:

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall()
tar.close()
```

tar アーカイブの一部を、リストの代わりにジェネレーター関数を利用して *TarFile.extractall()* で展開する方法:

```
import os
import tarfile

def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py":
            yield tarinfo

tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

非圧縮 tar アーカイブをファイル名のリストから作成する方法:

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

`with` 文を利用した同じ例:

```
import tarfile
with tarfile.open("sample.tar", "w") as tar:
```

(次のページに続く)

(前のページからの続き)

```
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

gzip 圧縮 tar アーカイブを作成してメンバー情報のいくつかを表示する方法:

```
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print(tarinfo.name, "is", tarinfo.size, "bytes in size and is ", end="")
    if tarinfo.isreg():
        print("a regular file.")
    elif tarinfo.isdir():
        print("a directory.")
    else:
        print("something else.")
tar.close()
```

TarFile.add() 関数の *filter* 引数を利用してユーザー情報をリセットしながらアーカイブを作成する方法:

```
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
    tarinfo.uname = tarinfo.gname = "root"
    return tarinfo
tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```

13.6.6 サポートしている tar フォーマット

tarfile モジュールは 3 種類の tar フォーマットを作成することができます:

- POSIX.1-1988 ustar format (*USTAR_FORMAT*). ファイル名の長さは 256 文字までで、リンク名の長さは 100 文字までです。最大のファイルサイズは 8GiB です。このフォーマットは古くて制限が多いですが、広くサポートされています。
- GNU tar format (*GNU_FORMAT*). 長いファイル名とリンク名、8GiB を超えるファイルやスパース (sparse) ファイルをサポートしています。これは GNU/Linux システムにおいてデファクト・スタンダードになっています。*tarfile* モジュールは長いファイル名を完全にサポートしています。スパースファイルは読み込みのみサポートしています。
- The POSIX.1-2001 pax format (*PAX_FORMAT*). It is the most flexible format with virtually no limits. It supports long filenames and linknames, large files and stores pathnames in a portable way. Modern tar implementations, including GNU tar, bsdtar/libarchive and star, fully support extended *pax* features; some old or unmaintained libraries may not, but should treat *pax* archives as if they were in the universally-supported *ustar* format. It is the current default format for new archives.

It extends the existing *ustar* format with extra headers for information that cannot be stored otherwise. There are two flavours of pax headers: Extended headers only affect the subsequent file header, global headers are valid for the complete archive and affect all following files. All the data in a pax header is encoded in *UTF-8* for portability reasons.

他にも、読み込みのみサポートしている tar フォーマットがいくつかあります:

- ancient V7 フォーマット。これは Unix 7th Edition から存在する、最初の tar フォーマットです。通常のファイルとディレクトリのみ保存します。名前は 100 文字を超えてはならず、ユーザー/グループ名に関する情報は保存されません。いくつかのアーカイブは、フィールドが ASCII でない文字を含む場合に、ヘッダーのチェックサムの計算を誤ります。
- SunOS tar 拡張フォーマット。POSIX.1-2001 pax フォーマットの亜流ですが、互換性がありません。

13.6.7 Unicode に関する問題

tar フォーマットは、もともとテープドライブにファイルシステムのバックアップをとる目的で設計されました。現在、tar アーカイブはファイルを配布する際に一般的に用いられ、ネットワーク上で交換されています。オリジナルフォーマットが抱える一つの問題は (他の多くのフォーマットでも同じですが)、様々な文字エンコーディングのサポートについて考慮していないことです。例えば、*UTF-8* システム上で作成された通常の tar アーカイブは、非 *ASCII* 文字を含んでいた場合、*Latin-1* システムでは正しく読み取ることができません。テキストのメタデータ (ファイル名、リンク名、ユーザー/グループ名など) は破壊されます。残念なことに、アーカイブのエンコーディングを自動検出する方法はありません。pax フォーマットはこの問題を解決するために設計されました。これは非 ASCII メタデータをユニバーサル文字エンコーディング *UTF-8* を使用して格納します。

tarfile における文字変換処理の詳細は *TarFile* クラスのキーワード引数 *encoding* および *errors* によって制御されます。

encoding はアーカイブのメタデータに使用する文字エンコーディングを指定します。デフォルト値は *sys.getfilesystemencoding()* で、フォールバックとして `'ascii'` が使用されます。アーカイブの読み書き時に、メタデータはそれぞれデコードまたはエンコードしなければなりません。*encoding* に適切な値が設定されていない場合、その変換は失敗することがあります。

引数 *errors* は文字を変換できない時の扱いを指定します。指定できる値は **エラーハンドラ** 節を参照してください。デフォルトのスキームは `'surrogateescape'` で、Python はそのファイルシステムの呼び出しも使用します。**ファイル名、コマンドライン引数、および環境変数** を参照してください。

デフォルトの *PAX_FORMAT* アーカイブでは、メタデータはすべて *UTF-8* で格納されるため、*encoding* は通常指定する必要はありません。*encoding* は、まれにある、バイナリの pax ヘッダーがデコードされた場合、あるいはサロゲート文字を含む文字列が格納されていた場合に使用されます。

第

FOURTEEN

ファイルフォーマット

この章で説明されるモジュールはマークアップや E メールでない様々なファイルフォーマットを構文解析します。

14.1 csv --- CSV ファイルの読み書き

ソースコード: Lib/csv.py

CSV (Comma Separated Values、カンマ区切り値列) と呼ばれる形式は、スプレッドシートやデータベース間でのデータのインポートやエクスポートにおける最も一般的な形式です。CSV フォーマットは、**RFC 4180** によって標準的な方法でフォーマットを記述する試みが行われる以前から長年使用されました。明確に定義された標準がないということは、異なるアプリケーション によって生成されたり取り込まれたりするデータ間では、しばしば微妙な違いが発生するということを意味します。こうした違いのために、複数のデータ源から得られた CSV ファイルを処理する作業が鬱陶しいものになることがあります。とはいえ、デリミタ (delimiter) やクオート文字の 相違はあっても、全体的な形式は十分似通っているため、こうしたデータを効率的に操作し、データの読み書きにおける細々としたことをプログラマ から隠蔽するような単一のモジュールを書くことは可能です。

csv モジュールでは、CSV 形式で書かれたテーブル状のデータを読み書きするためのクラスを実装しています。このモジュールを使うことで、プログラマは Excel で使われている CSV 形式に関して詳しい知識をもっていなくても、" このデータを Excel で推奨されている形式で書いてください" とか、" データを Excel で作成されたこのファイルから読み出してください" と言うことができます。プログラマはまた、他のアプリケーションが解釈できる CSV 形式を記述したり、独自の特殊な目的をもった CSV 形式を定義することができます。

csv モジュールの *reader* および *writer* オブジェクトはシーケンス型を読み書きします。プログラマは *DictReader* や *DictWriter* クラスを使うことで、データを辞書形式で読み書きすることもできます。

参考:

PEP 305 - CSV File API　Python へのこのモジュールの追加を提案している Python 改良案 (PEP: Python Enhancement Proposal)。

14.1.1 モジュールコンテンツ

csv モジュールでは以下の関数を定義しています:

csv.**reader**(*csvfile*, *dialect='excel'*, ***fmtparams*)

与えられた *csvfile* 内の行を反復処理するような reader オブジェクトを返します。*csvfile* は **イテレータ** プロトコルをサポートし、`__next__()` メソッドが呼ばれた際に常に文字列を返すような任意のオブジェクトにすることができます --- **ファイルオブジェクト** でもリストでも構いません。*csvfile* がファイルオブジェクトの場合、`newline=''` として開くべきです。[*1] オプションとして *dialect* パラメータを与えることができ、特定の CSV 表現形式 (dialect) 特有のパラメータの集合を定義するために使われます。*dialect* 引数は *Dialect* クラスのサブクラスのインスタンスか、*list_dialects()* 関数が返す文字列の一つにすることができます。別のオプションである *fmtparams* キーワード引数は、現在の表現形式における個々の書式パラメータを上書きするために与えることができます。表現形式および書式化パラメータの詳細については、*Dialect* **クラスと書式化パラメータ** 節を参照してください。

csv ファイルから読み込まれた各行は、文字列のリストとして返されます。`QUOTE_NONNUMERIC` フォーマットオプションが指定された場合を除き、データ型の変換が自動的に行われることはありません (このオプションが指定された場合、クォートされていないフィールドは浮動小数点数に変換されます)。

短い利用例:

```
>>> import csv
>>> with open('eggs.csv', newline='') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
...     for row in spamreader:
...         print(', '.join(row))
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

csv.**writer**(*csvfile*, *dialect='excel'*, ***fmtparams*)

ユーザが与えたデータをデリミタで区切られた文字列に変換し、与えられたファイルオブジェクトに書き込むための writer オブジェクトを返します。*csvfile* は `write()` メソッドを持つ任意のオブジェクトです。*csvfile* がファイルオブジェクトの場合、`newline=''` として開くべきです[*1] 。オプションとして *dialect* 引数を与えることができ、利用する CSV 表現形式 (dialect) を指定することができます。*dialect* パラメータは *Dialect* クラスのサブクラスのインスタンスか、*list_dialects()* 関数が返す文字列の 1 つにすることができます。別のオプション引数である *fmtparams* キーワード引数は、現在の表現形式における個々の書式パラメータを上書きするために与えることができます。dialect と書式パラメータについての詳細は、*Dialect* **クラスと書式化パラメータ** 節を参照してください。DB API を実装するモジュールとのインタフェースを可能な限り容易にするために、*None* は空文字列として書き込まれます。この処理は可逆な変換ではありませんが、SQL で NULL データ値を CSV にダンプする処理を、`cursor.fetch*` 呼び出しによって返されたデータを前処理することなく簡単に行うことができます。他の非文字列データは、書き出される前に *str()* を使って文字列に変換されます。

短い利用例:

[*1] `newline=''` が指定されない場合、クォートされたフィールド内の改行は適切に解釈されず、書き込み時に `\r\n` を行末に用いる処理系では余分な `\r` が追加されてしまいます。csv モジュールは独自 (*universal*) の改行処理を行うため、`newline=''` を指定することは常に安全です。

```
import csv
with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                            quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

csv.register_dialect(*name*[, *dialect*[, ***fmtparams*]])

dialect を *name* と関連付けます。*name* は文字列でなければなりません。表現形式 (dialect) は *Dialect* のサブクラスを渡すか、またはキーワード引数 *fmtparams* 、もしくは両方で指定できますが、キーワード引数の方が優先されます。表現形式と書式化パラメータについての詳細は、*Dialect* クラスと書式化パラメータ 節を参照してください。

csv.unregister_dialect(*name*)

name に関連づけられた表現形式を表現形式レジストリから削除します。*name* が表現形式名でない場合には *Error* を送出します。

csv.get_dialect(*name*)

name に関連づけられた表現形式を返します。*name* が表現形式名でない場合には *Error* を送出します。この関数は不変の *Dialect* を返します。

csv.list_dialects()

登録されている全ての表現形式を返します。

csv.field_size_limit([*new_limit*])

パーサが許容する現在の最大フィールドサイズを返します。*new_limit* が渡されたときは、その値が新しい上限になります。

csv モジュールでは以下のクラスを定義しています:

class csv.DictReader(*f*, *fieldnames=None*, *restkey=None*, *restval=None*, *dialect='excel'*, **args*, ***kwds*)

通常の reader のように動作しますが、個々の列の情報を *dict* にマップするオブジェクトを生成します。マップのキーは省略可能な *fieldnames* パラメータで与えられます。

fieldnames パラメータは *sequence* です。*fieldnames* が省略された場合、ファイル *f* の最初の列の値が fieldnames として使われます。*fieldnames* がどのように決定されるかに関わらず、辞書はそれらのオリジナルの順番を維持します。

列が fieldnames より多くのフィールドを持っていた場合、残りのデータはリストに入れられて、*restkey* により指定されたフィールド名 (デフォルトでは `None`) で保存されます。非ブランクの列が fieldnames よりも少ないフィールドしか持たない場合、不明の値は *restval* の値 (デフォルトは `None`) によって埋められます。

その他の省略可能またはキーワード形式のパラメータは、ベースになっている *reader* インスタンスに渡されます。

バージョン 3.6 で変更: 返される列の型は `OrderedDict` になりました。

バージョン 3.8 で変更: 返される列の型は *dict* になりました。

短い利用例:

```
>>> import csv
>>> with open('names.csv', newline='') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Eric Idle
John Cleese

>>> print(row)
{'first_name': 'John', 'last_name': 'Cleese'}
```

class csv.DictWriter(*f*, *fieldnames*, *restval=''*, *extrasaction='raise'*, *dialect='excel'*, **args*, ***kwds*)

通常の writer のように動作しますが、辞書を出力行にマップするオブジェクトを生成します。*fieldnames* パラメータは、`writerow()` メソッドに渡された辞書の値がどのような順番でファイル *f* に書かれるかを指定するキーの *sequence* です。`writerow()` メソッドに渡された辞書に *fieldnames* には存在しないキーが含まれている場合、オプションの *extrasaction* パラメータによってどんな動作を行うかが指定されます。この値がデフォルト値である `'raise'` に設定されている場合、*ValueError* が送出されます。`'ignore'` に設定されている場合、辞書の余分な値は無視されます。その他のパラメータはベースになっている *writer* インスタンスに渡されます。

DictReader クラスとは異なり、*DictWriter* の *fieldnames* パラメータは省略可能ではありません。

短い利用例:

```
import csv

with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

class csv.Dialect

Dialect クラスはコンテナクラスで、基本的な用途としては、その属性を特定の *reader* や *writer* インスタンスのパラメータを定義するために用います。

class csv.excel

excel クラスは Excel で生成される CSV ファイルの通常のプロパティを定義します。これは `'excel'` という名前の dialect として登録されています。

class csv.excel_tab

excel_tab クラスは Excel で生成されるタブ分割ファイルの通常のプロパティを定義します。これは `'excel-tab'` という名前の dialect として登録されています。

class csv.unix_dialect

unix_dialect クラスは UNIX システムで生成される CSV ファイルの通常のプロパティ (行終端記号として '\n' を用い全てのフィールドをクォートするもの) を定義します。これは 'unix' という名前の dialect として登録されています。

バージョン 3.2 で追加.

class csv.Sniffer

Sniffer クラスは CSV ファイルの書式を推理するために用いられるクラスです。

Sniffer クラスではメソッドを二つ提供しています:

sniff(*sample*, *delimiters=None*)

与えられた *sample* を解析し、発見されたパラメータを反映した *Dialect* サブクラスを返します。オプションの *delimiters* パラメータを与えた場合、有効なデリミタ文字を含んでいるはずの文字列として解釈されます。

has_header(*sample*)

(CSV 形式と仮定される) サンプルテキストを解析して、最初の行がカラムヘッダの羅列のように推察される場合 *True* を返します。

Sniffer の利用例:

```
with open('example.csv', newline='') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... process CSV file contents here ...
```

csv モジュールでは以下の定数を定義しています:

csv.QUOTE_ALL

writer オブジェクトに対し、全てのフィールドをクオートするように指示します。

csv.QUOTE_MINIMAL

writer オブジェクトに対し、*delimiter* 、*quotechar* または *lineterminator* に含まれる任意の文字のような特別な文字を含むフィールドだけをクオートするように指示します。

csv.QUOTE_NONNUMERIC

writer オブジェクトに対し、全ての非数値フィールドをクオートするように指示します。

reader に対しては、クオートされていない全てのフィールドを *float* 型に変換するよう指示します。

csv.QUOTE_NONE

writer オブジェクトに対し、フィールドを決してクオートしないように指示します。現在の *delimiter* が出力データ中に現れた場合、現在設定されている *escapechar* 文字が前に付けられます。*escapechar* がセットされていない場合、エスケープが必要な文字に遭遇した writer は *Error* を送出します。

reader に対しては、クオート文字の特別扱いをしないように指示します。

csv モジュールでは以下の例外を定義しています:

exception csv.Error
: 全ての関数において、エラーが検出された際に送出される例外です。

14.1.2 Dialect クラスと書式化パラメータ

レコードに対する入出力形式の指定をより簡単にするために、特定の書式化パラメータは表現形式 (dialect) にまとめてグループ化されます。表現形式は *Dialect* クラスのサブクラスで、様々なクラス特有のメソッドと、validate() メソッドを一つ持っています。*reader* または *writer* オブジェクトを生成するとき、プログラマは文字列または *Dialect* クラスのサブクラスを表現形式パラメータとして渡さなければなりません。さらに、*dialect* パラメータの代りに、プログラマは上で定義されている属性と同じ名前を持つ個々の書式化パラメータを *Dialect* クラスに指定することができます。

Dialect は以下の属性をサポートしています:

Dialect.delimiter
: フィールド間を分割するのに用いられる 1 文字からなる文字列です。デフォルトでは ',' です。

Dialect.doublequote
: フィールド内に現れた *quotechar* のインスタンスで、クオートではないその文字自身でなければならない文字をどのようにクオートするかを制御します。*True* の場合、この文字は二重化されます。*False* の場合、*escapechar* は *quotechar* の前に置かれます。デフォルトでは *True* です。

  出力においては、*doublequote* が *False* で *escapechar* がセットされていない場合、フィールド内に *quotechar* が現れると *Error* が送出されます。

Dialect.escapechar
: writer が、*quoting* が *QUOTE_NONE* に設定されている場合に *delimiter* をエスケープするため、および、*doublequote* が *False* の場合に *quotechar* をエスケープするために用いられる、1 文字からなる文字列です。読み込み時には *escapechar* はそれに引き続く文字の特別な意味を取り除きます。デフォルトでは *None* で、エスケープを行ないません。

Dialect.lineterminator
: *writer* が作り出す各行を終端する際に用いられる文字列です。デフォルトでは '\r\n' です。

注釈: *reader* は '\r' または '\n' のどちらかを行末と認識するようにハードコードされており、*lineterminator* を無視します。この振る舞いは将来変更されるかもしれません。

Dialect.quotechar
: *delimiter* や *quotechar* といった特殊文字を含むか、改行文字を含むフィールドをクオートする際に用いられる 1 文字からなる文字列です。デフォルトでは '"' です。

Dialect.quoting
: クオートがいつ writer によって生成されるか、また reader によって認識されるかを制御します。QUOTE_* 定数のいずれか (モジュールコンテンツ 節参照) をとることができ、デフォルトでは *QUOTE_MINIMAL* です。

Dialect.skipinitialspace
: *True* の場合、*delimiter* の直後に続く空白は無視されます。デフォルトでは *False* です。

Dialect.strict
: True の場合、不正な CSV 入力に対して *Error* を送出します。デフォルトでは False です。

14.1.3 reader オブジェクト

reader オブジェクト (*DictReader* インスタンス、および *reader()* 関数によって返されたオブジェクト) は、以下の public なメソッドを持っています:

csvreader.__next__()
: reader の反復可能なオブジェクトから、現在の表現形式に基づいて次の行を解析してリスト（オブジェクトが *reader()* から返された場合）または辞書（ *DictReader* のインスタンスの場合）として返します。通常は next(reader) のようにして呼び出すことになります。

reader オブジェクトには以下の公開属性があります:

csvreader.dialect
: パーサで使われる表現形式の読み出し専用の記述です。

csvreader.line_num
: ソースイテレータから読んだ行数です。この数は返されるレコードの数とは、レコードが複数行に亘ることがあるので、一致しません。

DictReader オブジェクトは、以下の public な属性を持っています:

csvreader.fieldnames
: オブジェクトを生成するときに渡されなかった場合、この属性は最初のアクセス時か、ファイルから最初のレコードを読み出したときに初期化されます。

14.1.4 writer オブジェクト

Writer オブジェクト (*DictWriter* インスタンス、および *writer()* 関数によって返されたオブジェクト) は、以下の public なメソッドを持っています: *row* には、Writer オブジェクトの場合には文字列か数値のイテラブルを指定し、*DictWriter* オブジェクトの場合はフィールド名をキーとして対応する文字列か数値を格納した辞書オブジェクトを指定します (数値は *str()* で変換されます)。複素数を出力する場合、値をかっこで囲んで出力します。このため、CSV ファイルを読み込むアプリケーションで（そのアプリケーションが複素数をサポートしていたとしても）問題が発生する場合があります。

csvwriter.writerow(*row*)
: 現在の表現形式 (dialect) に沿ってフォーマットされた *row* パラメータを writer のファイルオブジェクトに書き込みます。ファイルオブジェクトの *write* メソッドを呼び出した際の戻り値を返します。

  バージョン 3.5 で変更: 任意のイテラブルのサポートの追加。

csvwriter.writerows(*rows*)

rows 引数 (上で解説した *row* オブジェクトのイテラブル) の全ての要素を現在の表現形式に基づいて書式化し、writer のファイルオブジェクトに書き込みます。

writer オブジェクトには以下の公開属性があります:

csvwriter.dialect
writer で使われる表現形式の読み出し専用の記述です。

DictWriter のオブジェクトは以下の public メソッドを持っています:

DictWriter.writeheader()
Write a row with the field names (as specified in the constructor) to the writer's file object, formatted according to the current dialect. Return the return value of the *csvwriter.writerow()* call used internally.

バージョン 3.2 で追加.

バージョン 3.8 で変更: *writeheader()* は内部的に利用している *csvwriter.writerow()* の返り値を返すようになりました。

14.1.5 使用例

最も簡単な CSV ファイル読み込みの例です:

```
import csv
with open('some.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

別の書式での読み込み:

```
import csv
with open('passwd', newline='') as f:
    reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE_NONE)
    for row in reader:
        print(row)
```

上に対して、単純な書き込みのプログラム例は以下のようになります。

```
import csv
with open('some.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

open() が CSV ファイルの読み込みに使われるため、ファイルはデフォルトではシステムのデフォルトエンコーディングでユニコード文字列にデコードされます (*locale.getpreferredencoding()* を参照)。他のエンコーディングを用いてデコードするには、open の引数 encoding を設定して、以下のようにします:

```
import csv
with open('some.csv', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

システムのデフォルトエンコーディング以外で書き込む場合も同様です。出力ファイルを開く際に引数 encoding を明示してください。

新しい表現形式の登録:

```
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
with open('passwd', newline='') as f:
    reader = csv.reader(f, 'unixpwd')
```

もう少し手の込んだ reader の使い方 --- エラーを捉えてレポートします。

```
import csv, sys
filename = 'some.csv'
with open(filename, newline='') as f:
    reader = csv.reader(f)
    try:
        for row in reader:
            print(row)
    except csv.Error as e:
        sys.exit('file {}, line {}: {}'.format(filename, reader.line_num, e))
```

このモジュールは文字列の解析は直接サポートしませんが、簡単にできます。

```
import csv
for row in csv.reader(['one,two,three']):
    print(row)
```

脚注

14.2 configparser --- 設定ファイルのパーサー

ソースコード: Lib/configparser.py

このモジュールは、Microsoft Windows の INI ファイルに似た構造を持ったベーシックな設定用言語を実装した *ConfigParser* クラスを提供します。このクラスを使ってユーザーが簡単にカスタマイズできる Python プログラムを作ることができます。

注釈: このライブラリでは、Windows のレジストリ用に拡張された INI 文法はサポート **していません** 。

参考:

shlex **モジュール** アプリケーション設定ファイルのフォーマットとして使える、Unix シェルに似たミニ言語の作成を支援します。

json **モジュール** json モジュールは、同じ目的に利用できる JavaScript の文法のサブセットを実装しています。

14.2.1 クイックスタート

次のような、非常に簡単な設定ファイルを例に考えましょう:

```
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes

[bitbucket.org]
User = hg

[topsecret.server.com]
Port = 50022
ForwardX11 = no
```

INI ファイルの構造は **下のセクション** で解説します。基本的に、ファイルは複数のセクションからなり、各セクションは複数のキーと値を持ちます。*configparser* のクラス群はそれらのファイルを読み書きできます。まずは上のような設定ファイルをプログラムから作成してみましょう。

```
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config['DEFAULT'] = {'ServerAliveInterval': '45',
...                      'Compression': 'yes',
...                      'CompressionLevel': '9'}
>>> config['bitbucket.org'] = {}
>>> config['bitbucket.org']['User'] = 'hg'
>>> config['topsecret.server.com'] = {}
>>> topsecret = config['topsecret.server.com']
>>> topsecret['Port'] = '50022'     # mutates the parser
>>> topsecret['ForwardX11'] = 'no'  # same here
>>> config['DEFAULT']['ForwardX11'] = 'yes'
>>> with open('example.ini', 'w') as configfile:
...   config.write(configfile)
...
```

この例でわかるように、config parser は辞書のように扱うことができます。辞書との違いは **後に** 説明しますが、このインターフェイスは辞書に対して期待するのととても近い動作をします。

これで設定ファイルを作成して保存できました。次はこれを読み込み直して、中のデータを取り出してみましょう。

```
>>> config = configparser.ConfigParser()
>>> config.sections()
[]
>>> config.read('example.ini')
['example.ini']
>>> config.sections()
['bitbucket.org', 'topsecret.server.com']
>>> 'bitbucket.org' in config
True
>>> 'bytebong.com' in config
False
>>> config['bitbucket.org']['User']
'hg'
>>> config['DEFAULT']['Compression']
'yes'
>>> topsecret = config['topsecret.server.com']
>>> topsecret['ForwardX11']
'no'
>>> topsecret['Port']
'50022'
>>> for key in config['bitbucket.org']:
...     print(key)
user
compressionlevel
serveraliveinterval
compression
forwardx11
>>> config['bitbucket.org']['ForwardX11']
'yes'
```

上の例からわかるように、API はとても直感的です。唯一の魔術は、DEFAULT セクションが他の全てのセクションのためのデフォルト値を提供していることです[*1]。また、セクション内の各キーは大文字小文字を区別せず、全て小文字で保存されていることにも注意してください[*1]。

14.2.2 サポートされるデータ型

Config parser は値のデータ型について何も推論せず、常に文字列のまま内部に保存します。他のデータ型が必要な場合は自分で変換する必要があります:

```
>>> int(topsecret['Port'])
50022
>>> float(topsecret['CompressionLevel'])
9.0
```

このタスクはとても一般的なため、設定パーサーでは整数、浮動小数点数、真偽値を扱うための手頃なゲッターメソッドが提供されています。真偽値の扱いは一筋縄ではいきません。文字列を `bool()` に渡しても、`bool('False')` が `True` になってしまいます。そこで config parser は *`getboolean()`* を提供しています。

[*1] 設定パーサーは大々的にカスタマイズできます。脚注の参照で概説された挙動の変更に関心がある場合、*Customizing Parser Behaviour* セクションを参照してください。

このメソッドは大文字小文字を区別せず、'yes'/'no'、'on'/'off'、'true'/'false'、'1'/'0' を真偽値として認識します[*1]。例えば:

```
>>> topsecret.getboolean('ForwardX11')
False
>>> config['bitbucket.org'].getboolean('ForwardX11')
True
>>> config.getboolean('bitbucket.org', 'Compression')
True
```

config parser では、*getboolean()* 以外に *getint()* と *getfloat()* メソッドも提供されています。独自のコンバーターの登録、提供されたメソッドのカスタマイズもできます。[*1]

14.2.3 代替値

辞書と同じように、セクションの get() メソッドは代替値を提供しています:

```
>>> topsecret.get('Port')
'50022'
>>> topsecret.get('CompressionLevel')
'9'
>>> topsecret.get('Cipher')
>>> topsecret.get('Cipher', '3des-cbc')
'3des-cbc'
```

デフォルト値は代替値よりも優先されることに注意してください。例えば上の例では、'CompressionLevel' キーは 'DEFAULT' セクションにしか存在しません。その値を 'topsecret.server.com' から取得しようとした場合、代替値を指定しても常にデフォルト値を返します:

```
>>> topsecret.get('CompressionLevel', '3')
'9'
```

もう一つ注意すべき点は、パーサーレベルの (訳注: ConfigParser クラスの) get() メソッドは、後方互換性のために、カスタムのより複雑なインターフェースを提供します。このメソッドを使用する際には、フォールバック値はキーワード専用引数 fallback を介して提供されます:

```
>>> config.get('bitbucket.org', 'monster',
...            fallback='No such things as monsters')
'No such things as monsters'
```

同様の fallback 引数を、*getint()* 、*getfloat()* と *getboolean()* メソッドでも使えます。例えば:

```
>>> 'BatchMode' in topsecret
False
>>> topsecret.getboolean('BatchMode', fallback=True)
True
>>> config['DEFAULT']['BatchMode'] = 'no'
>>> topsecret.getboolean('BatchMode', fallback=True)
False
```

14.2.4 サポートする INI ファイルの構造

設定ファイルは複数のセクションから構成されます。セクションは、[section] ヘッダに続いた、特定の文字列 (デフォルトでは = または :[*1]) で区切られたキーと値のエントリです。デフォルトでは、セクション名は大文字と小文字を区別しますが、キーはそうではありません[*1]。キーと値、それぞれの先頭と末尾の空白は取り除かれます。値は省略することができ、その際でも、キーと値の区切り文字は残しておけます。値はまた、値の先頭の行より深くインデントされていれば、複数の行にまたがっても構いません。パーサーのモードによって、空白行は、複数行からなる値の一部として扱われるか、無視されます。

設定ファイルには先頭に特定の文字 (デフォルトでは # および ;[*1]) をつけてコメントをつけることができます。コメントは、他の内容がない行に置くことができ、インデントされていても構いません。[*1]

例えば:

```
[Simple Values]
key=value
spaces in keys=allowed
spaces in values=allowed as well
spaces around the delimiter = obviously
you can also use : to delimit keys from values

[All Values Are Strings]
values like this: 1000000
or this: 3.14159265359
are they treated as numbers? : no
integers, floats and booleans are held as: strings
can use the API to get converted values directly: true

[Multiline Values]
chorus: I'm a lumberjack, and I'm okay
    I sleep all night and I work all day

[No Values]
key_without_value
empty string value here =

[You can use comments]
# like this
; or this

# By default only in an empty line.
# Inline comments can be harmful because they prevent users
# from using the delimiting characters as parts of values.
# That being said, this can be customized.

    [Sections Can Be Indented]
        can_values_be_as_well = True
        does_that_mean_anything_special = False
        purpose = formatting for readability
        multiline_values = are
            handled just fine as
```

(次のページに続く)

(前のページからの続き)

```
            long as they are indented
            deeper than the first line
            of a value
        # Did I mention we can indent comments, too?
```

14.2.5 値の補間

コア機能に加えて、*ConfigParser* は補間 (interpolation, 内挿とも) をサポートします。これは get() コールが値を返す前に、その値に対して前処理を行えることを意味します。

class configparser.**BasicInterpolation**

ConfigParser が使用するデフォルト実装です。値に、同じセクションか特別なデフォルトセクション中[*1] の他の値を参照するフォーマット文字列を含めることができます。追加のデフォルト値を初期化時に提供できます。

例えば:

```
[Paths]
home_dir: /Users
my_dir: %(home_dir)s/lumberjack
my_pictures: %(my_dir)s/Pictures

[Escape]
gain: 80%%  # use a %% to escape the % sign (% is the only character that needs to be␣
↪escaped)
```

上の例では、*interpolation* に BasicInterpolation() を設定した *ConfigParser* が %(home_dir)s を home_dir の値 (このケースでは /Users) として解決しています、その結果 %(my_dir)s は /Users/lumberjack になります。全ての補間は必要に応じて実行されるため、設定ファイル中で参照の連鎖をもつキーを特定の順序で記述する必要はありません。

interpolation に None を設定すれば、パーサーは単に my_pictures の値として %(my_dir)s/Pictures を返し、my_dir の値として %(home_dir)s/lumberjack を返します。

class configparser.**ExtendedInterpolation**

zc.buildout で使用されるような、より高度な文法を実装した補間ハンドラの別の選択肢です。拡張された補間は、他のセクション中の値を示すのに ${section:option} と書けます。補間は複数のレベルに及べます、利便性のために、もし section: の部分が省略されると、現在のセクションがデフォルト値となります (スペシャルセクション中のデフォルト値を使用することもできます)。

たとえば、上記の basic interpolation で指定した設定は、extended interpolation を使うと下記のようになります:

```
[Paths]
home_dir: /Users
my_dir: ${home_dir}/lumberjack
my_pictures: ${my_dir}/Pictures
```

(次のページに続く)

(前のページからの続き)

```
[Escape]
cost: $$80  # use a $$ to escape the $ sign ($ is the only character that needs to be␣
↪escaped)
```

他のセクションから値を持ってくることもできます:

```
[Common]
home_dir: /Users
library_dir: /Library
system_dir: /System
macports_dir: /opt/local

[Frameworks]
Python: 3.2
path: ${Common:system_dir}/Library/Frameworks/

[Arthur]
nickname: Two Sheds
last_name: Jackson
my_dir: ${Common:home_dir}/twosheds
my_pictures: ${my_dir}/Pictures
python_dir: ${Frameworks:path}/Python/Versions/${Frameworks:Python}
```

14.2.6 マップ型プロトコルアクセス

バージョン 3.2 で追加.

マップ型プロトコルアクセスは、カスタムオブジェクトを辞書であるかのように使うための機能の総称です。*configparser* の場合、マップ型インタフェースの実装は `parser['section']['option']` 表記を使います。

とくに、`parser['section']` はパーサー内のそのセクションのデータへのプロキシを返します。つまり、値はコピーされるのではなく必要に応じてオリジナルのパーサーから取られます。さらに重要なことに、セクションのプロキシの値が変更されると、オリジナルのパーサー中の値が実際に変更されます。

configparser は可能な限り実際の辞書と近い振る舞いをします。マップ型インタフェースは *MutableMapping* を矛盾なく完成します。しかし、考慮するべき違いがいくつかあります:

- デフォルトでは、セクション内の全てのキーは大文字小文字の区別なくアクセスできます[*1]。例えば、`for option in parser["section"]` は `optionxform` されたオプションキー名のみを yield します。つまり小文字のキーがデフォルトです。同時に、キー `'a'` を含むセクションにおいて、どちらの式も `True` を返します:

```
"a" in parser["section"]
"A" in parser["section"]
```

- 全てのセクションは `DEFAULTSECT` 値を持ち、すなわちセクションで `.clear()` してもセクションは見

た目上空になりません。これは、デフォルト値は (技術的にはそこにないので) セクションから削除できないためです。デフォルト値が上書きされた場合、それが削除されるとデフォルト値が再び見えるようになります。デフォルト値を削除しようとすると *KeyError* が発生します。

- DEFAULTSECT はパーサーから取り除けません:
  - 削除しようとすると *ValueError* が発生します。
  - `parser.clear()` はこれをそのまま残し、
  - `parser.popitem()` がこれを返すことはありません。
- `parser.get(section, option, **kwargs)` - 第二引数は代替値では **ありません**。ただし、セクションごとの `get()` メソッドはマップ型プロトコルと旧式の configparser API の両方に互換です。
- `parser.items()` はマップ型プロトコルと互換です (DEFAULTSECT を含む *section_name*, *section_proxy* 対のリストを返します)。ただし、このメソッドは `parser.items(section, raw, vars)` のようにして引数を与えることでも呼び出せます。後者の呼び出しは指定された `section` の *option*, *value* 対のリストを、(`raw=True` が与えられない限り) 全ての補間を展開して返します。

マップ型プロトコルは、既存のレガシーな API の上に実装されているので、オリジナルのインタフェースを上書きする派生クラスもまたは期待どおりにはたらきます。

14.2.7 パーサーの振る舞いをカスタマイズする

INI フォーマットの変種は、それを使うアプリケーションの数と同じくらい多く存在します。*configparser* は、可能な限り広い範囲の INI スタイルを集めた集合をサポートするために、非常に役立ちます。デフォルトの機能は主に歴史的背景によって決められたので、機能によってはカスタマイズしてお使いください。

特定の設定パーサーのはたらきを変える最も一般的な方法は `__init__()` オプションを使うことです:

- *defaults*, デフォルト値: `None`

  このオプションは最初に `DEFAULT` セクションに加えられるキー-値の対の辞書を受け付けます。

  ヒント: 特定のセクションにデフォルト値を指定したいなら、実際のファイルを読み込む前に `read_dict()` を使ってください。

- *dict_type*, デフォルト値: *dict*

  このオプションはマップ型プロトコルの振る舞い方や書き込まれる設定ファイルの見た目に大きく影響します。標準の辞書では、全てのセクションはパーサーに加えられた順に並びます。同じことがセクション内のオプションにも言えます。

  セクションとオプションをライトバック時にソートするためなどに、別の辞書型も使えます。

  注意: 一度の操作でキー-値の対を複数追加する方法もあります。そのような操作に普通の辞書を使うと、キーの並びは挿入順になります。例えば:

```
>>> parser = configparser.ConfigParser()
>>> parser.read_dict({'section1': {'key1': 'value1',
...                                'key2': 'value2',
...                                'key3': 'value3'},
...                   'section2': {'keyA': 'valueA',
...                                'keyB': 'valueB',
...                                'keyC': 'valueC'},
...                   'section3': {'foo': 'x',
...                                'bar': 'y',
...                                'baz': 'z'}
... })
>>> parser.sections()
['section1', 'section2', 'section3']
>>> [option for option in parser['section3']]
['foo', 'bar', 'baz']
```

- *allow_no_value*, デフォルト値: `False`

  一部の設定ファイルには値のない設定項目がありますが、それ以外は `ConfigParser` がサポートする文法に従います。コンストラクタの *allow_no_value* 引数で、そのような値を許可することができます。

```
>>> import configparser

>>> sample_config = """
... [mysqld]
...   user = mysql
...   pid-file = /var/run/mysqld/mysqld.pid
...   skip-external-locking
...   old_passwords = 1
...   skip-bdb
...   # we don't need ACID today
...   skip-innodb
... """
>>> config = configparser.ConfigParser(allow_no_value=True)
>>> config.read_string(sample_config)

>>> # Settings with values are treated as before:
>>> config["mysqld"]["user"]
'mysql'

>>> # Settings without values provide None:
>>> config["mysqld"]["skip-bdb"]

>>> # Settings which aren't specified still raise an error:
>>> config["mysqld"]["does-not-exist"]
Traceback (most recent call last):
  ...
KeyError: 'does-not-exist'
```

- *delimiters*, デフォルト値: `('=', ':')`

デリミタはセクション内でキーを値から区切る部分文字列です。行中で最初に現れた区切り部分文字列がデリミタと見なされます。つまり値にはデリミタを含めることができます (キーには含めることができません)。

ConfigParser.write() の *space_around_delimiters* 引数も参照してください。

- *comment_prefixes*, デフォルト値: `('#', ';')`
- *inline_comment_prefixes*, デフォルト値: `None`

コメント接頭辞は設定ファイル中で有効なコメントの開始を示す文字列です。*comment_prefixes* は他の内容がない行 (インデントは自由) にのみ使用でき、*inline_comment_prefixes* は任意の有効な値 (例えば、セクション名、オプション、空行も可能) の後に使えます。デフォルトではインラインコメントは無効化されていて、`'#'` と `';'` を行全体のコメントに使用します。

バージョン 3.2 で変更: 以前のバージョンの *configparser* の振る舞いは `comment_prefixes=('#', ';')` および `inline_comment_prefixes=(';',)` に該当します。

設定パーサーはコメント接頭辞のエスケープをサポートしないので、*inline_comment_prefixes* はユーザーがコメント接頭辞として使われる文字を含むオプション値を指定するのを妨げる可能性があります。疑わしい場合には、*inline_comment_prefixes* を設定しないようにしてください。どのような状況でも、複数行にわたる値で、行の先頭にコメント接頭辞文字を保存する唯一の方法は、次の例のように接頭辞を補間することです:

```
>>> from configparser import ConfigParser, ExtendedInterpolation
>>> parser = ConfigParser(interpolation=ExtendedInterpolation())
>>> # the default BasicInterpolation could be used as well
>>> parser.read_string("""
... [DEFAULT]
... hash = #
...
... [hashes]
... shebang =
...   ${hash}!/usr/bin/env python
...   ${hash} -*- coding: utf-8 -*-
...
... extensions =
...   enabled_extension
...   another_extension
...   #disabled_by_comment
...   yet_another_extension
...
... interpolation not necessary = if # is not at line start
... even in multiline values = line #1
...   line #2
...   line #3
... """)
>>> print(parser['hashes']['shebang'])

#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

(次のページに続く)

(前のページからの続き)

```
>>> print(parser['hashes']['extensions'])

enabled_extension
another_extension
yet_another_extension
>>> print(parser['hashes']['interpolation not necessary'])
if # is not at line start
>>> print(parser['hashes']['even in multiline values'])
line #1
line #2
line #3
```

- *strict*, デフォルト値: `True`

  `True` に設定された場合、パーサーは単一のソースから (`read_file()`, `read_string()` または `read_dict()` を使って) 読み込むときにセクションやオプションの重複を許さなくなります。新しいアプリケーションには strict なパーサーを使うことが推奨されます。

  バージョン 3.2 で変更: 以前のバージョンの *configparser* の振る舞いは `strict=False` に該当します。

- *empty_lines_in_values*, デフォルト値: `True`

  設定パーサーでは、キーよりもその値を深くインデントするかぎり、複数行にまたがる値を使えます。デフォルトのパーサーはさらにその値の間に空行を置けます。同時に、キーは読みやすくするため任意にインデントできます。結果として、設定ファイルが大きく複雑になったとき、ユーザーがファイル構造を見失いやすいです。この例をご覧ください:

  ```
  [Section]
  key = multiline
    value with a gotcha

   this = is still a part of the multiline value of 'key'
  ```

  これは特にプロポーショナルフォントを使ってファイルを編集しているユーザーにとって問題になることがあります。だから、アプリケーションの値に空行が必要ないなら、空行を認めないべきです。これによって空行で必ずキーが分かれます。上の例では、2 つのキー、`key` および `this` が作られます。

- *default_section*, デフォルト値: `configparser.DEFAULTSECT` (すなわち: `"DEFAULT"`)

  他のセクションのデフォルト値や補間目的での特別なセクションを認める慣行はこのライブラリの明確なコンセプトの一つで、ユーザーは複雑で宣言的な設定を作成できます。このセクションは通常 `"DEFAULT"` と呼ばれますが、任意の有効なセクション名を指すようにカスタマイズできます。典型的な値には `"general"` や `"common"` があります。与えられた名前はソースを読み込む際にデフォルトセクションを認識するのに使われ、設定をファイルに書き戻すときにも使われます。現在の値は `parser_instance.default_section` 属性から取り出すことができ、実行時 (すなわちファイルを別のフォーマットに変換するとき) に変更することもできます。

- *interpolation*, デフォルト値: `configparser.BasicInterpolation`

補間の振る舞いは、*interpolation* 引数を通してカスタムハンドラを与えることでカスタマイズできます。None 引数を使うと補間を完全に無効にできます。ExtendedInterpolation() は、zc.buildout に影響を受けたより高度な補間を提供します。この話題に 特化したドキュメントのセクション をご覧ください。*RawConfigParser* のデフォルト値は None です。

- *converters*, デフォルト値: 未設定

  設定パーサーは、型変換を実行するオプションの値ゲッターを提供します。デフォルトでは、*getint()*、*getfloat()*、*getboolean()* が実装されています。他のゲッターが必要な場合、ユーザーはそれらをサブクラスで定義するか、辞書を渡します。辞書を渡す場合、各キーはコンバーターの名前で、値は当該変換を実装する呼び出し可能オブジェクトです。例えば、{'decimal': decimal.Decimal} を渡すと、パーサーオブジェクトとすべてのセクションプロキシの両方に、getdecimal() が追加されます。つまり、parser_instance.getdecimal('section', 'key', fallback=0) と parser_instance['section'].getdecimal('key', 0) の両方の方法で書くことができます。

  コンバーターがパーサーの状態にアクセスする必要がある場合、設定パーサーサブクラスでメソッドとして実装することができます。このメソッドの名前が get から始まる場合、すべてのセクションプロキシで、辞書と互換性のある形式で利用できます (上記の getdecimal() の例を参照)。

これらのパーサー引数のデフォルト値を上書きすれば、さらに進んだカスタマイズができます。デフォルトはクラスで定義されているので、派生クラスや属性の代入で上書きできます。

ConfigParser.BOOLEAN_STATES

デフォルトでは、*getboolean()* を使うことで、設定パーサーは以下の値を True と見なします: '1', 'yes', 'true', 'on' 。以下の値を False と見なします: '0', 'no', 'false', 'off' 。文字列と対応するブール値のカスタム辞書を指定することでこれを上書きできます。たとえば:

```
>>> custom = configparser.ConfigParser()
>>> custom['section1'] = {'funky': 'nope'}
>>> custom['section1'].getboolean('funky')
Traceback (most recent call last):
...
ValueError: Not a boolean: nope
>>> custom.BOOLEAN_STATES = {'sure': True, 'nope': False}
>>> custom['section1'].getboolean('funky')
False
```

ほかの典型的なブール値ペアには accept/reject や enabled/disabled などがあります。

ConfigParser.optionxform(*option*)

このメソッドは読み込み、取得、設定操作のたびにオプション名を変換します。デフォルトでは名前を小文字に変換します。従って設定ファイルが書き込まれるとき、すべてのキーは小文字になります。それがふさわしくなければ、このメソッドを上書きしてください。例えば:

```
>>> config = """
... [Section1]
... Key = Value
...
... [Section2]
```

(次のページに続く)

(前のページからの続き)

```
... AnotherKey = Value
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> list(typical['Section1'].keys())
['key']
>>> list(typical['Section2'].keys())
['anotherkey']
>>> custom = configparser.RawConfigParser()
>>> custom.optionxform = lambda option: option
>>> custom.read_string(config)
>>> list(custom['Section1'].keys())
['Key']
>>> list(custom['Section2'].keys())
['AnotherKey']
```

注釈: The optionxform function transforms option names to a canonical form. This should be an idempotent function: if the name is already in canonical form, it should be returned unchanged.

ConfigParser.SECTCRE

セクションヘッダを解析するのに使われる、コンパイルされた正規表現です。デフォルトでは [section] が "section" という名前にマッチします。空白はセクション名の一部と見なされるので、[larch] は " larch " という名のセクションとして読み込まれます。これがふさわしくない場合、このメソッドを上書きしてください。例えば:

```
>>> import re
>>> config = """
... [Section 1]
... option = value
...
... [  Section 2  ]
... another = val
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> typical.sections()
['Section 1', '  Section 2  ']
>>> custom = configparser.ConfigParser()
>>> custom.SECTCRE = re.compile(r"\[ *(?P<header>[^]]+?) *\]")
>>> custom.read_string(config)
>>> custom.sections()
['Section 1', 'Section 2']
```

注釈: ConfigParser オブジェクトはオプション行の認識に OPTCRE 属性も使いますが、これを上書きすることは推奨されません。上書きするとコンストラクタオプション *allow_no_value* および

delimiters に干渉します。

14.2.8 レガシーな API の例

主に後方互換性問題の理由から、*configparser* は get/set メソッドを明示するレガシーな API も提供します。メソッドを以下に示すように使うこともできますが、新しいプロジェクトではマップ型プロトコルでアクセスするのが望ましいです。レガシーな API は時折高度で、低レベルで、まったく直感的ではありません。

設定ファイルを書き出す例:

```
import configparser

config = configparser.RawConfigParser()

# Please note that using RawConfigParser's set functions, you can assign
# non-string values to keys internally, but will receive an error when
# attempting to write to a file or when you get it in non-raw mode. Setting
# values using the mapping protocol or ConfigParser's set() does not allow
# such assignments to take place.
config.add_section('Section1')
config.set('Section1', 'an_int', '15')
config.set('Section1', 'a_bool', 'true')
config.set('Section1', 'a_float', '3.1415')
config.set('Section1', 'baz', 'fun')
config.set('Section1', 'bar', 'Python')
config.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# Writing our configuration file to 'example.cfg'
with open('example.cfg', 'w') as configfile:
    config.write(configfile)
```

設定ファイルを読み込む例:

```
import configparser

config = configparser.RawConfigParser()
config.read('example.cfg')

# getfloat() raises an exception if the value is not a float
# getint() and getboolean() also do this for their respective types
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print(a_float + an_int)

# Notice that the next output does not interpolate '%(bar)s' or '%(baz)s'.
# This is because we are using a RawConfigParser().
if config.getboolean('Section1', 'a_bool'):
    print(config.get('Section1', 'foo'))
```

補間するには、*ConfigParser* を使ってください:

```
import configparser

cfg = configparser.ConfigParser()
cfg.read('example.cfg')

# Set the optional *raw* argument of get() to True if you wish to disable
# interpolation in a single get operation.
print(cfg.get('Section1', 'foo', raw=False))  # -> "Python is fun!"
print(cfg.get('Section1', 'foo', raw=True))   # -> "%(bar)s is %(baz)s!"

# The optional *vars* argument is a dict with members that will take
# precedence in interpolation.
print(cfg.get('Section1', 'foo', vars={'bar': 'Documentation',
                                       'baz': 'evil'}))

# The optional *fallback* argument can be used to provide a fallback value
print(cfg.get('Section1', 'foo'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'foo', fallback='Monty is not.'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'monster', fallback='No such things as monsters.'))
      # -> "No such things as monsters."

# A bare print(cfg.get('Section1', 'monster')) would raise NoOptionError
# but we can also use:

print(cfg.get('Section1', 'monster', fallback=None))
      # -> None
```

どちらの型の ConfigParsers でもデフォルト値が利用できます。使われているオプションがどこにも定義されていなければ、そのデフォルト値が補間に使われます。

```
import configparser

# New instance with 'bar' and 'baz' defaulting to 'Life' and 'hard' each
config = configparser.ConfigParser({'bar': 'Life', 'baz': 'hard'})
config.read('example.cfg')

print(config.get('Section1', 'foo'))     # -> "Python is fun!"
config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print(config.get('Section1', 'foo'))     # -> "Life is hard!"
```

14.2.9 ConfigParser オブジェクト

class configparser.ConfigParser(*defaults=None*, *dict_type=dict*, *allow_no_value=False*, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*, *interpolation=BasicInterpolation()*, *converters={}*)

主要な設定パーサーです。*defaults* が与えられれば、その辞書の持つ初期値で初期化されます。*dict_type* が与えられれば、それがセクションの一覧、セクション中のオプション、およびデフォルト値の辞書オブジェクトを作成するのに使われます。

delimiters が与えられた場合、キーと値を分割する部分文字列の組み合わせとして使われます。*comment_prefixes* が与えられた場合、他の内容がない行のコメントに接頭する部分文字列の組み合わせとして使われます。コメントはインデントできます。*inline_comment_prefixes* が与えられた場合、非空行のコメントに接頭する部分文字列としての組み合わせとして使われます。

strict が `True` (デフォルト) であれば、パーサーは単一のソース (ファイル、文字列、辞書) 中にセクションやオプションの重複を認めず、*DuplicateSectionError* や *DuplicateOptionError* を送出します。*empty_lines_in_values* が `False` (デフォルト: `True`) なら、空行はそれぞれオプションの終わりを示します。*allow_no_value* が `True` (デフォルト: `False`) なら、値のないオプションが受け付けられます。そのオプションの値は `None` となり、後端のデリミタを除いてシリアル化されます。

default_section が与えられた場合、他のセクションへのデフォルト値や補間のためのデフォルト値を保持する特別なセクションの名前を指定します (通常は `"DEFAULT"` という名前です)。この値は実行時に `default_section` インスタンス属性を使って取得や変更ができます。

補間の動作は、*interpolation* 引数を通してカスタムハンドラを与えることでカスタマイズできます。`None` 引数を使うと補間を完全に無効にできます。`ExtendedInterpolation()` は、`zc.buildout` に影響を受けたより高度な補間を提供します。この件に **特化したドキュメントのセクション** を参照してください。

補間に使われるすべてのオプション名は、他のオプション名参照と同様に、*optionxform()* メソッドを通して渡されます。例えば、*optionxform()* のデフォルトの実装を使うと、値 `foo %(bar)s` と `foo %(BAR)s` は等しくなります。

converters が与えられた場合、各キーが型コンバーターの名前を表し、各値が文字列から目的のデータ型への変換を実装する呼び出し可能オブジェクトです。各コンバーターは、自身の対応する `get*()` メソッドをパーサーオブジェクトとセクションプロキシで取得します。

バージョン 3.1 で変更: デフォルトの *dict_type* は *collections.OrderedDict* です。

バージョン 3.2 で変更: *allow_no_value*, *delimiters*, *comment_prefixes*, *strict*, *empty_lines_in_values*, *default_section* および *interpolation* が追加されました。

バージョン 3.5 で変更: *converters* 引数が追加されました。

バージョン 3.7 で変更: The *defaults* argument is read with *read_dict()*, providing consistent behavior across the parser: non-string keys and values are implicitly converted to strings.

バージョン 3.8 で変更: The default *dict_type* is *dict*, since it now preserves insertion order.

defaults()
インスタンス全体で使われるデフォルト値の辞書を返します。

sections()
利用できるセクションのリストを返します。*default section* はリストに含まれません。

add_section(*section*)
section という名のセクションをインスタンスに追加します。与えられた名前のセクション名がすでに存在したら、*DuplicateSectionError* が送出されます。*default section* 名が渡されたら、*ValueError* が送出されます。セクションの名前は文字列でなければなりません。そうでなければ、*TypeError* が送出されます。

バージョン 3.2 で変更: 文字列でないセクション名は *TypeError* を送出します。

has_section(*section*)
指名された *section* が設定中に存在するかを示します。*default section* は認識されません。

options(*section*)
指定された *section* 中で利用できるオプションのリストを返します。

has_option(*section*, *option*)
与えられた *section* が存在し、与えられた *option* を含む場合、*True* を返します。それ以外の場合には、*False* を返します。指定された *section* が *None* または空文字列の場合、DEFAULT が仮定されます。

read(*filenames*, *encoding=None*)
ファイル名の iterable を読み込んでパースしようと試みます。正常にパースできたファイル名のリストを返します。

もし *filenames* が文字列か *path-like object* なら、この引数は 1 つのファイル名として扱われます。*filenames* 中に開けないファイルがある場合、そのファイルは無視されます。この挙動は、設定ファイルが置かれる可能性のある場所 (例えば、カレントディレクトリ、ホームディレクトリ、システム全体の設定を行うディレクトリ) のイテラブルを指定して、イテラブルの中で存在する全ての設定ファイルを読むことを想定して設計されています。

どの設定ファイルも存在しなかった場合、*ConfigParser* のインスタンスは 空のデータセットを持ちます。初期値の設定ファイルを先に読み込んでおく必要があるアプリケーションでは、オプションのファイルを読み込むために *read()* を呼ぶ前に 、まず *read_file()* を用いて必要なファイルを読み込んでください:

```
import configparser, os

config = configparser.ConfigParser()
config.read_file(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')],
            encoding='cp1250')
```

バージョン 3.2 で追加: *encoding* 引数。以前は、すべてのファイルが *open()* のデフォルトエンコーディングを使って読まれていました。

バージョン 3.6.1 で追加: *filenames* 引数が *path-like object* を受け入れるようになりました。

バージョン 3.7 で追加: *filenames* 引数が *bytes* オブジェクトを受け入れるようになりました。

read_file(*f*, *source=None*)
設定データを *f* から読み込んで解析します。*f* は Unicode 文字列を yield するイテラブル (例えばテキストモードで開かれたファイル) です。

オプションの引数 *source* は読み込まれるファイルの名前を指定します。与えられず、*f* に name 属性があれば、それが *source* として使われます。デフォルトは '<???>' です。

バージョン 3.2 で追加: *readfp()* を置き換えます。

read_string(*string*, *source='<string>'*)
設定データを文字列から解析します。

オプションの引数 *source* はコンテキストにおける渡された文字列の名前を指定します。与えられなければ、'<string>' が使われます。これは一般にファイルシステムパスや URL にします。

バージョン 3.2 で追加.

read_dict(*dictionary*, *source='<dict>'*)
辞書的な items() メソッドを提供する任意のオブジェクトから設定を読み込みます。キーはセクション名で、値はそのセクションに現れるキーと値をもつ辞書です。使われた辞書型が順序を保存するなら、セクションおよびそのキーは順に加えられます。値は自動で文字列に変換されます。

オプションの引数 *source* はコンテキストにおける渡された辞書の名前を指定します。与えられなければ、<dict> が使われます。

このメソッドを使ってパーサー間で状態をコピーできます。

バージョン 3.2 で追加.

get(*section*, *option*, ***, *raw=False*, *vars=None*[, *fallback*])
指名された *section* の *option* の値を取得します。*vars* が提供されるなら、それは辞書でなければならず、(与えられたなら) *vars*, *section*, *DEFAULTSECT* 内からこの順で *option* が探索されます。*fallback* の値として None を与えられます。

raw が真でない時には、全ての '%' 置換は展開されてから返されます。置換後の値はオプションと同じ順序で探されます。

バージョン 3.2 で変更: 引数 *raw*, *vars* および *fallback* は、(特にマッピングプロトコルを使用するときに) ユーザーが第 3 引数を *fallback* フォールバックとして使おうとしないように、キーワード専用となりました。

getint(*section*, *option*, ***, *raw=False*, *vars=None*[, *fallback*])
指定された *section* 中の *option* を整数に型強制する補助メソッドです。*raw*, *vars* および *fallback* の説明は *get()* を参照してください。

getfloat(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])
指定された *section* 中の *option* を浮動小数点数に型強制する補助メソッドです。*raw*, *vars* および *fallback* の説明は *get()* を参照してください。

getboolean(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])
指定された *section* 中の *option* をブール値に型強制する補助メソッドです。なお、このオプションで受け付けられる値はこのメソッドが `True` を返す `'1'`, `'yes'`, `'true'`, および `'on'`, と、このメソッドが `False` を返す `'0'`, `'no'`, `'false'`, and `'off'` です。その他のいかなる値も *ValueError* を送出します。*raw*, *vars* および *fallback* の説明は *get()* を参照してください。

items(*raw=False*, *vars=None*)
items(*section*, *raw=False*, *vars=None*)
section が与えられなければ、DEFAULTSECT を含めた *section_name*, *section_proxy* の対のリストを返します。

与えられれば、与えられた *section* 中のオプションの *name*, *value* の対のリストを返します。オプションの引数は *get()* メソッドに与えるものと同じ意味を持ちます。

バージョン 3.8 で変更: *vars* に現れる項目は結果に表れなくなりました。以前の挙動は、実際のパーサーオプションを補間のために与えられた変数と混合していました。

set(*section*, *option*, *value*)
与えられたセクションが存在すれば、与えられたオプションを指定された値に設定します。そうでなければ *NoSectionError* を送出します。*option* および *value* は文字列でなければなりません。そうでなければ *TypeError* が送出されます。

write(*fileobject*, *space_around_delimiters=True*)
設定の表現を指定された *file object* に書き込みます。*fileobject* は (文字列を受け付ける) テキストモードで開かれていなければなりません。この表現は後で *read()* を呼び出すことでパースできます。*space_around_delimiters* が真なら、キーと値の間のデリミタはスペースで囲まれます。

remove_option(*section*, *option*)
指定された *option* を指定された *section* から削除します。セクションが存在しなければ、*NoSectionError* を送出します。オプションが存在して削除されれば、*True* を返します。そうでなければ *False* を返します。

remove_section(*section*)
指定された *section* を設定から削除します。セクションが実際に存在すれば、`True` を返します。そうでなければ `False` を返します。

optionxform(*option*)
入力ファイルに現れた、またはクライアントコードで渡されたオプション名 *option* を内部構造で実際に使われる形式に変換します。デフォルトの実装では *option* の小文字版を返します。派生クラスでこれを上書きするか、クライアントコードでインスタンス上のこの名前の属性を設定して、この動作に影響を与えることができます。

このメソッドを使うためにパーサーを派生クラス化させる必要はなく、インスタンス上で、これを文字列引数をとって文字列を返す関数に設定できます。例えば、これを `str` に設定すると、オプ

ション名に大文字小文字の区別をつけられます:

```
cfgparser = ConfigParser()
cfgparser.optionxform = str
```

なお、設定ファイルを読み込むとき、オプション名の周りの空白は *optionxform()* が呼び出される前に取り除かれます。

readfp(*fp*, *filename=None*)

バージョン 3.2 で非推奨: 代わりに *read_file()* を使ってください。

バージョン 3.2 で変更: *readfp()* は `fp.readline()` を呼び出す代わりに *fp* をイテレートするようになりました。

readfp() をイテレーションをサポートしない引数で呼び出す既存のコードには、ファイル的なオブジェクトまわりのラッパーとして以下のジェネレーターが使えます:

```
def readline_generator(fp):
    line = fp.readline()
    while line:
        yield line
        line = fp.readline()
```

`parser.readfp(fp)` の代わりに `parser.read_file(readline_generator(fp))` を使ってください。

configparser.MAX_INTERPOLATION_DEPTH

`get()` の *raw* が偽であるときの再帰的な補間の最大の深さです。これはデフォルトの *interpolation* を使うときのみ関係します。

14.2.10 RawConfigParser オブジェクト

class configparser.RawConfigParser(*defaults=None*, *dict_type=dict*, *allow_no_value=False*, ***, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*[, *interpolation*])

Legacy variant of the *ConfigParser*. It has interpolation disabled by default and allows for non-string section names, option names, and values via its unsafe `add_section` and `set` methods, as well as the legacy `defaults=` keyword argument handling.

バージョン 3.8 で変更: The default *dict_type* is *dict*, since it now preserves insertion order.

注釈: 代わりに内部に保存する値の型を検査する *ConfigParser* を使うことを検討してください。補間を望まない場合、`ConfigParser(interpolation=None)` を使用できます。

---

add_section(*section*)
インスタンスに *section* という名のセクションを追加します。与えられた名前のセクションがすでに存在すれば、*DuplicateSectionError* が送出されます。*default section* 名が渡されると、*ValueError* が送出されます。

section の型は検査されないため、ユーザーは非文字列の名前付きセクションを作ることができます。この振る舞いはサポートされておらず、内部エラーを起こす可能性があります。

set(*section*, *option*, *value*)
与えられたセクションが存在していれば、オプションを指定された値に設定します。セクションが存在しなければ *NoSectionError* を発生させます。*RawConfigParser* (あるいは *raw* パラメータをセットした *ConfigParser*) を文字列型でない値の **内部的な** 格納場所として使うことは可能ですが、すべての機能 (置換やファイルへの出力を含む) がサポートされるのは文字列を値として使った場合だけです。

ユーザーは、このメソッドを使って非文字列の値をキーに代入できます。この振る舞いはサポートされておらず、非 raw モードでの値の取得や、ファイルへの書き出しを試みた際にエラーの原因となりえます。このような代入を許さない **マッピングプロトコル API を使用してください**。

14.2.11 例外

exception configparser.Error
他の全ての *configparser* 例外の基底クラスです。

exception configparser.NoSectionError
指定したセクションが見つからなかった時に起きる例外です。

exception configparser.DuplicateSectionError
add_section() がすでに存在するセクションの名前で呼び出された場合や、strict なパーサーで単一の入力ファイル、文字列、辞書中に同じセクションが複数回現れたときに送出される例外です。

バージョン 3.2 で追加: オプションの source と lineno が属性および __init__() への引数として加えられました。

exception configparser.DuplicateOptionError
strict なパーサーで、単一の入力ファイル、文字列、辞書中に同じオプションが複数回現れたときに送出される例外です。これはミススペルや大文字小文字の区別に関係するエラー、例えば辞書の二つのキーが同じ大文字小文字の区別のない設定キーを表すこと、を捕捉します。

exception configparser.NoOptionError
指定されたオプションが指定されたセクションに見つからないときに送出される例外です。

exception configparser.InterpolationError
文字列の補間中に問題が起きた時に発生する例外の基底クラスです。

exception configparser.InterpolationDepthError
繰り返しの回数が *MAX_INTERPOLATION_DEPTH* を超えたために文字列補間が完了しなかったときに送出される例外です。*InterpolationError* の派生クラスです。

exception configparser.InterpolationMissingOptionError
InterpolationError の派生クラスで、値が参照しているオプションが見つからない場合に発生する例外です。

exception configparser.InterpolationSyntaxError
置換がなされるソーステキストが要求された文法を満たさないときに送出される例外です。*InterpolationError* の派生クラスです。

exception configparser.MissingSectionHeaderError
セクションヘッダを持たないファイルを構文解析しようとした時に起きる例外です。

exception configparser.ParsingError
ファイルの構文解析中にエラーが起きた場合に発生する例外です。

バージョン 3.2 で変更: filename という属性および __init__() の引数は source に名前が変更されました。

脚注

14.3 netrc --- netrc ファイルの処理

ソースコード: Lib/netrc.py

---

netrc クラスは、Unix **ftp** プログラムや他の FTP クライアントで用いられる netrc ファイル形式を解析し、カプセル化 (encapsulate) します。

class netrc.netrc([*file*])
netrc のインスタンスやサブクラスのインスタンスは netrc ファイルのデータをカプセル化します。初期化の際の引数が存在する場合、解析対象となるファイルの指定になります。引数がない場合、(*os.path.expanduser()* で特定される) ユーザのホームディレクトリ下にある .netrc が読み出されます。ファイルが見付からなかった場合は *FileNotFoundError* 例外が送出されます。解析エラーが発生した場合、ファイル名、行番号、解析を中断したトークンに関する情報の入った *NetrcParseError* を送出します。POSIX システムにおいて引数を指定しない場合、ファイルのオーナシップやパーミッションが安全でない (プロセスを実行しているユーザ以外が所有者であるか、誰にでも読み書き出来てしまう) のに .netrc ファイル内にパスワードが含まれていると、*NetrcParseError* を送出します。このセキュリティ的な振る舞いは、ftp などの .netrc を使うプログラムと同じものです。

バージョン 3.4 で変更: POSIX パーミッションのチェックが追加されました。

バージョン 3.7 で変更: 引数で *file* が渡されなかったときは、.netrc ファイルの場所を探すのに *os.path.expanduser()* が使われるようになりました。

exception netrc.NetrcParseError
ソースファイルのテキスト中で文法エラーに遭遇した場合に *netrc* クラスによって送出される例外です。この例外のインスタンスは 3 つのインスタンス変数を持っています: msg はテキストによるエラーの説明、filename はソースファイルの名前、そして lineno はエラーが発見された行番号です。

14.3.1 netrc オブジェクト

netrc インスタンスは以下のメソッドを持っています:

netrc.authenticators(*host*)
host の認証情報として、三要素のタプル (login, account, password) を返します。与えられた host に対するエントリが netrc ファイルにない場合、'default' エントリに関連付けられたタプルが返されます。host に対応するエントリがなく、default エントリもない場合、None を返します。

netrc.__repr__()
クラスの持っているデータを netrc ファイルの書式に従った文字列で出力します。(コメントは無視され、エントリが並べ替えられる可能性があります。)

netrc のインスタンスは以下の public なインスタンス変数を持っています:

netrc.hosts
ホスト名を (login, account, password) からなるタプルに対応づけている辞書です。'default' エントリがある場合、その名前の擬似ホスト名として表現されます。

netrc.macros
マクロ名を文字列のリストに対応付けている辞書です。

注釈: 利用可能なパスワードの文字セットは、ASCII のサブセットのみです。全ての ASCII の記号を使用することができます。しかし、空白文字と印刷不可文字を使用することはできません。この制限は .netrc ファイルの解析方法によるものであり、将来解除されます。

14.4 xdrlib --- XDR データのエンコードおよびデコード

ソースコード: Lib/xdrlib.py

---

xdrlib モジュールは外部データ表現標準 (External Data Representation Standard) のサポートを実現します。この標準は 1987 年に Sun Microsystems, Inc. によって書かれ、**RFC 1014** で定義されています。このモジュールでは RFC で記述されているほとんどのデータ型をサポートしています。

xdrlib モジュールでは 2 つのクラスが定義されています。一つは変数を XDR 表現にパックするためのクラスで、もう一方は XDR 表現からアンパックするためのものです。2 つの例外クラスが同様にして定義されています。

class xdrlib.Packer
Packer はデータを XDR 表現にパックするためのクラスです。*Packer* クラスのインスタンス生成は引数なしで行われます。

class xdrlib.Unpacker(*data*)
Unpacker は Packer と対をなしていて、文字列バッファから XDR をアンパックするためのクラスです。入力バッファ *data* を引数に与えてインスタンスを生成します。

参考:

RFC 1014 - XDR: External Data Representation Standard　この RFC が、かつてこのモジュールが最初に書かれた当時に XDR 標準であったデータのエンコード方法を定義していました。現在は **RFC 1832** に更新されているようです。

RFC 1832 - XDR: External Data Representation Standard　こちらが新しい方の RFC で、XDR の改訂版が定義されています。

14.4.1 Packer オブジェクト

Packer インスタンスには以下のメソッドがあります:

Packer.**get_buffer**()
　現在のパック処理用バッファを文字列で返します。

Packer.**reset**()
　パック処理用バッファをリセットして、空文字にします。

一般的には、適切な pack_type() メソッドを使えば、一般に用いられているほとんどの XDR データをパックすることができます。各々のメソッドは一つの引数をとり、パックしたい値を与えます。単純なデータ型をパックするメソッドとして、以下のメソッド: pack_uint() 、pack_int() 、pack_enum() 、pack_bool() 、pack_uhyper() そして pack_hyper() がサポートされています。

Packer.**pack_float**(*value*)
　単精度 (single-precision) の浮動小数点数 *value* をパックします。

Packer.**pack_double**(*value*)
　倍精度 (double-precision) の浮動小数点数 *value* をパックします。

以下のメソッドは文字列、バイト列、不透明データ (opaque data) のパック処理をサポートします:

Packer.**pack_fstring**(*n*, *s*)
　固定長の文字列、*s* をパックします。*n* は文字列の長さですが、この値自体はデータバッファにはパック **されません**。4 バイトのアラインメントを保証するために、文字列は必要に応じて null バイト列でパディングされます。

Packer.**pack_fopaque**(*n*, *data*)
　pack_fstring() と同じく、固定長の不透明データストリームをパックします。

Packer.**pack_string**(*s*)
　可変長の文字列 *s* をパックします。文字列の長さが最初に符号なし整数でパックされ、続いて *pack_fstring()* を使って文字列データがパックされます。

Packer.**pack_opaque**(*data*)
　pack_string() と同じく、可変長の不透明データ文字列をパックします。

Packer.**pack_bytes**(*bytes*)
　pack_string() と同じく、可変長のバイトストリームをパックします。

以下のメソッドはアレイやリストのパック処理をサポートします:

Packer.**pack_list**(*list*, *pack_item*)
: 一様な項目からなる *list* をパックします。このメソッドはサイズ不定、すなわち、全てのリスト内容を網羅するまでサイズが分からないリストに対して有用です。リストのすべての項目に対し、最初に符号無し整数 1 がパックされ、続いてリスト中のデータがパックされます。*pack_item* は個々の項目をパックするために呼び出される関数です。リストの末端に到達すると、符号無し整数 0 がパックされます。

  例えば、整数のリストをパックするには、コードは以下のようになるはずです:

```
import xdrlib
p = xdrlib.Packer()
p.pack_list([1, 2, 3], p.pack_int)
```

Packer.**pack_farray**(*n*, *array*, *pack_item*)
: 一様な項目からなる固定長のリスト (*array*) をパックします。*n* はリストの長さです。この値はデータバッファにパック **されません** が、`len(array)` が *n* と等しくない場合、例外 *ValueError* が送出されます。上と同様に、*pack_item* は個々の要素をパック処理するための関数です。

Packer.**pack_array**(*list*, *pack_item*)
: 一様の項目からなる可変長の *list* をパックします。まず、リストの長さが符号無し整数でパックされ、つづいて各要素が上の *pack_farray()* と同じやり方でパックされます。

14.4.2 Unpacker オブジェクト

Unpacker クラスは以下のメソッドを提供します:

Unpacker.**reset**(*data*)
: 文字列バッファを *data* でリセットします。

Unpacker.**get_position**()
: データバッファ中の現在のアンパック処理位置を返します。

Unpacker.**set_position**(*position*)
: データバッファ中のアンパック処理位置を *position* に設定します。*get_position()* および *set_position()* は注意して使わなければなりません。

Unpacker.**get_buffer**()
: 現在のアンパック処理用データバッファを文字列で返します。

Unpacker.**done**()
: アンパック処理を終了させます。全てのデータがまだアンパックされていなければ、例外 *Error* が送出されます。

上のメソッドに加えて、*Packer* でパック処理できるデータ型はいずれも *Unpacker* でアンパック処理できます。アンパック処理メソッドは unpack_type() の形式をとり、引数をとりません。これらのメソッドはアンパックされたデータオブジェクトを返します。

Unpacker.**unpack_float**()
: 単精度の浮動小数点数をアンパックします。

Unpacker.**unpack_double**()
: *unpack_float()* と同様に、倍精度の浮動小数点数をアンパックします。

上のメソッドに加えて、文字列、バイト列、不透明データをアンパックする以下のメソッドが提供されています:

Unpacker.**unpack_fstring**(*n*)
: 固定長の文字列をアンパックして返します。*n* は予想される文字列の長さです。4 バイトのアラインメントを保証するために null バイトによるパディングが行われているものと仮定して処理を行います。

Unpacker.**unpack_fopaque**(*n*)
: *unpack_fstring()* と同様に、固定長の不透明データストリームをアンパックして返します。

Unpacker.**unpack_string**()
: 可変長の文字列をアンパックして返します。最初に文字列の長さが符号無し整数としてアンパックされ、次に *unpack_fstring()* を使って文字列データがアンパックされます。

Unpacker.**unpack_opaque**()
: *unpack_string()* と同様に、可変長の不透明データ文字列をアンパックして返します。

Unpacker.**unpack_bytes**()
: *unpack_string()* と同様に、可変長のバイトストリームをアンパックして返します。

以下メソッドはアレイおよびリストのアンパック処理をサポートします:

Unpacker.**unpack_list**(*unpack_item*)
: 一様な項目からなるリストをアンパック処理して返します。リストは、まず符号無し整数によるフラグをアンパックすることで、一度に 1 要素づつアンパック処理されます。フラグが 1 の場合、要素はアンパックされ、返り値のリストに追加されます。フラグが 0 の場合、リストの終端を示します。*unpack_item* は個々の項目をアンパック処理するために呼び出される関数です。

Unpacker.**unpack_farray**(*n*, *unpack_item*)
: 一様な項目からなる固定長のアレイをアンパックして（リストとして）返します。*n* はバッファ内に存在すると期待されるリストの要素数です。上と同様に、*unpack_item* は各要素をアンパックするために使われる関数です。

Unpacker.**unpack_array**(*unpack_item*)
: 一様な項目からなる可変長の *list* をアンパックして返します。まず、リストの長さが符号無し整数としてアンパックされ、続いて各要素が上の *unpack_farray()* のようにしてアンパック処理されます。

14.4.3 例外

このモジュールでの例外はクラスインスタンスとしてコードされています:

exception xdrlib.Error
ベースとなる例外クラスです。*Error* public な属性として msg を持ち、エラーの詳細が収められています。

exception xdrlib.ConversionError
Error から派生したクラスです。インスタンス変数は追加されていません。

これらの例外を補足する方法を以下の例に示します:

```
import xdrlib
p = xdrlib.Packer()
try:
    p.pack_double(8.01)
except xdrlib.ConversionError as instance:
    print('packing the double failed:', instance.msg)
```

14.5 plistlib --- Mac OS X .plist ファイルの生成と解析

ソースコード: Lib/plistlib.py

このモジュールは主に Mac OS X で使われる「プロパティーリスト」ファイルを読み書きするインターフェイスを提供します。バイナリと XML の plist ファイルの両方をサポートします。

プロパティーリスト (.plist) ファイル形式は基本的型のオブジェクト、たとえば辞書やリスト、数、文字列など、に対する単純な直列化です。たいてい、トップレベルのオブジェクトは辞書です。

plist ファイルを書き出したり解析したりするには *dump()* や *load()* 関数を利用します。

バイトオブジェクトの plist データを扱うためには *dumps()* や *loads()* を利用します。

値は文字列、整数、浮動小数点数、ブール型、タプル、リスト、辞書 (ただし文字列だけがキーになれます)、*Data*、*bytes*、bytesarray または *datetime.datetime* のオブジェクトです。

バージョン 3.4 で変更: 新しい API。古い API は撤廃されました。バイナリ形式の plist がサポートされました。

バージョン 3.8 で変更: Support added for reading and writing *UID* tokens in binary plists as used by NSKeyedArchiver and NSKeyedUnarchiver.

参考:

PList マニュアルページ　このファイル形式の Apple の文書。

このモジュールは以下の関数を定義しています:

plistlib.load(*fp*, *, *fmt=None*, *use_builtin_types=True*, *dict_type=dict*)

plist ファイルを読み込みます。*fp* は読み込み可能かつバイナリのファイルオブジェクトです。展開されたルートオブジェクト (通常は辞書です) を返します。

fmt はファイルの形式で、次の値が有効です。

- *None*: ファイル形式を自動検出します
- *FMT_XML*: XML ファイル形式です
- *FMT_BINARY*: バイナリの plist 形式です

use_builtin_types が真 (デフォルト) の場合、バイナリデータが *bytes* のインスタンスとして返されます。偽の場合、*Data* のインスタンスとして返されます。

dict_type は、plist ファイルから読み出された辞書に使われる型です。

FMT_XML 形式の XML データは *xml.parsers.expat* にある Expat パーサーを使って解析されます。不正な形式の XML に対して送出される可能性のある例外については、そちらの文書を参照してください。plist 解析器では、未知の要素は単純に無視されます。

バイナリ形式のパーサーは、ファイルを解析できない場合に InvalidFileException を送出します。

バージョン 3.4 で追加.

plistlib.loads(*data*, *, *fmt=None*, *use_builtin_types=True*, *dict_type=dict*)

バイナリオブジェクトから plist をロードします。キーワード引数の説明については、*load()* を参照してください。

バージョン 3.4 で追加.

plistlib.dump(*value*, *fp*, *, *fmt=FMT_XML*, *sort_keys=True*, *skipkeys=False*)

plist ファイルに *value* を書き込みます。*Fp* は、書き込み可能なバイナリファイルオブジェクトにしてください。

fmt 引数は plist ファイルの形式を指定し、次のいずれかの値をとることができます。

- *FMT_XML*: XML 形式の plist ファイルです
- *FMT_BINARY*: バイナリ形式の plist ファイルです

sort_keys が真 (デフォルト) の場合、辞書内のキーは、plist にソートされた順序で書き込まれます。偽の場合、ディクショナリのイテレートの順序で書き込まれます。

skipkeys が偽 (デフォルト) の場合、この関数は辞書のキーが文字列でない場合に *TypeError* を送出します。真の場合、そのようなキーは読み飛ばされます。

TypeError が、オブジェクトがサポート外の型のものであったりサポート外の型のオブジェクトを含むコンテナだった場合に、送出されます。

(バイナリの) plist ファイル内で表現できない整数値に対しては、*OverflowError* が送出されます。

バージョン 3.4 で追加.

plistlib.**dumps**(*value*, ***, *fmt=FMT_XML*, *sort_keys=True*, *skipkeys=False*)

value を plist 形式のバイトオブジェクトとして返します。この関数のキーワード引数の説明については、*dump()* を参照してください。

バージョン 3.4 で追加.

以下の関数は廃止されています。

plistlib.**readPlist**(*pathOrFile*)

plist ファイルを読み込みます。*pathOrFile* はファイル名もしくは (読み込み可能かつバイナリの) ファイルです。展開されたルートオブジェクト (通常は辞書です) を返します。

この関数は、実際の動作のために *load()* を呼び出します。キーワード引数の説明については、**その関数** のドキュメントを参照してください。

バージョン 3.4 で非推奨: 代わりに *load()* を使ってください。

バージョン 3.7 で変更: 結果にある辞書値は通常の辞書となりました。属性アクセスを使って、辞書の項目にアクセスできなくなりました。

plistlib.**writePlist**(*rootObject*, *pathOrFile*)

rootObject を XML plist ファイルに書き込みます。*pathOrFile* は、ファイル名もしくは (書き込み可能かつバイナリの) ファイルオブジェクトです。

バージョン 3.4 で非推奨: 代わりに *dump()* を使ってください。

plistlib.**readPlistFromBytes**(*data*)

バイト列オブジェクトから plist データを読み取ります。ルートオブジェクトを返します。

キーワード引数の説明については、*load()* を参照してください。

バージョン 3.4 で非推奨: 代わりに *loads()* を使ってください。

バージョン 3.7 で変更: 結果にある辞書値は通常の辞書となりました。属性アクセスを使って、辞書の項目にアクセスできなくなりました。

plistlib.**writePlistToBytes**(*rootObject*)

rootObject を XML plist 形式のバイト列オブジェクトとして返します。

バージョン 3.4 で非推奨: 代わりに *dumps()* を使ってください。

以下のクラスが使用可能です:

class plistlib.**Data**(*data*)

バイト列オブジェクト *data* を包むラッパオブジェクトを返します。plist 中に入れられる `<data>` 型を表すものとして plist への/からの変換関数で使われます。

これには `data` という一つの属性があり、そこに収められた Python バイト列オブジェクトを取り出すのに使えます。

バージョン 3.4 で非推奨: 代わりに *bytes* オブジェクトを使ってください。

class plistlib.UID(*data*)

Wraps an *int*. This is used when reading or writing NSKeyedArchiver encoded data, which contains UID (see PList manual).

It has one attribute, `data`, which can be used to retrieve the int value of the UID. `data` must be in the range *0 <= data < 2**64*.

バージョン 3.8 で追加.

以下の定数が利用可能です:

plistlib.FMT_XML

plist ファイルの XML 形式です

バージョン 3.4 で追加.

plistlib.FMT_BINARY

plist ファイルのバイナリ形式です

バージョン 3.4 で追加.

14.5.1 使用例

plist を作ります:

```
pl = dict(
    aString = "Doodah",
    aList = ["A", "B", 12, 32.1, [1, 2, 3]],
    aFloat = 0.1,
    anInt = 728,
    aDict = dict(
        anotherString = "<hello & hi there!>",
        aThirdString = "M\xe4ssig, Ma\xdf",
        aTrueValue = True,
        aFalseValue = False,
    ),
    someData = b"<binary gunk>",
    someMoreData = b"<lots of binary gunk>" * 10,
    aDate = datetime.datetime.fromtimestamp(time.mktime(time.gmtime())),
)
with open(fileName, 'wb') as fp:
    dump(pl, fp)
```

plist を解析します:

```
with open(fileName, 'rb') as fp:
    pl = load(fp)
print(pl["aKey"])
```

第

FIFTEEN

暗号関連のサービス

この章で説明しているモジュールは、暗号的な性質の様々なアルゴリズムを実装しています。これらはインストールする際の選択によって利用可能です。Unix システムにおいては、さらに *crypt* モジュールが利用可能な場合があります。これは概観です:

15.1 hashlib --- セキュアハッシュおよびメッセージダイジェスト

ソースコード: Lib/hashlib.py

このモジュールは、セキュアハッシュやメッセージダイジェスト用のさまざまなアルゴリズムを実装したものです。FIPS のセキュアなハッシュアルゴリズムである SHA1、SHA224、SHA256、SHA384 および SHA512 (FIPS 180-2 で定義されているもの) だけでなく RSA の MD5 アルゴリズム (Internet **RFC 1321** で定義されています) も実装しています。「セキュアなハッシュ」と「メッセージダイジェスト」はどちらも同じ意味です。古くからあるアルゴリズムは「メッセージダイジェスト」と呼ばれていますが、最近は「セキュアハッシュ」という用語が用いられています。

注釈: adler32 や crc32 ハッシュ関数は *zlib* モジュールで提供されています。

> **警告:** 幾つかのアルゴリズムはハッシュの衝突に弱いことが知られています。最後の ” 参考” セクションを見てください。

15.1.1 ハッシュアルゴリズム

各 *hash* の名前が付いたコンストラクタがあります。いずれも同一で簡単なインターフェイスのあるハッシュオブジェクトを返します。例えば、SHA-256 ハッシュオブジェクトを作るには `sha256()` を使います。このオブジェクトには `update()` メソッドを用いて *bytes-like* **オブジェクト** (通常 *bytes*) を渡すことができます。`digest()` や `hexdigest()` メソッドを用いて、それまでに渡したデータを連結したものの *digest* をいつでも要求することができます。

注釈: マルチスレッドにおける良好なパフォーマンスを得るために、オブジェクトの生成時または更新時に与えるデータが 2047 バイトを超えている場合、Python *GIL* が解除されます。

注釈: 文字列オブジェクトを update() に渡すのはサポートされていません。ハッシュはバイトには機能しますが、文字には機能しないからです。

このモジュールで常に使用できるハッシュアルゴリズムのコンストラクタは sha1()、sha224()、sha256()、sha384()、sha512()、*blake2b()*、および *blake2s()* です。通常は md5() も使用できますが、もしあなたが珍しい FIPS 準拠の Python ビルドを使用しているのであれば、md5() は使用できないでしょう。Python がプラットフォームで利用している OpenSSL ライブラリによっては、追加のアルゴリズムが利用できます。多くのプラットフォームでは sha3_224`や :func:`sha3_256()、sha3_384()、sha3_512()、shake_128()、shake_256() も利用できます。

バージョン 3.6 で追加: SHA3 (Keccak) ならびに SHAKE コンストラクタ sha3_224(), sha3_256(), sha3_384(), sha3_512(), shake_128(), shake_256()。

バージョン 3.6 で追加: *blake2b()* と *blake2s()* が追加されました。

たとえば、b'Nobody inspects the spammish repetition' というバイト文字列のダイジェストを取得するには次のようにします:

```
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
b'\x03\x1e\xdd}Ae\x15\x93\xc5\xfe\\\x00o\xa5u+7\xfd\xdf\xf7\xbcN\x84:\xa6\xaf\x0c\x95\x0fK\x94\
↪x06'
>>> m.digest_size
32
>>> m.block_size
64
```

もっと簡潔に書くと、このようになります:

```
>>> hashlib.sha224(b"Nobody inspects the spammish repetition").hexdigest()
'a4337bc45a8fc544c03f52dc550cd6e1e87021bc896588bd79e901e2'
```

hashlib.new(*name*[, *data*])

一般的なコンストラクタで、第一引数にアルゴリズム名を文字列 *name* で受け取ります。他にも、前述のハッシュアルゴリズムだけでなく OpenSSL ライブラリーが提供するような他のアルゴリズムにアクセスすることができます。名前のあるコンストラクタの方が *new()* よりもずっと速いので望ましいです。

new() に OpenSSL のアルゴリズムを指定する例です:

```
>>> h = hashlib.new('sha512_256')
>>> h.update(b"Nobody inspects the spammish repetition")
>>> h.hexdigest()
'19197dc4d03829df858011c6c87600f994a858103bbc19005f20987aa19a97e2'
```

Hashlib は以下の定数属性を提供しています:

hashlib.algorithms_guaranteed

このモジュールによってすべてのプラットフォームでサポートされていることが保証されるハッシュアルゴリズムの名前を含む集合です。一部のアップストリームのベンダーが提供する奇妙な "FIPS 準拠の" Python ビルドでは md5 のサポートを除外していますが、その場合であっても 'md5' がリストに含まれることに注意してください。

バージョン 3.2 で追加.

hashlib.algorithms_available

実行中の Python インタープリタで利用可能なハッシュアルゴリズム名の set です。これらの名前は *new()* に渡すことができます。*algorithms_guaranteed* は常にサブセットです。この set の中に同じアルゴリズムが違う名前で複数回現れることがあります (OpenSSL 由来)。

バージョン 3.2 で追加.

コンストラクタが返すハッシュオブジェクトには、次のような定数属性が用意されています:

hash.digest_size

生成されたハッシュのバイト数。

hash.block_size

内部で使われるハッシュアルゴリズムのブロックのバイト数。

ハッシュオブジェクトには次のような属性があります:

hash.name

このハッシュの正規名です。常に小文字で、*new()* の引数として渡してこのタイプの別のハッシュを生成することができます。

バージョン 3.4 で変更: name 属性は CPython には最初からありましたが、Python 3.4 までは正式に明記されていませんでした。そのため、プラットフォームによっては存在しないかもしれません。

ハッシュオブジェクトには次のようなメソッドがあります:

hash.update(*data*)

hash オブジェクトを *bytes-like object* で更新します。このメソッドの呼出しの繰り返しは、それらの引数を全て結合した引数で単一の呼び出しをした際と同じになります。すなわち `m.update(a); m.update(b)` は `m.update(a + b)` と等価です。

バージョン 3.1 で変更: ハッシュアルゴリズムが OpenSSL によって提供されていて、データが 2047 バイトを超えている場合には、ハッシュの更新が実行中でも他のスレッドが実行できるように、Python *GIL* が解放されます。

hash.digest()
: これまで *update()* メソッドに渡されたデータのダイジェスト値を返します。これは *digest_size* と同じ長さの、0 から 255 の範囲全てを含み得るバイトの列です。

hash.hexdigest()
: *digest()* と似ていますが、倍の長さの、16 進形式文字列を返します。これは、電子メールなどの非バイナリ環境で値を交換する場合に便利です。

hash.copy()
: ハッシュオブジェクトのコピー (" クローン") を返します。これは、最初の部分文字列が共通なデータのダイジェストを効率的に計算するために使用します。

15.1.2 SHAKE 可変長ダイジェスト

The shake_128() and shake_256() algorithms provide variable length digests with length_in_bits//2 up to 128 or 256 bits of security. As such, their digest methods require a length. Maximum length is not limited by the SHAKE algorithm.

shake.digest(*length*)
: これまで update() メソッドに渡されたデータのダイジェスト値を返します。これは *length* と同じ長さの、0 から 255 の範囲全てを含み得るバイトの列です。

shake.hexdigest(*length*)
: *digest()* と似ていますが、倍の長さの、16 進形式文字列を返します。これは、電子メールなどの非バイナリ環境で値を交換する場合に便利です。

15.1.3 鍵導出

鍵の導出 (derivation) と引き伸ばし (stretching) のアルゴリズムはセキュアなパスワードのハッシュ化のために設計されました。sha1(password) のような甘いアルゴリズムは、ブルートフォース攻撃に抵抗できません。良いパスワードハッシュ化は調節可能で、遅くて、salt を含まなければなりません。

hashlib.pbkdf2_hmac(*hash_name*, *password*, *salt*, *iterations*, *dklen=None*)
: この関数は PKCS#5 のパスワードに基づいた鍵導出関数 2 を提供しています。疑似乱数関数として HMAC を使用しています。

  文字列 *hash_name* は、HMAC のハッシュダイジェストアルゴリズムの望ましい名前で、例えば 'sha1' や 'sha256' です。*password* と *salt* はバイト列のバッファとして解釈されます。アプリケーションとライブラリは、*password* を適切な長さ (例えば 1024) に制限すべきです。*salt* は *os.urandom()* のような適切なソースからの、およそ 16 バイトかそれ以上のバイト列にするべきです。

  iterations 数はハッシュアルゴリズムと計算機の能力に基づいて決めるべきです。2013 年現在の場合、SHA-256 に対して最低でも 100,000 反復が推奨されています。

  dklen は、導出された鍵の長さです。*dklen* が None の場合、ハッシュアルゴリズム *hash_name* のダイジェストサイズが使われます。例えば SHA-512 では 64 です。

```
>>> import hashlib
>>> dk = hashlib.pbkdf2_hmac('sha256', b'password', b'salt', 100000)
>>> dk.hex()
'0394a2ede332c9a13eb82e9b24631604c31df978b4e2f0fbd2c549944f9d79a5'
```

バージョン 3.4 で追加.

注釈: *pbkdf2_hmac* の高速な実装は OpenSSL 使用版で利用可能です。Python 実装は *hmac* のインラインバージョンを使います。それはおよそ 3 倍遅く、GIL を解放しません。

---

hashlib.**scrypt**(*password*, ***, *salt*, *n*, *r*, *p*, *maxmem=0*, *dklen=64*)

この関数は、**RFC 7914** で定義される scrypt のパスワードに基づいた鍵導出関数を提供します。

password と *salt* は bytes-like objects でなければなりません。アプリケーションとライブラリは、*password* を適切な長さ (例えば 1024) に制限すべきです。*salt* は *os.urandom()* のような適切なソースからの、およそ 16 バイトかそれ以上のバイト列にするべきです。

n is the CPU/Memory cost factor, *r* the block size, *p* parallelization factor and *maxmem* limits memory (OpenSSL 1.1.0 defaults to 32 MiB). *dklen* is the length of the derived key.

Availability: OpenSSL 1.1+.

バージョン 3.6 で追加.

15.1.4 BLAKE2

BLAKE2 is a cryptographic hash function defined in **RFC 7693** that comes in two flavors:

- **BLAKE2b**, optimized for 64-bit platforms and produces digests of any size between 1 and 64 bytes,
- **BLAKE2s**, optimized for 8- to 32-bit platforms and produces digests of any size between 1 and 32 bytes.

BLAKE2 supports **keyed mode** (a faster and simpler replacement for HMAC), **salted hashing**, **personalization**, and **tree hashing**.

このモジュールのハッシュオブジェクトは標準ライブラリーの *hashlib* オブジェクトの API に従います。

ハッシュオブジェクトの作成

新しいハッシュオブジェクトは、コンストラクタ関数を呼び出すことで生成されます:

hashlib.blake2b(*data=b''*, ***, *digest_size=64*, *key=b''*, *salt=b''*, *person=b''*, *fanout=1*, *depth=1*, *leaf_size=0*, *node_offset=0*, *node_depth=0*, *inner_size=0*, *last_node=False*)

hashlib.blake2s(*data=b''*, ***, *digest_size=32*, *key=b''*, *salt=b''*, *person=b''*, *fanout=1*, *depth=1*, *leaf_size=0*, *node_offset=0*, *node_depth=0*, *inner_size=0*, *last_node=False*)

These functions return the corresponding hash objects for calculating BLAKE2b or BLAKE2s. They optionally take these general parameters:

- *data*: initial chunk of data to hash, which must be *bytes-like object*. It can be passed only as positional argument.
- *digest_size*: 出力するダイジェストのバイト数。
- *key*: key for keyed hashing (up to 64 bytes for BLAKE2b, up to 32 bytes for BLAKE2s).
- *salt*: salt for randomized hashing (up to 16 bytes for BLAKE2b, up to 8 bytes for BLAKE2s).
- *person*: personalization string (up to 16 bytes for BLAKE2b, up to 8 bytes for BLAKE2s).

下の表は一般的なパラメータの上限 (バイト単位) です:

Hash	digest_size	len(key)	len(salt)	len(person)
BLAKE2b	64	64	16	16
BLAKE2s	32	32	8	8

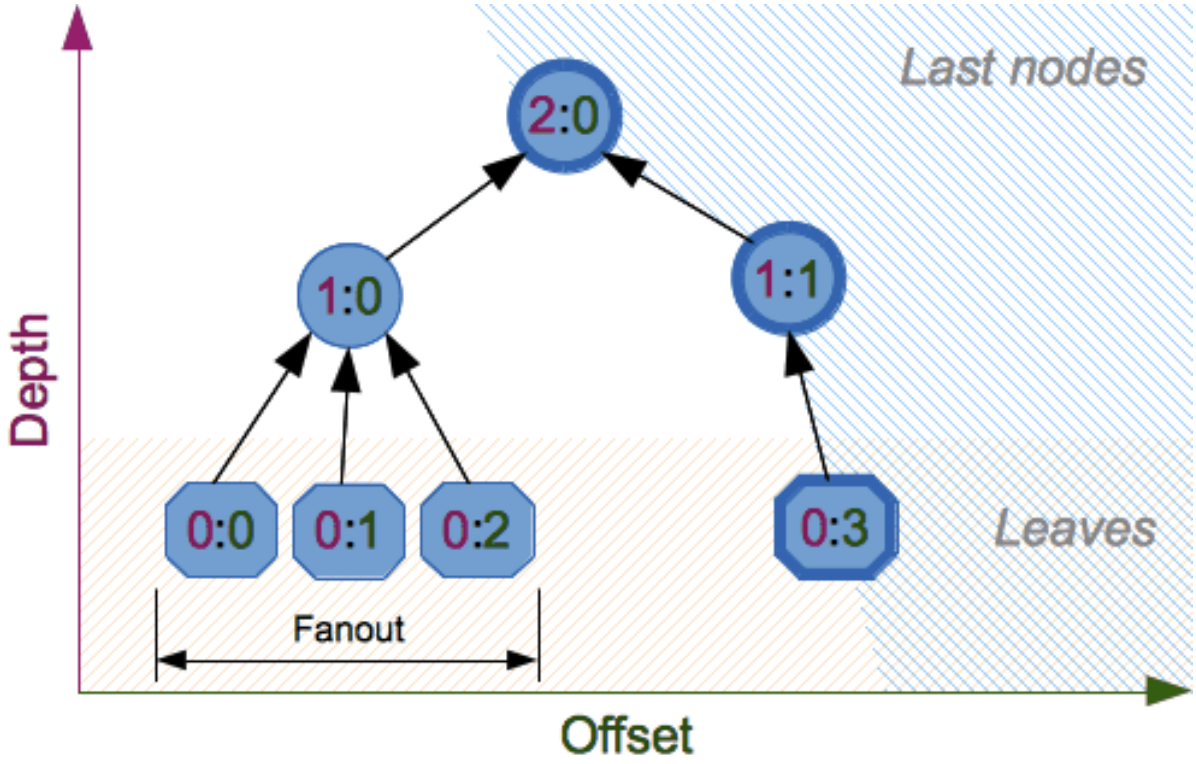
注釈: BLAKE2 specification defines constant lengths for salt and personalization parameters, however, for convenience, this implementation accepts byte strings of any size up to the specified length. If the length of the parameter is less than specified, it is padded with zeros, thus, for example, `b'salt'` and `b'salt\x00'` is the same value. (This is not the case for *key*.)

これらのサイズは以下に述べるモジュール *constants* で利用できます。

コンストラクタ関数は以下のツリーハッシングパラメタを受け付けます:

- *fanout*: fanout (0 to 255, 0 if unlimited, 1 in sequential mode).
- *depth*: ツリーの深さの最大値（1 から 255 までの値で、無制限であれば 255、シーケンスモードでは 1)。
- *leaf_size*: maximal byte length of leaf (0 to 2**32-1, 0 if unlimited or in sequential mode).
- *node_offset*: node offset (0 to 2**64-1 for BLAKE2b, 0 to 2**48-1 for BLAKE2s, 0 for the first, leftmost, leaf, or in sequential mode).
- *node_depth*: node depth (0 to 255, 0 for leaves, or in sequential mode).

- *inner_size*: inner digest size (0 to 64 for BLAKE2b, 0 to 32 for BLAKE2s, 0 in sequential mode).
- *last_node*: boolean indicating whether the processed node is the last one (*False* for sequential mode).



See section 2.10 in BLAKE2 specification for comprehensive review of tree hashing.

定数

`blake2b.SALT_SIZE`

`blake2s.SALT_SIZE`

ソルト長（コンストラクタが受け付けれる最大長）

`blake2b.PERSON_SIZE`

`blake2s.PERSON_SIZE`

Personalization string length (maximum length accepted by constructors).

`blake2b.MAX_KEY_SIZE`

`blake2s.MAX_KEY_SIZE`

最大キー長

`blake2b.MAX_DIGEST_SIZE`

`blake2s.MAX_DIGEST_SIZE`

ハッシュ関数が出力しうるダイジェストの最大長

使用例

簡単なハッシュ化

To calculate hash of some data, you should first construct a hash object by calling the appropriate constructor function (*blake2b()* or *blake2s()*), then update it with the data by calling `update()` on the object, and, finally, get the digest out of the object by calling `digest()` (or `hexdigest()` for hex-encoded string).

```
>>> from hashlib import blake2b
>>> h = blake2b()
>>> h.update(b'Hello world')
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc81039e3eeb
↪'
```

As a shortcut, you can pass the first chunk of data to update directly to the constructor as the positional argument:

```
>>> from hashlib import blake2b
>>> blake2b(b'Hello world').hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc81039e3eeb
↪'
```

You can call *hash.update()* as many times as you need to iteratively update the hash:

```
>>> from hashlib import blake2b
>>> items = [b'Hello', b' ', b'world']
>>> h = blake2b()
>>> for item in items:
...     h.update(item)
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc81039e3eeb
↪'
```

Using different digest sizes

BLAKE2 はダイジェストの長さを、BLAKE2b では 64 バイトまで、BLAKE2s では 32 バイトまでのダイジェスト長を指定できます。例えば SHA-1 を、出力を同じ長さのままで BLAKE2b で置き換えるには、BLAKE2b に 20 バイトのダイジェストを生成するように指示できます:

```
>>> from hashlib import blake2b
>>> h = blake2b(digest_size=20)
```

(次のページに続く)

(前のページからの続き)

```
>>> h.update(b'Replacing SHA1 with the more secure function')
>>> h.hexdigest()
'd24f26cf8de66472d58d4e1b1774b4c9158b1f4c'
>>> h.digest_size
20
>>> len(h.digest())
20
```

Hash objects with different digest sizes have completely different outputs (shorter hashes are *not* prefixes of longer hashes); BLAKE2b and BLAKE2s produce different outputs even if the output length is the same:

```
>>> from hashlib import blake2b, blake2s
>>> blake2b(digest_size=10).hexdigest()
'6fa1d8fcfd719046d762'
>>> blake2b(digest_size=11).hexdigest()
'eb6ec15daf9546254f0809'
>>> blake2s(digest_size=10).hexdigest()
'1bf21a98c78a1c376ae9'
>>> blake2s(digest_size=11).hexdigest()
'567004bf96e4a25773ebf4'
```

Keyed hashing

Keyed hashing can be used for authentication as a faster and simpler replacement for Hash-based message authentication code (HMAC). BLAKE2 can be securely used in prefix-MAC mode thanks to the indifferentiability property inherited from BLAKE.

This example shows how to get a (hex-encoded) 128-bit authentication code for message `b'message data'` with key `b'pseudorandom key'`:

```
>>> from hashlib import blake2b
>>> h = blake2b(key=b'pseudorandom key', digest_size=16)
>>> h.update(b'message data')
>>> h.hexdigest()
'3d363ff7401e02026f4a4687d4863ced'
```

As a practical example, a web application can symmetrically sign cookies sent to users and later verify them to make sure they weren't tampered with:

```
>>> from hashlib import blake2b
>>> from hmac import compare_digest
>>>
>>> SECRET_KEY = b'pseudorandomly generated server secret key'
>>> AUTH_SIZE = 16
>>>
>>> def sign(cookie):
...     h = blake2b(digest_size=AUTH_SIZE, key=SECRET_KEY)
```

(次のページに続く)

(前のページからの続き)

```
...     h.update(cookie)
...     return h.hexdigest().encode('utf-8')
>>>
>>> def verify(cookie, sig):
...     good_sig = sign(cookie)
...     return compare_digest(good_sig, sig)
>>>
>>> cookie = b'user-alice'
>>> sig = sign(cookie)
>>> print("{0},{1}".format(cookie.decode('utf-8'), sig))
user-alice,b'43b3c982cf697e0c5ab22172d1ca7421'
>>> verify(cookie, sig)
True
>>> verify(b'user-bob', sig)
False
>>> verify(cookie, b'0102030405060708090a0b0c0d0e0f00')
False
```

Even though there's a native keyed hashing mode, BLAKE2 can, of course, be used in HMAC construction with *hmac* module:

```
>>> import hmac, hashlib
>>> m = hmac.new(b'secret key', digestmod=hashlib.blake2s)
>>> m.update(b'message')
>>> m.hexdigest()
'e3c8102868d28b5ff85fc35dda07329970d1a01e273c37481326fe0c861c8142'
```

Randomized hashing

By setting *salt* parameter users can introduce randomization to the hash function. Randomized hashing is useful for protecting against collision attacks on the hash function used in digital signatures.

> Randomized hashing is designed for situations where one party, the message preparer, generates all or part of a message to be signed by a second party, the message signer. If the message preparer is able to find cryptographic hash function collisions (i.e., two messages producing the same hash value), then they might prepare meaningful versions of the message that would produce the same hash value and digital signature, but with different results (e.g., transferring $1,000,000 to an account, rather than $10). Cryptographic hash functions have been designed with collision resistance as a major goal, but the current concentration on attacking cryptographic hash functions may result in a given cryptographic hash function providing less collision resistance than expected. Randomized hashing offers the signer additional protection by reducing the likelihood that a preparer can generate two or more messages that ultimately yield the same hash value during the digital signature generation process --- even if it is practical to find collisions for the hash function. However, the use of randomized hashing may reduce the amount of security provided by a digital signature when all portions of the message are prepared by the signer.
>
> (NIST SP-800-106 "Randomized Hashing for Digital Signatures")

In BLAKE2 the salt is processed as a one-time input to the hash function during initialization, rather than as an input to each compression function.

警告: *Salted hashing* (or just hashing) with BLAKE2 or any other general-purpose cryptographic hash function, such as SHA-256, is not suitable for hashing passwords. See BLAKE2 FAQ for more information.

```
>>> import os
>>> from hashlib import blake2b
>>> msg = b'some message'
>>> # Calculate the first hash with a random salt.
>>> salt1 = os.urandom(blake2b.SALT_SIZE)
>>> h1 = blake2b(salt=salt1)
>>> h1.update(msg)
>>> # Calculate the second hash with a different random salt.
>>> salt2 = os.urandom(blake2b.SALT_SIZE)
>>> h2 = blake2b(salt=salt2)
>>> h2.update(msg)
>>> # The digests are different.
>>> h1.digest() != h2.digest()
True
```

Personalization

Sometimes it is useful to force hash function to produce different digests for the same input for different purposes. Quoting the authors of the Skein hash function:

> We recommend that all application designers seriously consider doing this; we have seen many protocols where a hash that is computed in one part of the protocol can be used in an entirely different part because two hash computations were done on similar or related data, and the attacker can force the application to make the hash inputs the same. Personalizing each hash function used in the protocol summarily stops this type of attack.
>
> (The Skein Hash Function Family, p. 21)

BLAKE2 は *person* 引数にバイト列を渡すことでパーソナライズできます:

```
>>> from hashlib import blake2b
>>> FILES_HASH_PERSON = b'MyApp Files Hash'
>>> BLOCK_HASH_PERSON = b'MyApp Block Hash'
>>> h = blake2b(digest_size=32, person=FILES_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'20d9cd024d4fb086aae819a1432dd2466de12947831b75c5a30cf2676095d3b4'
>>> h = blake2b(digest_size=32, person=BLOCK_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'cf68fb5761b9c44e7878bfb2c4c9aea52264a80b75005e65619778de59f383a3'
```

Personalization together with the keyed mode can also be used to derive different keys from a single one.

```
>>> from hashlib import blake2s
>>> from base64 import b64decode, b64encode
>>> orig_key = b64decode(b'Rm5EPJai72qcK3RGBpW3vPNfZy5OZothY+kHY6h21KM=')
>>> enc_key = blake2s(key=orig_key, person=b'kEncrypt').digest()
>>> mac_key = blake2s(key=orig_key, person=b'kMAC').digest()
>>> print(b64encode(enc_key).decode('utf-8'))
rbPb15S/Z9t+agffno5wuhB77VbRi6F9Iv2qIxU7WHw=
>>> print(b64encode(mac_key).decode('utf-8'))
G9GtHFE1YluXY1zWPlYk1e/nWfu0WSEb0KRcjhDeP/o=
```

ツリーモード

これが二つの葉ノードからなる最小の木をハッシュする例です:

```
  10
 /  \
00  01
```

次の例では、64 バイトの内部桁が使われ、32 バイトの最終的なダイジェストを返しています:

```
>>> from hashlib import blake2b
>>>
>>> FANOUT = 2
>>> DEPTH = 2
>>> LEAF_SIZE = 4096
>>> INNER_SIZE = 64
>>>
>>> buf = bytearray(6000)
>>>
>>> # Left leaf
... h00 = blake2b(buf[0:LEAF_SIZE], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=0, last_node=False)
>>> # Right leaf
... h01 = blake2b(buf[LEAF_SIZE:], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=1, node_depth=0, last_node=True)
>>> # Root node
... h10 = blake2b(digest_size=32, fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=1, last_node=True)
>>> h10.update(h00.digest())
>>> h10.update(h01.digest())
>>> h10.hexdigest()
'3ad2a9b37c6070e374c7a8c508fe20ca86b6ed54e286e93a0318e95e881db5aa'
```

クレジット:

BLAKE2 は*Jean-Philippe Aumasson*、*Luca Henzen*、*Willi Meier* そして *Raphael C.-W. Phan* によって作成された SHA-3 の最終候補である BLAKE を元に、*Jean-Philippe Aumasson*、*Samuel Neves*、*Zooko Wilcox-O'Hearn*, そして *Christian Winnerlein* によって設計されました。

それは、*Daniel J. Bernstein* によって設計された ChaCha 暗号由来のアルゴリズムを用いています。

標準ライブラリは pyblake2 モジュールを基礎として実装されています。このモジュールは *Dmitry Chestnykh* によって、*Samuel Neves* が作成した C 実装を元に書かれました。このドキュメントは、pyblake2 からコピーされ、*Dmitry Chestnykh* によって書かれました。

Christian Heimes によって、一部の C コードが Python 向けに書き直されました。

以下の public domain dedication が、C のハッシュ関数実装と、拡張コードと、このドキュメントに適用されます:

> To the extent possible under law, the author(s) have dedicated all copyright and related and neighboring rights to this software to the public domain worldwide. This software is distributed without any warranty.
>
> You should have received a copy of the CC0 Public Domain Dedication along with this software. If not, see https://creativecommons.org/publicdomain/zero/1.0/.

The following people have helped with development or contributed their changes to the project and the public domain according to the Creative Commons Public Domain Dedication 1.0 Universal:

- *Alexandr Sokolovskiy*

参考:

hmac **モジュール**　ハッシュを用いてメッセージ認証コードを生成するモジュールです。

base64 **モジュール**　バイナリハッシュを非バイナリ環境用にエンコードするもうひとつの方法です。

https://blake2.net　BLAKE2 の公式ウェブサイト

https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf
　FIPS 180-2 のセキュアハッシュアルゴリズムについての説明。

https://en.wikipedia.org/wiki/Cryptographic_hash_function#Cryptographic_hash_algorithms (**日本語版**: https://**暗号学的ハ**
　どのアルゴリズムにどんな既知の問題があって、それが実際に利用する際にどう影響するかについての Wikipedia の記事。

https://www.ietf.org/rfc/rfc2898.txt　PKCS #5: Password-Based Cryptography Specification Version 2.0

15.2 hmac --- メッセージ認証のための鍵付きハッシュ化

ソースコード: Lib/hmac.py

このモジュールでは **RFC 2104** で記述されている HMAC アルゴリズムを実装しています。

hmac.new(*key*, *msg=None*, *digestmod=''*)

Return a new hmac object. *key* is a bytes or bytearray object giving the secret key. If *msg* is present, the method call `update(msg)` is made. *digestmod* is the digest name, digest constructor or module for the HMAC object to use. It may be any name suitable to *hashlib.new()*. Despite its argument position, it is required.

バージョン 3.4 で変更: 引数 *key* に bytes または bytearray オブジェクトを渡せるようになりました。引数 *msg* に *hashlib* がサポートする全てのタイプを渡せるようになりました。引数 *digestmod* にハッシュアルゴリズム名を渡せるようになりました。

Deprecated since version 3.4, removed in version 3.8: MD5 as implicit default digest for *digestmod* is deprecated. The digestmod parameter is now required. Pass it as a keyword argument to avoid awkwardness when you do not have an initial msg.

hmac.digest(*key*, *msg*, *digest*)

与えられた secret *key* と *digest* の *msg* のダイジェストを返します。この関数は `HMAC(key, msg, digest).digest()` に似ていますが、最適化された C やインラインの実装を使用しており、メモリに収まるメッセージに対しては高速です。パラメータ *key* 、*msg* 、および *digest* は、*new()* と同じ意味を持ちます。

CPython 実装の詳細、最適化された C 実装は、OpenSSL がサポートするダイジェストアルゴリズムの文字列と名前が *digest* の場合にのみ使用されます。

バージョン 3.7 で追加.

HMAC オブジェクトは以下のメソッドを持っています:

HMAC.update(*msg*)

hmac オブジェクトを *msg* で更新します。このメソッドの呼出の繰り返しは、それらの引数を全て結合した引数で単一の呼び出しをした際と同じになります。すなわち `m.update(a); m.update(b)` は `m.update(a + b)` と等価です。

バージョン 3.4 で変更: 引数 *msg* は *hashlib* がサポートしているあらゆる型のいずれかです。

HMAC.digest()

これまで *update()* メソッドに渡されたバイト列のダイジェスト値を返します。これはコンストラクタに与えられた *digest_size* と同じ長さのバイト列で、NUL バイトを含む非 ASCII 文字が含まれることがあります。

> **警告:** *digest()* の出力結果と外部から供給されたダイジェストを検証ルーチン内で比較しようとするのであれば、タイミング攻撃への脆弱性を減らすために、== 演算子ではなく *compare_digest()* を使うことをお奨めします。

HMAC.**hexdigest**()
: *digest()* と似ていますが、返される文字列は倍の長さとなり、16 進形式となります。これは、電子メールなどの非バイナリ環境で値を交換する場合に便利です。

> **警告:** *hexdigest()* の出力結果と外部から供給されたダイジェストを検証ルーチン内で比較しようとするのであれば、タイミング攻撃への脆弱性を減らすために、== 演算子ではなく *compare_digest()* を使うことをお奨めします。

HMAC.**copy**()
: hmac オブジェクトのコピー (" クローン") を返します。このコピーは最初の部分文字列が共通になっている文字列のダイジェスト値を効率よく計算するために使うことができます。

ハッシュオブジェクトには次のような属性があります:

HMAC.**digest_size**
: 生成された HMAC ダイジェストのバイト数。

HMAC.**block_size**
: 内部で使われるハッシュアルゴリズムのブロックのバイト数。

  バージョン 3.4 で追加.

HMAC.**name**
: この HMAC の正規名で、例えば `hmac-md5` のように常に小文字です。

  バージョン 3.4 で追加.

このモジュールは以下のヘルパ関数も提供しています:

hmac.**compare_digest**(*a*, *b*)
: `a == b` を返します。この関数は、内容ベースの短絡的な振る舞いを避けることによってタイミング分析を防ぐよう設計されたアプローチを用い、暗号化に用いるのに相応しいものとしています。*a* と *b* は両方が同じ型でなければなりません: (例えば *HMAC.hexdigest()* が返したような ASCII のみの) *str* または *bytes-like object* のどちらか一方。

  > **注釈:** *a* と *b* が異なる長さであったりエラーが発生した場合には、タイミング攻撃で理論上 *a* と *b* の型と長さについての情報が暴露されますが、その値は明らかになりません。

  バージョン 3.3 で追加.

参考:

hashlib **モジュール**　セキュアハッシュ関数を提供する Python モジュールです。

15.3 secrets --- 機密を扱うために安全な乱数を生成する

バージョン 3.6 で追加.

ソースコード: Lib/secrets.py

---

secrets モジュールを使って、パスワードやアカウント認証、セキュリティトークンなどの機密を扱うのに適した、暗号学的に強い乱数を生成することができます。

特に、*random* モジュールのデフォルトの擬似乱数よりも *secrets* を使用するべきです。*random* モジュールはモデル化やシミュレーション向けで、セキュリティや暗号学的に設計されてはいません。

参考:

PEP 506

15.3.1 乱数

secrets モジュールは OS が提供する最も安全な乱雑性のソースへのアクセスを提供します。

class secrets.SystemRandom

OS が提供する最も高品質なソースを用いて乱数を生成するためのクラスです。更に詳しいことについては *random.SystemRandom* を参照してください。

secrets.choice(*sequence*)

空でないシーケンスから要素をランダムに選択して返します。

secrets.randbelow(n)

$[0, n)$ のランダムな整数を返します。

secrets.randbits(k)

ランダムな k ビットの整数を返します。

15.3.2 トークンの生成

secrets モジュールはパスワードのリセットや想像しにくい URL などの用途に適した、安全なトークンを生成するための関数を提供します。

secrets.token_bytes([*nbytes=None*])

nbytes バイトを含むバイト文字列を返します。*nbytes* が None の場合や与えられなかった場合は妥当なデフォルト値が使われます。

```
>>> token_bytes(16)
b'\xebr\x17D*t\xae\xd4\xe3S\xb6\xe2\xebP1\x8b'
```

secrets.**token_hex**([*nbytes=None*])

十六進数のランダムなテキスト文字列を返します。文字列は *nbytes* のランダムなバイトを持ち、各バイトは二つの十六進数に変換されます。*nbytes* が None の場合や与えられなかった場合は妥当なデフォルト値が使われます。

```
>>> token_hex(16)
'f9bf78b9a18ce6d46a0cd2b0b86df9da'
```

secrets.**token_urlsafe**([*nbytes=None*])

nbytes のランダムなバイトを持つ URL 安全なテキスト文字列を返します。テキストは Base64 でエンコードされていて、平均的に各バイトは約 1.3 文字になります。*nbytes* が None の場合や与えられなかった場合は妥当なデフォルト値が使われます。

```
>>> token_urlsafe(16)
'Drmhze6EPcv0fN_81Bj-nA'
```

トークンは何バイト使うべきか？

総当たり攻撃 に耐えるには、トークンは十分にランダムでなければなりません。残念なことに、コンピュータの性能が向上し、より短時間により多くの推測ができるようになるにつれ、十分とされるランダムさというのは必然的に増えます。2015 年の時点で、*secrets* モジュールに想定される通常の用途では、32 バイト (256 ビット) のランダムさは十分と考えられています。

独自の長さのトークンを扱いたい場合、様々な token_* 関数に *int* 引数で渡すことで、トークンに使用するランダムさを明示的に指定することができます。引数はランダムさのバイト数として使用されます。

それ以外の場合、すなわち引数がない場合や None の場合、token_* 関数は妥当なデフォルト値を代わりに使います。

注釈: デフォルトはメンテナンスリリースの間を含め、いつでも変更される可能性があります。

15.3.3 その他の関数

secrets.**compare_digest**(*a*, *b*)

文字列 *a* と *b* が等しければ True を、そうでなければ False を返します。比較は タイミング攻撃 のリスクを減らす方法で行われます。詳細については *hmac.compare_digest()* を参照してください。

15.3.4 レシピとベストプラクティス

この節では *secrets* を使用してセキュリティの基礎的なレベルを扱う際のレシピとベストプラクティスを説明します。

8 文字のアルファベットと数字を含むパスワードを生成するには:

```
import string
import secrets
alphabet = string.ascii_letters + string.digits
password = ''.join(secrets.choice(alphabet) for i in range(8))
```

注釈: アプリケーションは、平文であろうと暗号化されていようと、復元可能な形式でパスワードを保存 してはいけません。パスワードは暗号学的に強い一方向 (非可逆) ハッシュ関数を用いてソルトしハッシュしなければなりません。

アルファベットと数字からなり、小文字を少なくとも 1 つと数字を少なくとも 3 つ含む、10 文字のパスワードを生成するには:

```
import string
import secrets
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(secrets.choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
            and any(c.isupper() for c in password)
            and sum(c.isdigit() for c in password) >= 3):
        break
```

XKCD スタイルのパスフレーズ を生成するには:

```
import secrets
# On standard Linux systems, use a convenient dictionary file.
# Other platforms may need to provide their own word-list.
with open('/usr/share/dict/words') as f:
    words = [word.strip() for word in f]
    password = ' '.join(secrets.choice(words) for i in range(4))
```

パスワードの復元用途に適したセキュリティトークンを含む、推測しにくい一時 URL を生成するには:

```
import secrets
url = 'https://mydomain.com/reset=' + secrets.token_urlsafe()
```

第

SIXTEEN

汎用オペレーティングシステムサービス

本章に記述されたモジュールは、ファイルの取り扱いや時間計測のような (ほぼ) すべてのオペレーティングシステムで利用可能な機能にインタフェースを提供します。これらのインタフェースは、Unix もしくは C のインタフェースを基に作られますが、ほとんどの他のシステムで同様に利用可能です。概要を以下に記述します:

16.1 os --- 雑多なオペレーティングシステムインタフェース

ソースコード: Lib/os.py

このモジュールは、OS 依存の機能を利用するポータブルな方法を提供します。単純なファイルの読み書きについては、*open()* を参照してください。パス操作については、*os.path* モジュールを参照してください。コマンドラインに与えられたすべてのファイルから行を読み込んでいくには、*fileinput* モジュールを参照してください。一時ファイルや一時ディレクトリの作成については、*tempfile* モジュールを参照してください。高水準のファイルとディレクトリの操作については、*shutil* モジュールを参照してください。

利用可能性に関する注意：

- Python の、すべての OS 依存モジュールの設計方針は、可能な限り同一のインタフェースで同一の機能を利用できるようにする、というものです。例えば、`os.stat(path)` は *path* に関する stat 情報を、(POSIX を元にした) 同じフォーマットで返します。
- 特定のオペレーティングシステム固有の拡張も *os* を介して利用することができますが、これらの利用はもちろん、可搬性を脅かします。
- パスやファイル名を受け付けるすべての関数は、バイト列型および文字列型両方のオブジェクトを受け付け、パスやファイル名を返す時は、同じ型のオブジェクトを返します。
- VxWorks では、os.fork, os.execv および os.spawn*p* はサポートされていません。

注釈: このモジュール内のすべての関数は、間違った、あるいはアクセス出来ないファイル名やファイルパス、その他型が合っていても OS が受理しない引数に対して、*OSError* (またはそのサブクラス) を送出します。

---

exception os.error
: 組み込みの *OSError* 例外に対するエイリアスです。

os.name
: import されているオペレーティングシステムに依存するモジュールの名前です。現在次の名前が登録されています: 'posix', 'nt', 'java'。

  参考:

  sys.platform はより細かな粒度を持っています。*os.uname()* はシステム依存のバージョン情報を提供します。

  platform モジュールはシステムの詳細な識別情報をチェックする機能を提供しています。

16.1.1 ファイル名、コマンドライン引数、および環境変数

Python では、ファイル名、コマンドライン引数、および環境変数を表すのに文字列型を使用します。一部のシステムでは、これらをオペレーティングシステムに渡す前に、文字列からバイト列へ、またはその逆のデコードが必要です。Python はこの変換を行うためにファイルシステムのエンコーディングを使用します (*sys.getfilesystemencoding()* 参照)。

バージョン 3.1 で変更: 一部のシステムでは、ファイルシステムのエンコーディングを使用して変換すると失敗する場合があります。この場合、Python は *surrogateescape* **エンコーディングエラーハンドラー** を使用します。これは、デコード時にデコードできないバイト列は Unicode 文字 U+DCxx に置き換えられ、それらはエンコード時に再び元のバイト列に変換されることを意味します。

ファイルシステムのエンコーディングでは、すべてが 128 バイト以下に正常にデコードされることが保証されなくてはなりません。ファイルシステムのエンコーディングでこれが保証されなかった場合は、API 関数が UnicodeError を送出します。

16.1.2 プロセスのパラメーター

これらの関数とデータアイテムは、現在のプロセスおよびユーザーに対する情報提供および操作のための機能を提供しています。

os.ctermid()
: プロセスの制御端末に対応するファイル名を返します。

  利用可能な環境: Unix。

os.environ
: 文字列の環境を表す **マップ型** オブジェクトです。例えば、environ['HOME'] は (一部のプラットフォームでは) ホームディレクトリのパス名であり、C における getenv("HOME") と等価です。

  このマップ型の内容は、*os* モジュールの最初の import の時点、通常は Python の起動時に site.py が処理される中で取り込まれます。それ以後に変更された環境変数は os.environ を直接変更しない限り os.environ には反映されません。

プラットフォーム上で *putenv()* がサポートされている場合、このマップ型オブジェクトは環境変数に対する変更に使うこともできます。*putenv()* はマップ型オブジェクトが修正される時に、自動的に呼ばれることになります。

Unix では、キーと値に *sys.getfilesystemencoding()*、エラーハンドラーに 'surrogateescape' を使用します。異なるエンコーディングを使用したい場合は *environb* を使用します。

注釈: *putenv()* を直接呼び出しても os.environ の内容は変わらないので、os.environ を直接変更する方が良いです。

注釈: FreeBSD と Mac OS X を含む一部のプラットフォームでは、environ の値を変更するとメモリリークの原因になる場合があります。システムの putenv() に関するドキュメントを参照してください。

---

putenv() が提供されていない場合、このマップ型オブジェクトに変更を加えたコピーを適切なプロセス生成機能に渡すことで、生成された子プロセスが変更された環境変数を利用するようにできます。

プラットフォームが *unsetenv()* 関数をサポートしている場合、このマップ型オブジェクトからアイテムを削除することで環境変数を消すことができます。*unsetenv()* は os.environ からアイテムが取り除かれた時に自動的に呼ばれます。pop() または clear() が呼ばれた時も同様です。

os.environb

environ のバイト列版です。環境変数をバイト文字列で表す **マップ型** オブジェクトです。*environ* と *environb* は同期されます。(*environb* を変更すると *environ* が更新され、逆の場合も同様に更新されます)。

environb は *supports_bytes_environ* が True の場合のみ利用可能です。

バージョン 3.2 で追加.

os.chdir(*path*)

os.fchdir(*fd*)

os.getcwd()

これらの関数は、**ファイルとディレクトリ** 節で説明されています。

os.fsencode(*filename*)

path-like な *filename* をファイルシステムのエンコーディングにエンコードします。エラーハンドラーに 'surrogateescape' (Windows の場合は 'strict') が指定されます; 未変更の *bytes* オブジェクトを返します。

fsdecode() はこの逆変換を行う関数です。

バージョン 3.2 で追加.

バージョン 3.6 で変更: *os.PathLike* インタフェースを実装したオブジェクトを受け入れるようになりました。

os.fsdecode(*filename*)

ファイルシステムのエンコーディングから path-like な *filename* にデコードします。エラーハンドラーに 'surrogateescape' (Windows の場合は 'strict') が指定されます; 未変更の *bytes* オブジェクトを返します。

fsencode() はこの逆変換を行う関数です。

バージョン 3.2 で追加.

バージョン 3.6 で変更: *os.PathLike* インタフェースを実装したオブジェクトを受け入れるようになりました。

os.fspath(*path*)

path のファイルシステム表現を返します。

もし *str* か bytes: のオブジェクトが渡された場合は、変更せずにそのまま返します。さもなければ、*__fspath__()* が呼び出され、その戻り値が *str* か *bytes* のオブジェクトであれば、その値を返します。他のすべてのケースでは *TypeError* が送出されます。

バージョン 3.6 で追加.

class os.PathLike

ファイルシステムパスを表すオブジェクト (例: *pathlib.PurePath*) 向けの *abstract base class* です。

バージョン 3.6 で追加.

abstractmethod __fspath__()

このオブジェクトが表現するファイルシステムパスを返します。

このメソッドは *str* か *bytes* のオブジェクトのみを返す必要があります (*str* が好まれます)。

os.getenv(*key*, *default=None*)

環境変数 *key* が存在すればその値を返し、存在しなければ *default* を返します。*key*、*default*、および返り値は文字列です。

Unix では、キーと値は *sys.getfilesystemencoding()*、エラーハンドラー 'surrogateescape' でデコードされます。異なるエンコーディングを使用したい場合は *os.getenvb()* を使用します。

利用できる環境: 主な Unix 互換環境、Windows。

os.getenvb(*key*, *default=None*)

環境変数 *key* が存在すればその値を返し、存在しなければ *default* を返します。*key*、*default*、および返り値はバイト列型です。

getenvb() は *supports_bytes_environ* が "True"の場合のみ利用可能です。

利用できる環境: 主な Unix 互換環境。

バージョン 3.2 で追加.

os.get_exec_path(*env=None*)

プロセスを起動する時に名前付き実行ファイルを検索するディレクトリのリストを返します。*env* が指

定されると、それを環境変数の辞書とみなし、その辞書からキー PATH の値を探します。デフォルトでは *env* は None であり、*environ* が使用されます。

バージョン 3.2 で追加.

os.getegid()
現在のプロセスの実効グループ id を返します。この id は現在のプロセスで実行されているファイルの "set id" ビットに対応します。

利用可能な環境: Unix。

os.geteuid()
現在のプロセスの実効ユーザー id を返します。

利用可能な環境: Unix。

os.getgid()
現在のプロセスの実グループ id を返します。

利用可能な環境: Unix。

os.getgrouplist(*user*, *group*)
user が所属するグループ id のリストを返します。*group* がリストにない場合、それを追加します。通常、*group* にはユーザー *user* のパスワードレコードに書かれているグループ ID を指定します。

利用可能な環境: Unix。

バージョン 3.3 で追加.

os.getgroups()
現在のプロセスに関連付けられた従属グループ id のリストを返します。

利用可能な環境: Unix。

注釈: Mac OS X では *getgroups()* の挙動は他の Unix プラットフォームとはいくぶん異なります。Python のインタープリタが 10.5 以前の Deployment Target でビルドされている場合、*getgroups()* は現在のユーザープロセスに関連付けられている実効グループ id を返します。このリストはシステムで定義されたエントリ数 (通常は 16) に制限され、適切な特権があれば *setgroups()* の呼び出しによって変更することができます。10.5 より新しい Deployment Target でビルドされている場合、*getgroups()* はプロセスの実効ユーザー id に関連付けられたユーザーの現在のグループアクセスリストを返します。このグループアクセスリストは、プロセスのライフタイムで変更される可能性があり、*setgroups()* の呼び出しの影響を受けず、長さ 16 の制限を受けません。Deployment Target の値 MACOSX_DEPLOYMENT_TARGET は、*sysconfig.get_config_var()* で取得することができます。

---

os.getlogin()
プロセスの制御端末にログインしているユーザー名を返します。ほとんどの場合、*getpass.getuser()* を使う方が便利です。なぜなら、*getpass.getuser()* は、ユーザーを見つけるために、環境変数

LOGNAME や USERNAME を調べ、さらには pwd.getpwuid(os.getuid())[0] まで調べに行くからです。

Availability: Unix, Windows。

os.getpgid(*pid*)
プロセス id *pid* のプロセスのプロセスグループ id を返します。*pid* が 0 の場合、現在のプロセスのプロセスグループ id を返します。

利用可能な環境: Unix。

os.getpgrp()
現在のプロセスグループの id を返します。

利用可能な環境: Unix。

os.getpid()
現在のプロセス id を返します。

os.getppid()
親プロセスのプロセス id を返します。親プロセスが終了していた場合、Unix では init プロセスの id (1) が返され、Windows では親のプロセス id だったもの (別のプロセスで再利用されているかもしれない) がそのまま返されます。

Availability: Unix, Windows。

バージョン 3.2 で変更: Windows サポートが追加されました。

os.getpriority(*which*, *who*)
プログラムのスケジューリング優先度を取得します。*which* の値は *PRIO_PROCESS*、*PRIO_PGRP*、あるいは *PRIO_USER* のいずれか一つで、*who* の値は *which* に応じて解釈されます (*PRIO_PROCESS* であればプロセス識別子、*PRIO_PGRP* であればプロセスグループ識別子、そして *PRIO_USER* であればユーザー ID)。*who* の値がゼロの場合、呼び出したプロセス、呼び出したプロセスのプロセスグループ、および呼び出したプロセスの実ユーザー id を (それぞれ) 意味します。

利用可能な環境: Unix。

バージョン 3.3 で追加.

os.PRIO_PROCESS
os.PRIO_PGRP
os.PRIO_USER
getpriority() と *setpriority()* 用のパラメータです。

利用可能な環境: Unix。

バージョン 3.3 で追加.

os.getresuid()
現在のプロセスの実ユーザー id 、実効ユーザー id 、および保存ユーザー id を示す、(ruid, euid, suid) のタプルを返します。

利用可能な環境: Unix。

バージョン 3.2 で追加.

os.**getresgid**()
現在のプロセスの実グループ id 、実効グループ id 、および保存グループ id を示す、(rgid, egid, sgid) のタプルを返します。

利用可能な環境: Unix。

バージョン 3.2 で追加.

os.**getuid**()
現在のプロセスの実ユーザー id を返します。

利用可能な環境: Unix。

os.**initgroups**(*username*, *gid*)
システムの initgroups() を呼び出し、指定された *username* がメンバーである全グループと *gid* で指定されたグループでグループアクセスリストを初期化します。

利用可能な環境: Unix。

バージョン 3.2 で追加.

os.**putenv**(*key*, *value*)
key という名前の環境変数に文字列 *value* を設定します。このような環境変数の変更は、*os.system()*、*popen()*、または *fork()* と *execv()* で起動されたサブプロセスに影響を与えます。

利用できる環境: 主な Unix 互換環境、Windows。

注釈: FreeBSD と Mac OS X を含む一部のプラットフォームでは、environ の値を変更するとメモリリークの原因になる場合があります。システムの putenv に関するドキュメントを参照してください。

---

putenv() がサポートされている場合、os.environ のアイテムに対する代入を行うと、自動的に *putenv()* の対応する呼び出しに変換されます。直接 *putenv()* を呼び出した場合 os.environ は更新されないため、実際には os.environ のアイテムに代入する方が望ましい操作です。

引数 key, value を指定して **監査イベント** os.putenv を送出します。

os.**setegid**(*egid*)
現在のプロセスに実効グループ id をセットします。

利用可能な環境: Unix。

os.**seteuid**(*euid*)
現在のプロセスに実効ユーザー id をセットします。

利用可能な環境: Unix。

os.setgid(*gid*)
現在のプロセスにグループ id をセットします。

利用可能な環境: Unix。

os.setgroups(*groups*)
現在のグループに関連付けられた従属グループ id のリストを *groups* に設定します。*groups* はシーケンス型でなくてはならず、各要素はグループを特定する整数でなくてはなりません。通常、この操作はスーパユーザーしか利用できません。

利用可能な環境: Unix。

注釈: Mac OS X では、*groups* の長さはシステムで定義された実効グループ id の最大数 (通常は 16) を超えない場合があります。setgroups() 呼び出しで設定されたものと同じグループリストが返されないケースについては、*getgroups()* のドキュメントを参照してください。

---

os.setpgrp()
システムコール `setpgrp()` か `setpgrp(0, 0)` のどちらか (実装されているもの) を呼び出します。機能については UNIX マニュアルを参照して下さい。

利用可能な環境: Unix。

os.setpgid(*pid*, *pgrp*)
システムコール `setpgid()` を呼び出してプロセス id *pid* のプロセスのプロセスグループ id を *pgrp* に設定します。この動作に関しては Unix のマニュアルを参照してください。

利用可能な環境: Unix。

os.setpriority(*which*, *who*, *priority*)
プログラムのスケジューリング優先度を設定します。*which* は *PRIO_PROCESS*、*PRIO_PGRP*、あるいは *PRIO_USER* のいずれか一つで、*who* の値は *which* に応じて解釈されます (*PRIO_PROCESS* であればプロセス識別子、*PRIO_PGRP* であればプロセスグループ識別子、そして *PRIO_USER* であればユーザー ID)。*who* の値がゼロの場合、呼び出したプロセス、呼び出したプロセスのプロセスグループ、および呼び出したプロセスの実ユーザー id を (それぞれ) 意味します。*priority* は -20 から 19 の整数値で、デフォルトの優先度は 0 です。小さい数値ほど優先されるスケジューリングとなります。

利用可能な環境: Unix。

バージョン 3.3 で追加.

os.setregid(*rgid*, *egid*)
現在のプロセスの実グループ id および実効グループ id を設定します。

利用可能な環境: Unix。

os.setresgid(*rgid*, *egid*, *sgid*)
現在のプロセスの、実グループ id 、実効グループ id 、および保存グループ id を設定します。

利用可能な環境: Unix。

バージョン 3.2 で追加.

os.**setresuid**(*ruid*, *euid*, *suid*)
現在のプロセスの実ユーザー id 、実効ユーザー id 、および保存ユーザー id を設定します。

利用可能な環境: Unix。

バージョン 3.2 で追加.

os.**setreuid**(*ruid*, *euid*)
現在のプロセスの実ユーザー id および実効ユーザー id を設定します。

利用可能な環境: Unix。

os.**getsid**(*pid*)
getsid() システムコールを呼び出します。機能については Unix のマニュアルを参照してください。

利用可能な環境: Unix。

os.**setsid**()
setsid() システムコールを呼び出します。機能については Unix のマニュアルを参照してください。

利用可能な環境: Unix。

os.**setuid**(*uid*)
現在のプロセスのユーザー id を設定します。

利用可能な環境: Unix。

os.**strerror**(*code*)
エラーコード *code* に対応するエラーメッセージを返します。未知のエラーコードの対して strerror() が NULL を返すプラットフォームでは、*ValueError* が送出されます。

os.**supports_bytes_environ**
環境のネイティブ OS タイプがバイト型の場合、True です (例: Windows では、False です)。

バージョン 3.2 で追加.

os.**umask**(*mask*)
現在の数値 umask を設定し、以前の umask 値を返します。

os.**uname**()
現在のオペレーティングシステムを識別する情報を返します。返り値は 5 個の属性を持つオブジェクトです:

- sysname - OS の名前
- nodename - (実装時に定義された) ネットワーク上でのマシン名
- release - OS のリリース
- version - OS のバージョン
- machine - ハードウェア識別子

後方互換性のため、このオブジェクトはイテラブルでもあり、sysname、nodename、release、version、および machine の 5 個の要素をこの順序で持つタプルのように振る舞います。

一部のシステムでは、nodename はコンポーネントを読み込むために 8 文字または先頭の要素だけに切り詰められます; ホスト名を取得する方法としては、*socket.gethostname()* を使う方がよいでしょう。あるいは socket.gethostbyaddr(socket.gethostname()) でもかまいません。

利用できる環境: 最近の Unix 互換環境。

バージョン 3.3 で変更: 返り値の型が、タプルから属性名のついたタプルライクオブジェクトに変更されました。

os.unsetenv(*key*)

key という名前の環境変数を unset (削除) します。このような環境変数の変更は、*os.system()*、*popen()*、または *fork()* と *execv()* で起動されたサブプロセスに影響を与えます。

unsetenv() がサポートされている場合、os.environ のアイテムの削除を行うと、自動的に *unsetenv()* の対応する呼び出しに変換されます。直接 *unsetenv()* を呼び出した場合 os.environ は更新されないため、実際には os.environ のアイテムを削除する方が望ましい操作です。

引数 key を指定して 監査イベント os.unsetenv を送出します。

利用できる環境: 主な Unix 互換環境。

16.1.3 ファイルオブジェクトの生成

以下の関数は新しい ファイルオブジェクト を作成します。(ファイル記述子のオープンについては *open()* も参照してください)

os.fdopen(*fd*, **args*, ***kwargs*)

ファイル記述子 *fd* に接続し、オープンしたファイルオブジェクトを返します。これは組み込み関数 *open()* の別名であり、同じ引数を受け取ります。唯一の違いは *fdopen()* の第一引数が常に整数でなければならないことです。

16.1.4 ファイル記述子の操作

これらの関数は、ファイル記述子を使って参照されている I/O ストリームを操作します。

ファイル記述子とは現在のプロセスで開かれたファイルに対応する小さな整数です。例えば、標準入力のファイル記述子は通常 0 で、標準出力は 1 、標準エラーは 2 です。プロセスから開かれたその他のファイルには 3 、4 、5 と割り振られていきます。「ファイル記述子」という名称は少し誤解を与えるものかもしれません。Unix プラットフォームでは、ソケットやパイプもファイル記述子によって参照されます。

fileno() メソッドを使用して、必要な場合に *file object* に関連付けられているファイル記述子を取得することができます。ファイル記述子を直接使用すると、ファイルオブジェクトのメソッドが使用されないため、データの内部バッファなどの性質は無視されることに注意してください。

os.close(*fd*)

ファイル記述子 *fd* をクローズします。

注釈: この関数は低水準の I/O 向けのもので、*os.open()* や *pipe()* が返すファイル記述子に対して使用しなければなりません。組み込み関数 *open()* や *popen()* 、*fdopen()* が返す ”ファイルオブジェクト” を閉じるには、オブジェクトの *close()* メソッドを使用してください。

---

os.closerange(*fd_low*, *fd_high*)

fd_low 以上 *fd_high* 未満のすべてのファイル記述子をエラーを無視してクローズします。以下のコードと等価です:

```
for fd in range(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

os.copy_file_range(*src*, *dst*, *count*, *offset_src=None*, *offset_dst=None*)

ファイル記述子 *src* の *offset_src* から *count* バイトを、ファイル記述子 *dst* の *offset_dst* にコピーします。もし *offset_src* が None の場合は *src* は現在の位置から読まれます。*offset_dst* についても同様です。*src* および *dst* のファイルは同じファイルシステム上になければなりません。違う場合には *errno* を *errno.EXDEV* として *OSError* が送出されます。

このコピーは、カーネルからユーザースペースにデータを転送した後カーネルに戻すという追加のコスト無しに完了します。加えて、追加の最適化ができるファイルシステムもあります。このコピーはファイルが両方ともバイナリファイルとして開かれたかのように行われます。

返り値はコピーされたバイトの量です。この値は、要求した量より少なくなることもあります。

Availability: Linux kernel >= 4.5 または glibc >= 2.27。

バージョン 3.8 で追加.

os.device_encoding(*fd*)

fd に関連付けられたデバイスが端末 (ターミナル) に接続されている場合に、そのデバイスのエンコーディングを表す文字列を返します。端末に接続されていない場合、*None* を返します。

os.dup(*fd*)

ファイル記述子 *fd* の複製を返します。新しいファイル記述子は **継承不可** です。

Windows では、標準ストリーム (0: 標準入力、1: 標準出力、2: 標準エラー出力) を複製する場合、新しいファイル記述子は **継承可能** です。

バージョン 3.4 で変更: 新しいファイル記述子が継承不可になりました。

os.dup2(*fd*, *fd2*, *inheritable=True*)

ファイル記述子 *fd* を *fd2* に複製し、必要な場合には後者を先に閉じます。*fd2* が返ります。新しいファイル記述子はデフォルトでは **継承可能** で、*inheritable* が `False` の場合は継承不可です。

バージョン 3.4 で変更: オプションの *inheritable* 引数が追加されました。

バージョン 3.7 で変更: 成功したときは *fd2* が返ります。以前は常に None が返っていました。

os.fchmod(*fd*, *mode*)

fd で指定されたファイルのモードを *mode* に変更します。*mode* に指定できる値については、*chmod()* のドキュメントを参照してください。Python 3.3 以降では os.chmod(fd, mode) と等価です。

引数 path, mode, dir_fd を指定して **監査イベント** os.chmod を送出します。

利用可能な環境: Unix。

os.fchown(*fd*, *uid*, *gid*)

fd で指定されたファイルの所有者 id およびグループ id を数値 *uid* および *gid* に変更します。いずれかの id を変更せずにおくにはその値として -1 を指定します。*chown()* を参照してください。Python 3.3 以降では os.chown(fd, uid, gid) と等価です。

引数 path, uid, gid, dir_fd を指定して **監査イベント** os.chown を送出します。

利用可能な環境: Unix。

os.fdatasync(*fd*)

ファイル記述子 *fd* を持つファイルのディスクへの書き込みを強制します。メタデータの更新は強制しません。

利用可能な環境: Unix。

注釈: この関数は MacOS では利用できません。

---

os.fpathconf(*fd*, *name*)

開いているファイルに関連するシステム設定情報を返します。*name* は取得する設定名を指定します。これは、いくつかの標準 (POSIX.1 、Unix 95 、Unix 98 その他) で定義された定義済みのシステム値名の文字列である場合があります。プラットフォームによっては別の名前も定義されています。ホストオペレーティングシステムの関知する名前は pathconf_names 辞書で与えられています。このマップ型オブジェクトに含まれていない構成変数については、*name* に整数を渡してもかまいません。

name が不明の文字列である場合、*ValueError* を送出します。*name* の特定の値がホストシステムでサポートされていない場合、pathconf_names に含まれていたとしても、*errno.EINVAL* をエラー番号として *OSError* を送出します。

Python 3.3 以降では os.pathconf(fd, name) と等価です。

利用可能な環境: Unix。

os.fstat(*fd*)

ファイル記述子 *fd* の状態を取得します。*stat_result* オブジェクトを返します。

Python 3.3 以降では os.stat(fd) と等価です。

参考:

stat() 関数。

os.fstatvfs(*fd*)

statvfs() と同様に、ファイル記述子 *fd* に関連付けられたファイルが格納されているファイルシステムに関する情報を返します。Python 3.3 以降では `os.statvfs(fd)` と等価です。

利用可能な環境: Unix。

os.fsync(*fd*)

ファイル記述子 *fd* を持つファイルのディスクへの書き込みを強制します。Unix では、ネイティブの `fsync()` 関数を、Windows では `_commit()` 関数を呼び出します。

Python の ファイルオブジェクト *f* を使う場合、*f* の内部バッファを確実にディスクに書き込むために、まず `f.flush()` を、その後 `os.fsync(f.fileno())` を実行してください。

Availability: Unix, Windows。

os.ftruncate(*fd*, *length*)

ファイル記述子 *fd* に対応するファイルを、サイズが最長で *length* バイトになるように切り詰めます。Python 3.3 以降では `os.truncate(fd, length)` と等価です。

引数 `fd`, `length` を指定して 監査イベント `os.truncate` を送出します。

Availability: Unix, Windows。

バージョン 3.5 で変更: Windows サポートを追加しました。

os.get_blocking(*fd*)

記述子のブロッキングモードを取得します。*O_NONBLOCK* フラグが設定されている場合は `False` で、フラグがクリアされている場合は `True` です。

set_blocking() および *socket.socket.setblocking()* も参照してください。

利用可能な環境: Unix。

バージョン 3.5 で追加.

os.isatty(*fd*)

ファイル記述子 *fd* がオープンされていて、tty (のような) デバイスに接続されている場合、`True` を返します。そうでない場合は `False` を返します。

os.lockf(*fd*, *cmd*, *len*)

オープンされたファイル記述子に対して、POSIX ロックの適用、テスト、解除を行います。*fd* はオープンされたファイル記述子です。*cmd* には使用するコマンド (*F_LOCK*、*F_TLOCK*、*F_ULOCK*、あるいは *F_TEST* のいずれか一つ) を指定します。*len* にはロックするファイルのセクションを指定します。

引数 `fd`, `cmd`, `len` を指定して 監査イベント `os.lockf` を送出します。

利用可能な環境: Unix。

バージョン 3.3 で追加.

os.F_LOCK

os.F_TLOCK
os.F_ULOCK
os.F_TEST
lockf() がとる動作を指定するフラグです。

利用可能な環境: Unix。

バージョン 3.3 で追加.

os.lseek(*fd*, *pos*, *how*)
ファイル記述子 *fd* の現在の位置を *pos* に設定します。*pos* の意味は *how* で次のように修飾されます。ファイルの先頭からの相対位置には *SEEK_SET* か 0 を、現在の位置からの相対位置には *SEEK_CUR* か 1 を、ファイルの末尾からの相対位置には *SEEK_END* か 2 を設定します。戻り値は、新しいカーソル位置のファイルの先頭からのバイト数です。

os.SEEK_SET
os.SEEK_CUR
os.SEEK_END
lseek() 関数に渡すパラメーター。値は順に 0, 1, 2 です。

バージョン 3.3 で追加: 一部のオペレーティングシステムは os.SEEK_HOLE や os.SEEK_DATA など、追加の値をサポートすることがあります。

os.open(*path*, *flags*, *mode=0o777*, ***, *dir_fd=None*)
ファイル *path* を開き、*flag* に従って様々なフラグを設定し、可能なら *mode* に従ってファイルモードを設定します。*mode* を計算する際、まず現在の umask 値でマスクされます。新たに開いたファイルのファイル記述子を返します。新しいファイル記述子は 継承不可 です。

フラグとファイルモードの値についての詳細は C ランタイムのドキュメントを参照してください; (*O_RDONLY* や *O_WRONLY* のような) フラグ定数は *os* モジュールでも定義されています。特に、Windows ではバイナリモードでファイルを開く時に *O_BINARY* を加える必要があります。

この関数は *dir_fd* パラメタで ディレクトリ記述子への相対パス をサポートしています。

引数 path, mode, flags を指定して 監査イベント open を送出します。

バージョン 3.4 で変更: 新しいファイル記述子が継承不可になりました。

注釈: この関数は低水準の I/O 向けのものです。通常の利用では、組み込み関数 *open()* を使用してください。*open()* は read() や write() (そしてさらに多くの) メソッドを持つ ファイルオブジェクト を返します。ファイル記述子をファイルオブジェクトでラップするには *fdopen()* を使用してください。

バージョン 3.3 で追加: 引数 *dir_fd* が追加されました。

バージョン 3.5 で変更: システムコールが中断されシグナルハンドラが例外を送出しなかった場合、この関数は *InterruptedError* 例外を送出する代わりにシステムコールを再試行するようになりました (論拠については **PEP 475** を参照してください)。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

以下の定数は *open()* 関数の *flags* 引数に利用します。これらの定数は、ビット単位に OR 演算子 | で組み合わせることができます。一部、すべてのプラットフォームでは使用できない定数があります。利用可能かどうかや使い方については、Unix では *open(2)*、Windows では MSDN を参照してください。

os.O_RDONLY
os.O_WRONLY
os.O_RDWR
os.O_APPEND
os.O_CREAT
os.O_EXCL
os.O_TRUNC

上記の定数は Unix および Windows で利用可能です。

os.O_DSYNC
os.O_RSYNC
os.O_SYNC
os.O_NDELAY
os.O_NONBLOCK
os.O_NOCTTY
os.O_CLOEXEC

上記の定数は Unix でのみ利用可能です。

バージョン 3.3 で変更: 定数 *O_CLOEXEC* が追加されました。

os.O_BINARY
os.O_NOINHERIT
os.O_SHORT_LIVED
os.O_TEMPORARY
os.O_RANDOM
os.O_SEQUENTIAL
os.O_TEXT

上記の定数は Windows でのみ利用可能です。

os.O_ASYNC
os.O_DIRECT
os.O_DIRECTORY
os.O_NOFOLLOW
os.O_NOATIME
os.O_PATH
os.O_TMPFILE
os.O_SHLOCK
os.O_EXLOCK

上記の定数は拡張仕様であり、C ライブラリで定義されていない場合は利用できません。

バージョン 3.4 で変更: *O_PATH* を、それをサポートするシステムで追加しました。また、*O_TMPFILE*

を追加しました (Linux Kernel 3.11 以降でのみ利用可能です)。

os.openpty()
新しい擬似端末のペアを開きます。pty および tty を表すファイル記述子のペア (master, slave) を返します。新しいファイル記述子は 継承不可 です。(若干) 可搬性の高いアプローチには *pty* を使用してください。

利用できる環境: 一部の Unix 互換環境。

バージョン 3.4 で変更: 新しいファイル記述子が継承不可になりました。

os.pipe()
パイプを作成します。読み込み、書き込みに使うことの出来るファイル記述子のペア (r, w) を返します。新しいファイル記述子は 継承不可 です。

Availability: Unix, Windows。

バージョン 3.4 で変更: 新しいファイル記述子が継承不可になりました。

os.pipe2(*flags*)
flags を設定したパイプをアトミックに作成します。*flags* には値 *O_NONBLOCK* と *O_CLOEXEC* を一つ以上論理和指定できます。読み込み、書き込みに使うことの出来るファイル記述子のペア (r, w) を返します。

利用できる環境: 一部の Unix 互換環境。

バージョン 3.3 で追加.

os.posix_fallocate(*fd*, *offset*, *len*)
fd で指定されたファイルに対し、開始位置 *offset* から *len* バイト分割り当てるに十分なディスクスペースを確保します。

利用可能な環境: Unix。

バージョン 3.3 で追加.

os.posix_fadvise(*fd*, *offset*, *len*, *advice*)
データへアクセスする意思を、パターンを指定して宣言します。これによりカーネルが最適化を行えるようになります。*advice* は *fd* で指定されたファイルに対し、開始位置 *offset* から *len* バイト分の領域に適用されます。*advice* には *POSIX_FADV_NORMAL*、*POSIX_FADV_SEQUENTIAL*、*POSIX_FADV_RANDOM*、*POSIX_FADV_NOREUSE*、*POSIX_FADV_WILLNEED*、または *POSIX_FADV_DONTNEED* のいずれか一つを指定します。

利用可能な環境: Unix。

バージョン 3.3 で追加.

os.POSIX_FADV_NORMAL
os.POSIX_FADV_SEQUENTIAL
os.POSIX_FADV_RANDOM
os.POSIX_FADV_NOREUSE

os.POSIX_FADV_WILLNEED

os.POSIX_FADV_DONTNEED

posix_fadvise() において、使われるであろうアクセスパターンを指定する *advice* に使用できるフラグです。

利用可能な環境: Unix。

バージョン 3.3 で追加.

os.pread(*fd*, *n*, *offset*)

ファイル記述子の位置 *offset* から最大で *n* バイトを読み出します。ファイルオフセットは変化しません。

読み込んだバイト分のバイト列を返します。*fd* が参照しているファイルの終端に達した場合、空のバイト列が返されます。

利用可能な環境: Unix。

バージョン 3.3 で追加.

os.preadv(*fd*, *buffers*, *offset*, *flags=0*)

ファイル記述子 *fd* の *offset* の位置から、可変な *bytes-like* **オブジェクト** *buffers* にオフセットを変更せずに読み込みます。データをそれぞれのバッファがいっぱいになるまで移し、いっぱいになったらシーケンスの次のバッファに処理を移し、残りのデータを読み込ませます。

flags 引数にはゼロあるいは次のフラグのバイトごとの OR を取った結果が保持されています。

- *RWF_HIPRI*
- *RWF_NOWAIT*

実際に読み込んだ合計バイト数を返します。この値は、すべてのオブジェクトの容量の総量よりも小さくなることがあります。

オペレーティングシステムは、使用可能なバッファの個数に基づいて上限 (func:*sysconf* の 'SC_IOV_MAX' の値) を設定することがあります。

os.readv() と *os.pread()* の機能を統合します。

Availability: Linux 2.6.30 and newer, FreeBSD 6.0 and newer, OpenBSD 2.7 and newer. Using flags requires Linux 4.6 or newer.

バージョン 3.7 で追加.

os.RWF_NOWAIT

即座に利用できないデータを待ちません。このフラグを指定すると、バックエンドのストレージからデータを読む必要があるか、ロックを待機する場合、システムコールは即座にリターンします。

読み込みに成功したデータがある場合、読み込んだバイト数を返します。読み込めるバイト列がない場合、-1 を返し、errno に *errno.EAGAIN* を設定します。

利用可能な環境: Linux 4.14 以上。

バージョン 3.7 で追加.

os.RWF_HIPRI
: 優先度の高い読み込み・書き込み (read/write) フラグです。ブロックストレージに対して、追加のリソースを必要とする一方で低レイテンシなデバイスのポーリングを使うことを許可します。

  現状、Linux では、ファイル記述子を *O_DIRECT* フラグを指定したオープンした場合でのみ、この機能を利用できます。

  利用可能な環境: Linux 4.6 以上。

  バージョン 3.7 で追加.

os.pwrite(*fd*, *str*, *offset*)
: *str* 中のバイト文字列をファイル記述子 *fd* の *offset* の位置に書き込みます。ファイルオフセットを変化しません。

  実際に書き込まれたバイト数を返します。

  利用可能な環境: Unix。

  バージョン 3.3 で追加.

os.pwritev(*fd*, *buffers*, *offset*, *flags=0*)
: *buffers* の内容をファイル記述子 *fd* のオフセット位置 *offset* に書き込みます。ファイルのオフセット位置は変更しません。*buffers* は *bytes-like* **オブジェクト** のシーケンスでなければなりません。バッファは配列の順番で処理されます。すなわち、最初のバッファの内容は、次のバッファの処理に移る前に全て書き込まれ、以降も同様に処理されます。

  flags 引数にはゼロあるいは次のフラグのバイトごとの OR を取った結果が保持されています。

  - *RWF_DSYNC*
  - *RWF_SYNC*

  実際に書き込まれた合計バイト数を返します。

  オペレーティングシステムは、使用可能なバッファの個数に基づいて上限 (func:*sysconf* の `'SC_IOV_MAX'` の値) を設定することがあります。

  os.writev() と *os.pwrite()* の機能を統合します。

  Availability: Linux 2.6.30 and newer, FreeBSD 6.0 and newer, OpenBSD 2.7 and newer. Using flags requires Linux 4.7 or newer.

  バージョン 3.7 で追加.

os.RWF_DSYNC
: Provide a per-write equivalent of the *O_DSYNC* `open(2)` flag. This flag effect applies only to the data range written by the system call.

  利用可能な環境: Linux 4.7 以上。

バージョン 3.7 で追加.

os.RWF_SYNC
: Provide a per-write equivalent of the *O_SYNC* `open(2)` flag. This flag effect applies only to the data range written by the system call.

  利用可能な環境: Linux 4.7 以上。

  バージョン 3.7 で追加.

os.read(*fd*, *n*)
: ファイル記述子 *fd* から 最大 *n* バイトを読み込みます。

  読み込んだバイト分のバイト列を返します。*fd* が参照しているファイルの終端に達した場合、空のバイト列が返されます。

  注釈: この関数は低水準の I/O 向けのもので、*os.open()* や *pipe()* が返すファイル記述子に対して使用されなければなりません。組み込み関数 *open()* や *popen()* 、*fdopen()* 、あるいは *sys.stdin* が返す ”ファイルオブジェクト” を読み込むには、オブジェクトの `read()` か `readline()` メソッドを使用してください。

  バージョン 3.5 で変更: システムコールが中断されシグナルハンドラが例外を送出しなかった場合、この関数は *InterruptedError* 例外を送出する代わりにシステムコールを再試行するようになりました (論拠については **PEP 475** を参照してください)。

os.sendfile(*out*, *in*, *offset*, *count*)
os.sendfile(*out*, *in*, *offset*, *count*[, *headers*][, *trailers*], *flags=0*)
: ファイル記述子 *in* からファイル記述子 *out* への開始位置 *offset* へ *count* バイトコピーします。送信バイト数を返します。EOF に達した場合は 0 を返します。

  前者の関数表記は *sendfile()* が定義されているすべてのプラットフォームでサポートされています。

  Linux では、*offset* に `None` が与えられると、バイト列は *in* の現在の位置から読み込まれ、*in* の位置は更新されます。

  後者は Mac OS X および FreeBSD で使用される場合があります。*headers* および *trailers* は任意のバッファのシーケンス型オブジェクトで、*in* からのデータが書き出される前と後に書き出されます。返り値は前者と同じです。

  Mac OS X と FreeBSD では、*count* の値に 0 を指定すると、*in* の末尾に達するまで送信します。

  全てのプラットフォームはソケットをファイル記述子 *out* としてサポートし、あるプラットフォームは他の種類 (例えば、通常のファイル、パイプ) も同様にサポートします。

  クロスプラットフォームのアプリケーションは *headers*、*trailers* ならびに *flags* 引数を使用するべきではありません。

  利用可能な環境: Unix。

注釈: *sendfile()* のより高水準のラッパについては *socket.socket.sendfile()* を参照してください。

バージョン 3.3 で追加.

os.set_blocking(*fd*, *blocking*)

指定されたファイル記述子のブロッキングモードを設定します。ブロッキングが False の場合 *O_NONBLOCK* フラグを設定し、そうでない場合はクリアします。

get_blocking() および *socket.socket.setblocking()* も参照してください。

利用可能な環境: Unix。

バージョン 3.5 で追加.

os.SF_NODISKIO

os.SF_MNOWAIT

os.SF_SYNC

実装がサポートしている場合 *sendfile()* 関数に渡すパラメーターです。

利用可能な環境: Unix。

バージョン 3.3 で追加.

os.readv(*fd*, *buffers*)

Read from a file descriptor *fd* into a number of mutable *bytes-like objects* *buffers*. Transfer data into each buffer until it is full and then move on to the next buffer in the sequence to hold the rest of the data.

実際に読み込んだ合計バイト数を返します。この値は、すべてのオブジェクトの容量の総量よりも小さくなることがあります。

オペレーティングシステムは、使用可能なバッファの個数に基づいて上限 (func:*sysconf* の 'SC_IOV_MAX' の値) を設定することがあります。

利用可能な環境: Unix。

バージョン 3.3 で追加.

os.tcgetpgrp(*fd*)

fd (*os.open()* が返すオープンしたファイル記述子) で与えられる端末に関連付けられたプロセスグループを返します。

利用可能な環境: Unix。

os.tcsetpgrp(*fd*, *pg*)

fd (*os.open()* が返すオープンしたファイル記述子) で与えられる端末に関連付けられたプロセスグループを *pg* に設定します。

利用可能な環境: Unix。

os.**ttyname**(*fd*)

ファイル記述子 *fd* に関連付けられている端末デバイスを特定する文字列を返します。*fd* が端末に関連付けられていない場合、例外が送出されます。

利用可能な環境: Unix。

os.**write**(*fd*, *str*)

str のバイト列をファイル記述子 *fd* に書き出します。

実際に書き込まれたバイト数を返します。

注釈: この関数は低水準の I/O 向けのもので、*os.open()* や *pipe()* が返すファイル記述子に対して使用しなければなりません。組み込み関数 *open()* や *popen()* 、*fdopen()* 、あるいは *sys.stdout* や *sys.stderr* が返す "ファイルオブジェクト" に書き込むには、オブジェクトの `write()` メソッドを使用してください。

バージョン 3.5 で変更: システムコールが中断されシグナルハンドラが例外を送出しなかった場合、この関数は *InterruptedError* 例外を送出する代わりにシステムコールを再試行するようになりました (論拠については **PEP 475** を参照してください)。

os.**writev**(*fd*, *buffers*)

buffers の内容をファイル記述子 *fd* へ書き出します。*buffers* は *bytes-like* **オブジェクト** のシーケンスでなければなりません。バッファは配列の順番で処理されます。最初のバッファの内容全体は 2 番目のバッファに進む前に書き込まれ、その次も同様です。

実際に書き込まれた合計バイト数を返します。

オペレーティングシステムは、使用可能なバッファの個数に基づいて上限 (func:*sysconf* の `'SC_IOV_MAX'` の値) を設定することがあります。

利用可能な環境: Unix。

バージョン 3.3 で追加.

ターミナルのサイズの問い合わせ

バージョン 3.3 で追加.

os.**get_terminal_size**(*fd=STDOUT_FILENO*)

ターミナル (端末) のサイズ (`columns`, `lines`) を、*terminal_size* 型のタプルで返します。

オプションの引数 `fd` には問い合わせるファイル記述子を指定します (デフォルトは `STDOUT_FILENO`、または標準出力)。

ファイル記述子が接続されていなかった場合、*OSError* が送出されます。

通常は高水準関数である *shutil.get_terminal_size()* を使用してください。`os.get_terminal_size` は低水準の実装です。

Availability: Unix, Windows。

class os.terminal_size
ターミナルウィンドウのサイズ (`columns, lines`) を保持するタプルのサブクラスです。

columns
ターミナルウィンドウの横幅 (文字数) です。

lines
ターミナルウィンドウの高さ (文字数) です。

ファイル記述子の継承

バージョン 3.4 で追加.

ファイル記述子には「継承可能 (inheritable)」フラグというものがあって、これにより子プロセスにファイル記述子が引き継がれるかどうかが決定されます。Python 3.4 より、Python によって作成されるファイル記述子はデフォルトで継承不可 (non-inheritable) となりました。

UNIX の場合、継承不可のファイル記述子は新規プロセス実行時にクローズされ、そうでないファイル記述子は引き継がれます。

Windows の場合は、標準ストリームを除き、継承不可のハンドルと継承不可のファイル記述子は子プロセスでクローズされます。標準ストリーム (ファイル記述子の 0, 1, 2: 標準入力, 標準出力, 標準エラー出力) は常に引き継がれます。*spawn** 関数を使う場合、全ての継承可能なハンドルと全ての継承可能なファイル記述子は引き継がれます。*subprocess* モジュールを使う場合、標準ストリームを除く全てのファイル記述子はクローズされ、継承可能なハンドルは *close_fds* 引数が `False` の場合にのみ引き継がれます。

os.get_inheritable(*fd*)
指定したファイル記述子の「継承可能 (inheritable)」フラグを取得します (boolean)。

os.set_inheritable(*fd*, *inheritable*)
指定したファイル記述子の「継承可能 (inheritable)」フラグをセットします。

os.get_handle_inheritable(*handle*)
指定したハンドルの「継承可能 (inheritable)」フラグを取得します (boolean)。

利用可能な環境: Windows 。

os.set_handle_inheritable(*handle*, *inheritable*)
指定したハンドルの「継承可能 (inheritable)」フラグをセットします。

利用可能な環境: Windows 。

16.1.5 ファイルとディレクトリ

一部の Unix プラットフォームでは、このセクションの関数の多くが以下の機能を一つ以上サポートしています。

- **ファイル記述子の指定:** *os* モジュールの関数で *path* 引数に渡される値はファイルパスでなければなりません。しかしながら、いくつかの関数では *path* 引数にファイルパスではなく、そのファイルをオープンしたファイル記述子を指定できるようになりました。この場合それらの関数はファイル記述子が参照するファイルに対して操作を行います。(POSIX システムの場合、Python はプレフィックス `f` の付いた関数の亜種 (たとえば `chdir``の代わりに ``fchdir`) を呼び出します。)

  os.supports_fd を使うことで、そのプラットフォーム上で *path* にファイル記述子を指定できるかどうかを確認することができます。この機能が利用可能でない場合、*os.supports_fd* の利用は *NotImplementedError* 例外を送出します。

  その関数が引数に *dir_fd* または *follow_symlinks* もサポートしている場合、*path* にファイル記述子を指定した時にそれらのいずれかを指定するとエラーになります。

- **ディレクトリ記述子からの相対パス:** *dir_fd* が `None` でない場合、その値はディレクトリを参照するファイル記述子である必要があり、また操作対象のファイルパスは相対パスである必要があります; このときパスはファイル記述子が指すディレクトリからの相対パスと解釈されます。パスが絶対パスの場合、*dir_fd* は無視されます。(POSIX システムでは、Python はサフィックス `at` が付いた関数の亜種、もしくはさらにプレフィックス `f` が付いたもの (たとえば `access` の代わりに `faccessat`) を呼び出します。

  そのプラットフォーム上で特別な関数に *dir_fd* がサポートされているかどうかは、*os.supports_dir_fd* で確認できます。利用できない場合 *NotImplementedError* が送出されます。

- **シンボリックリンクをたどらない:** *follow_symlinks* が `False` で、かつパスの末尾の要素がシンボリックリンクの場合、関数はシンボリックリンクが指すファイルではなくシンボリックリンク自身を操作対象とします。(POSIX システムの場合、Python はプレフィックス `l...` 付きの関数の亜種を呼び出します。)

  そのプラットフォーム上で特別な関数に *follow_symlinks* がサポートされているかどうかは、*os.supports_follow_symlinks* で確認できます。利用できない場合 *NotImplementedError* が送出されます。

`os.access`(*path*, *mode*, ***, *dir_fd=None*, *effective_ids=False*, *follow_symlinks=True*)

実 uid/gid を使って *path* に対するアクセスが可能か調べます。ほとんどのオペレーティングシステムは実効 uid/gid を使うため、このルーチンは suid/sgid 環境において、プログラムを起動したユーザーが *path* に対するアクセス権をもっているかを調べるために使われます。*path* が存在するかどうかを調べるには *mode* を *F_OK* にします。ファイルアクセス権限 (パーミッション) を調べるには、*R_OK*, *W_OK*, *X_OK* から一つまたはそれ以上のフラグを論理和指定でとることもできます。アクセスが許可されている場合 *True* を、そうでない場合 *False* を返します。詳細は *access(2)* の Unix マニュアルページを参照してください。

この関数は **ディレクトリ記述子への相対パス** および **シンボリックリンクをたどらない** をサポートしています。

effective_ids が True の場合、*access()* は実 uid/gid ではなく実効 uid/gid を使用してアクセス権を調べます。プラットフォームによっては *effective_ids* がサポートされていない場合があります; サポートされているかどうかは *os.supports_effective_ids* で確認できます。利用できない場合 *NotImplementedError* が送出されます。

注釈: ユーザーが、例えばファイルを開く権限を持っているかどうかを調べるために実際に *open()* を行う前に *access()* を使用することはセキュリティホールの原因になります。なぜなら、調べた時点とオープンした時点との時間差を利用してそのユーザーがファイルを不当に操作してしまうかもしれないからです。その場合は *EAFP* テクニックを利用するのが望ましいやり方です。例えば

```
if os.access("myfile", os.R_OK):
    with open("myfile") as fp:
        return fp.read()
return "some default data"
```

このコードは次のように書いたほうが良いです

```
try:
    fp = open("myfile")
except PermissionError:
    return "some default data"
else:
    with fp:
        return fp.read()
```

注釈: I/O 操作は *access()* が成功を示した時でも失敗することがあります。特にネットワークファイルシステムが通常の POSIX のパーミッションビットモデルをはみ出すアクセス権限操作を備える場合にはそのようなことが起こりえます。

バージョン 3.3 で変更: 引数 *dir_fd*、*effective_ids*、および *follow_symlinks* が追加されました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.F_OK

os.R_OK

os.W_OK

os.X_OK

access() で *path* をテストする時に *mode* 引数に渡す値です。上からそれぞれ、ファイルの存在、読み込み許可、書き込み許可、および実行許可になります。

os.chdir(*path*)

現在の作業ディレクトリを *path* に設定します。

この関数は **ファイル記述子の指定** をサポートしています。記述子は、オープンしているファイルではなく、オープンしているディレクトリを参照していなければなりません。

この関数は *OSError* やそのサブクラスである *FileNotFoundError*, *PermissionError*, *NotADirectoryError* などの例外を送出することがあります。

引数 path を指定して **監査イベント** os.chdir を送出します。

バージョン 3.3 で追加: 一部のプラットフォームで、*path* にファイル記述子の指定をサポートしました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.chflags(*path*, *flags*, ***, *follow_symlinks=True*)

path のフラグを *flags* に変更します。*flags* は、以下の値 (*stat* モジュールで定義されているもの) をビット単位の論理和で組み合わせることができます:

- *stat.UF_NODUMP*
- *stat.UF_IMMUTABLE*
- *stat.UF_APPEND*
- *stat.UF_OPAQUE*
- *stat.UF_NOUNLINK*
- *stat.UF_COMPRESSED*
- *stat.UF_HIDDEN*
- *stat.SF_ARCHIVED*
- *stat.SF_IMMUTABLE*
- *stat.SF_APPEND*
- *stat.SF_NOUNLINK*
- *stat.SF_SNAPSHOT*

この関数は **シンボリックリンクをたどらない** をサポートしています。

引数 path, flags を指定して **監査イベント** os.chflags を送出します。

利用可能な環境: Unix。

バージョン 3.3 で追加: 引数 *follow_symlinks* を追加しました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.chmod(*path*, *mode*, ***, *dir_fd=None*, *follow_symlinks=True*)

path のモードを数値 *mode* に変更します。*mode* は、(*stat* モジュールで定義されている) 以下の値のいずれかまたはビット単位の論理和で組み合わせた値を取り得ます:

- *stat.S_ISUID*
- *stat.S_ISGID*

- *stat.S_ENFMT*
- *stat.S_ISVTX*
- *stat.S_IREAD*
- *stat.S_IWRITE*
- *stat.S_IEXEC*
- *stat.S_IRWXU*
- *stat.S_IRUSR*
- *stat.S_IWUSR*
- *stat.S_IXUSR*
- *stat.S_IRWXG*
- *stat.S_IRGRP*
- *stat.S_IWGRP*
- *stat.S_IXGRP*
- *stat.S_IRWXO*
- *stat.S_IROTH*
- *stat.S_IWOTH*
- *stat.S_IXOTH*

この関数は **ファイル記述子の指定** 、**ディレクトリ記述子への相対パス** 、および **シンボリックリンクをたどらない** をサポートしています。

注釈: Windows は *chmod()* をサポートしていますが、ファイルの読み出し専用フラグを (`stat.S_IWRITE` および `stat.S_IREAD` 定数または対応する整数値によって) 設定できるだけです。その他のビットはすべて無視されます。

引数 `path`, `mode`, `dir_fd` を指定して **監査イベント** `os.chmod` を送出します。

バージョン 3.3 で追加: *path* にオープンしているファイル記述子の指定のサポート、および引数 *dir_fd* と *follow_symlinks* を追加しました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

`os.chown`(*path*, *uid*, *gid*, ***, *dir_fd=None*, *follow_symlinks=True*)

path の所有者 id およびグループ id を、数値 *uid* および *gid* に変更します。いずれかの id を変更せずにおくには、その値として -1 を指定します。

この関数は **ファイル記述子の指定** 、**ディレクトリ記述子への相対パス** 、および **シンボリックリンクをたどらない** をサポートしています。

数値 id の他に名前でも受け取る高水準関数の *shutil.chown()* を参照してください。

引数 path, uid, gid, dir_fd を指定して **監査イベント** os.chown を送出します。

利用可能な環境: Unix。

バージョン 3.3 で追加: *path* にオープンしているファイル記述子の指定のサポート、および引数 *dir_fd* と *follow_symlinks* を追加しました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.chroot(*path*)

現在のプロセスのルートディレクトリを *path* に変更します。

利用可能な環境: Unix。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.fchdir(*fd*)

現在の作業ディレクトリをファイル記述子 *fd* が表すディレクトリに変更します。記述子はオープンしているファイルではなく、オープンしたディレクトリを参照していなければなりません。Python 3.3 以降では os.chdir(fd) と等価です。

引数 path を指定して **監査イベント** os.chdir を送出します。

利用可能な環境: Unix。

os.getcwd()

現在の作業ディレクトリを表す文字列を返します。

os.getcwdb()

現在の作業ディレクトリを表すバイト列を返します。

バージョン 3.8 で変更: この関数は Windows において ANSI コードページ ではなく UTF-8 エンコーディングを使うようになりました: 変更の背景については **PEP 529** をご覧ください。この関数は Windows において非推奨になりません。

os.lchflags(*path*, *flags*)

path のフラグを数値 *flags* に設定します。*chflags()* に似ていますが、シンボリックリンクをたどりません。Python 3.3 以降では os.chflags(path, flags, follow_symlinks=False) と等価です。

引数 path, flags を指定して **監査イベント** os.chflags を送出します。

利用可能な環境: Unix。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.lchmod(*path*, *mode*)

path のモードを数値 *mode* に変更します。パスがシンボリックリンクの場合はそのリンク先ではなくシンボリックリンクそのものに対して作用します。*mode* に指定できる値については *chmod()* のドキュメ

ントを参照してください。Python 3.3 以降では `os.chmod(path, mode, follow_symlinks=False)` と等価です。

引数 `path`, `mode`, `dir_fd` を指定して **監査イベント** `os.chmod` を送出します。

利用可能な環境: Unix。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

`os.lchown`(*path*, *uid*, *gid*)

path の所有者 id およびグループ id を、数値 *uid* および *gid* に変更します。この関数はシンボリックリンクをたどりません。Python 3.3 以降では `os.chown(path, uid, gid, follow_symlinks=False)` と等価です。

引数 `path`, `uid`, `gid`, `dir_fd` を指定して **監査イベント** `os.chown` を送出します。

利用可能な環境: Unix。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

`os.link`(*src*, *dst*, ***, *src_dir_fd=None*, *dst_dir_fd=None*, *follow_symlinks=True*)

src を指し示すハードリンク *dst* を作成します。

この関数は *src_dir_fd* と *dst_dir_fd* の両方またはどちらかに対し **ディレクトリ記述子への相対パス** および **シンボリックリンクをたどらない** をサポートしています。

引数 `src`, `dst`, `src_dir_fd`, `dst_dir_fd` を指定して **監査イベント** `os.link` を送出します。

Availability: Unix, Windows。

バージョン 3.2 で変更: Windows サポートを追加しました。

バージョン 3.3 で追加: 引数 *src_dir_fd*、*dst_dir_fd*、および *follow_symlinks* を追加しました。

バージョン 3.6 で変更: *src* と *dst* が *path-like object* を受け付けるようになりました。

`os.listdir`(*path='.'*)

path に指定したディレクトリに含まれるエントリ名のリストを返します。リストの順番は不定です。特別なエントリ `'.'` と `'..'` はリストに含まれません。この関数の呼び出し中にディレクトリからファイルが削除されたり、ディレクトリにファイルが追加されたりした場合、それらのファイルがリストに含まれるかどうかは不定です。

path に path-like オブジェクト を指定することもできます。*path* が (直接的または間接的に *PathLike* インターフェースを介した) `bytes` 型の場合、戻り値のファイル名も `bytes` 型になります; それ以外の場合、ファイル名は `str` 型です。

この関数は **ファイル記述子の指定** もサポートしています; ファイル記述子はディレクトリを参照していなくてはなりません。

引数 `path` を指定して **監査イベント** `os.listdir` を送出します。

注釈: **文字列型** のファイル名を **バイト列型** にエンコードするには、*fsencode()* を使用します。

参考:

ディレクトリエントリに加えてファイル属性情報も返す *scandir()* 関数の方が、多くの一般的な用途では使い勝手が良くなります。

バージョン 3.2 で変更: 引数 *path* は任意になりました。

バージョン 3.3 で追加: *path* へのオープン・ファイル記述子の指定をサポートしました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.lstat(*path*, ***, *dir_fd=None*)

与えられたパスに対して `lstat()` システムコールと同じ処理を行います。*stat()* と似ていますが、シンボリックリンクをたどりません。*stat_result* オブジェクトを返します。

シンボリックリンクをサポートしていないプラットフォームでは *stat()* の別名です。

Python 3.3 以降では `os.stat(path, dir_fd=dir_fd, follow_symlinks=False)` と等価です。

この関数は **ディレクトリ記述子への相対パス** もサポートすることができます。

参考:

stat() 関数。

バージョン 3.2 で変更: Windows 6.0 (Vista) のシンボリックリンクをサポートしました。

バージョン 3.3 で変更: 引数 *dir_fd* を追加しました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

バージョン 3.8 で変更: On Windows, now opens reparse points that represent another path (name surrogates), including symbolic links and directory junctions. Other kinds of reparse points are resolved by the operating system as for *stat()*.

os.mkdir(*path*, *mode=0o777*, ***, *dir_fd=None*)

ディレクトリ *path* を数値モード *mode* で作成します。

すでにディレクトリが存在したら、*FileExistsError* が上げられます。

いくつかのシステムにおいては *mode* は無視されます。それが使われる時には、最初に現在の umask 値でマスクされます。もし最後の 9 ビット (つまり *mode* の 8 進法表記の最後の 3 桁) を除いたビットが設定されていたら、それらの意味はプラットフォームに依存します。いくつかのプラットフォームではそれらは無視され、それらを設定するためには明示的に *chmod()* を呼ぶ必要があるでしょう。

この関数は **ディレクトリ記述子への相対パス** もサポートすることができます。

一時ディレクトリを作成することもできます： *tempfile* モジュールの *tempfile.mkdtemp()* 関数を参照してください。

引数 path, mode, dir_fd を指定して 監査イベント os.mkdir を送出します。

バージョン 3.3 で追加: 引数 *dir_fd* が追加されました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.makedirs(*name*, *mode=0o777*, *exist_ok=False*)

再帰的にディレクトリを作成する関数です。*mkdir()* と似ていますが、末端ディレクトリを作成するために必要なすべての中間ディレクトリも作成します。

The *mode* parameter is passed to *mkdir()* for creating the leaf directory; see *the mkdir() description* for how it is interpreted. To set the file permission bits of any newly-created parent directories you can set the umask before invoking *makedirs()*. The file permission bits of existing parent directories are not changed.

exist_ok の値が False の場合 (デフォルト)、対象のディレクトリがすでに存在すると *FileExistsError* を送出します。

注釈: 作成するパス要素に *pardir* (UNIX では "..") が含まれる場合、*makedirs()* は混乱します。

この関数は UNC パスを正しく扱えるようになりました。

引数 path, mode, dir_fd を指定して 監査イベント os.mkdir を送出します。

バージョン 3.2 で追加: 引数 *exist_ok* が追加されました。

バージョン 3.4.1 で変更: Python 3.4.1 より前、*exist_ok* が True でそのディレクトリが既存の場合でも、*makedirs()* は *mode* が既存ディレクトリのモードと合わない場合にはエラーにしようとしていました。このモードチェックの振る舞いを安全に実装することが出来なかったため、Python 3.4.1 でこのチェックは削除されました。bpo-21082 を参照してください。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

バージョン 3.7 で変更: The *mode* argument no longer affects the file permission bits of newly-created intermediate-level directories.

os.mkfifo(*path*, *mode=0o666*, ***, *dir_fd=None*)

FIFO (名前付きパイプ) *path* を数値モード *mode* で作成します。先に現在の umask 値でマスクされます。

この関数は ディレクトリ記述子への相対パス もサポートすることができます。

FIFO は通常のファイルのようにアクセスできるパイプです。FIFO は (例えば *os.unlink()* を使って) 削除されるまで存在しつづけます。一般的に、FIFO は " クライアント " と " サーバー " 形式のプロセス間でランデブーを行うために使われます : この時、サーバーは FIFO を読み込み用に、クライアントは書き出し用にオープンします。*mkfifo()* は FIFO をオープンしない --- 単にランデブーポイントを作成するだけ --- なので注意してください。

利用可能な環境: Unix。

バージョン 3.3 で追加: 引数 *dir_fd* が追加されました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.**mknod**(*path*, *mode=0o600*, *device=0*, ***, *dir_fd=None*)

path という名前で、ファイルシステムノード (ファイル、デバイス特殊ファイル、または名前つきパイプ) を作成します。*mode* は、作成するノードのアクセス権限とタイプの両方を stat.S_IFREG、stat.S_IFCHR、stat.S_IFBLK、および stat.S_IFIFO の組み合わせ (ビット単位の論理和) で指定します (これらの定数は *stat* で利用可能です)。stat.S_IFCHR と stat.S_IFBLK を指定した場合、*devide* は新しく作成されたデバイス特殊ファイルを (おそらく *os.makedev()* を使って) 定義し、それ以外の定数を指定した場合は無視されます。

この関数は **ディレクトリ記述子への相対パス** もサポートすることができます。

利用可能な環境: Unix。

バージョン 3.3 で追加: 引数 *dir_fd* が追加されました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.**major**(*device*)

RAW デバイス番号から、デバイスのメジャー番号を取り出します (通常 stat の st_dev か st_rdev フィールドです) 。

os.**minor**(*device*)

RAW デバイス番号から、デバイスのマイナー番号を取り出します (通常 stat の st_dev か st_rdev フィールドです) 。

os.**makedev**(*major*, *minor*)

メジャーおよびマイナーデバイス番号から、新しく RAW デバイス番号を作成します。

os.**pathconf**(*path*, *name*)

名前付きファイルに関連するシステム設定情報を返します。*name* には取得したい設定名を指定します ; これは定義済みのシステム値名の文字列で、多くの標準 (POSIX.1 、Unix 95 、Unix 98 その他) で定義されています。プラットフォームによっては別の名前も定義しています。ホストオペレーティングシステムの関知する名前は pathconf_names 辞書で与えられています。このマップ型オブジェクトに入っていない設定変数については、*name* に整数を渡してもかまいません。

name が不明の文字列である場合、*ValueError* を送出します。*name* の特定の値がホストシステムでサポートされていない場合、pathconf_names に含まれていたとしても、*errno.EINVAL* をエラー番号として *OSError* を送出します。

この関数は **ファイル記述子の指定** をサポートしています。

利用可能な環境: Unix。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.**pathconf_names**

pathconf() および *fpathconf()* が受理するシステム設定名を、ホストオペレーティングシステムで

定義されている整数値に対応付けている辞書です。この辞書はシステムでどの設定名が定義されているかを知るために利用できます。

利用可能な環境: Unix。

os.readlink(*path*, *, *dir_fd=None*)

シンボリックリンクが指しているパスを表す文字列を返します。返される値は絶対パスにも、相対パスにもなり得ます ; 相対パスの場合、`os.path.join(os.path.dirname(path), result)` を使って絶対パスに変換することができます。

If the *path* is a string object (directly or indirectly through a *PathLike* interface), the result will also be a string object, and the call may raise a UnicodeDecodeError. If the *path* is a bytes object (direct or indirectly), the result will be a bytes object.

この関数は ディレクトリ記述子への相対パス もサポートすることができます。

When trying to resolve a path that may contain links, use *realpath()* to properly handle recursion and platform differences.

Availability: Unix, Windows。

バージョン 3.2 で変更: Windows 6.0 (Vista) のシンボリックリンクをサポートしました。

バージョン 3.3 で追加: 引数 *dir_fd* が追加されました。

バージョン 3.6 で変更: Unix で、*path-like object* を受け取るようになりました。

バージョン 3.8 で変更: Windows で、*path-like object* と bytes オブジェクトを受け入れるようになりました。

バージョン 3.8 で変更: Added support for directory junctions, and changed to return the substitution path (which typically includes `\\?\` prefix) rather than the optional "print name" field that was previously returned.

os.remove(*path*, *, *dir_fd=None*)

Remove (delete) the file *path*. If *path* is a directory, an *IsADirectoryError* is raised. Use *rmdir()* to remove directories. If the file does not exist, a *FileNotFoundError* is raised.

この関数は ディレクトリ記述子への相対パス をサポートしています。

Windows では、使用中のファイルを削除しようとすると例外を送出します; Unix では、ディレクトリエントリは削除されますが、記憶装置上に割り当てられたファイル領域は元のファイルが使われなくなるまで残されます。

この関数は意味論的に *unlink()* と同一です。

引数 `path`, `dir_fd` を指定して 監査イベント `os.remove` を送出します。

バージョン 3.3 で追加: 引数 *dir_fd* が追加されました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.**removedirs**(*name*)

再帰的なディレクトリ削除関数です。*rmdir()* と同じように動作しますが、末端ディレクトリがうまく削除できるかぎり、*removedirs()* は *path* に現れる親ディレクトリをエラーが送出されるまで (このエラーは通常、指定したディレクトリの親ディレクトリが空でないことを意味するだけなので無視されます) 順に削除することを試みます。例えば、`os.removedirs('foo/bar/baz')` では最初にディレクトリ `'foo/bar/baz'` を削除し、次に `'foo/bar'` さらに `'foo'` をそれらが空ならば削除します。末端のディレクトリが削除できなかった場合には *OSError* が送出されます。

引数 `path`, `dir_fd` を指定して **監査イベント** `os.remove` を送出します。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.**rename**(*src*, *dst*, ***, *src_dir_fd=None*, *dst_dir_fd=None*)

Rename the file or directory *src* to *dst*. If *dst* exists, the operation will fail with an *OSError* subclass in a number of cases:

On Windows, if *dst* exists a *FileExistsError* is always raised.

On Unix, if *src* is a file and *dst* is a directory or vice-versa, an *IsADirectoryError* or a *NotADirectoryError* will be raised respectively. If both are directories and *dst* is empty, *dst* will be silently replaced. If *dst* is a non-empty directory, an *OSError* is raised. If both are files, *dst* it will be replaced silently if the user has permission. The operation may fail on some Unix flavors if *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement).

この関数は *src_dir_fd* と *dst_dir_fd* のどちらかまたは両方の指定に **ディレクトリ記述子への相対パス** をサポートしています。

対象の上書きがクロスプラットフォームになる場合は *replace()* を使用してください。

引数 `src`, `dst`, `src_dir_fd`, `dst_dir_fd` を指定して **監査イベント** `os.rename` を送出します。

バージョン 3.3 で追加: 引数 *src_dir_fd* および *dst_dir_fd* が追加されました。

バージョン 3.6 で変更: *src* と *dst* が *path-like object* を受け付けるようになりました。

os.**renames**(*old*, *new*)

再帰的にディレクトリやファイル名を変更する関数です。*rename()* のように動作しますが、新たなパス名を持つファイルを配置するために必要な途中のディレクトリ構造をまず作成しようと試みます。名前変更の後、元のファイル名のパス要素は *removedirs()* を使って右側から順に削除されます。

注釈: この関数はコピー元の末端のディレクトリまたはファイルを削除する権限がない場合には失敗します。

引数 `src`, `dst`, `src_dir_fd`, `dst_dir_fd` を指定して **監査イベント** `os.rename` を送出します。

バージョン 3.6 で変更: *old* と *new* が *path-like object* を受け付けるようになりました。

os.**replace**(*src*, *dst*, ***, *src_dir_fd=None*, *dst_dir_fd=None*)

ファイルまたはディレクトリ *src* の名前を *dst* へ変更します。*dst* がディレクトリの場合 *OSError* が送出されます。*dst* が存在し、かつファイルの場合、ユーザーの権限がある限り暗黙のうちに置き換えられます。*src* と *dst* が異なるファイルシステム上にあると失敗することがあります。ファイル名の変更が成功する場合はアトミック操作となります (これは POSIX 要求仕様です)。

この関数は *src_dir_fd* と *dst_dir_fd* のどちらかまたは両方の指定に **ディレクトリ記述子への相対パス** をサポートしています。

引数 `src`, `dst`, `src_dir_fd`, `dst_dir_fd` を指定して **監査イベント** `os.rename` を送出します。

バージョン 3.3 で追加.

バージョン 3.6 で変更: *src* と *dst* が *path-like object* を受け付けるようになりました。

os.**rmdir**(*path*, ***, *dir_fd=None*)

Remove (delete) the directory *path*. If the directory does not exist or is not empty, an *FileNotFoundError* or an *OSError* is raised respectively. In order to remove whole directory trees, *shutil.rmtree()* can be used.

この関数は **ディレクトリ記述子への相対パス** をサポートしています。

引数 `path`, `dir_fd` を指定して **監査イベント** `os.rmdir` を送出します。

バージョン 3.3 で追加: 引数 *dir_fd* が追加されました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.**scandir**(*path='.'*)

Return an iterator of *os.DirEntry* objects corresponding to the entries in the directory given by *path*. The entries are yielded in arbitrary order, and the special entries `'.'` and `'..'` are not included. If a file is removed from or added to the directory after creating the iterator, whether an entry for that file be included is unspecified.

listdir() の代わりに *scandir()* を使用すると、ファイルタイプや属性情報も必要とするコードのパフォーマンスが大幅に向上します。これは、オペレーティングシステムがディレクトリのスキャン中にこの情報を提供した場合、*os.DirEntry* オブジェクトがその情報を公開するからです。すべての *os.DirEntry* メソッドはシステムコールを実行する場合がありますが、*is_dir()* と *is_file()* は、通常はシンボリックリンクにしかシステムコールを必要としません。*os.DirEntry.stat()* は、Unix 上では常にシステムコールを必要としますが、Windows ではシンボリックリンク用にシステムコールを一つ必要とするだけです。

path may be a *path-like object*. If *path* is of type `bytes` (directly or indirectly through the *PathLike* interface), the type of the *name* and *path* attributes of each *os.DirEntry* will be `bytes`; in all other circumstances, they will be of type `str`.

この関数は **ファイル記述子の指定** もサポートしています; ファイル記述子はディレクトリを参照していなくてはなりません。

引数 `path` を指定して **監査イベント** `os.scandir` を送出します。

scandir() イテレータは、**コンテキストマネージャ** プロトコルをサポートし、次のメソッドを持ちます。

scandir.close()
イテレータを閉じ、獲得した資源を開放します。

この関数は、イテレータがすべて消費されるか、ガーベージコレクトされた、もしくはイテレート中にエラーが発生した際に自動的に呼び出されます。しかし、with 文を用いるか、明示的に呼び出すことを推奨します。

バージョン 3.6 で追加.

次の単純な例では、*scandir()* を使用して、指定した *path* 内の先頭が '.' でないすべてのファイル (ディレクトリを除く) をすべて表示します。entry.is_file() を呼び出しても、通常は追加のシステムコールは行われません:

```
with os.scandir(path) as it:
    for entry in it:
        if not entry.name.startswith('.') and entry.is_file():
            print(entry.name)
```

注釈: On Unix-based systems, *scandir()* uses the system's opendir() and readdir() functions. On Windows, it uses the Win32 FindFirstFileW and FindNextFileW functions.

バージョン 3.5 で追加.

バージョン 3.6 で追加: Added support for the *context manager* protocol and the *close()* method. If a *scandir()* iterator is neither exhausted nor explicitly closed a *ResourceWarning* will be emitted in its destructor.

関数が *path-like object* を受け入れるようになりました。

バージョン 3.7 で変更: Unix で **ファイル記述子の指定** のサポートが追加されました。

class os.DirEntry
ディレクトリエントリのファイルパスとその他のファイル属性を公開するために、*scandir()* が yield するオブジェクトです。

scandir() は、追加のシステムコールを実行することなく、この情報をできるだけ多く提供します。stat() または lstat() システムコールが実行された場合、os.DirEntry オブジェクトは結果をキャッシュします。

os.DirEntry インスタンスは、寿命の長いデータ構造に保存されることは想定されていません。ファイルメタデータが変更された場合や、*scandir()* が呼び出されてから長時間が経過した場合は、os.stat(entry.path) を呼び出して最新の情報を取得してください。

os.DirEntry のメソッドはオペレーティングシステムコールを実行する場合があるため、それらは *OSError* も送出する場合があります。エラーを細かく制御する必要がある場合、os.DirEntry のメソッドの一つの呼び出し時に *OSError* を捕捉して、適切な処理を行うことができます。

To be directly usable as a *path-like object*, os.DirEntry implements the *PathLike* interface.

os.DirEntry インスタンスの属性とメソッドは以下の通りです:

name

scandir() の *path* 引数に対して相対的な、エントリのベースファイル名です。

The *name* attribute will be bytes if the *scandir()* *path* argument is of type bytes and str otherwise. Use *fsdecode()* to decode byte filenames.

path

The entry's full path name: equivalent to os.path.join(scandir_path, entry.name) where *scandir_path* is the *scandir()* *path* argument. The path is only absolute if the *scandir()* *path* argument was absolute. If the *scandir()* *path* argument was a *file descriptor*, the *path* attribute is the same as the *name* attribute.

The *path* attribute will be bytes if the *scandir()* *path* argument is of type bytes and str otherwise. Use *fsdecode()* to decode byte filenames.

inode()

項目の inode 番号を返します。

結果は os.DirEntry オブジェクトにキャッシュされます。最新の情報を取得するには os.stat(entry.path, follow_symlinks=False).st_ino を使用してください。

Windows 上では、最初のキャッシュされていない呼び出しでシステムコールが必要ですが、Unix 上では必要ありません。

is_dir(***, *follow_symlinks=True*)

この項目がディレクトリまたはディレクトリへのシンボリックリンクである場合、True を返します。項目がそれ以外のファイルやそれ以外のファイルへのシンボリックリンクである場合や、もはや存在しない場合は False を返します。

follow_symlinks が False の場合、項目がディレクトリ (シンボリックリンクはたどりません) の場合にのみ True を返します。項目がディレクトリ以外のファイルである場合や、項目がもはや存在しない場合は False を返します。

結果は os.DirEntry オブジェクトにキャッシュされます。*follow_symlinks* が True の場合と False の場合とでは、別のオブジェクトにキャッシュされます。最新の情報を取得するには *stat.S_ISDIR()* と共に *os.stat()* を呼び出してください。

多くの場合、最初のキャッシュされない呼び出しでは、システムコールは必要とされません。具体的には、シンボリックリンク以外では、Windows も Unix もシステムコールを必要としません。ただし、dirent.d_type == DT_UNKNOWN を返す、ネットワークファイルシステムなどの特定の Unix ファイルシステムは例外です。項目がシンボリックリンクの場合、*follow_symlinks* が False の場合を除き、シンボリックリンクをたどるためにシステムコールが必要となります。

このメソッドは *PermissionError* のような *OSError* を送出することがありますが、*FileNotFoundError* は捕捉され送出されません。

is_file(*, *follow_symlinks=True*)
この項目がファイルまたはファイルへのシンボリックリンクである場合、True を返します。項目がディレクトリやファイル以外の項目へのシンボリックリンクである場合や、もはや存在しない場合は False を返します。

follow_symlinks が False の場合、項目がファイル (シンボリックリンクはたどりません) の場合にのみ True を返します。項目がディレクトリやその他のファイル以外の項目である場合や、項目がもはや存在しない場合は False を返します。

結果は os.DirEntry オブジェクトにキャッシュされます。キャッシュ、システムコール、例外は、*is_dir()* と同様に行われます。

is_symlink()
この項目がシンボリックリンクの場合 (たとえ破損していても)、True を返します。項目がディレクトリやあらゆる種類のファイルの場合、またはもはや存在しない場合は False を返します。

結果は os.DirEntry オブジェクトにキャッシュされます。最新の情報をフェッチするには *os.path.islink()* を呼び出してください。

多くの場合、最初のキャッシュされない呼び出しでは、システムコールは必要とされません。具体的には、Windows も Unix もシステムコールを必要としません。ただし、dirent.d_type == DT_UNKNOWN を返す、ネットワークファイルシステムなどの特定の Unix ファイルシステムは例外です。

このメソッドは *PermissionError* のような *OSError* を送出することがありますが、*FileNotFoundError* は捕捉され送出されません。

stat(*, *follow_symlinks=True*)
この項目の *stat_result* オブジェクトを返します。このメソッドは、デフォルトでシンボリックリンクをたどります。シンボリックリンクを開始するには、follow_symlinks=False 引数を追加します。

On Unix, this method always requires a system call. On Windows, it only requires a system call if *follow_symlinks* is True and the entry is a reparse point (for example, a symbolic link or directory junction).

Windows では、*stat_result* の st_ino 、st_dev 、st_nlink 属性は常にゼロに設定されます。これらの属性を取得するには、*os.stat()* を呼び出します。

結果は os.DirEntry オブジェクトにキャッシュされます。*follow_symlinks* が True の場合と False の場合とでは、別のオブジェクトにキャッシュされます。最新の情報を取得するには、*os.stat()* を呼び出してください。

os.DirEntry と *pathlib.Path* では、いくつかの属性やメソッドがよい対応関係にあります。特に、name 属性は同じ意味を持ちます。is_dir() 、is_file() 、is_symlink() 、stat() メソッドも同じ意味を持ちます。

バージョン 3.5 で追加.

バージョン 3.6 で変更: *PathLike* インターフェースをサポートしました。Windows で:class:*bytes* パスをサポートしました。

os.stat(*path*, *, *dir_fd=None*, *follow_symlinks=True*)

Get the status of a file or a file descriptor. Perform the equivalent of a `stat()` system call on the given path. *path* may be specified as either a string or bytes -- directly or indirectly through the *PathLike* interface -- or as an open file descriptor. Return a *stat_result* object.

この関数は通常はシンボリックリンクをたどります。シンボリックリンクに対して stat したい場合は `follow_symlinks=False` とするか、*lstat()* を利用してください。

この関数は **ファイル記述子の指定** および **シンボリックリンクをたどらない** をサポートしています。

On Windows, passing `follow_symlinks=False` will disable following all name-surrogate reparse points, which includes symlinks and directory junctions. Other types of reparse points that do not resemble links or that the operating system is unable to follow will be opened directly. When following a chain of multiple links, this may result in the original link being returned instead of the non-link that prevented full traversal. To obtain stat results for the final path in this case, use the *os.path.realpath()* function to resolve the path name as far as possible and call *lstat()* on the result. This does not apply to dangling symlinks or junction points, which will raise the usual exceptions.

以下はプログラム例です:

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
os.stat_result(st_mode=33188, st_ino=7876932, st_dev=234881026,
st_nlink=1, st_uid=501, st_gid=501, st_size=264, st_atime=1297230295,
st_mtime=1297230027, st_ctime=1297230027)
>>> statinfo.st_size
264
```

参考:

fstat() と *lstat()*。

バージョン 3.3 で追加: *dir_fd*, *follow_symlinks* 引数の追加、ファイル記述子の指定の追加。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

バージョン 3.8 で変更: On Windows, all reparse points that can be resolved by the operating system are now followed, and passing `follow_symlinks=False` disables following all name surrogate reparse points. If the operating system reaches a reparse point that it is not able to follow, *stat* now returns the information for the original path as if `follow_symlinks=False` had been specified instead of raising an error.

class os.stat_result

おおむね `stat` 構造体のメンバーに対応する属性を持つオブジェクトです。*os.stat()*、*os.fstat()*、*os.lstat()* の結果に使用されます。

属性:

st_mode
ファイルモード。ファイルタイプとファイルモードのビット（権限）。

st_ino
Platform dependent, but if non-zero, uniquely identifies the file for a given value of `st_dev`. Typically:

- the inode number on Unix,
- the file index on Windows

st_dev
このファイルが存在するデバイスの識別子。

st_nlink
ハードリンクの数。

st_uid
ファイル所有者のユーザ識別子。

st_gid
ファイル所有者のグループ識別子。

st_size
ファイルが通常のファイルまたはシンボリックリンクの場合、そのファイルのバイト単位でのサイズです。シンボリックリンクのサイズは、含まれるパス名の長さで、null バイトで終わることはありません。

タイムスタンプ:

st_atime
秒で表した最終アクセス時刻。

st_mtime
秒で表した最終内容更新時刻。

st_ctime
プラットフォーム依存:

- Unix ではメタデータの最終更新時刻
- Windows では作成時刻、単位は秒

st_atime_ns
ナノ秒 (整数) で表した最終アクセス時刻。

st_mtime_ns
ナノ秒 (整数) で表した最終内容更新時刻。

st_ctime_ns
プラットフォーム依存:

- Unix ではメタデータの最終更新時刻
- Windows で、ナノ秒 (整数) で表した作成時刻。

注釈: *st_atime* 、*st_mtime* 、および *st_ctime* 属性の厳密な意味や精度はオペレーティングシステムやファイルシステムによって変わります。例えば、FAT や FAT32 ファイルシステムを使用している Windows システムでは、*st_mtime* の精度は 2 秒であり、*st_atime* の精度は 1 日に過ぎません。詳しくはお使いのオペレーティングシステムのドキュメントを参照してください。

同じように、*st_atime_ns*、*st_mtime_ns*、および *st_ctime_ns* は常にナノ秒で表されますが、多くのシステムではナノ秒単位の精度では提供していません。ナノ秒単位の精度を提供するシステムであっても、*st_atime*、*st_mtime*、および *st_ctime* についてはそれらが格納される浮動小数点オブジェクトがそのすべてを保持できず、それ自体が少々不正確です。正確なタイムスタンプが必要な場合は、*st_atime_ns*、*st_mtime_ns*、および *st_ctime_ns* を使用するべきです。

(Linux のような) 一部の Unix システムでは、以下の属性が利用できる場合があります :

st_blocks
ファイルに対して割り当てられている 512 バイトのブロックの数です。ファイルにホール (hole) が含まれている場合、*st_size*/512 より小さくなる場合があります。

st_blksize
効率的なファイルシステム I/O のための「推奨される」ブロックサイズです。ファイルに、これより小さいチャンクで書き込むと、非効率的な読み込み、編集、再書き込みが起こる場合があります。

st_rdev
inode デバイスの場合デバイスタイプ

st_flags
ファイルのユーザ定義フラグ

他の (FreeBSD のような) Unix システムでは、以下の属性が利用できる場合があります (ただし root ユーザ以外が使うと値が入っていない場合があります):

st_gen
ファイル生成番号

st_birthtime
ファイル作成時刻

On Solaris and derivatives, the following attributes may also be available:

st_fstype
String that uniquely identifies the type of the filesystem that contains the file.

Mac OS システムでは、以下の属性も利用できる場合があります:

st_rsize
ファイルの実際のサイズ

st_creator
ファイルの作成者

st_type
ファイルタイプ

On Windows systems, the following attributes are also available:

st_file_attributes
Windows のファイルの属性。GetFileInformationByHandle() の返す BY_HANDLE_FILE_INFORMATION 構造の dwFileAttributes メンバーです。*stat* モジュールの FILE_ATTRIBUTE_* 定数を参照してください。

st_reparse_tag
When *st_file_attributes* has the FILE_ATTRIBUTE_REPARSE_POINT set, this field contains the tag identifying the type of reparse point. See the IO_REPARSE_TAG_* constants in the *stat* module.

標準モジュール *stat* は stat 構造体からの情報の取り出しに役立つ関数と定数を定義しています。(Windows では、一部のアイテムにダミー値が入ります)

後方互換性のため、*stat_result* インスタンスには、stat 構造体の最も重要な (そして移植性の高い) メンバーを表す少なくとも 10 個の整数からなるタプルとしてもアクセス可能です。このタプルは、*st_mode*、*st_ino*、*st_dev*、*st_nlink*、*st_uid*、*st_gid*、*st_size*、*st_atime*、*st_mtime*、*st_ctime* の順になります。実装によってはそれ以上のアイテムが末尾に追加されます。古いバージョンの Python との互換性のため、*stat_result* にタプルとしてアクセスすると、常に整数を返します。

バージョン 3.3 で追加: *st_atime_ns*、*st_mtime_ns*、*st_ctime_ns* メンバが追加されました。

バージョン 3.5 で追加: Windows において *st_file_attributes* メンバが追加されました。

バージョン 3.5 で変更: Windows now returns the file index as *st_ino* when available.

バージョン 3.7 で追加: Added the *st_fstype* member to Solaris/derivatives.

バージョン 3.8 で追加: Added the *st_reparse_tag* member on Windows.

バージョン 3.8 で変更: On Windows, the *st_mode* member now identifies special files as S_IFCHR, S_IFIFO or S_IFBLK as appropriate.

os.**statvfs**(*path*)
与えられたパスに対して statvfs() システムコールを実行します。返り値はオブジェクトで、その属性は与えられたパスが格納されているファイルシステムについて記述したものです。各属性は statvfs 構造体のメンバーに対応します : f_bsize, f_frsize, f_blocks, f_bfree, f_bavail, f_files, f_ffree, f_favail, f_flag, f_namemax, :attr:‘f_fsid 。

f_flag 属性のビットフラグ用に 2 つのモジュールレベル定数が定義されています: ST_RDONLY が設定されるとファイルシステムは読み出し専用でマウントされ、ST_NOSUID が設定されると setuid/setgid

ビットの動作は無効になるか、サポートされません。

GNU/glibc ベースのシステム用に、追加のモジュールレベルの定数が次のように定義されています。ST_NODEV (デバイス特殊ファイルへのアクセスを許可しない) 、ST_NOEXEC (プログラムの実行を許可しない) 、ST_SYNCHRONOUS (書き込みが一度に同期される) 、ST_MANDLOCK (ファイルシステムで強制的なロックを許可する) 、ST_WRITE (ファイル/ディレクトリ/シンボリックリンクに書き込む) 、ST_APPEND (追記のみのファイル) 、ST_IMMUTABLE (変更不能なファイル) 、ST_NOATIME (アクセス時刻を更新しない) 、ST_NODIRATIME (ディレクトリアクセス時刻を更新しない) 、ST_RELATIME (mtime/ctime に対して相対的に atime を更新する)。

この関数は **ファイル記述子の指定** をサポートしています。

利用可能な環境: Unix。

バージョン 3.2 で変更: 定数 ST_RDONLY および ST_NOSUID が追加されました。

バージョン 3.3 で追加: *path* へのオープン・ファイル記述子の指定をサポートしました。

バージョン 3.4 で変更: ST_NODEV, ST_NOEXEC, ST_SYNCHRONOUS, ST_MANDLOCK, ST_WRITE, ST_APPEND, ST_IMMUTABLE, ST_NOATIME, ST_NODIRATIME, ST_RELATIME 定数が追加されました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

バージョン 3.7 で追加: f_fsid が追加されました。

os.supports_dir_fd

A *set* object indicating which functions in the *os* module accept an open file descriptor for their *dir_fd* parameter. Different platforms provide different features, and the underlying functionality Python uses to implement the *dir_fd* parameter is not available on all platforms Python supports. For consistency's sake, functions that may support *dir_fd* always allow specifying the parameter, but will throw an exception if the functionality is used when it's not locally available. (Specifying None for *dir_fd* is always supported on all platforms.)

To check whether a particular function accepts an open file descriptor for its *dir_fd* parameter, use the in operator on supports_dir_fd. As an example, this expression evaluates to True if *os.stat()* accepts open file descriptors for *dir_fd* on the local platform:

```
os.stat in os.supports_dir_fd
```

現在 *dir_fd* 引数は Unix プラットフォームでのみ動作します。Windows で動作する関数はありません。

バージョン 3.3 で追加.

os.supports_effective_ids

A *set* object indicating whether *os.access()* permits specifying True for its *effective_ids* parameter on the local platform. (Specifying False for *effective_ids* is always supported on all platforms.) If the local platform supports it, the collection will contain *os.access()*; otherwise it will be empty.

This expression evaluates to `True` if *os.access()* supports `effective_ids=True` on the local platform:

```
os.access in os.supports_effective_ids
```

現在 *effective_ids* は Unix プラットフォームでのみサポートされています。Windows では動作しません。

バージョン 3.3 で追加.

os.supports_fd

A *set* object indicating which functions in the *os* module permit specifying their *path* parameter as an open file descriptor on the local platform. Different platforms provide different features, and the underlying functionality Python uses to accept open file descriptors as *path* arguments is not available on all platforms Python supports.

To determine whether a particular function permits specifying an open file descriptor for its *path* parameter, use the `in` operator on `supports_fd`. As an example, this expression evaluates to `True` if *os.chdir()* accepts open file descriptors for *path* on your local platform:

```
os.chdir in os.supports_fd
```

バージョン 3.3 で追加.

os.supports_follow_symlinks

A *set* object indicating which functions in the *os* module accept `False` for their *follow_symlinks* parameter on the local platform. Different platforms provide different features, and the underlying functionality Python uses to implement *follow_symlinks* is not available on all platforms Python supports. For consistency's sake, functions that may support *follow_symlinks* always allow specifying the parameter, but will throw an exception if the functionality is used when it's not locally available. (Specifying `True` for *follow_symlinks* is always supported on all platforms.)

To check whether a particular function accepts `False` for its *follow_symlinks* parameter, use the `in` operator on `supports_follow_symlinks`. As an example, this expression evaluates to `True` if you may specify `follow_symlinks=False` when calling *os.stat()* on the local platform:

```
os.stat in os.supports_follow_symlinks
```

バージョン 3.3 で追加.

os.symlink(*src*, *dst*, *target_is_directory=False*, ***, *dir_fd=None*)

src を指し示すシンボリックリンク *dst* を作成します。

Windows では、シンボリックリンクはファイルかディレクトリのどちらかを表しますが、ターゲットに合わせて動的に変化することはありません。ターゲットが存在する場合、シンボリックリンクの種類は対象に合わせて作成されます。ターゲットが存在せず *target_is_directory* に `True` が設定された場合、シンボリックリンクはディレクトリのリンクとして作成され、`False` に設定された場合 (デフォルト) はファイルのリンクになります。Windows 以外のプラットフォームでは *target_is_directory* は無視されます。

この関数は ディレクトリ記述子への相対パス をサポートしています。

注釈: On newer versions of Windows 10, unprivileged accounts can create symlinks if Developer Mode is enabled. When Developer Mode is not available/enabled, the *SeCreateSymbolicLinkPrivilege* privilege is required, or the process must be run as an administrator.

この関数が特権を持たないユーザーに呼び出されると、*OSError* が送出されます。

引数 `src`, `dst`, `dir_fd` を指定して 監査イベント `os.symlink` を送出します。

Availability: Unix, Windows。

バージョン 3.2 で変更: Windows 6.0 (Vista) のシンボリックリンクをサポートしました。

バージョン 3.3 で追加: 引数 *dir_fd* が追加され、非 Windows プラットフォームでの *target_is_directory* 指定がサポートされました。

バージョン 3.6 で変更: *src* と *dst* が *path-like object* を受け付けるようになりました。

バージョン 3.8 で変更: Added support for unelevated symlinks on Windows with Developer Mode.

`os.sync()`

ディスクキャッシュのディスクへの書き出しを強制します。

利用可能な環境: Unix。

バージョン 3.3 で追加.

`os.truncate(`*path*, *length*`)`

path に対応するファイルを、サイズが最大で *length* バイトになるよう切り詰めます。

この関数は ファイル記述子の指定 をサポートしています。

引数 `path`, `length` を指定して 監査イベント `os.truncate` を送出します。

Availability: Unix, Windows。

バージョン 3.3 で追加.

バージョン 3.5 で変更: Windows サポートを追加しました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

`os.unlink(`*path*, *, *dir_fd=None*`)`

ファイル *path* を削除します。意味上は *remove()* と等価です。`unlink` の名前は伝統的な Unix の関数名です。詳細は *remove()* のドキュメントを参照してください。

引数 `path`, `dir_fd` を指定して 監査イベント `os.remove` を送出します。

バージョン 3.3 で追加: 引数 *dir_fd* が追加されました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.utime(*path*, *times=None*, **[*, *ns]*, *dir_fd=None*, *follow_symlinks=True*)

path で指定されたファイルに最終アクセス時刻および最終修正時刻を設定します。

utime() は 2 つの任意引数 *times* と *ns* をとります。これらは *path* に設定する時刻を指定し、以下のように使用されます:

- *ns* を指定する場合、ナノ秒を表す整数値をメンバーとして使用して、`(atime_ns, mtime_ns)` の形式の 2 要素タプルを指定する必要があります。
- *times* が `None` ではない場合、`(atime, mtime)` の形式で各メンバーは単位を秒で表す整数か浮動小数点値のタプルを指定しなければなりません。
- *times* が `None` で、*ns* が指定されていない場合、これは両方の時間を現在時刻として `ns=(atime_ns, mtime_ns)` を指定することと等価です。

times と *ns* の両方にタプルが指定されるとエラーになります。

Note that the exact times you set here may not be returned by a subsequent *stat()* call, depending on the resolution with which your operating system records access and modification times; see *stat()*. The best way to preserve exact times is to use the *st_atime_ns* and *st_mtime_ns* fields from the *os.stat()* result object with the *ns* parameter to *utime*.

この関数は **ファイル記述子の指定** 、**ディレクトリ記述子への相対パス** 、および **シンボリックリンクをたどらない** をサポートしています。

引数 `path`, `times`, `ns`, `dir_fd` を指定して **監査イベント** `os.utime` を送出します。

バージョン 3.3 で追加: *path* にオープンしているファイル記述子の指定のサポート、および引数 *dir_fd*, *follow_symlinks*, *ns* を追加しました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.walk(*top*, *topdown=True*, *onerror=None*, *followlinks=False*)

ディレクトリツリー以下のファイル名を、ツリーをトップダウンもしくはボトムアップに走査することで作成します。ディレクトリ *top* を根に持つディレクトリツリーに含まれる、各ディレクトリ (*top* 自身を含む) ごとに、タプル `(dirpath, dirnames, filenames)` を yield します。

dirpath is a string, the path to the directory. *dirnames* is a list of the names of the subdirectories in *dirpath* (excluding `'.'` and `'..'`). *filenames* is a list of the names of the non-directory files in *dirpath*. Note that the names in the lists contain no path components. To get a full path (which begins with *top*) to a file or directory in *dirpath*, do `os.path.join(dirpath, name)`. Whether or not the lists are sorted depends on the file system. If a file is removed from or added to the *dirpath* directory during generating the lists, whether a name for that file be included is unspecified.

オプション引数 *topdown* が `True` であるか、指定されなかった場合、各ディレクトリからタプルを生成した後で、サブディレクトリからタプルを生成します。(ディレクトリはトップダウンで生成) 。*topdown* が `False` の場合、ディレクトリに対応するタプルは、そのディレクトリ以下の全てのサブディレクトリに対応するタプルの後で (ボトムアップで) 生成されます。*topdown* の値によらず、サブディレクトリのリストは、ディレクトリとそのサブディレクトリのタプルを生成する前に取り出されます。

topdown が True のとき、呼び出し側は *dirnames* リストを、インプレースで (たとえば、del やスライスを使った代入で) 変更でき、*walk()* は *dirnames* に残っているサブディレクトリ内のみを再帰します。これにより、検索を省略したり、特定の訪問順序を強制したり、呼び出し側が *walk()* を再開する前に、呼び出し側が作った、または名前を変更したディレクトリを、*walk()* に知らせたりすることができます。*topdown* が False のときに *dirnames* を変更しても効果はありません。ボトムアップモードでは *dirpath* 自身が生成される前に *dirnames* 内のディレクトリの情報が生成されるからです。

デフォルトでは、*scandir()* 呼び出しからのエラーは無視されます。オプション引数の *onerror* を指定する場合は関数でなければなりません ; この関数は単一の引数として *OSError* インスタンスを伴って呼び出されます。この関数でエラーを報告して走査を継続したり、例外を送出して走査を中止したりできます。ファイル名は例外オブジェクトの filename 属性として利用できます。

デフォルトでは、*walk()* はディレクトリへのシンボリックリンクをたどりません。*followlinks* に True を指定すると、ディレクトリへのシンボリックリンクをサポートしているシステムでは、シンボリックリンクの指しているディレクトリを走査します。

注釈: *followlinks* に True を指定すると、シンボリックリンクが親ディレクトリを指していた場合に、無限ループになることに注意してください。*walk()* はすでにたどったディレクトリを管理したりはしません。

注釈: 相対パスを渡した場合、*walk()* が再開されるまでの間に現在の作業ディレクトリを変更しないでください。*walk()* はカレントディレクトリを変更しませんし、呼び出し側もカレントディレクトリを変更しないと仮定しています。

以下の例では、最初のディレクトリ以下にある各ディレクトリに含まれる、非ディレクトリファイルのバイト数を表示します。ただし、CVS サブディレクトリ以下は見に行きません

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print(root, "consumes", end=" ")
    print(sum(getsize(join(root, name)) for name in files), end=" ")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

次の例 (*shutil.rmtree()* の単純な実装) では、ツリーをボトムアップで走査することが不可欠になります; *rmdir()* はディレクトリが空になるまで削除を許さないからです:

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
```

(次のページに続く)

(前のページからの続き)

```
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
```

バージョン 3.5 で変更: この関数は、今では *os.listdir()* ではなく *os.scandir()* を呼び出します。これにより、*os.stat()* の呼び出し回数を削減でき、動作が高速化します。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.fwalk(*top='.'*, *topdown=True*, *onerror=None*, ***, *follow_symlinks=False*, *dir_fd=None*)

挙動は *walk()* と同じですが、dir_fd をサポートし、タプル (dirpath, dirnames, filenames, dirfd) を yield します。

dirpath、*dirnames*、および *filenames* は *walk()* の出力と同じで、*dirfd* は *dirpath* を参照するファイル記述子です。

この関数は常に **ディレクトリ記述子への相対パス** および **シンボリックリンクをたどらない** をサポートしています。ただし、他の関数と異なり、*fwalk()* での *follow_symlinks* のデフォルト値は False になることに注意してください。

注釈: *fwalk()* はファイル記述子を yield するため、それらが有効なのは次のイテレートステップまでです。それ以後も保持したい場合は *dup()* などを使って複製して使用してください。

以下の例では、最初のディレクトリ以下にある各ディレクトリに含まれる、非ディレクトリファイルのバイト数を表示します。ただし、CVS サブディレクトリ以下は見に行きません

```
import os
for root, dirs, files, rootfd in os.fwalk('python/Lib/email'):
    print(root, "consumes", end="")
    print(sum([os.stat(name, dir_fd=rootfd).st_size for name in files]),
          end="")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

次の例では、ツリーをボトムアップで走査することが不可欠になります；*rmdir()* はディレクトリが空になるまで削除を許さないからです

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files, rootfd in os.fwalk(top, topdown=False):
    for name in files:
```

(次のページに続く)

(前のページからの続き)

```
        os.unlink(name, dir_fd=rootfd)
    for name in dirs:
        os.rmdir(name, dir_fd=rootfd)
```

利用可能な環境: Unix。

バージョン 3.3 で追加.

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

バージョン 3.7 で変更: Added support for *bytes* paths.

os.memfd_create(*name*[, *flags=os.MFD_CLOEXEC*])

Create an anonymous file and return a file descriptor that refers to it. *flags* must be one of the `os.MFD_*` constants available on the system (or a bitwise ORed combination of them). By default, the new file descriptor is *non-inheritable*.

The name supplied in *name* is used as a filename and will be displayed as the target of the corresponding symbolic link in the directory `/proc/self/fd/`. The displayed name is always prefixed with `memfd:` and serves only for debugging purposes. Names do not affect the behavior of the file descriptor, and as such multiple files can have the same name without any side effects.

利用可能な環境: Linux 3.17 以上または glibc 2.27 以上。

バージョン 3.8 で追加.

os.MFD_CLOEXEC
os.MFD_ALLOW_SEALING
os.MFD_HUGETLB
os.MFD_HUGE_SHIFT
os.MFD_HUGE_MASK
os.MFD_HUGE_64KB
os.MFD_HUGE_512KB
os.MFD_HUGE_1MB
os.MFD_HUGE_2MB
os.MFD_HUGE_8MB
os.MFD_HUGE_16MB
os.MFD_HUGE_32MB
os.MFD_HUGE_256MB
os.MFD_HUGE_512MB
os.MFD_HUGE_1GB
os.MFD_HUGE_2GB
os.MFD_HUGE_16GB

These flags can be passed to *memfd_create()*.

Availability: Linux 3.17 or newer with glibc 2.27 or newer. The `MFD_HUGE*` flags are only available since Linux 4.14.

バージョン 3.8 で追加.

Linux 拡張属性

バージョン 3.3 で追加.

以下の関数はすべて Linux でのみ使用可能です。

os.getxattr(*path*, *attribute*, ***, *follow_symlinks=True*)

Return the value of the extended filesystem attribute *attribute* for *path*. *attribute* can be bytes or str (directly or indirectly through the *PathLike* interface). If it is str, it is encoded with the filesystem encoding.

この関数は **ファイル記述子の指定** および **シンボリックリンクをたどらない** をサポートしています。

引数 path, attribute を指定して **監査イベント** os.getxattr を送出します。

バージョン 3.6 で変更: *path* と *attribute* が *path-like object* を受け付けるようになりました。

os.listxattr(*path=None*, ***, *follow_symlinks=True*)

path の拡張ファイルシステム属性のリストを返します。リスト内の属性はファイルシステムのエンコーディングでデコードされた文字列で表されます。*path* が None の場合、*listxattr()* はカレントディレクトリを調べます。

この関数は **ファイル記述子の指定** および **シンボリックリンクをたどらない** をサポートしています。

引数 path を指定して **監査イベント** os.listxattr を送出します。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os.removexattr(*path*, *attribute*, ***, *follow_symlinks=True*)

Removes the extended filesystem attribute *attribute* from *path*. *attribute* should be bytes or str (directly or indirectly through the *PathLike* interface). If it is a string, it is encoded with the filesystem encoding.

この関数は **ファイル記述子の指定** および **シンボリックリンクをたどらない** をサポートしています。

引数 path, attribute を指定して **監査イベント** os.removexattr を送出します。

バージョン 3.6 で変更: *path* と *attribute* が *path-like object* を受け付けるようになりました。

os.setxattr(*path*, *attribute*, *value*, *flags=0*, ***, *follow_symlinks=True*)

Set the extended filesystem attribute *attribute* on *path* to *value*. *attribute* must be a bytes or str with no embedded NULs (directly or indirectly through the *PathLike* interface). If it is a str, it is encoded with the filesystem encoding. *flags* may be *XATTR_REPLACE* or *XATTR_CREATE*. If *XATTR_REPLACE* is given and the attribute does not exist, ENODATA will be raised. If *XATTR_CREATE* is given and the attribute already exists, the attribute will not be created and EEXISTS will be raised.

この関数は **ファイル記述子の指定** および **シンボリックリンクをたどらない** をサポートしています。

注釈: Linux カーネル 2.6.39 以前では、バグのため一部のファイルシステムで引数 flags が無視されます。

引数 path, attribute, value, flags を指定して 監査イベント os.setxattr を送出します。

バージョン 3.6 で変更: *path* と *attribute* が *path-like object* を受け付けるようになりました。

os.XATTR_SIZE_MAX
拡張属性の値にできる最大サイズです。現在、Linux では 64 キロバイトです。

os.XATTR_CREATE
setxattr() の引数 flags に指定できる値です。その操作で属性を作成しなければならないことを意味します。

os.XATTR_REPLACE
setxattr() の引数 flags に指定できる値です。その操作で既存の属性を置き換えなければならないことを意味します。

16.1.6 プロセス管理

以下の関数はプロセスの生成や管理に利用できます。

さまざまな *exec** 関数は、プロセス内にロードされる新しいプログラムに与えるための、引数のリストを取ります。どの関数の場合でも、新しいプログラムに渡されるリストの最初の引数は、ユーザがコマンドラインで入力する引数ではなく、そのプログラム自体の名前です。C プログラマならば、プログラムの main() に渡される argv[0] だと考えれば良いでしょう。たとえば、os.execv('/bin/echo', ['foo', 'bar']) が標準出力に出力するのは bar だけで、foo は無視されたかのように見えることになります。

os.abort()
SIGABRT シグナルを現在のプロセスに対して生成します。Unix では、デフォルトの動作はコアダンプの生成です ; Windows では、プロセスは即座に終了コード 3 を返します。この関数の呼び出しは *signal.signal()* を使って SIGABRT に対し登録された Python シグナルハンドラーを呼び出さないことに注意してください。

os.add_dll_directory(*path*)
Add a path to the DLL search path.

This search path is used when resolving dependencies for imported extension modules (the module itself is resolved through sys.path), and also by *ctypes*.

Remove the directory by calling **close()** on the returned object or using it in a with statement.

See the Microsoft documentation for more information about how DLLs are loaded.

引数 path を指定して 監査イベント os.add_dll_directory を送出します。

利用可能な環境: Windows 。

バージョン 3.8 で追加: Previous versions of CPython would resolve DLLs using the default behavior for the current process. This led to inconsistencies, such as only sometimes searching `PATH` or the current working directory, and OS functions such as `AddDllDirectory` having no effect.

In 3.8, the two primary ways DLLs are loaded now explicitly override the process-wide behavior to ensure consistency. See the porting notes for information on updating libraries.

os.execl(*path*, *arg0*, *arg1*, ...)
os.execle(*path*, *arg0*, *arg1*, ..., *env*)
os.execlp(*file*, *arg0*, *arg1*, ...)
os.execlpe(*file*, *arg0*, *arg1*, ..., *env*)
os.execv(*path*, *args*)
os.execve(*path*, *args*, *env*)
os.execvp(*file*, *args*)
os.execvpe(*file*, *args*, *env*)

これらの関数はすべて、現在のプロセスを置き換える形で新たなプログラムを実行します；現在のプロセスは返り値を返しません。Unix では、新たに実行される実行コードは現在のプロセス内に読み込まれ、呼び出し側と同じプロセス ID を持つことになります。エラーは *OSError* 例外として報告されます。

現在のプロセスは瞬時に置き換えられます。開かれているファイルオブジェクトやファイル記述子はフラッシュされません。そのため、バッファ内にデータが残っているかもしれない場合、*exec** 関数を実行する前に `sys.stdout.flush()` か *os.fsync()* を利用してバッファをフラッシュしておく必要があります。

"l" および "v" のついた *exec** 関数は、コマンドライン引数をどのように渡すかが異なります。"l" 型は、コードを書くときにパラメタ数が決まっている場合に、おそらくもっとも簡単に利用できます。個々のパラメタは単に `execl*()` 関数の追加パラメタとなります。"v" 型は、パラメタの数が可変の時に便利で、リストかタプルの引数が *args* パラメタとして渡されます。どちらの場合も、子プロセスに渡す引数は動作させようとしているコマンドの名前から始まるべきですが、これは強制されません。

末尾近くに "p" をもつ型 (*execlp()*, *execlpe()*, *execvp()*, および *execvpe()*) は、プログラム *file* を探すために環境変数 `PATH` を利用します。環境変数が (次の段で述べる *exec*e* 型関数で) 置き換えられる場合、環境変数は `PATH` を決定する上の情報源として使われます。その他の型、*execl()*, *execle()*, *execv()*, および *execve()* では、実行コードを探すために `PATH` を使いません。*path* には適切に設定された絶対パスまたは相対パスが入っていなくてはなりません。

execle() 、*execlpe()* 、*execve()* 、および *execvpe()* (すべて末尾に "e" がついています) では、*env* 引数は新たなプロセスで利用される環境変数を定義するためのマップ型でなくてはなりません (現在のプロセスの環境変数の代わりに利用されます); *execl()* 、*execlp()* 、*execv()* 、および *execvp()* では、すべて新たなプロセスは現在のプロセスの環境を引き継ぎます。

一部のプラットフォームの *execve()* では、*path* はオープンしているファイル記述子で指定することもできます。この機能をサポートしていないプラットフォームもあります; *os.supports_fd* を使うことで利用可能かどうか調べることができます。利用できない場合、*NotImplementedError* が送出されます。

引数 path, args, env を指定して 監査イベント os.exec を送出します。

Availability: Unix, Windows。

バージョン 3.3 で追加: Added support for specifying *path* as an open file descriptor for *execve()*.

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

os._exit(*n*)

終了ステータス *n* でプロセスを終了します。この時クリーンアップハンドラーの呼び出しや、標準入出力バッファのフラッシュなどは行いません。

注釈: 終了する標準的な方法は sys.exit(n) です。*_exit()* は通常、*fork()* された後の子プロセスでのみ使われます。

以下の終了コードは必須ではありませんが *_exit()* で使うことができます。一般に、メールサーバーの外部コマンド配送プログラムのような、Python で書かれたシステムプログラムに使います。

注釈: いくつかのバリエーションがあって、これらのすべてがすべての Unix プラットフォームで使えるわけではありません。以下の定数は下層のプラットフォームで定義されていれば定義されます。

---

os.EX_OK

エラーが起きなかったことを表す終了コード。

利用可能な環境: Unix。

os.EX_USAGE

誤った個数の引数が渡された時など、コマンドが間違って使われたことを表す終了コード。

利用可能な環境: Unix。

os.EX_DATAERR

入力データが誤っていたことを表す終了コード。

利用可能な環境: Unix。

os.EX_NOINPUT

入力ファイルが存在しなかった、または、読み込み不可だったことを表す終了コード。

利用可能な環境: Unix。

os.EX_NOUSER

指定されたユーザーが存在しなかったことを表す終了コード。

利用可能な環境: Unix。

os.EX_NOHOST

指定されたホストが存在しなかったことを表す終了コード。

利用可能な環境: Unix。

os.EX_UNAVAILABLE
要求されたサービスが利用できないことを表す終了コード。

利用可能な環境: Unix。

os.EX_SOFTWARE
内部ソフトウェアエラーが検出されたことを表す終了コード。

利用可能な環境: Unix。

os.EX_OSERR
fork できない、pipe の作成ができないなど、オペレーティングシステムのエラーが検出されたことを表す終了コード。

利用可能な環境: Unix。

os.EX_OSFILE
システムファイルが存在しなかった、開けなかった、あるいはその他のエラーが起きたことを表す終了コード。

利用可能な環境: Unix。

os.EX_CANTCREAT
ユーザーには作成できない出力ファイルを指定したことを表す終了コード。

利用可能な環境: Unix。

os.EX_IOERR
ファイルの I/O を行っている途中にエラーが発生した時の終了コード。

利用可能な環境: Unix。

os.EX_TEMPFAIL
一時的な失敗が発生したことを表す終了コード。これは、再試行可能な操作の途中に、ネットワークに接続できないというような、実際にはエラーではないかも知れないことを意味します。

利用可能な環境: Unix。

os.EX_PROTOCOL
プロトコル交換が不正、不適切、または理解不能なことを表す終了コード。

利用可能な環境: Unix。

os.EX_NOPERM
操作を行うために十分な許可がなかった（ファイルシステムの問題を除く）ことを表す終了コード。

利用可能な環境: Unix。

os.EX_CONFIG
設定エラーが起こったことを表す終了コード。

利用可能な環境: Unix。

os.EX_NOTFOUND
"an entry was not found" のようなことを表す終了コード。

利用可能な環境: Unix。

os.fork()
子プロセスを fork します。子プロセスでは 0 が返り、親プロセスでは子プロセスの id が返ります。エラーが発生した場合は、*OSError* を送出します。

FreeBSD 6.3 以下、Cygwin を含む一部のプラットフォームにおいて、fork() をスレッド内から利用した場合に既知の問題があることに注意してください。

引数無しで 監査イベント os.fork を送出します。

バージョン 3.8 で変更: Calling fork() in a subinterpreter is no longer supported (*RuntimeError* is raised).

> **警告:** SSL モジュールを fork() とともに使うアプリケーションについて、*ssl* を参照して下さい。

利用可能な環境: Unix。

os.forkpty()
子プロセスを fork します。この時新しい擬似端末を子プロセスの制御端末として使います。親プロセスでは (pid, fd) からなるペアが返り、*fd* は擬似端末のマスター側のファイル記述子となります。可搬性のあるアプローチを取るには、*pty* モジュールを利用してください。エラーが発生した場合は、*OSError* を送出します。

引数無しで 監査イベント os.forkpty を送出します。

バージョン 3.8 で変更: Calling forkpty() in a subinterpreter is no longer supported (*RuntimeError* is raised).

利用できる環境: 一部の Unix 互換環境。

os.kill(*pid*, *sig*)
プロセス *pid* にシグナル *sig* を送ります。ホストプラットフォームで利用可能なシグナルを特定する定数は *signal* モジュールで定義されています。

Windows: *signal.CTRL_C_EVENT* と *signal.CTRL_BREAK_EVENT* は、同じコンソールウィンドウを共有しているコンソールプロセス (例 : 子プロセス) にだけ送ることができる特別なシグナルです。その他の値を *sig* に与えると、そのプロセスが無条件に TerminateProcess API によって kill され、終了コードが *sig* に設定されます。Windows の *kill()* は kill するプロセスのハンドルも受け取ります。

signal.pthread_kill() も参照してください。

引数 pid, sig を指定して 監査イベント os.kill を送出します。

バージョン 3.2 で追加: Windows をサポートしました。

os.killpg(*pgid*, *sig*)
プロセスグループ *pgid* にシグナル *sig* を送ります。

引数 pgid, sig を指定して 監査イベント os.killpg を送出します。

利用可能な環境: Unix。

os.nice(*increment*)
プロセスの ”nice 値 ” に *increment* を加えます。新たな nice 値を返します。

利用可能な環境: Unix。

os.plock(*op*)
プログラムのセグメントをメモリ内にロックします。*op* (<sys/lock.h> で定義されています) にはどのセグメントをロックするかを指定します。

利用可能な環境: Unix。

os.popen(*cmd*, *mode='r'*, *buffering=-1*)
コマンド *cmd* への、または *cmd* からのパイプ入出力を開きます。戻り値はパイプに接続されている開かれたファイルオブジェクトで、*mode* が 'r' (デフォルト) または 'w' かによって読み出しまたは書き込みを行うことができます。引数 *bufsize* は、組み込み関数 *open()* における対応する引数と同じ意味を持ちます。返されるファイルオブジェクトは、バイトではなくテキスト文字列を読み書きします。

close メソッドは、サブプロセスが正常に終了した場合は *None* を返し、エラーが発生した場合にはサブプロセスの返りコードを返します。POSIX システムでは、返りコードが正の場合、そのコードは 1 バイト左にシフトしてプロセスが終了したことを示します。返りコードが負の場合、プロセスは返りコードの符号を変えた信号により終了します 。(例えば、サブプロセスが kill された場合、返り値は -signal.SIGKILL となる場合があります。) Windows システムでは、返り値には子プロセスからの符号のついた整数の返りコードを含まれます。

これは、*subprocess.Popen* を使用して実装されています。サブプロセスを管理し、サブプロセスと通信を行うためのより強力な方法については、クラスのドキュメンテーションを参照してください。

os.posix_spawn(*path*, *argv*, *env*, ***, *file_actions=None*, *setpgroup=None*, *resetids=False*, *setsid=False*, *setsigmask=()*, *setsigdef=()*, *scheduler=None*)
C ライブラリ API の posix_spawn() を Python から利用できるようにラップしたものです。

大部分のユーザーは posix_spawn`ではなく、func:`subprocess.run() を使うべきです。

位置引数 *path*, *args*, *env* は *execve()* と同じように解釈されます。

path には実行ファイルへのパスを指定します。*path* はディレクトリを含む形（実行ファイルへの絶対パスまたは相対パス）で指定する必要があります。実行ファイル名のみを指定したい場合は *posix_spawnp()* を使ってください。

file_actions 引数は C ライブラリ実装の fork() と exec() の間で子プロセスが持つファイルデスクリプタに対して行うアクションを記述するタプルのシーケンスです。各タプルの最初の要素は、残りのタプル要素の解釈方法を指定する以下の 3 つの型指定子のうちのひとつでなければなりません。

os.POSIX_SPAWN_OPEN
(os.POSIX_SPAWN_OPEN, *fd*, *path*, *flags*, *mode*)

os.dup2(os.open(path, flags, mode), fd) を実行します。

os.POSIX_SPAWN_CLOSE
(os.POSIX_SPAWN_CLOSE, *fd*)

os.close(fd) を実行します。

os.POSIX_SPAWN_DUP2
(os.POSIX_SPAWN_DUP2, *fd*, *new_fd*)

os.dup2(fd, new_fd) を実行します。

これらのタプルは、posix_spawn() の準備のために使われる C ライブラリの posix_spawn_file_actions_addopen(), posix_spawn_file_actions_addclose(), および posix_spawn_file_actions_adddup2() の 3 つの API コールに対応します。

The *setpgroup* argument will set the process group of the child to the value specified. If the value specified is 0, the child's process group ID will be made the same as its process ID. If the value of *setpgroup* is not set, the child will inherit the parent's process group ID. This argument corresponds to the C library POSIX_SPAWN_SETPGROUP flag.

If the *resetids* argument is True it will reset the effective UID and GID of the child to the real UID and GID of the parent process. If the argument is False, then the child retains the effective UID and GID of the parent. In either case, if the set-user-ID and set-group-ID permission bits are enabled on the executable file, their effect will override the setting of the effective UID and GID. This argument corresponds to the C library POSIX_SPAWN_RESETIDS flag.

If the *setsid* argument is True, it will create a new session ID for *posix_spawn*. *setsid* requires POSIX_SPAWN_SETSID or POSIX_SPAWN_SETSID_NP flag. Otherwise, *NotImplementedError* is raised.

The *setsigmask* argument will set the signal mask to the signal set specified. If the parameter is not used, then the child inherits the parent's signal mask. This argument corresponds to the C library POSIX_SPAWN_SETSIGMASK flag.

The *sigdef* argument will reset the disposition of all signals in the set specified. This argument corresponds to the C library POSIX_SPAWN_SETSIGDEF flag.

The *scheduler* argument must be a tuple containing the (optional) scheduler policy and an instance of *sched_param* with the scheduler parameters. A value of None in the place of the scheduler policy indicates that is not being provided. This argument is a combination of the C library POSIX_SPAWN_SETSCHEDPARAM and POSIX_SPAWN_SETSCHEDULER flags.

引数 path, argv, env を指定して 監査イベント os.posix_spawn を送出します。

バージョン 3.8 で追加.

利用可能な環境: Unix。

os.posix_spawnp(*path*, *argv*, *env*, ***, *file_actions=None*, *setpgroup=None*, *resetids=False*, *setsid=False*, *setsigmask=()*, *setsigdef=()*, *scheduler=None*)

Wraps the `posix_spawnp()` C library API for use from Python.

Similar to *posix_spawn()* except that the system searches for the *executable* file in the list of directories specified by the `PATH` environment variable (in the same way as for `execvp(3)`).

引数 `path`, `argv`, `env` を指定して **監査イベント** `os.posix_spawn` を送出します。

バージョン 3.8 で追加.

Availability: See *posix_spawn()* documentation.

os.register_at_fork(***, *before=None*, *after_in_parent=None*, *after_in_child=None*)

Register callables to be executed when a new child process is forked using *os.fork()* or similar process cloning APIs. The parameters are optional and keyword-only. Each specifies a different call point.

- *before* is a function called before forking a child process.
- *after_in_parent* is a function called from the parent process after forking a child process.
- *after_in_child* is a function called from the child process.

These calls are only made if control is expected to return to the Python interpreter. A typical *subprocess* launch will not trigger them as the child is not going to re-enter the interpreter.

Functions registered for execution before forking are called in reverse registration order. Functions registered for execution after forking (either in the parent or in the child) are called in registration order.

Note that `fork()` calls made by third-party C code may not call those functions, unless it explicitly calls `PyOS_BeforeFork()`, `PyOS_AfterFork_Parent()` and `PyOS_AfterFork_Child()`.

There is no way to unregister a function.

利用可能な環境: Unix。

バージョン 3.7 で追加.

os.spawnl(*mode*, *path*, *...*)
os.spawnle(*mode*, *path*, *...*, *env*)
os.spawnlp(*mode*, *file*, *...*)
os.spawnlpe(*mode*, *file*, *...*, *env*)
os.spawnv(*mode*, *path*, *args*)
os.spawnve(*mode*, *path*, *args*, *env*)
os.spawnvp(*mode*, *file*, *args*)
os.spawnvpe(*mode*, *file*, *args*, *env*)

新たなプロセス内でプログラム *path* を実行します。

(*subprocess* モジュールが、新しいプロセスを実行して結果を取得するための、より強力な機能を提供しています。この関数の代わりに *subprocess* モジュールを利用することが推奨されています。

subprocess モジュールのドキュメントの、**古い関数を *subprocess* モジュールで置き換える** セクションを参照してください)

mode が *P_NOWAIT* の場合、この関数は新たなプロセスのプロセス ID を返します ; *mode* が *P_WAIT* の場合、子プロセスが正常に終了するとその終了コードが返ります。そうでない場合にはプロセスを kill したシグナル *signal* に対して `-signal` が返ります。Windows では、プロセス ID は実際にはプロセスハンドル値になるので、*waitpid()* 関数で使えます。

Note on VxWorks, this function doesn't return `-signal` when the new process is killed. Instead it raises OSError exception.

"l" および "v" のついた *spawn** 関数は、コマンドライン引数をどのように渡すかが異なります。"l" 型は、コードを書くときにパラメタ数が決まっている場合に、おそらくもっとも簡単に利用できます。個々のパラメタは単に `spawnl*()` 関数の追加パラメタとなります。"v" 型は、パラメタの数が可変の時に便利で、リストかタプルの引数が *args* パラメタとして渡されます。どちらの場合も、子プロセスに渡す引数は動作させようとしているコマンドの名前から始まらなければなりません。

末尾近くに "p" をもつ型 (*spawnlp()*, *spawnlpe()*, *spawnvp()*, *spawnvpe()*) は、プログラム *file* を探すために環境変数 `PATH` を利用します。環境変数が (次の段で述べる *spawn*e* 型関数で) 置き換えられる場合、環境変数は `PATH` を決定する上の情報源として使われます。その他の型、*spawnl()*, *spawnle()*, *spawnv()*, および *spawnve()* では、実行コードを探すために `PATH` を使いません。*path* には適切に設定された絶対パスまたは相対パスが入っていなくてはなりません。

spawnle(), *spawnlpe()*, *spawnve()*, および *spawnvpe()* (すべて末尾に "e" がついています) では、*env* 引数は新たなプロセスで利用される環境変数を定義するためのマップ型でなくてはなりません ; *spawnl()* 、*spawnlp()* 、*spawnv()* 、および *spawnvp()* では、すべて新たなプロセスは現在のプロセスの環境を引き継ぎます。*env* 辞書のキーと値はすべて文字列である必要があります。不正なキーや値を与えると関数が失敗し、127 を返します。

例えば、以下の *spawnlp()* および *spawnvpe()* 呼び出しは等価です

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

引数 `mode`, `path`, `args`, `env` を指定して **監査イベント** `os.spawn` を送出します。

Availability: Unix, Windows. *spawnlp()*, *spawnlpe()*, *spawnvp()* and *spawnvpe()* are not available on Windows. *spawnle()* and *spawnve()* are not thread-safe on Windows; we advise you to use the *subprocess* module instead.

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

`os.P_NOWAIT`

`os.P_NOWAITO`

*spawn** 関数ファミリに対する *mode* パラメタとして取れる値です。この値のいずれかを *mode* として与えた場合、`spawn*()` 関数は新たなプロセスが生成されるとすぐに、プロセスの ID を戻り値として

返ります。

Availability: Unix, Windows。

os.P_WAIT

*spawn** 関数ファミリに対する *mode* パラメタとして取れる値です。この値を *mode* として与えた場合、spawn*() 関数は新たなプロセスを起動して完了するまで返らず、プロセスがうまく終了した場合には終了コードを、シグナルによってプロセスが kill された場合には -signal を返します。

Availability: Unix, Windows。

os.P_DETACH

os.P_OVERLAY

*spawn** 関数ファミリに対する *mode* パラメタとして取れる値です。これらの値は上の値よりもやや可搬性において劣っています。*P_DETACH* は *P_NOWAIT* に似ていますが、新たなプロセスは呼び出しプロセスのコンソールから切り離され (detach) ます。*P_OVERLAY* が使われた場合、現在のプロセスは置き換えられます。したがって *spawn** は返りません。

利用可能な環境: Windows 。

os.startfile(*path*[, *operation*])

ファイルを関連付けられたアプリケーションを使ってスタートします。

operation が指定されないか、または 'open' である時、この動作は、Windows の Explorer 上でのファイルをダブルクリックした、あるいはコマンドプロンプト上でファイル名を start コマンドの引数としての実行した場合と等価です : ファイルは拡張子が関連付けされているアプリケーション (が存在する場合) を使って開かれます。

他の *operation* が与えられる場合、それはファイルに対して何がなされるべきかを表す "command verb" (コマンドを表す動詞) でなければなりません。Microsoft が文書化している動詞は、'print' と 'edit' (ファイルに対して) および 'explore' と 'find' (ディレクトリに対して) です。

startfile() は関連付けされたアプリケーションが起動すると同時に返ります。アプリケーションが閉じるまで待機させるためのオプションはなく、アプリケーションの終了状態を取得する方法もありません。引数 *path* はカレントディレクトリからの相対パスです。絶対パスで指定したい場合は、最初の文字はスラッシュ ('/') ではないので注意してください。最初の文字がスラッシュの場合、下層の Win32 ShellExecute() 関数は動作しません。*os.path.normpath()* 関数を使って、Win32 用に正しくコード化されたパスになるようにしてください。

インタープリタの起動時のオーバーヘッドを削減するため、この関数が最初に呼ばれるまで、Win32 ShellExecute() 関数は決定されません。関数を決定できない場合、*NotImplementedError* が送出されます。

引数 path, operation を指定して 監査イベント os.startfile を送出します。

利用可能な環境: Windows 。

os.system(*command*)

サブシェル内でコマンド (文字列) を実行します。この関数は標準 C 関数 system() を使って実装されており、system() と同じ制限があります。*sys.stdin* などに対する変更を行っても、実行されるコ

マンドの環境には反映されません。*command* が何らかの出力を生成した場合、インタープリターの標準出力ストリームに送られます。

Unix では、返り値はプロセスの終了ステータスで、*wait()* で定義されている書式にコード化されています。POSIX は `system()` 関数の返り値の意味について定義していないので、Python の *system()* における返り値はシステム依存となることに注意してください。

Windows では、返り値は *command* を実行した後にシステムシェルから返される値です。シェルは通常 **cmd.exe** であり、返す値は実行したコマンドの終了ステータスになります。シェルの種類は Windows の環境変数 `COMSPEC:` に指定されています。ネイティブでないシェルを使用している場合は、そのドキュメントを参照してください。

subprocess モジュールは、新しいプロセスを実行して結果を取得するためのより強力な機能を提供しています。この関数の代わりに *subprocess* モジュールを利用することが推奨されています。*subprocess* モジュールのドキュメントの **古い関数を** *subprocess* **モジュールで置き換える** 節のレシピを参考にして下さい。

引数 `command` を指定して **監査イベント** `os.system` を送出します。

Availability: Unix, Windows。

`os.times()`

現在の全体的なプロセス時間を返します。返り値は 5 個の属性を持つオブジェクトになります:

- `user` - ユーザー時間
- *system* - システム時間
- `children_user` - すべての子プロセスのユーザー時間
- `children_system` - すべての子プロセスのシステム時間
- `elapsed` - 去のある固定時点からの経過実時間

後方互換性のため、このオブジェクトは 5 個のアイテム `user` 、*system* 、`children_user` 、`children_system` 、および `elapsed` を持つタプルのようにも振る舞います。

See the Unix manual page *times(2)* and *times(3)* manual page on Unix or the GetProcessTimes MSDN on Windows. On Windows, only `user` and *system* are known; the other attributes are zero.

Availability: Unix, Windows。

バージョン 3.3 で変更: 返り値の型が、タプルから属性名のついたタプルライクオブジェクトに変更されました。

`os.wait()`

子プロセスの実行完了を待機し、子プロセスの pid と終了コードインジケーター --- 16 ビットの数値で、下位バイトがプロセスを kill したシグナル番号、上位バイトが終了ステータス (シグナル番号がゼロの場合) --- の入ったタプルを返します ; コアダンプファイルが生成された場合、下位バイトの最上桁ビットが立てられます。

利用可能な環境: Unix。

os.waitid(*idtype*, *id*, *options*)

一つ以上のプロセスの完了を待機します。*idtype* には *P_PID*、*P_PGID*、または *P_ALL* を指定できます。*id* は待機する pid を指定します。*options* は *WEXITED*、*WSTOPPED*、または *WCONTINUED* を一つ以上、論理和で指定でき、他に *WNOHANG* または *WNOWAIT* も追加できます。返り値は `siginfo_t` 構造体に含まれるデータ (`si_pid`、`si_uid`、`si_signo`、`si_status`、および `si_code`) を表すオブジェクトになります。*WNOHANG* が指定され、待機状態の子プロセスがない場合は `None` を返します。

利用可能な環境: Unix。

バージョン 3.3 で追加.

os.P_PID

os.P_PGID

os.P_ALL

waitid() の *idtype* に指定できる値です。これらは *id* がどう解釈されるかに影響します。

利用可能な環境: Unix。

バージョン 3.3 で追加.

os.WEXITED

os.WSTOPPED

os.WNOWAIT

waitid() の *options* で使用できるフラグです。子プロセスのどのシグナルを待機するかを指定します。

利用可能な環境: Unix。

バージョン 3.3 で追加.

os.CLD_EXITED

os.CLD_DUMPED

os.CLD_TRAPPED

os.CLD_CONTINUED

waitid() の返り値の `si_code` に設定され得る値です。

利用可能な環境: Unix。

バージョン 3.3 で追加.

os.waitpid(*pid*, *options*)

この関数の詳細は Unix と Windows で異なります。

Unix の場合：プロセス id *pid* で与えられた子プロセスの完了を待機し、子プロセスのプロセス id と (*wait()* と同様にコード化された) 終了ステータスインジケーターからなるタプルを返します。この関数の動作は *options* によって変わります。通常の操作では `0` にします。

pid が `0` よりも大きい場合、*waitpid()* は特定のプロセスのステータス情報を要求します。*pid* が `0` の場合、現在のプロセスグループ内の任意の子プロセスの状態に対する要求です。*pid* が `-1` の場合、

現在のプロセスの任意の子プロセスに対する要求です。*pid* が -1 よりも小さい場合、プロセスグループ -pid (すなわち *pid* の絶対値) 内の任意のプロセスに対する要求です。

システムコールが -1 を返した時、*OSError* を errno と共に送出します。

Windows では、プロセスハンドル *pid* を指定してプロセスの終了を待って、*pid* と、終了ステータスを 8bit 左シフトした値のタプルを返します。(シフトは、この関数をクロスプラットフォームで利用しやすくするために行われます) 0 以下の *pid* は Windows では特別な意味を持っておらず、例外を発生させます。*options* の値は効果がありません。*pid* は、子プロセスで無くても、プロセス ID を知っているどんなプロセスでも参照することが可能です。*spawn** 関数を *P_NOWAIT* と共に呼び出した場合、適切なプロセスハンドルが返されます。

バージョン 3.5 で変更: システムコールが中断されシグナルハンドラが例外を送出しなかった場合、この関数は *InterruptedError* 例外を送出する代わりにシステムコールを再試行するようになりました (論拠については **PEP 475** を参照してください)。

os.wait3(*options*)

waitpid() に似ていますが、プロセス id を引数に取らず、子プロセス id 、終了ステータスインジケータ、リソース使用情報の 3 要素からなるタプルを返します。リソース使用情報の詳しい情報は *resource*. *getrusage()* を参照してください。オプション引数は *waitpid()* および *wait4()* と同じです。

利用可能な環境: Unix。

os.wait4(*pid*, *options*)

waitpid() に似ていますが、子プロセス id 、終了ステータスインジケータ、リソース使用情報の 3 要素からなるタプルを返します。リソース使用情報の詳しい情報は *resource*. *getrusage()* を参照してください。*wait4()* の引数は *waitpid()* に与えられるものと同じです。

利用可能な環境: Unix。

os.WNOHANG

子プロセス状態がすぐに取得できなかった場合に直ちに終了するようにするための *waitpid()* のオプションです。この場合、関数は (0, 0) を返します。

利用可能な環境: Unix。

os.WCONTINUED

このオプションによって子プロセスは前回状態が報告された後にジョブ制御による停止状態から実行を再開された場合に報告されるようになります。

利用可能な環境: 一部の Unix システム。

os.WUNTRACED

このオプションによって子プロセスは停止されていながら停止されてから状態が報告されていない場合に報告されるようになります。

利用可能な環境: Unix。

以下の関数は *system()* 、*wait()* 、あるいは *waitpid()* が返すプロセス状態コードを引数にとります。これらの関数はプロセスの配置を決めるために利用できます。

os.WCOREDUMP(*status*)

プロセスに対してコアダンプが生成されていた場合には True を、それ以外の場合は False を返します。

この関数は *WIFSIGNALED()* が真である場合のみ使用されるべきです。

利用可能な環境: Unix。

os.WIFCONTINUED(*status*)

停止していた子プロセスが *SIGCONT* シグナルの送信によって再開された場合 (ジョブ制御の停止から親プロセスの実行が継続している場合) True を返します。そうでない場合は False を返します。

WCONTINUED オプションを参照してください。

利用可能な環境: Unix。

os.WIFSTOPPED(*status*)

プロセスがシグナルの送信によって中断させられた場合に True を返します。それ以外の場合は False を返します。

WIFSTOPPED() は *waitpid()* が *WUNTRACED* オプションを使って実行されたか、もしくはプロセスがトレースされている場合 (*ptrace(2)* を参照してください) にのみ True を返します。

利用可能な環境: Unix。

os.WIFSIGNALED(*status*)

プロセスがシグナルによって終了させられた場合に True を返します。そうでない場合は False を返します。

利用可能な環境: Unix。

os.WIFEXITED(*status*)

プロセスが正常終了した場合、すなわち exit() や _exit() を呼び出したか、もしくは main() から戻ることにより終了した場合に True を返します。それ以外は False を返します。

利用可能な環境: Unix。

os.WEXITSTATUS(*status*)

プロセスの終了ステータスを返します。

この関数は *WIFEXITED()* が真である場合のみ使用されるべきです。

利用可能な環境: Unix。

os.WSTOPSIG(*status*)

プロセスを停止させたシグナル番号を返します。

この関数は *WIFSTOPPED()* が真である場合のみ使用されるべきです。

利用可能な環境: Unix。

os.WTERMSIG(*status*)
プロセスを終了させたシグナルの番号を返します。

この関数は *WIFSIGNALED()* が真である場合のみ使用されるべきです。

利用可能な環境: Unix。

16.1.7 スケジューラーへのインターフェイス

以下の関数は、オペレーティングシステムがプロセスに CPU 時間を割り当てる方法を制御します。これらは一部の Unix プラットフォームでのみ利用可能です。詳しくは Unix マニュアルページを参照してください。

バージョン 3.3 で追加.

次のスケジューリングポリシーは、オペレーティングシステムでサポートされていれば公開されます。

os.SCHED_OTHER
デフォルトのスケジューリングポリシーです。

os.SCHED_BATCH
常に CPU に負荷のかかる (CPU-intensive) プロセス用のポリシーです。他の対話式プロセスなどの応答性を維持するよう試みます。

os.SCHED_IDLE
非常に優先度の低いバックグラウンドタスク用のスケジューリングポリシーです。

os.SCHED_SPORADIC
散発的なサーバープログラム用のスケジューリングポリシーです。

os.SCHED_FIFO
FIFO (First In, First Out) 型のスケジューリングポリシーです。

os.SCHED_RR
ラウンドロビン型のスケジューリングポリシーです。

os.SCHED_RESET_ON_FORK
このフラグは他のスケジューリングポリシーとともに論理和指定できます。このフラグが与えられたプロセスが fork されると、その子プロセスのスケジューリングポリシーおよび優先度はデフォルトにリセットされます。

class os.sched_param(*sched_priority*)
このクラスは、*sched_setparam()*、*sched_setscheduler()*、および *sched_getparam()* で使用される、調節可能なスケジューリングパラメーターを表します。これはイミュータブルです。

現在、一つの引数のみ指定できます:

sched_priority
スケジューリングポリシーのスケジューリング優先度です。

os.**sched_get_priority_min**(*policy*)

policy の最小優先度値を取得します。*policy* には上記のスケジューリングポリシー定数の一つを指定します。

os.**sched_get_priority_max**(*policy*)

policy の最大優先度値を取得します。*policy* には上記のスケジューリングポリシー定数の一つを指定します。

os.**sched_setscheduler**(*pid*, *policy*, *param*)

PID *pid* のプロセスのスケジューリングポリシーを設定します。*pid* が 0 の場合、呼び出しプロセスを意味します。*policy* には上記のスケジューリングポリシー定数の一つを指定します。*param* は *sched_param* のインスタンスです。

os.**sched_getscheduler**(*pid*)

PID *pid* のプロセスのスケジューリングポリシーを返します。*pid* が 0 の場合、呼び出しプロセスを意味します。返り値は上記のスケジューリングポリシー定数の一つになります。

os.**sched_setparam**(*pid*, *param*)

PID *pid* のプロセスのスケジュールパラメーターを設定します。*pid* が 0 の場合、呼び出しプロセスを意味します。*param* は *sched_param* のインスタンスです。

os.**sched_getparam**(*pid*)

PID *pid* のプロセスのスケジューリングパラメーターを *sched_param* のインスタンスとして返します。*pid* が 0 の場合、呼び出しプロセスを意味します。

os.**sched_rr_get_interval**(*pid*)

PID *pid* のプロセスのラウンドロビンクォンタム (秒) を返します。*pid* が 0 の場合、呼び出しプロセスを意味します。

os.**sched_yield**()

自発的に CPU を解放します。

os.**sched_setaffinity**(*pid*, *mask*)

PID *pid* のプロセス (0 であれば現在のプロセス) を CPU の集合に制限します。*mask* はプロセスを制限する CPU の集合を表す整数のイテラブルなオブジェクトです。

os.**sched_getaffinity**(*pid*)

PID *pid* のプロセス (0 の場合、現在のプロセス) が制限されている CPU の集合を返します。

16.1.8 雑多なシステム情報

os.**confstr**(*name*)

システム設定値を文字列で返します。*name* には取得したい設定名を指定します ; この値は定義済みのシステム値名を表す文字列にすることができます ; 名前は多くの標準 (POSIX.1 、Unix 95 、Unix 98 その他) で定義されています。ホストオペレーティングシステムの関知する名前は `confstr_names` 辞書のキーとして与えられています。このマップ型オブジェクトに入っていない設定変数については、*name* に整数を渡してもかまいません。

name に指定された設定値が定義されていない場合、None を返します。

name が文字列で、かつ不明の場合、*ValueError* を送出します。*name* の指定値がホストシステムでサポートされておらず、confstr_names にも入っていない場合、*errno.EINVAL* をエラー番号として *OSError* を送出します。

利用可能な環境: Unix。

os.confstr_names
: *confstr()* が受理する名前を、ホストオペレーティングシステムで定義されている整数値に対応付けている辞書です。この辞書はシステムでどの設定名が定義されているかを決定するために利用できます。

利用可能な環境: Unix。

os.cpu_count()
: システムの CPU 数を返します。未定の場合は None を返します。

この数は現在のプロセスが使える CPU 数と同じものではありません。使用可能な CPU 数は len(os.sched_getaffinity(0)) で取得できます。

バージョン 3.4 で追加.

os.getloadavg()
: 過去 1 分、5 分、および 15 分間の、システムの実行キューの平均プロセス数を返します。平均負荷が得られない場合には *OSError* を送出します。

利用可能な環境: Unix。

os.sysconf(*name*)
: 整数値のシステム設定値を返します。*name* で指定された設定値が定義されていない場合、-1 が返されます。*name* に関するコメントとしては、*confstr()* で述べた内容が同様に当てはまります ; 既知の設定名についての情報を与える辞書は sysconf_names で与えられています。

利用可能な環境: Unix。

os.sysconf_names
: *sysconf()* が受理する名前を、ホストオペレーティングシステムで定義されている整数値に対応付けている辞書です。この辞書はシステムでどの設定名が定義されているかを決定するために利用できます。

利用可能な環境: Unix。

以下のデータ値はパス名編集操作をサポートするために利用されます。これらの値はすべてのプラットフォームで定義されています。

パス名に対する高水準の操作は *os.path* モジュールで定義されています。

os.curdir
: 現在のディレクトリ参照するためにオペレーティングシステムで使われる文字列定数です。POSIX と Windows では '.' になります。*os.path* からも利用できます。

os.pardir
: 親ディレクトリを参照するためにオペレーティングシステムで使われる文字列定数です。POSIX と Windows では '..' になります。*os.path* からも利用できます。

os.sep
: パス名を要素に分割するためにオペレーティングシステムで利用されている文字です。例えば POSIX では '/' で、Windows では '\\' です。しかし、このことを知っているだけではパス名を解析したり、パス名同士を結合したりするには不十分です --- こうした操作には *os.path.split()* や *os.path.join()* を使用してください --- が、たまに便利なこともあります。*os.path* からも利用できます。

os.altsep
: 文字パス名を要素に分割する際にオペレーティングシステムで利用されるもう一つの文字で、分割文字が一つしかない場合には None になります。この値は sep がバックスラッシュとなっている DOS や Windows システムでは '/' に設定されています。*os.path* からも利用できます。

os.extsep
: ベースのファイル名と拡張子を分ける文字です。例えば、os.py であれば '.' です。*os.path* からも利用できます。

os.pathsep
: (PATH のような) サーチパス内の要素を分割するためにオペレーティングシステムが慣習的に用いる文字で、POSIX における ':' や DOS および Windows における ';' に相当します。*os.path* からも利用できます。

os.defpath
: *exec*p** や *spawn*p** において、環境変数辞書内に 'PATH' キーがない場合に使われる標準設定のサーチパスです。*os.path* からも利用できます。

os.linesep
: 現在のプラットフォーム上で行を分割 (あるいは終端) するために用いられている文字列です。この値は例えば POSIX での '\n' や Mac OS での '\r' のように、単一の文字にもなりますし、例えば Windows での '\r\n' のように複数の文字列にもなります。テキストモードで開いたファイルに書き込む時には、*os.linesep* を利用しないでください。すべてのプラットフォームで、単一の '\n' を使用してください。

os.devnull
: ヌルデバイスのファイルパスです。例えば POSIX では '/dev/null' で、Windows では 'nul' です。この値は *os.path* からも利用できます。

os.RTLD_LAZY

os.RTLD_NOW

os.RTLD_GLOBAL

os.RTLD_LOCAL

os.RTLD_NODELETE

os.RTLD_NOLOAD

os.RTLD_DEEPBIND

setdlopenflags() 関数と *getdlopenflags()* 関数と一緒に使用するフラグ。それぞれのフラグの意味については、Unix マニュアルの *dlopen(3)* ページを参照してください。

バージョン 3.3 で追加.

16.1.9 乱数

os.getrandom(*size*, *flags=0*)

最大で *size* バイトからなるランダムなバイト列を返します。この関数は要求されたバイト数よりも少ないバイト数を返すことがあります。

バイト列は、ユーザー空間の乱数生成器や暗号目的ののシードとして利用できます。

getrandom() はデバイスドライバや他の環境ノイズ源から収集されたエントロピーに頼っています。不必要な大量のデータの読出しは、/dev/random と /dev/urandom デバイスの他のユーザーに負の影響を与えるでしょう。

flags 引数には、次に示す値の 0 個以上の論理和で与えられるビットマスクを指定できます: *os.GRND_RANDOM* および *GRND_NONBLOCK*。

Linux getrandom() manual page も参照してください。

利用可能な環境: Linux 3.17 以上。

バージョン 3.6 で追加.

os.urandom(*size*)

暗号に関する用途に適した *size* バイトからなるランダムな文字列を返します。

この関数は OS 固有の乱数発生源からランダムなバイト列を生成して返します。この関数の返すデータは暗号を用いたアプリケーションで十分利用できる程度に予測不能ですが、実際のクオリティは OS の実装によって異なります。

Linux では、getrandom() システムコールが利用可能ならブロッキングモードで呼び出されます: すなわちシステムの urandom エントロピープールが初期化されるまで (128 ビットのエントロピーがカーネルにより収集されるまで) 処理がブロックされます。論拠については **PEP 524** を参照してください。Linux では、(*GRND_NONBLOCK* フラグを使って) 非ブロッキングモードでランダムなバイトを取得したり、システムの urandom エントロピープールが初期化されるまでポーリングするために *getrandom()* 関数を利用することができます。

Unix ライクなシステムでは、ランダムなバイトは /dev/urandom デバイスから読み込みます。/dev/urandom デバイスが利用できないか、もしくは読み取り不可のときは、*NotImplementedError* 例外が送出されます。

Windows で、CryptGenRandom() を使用します。

参考:

secrets モジュールは高レベルの乱数生成機能を提供します。プラットフォームが提供する乱数生成器に対する簡便なインターフェースについては、*random.SystemRandom* を参照してください。

バージョン 3.6.0 で変更: Linux で、セキュリティを高めるために、getrandom() をブロッキングモードで使用するようになりました。

バージョン 3.5.2 で変更: Linux において、getrandom() システムコールがブロックするなら (urandom エントロピープールが初期化されていなければ) 、/dev/urandom を読む方法にフォールバックします。

バージョン 3.5 で変更: Linux 3.17 以降では、使用可能な場合に getrandom() システムコールが使用されるようになりました。OpenBSD 5.6 以降では、C getentropy() 関数が使用されるようになりました。これらの関数は、内部ファイル記述子を使用しません。

os.GRND_NONBLOCK

デフォルトでは、*getrandom()* は "/dev/random"から読み込んだときにランダムなバイトが存在しない場合や、"/dev/urandom"から読み込んだときにエントロピープールが初期化されていない場合に処理をブロックします。

GRND_NONBLOCK フラグがセットされると、*getrandom()* はこれらの場合に処理をブロックせず、ただちに *BlockingIOError* 例外を送出します。

バージョン 3.6 で追加.

os.GRND_RANDOM

このビットがセットされた場合、ランダムバイトは /dev/urandom プールの代わりに /dev/random プールから取り出されます。

バージョン 3.6 で追加.

16.2 io --- ストリームを扱うコアツール

ソースコード: Lib/io.py

16.2.1 概要

io モジュールは様々な種類の I/O を扱う Python の主要な機能を提供しています。I/O には主に 3 つの種類があります; **テキスト** *I/O*, **バイナリ** *I/O*, *raw I/O* です。これらは汎用的なカテゴリで、各カテゴリには様々なストレージが利用されます。これらのいずれかのカテゴリに属する具象オブジェクトは全て *file object* と呼ばれます。他によく使われる用語として **ストリーム** と *file-like* **オブジェクト** があります。

それぞれの具象ストリームオブジェクトは、カテゴリに応じた機能を持ちます。ストリームは読み込み専用、書き込み専用、読み書き可能のいずかになります。任意のランダムアクセス（前方、後方の任意の場所にシークする）が可能かもしれませんし、シーケンシャルアクセスしかできないかもしれません（例えばソケットやパイプなど）。

全てのストリームは、与えられたデータの型に対して厳密です。例えば、バイナリストリームの write() メソッドに対して *str* オブジェクトを渡すと *TypeError* 例外を発生させます。テキストストリームの write() メソッドに *bytes* オブジェクトを渡しても同じです。

バージョン 3.3 で変更: 以前 *IOError* を送出していた操作が *OSError* を送出するようになりました。*IOError* は今は *OSError* の別名です。

テキスト I/O

テキスト I/O は、*str* オブジェクトを受け取り、生成します。すなわち、背後にあるストレージがバイト列 (例えばファイルなど) を格納するときは常に、透過的にデータのエンコード・デコードを行ない、オプションでプラットフォーム依存の改行文字変換を行います。

テキストストリームを作る一番簡単な方法は、オプションでエンコーディングを指定して、*open()* を利用することです:

```
f = open("myfile.txt", "r", encoding="utf-8")
```

StringIO オブジェクトはインメモリーのテキストストリームです:

```
f = io.StringIO("some initial text data")
```

テキストストリームの API は *TextIOBase* のドキュメントで詳しく解説します。

バイナリ I/O

バイナリー I/O (*buffered I/O* とも呼ばれます) は *bytes-like* **オブジェクト** を受け取り *bytes* オブジェクトを生成します。エンコード、デコード、改行文字変換は一切行いません。このカテゴリのストリームは全ての非テキストデータや、テキストデータの扱いを手動で管理したい場合に利用することができます。

バイナリーストリームを生成する一番簡単な方法は、*open()* の mode 文字列に `'b'` を指定することです:

```
f = open("myfile.jpg", "rb")
```

BytesIO はインメモリーのバイナリストリームです:

```
f = io.BytesIO(b"some initial binary data: \x00\x01")
```

バイナリーストリーム API は *BufferedIOBase* のドキュメントで詳しく解説します。

他のライブラリモジュールが、別のテキスト・バイナリーストリームを生成する方法を提供しています。例えば *socket.socket.makefile()* などです。

Raw I/O

Raw I/O (*unbuffered I/O* とも呼ばれます) は、バイナリーストリームやテキストストリームの低水準の部品としてよく利用されます。ユーザーコードで直接 raw ストリームを扱うべき場面は滅多にありません。とはいえ、バッファリングを無効にしてファイルをバイナリーモードで開くことで raw ストリームを作ることができます:

```
f = open("myfile.jpg", "rb", buffering=0)
```

raw ストリーム API は *RawIOBase* のドキュメントで詳しく解説します。

16.2.2 高水準のモジュールインターフェイス

io.DEFAULT_BUFFER_SIZE

このモジュールの buffered I/O クラスで利用されるデフォルトのバッファーサイズを表す整数です。可能であれば、*open()* は file の blksize (*os.stat()* で取得される) を利用します。

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

組み込みの *open()* 関数のエイリアスです。

引数 `path`, `mode`, `flags` を指定して **監査イベント** `open` を送出します。

io.open_code(*path*)

"rb" モードでファイルを開きます。この機能はファイルの中身を実行可能なコードとして扱いたい場合にのみ使用します。

"path"は:class:*str* 型の絶対パスです。

この関数の振る舞いは、この関数より以前に `PyFile_SetOpenCodeHook()` を呼び出すことにより上書きされているかもしれません。しかし、`path` が *str* で絶対パスをあらわすことを前提として、`open_code(path)` は常に `open(path, 'rb')` と同じように振る舞うべきです。振る舞いの上書きは、ファイルに対する追加の検証や処理を目的とするべきです。

バージョン 3.8 で追加.

exception io.BlockingIOError

互換性のための、組み込みの *BlockingIOError* 例外のエイリアスです。

exception io.UnsupportedOperation

OSError と *ValueError* を継承した例外です。ストリームがサポートしていない操作を行おうとした時に送出されます。

参考:

sys 標準 IO ストリームを持っています: *sys.stdin*, *sys.stdout*, *sys.stderr*。

16.2.3 クラス階層

I/O ストリームの実装はクラス階層に分けて整理されています。まずストリームのカテゴリを分類するための**抽象基底クラス** (ABC) があり、続いて標準のストリーム実装を行う具象クラス群があります。

> **注釈:** 抽象基底クラス群は、具象ストリームクラスの実装を助けるために、いくつかのデフォルトの実装を提供しています。例えば、*BufferedIOBase* は `readinto()` と *readline()* の最適化されていない実装を提供しています。

I/O 階層の最上位には抽象基底クラスの *IOBase* があります。*IOBase* ではストリームに対して基本的なインタフェースを定義しています。しかしながら、ストリームに対する読み込みと書き込みが分離されていないことに注意してください。実装においては与えられた操作をサポートしない場合は *UnsupportedOperation* を送出することが許されています。

RawIOBase ABC は *IOBase* を拡張します。このクラスはストリームからの bytes の読み書きを扱います。*FileIO* は、*RawIOBase* を継承してマシンのファイルシステム中のファイルへのインタフェースを提供します。

BufferedIOBase ABC は生のバイトストリーム (*RawIOBase*) 上にバッファ処理を追加します。そのサブクラスの *BufferedWriter*, *BufferedReader*, *BufferedRWPair* では、それぞれ読み込み専用、書き込み専用、読み書き可能なストリームをバッファします。*BufferedRandom* ではランダムアクセスストリームに対してバッファされたインタフェースを提供します。*BytesIO* も *BufferedIOBase* のサブクラスで、インメモリのバイト列へのシンプルなストリームです。

もう一つの *IOBase* のサブクラスである *TextIOBase* ABC は、テキストを表すバイトストリームを扱い、文字列とのエンコードやデコードといった処理を行います。*TextIOWrapper* はその拡張で、バッファ付き raw ストリーム (*BufferedIOBase*) へのバッファされたテキストインタフェースです。最後に *StringIO* はテキストに対するインメモリストリームです。

引数名は規約に含まれていません。そして *open()* の引数だけがキーワード引数として用いられることが意図されています。

次のテーブルは *io* モジュールが提供する ABC の概要です:

ABC	継承元	スタブメソッド	Mixin するメソッドとプロパティ
IOBase		fileno, seek, truncate	close, closed, __enter__, __exit__, flush, isatty, __iter__, __next__, readable, readline, readlines, seekable, tell, writable, writelines
RawIOBase	*IOBase*	readinto, write	*IOBase* から継承したメソッド、read, readall
BufferedIOBase	*IOBase*	detach, read, read1, write	*IOBase* から継承したメソッド、readinto, readinto1
TextIOBase	*IOBase*	detach, read, readline, write	*IOBase* から継承したメソッド、encoding, errors, newlines

I/O 基底クラス

class io.IOBase

すべての I/O クラスの抽象基底クラスです。バイトストリームへの操作を行います。パブリックなコンストラクタはありません。

継承先のクラスが選択的にオーバライドできるように、このクラスは多くのメソッドに空の抽象実装をしています。デフォルトの実装では、読み込み、書き込み、シークができないファイルを表現します。

IOBase では read(), write() が宣言されていませんが、これはシグナチャが変化するためで、実装やクライアントはこれらのメソッドをインタフェースの一部として考えるべきです。また、実装はサポートしていない操作を呼び出されたときは *ValueError* (または *UnsupportedOperation*) を発生させるかもしれません。

ファイルへのバイナリデータの読み書きに用いられる基本型は *bytes* です。他の *bytes-like* **オブジェクト** もメソッドの引数として受け付けられます。テキスト I/O クラスは *str* データを扱います。

閉じられたストリームに対するメソッド呼び出しは (問い合わせであっても) 未定義です。この場合、実装は *ValueError* を送出することがあります。

IOBase (とそのサブクラス) はイテレータプロトコルをサポートします。*IOBase* オブジェクトをイテレートすると、ストリーム内の行が yield されます。ストリーム内の行の定義は、そのストリームが (バイト列を yield する) バイナリストリームか (文字列を yield する) テキストストリームかによって、少し異なります。下の *readline()* を参照してください。

IOBase はコンテキストマネージャでもあります。そのため with 構文をサポートします。次の例では、with 構文が終わった後で---たとえ例外が発生した場合でも、*file* は閉じられます。

```
with open('spam.txt', 'w') as file:
    file.write('Spam and eggs!')
```

IOBase は以下のデータ属性とメソッドを提供します:

close()
このストリームをフラッシュして閉じます。このメソッドはファイルが既に閉じられていた場合は特に何の効果もありません。いったんファイルが閉じられると、すべてのファイルに対する操作 (例えば読み込みや書き込み) で *ValueError* が発生します。

利便性のためにこのメソッドを複数回呼ぶことは許されています。しかし、効果があるのは最初の 1 回だけです。

closed
ストリームが閉じられていた場合 True になります。

fileno()
ストリームが保持しているファイル記述子 (整数値) が存在する場合はそれを返します。もし IO オブジェクトがファイル記述子を使っていない場合は *OSError* が発生します。

flush()
適用可能であればストリームの書き込みバッファをフラッシュします。読み出し専用や非ブロッキングストリームでは何もしません。

isatty()
ストリームが対話的であれば (つまりターミナルや tty デバイスにつながっている場合) True を返します。

readable()
ストリームが読み込める場合 True を返します。False の場合は read() は *OSError* を発生させます。

readline(*size=-1*)
ストリームから 1 行読み込んで返します。もし *size* が指定された場合、最大で *size* バイトが読み込まれます。

バイナリファイルでは行末文字は常に b'\n' となります。テキストファイルでは、認識される行末文字を選択するために *open()* に対する *newline* 引数が使われます。

readlines(*hint=-1*)
ストリームから行のリストを読み込んで返します。*hint* を指定することで、読み込む行数を制御できます。もし読み込んだすべての行のサイズ (バイト数、もしくは文字数) が *hint* の値を超えた場合、読み込みをそこで終了します。

ただし、file.readlines() を呼びださなくても for line in file: ... のように file オブジェクトを直接イテレートすることができます。

seek(*offset*, *whence=SEEK_SET*)
ストリーム位置を指定された *offset* バイトに変更します。*offset* は *whence* で指定された位置からの相対位置として解釈されます。*whence* のデフォルト値は SEEK_SET です。*whence* に指定できる値は:

- SEEK_SET または 0 -- ストリームの先頭 (デフォルト)。*offset* は 0 もしくは正の値でなければなりません。
- SEEK_CUR または 1 -- 現在のストリーム位置。*offset* は負の値も可能です。
- SEEK_END または 2 -- ストリームの末尾。*offset* は通常負の値です。

新しい絶対位置を返します。

バージョン 3.1 で追加: SEEK_* 定数.

バージョン 3.3 で追加: 一部のオペレーティングシステムは os.SEEK_HOLE や os.SEEK_DATA など、追加の値をサポートすることがあります。ファイルに対して利用できる値は、そのファイルがテキストモードで開かれたかバイナリモードで開かれたかに依存します。

seekable()
ストリームがランダムアクセスをサポートしている場合、True を返します。False の場合、*seek()*、*tell()*、*truncate()* を使用すると *OSError* を発生させます。

tell()
現在のストリーム位置を返します。

truncate(*size=None*)
ストリームのサイズを、指定された *size* バイト (または *size* が指定されていない場合、現在位置) に変更します。現在のストリーム位置は変更されません。このサイズ変更により、現在のファイルサイズを拡大または縮小させることができます。拡大の場合には、新しいファイル領域の内容はプラットホームによって異なります (ほとんどのシステムでは、追加のバイトが 0 で埋められます)。新しいファイルサイズが返されます。

バージョン 3.5 で変更: Windows で、拡大時に追加領域を 0 で埋めるようになりました。

writable()
ストリームが書き込みをサポートしている場合 True を返します。False の場合は write()、*truncate()* は *OSError* を返します。

writelines(*lines*)
ストリームに行のリストを書き込みます。行区切り文字は追加されないので、書き込む各行の行末に行区切り文字を含ませるのが一般的です。

__del__()
オブジェクトの破壊の用意をします。このメソッドはインスタンスの *close()* メソッドを呼びます。*IOBase* はこのメソッドのデフォルトの実装を提供します

class io.RawIOBase
生のバイナリ I/O への基底クラスです。*IOBase* を継承しています。パブリックコンストラクタはありません。

raw バイナリ I/O は典型的に、下にある OS デバイスや API への低水準のアクセスを提供し、高水準の基本要素へとカプセル化しようとはしません (これはこのページで後述する Buffered I/O や Text I/O に任せます)。

IOBase の属性やメソッドに加えて、*RawIOBase* は次のメソッドを提供します:

read(*size=-1*)
オブジェクトを *size* バイトまで読み込み、それを返します。簡単のため、*size* が指定されていないか -1 の場合は、EOF までの全てのバイトを返します。そうでない場合は、システムコール呼び出しが一度だけ行われます。オペレーティングシステムコールから返ってきたものが *size* バイトより少なければ、*size* バイトより少ない返り値になることがあります。

size が 0 でないのに 0 バイトが返った場合、それはファイルの終端を表します。オブジェクトがノンブロッキングモードで、1 バイトも読み込めなければ、None が返されます。

デフォルトの実装は *readall()* と *readinto()* に従います。

readall()
EOF までストリームからすべてのバイトを読み込みます。必要な場合はストリームに対して複数の呼び出しをします。

readinto(*b*)
あらかじめ確保された書き込み可能な *bytes* **類オブジェクト** *b* にバイト列を読み込み、読み込んだバイト数を返します。例えば、*b* は *bytearray* です。オブジェクトがノンブロッキングモードで、1 バイトも読み込めなければ、None が返されます。

write(*b*)
与えられた *bytes-like* **オブジェクト** *b* を生ストリームに書き込み、書き込んだバイト数を返します。これは、根底の生ストリームの性質や、特にノンブロッキングである場合に、*b* のバイト数より小さくなることがあります。生ストリームがブロックされないように設定されていて、かつ 1 バイトも即座に書き込むことができなければ、None が返されます。このメソッドから返った後で呼び出し元は *b* を解放したり変更したりするかもしれないので、実装はメソッド呼び出しの間だけ *b* にアクセスすべきです。

class io.BufferedIOBase
何らかのバッファリングをサポートするバイナリストリームの基底クラスです。*IOBase* を継承します。パブリックなコンストラクタはありません。

RawIOBase との主な違いは、メソッド *read()*、*readinto()* および *write()* は 、ことによると複数回のシステムコールを行って、(それぞれ) 要求されただけの入力を読み込もうとしたり与えられた出力の全てを消費しようとしたりする点です。

加えて、元になる生ストリームが非ブロッキングモードでかつ準備ができていない場合に、これらのメソッドは、*BlockingIOError* を送出するかもしれません。対応する *RawIOBase* バージョンと違って、None を返すことはありません。

さらに、*read()* メソッドは、*readinto()* に従うデフォルト実装を持ちません。

通常の *BufferedIOBase* 実装は *RawIOBase* 実装を継承せずに、*BufferedWriter* と *BufferedReader* がするようにこれをラップすべきです。

BufferedIOBase は *IOBase* からのメソッドと属性に加えて、以下のメソッドを提供もしくはオーバーライドします:

raw

BufferedIOBase が扱う根底の生ストリーム (*RawIOBase* インスタンス) を返します。これは *BufferedIOBase* API には含まれず、よって実装に含まれないことがあります。

detach()

根底の生ストリームをバッファから分離して返します。

生ストリームが取り外された後、バッファは使用不能状態になります。

バッファには、*BytesIO* など、このメソッドで返される単体のストリームという概念を持たないものがあります。これらは *UnsupportedOperation* を送出します。

バージョン 3.1 で追加.

read(*size=-1*)

最大で *size* バイト読み込んで返します。引数が省略されるか、None か、または負の値であった場合、データは EOF に到達するまで読み込まれます。ストリームが既に EOF に到達していた場合は空の *bytes* オブジェクトが返されます。

引数が正で、元になる生ストリームが対話的でなければ、必要なバイト数を満たすように複数回の生 read が発行されるかもしれません (先に EOF に到達しない限りは)。対話的な場合は、最大で一回の raw read しか発行されず、短い結果でも EOF に達したことを意味しません。

元になる生ストリームがノンブロッキングモードで、呼び出された時点でデータを持っていなければ、*BlockingIOError* が送出されます。

read1(*[size]*)

根底の raw ストリームの *read()* (または *readinto()*) メソッドを高々 1 回呼び出し、最大で *size* バイト読み込み、返します。これは、*BufferedIOBase* オブジェクトの上に独自のバッファリングを実装するときに便利です。

size に “-1“ (デフォルト値) を指定すると任意バイト長を返します (EOF に到達していなければ返されるバイト数は 0 より大きくなります)

readinto(*b*)

あらかじめ確保された書き込み可能な *bytes* **類オブジェクト** *b* にバイト列を読み込み、読み込んだバイト数を返します。例えば、*b* は *bytearray* です。

read() と同様に、下層の raw ストリームが対話的でない限り、複数の読み込みは下層の raw ストリームに与えられるかもしれません。

元になる生ストリームがノンブロッキングモードで、呼び出された時点でデータを持っていなければ、*BlockingIOError* が送出されます。

readinto1(*b*)

根底の raw ストリームの *read()* (または *readinto()*) メソッドを高々 1 回呼び出し、あらかじめ確保された書き込み可能な *bytes-like* **オブジェクト** *b* にバイト列を読み込みます。読み込んだバイト数を返します。

元になる生ストリームがノンブロッキングモードで、呼び出された時点でデータを持っていなけれ

ば、*BlockingIOError* が送出されます。

バージョン 3.5 で追加.

write(*b*)

与えられた *bytes-like* **オブジェクト** *b* を書き込み、書き込んだバイト数を返します (これは常に *b* のバイト数と等しくなります。なぜなら、もし書き込みに失敗した場合は *OSError* が発生するからです)。実際の実装に依存して、これらのバイト列は根底のストリームに即座に書き込まれることもあれば、パフォーマンスやレイテンシの関係でバッファに保持されることもあります。

ノンブロッキングモードであるとき、バッファが満杯で根底の生ストリームが書き込み時点でさらなるデータを受け付けられない場合 *BlockingIOError* が送出されます。

このメソッドが戻った後で、呼び出し元は *b* を解放、または変更するかもしれないので、実装はメソッド呼び出しの間だけ *b* にアクセスすべきです。

生ファイル I/O

class io.FileIO(*name*, *mode='r'*, *closefd=True*, *opener=None*)

FileIO はバイトデータを含む OS レベルのファイルを表します。*RawIOBase* インタフェースを (したがって *IOBase* インタフェースも) 実装しています。

name は、次の 2 つのいずれかです。

- 開くファイルへのパスを表す文字列または *bytes* オブジェクト。この場合、closefd は True (デフォルト) でなければなりません。True でない場合、エラーが送出されます。
- 結果の *FileIO* オブジェクトがアクセスを与える、既存の OS レベルファイル記述子の数を表す整数。FileIO オブジェクトが閉じられると、*closefd* が False に設定されていない場合、この fd も閉じられます。

mode は 読み込み（デフォルト)、書き込み、排他的作成、追記に対し 'r' 、'w' 、'x' 、'a' です。ファイルは書き込みや追記で開かれたときに存在しない場合作成されます。書き込みのときにファイルの内容は破棄されます。作成時に既に存在する場合は *FileExistsError* が送出されます。作成のためにファイルを開くのは暗黙的に書き込みなので、このモードは 'w' と同じように振る舞います。読み込みと書き込みを同時に許可するにはモードに '+' を加えてください。

このクラスの read() (正の引数で呼び出されたとき), readinto() および write() メソッドは、単にシステムコールを一度呼び出します。

呼び出し可能オブジェクトを *opener* として与えることで、カスタムのオープナーが使えます。そしてファイルオブジェクトの基底のファイルディスクリプタは、*opener* を (*name*, *flags*) で呼び出して得られます。*opener* は開いたファイルディスクリプタを返さなければなりません。(*os.open* を *opener* として渡すと、None を渡したのと同様の機能になります。)

新たに作成されたファイルは **継承不可** です。

opener 引数を使う例については *open()* 組み込み関数を参照してください。

バージョン 3.3 で変更: *opener* 引数が追加されました。'x' モードが追加されました。

バージョン 3.4 で変更: ファイルが継承不可になりました。

IOBase と *RawIOBase* の属性やメソッドに加え、*FileIO* は以下のデータ属性を提供します:

mode
コンストラクタに渡されたモードです。

name
ファイル名。コンストラクタに名前が渡されなかったときはファイル記述子になります。

バッファ付きストリーム

バッファ付き I/O ストリームは、I/O デバイスに生 I/O より高レベルなインタフェースを提供します。

class io.BytesIO(*[initial_bytes]*)
インメモリの bytes バッファを利用したストリームの実装です。*BufferedIOBase* を継承します。バッファは *close()* メソッドが呼び出された際に破棄されます。

省略可能な引数 *initial_bytes* は、初期データを含んだ *bytes-like* **オブジェクト** です。

BytesIO は *BufferedIOBase* または *IOBase* からのメソッドに加えて、以下のメソッドを提供もしくはオーバーライドします:

getbuffer()
バッファの内容をコピーすることなく、その内容の上に、読み込み及び書き込みが可能なビューを返します。また、このビューを変更すると、バッファの内容は透過的に更新されます:

```
>>> b = io.BytesIO(b"abcdef")
>>> view = b.getbuffer()
>>> view[2:4] = b"56"
>>> b.getvalue()
b'ab56ef'
```

注釈: ビューが存在する限り、*BytesIO* オブジェクトはリサイズやクローズされません。

バージョン 3.2 で追加.

getvalue()
バッファの全内容を含む *bytes* を返します。

read1(*[size]*)
BytesIO においては、このメソッドは *read()* と同じです。

バージョン 3.7 で変更: *size* 引数が任意になりました。

readinto1(*b*)
BytesIO においては、このメソッドは *readinto()* と同じです。

バージョン 3.5 で追加.

class io.BufferedReader(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

読み込み可能でシーケンシャルな *RawIOBase* オブジェクトへの高水準のアクセスを提供するバッファです。*BufferedIOBase* を継承します。このオブジェクトからデータを読み込むとき、より大きい量のデータが根底の raw ストリームから要求され、内部バッファに保存される場合があります。バッファされたデータは、その次の読み込み時に直接返されます。

このコンストラクタは与えられた *raw* ストリームと *buffer_size* に対し *BufferedReader* を生成します。*buffer_size* が省略された場合、代わりに *DEFAULT_BUFFER_SIZE* が使われます。

BufferedReader は *BufferedIOBase* または *IOBase* からのメソッドに加えて、以下のメソッドを提供もしくはオーバーライドします:

peek([*size*])

位置を進めずにストリームからバイト列を返します。これを果たすために生ストリームに対して行われる read は高々一度だけです。返されるバイト数は、要求より少ないかもしれませんし、多いかもしれません。

read([*size*])

size バイトを読み込んで返します。*size* が与えられないか負の値ならば、EOF まで、または非ブロッキングモード中で read 呼び出しがブロックされるまでを返します。

read1([*size*])

raw ストリームに対しただ一度の呼び出しで最大 *size* バイトを読み込んで返します。少なくとも 1 バイトがバッファされていれば、バッファされているバイト列だけが返されます。それ以外の場合は raw ストリームの読み込みが一回呼び出されます。

バージョン 3.7 で変更: *size* 引数が任意になりました。

class io.BufferedWriter(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

書き込み可能でシーケンシャルな *RawIOBase* オブジェクトへの、高レベルなアクセスを提供するバッファです。*BufferedIOBase* を継承します。このオブジェクトに書き込むとき、データは通常内部バッファに保持されます。このバッファは、以下のような種々の状況で根底の *RawIOBase* オブジェクトに書き込まれます。

- 保留中の全データに対してバッファが足りなくなったとき;
- *flush()* が呼び出されたとき;
- seek() が (*BufferedRandom* オブジェクトに対して) 呼び出されたとき;
- *BufferedWriter* オブジェクトが閉じられたり破棄されたりしたとき。

このコンストラクタは与えられた書き込み可能な *raw* ストリームに対し *BufferedWriter* を生成します。*buffer_size* が省略された場合、*DEFAULT_BUFFER_SIZE* がデフォルトになります。

BufferedWriter は *BufferedIOBase* または *IOBase* からのメソッドに加えて、以下のメソッドを提供もしくはオーバーライドします:

flush()

バッファに保持されたバイト列を生ストリームに強制的に流し込みます。生ストリームがブロック

した場合 *BlockingIOError* が送出されます。

write(*b*)

bytes-like **オブジェクト** *b* を書き込み、書き込んだバイト数を返します。ノンブロッキング時、バッファが書き込まれるべきなのに生ストリームがブロックした場合 *BlockingIOError* が送出されます。

class io.BufferedRandom(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

A buffered interface to random access streams. It inherits *BufferedReader* and *BufferedWriter*.

このコンストラクタは第一引数として与えられるシーク可能な生ストリームに対し、リーダーおよびライターを作成します。*buffer_size* が省略された場合、*DEFAULT_BUFFER_SIZE* がデフォルトになります。

BufferedRandom は *BufferedReader* と *BufferedWriter* ができることは何でもできる能力があります。さらに `seek()` と `tell()` が実装されていることが保証されています。

class io.BufferedRWPair(*reader*, *writer*, *buffer_size=DEFAULT_BUFFER_SIZE*)

2 つの単方向 *RawIOBase* オブジェクト -- 一つは読み込み可能、他方が書き込み可能 -- を組み合わせてバッファ付きの双方向 I/O オブジェクトにしたものです。*BufferedIOBase* を継承しています。

reader と *writer* はそれぞれ読み込み可能、書き込み可能な *RawIOBase* オブジェクトです。*buffer_size* が省略された場合 *DEFAULT_BUFFER_SIZE* がデフォルトになります。

BufferedRWPair は、*UnsupportedOperation* を送出する *detach()* を除く、*BufferedIOBase* の全てのメソッドを実装します。

> **警告:** *BufferedRWPair* は下層の生ストリームのアクセスを同期しようとはしません。同じオブジェクトをリーダとライタとして渡してはいけません。その場合は代わりに *BufferedRandom* を使用してください。

テキスト I/O

class io.TextIOBase

テキストストリームの基底クラスです。このクラスはストリーム I/O への、文字と行に基づいたインタフェースを提供します。*IOBase* を継承します。パブリックなコンストラクタはありません。

IOBase から継承した属性とメソッドに加えて、*TextIOBase* は以下のデータ属性とメソッドを提供しています:

encoding

エンコーディング名で、ストリームのバイト列を文字列にデコードするとき、また文字列をバイト列にエンコードするときに使われます。

errors

このエンコーダやデコーダのエラー設定です。

newlines
文字列、文字列のタプル、または None で、改行がどのように読み換えられるかを指定します。実装や内部コンストラクタのフラグに依って、これは利用できないことがあります。

buffer
TextIOBase が扱う根底のバイナリバッファ (*BufferedIOBase* インスタンス) です。これは *TextIOBase* API には含まれず、よって実装に含まれない場合があります。

detach()
根底のバイナリバッファを *TextIOBase* から分離して返します。

根底のバッファが取り外された後、*TextIOBase* は使用不能状態になります。

TextIOBase 実装には、*StringIO* など、根底のバッファという概念を持たないものがあります。これらを呼び出すと *UnsupportedOperation* を送出します。

バージョン 3.1 で追加.

read(*size=-1*)
最大 *size* 文字をストリームから読み込み、一つの *str* にして返します。*size* が負の値または None ならば、EOF まで読みます。

readline(*size=-1*)
改行または EOF まで読み込み、一つの str を返します。ストリームが既に EOF に到達している場合、空文字列が返されます。

size が指定された場合、最大 *size* 文字が読み込まれます。

seek(*offset*, *whence=SEEK_SET*)
指定された *offset* にストリーム位置を変更します。挙動は *whence* 引数によります。*whence* のデフォルト値は SEEK_SET です。:

- SEEK_SET または 0: ストリームの先頭からシークします (デフォルト)。*offset* は *TextIOBase.tell()* が返す数か 0 のどちらかでなければなりません。それ以外の *offset* 値は未定義の挙動を起こします。
- SEEK_CUR または 1: 現在の位置に " シークします"。*offset* は 0 でなければなりません。つまり何もしません (他の値はサポートされていません)。
- SEEK_END または 2: ストリーム終端へシークします。*offset* は 0 でなければなりません (他の値はサポートされていません).

新しい絶対位置を、不透明な数値で返します。

バージョン 3.1 で追加: SEEK_* 定数.

tell()
ストリームの現在位置を不透明な数値で返します。この値は根底のバイナリストレージ内でのバイト数を表すとは限りません。

write(*s*)

文字列 *s* をストリームに書き出し、書き出された文字数を返します。

class io.TextIOWrapper(*buffer*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=False*, *write_through=False*)

BufferedIOBase バイナリストリーム上のバッファ付きテキストストリーム。*TextIOBase* を継承します。

encoding はストリームがエンコードやデコードされるエンコード名です。デフォルトは *locale.getpreferredencoding(False)* です。

errors はオプションの文字列で、エンコードやデコードの際のエラーをどのように扱うかを指定します。エンコードエラーがあったときに *ValueError* 例外を送出させるには `'strict'` を渡します (デフォルトの `None` でも同じです)。エラーを無視させるには `'ignore'` を渡します。(エンコーディングエラーを無視するとデータを喪失する可能性があることに注意してください。) `'replace'` は不正な形式の文字の代わりにマーカ (たとえば `'?'`) を挿入させます。`'backslashreplace'` を指定すると、不正な形式のデータをバックスラッシュ付きのエスケープシーケンスに置換します。書き込み時には `'xmlcharrefreplace'` (適切な XML 文字参照に置換) や `'namereplace'` (`\N{...}` エスケープシーケンスに置換) も使えます。他にも *codecs.register_error()* で登録されたエラー処理名が有効です。

newline は行末をどのように処理するかを制御します 。`None`, `''`, `'\n'`, `'\r'`, `'\r\n'` のいずれかです。これは以下のように動作します:

- ストリームからの入力を読み込んでいる時、*newline* が `None` の場合、*universal newlines* モードが有効になります。入力中の行は `'\n'`、`'\r'`、または `'\r\n'` で終わり、呼び出し元に返される前に `'\n'` に変換されます。`''` の場合、ユニバーサル改行モードは有効になりますが、行末は変換されずに呼び出し元に返されます。その他の合法な値の場合、入力行は与えられた文字列でのみ終わり、行末は変換されずに呼び出し元に返されます。
- ストリームへの出力の書き込み時、*newline* が `None` の場合、全ての `'\n'` 文字はシステムのデフォルトの行セパレータ *os.linesep* に変換されます。*newline* が `''` または `'\n'` の場合は変換されません。*newline* がその他の正当な値の場合、全ての `'\n'` 文字は与えられた文字列に変換されます。

line_buffering が `True` の場合、write への呼び出しが改行文字もしくはキャリッジリターンを含んでいれば、暗黙的に `flush()` が呼び出されます。

write_through が `True` の場合、`write()` の呼び出しはバッファされないことが保証されます。*TextIOWrapper* オブジェクトに書かれた全てのデータは直ちに下層のバイナリ *buffer* に処理されます。

バージョン 3.3 で変更: *write_through* 引数が追加されました。

バージョン 3.3 で変更: *encoding* の規定値が `locale.getpreferredencoding()` から `locale.getpreferredencoding(False)` になりました。*locale.setlocale()* を用いてロケールのエンコーディングを一時的に変更してはいけません。ユーザが望むエンコーディングではなく現在のロケールのエンコーディングを使用してください。

TextIOBase とその親クラスのメンバーに加えて、*TextIOWrapper* は以下のメンバーを提供しています:

line_buffering
行バッファリングが有効かどうか。

write_through
書き込みが、根柢のバイナリバッファに即座に渡されるかどうか。

バージョン 3.7 で追加.

reconfigure(**[, encoding][, errors][, newline][, line_buffering][, write_through]*)
このテキストストリームを *encoding*, *errors*, *newline*, *line_buffering* と *write_through* を新しい設定として再設定します。

encoding が指定されており、*errors* が指定されていないときに、`errors='strict'` が使われている場合を除き、指定されなかったパラメータは現在の設定が保持されます。

ストリームからすでにデータが読み出されていた場合、encoding と newline は変更できません。一方で、書き込み後に encoding を変更することはできます。

このメソッドは、新しい設定を適用するまえにストリームをフラッシュします。

バージョン 3.7 で追加.

class io.StringIO(*initial_value=''*, *newline='\n'*)
テキスト IO のためのインメモリストリーム。テキストバッファは *close()* メソッドが呼び出された際に破棄されます。

バッファの初期値を *initial_value* で与えることが出来ます。改行変換を有効にすると、改行コードは *write()* によってエンコードされます。ストリームはバッファの開始位置に配置されます。

newline 引数は *TextIOWrapper* のものと同じように働きます。デフォルトでは `\n` 文字だけを行末とみなし、また、改行の変換は行いません。*newline* に `None` をセットすると改行コードを全てのプラットフォームで `\n` で書き込みますが、読み込み時にはそれでもユニバーサル改行としてのデコードは実行されます。

TextIOBase およびその親クラスから継承したメソッドに加えて *StringIO* は以下のメソッドを提供しています:

getvalue()
バッファの全内容を含む `str` を返します。改行コードのデコードは *read()* によって行われますが、これによるストリーム位置の変更は起こりません。

使用例:

```
import io

output = io.StringIO()
output.write('First line.\n')
print('Second line.', file=output)
```

(次のページに続く)

(前のページからの続き)

```
# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

class io.**IncrementalNewlineDecoder**

改行を *universal newlines* モードにデコードするヘルパーコーデックです。*codecs.IncrementalDecoder* を継承しています。

16.2.4 性能

このセクションでは与えられた具体的な I/O 実装の性能について議論します。

バイナリ I/O

バッファ付き I/O は、ユーザが 1 バイトだけ要求した場合でさえ、データを大きな塊でのみ読み書きします。これにより、オペレーティングシステムのバッファ無し I/O ルーチンを呼び出して実行する非効率性はすべて隠されます。その成果は、OS と、実行される I/O の種類によって異なります。例えば、Linux のような現行の OS では、バッファ無しディスク I/O がバッファ付き I/O と同じくらい早いことがあります。しかし、どのプラットフォームとデバイスにおいても、バッファ付き I/O は最低でも予測可能なパフォーマンスを提供します。ですから、バイナリデータに対しては、バッファ無し I/O を使用するより、バッファ付きの I/O を使用するほうが望ましい場合がほとんどです。

テキスト I/O

(ファイルなどの) バイナリストレージ上のテキスト I/O は、同じストレージ上のバイナリ I/O より非常に遅いです。なぜならこれには、文字コーデックを使った Unicode とバイナリデータ間の変換を必要とするからです。これは大量のテキストデータ、例えば大きなログファイルを扱うときに顕著に成り得ます。同様に、`TextIOWrapper.tell()` や `TextIOWrapper.seek()` はどちらも、使われている復元アルゴリズムのために遅くなります。

しかし *StringIO* は、ネイティブなインメモリ Unicode コンテナで、*BytesIO* と同程度の速度を示します。

マルチスレッディング

(Unix における `read(2)` のような) オペレーティングシステムコールの、それがラッピングするものがスレッドセーフであるような範囲内では、*FileIO* オブジェクトもまた、スレッドセーフです。

バイナリバッファ付きオブジェクト (*BufferedReader*, *BufferedWriter*, *BufferedRandom* および *BufferedRWPair* のインスタンス) は、その内部構造をロックを使って保護します。このため、これらを複数のスレッドから同時に呼び出しても安全です。

TextIOWrapper オブジェクトはスレッドセーフではありません。

リエントラント性

バイナリバッファ付きオブジェクト (*BufferedReader*, *BufferedWriter*, *BufferedRandom* および *BufferedRWPair* のインスタンス) は、リエントラント (再入可能) ではありません。リエントラントな呼び出しは普通の状況では起こりませんが、I/O を *signal* ハンドラで行なっているときに起こりえます。スレッドが、すでにアクセスしているバッファ付きオブジェクトに再び入ろうとすると *RuntimeError* が送出されます。これは、バッファ付きオブジェクトに複数のスレッドから入ることを禁止するわけではありません。

open() 関数は *TextIOWrapper* 内部のバッファ付きオブジェクトをラップするため、テキストファイルにも暗黙に拡張されます。これは、標準ストリームを含むので、組み込み関数 *print()* にも同様に影響します。

16.3 `time` --- 時刻データへのアクセスと変換

このモジュールでは、時刻に関するさまざまな関数を提供します。関連した機能について、*datetime*, *calendar* モジュールも参照してください。

このモジュールは常に利用可能ですが、すべての関数がすべてのプラットフォームで利用可能なわけではありません。このモジュールで定義されているほとんどの関数は、プラットフォーム上の同名の C ライブラリ関数を呼び出します。これらの関数に対する意味付けはプラットフォーム間で異なるため、プラットフォーム提供のドキュメントを読んでおくと便利でしょう。

まずいくつかの用語の説明と慣習について整理します。

- **エポック** (*epoch*) は時刻の起点のことで、これはプラットフォーム依存です。Unix では、エポックは (UTC で) 1970 年 1 月 1 日 0 時 0 分 0 秒です。与えられたプラットフォームでエポックが何なのかを知るには、`time.gmtime(0)` の値を見てください。
- **エポック秒** (*seconds since the epoch*) は、エポックからの総経過秒数を示していますが、たいていはうるう秒 (leap seconds) は含まれていません。全ての POSIX 互換のプラットフォームで、うるう秒はこの総秒数には含まれません。
- このモジュールの中の関数は、エポック以前あるいは遠い未来の日付や時刻を扱うことができません。将来カットオフ（関数が正しく日付や時刻を扱えなくなる）が起きる時点は、C ライブラリによって決まります。32-bit システムではカットオフは通常 2038 年です。

- 関数 *strptime()* は %y 書式コードが与えられた時に 2 桁の年表記を解析できます。2 桁の年を解析する場合、それらは POSIX および ISO C 標準に従って変換されます: 69-99 の西暦年は 1969-1999 となり、0 – 68 の西暦年は 2000 – 2068 になります。

- UTC は協定世界時 (Coordinated Universal Time) のことです (以前はグリニッジ標準時または GMT として知られていました)。UTC の頭文字の並びは誤りではなく、英仏の妥協によるものです。

- DST は夏時間 (Daylight Saving Time) のことで、一年のうちの一定期間に 1 時間タイムゾーンを修正することです。DST のルールは不可思議で (地域ごとに法律で定められています)、年ごとに変わることもあります。C ライブラリはローカルルールを記したテーブルを持っており (柔軟に対応するため、たいていはシステムファイルから読み込まれます)、この点に関しては唯一の真実の知識の源です。

- 多くの現時刻を返す関数 (real-time functions) の精度は、値や引数を表現するために使う単位から想像されるよりも低いかも知れません。例えば、ほとんどの Unix システムにおいて、クロックの 1 ティックの精度は 50 から 100 分の 1 秒に過ぎません。

- 一方、*time()* および *sleep()* は Unix の同等の関数よりましな精度を持っています。時刻は浮動小数点数で表され、*time()* は可能なかぎり最も正確な時刻を (Unix の `gettimeofday()` があればそれを使って) 返します。また *sleep()* にはゼロでない端数を与えることができます (Unix の `select()` があれば、それを使って実装しています)。

- *gmtime()*, *localtime()*, *strptime()* が返す時刻値、および *asctime()*, *mktime()*, *strftime()* がとる時刻値は 9 個の整数からなるシーケンスです。*gmtime()*, *localtime()*, *strptime()* の戻り値は個々の値を属性名で取得することもできます。

  これらのオブジェクトについての解説は *struct_time* を参照してください。

  バージョン 3.3 で変更: *struct_time* オブジェクトは、プラットフォームが、対応する `struct tm` メンバーをサポートしている場合、`tm_gmtoff` および `tm_zone` 属性が拡張されるようになりました。

  バージョン 3.6 で変更: *struct_time* の属性 `tm_gmtoff` および `tm_zone` が全てのプラットフォームで利用できるようになりました。

- 時間の表現を変換するには、以下の関数を利用してください:

対象	変換先	関数
エポックからの秒数	UTC の *struct_time*	*gmtime()*
エポックからの秒数	ローカル時間の *struct_time*	*localtime()*
UTC の *struct_time*	エポックからの秒数	*calendar.timegm()*
ローカル時間の *struct_time*	エポックからの秒数	*mktime()*

16.3.1 関数

time.asctime([t])

Convert a tuple or *struct_time* representing a time as returned by *gmtime()* or *localtime()* to a string of the following form: `'Sun Jun 20 23:21:05 1993'`. The day field is two characters long and is space padded if the day is a single digit, e.g.: `'Wed Jun 9 04:26:40 1993'`.

If *t* is not provided, the current time as returned by *localtime()* is used. Locale information is not used by *asctime()*.

注釈: 同名の C の関数と違って、*asctime()* は末尾に改行文字を加えません。

---

time.pthread_getcpuclockid(*thread_id*)

Return the *clk_id* of the thread-specific CPU-time clock for the specified *thread_id*.

Use *threading.get_ident()* or the *ident* attribute of *threading.Thread* objects to get a suitable value for *thread_id*.

警告: Passing an invalid or expired *thread_id* may result in undefined behavior, such as segmentation fault.

利用可能な環境: Unix (更なる情報については *pthread_getcpuclockid(3)* の man を参照してください)。

バージョン 3.7 で追加.

time.clock_getres(*clk_id*)

指定された *clk_id* クロックの分解能 (精度) を返します。*clk_id* として受け付けられる値の一覧は *Clock ID Constants* を参照してください。

利用可能な環境: Unix。

バージョン 3.3 で追加.

time.clock_gettime(*clk_id*) → float

指定された *clk_id* クロックの時刻を返します。*clk_id* として受け付けられる値の一覧は *Clock ID Constants* を参照してください。

利用可能な環境: Unix。

バージョン 3.3 で追加.

time.clock_gettime_ns(*clk_id*) → int

clock_gettime() に似ていますが、ナノ秒単位の時刻を返します。

利用可能な環境: Unix。

バージョン 3.7 で追加.

time.**clock_settime**(*clk_id*, *time: float*)

指定された *clk_id* クロックの時刻を設定します。現在、*CLOCK_REALTIME* は *clk_id* が受け付ける唯一の値です。

利用可能な環境: Unix。

バージョン 3.3 で追加.

time.**clock_settime_ns**(*clk_id*, *time: int*)

clock_settime() に似ていますが、ナノ秒単位の時刻を設定します。

利用可能な環境: Unix。

バージョン 3.7 で追加.

time.**ctime**([*secs*])

Convert a time expressed in seconds since the epoch to a string of a form: `'Sun Jun 20 23:21:05 1993'` representing local time. The day field is two characters long and is space padded if the day is a single digit, e.g.: `'Wed Jun 9 04:26:40 1993'`.

secs を指定しないか *None* を指定した場合、*time()* が返す値を現在の時刻として使用します。`ctime(secs)` は `asctime(localtime(secs))` と等価です。ローカル情報は *ctime()* には使用されません。

time.**get_clock_info**(*name*)

指定されたクロックの情報を名前空間オブジェクトとして取得します。サポートされているクロック名およびそれらの値を取得する関数は以下の通りです:

- `'monotonic'`: *time.monotonic()*
- `'perf_counter'`: *time.perf_counter()*
- `'process_time'`: *time.process_time()*
- `'thread_time'`: *time.thread_time()*
- `'time'`: *time.time()*

結果は以下の属性をもちます:

- *adjustable*: 自動 (NTP デーモンによるなど) またはシステム管理者による手動で変更できる場合は `True`、それ以外の場合は `False` になります。
- *implementation*: クロック値を取得するために内部で使用している C 関数の名前です。使える値については *Clock ID Constants* を参照してください。
- *monotonic*: クロック値が後戻りすることがない場合 `True` が、そうでない場合は `False` になります。
- *resolution*: クロックの分解能を秒 (*float*) で表します。

バージョン 3.3 で追加.

time.**gmtime**([*secs*])

エポックからの経過時間で表現された時刻を、UTC で *struct_time* に変換します。このとき dst フラグは常にゼロとして扱われます。*secs* を指定しないか *None* を指定した場合、*time()* が返す値を現在の時刻として使用します。秒の端数は無視されます。*struct_time* オブジェクトについては前述の説明を参照してください。*calendar.timegm()* はこの関数と逆の変換を行います。

time.**localtime**([*secs*])

gmtime() に似ていますが、ローカル時間に変換します。*secs* を指定しないか *None* を指定した場合、*time()* が返す値を現在の時刻として使用します。DST が適用されている場合は dst フラグには 1 が設定されます。

localtime() may raise *OverflowError*, if the timestamp is outside the range of values supported by the platform C `localtime()` or `gmtime()` functions, and *OSError* on `localtime()` or `gmtime()` failure. It's common for this to be restricted to years between 1970 and 2038.

time.**mktime**(*t*)

localtime() の逆を行う関数です。引数は *struct_time* か 9 個の要素すべての値を持つ完全なタプル (dst フラグも必要です; 時刻に DST が適用されるか不明の場合は `-1` を使用してください) で、UTC ではなく **ローカル** 時間を指定します。戻り値は *time()* との互換性のために浮動小数点数になります。入力した値を正しい時刻として表現できない場合、例外 *OverflowError* または *ValueError* が送出されます (どちらが送出されるかは、無効な値を受け取ったのが Python と下層の C ライブラリのどちらなのかによって決まります)。この関数で時刻を生成できる最も古い日付はプラットフォームに依存します。

time.**monotonic**() → float

モノトニッククロック、すなわち後戻りしないクロックの値を (小数秒で) 返します。このクロックはシステムクロックの更新の影響を受けません。戻り値の基準点は定義されていないので、二回の呼び出しの結果の差だけが有効です。

バージョン 3.3 で追加.

バージョン 3.5 で変更: この関数は、常に利用でき、常にシステム全域で使えるようになりました。

time.**monotonic_ns**() → int

monotonic() に似ていますが、ナノ秒単位の時刻を返します。

バージョン 3.7 で追加.

time.**perf_counter**() → float

パフォーマンスカウンター、すなわち短い時間を計測するための可能な限り高い分解能を持つクロックの値を (小数秒で) 返します。これはスリープ中の経過時間を含み、システムワイドです。戻り値の基準点は定義されていないので、二回の呼び出しの結果の差だけが有効です。

バージョン 3.3 で追加.

time.**perf_counter_ns**() → int

perf_counter() に似ていますが、ナノ秒単位の時刻を返します。

バージョン 3.7 で追加.

time.**process_time**() → float

現在のプロセスのシステムおよびユーザー CPU 時間の値を (小数秒で) 返します。これはスリープ中の経過時間を含みません。これは定義上プロセスワイドです。戻り値の基準点は定義されていないので、二回の呼び出しの結果の差だけが有効です。

バージョン 3.3 で追加.

time.**process_time_ns**() → int

process_time() に似ていますが、ナノ秒単位の時刻を返します。

バージョン 3.7 で追加.

time.**sleep**(*secs*)

与えられた秒数の間、呼び出したスレッドの実行を停止します。より精度の高い実行停止時間を指定するために、引数は浮動小数点にしてもかまいません。何らかのシステムシグナルがキャッチされた場合、それに続いてシグナル処理ルーチンが実行され、*sleep()* を停止します。従って実際の実行停止時間は要求した時間よりも短くなるかもしれません。また、システムが他の処理をスケジュールするために、実行停止時間が要求した時間よりも多少長い時間になることもあります。

バージョン 3.5 で変更: スリープがシグナルに中断されてもシグナルハンドラが例外を送出しない限り、少なくとも *secs* だけスリープするようになりました (論拠については **PEP 475** を参照してください)。

time.**strftime**(*format*[, *t*])

gmtime() や *localtime()* が返す時刻値タプルまたは *struct_time* を、*format* で指定した文字列形式に変換します。*t* が与えられていない場合、*localtime()* が返す値を現在の時刻として使用します。*format* は文字列でなくてはなりません。*t* のいずれかのフィールドが許容範囲外の数値であった場合、*ValueError* を送出します。

0 は時刻タプル内のいずれの位置の引数にも使用できます; それが一般に不正な値であれば、正しい値に強制的に置き換えられます。

format 文字列には以下のディレクティブ (指示語) を埋め込むことができます。これらはフィールド長や精度のオプションを付けずに表され、*strftime()* の結果の対応する文字列に置き換えられます:

ディレクティブ	意味	注釈
%a	ロケールの短縮された曜日名になります。	
%A	ロケールの曜日名になります。	
%b	ロケールの短縮された月名になります。	
%B	ロケールの月名になります。	
%c	ロケールの日時を適切な形式で表します。	
%d	月中の日にちの 10 進表記になります [01,31]。	
%H	時 (24 時間表記) の 10 進表記になります [00,23]。	
%I	時 (12 時間表記) の 10 進表記になります [01,12]。	
%j	年中の日にちの 10 進表記になります [001,366]。	
%m	月の 10 進表記になります [01,12]。	
%M	分の 10 進表記になります [00,59]。	
%p	ロケールの AM もしくは PM と等価な文字列になります。	(1)
%S	秒の 10 進表記になります [00,61]。	(2)
%U	年の初めから何週目か (日曜を週の始まりとします) を表す 10 進数になります [00,53]。年が明けてから最初の日曜日までのすべての曜日は 0 週目に属すると見なされます。	(3)
%w	曜日の 10 進表記になります [0 (日曜日),6]。	
%W	年の初めから何週目か (月曜を週の始まりとします) を表す 10 進数になります [00,53]。年が明けてから最初の月曜日までの全ての曜日は 0 週目に属すると見なされます。	(3)
%x	ロケールの日付を適切な形式で表します。	
%X	ロケールの時間を適切な形式で表します。	
%y	西暦の下 2 桁の 10 進表記になります [00,99]。	
%Y	西暦 (4 桁) の 10 進表記を表します。	
%z	タイムゾーンと UTC/GMT との時差を表す正または負の時間を +HHMM、-HHMM で表します。H は時間の、M は分の 10 進表記になります [-23:59, +23:59]。	
%Z	タイムゾーンの名前を表します (タイムゾーンがない場合には空文字列)。	
%%	文字 '%' を表します。	

注釈:

(1) *strptime()* 関数で使う場合、%p ディレクティブが出力結果の時刻フィールドに影響を及ぼすのは、時刻を解釈するために %I を使ったときのみです。

(2) 値の幅は実際に 0 から 61 です; 60 は うるう秒<leap seconds> を表し、61 は歴史的理由によりサポートされています。

(3) *strptime()* 関数で使う場合、%U および %W を計算に使うのは曜日と年を指定したときだけです。

以下に **RFC 2822** インターネット電子メール標準で定義されている日付表現と互換の書式の例を示

します。[*1]

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

一部のプラットフォームではさらにいくつかのディレクティブがサポートされていますが、標準 ANSI C で意味のある値はここで列挙したものだけです。あなたのプラットフォームでサポートされている書式コードの全一覧については、*strftime(3)* のドキュメントを参照してください。

一部のプラットフォームでは、フィールドの幅や精度を指定するオプションがディレクティブの先頭の文字 '%' の直後に付けられるようになっていました; この機能も移植性はありません。フィールドの幅は通常 2 ですが、%j は例外で 3 です。

time.strptime(*string*[, *format*])

時刻を表現する文字列を書式に従って解釈します。返される値は *gmtime()* や *localtime()* が返すような *struct_time* です。

format パラメーターは *strftime()* で使うものと同じディレクティブを使います; このパラメーターの値はデフォルトでは "%a %b %d %H:%M:%S %Y" で、*ctime()* が返すフォーマットに一致します。*string* が *format* に従って解釈できなかった場合、例外 *ValueError* が送出されます。解析しようとする *string* が解析後に余分なデータを持っていた場合、*ValueError* が送出されます。欠落したデータについて、適切な値を推測できない場合はデフォルトの値で埋められ、その値は (1900, 1, 1, 0, 0, 0, 0, 1, -1) です。*string* も *format* も文字列でなければなりません。

例えば:

```
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")
time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0,
                 tm_sec=0, tm_wday=3, tm_yday=335, tm_isdst=-1)
```

%Z ディレクティブへのサポートは tzname に収められている値と daylight が真かどうかで決められます。このため、常に既知の (かつ夏時間でないと考えられている) UTC や GMT を認識する時以外はプラットフォーム固有の動作になります。

ドキュメント内で説明されているディレクティブだけがサポートされています。strftime() はプラットフォームごとに実装されているので、説明されていないディレクティブも利用できるかもしれません。しかし、strptime() はプラットフォーム非依存なので、ドキュメント内でサポートされているとされているディレクティブ以外は利用できません。

class time.struct_time

gmtime(), *localtime()* および *strptime()* が返す時刻値シーケンスの型です。これは **名前付きタ**

[*1] %Z の使用は現在非推奨です。ただし、ここで実現したい時間および分オフセットへの展開を行ってくれる %z エスケープはすべての ANSI C ライブラリでサポートされているわけではありません。また、1982 年に提出されたオリジナルの **RFC 822** 標準では西暦の表現を 2 桁とするよう要求している (%Y でなく %y) ものの、実際には 2000 年になるだいぶ以前から 4 桁の西暦表現に移行しています。その後 **RFC 822** は撤廃され、4 桁の西暦表現は **RFC 1123** で初めて勧告され、**RFC 2822** において義務付けられました。

プル のインタフェースをもったオブジェクトです。値はインデックスでも属性名でもアクセス可能です。以下の値があります:

インデックス	属性	値
0	tm_year	(例えば 1993)
1	tm_mon	[1,12] の間の数
2	tm_mday	[1,31] の間の数
3	tm_hour	[0,23] の間の数
4	tm_min	[0,59] の間の数
5	tm_sec	[0,61] の間の数 *strftime()* の説明にある **(2)** を読んで下さい
6	tm_wday	[0,6] の間の数、月曜が 0 になります
7	tm_yday	[1,366] の間の数
8	tm_isdst	0, 1 または -1; 以下を参照してください
N/A	tm_zone	タイムゾーンの短縮名
N/A	tm_gmtoff	UTC から東方向へのオフセット (秒)

C の構造体とは異なり、月の値は [0, 11] ではなく [1, 12] であることに注意してください。

mktime() の呼び出し時に、tm_isdst は夏時間が有効な場合は 1、そうでない場合は 0 に設定されることがあります。値が -1 の場合は夏時間について不明なことを表していて、普通 tm_isdst は正しい状態に設定されます。

struct_time を引数とする関数に正しくない長さの *struct_time* や要素の型が正しくない *struct_time* を与えた場合には、*TypeError* が送出されます。

time.time() → float

エポック からの秒数を浮動小数点数で返します。エポックの具体的な日付とうるう秒 (leap seconds) の扱いはプラットフォーム依存です。Windows とほとんどの Unix システムでは、エポックは (UTC で) 1970 年 1 月 1 日 0 時 0 分 0 秒で、うるう秒はエポック秒の時間の勘定には入りません。これは一般に Unix 時間 と呼ばれています。与えられたプラットフォームでエポックが何なのかを知るには、time.gmtime(0) の値を見てください。

時刻は常に浮動小数点数で返されますが、すべてのシステムが 1 秒より高い精度で時刻を提供するとは限らないので注意してください。この関数が返す値は通常減少していくことはありませんが、この関数を 2 回呼び出し、その呼び出しの間にシステムクロックの時刻を巻き戻して設定した場合には、以前の呼び出しよりも低い値が返ることがあります。

time() が返す数値は、*gmtime()* 関数に渡されて UTC の、あるいは *localtime()* 関数に渡されて現地時間の、より一般的な時間のフォーマット (つまり、年、月、日、時間など) に変換されているかもしれません。どちらの場合でも *struct_time* オブジェクトが返され、このオブジェクトの属性としてカレンダー日付の構成要素へアクセスできます。

time.thread_time() → float

現在のスレッドのシステムおよびユーザー CPU 時間の値を (小数秒で) 返します。これはスリープ中の経過時間を含みません。これは定義上スレッド固有です。戻り値の基準点は定義されていないので、同一スレッドにおける二回の呼び出しの結果の差だけが有効です。

利用可能な環境: Windows, Linux, CLOCK_THREAD_CPUTIME_ID をサポートしている Unix システム。

バージョン 3.7 で追加.

time.**thread_time_ns**() → int

thread_time() に似ていますが、ナノ秒単位の時刻を返します。

バージョン 3.7 で追加.

time.**time_ns**() → int

time() に似ていますが、時刻を *epoch* を基点としたナノ秒単位の整数で返します。

バージョン 3.7 で追加.

time.**tzset**()

Reset the time conversion rules used by the library routines. The environment variable TZ specifies how this is done. It will also set the variables **tzname** (from the TZ environment variable), **timezone** (non-DST seconds West of UTC), **altzone** (DST seconds west of UTC) and **daylight** (to 0 if this timezone does not have any daylight saving time rules, or to nonzero if there is a time, past, present or future when daylight saving time applies).

利用可能な環境: Unix。

注釈: 多くの場合、環境変数 TZ を変更すると、*tzset()* を呼ばない限り *localtime()* のような関数の出力に影響を及ぼすため、値が信頼できなくなってしまいます。

TZ 環境変数には空白文字を含めてはなりません。

環境変数 TZ の標準的な書式は以下の通りです (分かりやすいように空白を入れています):

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

各値は以下のようになっています:

std と dst 三文字またはそれ以上の英数字で、タイムゾーンの略称を与えます。この値は time.tzname になります。

offset オフセットは形式: ± hh[:mm[:ss]] をとります。この表現は、UTC 時刻にするためにローカルな時間に加算する必要のある時間値を示します。'-' が先頭につく場合、そのタイムゾーンは本初子午線 (Prime Meridian) より東側にあります。それ以外の場合は本初子午線の西側です。オフセットが dst の後ろに続かない場合、夏時間は標準時より一時間先行しているものと仮定します。

start[/time], end[/time] いつ DST に移動し、DST から戻ってくるかを示します。開始および終了日時の形式は以下のいずれかです:

Jn ユリウス日 (Julian day) n (1 <= n <= 365) を表します。うるう日は計算に含められないため、2 月 28 日は常に 59 で、3 月 1 日は 60 になります。

n ゼロから始まるユリウス日 (0 <= *n* <= 365) です。うるう日は計算に含められるため、2 月 29 日を参照することができます。

M*m*.*n*.*d* *m* 月の週 *n* における *d* 番目の日 (0 <= *d* <= 6, 1 <= *n* <= 5, 1 <= *m* <= 12) を表します。週 5 は月 *m* における最終週の *d* 番目の日を表し、第 4 週か第 5 週のどちらかになります。週 1 は日 *d* が最初に現れる日を指します。日 0 は日曜日です。

`time` は `offset` とほぼ同じで、先頭に符号 ('-' や '+') を付けてはいけないところだけが違います。時刻が指定されていなければ、デフォルトの値 02:00:00 になります。

```
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'
```

多くの Unix システム (*BSD, Linux, Solaris, および Darwin を含む) では、システムの zoneinfo (*tzfile(5)*) データベースを使ったほうが、タイムゾーンごとの規則を指定する上で便利です。これを行うには、必要なタイムゾーンデータファイルへのパスをシステムの 'zoneinfo' タイムゾーンデータベースからの相対で表した値を環境変数 `TZ` に設定します。システムの 'zoneinfo' は通常 `/usr/share/zoneinfo` にあります。例えば、`'US/Eastern'`、`'Australia/Melbourne'`、`'Egypt'` ないし `'Europe/Amsterdam'` と指定します。

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

16.3.2 Clock ID Constants

These constants are used as parameters for *clock_getres()* and *clock_gettime()*.

`time.CLOCK_BOOTTIME`

Identical to *CLOCK_MONOTONIC*, except it also includes any time that the system is suspended.

This allows applications to get a suspend-aware monotonic clock without having to deal with the complications of *CLOCK_REALTIME*, which may have discontinuities if the time is changed using `settimeofday()` or similar.

利用可能な環境: Linux 2.6.39 以上。

バージョン 3.7 で追加.

time.CLOCK_HIGHRES
: Solaris OS は任意のハードウェアソースの使用を試み、ナノ秒レベルの分解能を提供する CLOCK_HIGHRES タイマーを具備しています。CLOCK_HIGHRES は変更不可で、高分解能のクロックです。

  利用可能な環境: Solaris。

  バージョン 3.3 で追加.

time.CLOCK_MONOTONIC
: 設定不可で、モノトニック時刻 (不特定のエポックからの単調増加な時刻) を表します。

  利用可能な環境: Unix。

  バージョン 3.3 で追加.

time.CLOCK_MONOTONIC_RAW
: *CLOCK_MONOTONIC* と似ていますが、NTP の影響を受けていない、ハードウェアベースの時刻へのアクセスを提供します。

  利用可能な環境: Linux 2.6.28 以上、macOS 10.12 以上。

  バージョン 3.3 で追加.

time.CLOCK_PROCESS_CPUTIME_ID
: CPU による高分解能のプロセスごとのタイマーです。

  利用可能な環境: Unix。

  バージョン 3.3 で追加.

time.CLOCK_PROF
: CPU による高分解能のプロセスごとのタイマーです。

  利用可能な環境: FreeBSD、NetBSD 7 以上、OpenBSD.。

  バージョン 3.7 で追加.

time.CLOCK_THREAD_CPUTIME_ID
: スレッド固有の CPU タイムクロックです。

  利用可能な環境: Unix。

  バージョン 3.3 で追加.

time.CLOCK_UPTIME
: Time whose absolute value is the time the system has been running and not suspended, providing accurate uptime measurement, both absolute and interval.

  利用可能な環境: FreeBSD、OpenBSD 5.5 以上。

  バージョン 3.7 で追加.

time.**CLOCK_UPTIME_RAW**
Clock that increments monotonically, tracking the time since an arbitrary point, unaffected by frequency or time adjustments and not incremented while the system is asleep.

利用可能な環境: macOS 10.12 以上。

バージョン 3.8 で追加.

The following constant is the only parameter that can be sent to *clock_settime()*.

time.**CLOCK_REALTIME**
システム全体のリアルタイムクロックです。このクロックを設定するには適切な権限が必要です。

利用可能な環境: Unix。

バージョン 3.3 で追加.

16.3.3 Timezone Constants

time.**altzone**
ローカルの夏時間タイムゾーンにおける UTC からの時刻オフセットで、西に行くほど増加する、秒で表した値です (ほとんどの西ヨーロッパでは負になり、アメリカでは正、イギリスではゼロになります)。`daylight` がゼロでないときのみ使用してください。以下の注釈を参照してください。

time.**daylight**
DST タイムゾーンが定義されている場合ゼロでない値になります。以下の注釈を参照してください。

time.**timezone**
(DST でない) ローカルタイムゾーンの UTC からの時刻オフセットで、西に行くほど増加する秒で表した値です (ほとんどの西ヨーロッパでは負になり、アメリカでは正、イギリスではゼロになります)。以下の注釈を参照してください。

time.**tzname**
二つの文字列からなるタプルです。最初の要素は DST でないローカルのタイムゾーン名です。ふたつめの要素は DST のタイムゾーンです。DST のタイムゾーンが定義されていない場合。二つ目の文字列を使うべきではありません。以下の注釈を参照してください。

注釈: For the above Timezone constants (*altzone*, *daylight*, *timezone*, and *tzname*), the value is determined by the timezone rules in effect at module load time or the last time *tzset()* is called and may be incorrect for times in the past. It is recommended to use the `tm_gmtoff` and `tm_zone` results from *localtime()* to obtain timezone information.

参考:

datetime **モジュール** 日付と時刻に対する、よりオブジェクト指向のインタフェースです。

locale **モジュール**　国際化サービスです。ロケールの設定は *strftime()* および *strptime()* の多くの書式指定子の解釈に影響を及ぼします。

calendar **モジュール**　一般的なカレンダーに関する関数群です。*timegm()* はこのモジュールの *gmtime()* の逆を行う関数です。

脚注

16.4 argparse --- コマンドラインオプション、引数、サブコマンドのパーサー

バージョン 3.2 で追加.

ソースコード: Lib/argparse.py

チュートリアル

このページは API のリファレンス情報が記載しています。argparse チュートリアル では、コマンドラインの解析についてより優しく説明しています。

argparse モジュールはユーザーフレンドリなコマンドラインインターフェースの作成を簡単にします。プログラムがどんな引数を必要としているのかを定義すると、*argparse* が *sys.argv* からそのオプションを解析する方法を見つけ出します。*argparse* モジュールは自動的にヘルプと使用方法メッセージを生成し、ユーザーが不正な引数をプログラムに指定したときにエラーを発生させます。

16.4.1 使用例

次のコードは、整数のリストを受け取って合計か最大値を返す Python プログラムです:

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

上の Python コードが prog.py という名前のファイルに保存されたと仮定します。コマンドラインから便利なヘルプメッセージを表示できます:

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
 N           an integer for the accumulator

optional arguments:
 -h, --help  show this help message and exit
 --sum       sum the integers (default: find the max)
```

適切な引数を与えて実行した場合、このプログラムはコマンドライン引数の整数列の合計か最大値を表示します:

```
$ python prog.py 1 2 3 4
4

$ python prog.py 1 2 3 4 --sum
10
```

不正な引数が与えられた場合、エラーを発生させます:

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

以降の節では、この例をひと通り説明して行きます。

パーサーを作る

argparse を使うときの最初のステップは、*ArgumentParser* オブジェクトを生成することです:

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

ArgumentParser オブジェクトはコマンドラインを解析して Python データ型にするために必要なすべての情報を保持します。

引数を追加する

ArgumentParser にプログラム引数の情報を与えるために、*add_argument()* メソッドを呼び出します。一般的に、このメソッドの呼び出しは *ArgumentParser* に、コマンドラインの文字列を受け取ってそれをオブジェクトにする方法を教えます。この情報は保存され、*parse_args()* が呼び出されたときに利用されます。例えば:

```
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
...                     help='an integer for the accumulator')
>>> parser.add_argument('--sum', dest='accumulate', action='store_const',
```

(次のページに続く)

(前のページからの続き)

```
...                     const=sum, default=max,
...                     help='sum the integers (default: find the max)')
```

あとで *parse_args()* を呼び出すと、integers と accumulate という 2 つの属性を持ったオブジェクトを返します。integers 属性は 1 つ以上の整数のリストで、accumulate 属性はコマンドラインから --sum が指定された場合は *sum()* 関数に、それ以外の場合は *max()* 関数になります。

引数を解析する

ArgumentParser は引数を *parse_args()* メソッドで解析します。このメソッドはコマンドラインを調べ、各引数を正しい型に変換して、適切なアクションを実行します。ほとんどの場合、これはコマンドラインの解析結果から、シンプルな *Namespace* オブジェクトを構築することを意味します:

```
>>> parser.parse_args(['--sum', '7', '-1', '42'])
Namespace(accumulate=<built-in function sum>, integers=[7, -1, 42])
```

スクリプト内では、*parse_args()* は通常引数なしで呼び出され、*ArgumentParser* は自動的に *sys.argv* からコマンドライン引数を取得します。

16.4.2 ArgumentParser オブジェクト

class argparse.**ArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=argparse.HelpFormatter*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*)

新しい *ArgumentParser* オブジェクトを生成します。すべての引数はキーワード引数として渡すべきです。各引数についてはあとで詳しく説明しますが、簡単に言うと:

- *prog* - プログラム名 (デフォルト: sys.argv[0])
- *usage* - プログラムの利用方法を記述する文字列 (デフォルト: パーサーに追加された引数から生成されます)
- *description* - 引数のヘルプの前に表示されるテキスト (デフォルト: none)
- *epilog* - 引数のヘルプの後で表示されるテキスト (デフォルト: none)
- *parents* - *ArgumentParser* オブジェクトのリストで、このオブジェクトの引数が追加されます
- *formatter_class* - ヘルプ出力をカスタマイズするためのクラス
- *prefix_chars* - オプションの引数の prefix になる文字集合 (デフォルト: '-')
- *fromfile_prefix_chars* - 追加の引数を読み込むファイルの prefix になる文字集合 (デフォルト: None)
- *argument_default* - 引数のグローバルなデフォルト値 (デフォルト: None)

- *conflict_handler* - 衝突するオプションを解決する方法 (通常は不要)
- *add_help* - -h/--help オプションをパーサーに追加する (デフォルト: True)
- *allow_abbrev* - 長いオプションが先頭の 1 文字に短縮可能 (先頭の文字が一意) である場合に短縮指定を許可する。(デフォルト: True)

バージョン 3.5 で変更: *allow_abbrev* 引数が追加されました。

バージョン 3.8 で変更: 以前のバージョンでは、*allow_abbrev* は、-vv が -v -v と等価になるような、短いフラグのグループ化を無効にしていました。

以下の節では各オプションの利用方法を説明します。

prog

デフォルトでは、*ArgumentParser* オブジェクトはヘルプメッセージ中に表示するプログラム名を sys.argv[0] から取得します。このデフォルトの動作は、プログラムがコマンドライン上の起動方法に合わせてヘルプメッセージを作成するため、ほとんどの場合望ましい挙動になります。例えば、myprogram.py という名前のファイルに次のコードがあるとします:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

このプログラムのヘルプは、プログラム名として (プログラムがどこから起動されたのかに関わらず) myprogram.py を表示します:

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
$ cd ..
$ python subdir/myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

このデフォルトの動作を変更するには、*ArgumentParser* の prog= 引数に他の値を指定します:

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

optional arguments:
 -h, --help  show this help message and exit
```

プログラム名は、sys.argv[0] から取られた場合でも prog= 引数で与えられた場合でも、ヘルプメッセージ中では %(prog)s フォーマット指定子で利用できます。

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo of the myprogram program
```

usage

デフォルトでは、*ArgumentParser* は使用法メッセージを、保持している引数から生成します:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [-h] [--foo [FOO]] bar [bar ...]

positional arguments:
 bar          bar help

optional arguments:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

デフォルトのメッセージは usage= キーワード引数で変更できます:

```
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%(prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [options]

positional arguments:
 bar          bar help

optional arguments:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

%(prog)s フォーマット指定子を、使用法メッセージ内でプログラム名として利用できます。

description

多くの場合、*ArgumentParser* のコンストラクターを呼び出すときに description= キーワード引数が使用されます。この引数はプログラムが何をしてどう動くのかについての短い説明になります。ヘルプメッセージで、この説明がコマンドラインの利用法と引数のヘルプメッセージの間に表示されます:

```
>>> parser = argparse.ArgumentParser(description='A foo that bars')
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
 -h, --help  show this help message and exit
```

デフォルトでは、説明は与えられたスペースに合わせて折り返されます。この挙動を変更するには、*formatter_class* 引数を参照してください。

epilog

いくつかのプログラムは、プログラムについての追加の説明を引数の説明の後に表示します。このテキストは *ArgumentParser* の epilog= 引数に指定できます:

```
>>> parser = argparse.ArgumentParser(
...     description='A foo that bars',
...     epilog="And that's how you'd foo a bar")
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
 -h, --help  show this help message and exit

And that's how you'd foo a bar
```

description 引数と同じく、epilog= テキストもデフォルトで折り返され、*ArgumentParser* の *formatter_class* 引数で動作を調整できます。

parents

ときどき、いくつかのパーサーが共通の引数セットを共有することがあります。それらの引数を繰り返し定義する代わりに、すべての共通引数を持ったパーサーを *ArgumentParser* の parents= 引数に渡すことができます。parents= 引数は *ArgumentParser* オブジェクトのリストを受け取り、すべての位置アクションとオプションのアクションをそれらから集め、そのアクションを構築中の *ArgumentParser* オブジェクトに追加します:

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

一番親になるパーサーに add_help=False を指定していることに注目してください。こうしないと、*ArgumentParser* は 2 つの -h/--help オプションを与えられる (1 つは親から、もうひとつは子から) ことになり、エラーを発生します。

注釈: parents= に渡す前にパーサーを完全に初期化する必要があります。子パーサーを作成してから親パーサーを変更した場合、その変更は子パーサーに反映されません。

---

formatter_class

ArgumentParser オブジェクトは代わりのフォーマットクラスを指定することでヘルプのフォーマットをカスタマイズできます。現在、4 つのフォーマットクラスがあります:

class argparse.RawDescriptionHelpFormatter

class argparse.RawTextHelpFormatter

class argparse.ArgumentDefaultsHelpFormatter

class argparse.MetavarTypeHelpFormatter

RawDescriptionHelpFormatter と *RawTextHelpFormatter* はどのようにテキストの説明を表示するかを指定できます。デフォルトでは *ArgumentParser* オブジェクトはコマンドラインヘルプの中の *description* と *epilog* を折り返して表示します:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     description='''this description
...         was indented weird
...             but that is okay''',
...     epilog='''
...             likewise for this epilog whose whitespace will
...         be cleaned up and whose words will be wrapped
...         across a couple lines''')
>>> parser.print_help()
usage: PROG [-h]

this description was indented weird but that is okay
```

(次のページに続く)

(前のページからの続き)

```
optional arguments:
 -h, --help  show this help message and exit

likewise for this epilog whose whitespace will be cleaned up and whose words
will be wrapped across a couple lines
```

formatter_class= に *RawDescriptionHelpFormatter* を渡した場合、*description* と *epilog* は整形済みとされ改行されません:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.RawDescriptionHelpFormatter,
...     description=textwrap.dedent('''\
...         Please do not mess up this text!
...         --------------------------------
...             I have indented it
...             exactly the way
...             I want it
...         '''))
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!
--------------------------------
   I have indented it
   exactly the way
   I want it

optional arguments:
 -h, --help  show this help message and exit
```

RawTextHelpFormatter は引数の説明を含めてすべての種類のヘルプテキストで空白を維持します。例外として、複数の空行はひとつにまとめられます。複数の空白行を保ちたい場合には、行に空白を含めるようにして下さい。

ArgumentDefaultsHelpFormatter は各引数のデフォルト値を自動的にヘルプに追加します:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
>>> parser.add_argument('--foo', type=int, default=42, help='FOO!')
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar [bar ...]]

positional arguments:
 bar         BAR! (default: [1, 2, 3])

optional arguments:
```

(次のページに続く)

(前のページからの続き)

```
 -h, --help  show this help message and exit
 --foo FOO   FOO! (default: 42)
```

MetavarTypeHelpFormatter は、各引数の値の表示名に *type* 引数の値を使用します (通常は *dest* の値が使用されます):

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.MetavarTypeHelpFormatter)
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', type=float)
>>> parser.print_help()
usage: PROG [-h] [--foo int] float

positional arguments:
  float

optional arguments:
  -h, --help  show this help message and exit
  --foo int
```

prefix_chars

ほとんどのコマンドラインオプションは、`-f/--foo` のように接頭辞に `-` を使います。`+f` や `/foo` のような、他の、あるいは追加の接頭辞文字をサポートしなければならない場合、ArgumentParser のコンストラクターに `prefix_chars=` 引数を使って指定します:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='-+')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')
```

`prefix_chars=` 引数のデフォルトは `'-'` です。`-` を含まない文字セットを指定すると、`-f/--foo` オプションが使用できなくなります。

fromfile_prefix_chars

ときどき、例えば非常に長い引数リストを扱う場合に、その引数リストを毎回コマンドラインにタイプする代わりにファイルに置いておきたい場合があります。*ArgumentParser* のコンストラクターに `fromfile_prefix_chars=` 引数が渡された場合、指定された文字のいずれかで始まる引数はファイルとして扱われ、そのファイルに含まれる引数リストに置換されます。例えば:

```
>>> with open('args.txt', 'w') as fp:
...     fp.write('-f\nbar')
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
```

(次のページに続く)

(前のページからの続き)

```
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

ファイルから読み込まれる引数は、デフォルトでは 1 行に 1 つ (ただし、*convert_arg_line_to_args()* も参照してください) で、コマンドライン上でファイルを参照する引数があった場所にその引数があったものとして扱われます。このため、上の例では、`['-f', 'foo', '@args.txt']` は `['-f', 'foo', '-f', 'bar']` と等価になります。

`fromfile_prefix_chars=` 引数のデフォルト値は `None` で、引数がファイル参照として扱われることがないことを意味しています。

argument_default

一般的には、引数のデフォルト値は *add_argument()* メソッドにデフォルト値を渡すか、*set_defaults()* メソッドに名前と値のペアを渡すことで指定します。しかしまれに、1 つのパーサー全体に適用されるデフォルト引数が便利なことがあります。これを行うには、*ArgumentParser* に `argument_default=` キーワード引数を渡します。例えば、全体で *parse_args()* メソッド呼び出しの属性の生成を抑制するには、`argument_default=SUPPRESS` を指定します:

```
>>> parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar', nargs='?')
>>> parser.parse_args(['--foo', '1', 'BAR'])
Namespace(bar='BAR', foo='1')
>>> parser.parse_args([])
Namespace()
```

allow_abbrev

通常、*ArgumentParser* の *parse_args()* に引数のリストを渡すとき、長いオプションは **短縮しても認識されます**。

この機能は、`allow_abbrev` に `False` を指定することで無効にできます:

```
>>> parser = argparse.ArgumentParser(prog='PROG', allow_abbrev=False)
>>> parser.add_argument('--foobar', action='store_true')
>>> parser.add_argument('--foonley', action='store_false')
>>> parser.parse_args(['--foon'])
usage: PROG [-h] [--foobar] [--foonley]
PROG: error: unrecognized arguments: --foon
```

バージョン 3.5 で追加.

conflict_handler

ArgumentParser オブジェクトは同じオプション文字列に対して複数のアクションを許可していません。デフォルトでは、*ArgumentParser* オブジェクトは、すでに利用されているオプション文字列を使って新しい引数をつくろうとしたときに例外を送出します:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
 ..
ArgumentError: argument --foo: conflicting option string(s): --foo
```

ときどき (例えば *parents* を利用する場合など)、古い引数を同じオプション文字列で上書きするほうが便利な場合があります。この動作をするには、*ArgumentParser* の conflict_handler= 引数に 'resolve' を渡します:

```
>>> parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 -f FOO      old foo help
 --foo FOO   new foo help
```

ArgumentParser オブジェクトは、すべてのオプション文字列が上書きされた場合にだけアクションを削除することに注目してください。上の例では、--foo オプション文字列だけが上書きされているので、古い -f/--foo アクションは -f アクションとして残っています。

add_help

デフォルトでは、ArgumentParser オブジェクトはシンプルにパーサーのヘルプメッセージを表示するオプションを自動的に追加します。例えば、以下のコードを含む myprogram.py ファイルについて考えてください:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

コマンドラインに -h か --help が指定された場合、ArgumentParser の help が表示されます:

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
```

(次のページに続く)

(前のページからの続き)

```
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

必要に応じて、この help オプションを無効にする場合があります。これは *ArgumentParser* の add_help= 引数に False を渡すことで可能です:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [--foo FOO]

optional arguments:
 --foo FOO  foo help
```

ヘルプオプションは通常 -h/--help です。例外は prefix_chars= が指定されてその中に - が無かった場合で、その場合は -h と --help は有効なオプションではありません。この場合、prefix_chars の最初の文字がヘルプオプションの接頭辞として利用されます:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]

optional arguments:
  +h, ++help  show this help message and exit
```

16.4.3 add_argument() メソッド

ArgumentParser.**add_argument**(*name or flags...*[, *action*][, *nargs*][, *const*][, *default*][, *type*][, *choices*][, *required*][, *help*][, *metavar*][, *dest*])

1 つのコマンドライン引数がどう解析されるかを定義します。各引数についての詳細は後述しますが、簡単に言うと:

- *name* または *flags* - 名前か、あるいはオプション文字列のリスト (例: foo や -f, --foo)。
- *action* - コマンドラインにこの引数があったときのアクション。
- *nargs* - 受け取るべきコマンドライン引数の数。
- *const* - 一部の *action* と *nargs* の組み合わせで利用される定数。
- *default* - コマンドラインに引数がなかった場合に生成される値。
- *type* - コマンドライン引数が変換されるべき型。
- *choices* - 引数として許される値のコンテナー。
- *required* - コマンドラインオプションが省略可能かどうか (オプション引数のみ)。
- *help* - 引数が何なのかを示す簡潔な説明。

- *metavar* - 使用法メッセージの中で使われる引数の名前。
- *dest* - *parse_args()* が返すオブジェクトに追加される属性名。

以下の節では各オプションの利用方法を説明します。

name または flags

add_argument() メソッドは、指定されている引数が `-f` や `--foo` のようなオプション引数なのか、ファイル名リストなどの位置引数なのかを知る必要があります。そのため、*add_argument()* の第 1 引数は、フラグのリストか、シンプルな引数名のどちらかになります。例えば、オプション引数は次のようにして作成します:

```
>>> parser.add_argument('-f', '--foo')
```

一方、位置引数は次のように作成します:

```
>>> parser.add_argument('bar')
```

parse_args() が呼ばれたとき、オプション引数は接頭辞 `-` により識別され、それ以外の引数は位置引数として扱われます:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
Namespace(bar='BAR', foo='FOO')
>>> parser.parse_args(['--foo', 'FOO'])
usage: PROG [-h] [-f FOO] bar
PROG: error: the following arguments are required: bar
```

action

ArgumentParser オブジェクトはコマンドライン引数にアクションを割り当てます。このアクションは、割り当てられたコマンドライン引数に関してどんな処理でもできますが、ほとんどのアクションは単に *parse_args()* が返すオブジェクトに属性を追加するだけです。`action` キーワード引数は、コマンドライン引数がどう処理されるかを指定します。提供されているアクションは:

- `'store'` - これは単に引数の値を格納します。これはデフォルトのアクションです。例えば:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo')
  >>> parser.parse_args('--foo 1'.split())
  Namespace(foo='1')
  ```

- `'store_const'` - このアクションは *const* キーワード引数で指定された値を格納します。`'store_const'` アクションは、何かの種類のフラグを指定するオプション引数によく使われます。例えば:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_const', const=42)
>>> parser.parse_args(['--foo'])
Namespace(foo=42)
```

- 'store_true', 'store_false' - これらは 'store_const' の、それぞれ True と False を格納する特別版になります。加えて、これらはそれぞれデフォルト値を順に False と True にします。例えば:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('--bar', action='store_false')
>>> parser.add_argument('--baz', action='store_false')
>>> parser.parse_args('--foo --bar'.split())
Namespace(foo=True, bar=False, baz=True)
```

- 'append' - このアクションはリストを格納して、各引数の値をそのリストに追加します。このアクションは複数回指定を許可したいオプションに便利です。利用例:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append')
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['1', '2'])
```

- 'append_const' - このアクションはリストを格納して、*const* キーワード引数に与えられた値をそのリストに追加します (*const* キーワード引数のデフォルト値はあまり役に立たない None であることに注意)。'append_const' アクションは、定数を同じリストに複数回格納する場合に便利です。例えば:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const', const=str)
>>> parser.add_argument('--int', dest='types', action='append_const', const=int)
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<class 'str'>, <class 'int'>])
```

- 'count' - このアクションはキーワード引数の数を数えます。例えば、verbose レベルを上げるのに役立ちます:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count', default=0)
>>> parser.parse_args(['-vvv'])
Namespace(verbose=3)
```

  default は明示的に *0* と指定されない場合は None であることに注意してください。

- 'help' - このアクションは現在のパーサー中のすべてのオプションのヘルプメッセージを表示し、終了します。出力の生成方法の詳細については *ArgumentParser* を参照してください。

- 'version' - このアクションは *add_argument()* の呼び出しに version= キーワード引数を期待します。指定されたときはバージョン情報を表示して終了します:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--version', action='version', version='%(prog)s 2.0')
>>> parser.parse_args(['--version'])
PROG 2.0
```

- 'extend' - このアクションはリストを格納して、各引数の値でそのリストを拡張します。利用例:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument("--foo", action="extend", nargs="+", type=str)
>>> parser.parse_args(["--foo", "f1", "--foo", "f2", "f3", "f4"])
Namespace(foo=['f1', 'f2', 'f3', 'f4'])
```

バージョン 3.8 で追加.

Action のサブクラスまたは同じインターフェイスを実装したほかのオブジェクト渡すことで、任意のアクションを指定することもできます。これをするお奨めの方法は、*argparse.Action* を継承して、__call__ と、必要であれば __init__ をオーバライドすることです。

カスタムアクションの例です:

```
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None, **kwargs):
...         if nargs is not None:
...             raise ValueError("nargs not allowed")
...         super().__init__(option_strings, dest, **kwargs)
...     def __call__(self, parser, namespace, values, option_string=None):
...         print('%r %r %r' % (namespace, values, option_string))
...         setattr(namespace, self.dest, values)
...
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=FooAction)
>>> parser.add_argument('bar', action=FooAction)
>>> args = parser.parse_args('1 --foo 2'.split())
Namespace(bar=None, foo=None) '1' None
Namespace(bar='1', foo=None) '2' '--foo'
>>> args
Namespace(bar='1', foo='2')
```

詳細は *Action* を参照してください。

nargs

ArgumentParser オブジェクトは通常 1 つのコマンドライン引数を 1 つのアクションに渡します。nargs キーワード引数は 1 つのアクションにそれ以外の数のコマンドライン引数を割り当てます。指定できる値は:

- N (整数) -- N 個の引数がコマンドラインから集められ、リストに格納されます。例えば:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs=2)
```

(次のページに続く)

(前のページからの続き)

```
>>> parser.add_argument('bar', nargs=1)
>>> parser.parse_args('c --foo a b'.split())
Namespace(bar=['c'], foo=['a', 'b'])
```

nargs=1 は 1 要素のリストを作ることに注意してください。これはデフォルトの、要素がそのまま属性になる動作とは異なります。

- '?' -- 可能なら 1 つの引数がコマンドラインから取られ、1 つのアイテムを作ります。コマンドライン引数が存在しない場合、*default* の値が生成されます。オプション引数の場合、さらにオプション引数が指定され、その後にコマンドライン引数がないというケースもありえます。この場合は *const* の値が生成されます。この動作の例です:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='?', const='c', default='d')
>>> parser.add_argument('bar', nargs='?', default='d')
>>> parser.parse_args(['XX', '--foo', 'YY'])
Namespace(bar='XX', foo='YY')
>>> parser.parse_args(['XX', '--foo'])
Namespace(bar='XX', foo='c')
>>> parser.parse_args([])
Namespace(bar='d', foo='d')
```

nargs='?' のよくある利用例の 1 つは、入出力ファイルの指定オプションです:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', nargs='?', type=argparse.FileType('r'),
...                     default=sys.stdin)
>>> parser.add_argument('outfile', nargs='?', type=argparse.FileType('w'),
...                     default=sys.stdout)
>>> parser.parse_args(['input.txt', 'output.txt'])
Namespace(infile=<_io.TextIOWrapper name='input.txt' encoding='UTF-8'>,
          outfile=<_io.TextIOWrapper name='output.txt' encoding='UTF-8'>)
>>> parser.parse_args([])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>,
          outfile=<_io.TextIOWrapper name='<stdout>' encoding='UTF-8'>)
```

- '*' -- すべてのコマンドライン引数がリストに集められます。複数の位置引数が nargs='*' を持つことにあまり意味はありませんが、複数のオプション引数が nargs='*' を持つことはありえます。例えば:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='*')
>>> parser.add_argument('--bar', nargs='*')
>>> parser.add_argument('baz', nargs='*')
>>> parser.parse_args('a b --foo x y --bar 1 2'.split())
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])
```

- '+' -- '*' と同じように、すべてのコマンドライン引数をリストに集めます。加えて、最低でも 1 つのコマンドライン引数が存在しない場合にエラーメッセージを生成します。例えば:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.parse_args(['a', 'b'])
Namespace(foo=['a', 'b'])
>>> parser.parse_args([])
usage: PROG [-h] foo [foo ...]
PROG: error: the following arguments are required: foo
```

- argparse.REMAINDER -- コマンドライン引数の残りすべてをリストとして集めます。これは他のコマンドラインツールに対して処理を渡すようなツールによく使われます。例えば:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo')
>>> parser.add_argument('command')
>>> parser.add_argument('args', nargs=argparse.REMAINDER)
>>> print(parser.parse_args('--foo B cmd --arg1 XX ZZ'.split()))
Namespace(args=['--arg1', 'XX', 'ZZ'], command='cmd', foo='B')
```

nargs キーワード引数が指定されない場合、受け取る引数の数は *action* によって決定されます。通常これは、1 つのコマンドライン引数は 1 つのアイテムになる (リストにはならない) ことを意味します。

const

add_argument() の const 引数は、コマンドライン引数から読み込まれないけれども *ArgumentParser* のいくつかのアクションで必要とされる値のために使われます。この引数のよくある 2 つの使用法は:

- *add_argument()* が action='store_const' か action='append_const' で呼び出されたとき、これらのアクションは const の値を *parse_args()* が返すオブジェクトの属性に追加します。サンプルは *action* の説明を参照してください。
- *add_argument()* がオプション文字列 (-f や --foo) と nargs='?' で呼び出された場合。この場合 0 個か 1 つのコマンドライン引数を取るオプション引数が作られます。オプション引数にコマンドライン引数が続かなかった場合、const の値が代わりに利用されます。サンプルは *nargs* の説明を参照してください。

'store_const' と 'append_const' アクションでは、const キーワード引数を与える必要があります。他のアクションでは、デフォルトは None になります。

default

すべてのオプション引数といくつかの位置引数はコマンドライン上で省略されることがあります。*add_argument()* の default キーワード引数 (デフォルト: None) は、コマンドライン引数が存在しなかった場合に利用する値を指定します。オプション引数では、オプション文字列がコマンドライン上に存在しなかったときに default の値が利用されます:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
```

(次のページに続く)

(前のページからの続き)

```
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
>>> parser.parse_args([])
Namespace(foo=42)
```

default の値が文字列の場合、パーサーは値をコマンドライン引数のように解析します。具体的には、パーサーは返り値 *Namespace* の属性を設定する前に、*type* 変換引数が与えられていればそれらを適用します。そうでない場合、パーサーは値をそのまま使用します:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

nargs が ? か * である位置引数では、コマンドライン引数が指定されなかった場合 default の値が使われます。例えば:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

default=argparse.SUPPRESS を渡すと、コマンドライン引数が存在しないときに属性の追加をしなくなります:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

type

デフォルトでは、*ArgumentParser* オブジェクトはコマンドライン引数を単なる文字列として読み込みます。しかし、コマンドラインの文字列は *float*, *int* など別の型として扱うべき事がよくあります。*add_argument()* の type キーワード引数により型チェックと型変換を行うことができます。一般的なビルトインデータ型や関数を type 引数の値として直接指定できます:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.add_argument('bar', type=open)
>>> parser.parse_args('2 temp.txt'.split())
Namespace(bar=<_io.TextIOWrapper name='temp.txt' encoding='UTF-8'>, foo=2)
```

type 引数がデフォルト引数に適用されている場合の情報は、*default* キーワード引数の節を参照してください。

いろいろな種類のファイルを簡単に扱うために、argparse モジュールは *open()* 関数の mode=, bufsize=, encoding= および errors= 引数を取る FileType ファクトリを提供しています。例えば、書き込み可能なファイルを作るために FileType('w') を利用できます:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar', type=argparse.FileType('w'))
>>> parser.parse_args(['out.txt'])
Namespace(bar=<_io.TextIOWrapper name='out.txt' encoding='UTF-8'>)
```

type= には 1 つの文字列を引数に受け取って変換結果を返すような任意の呼び出し可能オブジェクトを渡すことができます:

```
>>> def perfect_square(string):
...     value = int(string)
...     sqrt = math.sqrt(value)
...     if sqrt != int(sqrt):
...         msg = "%r is not a perfect square" % string
...         raise argparse.ArgumentTypeError(msg)
...     return value
...
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=perfect_square)
>>> parser.parse_args(['9'])
Namespace(foo=9)
>>> parser.parse_args(['7'])
usage: PROG [-h] foo
PROG: error: argument foo: '7' is not a perfect square
```

さらに、*choices* キーワード引数を使って、値の範囲をチェックすることもできます:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=int, choices=range(5, 10))
>>> parser.parse_args(['7'])
Namespace(foo=7)
>>> parser.parse_args(['11'])
usage: PROG [-h] {5,6,7,8,9}
PROG: error: argument foo: invalid choice: 11 (choose from 5, 6, 7, 8, 9)
```

詳細は *choices* 節を参照してください。

choices

コマンドライン引数をいくつかの選択肢の中から選ばせたい場合があります。これは *add_argument()* に *choices* キーワード引数を渡すことで可能です。コマンドラインを解析するとき、引数の値がチェックされ、その値が選択肢の中に含まれていない場合はエラーメッセージを表示します:

```
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock',
'paper', 'scissors')
```

choices コンテナーに含まれているかどうかのチェックは、*type* による型変換が実行された後であることに注意してください。このため、*choices* に格納するオブジェクトの型は指定された *type* にマッチしている必要があります:

```
>>> parser = argparse.ArgumentParser(prog='doors.py')
>>> parser.add_argument('door', type=int, choices=range(1, 4))
>>> print(parser.parse_args(['3']))
Namespace(door=3)
>>> parser.parse_args(['4'])
usage: doors.py [-h] {1,2,3}
doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)
```

任意のコンテナを *choices* に渡すことができます。すなわち、*list* 、*set*、カスタムコンテナなどはすべてサポートされています。

required

通常 *argparse* モジュールは、-f や --bar といったフラグは **任意** の引数 (オプション引数) だと仮定し、コマンドライン上になくても良いものとして扱います。フラグの指定を **必須** にするには、*add_argument()* の required= キーワード引数に True を指定します:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
usage: [-h] --foo FOO
: error: the following arguments are required: --foo
```

上の例のように、引数が required と指定されると、*parse_args()* はそのフラグがコマンドラインに存在しないときにエラーを表示します。

注釈: ユーザーは、通常 **フラグ** の指定は **任意** であると認識しているため、必須にするのは一般的には悪い

やり方で、できる限り避けるべきです。

help

help の値はその引数の簡潔な説明を含む文字列です。ユーザーが (コマンドライン上で -h か --help を指定するなどして) ヘルプを要求したとき、この help の説明が各引数に表示されます:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', action='store_true',
...                     help='foo the bars before frobbling')
>>> parser.add_argument('bar', nargs='+',
...                     help='one of the bars to be frobbled')
>>> parser.parse_args(['-h'])
usage: frobble [-h] [--foo] bar [bar ...]

positional arguments:
 bar     one of the bars to be frobbled

optional arguments:
 -h, --help  show this help message and exit
 --foo   foo the bars before frobbling
```

help 文字列には、プログラム名や引数の *default* などを繰り返し記述するのを避けるためのフォーマット指定子を含めることができます。利用できる指定子には、プログラム名 %(prog)s と、%(default)s や %(type)s など *add_argument()* のキーワード引数の多くが含まれます:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
...                     help='the bar to %(prog)s (default: %(default)s)')
>>> parser.print_help()
usage: frobble [-h] [bar]

positional arguments:
 bar     the bar to frobble (default: 42)

optional arguments:
 -h, --help  show this help message and exit
```

ヘルプ文字列は %-フォーマットをサポートしているので、ヘルプ文字列内にリテラル % を表示したい場合は %% のようにエスケープしなければなりません。

argparse は help に argparse.SUPPRESS を設定することで、特定のオプションをヘルプに表示させないことができます:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]
```

(次のページに続く)

(前のページからの続き)

```
optional arguments:
  -h, --help  show this help message and exit
```

metavar

ArgumentParser がヘルプメッセージを出力するとき、各引数に対してなんらかの参照方法が必要です。デフォルトでは、ArgumentParser オブジェクトは各オブジェクトの ” 名前” として *dest* を利用します。デフォルトでは、位置引数には *dest* の値をそのまま 利用し、オプション引数については *dest* の値を大文字に変換して利用します。このため、1 つの `dest='bar'` である位置引数は `bar` として参照されます。1 つのオプション引数 `--foo` が 1 つのコマンドライン引数を要求するときは、その引数は `FOO` として参照されます。以下に例を示します:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo FOO] bar

positional arguments:
 bar

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO
```

代わりの名前を、`metavar` として指定できます:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo YYY] XXX

positional arguments:
 XXX

optional arguments:
 -h, --help  show this help message and exit
 --foo YYY
```

`metavar` は **表示される** 名前だけを変更することに注意してください。*parse_args()* の返すオブジェクトの属性名は *dest* の値のままです。

`nargs` を指定した場合、metavar が複数回利用されるかもしれません。`metavar` にタプルを渡すと、各引数に対して異なる名前を指定できます:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs=2)
>>> parser.add_argument('--foo', nargs=2, metavar=('bar', 'baz'))
>>> parser.print_help()
usage: PROG [-h] [-x X X] [--foo bar baz]

optional arguments:
 -h, --help     show this help message and exit
 -x X X
 --foo bar baz
```

dest

ほとんどの *ArgumentParser* のアクションは *parse_args()* が返すオブジェクトに対する属性として値を追加します。この属性の名前は *add_argument()* の dest キーワード引数によって決定されます。位置引数のアクションについては、dest は通常 *add_argument()* の第一引数として渡します:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

オプション引数のアクションについては、dest の値は通常オプション文字列から生成されます。*ArgumentParser* は最初の長いオプション文字列を選択し、先頭の -- を除去することで dest の値を生成します。長いオプション文字列が指定されていない場合、最初の短いオプション文字列から先頭の - 文字を除去することで dest を生成します。先頭以外のすべての - 文字は、妥当な属性名になるように _ 文字へ変換されます。次の例はこの動作を示しています:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args('-f 1 -x 2'.split())
Namespace(foo_bar='1', x='2')
>>> parser.parse_args('--foo 1 -y 2'.split())
Namespace(foo_bar='1', x='2')
```

dest にカスタムの属性名を与えることも可能です:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

Action クラス

Acrtion クラスは Action API、すなわちコマンドラインからの引数を処理する呼び出し可能オブジェクトを返す呼び出し可能オブジェクトを実装します。この API に従うあらゆるオブジェクトは `action` 引数として `add_argument()` に渡すことができます。

class argparse.Action(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Action オブジェクトは、コマンドラインからの一つ以上の文字列から単一の引数を解析するのに必要とされる情報を表現するために ArgumentParser によって使われます。Action クラス 2 つの位置引数と、`action` それ自身を除く *ArgumentParser.add_argument()* に渡されるすべてのキーワード引数を受け付けなければなりません。

Action のインスタンス (あるいは `action` 引数に渡す任意の呼び出し可能オブジェクトの返り値) は、属性 "dest", "option_strings", "default", "type", "required", "help", などを定義しなければなりません。これらの属性を定義するのを確実にするためにもっとも簡単な方法は、`Action.__init__` を呼び出すことです。

Action インスタンスは呼び出し可能でなければならず、したがって、サブクラスは 4 つの引数を受け取る `__call__` メソッドをオーバライドしなければなりません:

- `parser` - このアクションを持っている ArgumentParser オブジェクト。
- `namespace` - *parse_args()* が返す *Namespace* オブジェクト。ほとんどのアクションはこのオブジェクトに属性を *setattr()* を使って追加します。
- `values` - 型変換が適用された後の、関連付けられたコマンドライン引数。型変換は *add_argument()* メソッドの *type* キーワード引数で指定されます。
- `option_string` - このアクションを実行したオプション文字列。`option_string` 引数はオプションで、アクションが位置引数に関連付けられた場合は渡されません。

`__call__` メソッドでは任意のアクションを行えますが、典型的にはそれは `dest`, `values` に基づく `namespace` に属性をセットすることでしょう。

16.4.4 parse_args() メソッド

ArgumentParser.parse_args(*args=None*, *namespace=None*)

引数の文字列をオブジェクトに変換し、namespace オブジェクトの属性に代入します。結果の namespace オブジェクトを返します。

事前の *add_argument()* メソッドの呼び出しにより、どのオブジェクトが生成されてどう代入されるかが決定されます。詳細は *add_argument()* のドキュメントを参照してください。

- *args* - 解析する文字列のリスト。デフォルトでは *sys.argv* から取得されます。
- *namespace* - 属性を代入するオブジェクト。デフォルトでは、新しい空の *Namespace* オブジェクトです。

オプション値の文法

parse_args() メソッドは、オプションの値がある場合、そのオプションの値の指定に複数の方法をサポートしています。もっとも単純な場合には、オプションとその値は次のように 2 つの別々の引数として渡されます:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args(['-x', 'X'])
Namespace(foo=None, x='X')
>>> parser.parse_args(['--foo', 'FOO'])
Namespace(foo='FOO', x=None)
```

長いオプション (1 文字よりも長い名前を持ったオプション) では、オプションとその値は次のように = で区切られた 1 つのコマンドライン引数として渡すこともできます:

```
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

短いオプション (1 文字のオプション) では、オプションとその値は次のように連結して渡すことができます:

```
>>> parser.parse_args(['-xX'])
Namespace(foo=None, x='X')
```

最後の 1 つのオプションだけが値を要求する場合、または値を要求するオプションがない場合、複数の短いオプションは次のように 1 つの接頭辞 - だけで連結できます:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', action='store_true')
>>> parser.add_argument('-y', action='store_true')
>>> parser.add_argument('-z')
>>> parser.parse_args(['-xyzZ'])
Namespace(x=True, y=True, z='Z')
```

不正な引数

parse_args() は、コマンドラインの解析中に、曖昧なオプション、不正な型、不正なオプション、位置引数の数の不一致などのエラーを検証します。それらのエラーが発生した場合、エラーメッセージと使用法メッセージを表示して終了します:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')

>>> # invalid type
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'
```

(次のページに続く)

(前のページからの続き)

```
>>> # invalid option
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

>>> # wrong number of arguments
>>> parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
```

- を含む引数

parse_args() メソッドは、ユーザーが明らかなミスをした場合はエラーを表示しますが、いくつか本質的に曖昧な場面があります。例えば、コマンドライン引数 `-1` は、オプションの指定かもしれませんし位置引数かもしれません。*parse_args()* メソッドはこれを次のように扱います: 負の数として解釈でき、パーサーに負の数のように解釈できるオプションが存在しない場合にのみ、`-` で始まる位置引数になりえます:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional argument
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional arguments
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')

>>> # negative number options present, so -1 is an option
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')

>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

>>> # negative number options present, so both -1s are options
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

`-` で始まる位置引数があって、それが負の数として解釈できない場合、ダミーの引数 `'--'` を挿入して、*parse_args()* にそれ以降のすべてが位置引数だと教えることができます:

```
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

引数の短縮形 (先頭文字でのマッチング)

parse_args() メソッドは、デフォルトで、長いオプションに曖昧さがない (先頭の文字が一意である) かぎり、先頭の一文字に短縮して指定できます:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')
>>> parser.parse_args('-bac MMM'.split())
Namespace(bacon='MMM', badger=None)
>>> parser.parse_args('-bad WOOD'.split())
Namespace(bacon=None, badger='WOOD')
>>> parser.parse_args('-ba BA'.split())
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

先頭の文字が同じ引数が複数ある場合に短縮指定を行うとエラーを発生させます。この機能は *allow_abbrev* に `False` を指定することで無効にできます。

sys.argv 以外

ArgumentParser が *sys.argv* 以外の引数を解析できると役に立つ場合があります。その場合は文字列のリストを *parse_args()* に渡します。これはインタラクティブプロンプトからテストするときに便利です:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=range(10),
...     nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(
...     '--sum', dest='accumulate', action='store_const', const=sum,
...     default=max, help='sum the integers (default: find the max)')
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3, 4])
>>> parser.parse_args(['1', '2', '3', '4', '--sum'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```

Namespace オブジェクト

class argparse.Namespace

parse_args() が属性を格納して返すためのオブジェクトにデフォルトで使用されるシンプルなクラスです。

デフォルトでは、*parse_args()* は *Namespace* の新しいオブジェクトに必要な属性を設定して返します。このクラスはシンプルに設計されており、単に読みやすい文字列表現を持った *object* のサブクラスです。もし属性を辞書のように扱える方が良ければ、標準的な Python のイディオム *vars()* を利用できます:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

ArgumentParser が、新しい *Namespace* オブジェクトではなく、既存のオブジェクトに属性を設定する方が良い場合があります。これは namespace= キーワード引数を指定することで可能です:

```
>>> class C:
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

16.4.5 その他のユーティリティ

サブコマンド

ArgumentParser.add_subparsers([*title*][, *description*][, *prog*][, *parser_class*][, *action*][, *option_string*][, *dest*][, *required*][, *help*][, *metavar*])

多くのプログラムは、その機能をサブコマンドへと分割します。例えば svn プログラムは svn checkout, svn update, svn commit などのサブコマンドを利用できます。機能をサブコマンドに分割するのは、プログラムがいくつかの異なった機能を持っていて、それぞれが異なるコマンドライン引数を必要とする場合には良いアイデアです。*ArgumentParser* は *add_subparsers()* メソッドによりサブコマンドを サポートしています。*add_subparsers()* メソッドは通常引数なしに呼び出され、特殊なアクションオブジェクトを返します。このオブジェクトには 1 つのメソッド add_parser() があり、コマンド名と *ArgumentParser* コンストラクターの任意の引数を受け取り、通常の方法で操作できる *ArgumentParser* オブジェクトを返します。

引数の説明:

- title - ヘルプ出力でのサブパーサーグループのタイトルです。デフォルトは、description が指定されている場合は ”subcommands” に、指定されていない場合は位置引数のタイトルになります

- description - ヘルプ出力に表示されるサブパーサーグループの説明です。デフォルトは None になります
- prog - サブコマンドのヘルプに表示される使用方法の説明です。デフォルトではプログラム名と位置引数の後ろに、サブパーサーの引数が続きます
- parser_class - サブパーサーのインスタンスを作成するときに使用されるクラスです。デフォルトでは現在のパーサーのクラス (例: ArgumentParser) になります
- *action* - コマンドラインにこの引数があったときの基本のアクション。
- *dest* - サブコマンド名を格納する属性の名前です。デフォルトは None で値は格納されません
- *required* - サブコマンドが必須であるかどうかを指定し、デフォルトは False です。(3.7 より追加)
- *help* - ヘルプ出力に表示されるサブパーサーグループのヘルプです。デフォルトは None です
- *metavar* - 利用可能なサブコマンドをヘルプ内で表示するための文字列です。デフォルトは None で、サブコマンドを {cmd1, cmd2, ..} のような形式で表します

いくつかの使用例:

```
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true', help='foo help')
>>> subparsers = parser.add_subparsers(help='sub-command help')
>>>
>>> # create the parser for the "a" command
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')
>>>
>>> # create the parser for the "b" command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices='XYZ', help='baz help')
>>>
>>> # parse some argument lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

parse_args() が返すオブジェクトにはメインパーサーとコマンドラインで選択されたサブパーサーによる属性だけが設定されており、選択されなかったサブコマンドのパーサーの属性が設定されていないことに注意してください。このため、上の例では、a コマンドが指定されたときは foo, bar 属性だけが存在し、b コマンドが指定されたときは foo, baz 属性だけが存在しています。

同じように、サブパーサーにヘルプメッセージが要求された場合は、そのパーサーに対するヘルプだけが表示されます。ヘルプメッセージには親パーサーや兄弟パーサーのヘルプメッセージを表示しません。(ただし、各サブパーサーコマンドのヘルプメッセージは、上の例にもあるように add_parser() の help= 引数によって指定できます)

```
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}   sub-command help
    a     a help
    b     b help

optional arguments:
  -h, --help  show this help message and exit
  --foo   foo help

>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar     bar help

optional arguments:
  -h, --help  show this help message and exit

>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

optional arguments:
  -h, --help     show this help message and exit
  --baz {X,Y,Z}  baz help
```

add_subparsers() メソッドは title と description キーワード引数もサポートしています。どちらかが存在する場合、サブパーサーのコマンドはヘルプ出力でそれぞれのグループの中に表示されます。例えば:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
...                                    description='valid subcommands',
...                                    help='additional help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage:  [-h] {foo,bar} ...

optional arguments:
  -h, --help  show this help message and exit

subcommands:
  valid subcommands

  {foo,bar}   additional help
```

さらに、add_parser は aliases 引数もサポートしており、同じサブパーサーに対して複数の文字列で参照することもできます。以下の例では svn のように checkout の短縮形として co を使用できる

ようにしています:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>> checkout = subparsers.add_parser('checkout', aliases=['co'])
>>> checkout.add_argument('foo')
>>> parser.parse_args(['co', 'bar'])
Namespace(foo='bar')
```

サブコマンドを扱う 1 つの便利な方法は *add_subparsers()* メソッドと *set_defaults()* を組み合わせて、各サブパーサーにどの Python 関数を実行するかを教えることです。例えば:

```
>>> # sub-command functions
>>> def foo(args):
...     print(args.x * args.y)
...
>>> def bar(args):
...     print('((%s))' % args.z)
...
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>>
>>> # create the parser for the "foo" command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)
>>>
>>> # create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
2.0
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
((XYZYX))
```

こうすると、*parse_args()* が引数の解析が終わってから適切な関数を呼び出すようになります。このように関数をアクションに関連付けるのは一般的にサブパーサーごとに異なるアクションを扱うもっとも簡単な方法です。ただし、実行されたサブパーサーの名前を確認する必要がある場合は、*add_subparsers()* を呼び出すときに dest キーワードを指定できます:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(dest='subparser_name')
```

(次のページに続く)

(前のページからの続き)

```
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

バージョン 3.7 で変更: 新しい *required* キーワード引数。

FileType オブジェクト

class argparse.**FileType**(*mode='r'*, *bufsize=-1*, *encoding=None*, *errors=None*)

FileType ファクトリは *ArgumentParser.add_argument()* の type 引数に渡すことができるオブジェクトを生成します。type が *FileType* オブジェクトである引数はコマンドライン引数を、指定されたモード、バッファーサイズ、エンコーディング、エラー処理でファイルとして開きます (詳細は *open()* 関数を参照してください。):

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--raw', type=argparse.FileType('wb', 0))
>>> parser.add_argument('out', type=argparse.FileType('w', encoding='UTF-8'))
>>> parser.parse_args(['--raw', 'raw.dat', 'file.txt'])
Namespace(out=<_io.TextIOWrapper name='file.txt' mode='w' encoding='UTF-8'>, raw=<_io.
↪FileIO name='raw.dat' mode='wb'>)
```

FileType オブジェクトは擬似引数 `'-'` を識別し、読み込み用の *FileType* であれば `sys.stdin` を、書き込み用の *FileType* であれば `sys.stdout` に変換します:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>)
```

バージョン 3.4 で追加: *encoding* および *errors* キーワードが追加されました。

引数グループ

ArgumentParser.**add_argument_group**(*title=None*, *description=None*)

デフォルトでは、*ArgumentParser* はヘルプメッセージを表示するときに、コマンドライン引数を ”位置引数” と ”オプション引数” にグループ化します。このデフォルトの動作よりも良い引数のグループ化方法がある場合、*add_argument_group()* メソッドで適切なグループを作成できます:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
>>> parser.print_help()
usage: PROG [--foo FOO] bar
```

(次のページに続く)

(前のページからの続き)

```
group:
  bar    bar help
  --foo FOO  foo help
```

add_argument_group() メソッドは、通常の *ArgumentParser* と同じような *add_argument()* メソッドを持つ引数グループオブジェクトを返します。引数がグループに追加された時、パーサーはその引数を通常の引数のように扱いますが、ヘルプメッセージではその引数を分離されたグループの中に表示します。*add_argument_group()* メソッドには、この表示をカスタマイズするための *title* と *description* 引数があります:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo

group1:
  group1 description

  foo    foo help

group2:
  group2 description

  --bar BAR  bar help
```

ユーザー定義グループにないすべての引数は通常の ”位置引数” と ”オプション引数” セクションに表示されます。

相互排他

ArgumentParser.**add_mutually_exclusive_group**(*required=False*)

相互排他グループを作ります。*argparse* は相互排他グループの中でただ 1 つの引数のみが存在することを確認します:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
```

(次のページに続く)

(前のページからの続き)

```
PROG: error: argument --bar: not allowed with argument --foo
```

add_mutually_exclusive_group() メソッドの引数 *required* に True 値を指定すると、その相互排他引数のどれか 1 つを選ぶことが要求さます:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required
```

現在のところ、相互排他引数グループは *add_argument_group()* の *title* と *description* 引数をサポートしていません。

パーサーのデフォルト値

ArgumentParser.**set_defaults**(***kwargs*)

ほとんどの場合、*parse_args()* が返すオブジェクトの属性はコマンドライン引数の内容と引数のアクションによってのみ決定されます。*set_defaults()* を使うと与えられたコマンドライン引数の内容によらず追加の属性を決定することが可能です:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)
```

パーサーレベルのデフォルト値は常に引数レベルのデフォルト値を上書きします:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')
```

パーサーレベルの default は、複数のパーサーを扱うときに特に便利です。このタイプの例については *add_subparsers()* メソッドを参照してください。

ArgumentParser.**get_default**(*dest*)

add_argument() か *set_defaults()* によって指定された、namespace の属性のデフォルト値を取得します:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='badger')
>>> parser.get_default('foo')
'badger'
```

ヘルプの表示

ほとんどの典型的なアプリケーションでは、*parse_args()* が使用法やエラーメッセージのフォーマットと表示について面倒を見ます。しかし、いくつかのフォーマットメソッドが利用できます:

ArgumentParser.**print_usage**(*file=None*)

ArgumentParser がコマンドラインからどう実行されるべきかの短い説明を表示します。*file* が None の時は、*sys.stdout* に出力されます。

ArgumentParser.**print_help**(*file=None*)

プログラムの使用法と *ArgumentParser* に登録された引数についての情報を含むヘルプメッセージを表示します。*file* が None の時は、*sys.stdout* に出力されます。

これらのメソッドの、表示する代わりにシンプルに文字列を返すバージョンもあります:

ArgumentParser.**format_usage**()

ArgumentParser がコマンドラインからどう実行されるべきかの短い説明を格納した文字列を返します。

ArgumentParser.**format_help**()

プログラムの使用法と *ArgumentParser* に登録された引数についての情報を含むヘルプメッセージを格納した文字列を返します。

部分解析

ArgumentParser.**parse_known_args**(*args=None*, *namespace=None*)

ときどき、スクリプトがコマンドライン引数のいくつかだけを解析し、残りの引数は別のスクリプトやプログラムに渡すことがあります。こういった場合、*parse_known_args()* メソッドが便利です。これは *parse_args()* と同じように動作しますが、余分な引数が存在してもエラーを生成しません。代わりに、評価された namespace オブジェクトと、残りの引数文字列のリストからなる 2 要素タプルを返します。

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

警告: 先頭文字でのマッチング ルールは parse_known_args() にも適用されます。たとえ既知のオプションの先頭文字に過ぎない場合でも、パーサは引数リストに残さずに、オプションを受け取る場合があります。

ファイル解析のカスタマイズ

ArgumentParser.**convert_arg_line_to_args**(*arg_line*)

ファイルから引数を読み込む場合 (*ArgumentParser* コンストラクターの *fromfile_prefix_chars* キーワード引数を参照)、1 行につき 1 つの引数を読み込みます。*convert_arg_line_to_args()* を変更することでこの動作をカスタマイズできます。

このメソッドは、引数ファイルから読まれた文字列である 1 つの引数 *arg_line* を受け取ります。そしてその文字列を解析した結果の引数のリストを返します。このメソッドはファイルから 1 行読みこむごとに、順番に呼ばれます。

このメソッドをオーバーライドすると便利なこととして、スペースで区切られた単語を 1 つの引数として扱えます。次の例でその方法を示します:

```
class MyArgumentParser(argparse.ArgumentParser):
    def convert_arg_line_to_args(self, arg_line):
        return arg_line.split()
```

終了メソッド

ArgumentParser.**exit**(*status=0*, *message=None*)

このメソッドはプログラムを、*status* のステータスで終了させ、指定された場合は *message* を終了前に表示します。ユーザは、この振る舞いを違うものにするために、メソッドをオーバーライドすることができます。

```
class ErrorCatchingArgumentParser(argparse.ArgumentParser):
    def exit(self, status=0, message=None):
        if status:
            raise Exception(f'Exiting because of an error: {message}')
        exit(status)
```

ArgumentParser.**error**(*message*)

このメソッドは *message* を含む使用法メッセージを標準エラーに表示して、終了ステータス 2 でプログラムを終了します。

混在した引数の解析

ArgumentParser.**parse_intermixed_args**(*args=None*, *namespace=None*)

ArgumentParser.**parse_known_intermixed_args**(*args=None*, *namespace=None*)

多くの Unix コマンドは、オプション引数と位置引数を混在させることを許しています。*parse_intermixed_args()* と *parse_known_intermixed_args()* メソッドは、このような方法での解析をサポートしています。

このパーサーは、argparse のすべての機能をサポートしておらず、対応しない機能が使われた場合、例外を送出します。特に、サブパーサーや `argparse.REMAINDER`、位置引数とオプション引数を両方含むような相互排他的なグループは、サポートされていません。

この例は、*parse_known_args()* と *parse_intermixed_args()* の違いを表しています: 前者は ['2', '3'] を、解析されない引数として返し、後者は全ての位置引数を rest に入れて返しています:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('cmd')
>>> parser.add_argument('rest', nargs='*', type=int)
>>> parser.parse_known_args('doit 1 --foo bar 2 3'.split())
(Namespace(cmd='doit', foo='bar', rest=[1]), ['2', '3'])
>>> parser.parse_intermixed_args('doit 1 --foo bar 2 3'.split())
Namespace(cmd='doit', foo='bar', rest=[1, 2, 3])
```

parse_known_intermixed_args() は、解析した内容を含む名前空間と、残りの引数を含んだリストの、2つの要素を持つタプルを返します。*parse_intermixed_args()* は、解析されない引数が残された場合にはエラーを送出します。

バージョン 3.7 で追加.

16.4.6 optparse からのアップグレード

もともと、*argparse* モジュールは *optparse* モジュールとの互換性を保って開発しようと試みられました。しかし、特に新しい nargs= 指定子とより良い使用法メッセージのために必要な変更のために、*optparse* を透過的に拡張することは難しかったのです。*optparse* のほとんどすべてがコピーアンドペーストされたりモンキーパッチを当てられたりしたとき、もはや後方互換性を保とうとすることは現実的ではありませんでした。

argparse モジュールは標準ライブラリ *optparse* モジュールを、以下を含むたくさんの方法で改善しています:

- 位置引数を扱う
- サブコマンドのサポート
- +, / のような代替オプションプレフィクスを許容する
- zero-or-more スタイル、one-or-more スタイルの引数を扱う
- より有益な使用方法メッセージの生成
- カスタム type, カスタム action のために遥かに簡単なインターフェイスを提供する

optparse から *argparse* への現実的なアップグレードパス:

- すべての *optparse.OptionParser.add_option()* の呼び出しを、*ArgumentParser.add_argument()* の呼び出しに置き換える。
- (options, args) = parser.parse_args() を args = parser.parse_args() に置き換え、位置引数のために必要に応じて *ArgumentParser.add_argument()* の呼び出しを追加する。これまで options と呼ばれていたものが、*argparse* では args と呼ばれていることに留意してください。

- *optparse.OptionParser.disable_interspersed_args()* を、*parse_args()* ではなく *parse_intermixed_args()* で置き換える。
- コールバック・アクションと callback_* キーワード引数を type や action 引数に置き換える。
- type キーワード引数に渡していた文字列の名前を、それに応じたオブジェクト (例: int, float, complex, ...) に置き換える。
- optparse.Values を *Namespace* に置き換え、optparse.OptionError と optparse.OptionValueError を ArgumentError に置き換える。
- %default や %prog などの暗黙の引数を含む文字列を、%(default)s や %(prog)s などの、通常の Python で辞書を使う場合のフォーマット文字列に置き換える。
- OptionParser のコンストラクターの version 引数を、parser.add_argument('--version', action='version', version='<the version>') に置き換える

16.5 getopt --- C 言語スタイルのコマンドラインオプションパーサ

ソースコード: Lib/getopt.py

注釈: *getopt* モジュールは、C 言語の getopt() 関数に慣れ親しんだ人ためにデザインされた API を持つコマンドラインオプションのパーサです。getopt() 関数に慣れ親しんでない人や、コードを少なくしてよりよいヘルプメッセージを表示させたい場合は、*argparse* モジュールの使用を検討してください。

このモジュールは sys.argv に入っているコマンドラインオプションの構文解析を支援します。'-' や '--' の特別扱いも含めて、Unix の getopt() と同じ記法をサポートしています。3 番目の引数 (省略可能) を設定することで、GNU のソフトウェアでサポートされているような長形式のオプションも利用できます。

このモジュールは 2 つの関数と 1 つの例外を提供しています:

getopt.**getopt**(*args*, *shortopts*, *longopts=[]*)

コマンドラインオプションとパラメータのリストを構文解析します。*args* は構文解析の対象になる引数のリストです。これは先頭のプログラム名を除いたもので、通常 sys.argv[1:] で与えられます。*shortopts* はスクリプトで認識させたいオプション文字と、引数が必要な場合にはコロン (':') をつけます。つまり Unix の getopt() と同じフォーマットになります。

注釈: GNU の getopt() とは違って、オプションでない引数の後は全てオプションではないと判断されます。これは GNU でない、Unix システムの挙動に近いものです。

---

longopts は長形式のオプションの名前を示す文字列のリストです。名前には、先頭の '--' は含めません。引数が必要な場合には名前の最後に等号 ('=') を入れます。オプション引数はサポートしていません。長形式のオプションだけを受けつけるためには、*shortopts* は空文字列である必要があります。長

形式のオプションは、該当するオプションを一意に決定できる長さまで入力されていれば認識されます。たとえば、*longopts* が `['foo', 'frob']` の場合、`--fo` は `--foo` にマッチしますが、`--f` では一意に決定できないので、*GetoptError* が送出されます。

返り値は 2 つの要素から成っています: 最初は `(option, value)` のタプルのリスト、2 つ目はオプションリストを取り除いたあとに残ったプログラムの引数リストです (*args* の末尾部分のスライスになります)。それぞれの引数と値のタプルの最初の要素は、短形式の時はハイフン 1 つで始まる文字列 (例: `'-x'`)、長形式の時はハイフン 2 つで始まる文字列 (例: `'--long-option'`) となり、引数が 2 番目の要素になります。引数をとらない場合には空文字列が入ります。オプションは見つかった順に並んでいて、複数回同じオプションを指定できます。長形式と短形式のオプションは混在できます。

getopt.gnu_getopt(*args*, *shortopts*, *longopts=[]*)

この関数はデフォルトで GNU スタイルのスキャンモードを使う以外は *getopt()* と同じように動作します。つまり、オプションとオプションでない引数とを混在させることができます。*getopt()* 関数はオプションでない引数を見つけると解析を停止します。

オプション文字列の最初の文字を `'+'` にするか、環境変数 `POSIXLY_CORRECT` を設定することで、オプションでない引数を見つけると解析を停止するように振舞いを変えることができます。

exception getopt.GetoptError

引数リストの中に認識できないオプションがあった場合か、引数が必要なオプションに引数が与えられなかった場合に発生します。例外の引数は原因を示す文字列です。長形式のオプションについては、不要な引数が与えられた場合にもこの例外が発生します。`msg` 属性と `opt` 属性で、エラーメッセージと関連するオプションを取得できます。特に関係するオプションが無い場合には `opt` は空文字列となります。

exception getopt.error

GetoptError へのエイリアスです。後方互換性のために残されています。

Unix スタイルのオプションを使った例です:

```
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

長形式のオプションを使っても同様です:

```
>>> s = '--condition=foo --testing --output-file abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', [
...     'condition=', 'output-file=', 'testing'])
```

(次のページに続く)

(前のページからの続き)

```
>>> optlist
[('--condition', 'foo'), ('--testing', ''), ('--output-file', 'abc.def'), ('-x', '')]
>>> args
['a1', 'a2']
```

スクリプト中での典型的な使い方は以下のようになります:

```
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
    except getopt.GetoptError as err:
        # print help information and exit:
        print(err)  # will print something like "option -a not recognized"
        usage()
        sys.exit(2)
    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
            verbose = True
        elif o in ("-h", "--help"):
            usage()
            sys.exit()
        elif o in ("-o", "--output"):
            output = a
        else:
            assert False, "unhandled option"
    # ...

if __name__ == "__main__":
    main()
```

argparse モジュールを使えば、より良いヘルプメッセージとエラーメッセージを持った同じコマンドラインインタフェースをより少ないコードで実現できます:

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    args = parser.parse_args()
    # ... do something with args.output ...
    # ... do something with args.verbose ..
```

参考:

argparse **モジュール**　別のコマンドラインオプションと引数の解析ライブラリ。

16.6 logging --- Python 用ロギング機能

ソースコード: Lib/logging/__init__.py

Important

このページには、リファレンス情報だけが含まれています。チュートリアルは、以下のページを参照してください

- 基本チュートリアル
- 上級チュートリアル
- ロギングクックブック

このモジュールは、アプリケーションやライブラリのための柔軟なエラーログ記録 (logging) システムを実装するための関数やクラスを定義しています。

標準ライブラリモジュールとしてログ記録 API が提供される利点は、すべての Python モジュールがログ記録に参加できることであり、これによってあなたが書くアプリケーションのログにサードパーティーのモジュールが出力するメッセージを含ませることができます。

このモジュールは、多くの機能性と柔軟性を提供します。ロギングに慣れていないなら、つかむのに一番いいのはチュートリアルを読むことです (右のリンクを参照してください)。

モジュールで定義されている基本的なクラスと関数を、以下に列挙します。

- ロガーは、アプリケーションコードが直接使うインタフェースを公開します。
- ハンドラは、(ロガーによって生成された) ログ記録を適切な送信先に送ります。
- フィルタは、どのログ記録を出力するかを決定する、きめ細かい機能を提供します。
- フォーマッタは、ログ記録が最終的に出力されるレイアウトを指定します。

16.6.1 ロガーオブジェクト

ロガーには以下のような属性とメソッドがあります。ロガーを直接インスタンス化することは **絶対に** してはならず、常にモジュール関数 `logging.getLogger(name)` を介してインスタンス化することに注意してください。同じ name で *getLogger()* を複数回呼び出すと、常に同じロガー・オブジェクトへの参照が返されます。

`name` は `foo.bar.baz` のようにピリオドで分割された (ただし単なるプレーンな `foo` もありえます) 潜在的に階層的な値です。階層リスト中でより下位のロガーは、上位のロガーの子です。例えば、`foo` という名前を持つロガーがあるとき、`foo.bar`, `foo.bar.baz`, `foo.bam` という名前を持つロガーはすべて `foo` の子孫です。ロガー名の階層は Python パッケージ階層と類似していて、推奨される構築方法 `logging.getLogger(__name__)` を使用してロガーをモジュール単位で構成すれば、Python パッケージ階層と同一に

なります。これは、モジュールの中では __name__ が Python パッケージ名前空間におけるモジュール名だからです。

class logging.Logger

propagate
この属性が真と評価された場合、このロガーに記録されたイベントは、このロガーに取り付けられた全てのハンドラに加え、上位 (祖先) ロガーのハンドラにも渡されます。メッセージは、祖先ロガーのハンドラに直接渡されます - 今問題にしている祖先ロガーのレベルもフィルタも、どちらも考慮されません。

この値の評価結果が偽になる場合、ロギングメッセージは祖先ロガーのハンドラに渡されません。

コンストラクタはこの属性を True に設定します。

注釈: ハンドラを、あるロガー **と** その祖先のロガーに接続した場合、同一レコードが複数回発行される場合があります。一般的に、ハンドラを複数のロガーに接続する必要はありません。propagate 設定が True のままになっていれば、ロガーの階層において最上位にある適切なロガーにハンドラを接続するだけで、そのハンドラは全ての子孫ロガーが記録する全てのイベントを確認することができます。一般的なシナリオでは、ハンドラをルートロガーに対してのみ接続し、残りは propagate にすべて委ねます。

---

setLevel(*level*)
このロガーの閾値を *level* に設定します。*level* よりも深刻でないログメッセージは無視されます; 深刻さが *level* 以上のログメッセージは、ハンドラのレベルが *level* より上に設定されていない限り、このロガーに取り付けられているハンドラによって投げられます。

ロガーが生成された際、レベルは NOTSET (これによりすべてのメッセージについて、ロガーがルートロガーであれば処理される、そうでなくてロガーが非ルートロガーの場合には親ロガーに委譲させる) に設定されます。ルートロガーは WARNING レベルで生成されることに注意してください。

「親ロガーに委譲」という用語の意味は、もしロガーのレベルが NOTSET ならば、祖先ロガーの系列の中を NOTSET 以外のレベルの祖先を見つけるかルートに到達するまで辿っていく、ということです。

もし NOTSET 以外のレベルの祖先が見つかったなら、その祖先のレベルが探索を開始したロガーの実効レベルとして扱われ、ログイベントがどのように処理されるかを決めるのに使われます。

ルートに到達した場合、ルートのレベルが NOTSET ならばすべてのメッセージは処理されます。そうでなければルートのレベルが実効レベルとして使われます。

レベルの一覧については **ロギングレベル** を参照してください。

バージョン 3.2 で変更: *level* パラメータは、INFO のような整数定数の代わりに 'INFO' のようなレベルの文字列表現も受け付けるようになりました。ただし、レベルは内部で整数として保存されますし、*getEffectiveLevel()* や *isEnabledFor()* といったメソッドは、整数を返し、また渡

されるものと期待します。

isEnabledFor(*level*)
深刻度が *lvl* のメッセージが、このロガーで処理されることになっているかどうかを示します。このメソッドはまず、logging.disable(level) で設定されるモジュールレベルの深刻度レベルを調べ、次にロガーの実効レベルを *getEffectiveLevel()* で調べます。

getEffectiveLevel()
このロガーの実効レベルを示します。NOTSET 以外の値が *setLevel()* で設定されていた場合、その値が返されます。そうでない場合、NOTSET 以外の値が見つかるまでロガーの階層をルートロガーの方向に追跡します。見つかった場合、その値が返されます。返される値は整数で、典型的には logging.DEBUG, logging.INFO 等のうち一つです。

getChild(*suffix*)
このロガーの子であるロガーを、接頭辞によって決定し、返します。従って、logging.getLogger('abc').getChild('def.ghi') は、logging.getLogger('abc.def.ghi') によって返されるのと同じロガーを返すことになります。これは簡便なメソッドで、親ロガーがリテラルでなく __name__ などを使って名付けられているときに便利です。

バージョン 3.2 で追加.

debug(*msg*, **args*, ***kwargs*)
レベル DEBUG のメッセージをこのロガーで記録します。*msg* はメッセージの書式文字列で、*args* は *msg* に文字列書式化演算子を使って取り込むための引数です。(これは、書式化文字列の中でキーワードを使い、引数として単一の辞書を渡すことができる、ということを意味します。) *args* が提供されない場合は *msg* の % フォーマットは実行されません。

kwargs のうち、*exc_info*, *stack_info*, *stacklevel*, *extra* という 4 つのキーワード引数の中身を調べます。

exc_info は、この値の評価値が false でない場合、例外情報がロギングメッセージに追加されます。もし例外情報をあらわすタプル (*sys.exc_info()* 関数によって戻されるフォーマットにおいて)、または、例外情報をあらわすインスタンスが与えられていれば、それが使用されることになります。それ以外の場合には、*sys.exc_info()* を呼び出して例外情報を取得します。

2 つ目の省略可能なキーワード引数は *stack_info* で、デフォルトは False です。真の場合、実際のロギング呼び出しを含むスタック情報がロギングメッセージに追加されます。これは *exc_info* 指定によって表示されるスタック情報と同じものではないことに注意してください: 前者はカレントスレッド内での、一番下からロギング呼び出しまでのスタックフレームですが、後者は例外に呼応して、例外ハンドラが見つかるところまで巻き戻されたスタックフレームの情報です。

exc_info とは独立に *stack_info* を指定することもできます (例えば、例外が上げられなかった場合でも、コード中のある地点にどのように到着したかを単に示すために)。スタックフレームは、次のようなヘッダー行に続いて表示されます:

```
Stack (most recent call last):
```

これは、例外フレームを表示する場合に使用される Traceback (most recent call last): を

模倣します。

3 番目のオプションキーワード引数は *stacklevel* で、デフォルトは 1 です。もしこれが 1 よりも大きい場合は、*LogRecord* 内で行番号と関数名を算出する時に、指定されたスタックフレームの数をスキップします。これはログヘルパー内部で使われる場合、関数名、ファイル名、行番号はそのヘルパーの情報ではなく、そのヘルパーを呼び出した呼び出し元のものになります。このパラメータの名前は *warnings* モジュールと同じものになります。

4 番目のキーワード引数は *extra* で、当該ログイベント用に作られる *LogRecord* の ___dict___ にユーザー定義属性を加えるのに使われる辞書を渡すために用いられます。これらの属性は好きなように使えます。たとえば、ログメッセージの一部にすることもできます。以下の例を見てください:

```
FORMAT = '%(asctime)-15s %(clientip)s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logger = logging.getLogger('tcpserver')
logger.warning('Protocol problem: %s', 'connection reset', extra=d)
```

これは以下のような出力を行います

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection reset
```

extra で渡される辞書のキーはロギングシステムで使われているものと衝突しないようにしなければなりません。(どのキーがロギングシステムで使われているかについての詳細は *Formatter* のドキュメントを参照してください。)

これらの属性をログメッセージに使うことにしたなら、少し注意が必要です。上の例では、'clientip' と 'user' が *LogRecord* の属性辞書に含まれていることを期待した書式文字列で *Formatter* がセットアップされています。もしこれらが欠けていると、書式化例外が発生してしまうためメッセージはログに残りません。したがってこの場合、常にこれらのキーを含む *extra* 辞書を渡す必要があります。

このようなことは煩わしいかもしれませんが、この機能は限定された場面で使われるように意図しているものなのです。たとえば同じコードがいくつものコンテキストで実行されるマルチスレッドのサーバで、興味のある条件が現れるのがそのコンテキストに依存している (上の例で言えば、リモートのクライアント IP アドレスや認証されたユーザ名など)、というような場合です。そういった場面では、それ用の *Formatter* が特定の *Handler* と共に使われるというのはよくあることです。

バージョン 3.2 で変更: *stack_info* パラメータが追加されました。

バージョン 3.5 で変更: *exc_info* パラメータは例外インスタンスを受け入れることが可能です。

バージョン 3.8 で変更: *stacklevel* 引数が追加されました。

info(*msg*, **args*, ***kwargs*)

レベル INFO のメッセージをこのロガーで記録します。引数は *debug()* と同じように解釈されます。

warning(*msg*, **args*, ***kwargs*)

レベル WARNING のメッセージをこのロガーで記録します。引数は *debug()* と同じように解釈されます。

注釈: warning と機能的に等価な古いメソッド warn があります。warn は廃止予定なので使わないでください - 代わりに warning を使ってください。

---

error(*msg*, **args*, ***kwargs*)

レベル ERROR のメッセージをこのロガーで記録します。引数は *debug()* と同じように解釈されます。

critical(*msg*, **args*, ***kwargs*)

レベル CRITICAL のメッセージをこのロガーで記録します。引数は *debug()* と同じように解釈されます。

log(*level*, *msg*, **args*, ***kwargs*)

整数で表したレベル *level* のメッセージをこのロガーで記録します。その他の引数は *debug()* と同じように解釈されます。

exception(*msg*, **args*, ***kwargs*)

レベル ERROR のメッセージをこのロガーで記録します。引数は *debug()* と同じように解釈されます。例外情報がログメッセージに追加されます。このメソッドは例外ハンドラからのみ呼び出されるべきです。

addFilter(*filter*)

指定されたフィルタ *filter* をこのロガーに追加します。

removeFilter(*filter*)

指定されたフィルタ *filter* をこのロガーから取り除きます。

filter(*record*)

レコードに対してこのロガーのフィルタを適用し、レコードが処理されるべき場合に True を返します。フィルタのいずれかの値が偽を返すまで、それらは順番に試されていきます。いずれも偽を返さなければ、レコードは処理される (ハンドラに渡される) ことになります。ひとつでも偽を返せば、発生したレコードはもはや処理されることはありません。

addHandler(*hdlr*)

指定されたハンドラ *hdlr* をこのロガーに追加します。

removeHandler(*hdlr*)

指定されたハンドラ *hdlr* をこのロガーから取り除きます。

findCaller(*stack_info=False*, *stacklevel=1*)

呼び出し元のソースファイル名と行番号を調べます。ファイル名と行番号、関数名、スタック情報を 4 要素のタプルで返します。*stack_info* が True でなければ、スタック情報は None が返されます。

stacklevel パラメータは *debug()* や他の API を呼び出すコードから渡されます。もしこれが 1 よりも大きい場合は、その超過分は返す値を決定する前にスタックフレームをスキップする数として利用されます。これは通常、ログ API をヘルパーやラッパー経由で呼び出す場合に便利です。こうすることで、イベントログに記録される情報はヘルパーやラッパーのコードではなく、それらを呼び出しているコードのものとなります。

handle(*record*)
レコードを、このロガーおよびその上位ロガー (ただし *propagate* の値が false になったところまで) に関連付けられているすべてのハンドラに渡して処理します。このメソッドは、ローカルで生成されたレコードだけでなく、ソケットから受信した unpickle されたレコードに対しても同様に用いられます。*filter()* によって、ロガーレベルでのフィルタが適用されます。

makeRecord(*name*, *level*, *fn*, *lno*, *msg*, *args*, *exc_info*, *func=None*, *extra=None*, *sinfo=None*)
このメソッドは、特殊な *LogRecord* インスタンスを生成するためにサブクラスでオーバライドできるファクトリメソッドです。

hasHandlers()
このロガーにハンドラが設定されているかどうかを調べます。そのために、このロガーとロガー階層におけるその祖先についてハンドラ探していきます。ハンドラが見つかれば `True` 、そうでなければ `False` を返します。このメソッドは、'propagate' 属性が偽に設定されたロガーを見つけると、さらに上位の探索をやめます - そのロガーが、ハンドラが存在するかどうかチェックされる最後のロガー、という意味です。

バージョン 3.2 で追加.

バージョン 3.7 で変更: ロガーの pickle 化と unpickle 化ができるようになりました。

16.6.2 ロギングレベル

ログレベルの数値は以下の表のように与えられています。これらは基本的に自分でレベルを定義したい人のためのもので、定義するレベルを既存のレベルの間に位置づけるためには具体的な値が必要になります。もし数値が他のレベルと同じだったら、既存の値は上書きされその名前は失われます。

レベル	数値
`CRITICAL`	50
`ERROR`	40
`WARNING`	30
`INFO`	20
`DEBUG`	10
`NOTSET`	0

16.6.3 ハンドラオブジェクト

ハンドラ (Handler) は以下の属性とメソッドを持ちます。*Handler* は直接インスタンス化されることはありません; このクラスはより便利なサブクラスの基底クラスとして働きます。しかしながら、サブクラスにおける `__init__()` メソッドでは、*Handler.__init__()* を呼び出す必要があります。

class logging.**Handler**

__init__(*level=NOTSET*)
レベルを設定して、*Handler* インスタンスを初期化します。空のリストを使ってフィルタを設定し、I/O 機構へのアクセスを直列化するために (*createLock()* を使って) ロックを生成します。

createLock()
スレッドセーフでない背後の I/O 機能に対するアクセスを直列化するために用いられるスレッドロック (thread lock) を初期化します。

acquire()
createLock() で生成されたスレッドロックを獲得します。

release()
acquire() で獲得したスレッドロックを解放します。

setLevel(*level*)
このハンドラに対する閾値を *level* に設定します。*level* よりも深刻でないログメッセージは無視されます。ハンドラが生成された際、レベルは `NOTSET` (すべてのメッセージが処理される) に設定されます。

レベルの一覧については **ロギングレベル** を参照してください。

バージョン 3.2 で変更: *level* パラメータは、`INFO` のような整数定数の代わりに 'INFO' のようなレベルの文字列表現も受け付けるようになりました。

setFormatter(*fmt*)
このハンドラのフォーマッタを *fmt* に設定します。

addFilter(*filter*)
指定されたフィルタ *filter* をこのハンドラに追加します。

removeFilter(*filter*)
指定されたフィルタ *filter* をこのハンドラから除去します。

filter(*record*)
レコードに対してこのハンドラのフィルタを適用し、レコードが処理されるべき場合に `True` を返します。フィルタのいずれかの値が偽を返すまで、それらは順番に試されていきます。いずれも偽を返さなければ、レコードは発行されることになります。ひとつでも偽を返せば、ハンドラはレコードを発行しません。

flush()
すべてのログ出力がフラッシュされるようにします。このクラスのバージョンではなにも行わず、

サブクラスで実装するためのものです。

close()
ハンドラで使われているすべてのリソースの後始末を行います。このバージョンでは何も出力せず、*shutdown()* が呼ばれたときに閉じられたハンドラを内部リストから削除します。サブクラスではオーバライドされた *close()* メソッドからこのメソッドが必ず呼ばれるようにしてください。

handle(*record*)
ハンドラに追加されたフィルタの条件に応じて、指定されたログレコードを出力します。このメソッドは I/O スレッドロックの獲得/解放を伴う実際のログ出力をラップします。

handleError(*record*)
このメソッドは *emit()* の呼び出し中に例外に遭遇した際にハンドラから呼び出されます。モジュールレベル属性 `raiseExceptions` が `False` の場合、例外は暗黙のまま無視されます。ほとんどの場合、これがロギングシステムの望ましい動作です - というのは、ほとんどのユーザはロギングシステム自体のエラーは気にせず、むしろアプリケーションのエラーに興味があるからです。しかしながら、望むならこのメソッドを自作のハンドラと置き換えることもできます。*record* には、例外発生時に処理されていたレコードが入ります。(*raiseExceptions* のデフォルト値は `True` です。これは開発中はその方が便利だからです)。

format(*record*)
レコードに対する書式化を行います - フォーマッタが設定されていれば、それを使います。そうでない場合、モジュールにデフォルト指定されたフォーマッタを使います。

emit(*record*)
指定されたログ記録レコードを実際にログ記録する際のすべての処理を行います。このメソッドはサブクラスで実装されることを意図しており、そのためこのクラスのバージョンは *NotImplementedError* を送出します。

標準として含まれているハンドラについては、*logging.handlers* を参照してください。

16.6.4 フォーマッタオブジェクト

Formatter オブジェクトは以下の属性とメソッドを持っています。*Formatter* は *LogRecord* を (通常は) 人間か外部のシステムで解釈できる文字列に変換する役割を担っています。基底クラスの *Formatter* では書式文字列を指定することができます。何も指定されなかった場合、ロギングコール中のメッセージ以外の情報だけを持つ `'%(message)s'` の値が使われます。フォーマットされた出力に情報の要素 (タイムスタンプなど) を追加したいなら、このまま読み進めてください。

Formatter は *LogRecord* 属性の知識を利用できるような書式文字列を用いて初期化することができます。例えば、上で言及したデフォルト値では、ユーザによるメッセージと引数はあらかじめフォーマットされて、*LogRecord* の *message* 属性に入っていることを利用しています。この書式文字列は、Python 標準の % を使った変換文字列で構成されます。文字列整形に関する詳細は *printf* **形式の文字列書式化** を参照してください。

LogRecord の便利なマッピングキーは、*LogRecord* **属性** の節で与えられます。

class logging.Formatter(*fmt=None*, *datefmt=None*, *style='%'*)

Formatter クラスの新たなインスタンスを返します。インスタンスは全体としてのメッセージに対する書式文字列と、メッセージの日付/時刻部分のための書式文字列を伴って初期化されます。*fmt* が指定されない場合、`'%(message)s'` が使われます。*datefmt* が指定されない場合、*formatTime()* ドキュメントで解説されている書式が使われます。

style パラメータは '%', '{', '$' のうちのいずれかで、書式文字列がどのようにデータとマージされるかを決めます: %-format 、*str.format()* 、*string.Template* のうちのどれかが使用されます。ログメッセージに使用する { および $ 形式のフォーマットの情報は formatting-styles を参照してください。

バージョン 3.2 で変更: *style* パラメータが追加されました。

バージョン 3.8 で変更: *validate* パラメータが追加されました。style と fmt が不正だったりミスマッチだった場合に `ValueError` を送出します。例: `logging.Formatter('%(asctime)s - %(message)s', style='{')` 。

format(*record*)

レコードの属性辞書が、文字列を書式化する演算で被演算子として使われます。書式化された結果の文字列を返します。辞書を書式化する前に、二つの準備段階を経ます。レコードの *message* 属性が *msg* % *args* を使って処理されます。書式化された文字列が `'(asctime)'` を含むなら、*formatTime()* が呼び出され、イベントの発生時刻を書式化します。例外情報が存在する場合、*formatException()* を使って書式化され、メッセージに追加されます。ここで注意していただきたいのは、書式化された例外情報は *exc_text* にキャッシュされるという点です。これが有用なのは例外情報がピックル化されて回線上を送ることができるからですが、しかし二つ以上の *Formatter* サブクラスで例外情報の書式化をカスタマイズしている場合には注意が必要になります。この場合、フォーマッタが書式化を終えるごとにキャッシュをクリアして、次のフォーマッタがキャッシュされた値を使わずに新鮮な状態で再計算するようにしなければならないことになります。

スタック情報が利用可能な場合、(必要ならば *formatStack()* を使って整形した上で) スタック情報が例外情報の後に追加されます。

formatTime(*record*, *datefmt=None*)

このメソッドは、フォーマッタが書式化された時間を利用したい際に、*format()* から呼び出されます。このメソッドは特定の要求を提供するためにフォーマッタで上書きすることができますが、基本的な振る舞いは以下のようになります: *datefmt* (文字列) が指定された場合、レコードが生成された時刻を書式化するために *time.strftime()* で使われます。そうでない場合、'%Y-%m-%d %H:%M:%S,uuu' というフォーマットが使われます。uuu 部分はミリ秒値で、それ以外の文字は *time.strftime()* ドキュメントに従います。このフォーマットの時刻の例は `2003-01-23 00:29:50,411` です。結果の文字列が返されます。

この関数は、ユーザが設定できる関数を使って、生成時刻をタプルに変換します。デフォルトでは、*time.localtime()* が使われます。特定のフォーマッタインスタンスに対してこれを変更するには、`converter` 属性を *time.localtime()* や *time.gmtime()* と同じ署名をもつ関数に設定してください。すべてのフォーマッタインスタンスに対してこれを変更するには、例えば全てのロギング時刻を GMT で表示するには、`Formatter` クラスの `converter` 属性を設定してください。

バージョン 3.3 で変更: 以前は、デフォルトのフォーマットがこの例のようにハードコーディングされていました: 2010-09-06 22:38:15,292 ここで、コンマの前の部分は strptime フォーマット文字列 (`'%Y-%m-%d %H:%M:%S'`) によって扱われる部分で、コンマの後の部分はミリ秒値です。strptime にミリ秒のフォーマットプレースホルダーがないので、ミリ秒値は別のフォーマット文字列 `'%s,%03d'` を使用して追加されます。そして、これらのフォーマット文字列は両方ともこのメソッドでハードコーディングされていました。変更後は、これらの文字列はクラスレベル属性として定義され、必要ならインスタンスレベルでオーバーライドすることができます。属性の名前は `default_time_format` (strptime 書式文字列用) と `default_msec_format` (ミリ秒値の追加用) です。

formatException(*exc_info*)

指定された例外情報 (*sys.exc_info()* が返すような標準例外のタプル) を文字列として書式化します。デフォルトの実装は単に *traceback.print_exception()* を使います。結果の文字列が返されます。

formatStack(*stack_info*)

指定されたスタック情報を文字列としてフォーマットします (*traceback.print_stack()* によって返される文字列ですが、最後の改行が取り除かれています)。このデフォルト実装は、単に入力値をそのまま返します。

16.6.5 フィルタオブジェクト

フィルタ (`Filter`) は、**ハンドラ** や **ロガー** によって使われ、レベルによって提供されるのよりも洗練されたフィルタリングを実現します。基底のフィルタクラスは、ロガー階層構造内の特定地点の配下にあるイベントだけを許可します。例えば、'A.B' で初期化されたフィルタは、ロガー 'A.B', 'A.B.C', 'A.B.C.D', 'A.B.D' 等によって記録されたイベントは許可しますが、'A.BB', 'B.A.B' などは許可しません。空の文字列で初期化された場合、すべてのイベントを通過させます。

class logging.**Filter**(*name=''*)

Filter クラスのインスタンスを返します。*name* が指定されていれば、*name* はロガーの名前を表します。指定されたロガーとその子ロガーのイベントがフィルタを通過できるようになります。*name* が指定されなければ、すべてのイベントを通過させます。

filter(*record*)

指定されたレコードがログされるべきか？ no ならばゼロを、yes ならばゼロでない値を返します。適切と判断されれば、このメソッドによってレコードはその場で修正されることがあります。

ハンドラに対するフィルタはハンドラがイベントを発行する前に試され、一方ではロガーに対するフィルタは、イベントが (*debug()*, *info()* などによって) ロギングされる際には、ハンドラにイベントが送信される前にはいつでも試されることに注意してください。そのフィルタがそれら子孫ロガーにも適用されていない限り、子孫ロガーによって生成されたイベントはロガーのフィルタ設定によってフィルタされることはありません。

実際には、`Filter` をサブクラス化する必要はありません。同じ意味の `filter` メソッドを持つ、すべてのインスタンスを通せます。

バージョン 3.2 で変更: 特殊な Filter クラスを作ったり、filter メソッドを持つ他のクラスを使う必要はありません: 関数 (あるいは他の callable) をフィルタとして使用することができます。フィルタロジックは、フィルタオブジェクトが filter 属性を持っているかどうかチェックします: もし filter 属性を持っていたら、それは Filter であると仮定され、その *filter()* メソッドが呼び出されます。そうでなければ、それは callable であると仮定され、レコードを単一のパラメータとして呼び出されます。返される値は *filter()* によって返されるものと一致すべきです。

フィルタは本来、レコードをレベルよりも洗練された基準に基づいてフィルタするために使われますが、それが取り付けられたハンドラやロガーによって処理されるレコードをすべて監視します。これは、特定のロガーやハンドラに処理されたレコードの数を数えたり、処理されている *LogRecord* の属性を追加、変更、削除したりするときに便利です。もちろん、LogRecord を変更するには注意が必要ですが、これにより、ログにコンテキスト情報を注入できます (filters-contextual を参照してください)。

16.6.6 LogRecord オブジェクト

LogRecord インスタンスは、何かをログ記録するたびに *Logger* によって生成されます。また、*makeLogRecord()* を通して (例えば、ワイヤを通して受け取られた pickle 化されたイベントから) 手動で生成することも出来ます。

class logging.**LogRecord**(*name*, *level*, *pathname*, *lineno*, *msg*, *args*, *exc_info*, *func=None*, *sinfo=None*)

ロギングされているイベントに適切なすべての情報を含みます。

基本的な情報は msg と args に渡され、レコードの message フィールドは msg % args による結合で生成されます。

パラメータ

- **name** -- この LogRecord で表されるイベントをロギングするのに使われるロガーの名前です。ここで与える名前が、たとえ他の (祖先の) ロガーに結び付けられたハンドラによって発せられるとしても、与えたこの値のままであることに注意してください。
- **level** -- このロギングイベントの数値のレベル (DEBUG, INFO などのいずれか) です。なお、これは LogRecord の *2* **つの** 属性に変換されます。数値 levelno と、対応するレベル名 levelname です。
- **pathname** -- ロギングの呼び出しが発せられたファイルの完全なパス名。
- **lineno** -- ロギングの呼び出しが発せられたソース行番号。
- **msg** -- イベント記述メッセージで、これは変数データのプレースホルダを持つフォーマット文字列になり得ます。
- **args** -- *msg* 引数と組み合わせてイベント記述を得るための変数データです。
- **exc_info** -- 現在の例外情報を含む例外タプルか、利用できる例外情報がない場合は None です。
- **func** -- ロギングの呼び出しを行った関数またはメソッドの名前です。

- **sinfo** -- 現在のスレッドのスタックベースからログ呼び出しまでの間のスタック情報を表わすテキスト文字列。

getMessage()

ユーザが提供した引数をメッセージに交ぜた後、この *LogRecord* インスタンスへのメッセージを返します。ユーザがロギングの呼び出しに与えた引数が文字列でなければ、その引数に *str()* が呼ばれ、文字列に変換されます。これにより、__str__ メソッドが実際のフォーマット文字列を返せるようなユーザ定義のクラスをメッセージとして使えます。

バージョン 3.2 で変更: *LogRecord* の生成は、レコードを生成するために使用されるファクトリを提供することにより、さらに設定可能になりました。ファクトリは *getLogRecordFactory()* と *setLogRecordFactory()* を使用して設定することができます (ファクトリのシグネチャに関しては *setLogRecordFactory()* を参照)。

この機能を使うと *LogRecord* の生成時に独自の値を注入することができます。次のパターンが使えます:

```
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record

logging.setLogRecordFactory(record_factory)
```

このパターンでは複数のファクトリをつなぐこともできます。それらが互いの属性を上書きしたりせず、また上にリストされた標準属性を意図せず上書きしたりしない限り、驚くようなことは何も起こりません (there should be no surprises)。

16.6.7 LogRecord 属性

LogRecord には幾つかの属性があり、そのほとんどはコンストラクタの引数から得られます。(なお、LogRecord コンストラクタの引数と LogRecord 属性が常に厳密に対応するわけではありません。) これらの属性は、レコードからのデータをフォーマット文字列に統合するのに使えます。以下のテーブルに、属性名、意味、そして % 形式フォーマット文字列における対応するプレースホルダを (アルファベット順に) 列挙します。

{}-フォーマット (*str.format()*) を使用していれば、書式文字列の中でプレースホールダーとして {attrname} を使うことができます。$-フォーマット (*string.Template*) を使用している場合は、${attrname} 形式にしてください。もちろん、両方の場合で attrname は使用したい実際の属性名に置き換えてください。

{}-フォーマットの場合には、属性名の後にフォーマットフラグを指定することができます。属性名とフォーマットフラグの間はコロンで分割します。例: プレースホールダー {msecs:03d} は、ミリセカンド値 4 を 004 としてフォーマットします。利用可能なオプション上の全詳細に関しては *str.format()* ドキュメンテーションを参照してください。

属性名	フォーマット	説明
args	このフォーマットを自分で使う必要はないでしょう。	msg に組み合わせて message を生成するための引数のタプル、または、マージに用いられる辞書 (引数が一つしかなく、かつそれが辞書の場合)。
asctime	%(asctime)s	*LogRecord* が生成された時刻を人間が読める書式で表したもの。デフォルトでは ”2003-07-08 16:49:45,896” 形式 (コンマ以降の数字は時刻のミリ秒部分) です。
created	%(created)f	*LogRecord* が生成された時刻 (*time.time()* によって返される形式で)。
exc_info	このフォーマットを自分で使う必要はないでしょう。	(sys.exc_info 風の) 例外タプルか、例外が起こっていない場合は None。
ファイル名	%(filename)s	pathname のファイル名部分。
funcName	%(funcName)s	ロギングの呼び出しを含む関数の名前。
levelname	%(levelname)s	メッセージのための文字のロギングレベル ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL')。
levelno	%(levelno)s	メッセージのための数値のロギングレベル (DEBUG, INFO, WARNING, ERROR, CRITICAL)。
lineno	%(lineno)d	ロギングの呼び出しが発せられたソース行番号 (利用できる場合のみ)。
message	%(message)s	msg % args として求められた、ログメッセージ。*Formatter.format()* が呼び出されたときに設定されます。
module	%(module)s	モジュール (filename の名前部分)。
msecs	%(msecs)d	*LogRecord* が生成された時刻のミリ秒部分。
msg	このフォーマットを自分で使う必要はないでしょう。	元のロギングの呼び出しで渡されたフォーマット文字列。args と合わせて、message 、または任意のオブジェクトを生成します (arbitrary-object-messages 参照)。
name	%(name)s	ロギングに使われたロガーの名前。
pathname	%(pathname)s	ロギングの呼び出しが発せられたファイルの完全なパス名 (利用できる場合のみ)。
process	%(process)d	プロセス ID (利用可能な場合のみ)。
processName	%(processName)s	プロセス名 (利用可能な場合のみ)。
relativeCreated	%(relativeCreated)d	logging モジュールが読み込まれた時刻に対する、LogRecord が生成された時刻を、ミリ秒で表したもの。
stack_info	このフォーマットを自分で使う必要はないでしょう。	現在のスレッドでのスタックの底からこのレコードの生成に帰着したログ呼び出しまでのスタックフレーム情報 (利用可能な場合)。
thread	%(thread)d	スレッド ID (利用可能な場合のみ)。
threadName	%(threadName)s	スレッド名 (利用可能な場合のみ)。

バージョン 3.1 で変更: *processName* が追加されました。

16.6.8 LoggerAdapter オブジェクト

LoggerAdapter インスタンスは文脈情報をログ記録呼び出しに渡すのを簡単にするために使われます。使い方の例は コンテキスト情報をログ記録出力に付加する を参照してください。

class logging.**LoggerAdapter**(*logger*, *extra*)

内部で使う *Logger* インスタンスと辞書風 (dict-like) オブジェクトで初期化した *LoggerAdapter* のインスタンスを返します。

process(*msg*, *kwargs*)

文脈情報を挿入するために、ログ記録呼び出しに渡されたメッセージおよび/またはキーワード引数に変更を加えます。ここでの実装は *extra* としてコンストラクタに渡されたオブジェクトを取り、'extra' キーを使って *kwargs* に加えます。返り値は (*msg*, *kwargs*) というタプルで、(変更されているはずの) 渡された引数を含みます。

LoggerAdapter は上記に加え *Logger* のメソッド *debug()*, *info()*, *warning()*, *error()*, *exception()*, *critical()*, *log()*, *isEnabledFor()*, *getEffectiveLevel()*, *setLevel()*, *hasHandlers()* をサポートします。これらは *Logger* の対応するメソッドと同じシグニチャを持つため、2 つのインスタンスは区別せずに利用出来ます。

バージョン 3.2 で変更: *isEnabledFor()*, *getEffectiveLevel()*, *setLevel()*, *hasHandlers()* が *LoggerAdapter* に追加されました。これらメソッドは元のロガーに処理を委譲します。

16.6.9 スレッドセーフ性

logging モジュールは、クライアントで特殊な作業を必要としない限りスレッドセーフになっています。このスレッドセーフ性はスレッドロックによって達成されています; モジュールの共有データへのアクセスを直列化するためのロックが一つ存在し、各ハンドラでも背後にある I/O へのアクセスを直列化するためにロックを生成します。

signal モジュールを使用して非同期シグナルハンドラを実装している場合、そのようなハンドラからはログ記録を使用できないかもしれません。これは、*threading* モジュールにおけるロック実装が常にリエントラントではなく、そのようなシグナルハンドラから呼び出すことができないからです。

16.6.10 モジュールレベルの関数

上で述べたクラスに加えて、いくつかのモジュールレベルの関数が存在します。

logging.**getLogger**(*name=None*)

指定された名前のロガーを返します。名前が None であれば、ロガー階層のルート (root) にあるロガーを返します。*name* を指定する場合には、通常は *'a'*, *'a.b'*, *'a.b.c.d'* といったドット区切りの階層的な名前にします。名前の付け方はログ機能を使う開発者次第です。

与えられた名前に対して、この関数はどの呼び出しでも同じロガーインスタンスを返します。したがって、ロガーインスタンスをアプリケーションの各部でやりとりする必要はありません。

logging.**getLoggerClass**()

標準の *Logger* クラスか、最後に *setLoggerClass()* に渡したクラスを返します。この関数は、新たなクラス定義の中で呼び出して、カスタマイズした *Logger* クラスのインストールが既に他のコードで適用したカスタマイズを取り消さないことを保証するために使われることがあります。例えば以下のようにします:

```
class MyLogger(logging.getLoggerClass()):
    # ... override behaviour here
```

logging.**getLogRecordFactory**()

LogRecord を生成するのに使われる callable を返します。

バージョン 3.2 で追加: この関数は、ログイベントを表現する *LogRecord* の構築方法に関して開発者により多くのコントロールを与えるため、*setLogRecordFactory()* とともに提供されました。

このファクトリがどのように呼ばれるかに関する詳細は *setLogRecordFactory()* を参照してください。

logging.**debug**(*msg*, **args*, ***kwargs*)

レベル DEBUG のメッセージをルートロガーで記録します。*msg* はメッセージの書式文字列で、*args* は *msg* に文字列書式化演算子を使って取り込むための引数です。(これは、書式文字列の中でキーワードを使い、引数として単一の辞書を渡すことができる、ということを意味します。)

キーワード引数 *kwargs* からは 3 つのキーワードが調べられます。一つ目は *exc_info* で、この値の評価値が false でない場合、例外情報をログメッセージに追加します。(*sys.exc_info()* の返す形式の) 例外情報を表すタプルや例外インスタンスが与えられていれば、それをメッセージに使います。それ以外の場合には、*sys.exc_info()* を呼び出して例外情報を取得します。

2 つ目の省略可能なキーワード引数は *stack_info* で、デフォルトは False です。真の場合、実際のロギング呼び出しを含むスタック情報がロギングメッセージに追加されます。これは *exc_info* 指定によって表示されるスタック情報と同じものではないことに注意してください: 前者はカレントスレッド内での、一番下からロギング呼び出しまでのスタックフレームですが、後者は例外に呼応して、例外ハンドラが見つかるところまで巻き戻されたスタックフレームの情報です。

exc_info とは独立に *stack_info* を指定することもできます (例えば、例外が上げられなかった場合でも、コード中のある地点にどのように到着したかを単に示すために)。スタックフレームは、次のようなヘッダー行に続いて表示されます:

```
Stack (most recent call last):
```

これは、例外フレームを表示する場合に使用される `Traceback (most recent call last):` を模倣します。

3 番目のキーワード引数は *extra* で、当該ログイベント用に作られる LogRecoed の ___dict___ にユーザー定義属性を加えるのに使われる辞書を渡すために用いられます。これらの属性は好きなように使えます。たとえば、ログメッセージの一部にすることもできます。以下の例を見てください:

```
FORMAT = '%(asctime)-15s %(clientip)s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logging.warning('Protocol problem: %s', 'connection reset', extra=d)
```

これは以下のような出力を行います:

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection reset
```

extra で渡される辞書のキーはロギングシステムで使われているものと衝突しないようにしなければなりません。(どのキーがロギングシステムで使われているかについての詳細は *Formatter* のドキュメントを参照してください。)

これらの属性をログメッセージに使うことにしたなら、少し注意が必要です。上の例では、'clientip' と 'user' が LogRecord の属性辞書に含まれていることを期待した書式文字列で *Formatter* がセットアップされています。もしこれらが欠けていると、書式化例外が発生してしまうためメッセージはログに残りません。したがってこの場合、常にこれらのキーを含む *extra* 辞書を渡す必要があります。

このようなことは煩わしいかもしれませんが、この機能は限定された場面で使われるように意図しているものなのです。たとえば同じコードがいくつものコンテキストで実行されるマルチスレッドのサーバで、興味のある条件が現れるのがそのコンテキストに依存している (上の例で言えば、リモートのクライアント IP アドレスや認証されたユーザ名など)、というような場合です。そういった場面では、それ用の *Formatter* が特定の *Handler* と共に使われるというのはよくあることです。

バージョン 3.2 で変更: *stack_info* パラメータが追加されました。

logging.**info**(*msg*, **args*, ***kwargs*)

レベル INFO のメッセージをルートロガーで記録します。引数は *debug()* と同じように解釈されます。

logging.**warning**(*msg*, **args*, ***kwargs*)

レベル WARNING のメッセージをルートロガーで記録します。引数は *debug()* と同じように解釈されます。

注釈: warning と機能的に等価な古い関数 warn があります。warn は廃止予定なので使わないでください - 代わりに warning を使ってください。

---

logging.**error**(*msg*, **args*, ***kwargs*)

レベル ERROR のメッセージをルートロガーで記録します。引数は *debug()* と同じように解釈されます。

logging.**critical**(*msg*, **args*, ***kwargs*)

レベル CRITICAL のメッセージをルートロガーで記録します。引数は *debug()* と同じように解釈されます。

logging.**exception**(*msg*, **args*, ***kwargs*)

レベル ERROR のメッセージをルートロガーで記録します。引数は *debug()* と同じように解釈されます。例外情報がログメッセージに追加されます。このメソッドは例外ハンドラからのみ呼び出されます。

logging.**log**(*level*, *msg*, **args*, ***kwargs*)

レベル *level* のメッセージをルートロガーで記録します。その他の引数は *debug()* と同じように解釈されます。

注釈: 上述の便利なルートロガーに処理を委譲するモジュールレベル関数は *basicConfig()* を呼び出して、少なくとも 1 つのハンドラが利用できることを保証します。これにより Python の 2.7.1 以前や 3.2 以前のバージョンでは、スレッドが開始される **前に** 少なくともひとつのハンドラがルートロガーに加えられるのでない限り、スレッド内で使うべき **ではありません** 。以前のバージョンの Python では、*basicConfig()* のスレッドセーフ性の欠陥により、(珍しい状況下とはいえ) ハンドラがルートロガーに複数回加えられることがあり、ログ内のメッセージが重複するという予期しない結果をもたらすことがあります。

---

logging.**disable**(*level=CRITICAL*)

全てのロガーのレベル *level* を上書きし、これはロガー自身の出力レベルよりも優先されます。アプリケーション全体を横断するログ出力を一時的に調整する必要が生じたら、この関数は便利でしょう。これの効果は重大度 *level* 以下の全てのロギング呼び出しを無効にすることですので、INFO で呼び出しをすれば、INFO と DEBUG イベントが捨てられる一方で、重大度 WARNING 以上のものは、ロガーの有効レベルに基いて処理されます。`logging.disable(logging.NOTSET)` が呼び出されると、この上書きレベルは削除され、ログ出力は再び個々のロガーの有効レベルに依存するようになります。

`CRITICAL` より高い独自のログレベル (これは推奨されません) を定義した場合は、*level* 引数のデフォルト値を当てにできなくなり、適切な値を明示的に与える必要があります。

バージョン 3.7 で変更: *level* 引数のデフォルトが `CRITICAL` レベルになりました。この変更についてのより詳しいことは bpo-28524 を参照してください。

logging.**addLevelName**(*level*, *levelName*)

内部的な辞書の中でレベル *level* をテキスト *levelName* に関連付けます。これは例えば *Formatter* でメッセージを書式化する際のように、数字のレベルをテキスト表現に対応付ける際に用いられます。この関数は自作のレベルを定義するために使うこともできます。使われるレベルに対する唯一の制限は、レベルは正の整数でなくてはならず、メッセージの深刻度が上がるに従ってレベルの数も上がらなくてはならないということです。

注釈: 独自のレベルを定義したい場合、custom-levels のセクションを参照してください。

---

logging.**getLevelName**(*level*)

テキストまたは数値表現でログレベル *level* を返してください。

level が定義済みのレベル `CRITICAL`, `ERROR`, `WARNING`, `INFO`, `DEBUG` のいずれかである場合、対応する文字列が返されます。*addLevelName()* を使ってレベルに名前を関連付けていた場合、*level* に関連付けられた名前が返されます。定義済みのレベルに対応する数値を指定した場合、レベルに対応した文字列表現を返します。

level パラメータは、`INFO` のような整数定数の代わりに 'INFO' のようなレベルの文字列表現も受け付

けます。この場合、この関数は関連するレベルの数値表現を返します。

もし渡された数値や文字列がマッチしなければ、'Level %s' % level が返されます。

注釈: レベルは内部的には整数です (これはロギングのロジックが大小比較をする必要があるからです)。この関数は、数値のレベルを、書式記述子 `%(levelname)s` (*LogRecord* **属性** 参照) によって書式化されるログ出力の表示用レベル名に変換するなどの用途に使用されます。

バージョン 3.4 で変更: Python 3.4 以前のバージョンでは、この関数にはテキストのレベルも渡すことが出来、これは対応する数字レベルに読み替えられていました。このドキュメントされていなかった振る舞いは誤りであると判断され、Python 3.4 で一度削除されました。ただし後方互換性のために、これは 3.4.2 で元に戻されました。

`logging.makeLogRecord(`*attrdict*`)`

属性が *attrdict* で定義された、新しい *LogRecord* インスタンスを生成して返します。この関数は、pickle された *LogRecord* 属性の辞書をソケットを介して送信し、受信端で *LogRecord* インスタンスとして再構成する場合に便利です。

`logging.basicConfig(`***kwargs*`)`

デフォルトの *Formatter* を持つ *StreamHandler* を生成してルートロガーに追加し、ロギングシステムの基本的な環境設定を行います。関数 *debug()*, *info()*, *warning()*, *error()*, *critical()* は、ルートロガーにハンドラが定義されていない場合に自動的に *basicConfig()* を呼び出します。

この関数は *force* キーワード引数に `True` が設定されない限り、ルートロガーに設定されたハンドラがあれば何もしません。

注釈: この関数は、他のスレッドが開始される前にメインスレッドから呼び出されるべきです。Python の 2.7.1 や 3.2 以前のバージョンでは、この関数が複数のスレッドから呼ばれると (珍しい状況下とはいえ) ハンドラがルートロガーに複数回加えられることがあり、ログ内のメッセージが重複するという予期しない結果をもたらすことがあります。

以下のキーワード引数がサポートされます。

フォーマット	説明
filename	StreamHandler ではなく指定された名前で FileHandler が作られます。
filemode	*filename* が指定された場合、この **モード** でファイルが開かれます。デフォルトは `'a'` です。
format	ハンドラーで指定されたフォーマット文字列を使います。デフォルトは `levelname`, `name`, `message` 属性をコロン区切りにしたものです。
datefmt	指定された日時の書式で *time.strftime()* が受け付けるものを使います。
style	*format* が指定された場合、書式文字列にこのスタイルを仕様します。`'%'`, `'{'`, `'$'` のうち 1 つで、それぞれ *printf-style*, *str.format()*, *string.Template* に対応します。デフォルトは `'%'` です。
level	ルートロガーのレベルを指定された **レベル** に設定します。
stream	指定されたストリームを StreamHandler の初期化に使います。この引数は *filename* と同時には使えないことに注意してください。両方が指定されたときには `ValueError` が送出されます。
handlers	もし指定されれば、これは root ロガーに追加される既に作られたハンドラのイテラブルになります。まだフォーマッタがセットされていないすべてのハンドラは、この関数で作られたデフォルトフォーマッタが割り当てられることになります。この引数は *filename* や *stream* と互換性がないことに注意してください。両方が存在する場合 `ValueError` が上げられます。
force	このキーワード引数が真に設定されている場合、ルートのロガーに取り付けられたハンドラは全て取り除かれ、他の引数によって指定された設定が有効になる前に閉じられます。

バージョン 3.2 で変更: *style* 引数が追加されました。

バージョン 3.3 で変更: 互換性のない引数が指定された状況 (例えば *handlers* が *stream* や *filename* と一緒に指定されたり、*stream* が *filename* と一緒に指定された場合) を捕捉するために、追加のチェックが加えられました。

バージョン 3.8 で変更: *force* 引数が追加されました。

logging.shutdown()

ロギングシステムに対して、バッファのフラッシュを行い、すべてのハンドラを閉じることで順次シャットダウンを行うように告知します。この関数はアプリケーションの終了時に呼ばれるべきであり、また呼び出し以降はそれ以上ロギングシステムを使ってはなりません。

logging モジュールがインポートされると、この関数が終了ハンドラーとして登録されます (*atexit* 参照)。そのため、通常はこれを手動で行う必要はありません。

logging.setLoggerClass(*klass*)

ロギングシステムに対して、ロガーをインスタンス化する際にクラス *klass* を使うように指示します。指定するクラスは引数として名前だけをとるようなメソッド `__init__()` を定義していなければならず、`__init__()` では `Logger.__init__()` を呼び出さなければなりません。この関数が呼び出されるのはたいてい、独自の振る舞いをするロガーを使う必要のあるアプリケーションでロガーがインスタンス化される前です。呼び出された後は、いつでもそのサブクラスを使ってロガーのインスタンス化をし

てはいけません: 引き続き *logging.getLogger()* API を使用してロガーを取得してください。

logging.**setLogRecordFactory**(*factory*)

LogRecord を生成するのに使われる callable をセットします。

パラメータ **factory** -- ログレコードを生成するファクトリとして振舞う callable。

バージョン 3.2 で追加: この関数は、ログイベントを表現する *LogRecord* の構築方法に関して開発者により多くのコントロールを与えるため、*getLogRecordFactory()* とともに提供されました。

ファクトリは以下のようなシグネチャを持っています:

```
factory(name, level, fn, lno, msg, args, exc_info, func=None, sinfo=None,
**kwargs)
```

name　ロガーの名前。

level　ログレベル (数値)。

fn　ログ呼び出しが行われたファイルのフルパス名。

lno　ログ呼び出しが行われたファイルの行数。

msg　ログメッセージ。

args　ログメッセージに対する引数。

exc_info　例外タプルまたは None。

func　ログ呼び出しを起動した関数またはメソッドの名前。

sinfo　*traceback.print_stack()* で提供されるような、呼び出し階層を示すスタックトレースバック。

kwargs　追加のキーワード引数。

16.6.11 モジュールレベル属性

logging.**lastResort**

「最後の手段のハンドラ」が、この属性で利用可能です。これは *StreamHandler* が sys.stderr に WARNING レベルで書き出しているのがそうですし、ロギングの設定がなにか不在のロギングイベントを扱う場合に使われます。最終的な結果は、メッセージを単に sys.stderr に出力することです。これはかつて「logger XYZ についてのハンドラが見つかりません」と言っていたエラーメッセージを置き換えています。もしも何らかの理由でその昔の振る舞いが必要な場合は、lastResort に None をセットすれば良いです。

バージョン 3.2 で追加.

16.6.12 warnings モジュールとの統合

captureWarnings() 関数を使って、*logging* を *warnings* モジュールと統合できます。

logging.**captureWarnings**(*capture*)

この関数は、logging による警告の補足を、有効にまたは無効にします。

capture が True なら、*warnings* モジュールに発せられた警告は、ロギングシステムにリダイレクトされるようになります。具体的には、警告が *warnings.formatwarning()* でフォーマット化され、結果の文字列が 'py.warnings' という名のロガーに、WARNING の重大度でロギングされるようになります。

capture が False なら、警告のロギングシステムに対するリダイレクトは止められ、警告は元の (すなわち、captureWarnings(True) が呼び出される前に有効だった) 送信先にリダイレクトされるようになります。

参考:

logging.config **モジュール** logging モジュールの環境設定 API です。

logging.handlers **モジュール** logging モジュールに含まれる、便利なハンドラです。

PEP 282 - ログシステム この機能を Python 標準ライブラリに含めることを述べた提案です。

Original Python logging package これは、*logging* パッケージのオリジナルのソースです。このサイトから利用できるバージョンのパッケージは、*logging* パッケージを標準ライブラリに含まない、Python 1.5.2, 2.1.x および 2.2.x で使うのに適しています。

16.7 logging.config --- ロギングの環境設定

ソースコード: Lib/logging/config.py

Important

このページには、リファレンス情報だけが含まれています。チュートリアルは、以下のページを参照してください

- 基本チュートリアル
- 上級チュートリアル
- ロギングクックブック

この節は、logging モジュールを設定するための API を解説します。

16.7.1 環境設定のための関数

以下の関数は logging モジュールの環境設定をします。これらの関数は、*logging.config* にあります。これらの関数の使用はオプションです --- *logging* モジュールはこれらの関数を使うか、(*logging* 自体で定義されている) 主要な API を呼び出し、*logging* か *logging.handlers* で宣言されているハンドラを定義することで設定できます。

logging.config.**dictConfig**(*config*)

辞書からロギング環境設定を取得します。この辞書の内容は、以下の **環境設定辞書スキーマ** で記述されています。

環境設定中にエラーに遭遇すると、この関数は適宜メッセージを記述しつつ *ValueError*, *TypeError*, *AttributeError* または *ImportError* を送出します。例外を送出する条件を (不完全かもしれませんが) 以下に列挙します:

- 文字列でなかったり、実際のロギングレベルと関係ない文字列であったりする `level`。
- ブール値でない `propagate` の値。
- 対応する行き先を持たない id。
- インクリメンタルな呼び出しの中で見つかった存在しないハンドラ id。
- 無効なロガー名。
- 内部や外部のオブジェクトに関わる不可能性。

解析は `DictConfigurator` クラスによって行われます。このクラスのコンストラクタは環境設定に使われる辞書に渡され、このクラスは `configure()` メソッドを持ちます。*logging.config* モジュールは、呼び出し可能属性 `dictConfigClass` を持ち、これはまず `DictConfigurator` に設定されます。`dictConfigClass` の値は適切な独自の実装で置き換えられます。

dictConfig() は `dictConfigClass` を、指定された辞書を渡して呼び出し、それから返されたオブジェクトの `configure()` メソッドを呼び出して、環境設定を作用させます:

```
def dictConfig(config):
    dictConfigClass(config).configure()
```

例えば、`DictConfigurator` のサブクラスは、自身の `__init__()` で `DictConfigurator.__init__()` を呼び出し、それから続く `configure()` の呼び出しに使えるカスタムの接頭辞を設定できます。`dictConfigClass` は、この新しいサブクラスに束縛され、そして *dictConfig()* はちょうどデフォルトの、カスタマイズされていない状態のように呼び出せます。

バージョン 3.2 で追加.

logging.config.**fileConfig**(*fname*, *defaults=None*, *disable_existing_loggers=True*)

ログ記録の環境設定を *configparser* 形式ファイルから読み出します。そのファイルの形式は **環境設定ファイルの書式** で記述されているとおりにしなければなりません。この関数はアプリケーションから何度も呼び出すことができ、これによって、(設定を選択し、選択された設定を読み出す機構をデベロッパが提供していれば) 複数の準備済みの設定からエンドユーザが選択するようにできます。

パラメータ

- **fname** -- ファイル名、あるいはファイルのようなオブジェクト、または *RawConfigParser* 派生のインスタンス。*RawConfigParser* 派生のインスタンスが与えられれば、それはそのまま使われます。そうでない場合 Configparser がインスタンス化され、設定はそれを使って fname が指すオブジェクトから読み込まれます。それが *readline()* メソッドを持っていればそれはファイルのようなオブジェクトと仮定され、*read_file()* で読み込まれます; そうでない場合、それはファイル名と仮定されて、*read()* に渡されます。
- **defaults** -- ConfigParser に渡されるデフォルト値をこの引数で指定することができます。
- **disable_existing_loggers** -- False が指定された場合は、この呼び出しが行われたときに存在するロガーは有効のまま残されます。後方互換性のあるやり方で古い振る舞いを保つので、デフォルト値は True になっています。そのような振る舞いでは、既存のノンルートロガーまたはそれらのロガーの先祖がロギング設定の中で明示的に名付けられていない限り、既存のロガーを無効にします。

バージョン 3.4 で変更: fname として *RawConfigParser* のサブクラスのインスタンスが渡せ得るようになっています。これによってこのようなことが容易になります:

- ロギングの設定が、アプリケーション全体の設定における単なる一部であるような設定ファイルの使用。
- ファイルから設定を読み込み、fileConfig に通す前に (例えばコマンドラインパラメータやランタイム環境の他のなにかで) アプリケーションによって修正するようなこと。

logging.config.**listen**(*port=DEFAULT_LOGGING_CONFIG_PORT*, *verify=None*)

指定されたポートでソケットサーバを起動し、新しい設定を待ち受けます。ポートが指定されなかった場合は、モジュールのデフォルトの DEFAULT_LOGGING_CONFIG_PORT が使用されます。ロギング設定は *dictConfig()* あるいは *fileConfig()* で処理できるファイルとして送信されます。*Thread* インスタンスを返し、このインスタンスの *start()* を呼び出してサーバを起動し、適切なところで *join()* を呼び出すことができます。サーバを停止するには、*stopListening()* を呼び出します。

verify 引数を指定する場合は、これはソケットを通して受け取ったバイト文字列が妥当であるか、処理すべきであるかどうかを検査する callable である必要があります。ソケットを通じて、暗号化または署名あるいはその両方を受け取ることがあります。そのような場合に、verify callable が署名の正当性検査または暗号化の復号あるいはその両方を実施することが出来ます。verify callable は単一引数で呼び出されます - ソケットを通じて受け取ったバイト文字列です - そして処理すべきバイト文字列、または捨て去られるべきであることを示すための None を返す必要があります。返却されるバイト文字列は (たとえば正当性検査だけが行われて) 渡されたものと同じかもしれませんし、あるいは (おそらく暗号化の復号が行われて) まったく異なるものかもしれません。

ソケットに設定を送るには、まず設定ファイルを読み、それを struct.pack('>L', n) を使って長さ 4 バイトのバイナリにパックしたものを前に付けたバイト列としてソケットに送ります。

注釈: 設定の部分が *eval()* を通して渡されるので、この関数の使用はユーザに対してセキュリティ

リスクを公開してしまうかもしれません。この関数は単に localhost 上のソケットに接続してリモートマシンからは接続を受け付けませんが、*listen()* を呼んだプロセスのアカウントで信頼されていないコードが実行されるシナリオが存在します。特に、*listen()* を呼んだプロセスがユーザがお互いを信頼することができないマルチユーザのマシン上で実行される場合、悪意のあるユーザは、犠牲者のユーザのプロセスで本質的に任意のコードを実行するように細工することができます。単に犠牲者の *listen()* ソケットに接続して、犠牲者のプロセスで実行したいコードを実行するような設定を送るだけです。これは、特にデフォルトポートが使用されている場合に行うのがより簡単ですが、異なるポートが使用されていたとしてもそれほど難しくありません)。これが起こるリスクを避けるには、*listen()* に verify 引数を使用して、認識されていない設定が適用されないようにしてください。

バージョン 3.4 で変更: verify 引数が追加されました。

注釈: 既存のロガーを無効にしない構成をリスナーに送信する場合は、設定には JSON フォーマットを使用する必要があります。これは、設定に *dictConfig()* を使用します。このメソッドを使用すると、disable_existing_loggers に False を指定した設定を送信できます。

logging.config.**stopListening**()

listen() を呼び出して作成された、待ち受け中のサーバを停止します。通常 *listen()* の戻り値に対して join() が呼ばれる前に呼び出します。

16.7.2 環境設定辞書スキーマ

ロギング設定を記述するには、生成するさまざまなオブジェクトと、それらのつながりを列挙しなければなりません。例えば、'console' という名前のハンドラを生成し、'startup' という名前のロガーがメッセージを 'console' ハンドラに送るというようなことを記述します。これらのオブジェクトは、*logging* モジュールによって提供されるものに限らず、独自のフォーマッタやハンドラクラスを書くことも出来ます。このクラスへのパラメータは、sys.stderr のような外部オブジェクトを必要とすることもあります。これらのオブジェクトとつながりを記述する構文は、以下の **オブジェクトの接続** で定義されています。

辞書スキーマの詳細

dictConfig() に渡される辞書は、以下のキーを含んでいなければなりません:

- *version* - スキーマのバージョンを表す整数値に設定されます。現在有効な値は 1 だけですが、このキーがあることで、このスキーマは後方互換性を保ちながら発展できます。

その他すべてのキーは省略可能ですが、与えられたなら以下に記述するように解釈されます。以下のすべての場合において、' 環境設定辞書' と記載されている所では、その辞書に特殊な '()' キーがあるかを調べることで、カスタムのインスタント化が必要であるか判断されます。その場合は、以下の **ユーザ定義オブジェクト** で記述されている機構がインスタンス生成に使われます。そうでなければ、インスタンス化するべきものを決定するのにコンテキストが使われます。

- *formatters* - 対応する値は辞書で、そのそれぞれのキーがフォーマッタ id になり、それぞれの値が対

応する *Formatter* インスタンスをどのように環境設定するかを記述する辞書になります。

環境設定辞書から、(デフォルトが None の) キー format と datefmt を検索し、それらが *Formatter* インスタンスを構成するのに使われます。

バージョン 3.8 で変更: a validate key (with default of True) can be added into the formatters section of the configuring dict, this is to validate the format.

- *filters* - 対応する値は辞書で、そのそれぞれのキーがフィルタ id になり、それぞれの値が対応する Filter インスタンスをどのように環境設定するかを記述する辞書になります。

  環境設定辞書は、(デフォルトが空文字列の) キー name を検索され、それらが *logging.Filter* インスタンスを構成するのに使われます。

- *handlers* - 対応する値は辞書で、そのそれぞれのキーがハンドラ id になり、それぞれの値が対応する Handler インスタンスをどのように環境設定するかを記述する辞書になります。

  環境設定辞書は、以下のキーを検索されます:

  – class (必須)。これはハンドラクラスの完全に修飾された名前です。

  – level (任意)。ハンドラのレベルです。

  – formatter (任意)。このハンドラへのフォーマッタの id です。

  – filters (任意)。このハンドラへのフィルタの id のリストです。

  その他の **すべての** キーは、ハンドラのコンストラクタにキーワード引数として渡されます。例えば、以下のコード片が与えられたとすると:

```
handlers:
  console:
    class : logging.StreamHandler
    formatter: brief
    level   : INFO
    filters: [allow_foo]
    stream  : ext://sys.stdout
  file:
    class : logging.handlers.RotatingFileHandler
    formatter: precise
    filename: logconfig.log
    maxBytes: 1024
    backupCount: 3
```

  id が console であるハンドラが、sys.stdout を根底のストリームにして、*logging.StreamHandler* としてインスタンス化されます。id が file であるハンドラが、filename='logconfig.log', maxBytes=1024, backupCount=3 をキーワード引数にして、*logging.handlers.RotatingFileHandler* としてインスタンス化されます。

- *loggers* - 対応する値は辞書で、そのそれぞれのキーがロガー名になり、それぞれの値が対応する Logger インスタンスをどのように環境設定するかを記述する辞書になります。

  環境設定辞書は、以下のキーを検索されます:

  - `level` (任意)。ロガーのレベルです。
  - `propagate` (任意)。ロガーの伝播の設定です。
  - `filters` (任意)。このロガーへのフィルタの id のリストです。
  - `handlers` (任意)。このロガーへのハンドラの id のリストです。

  指定されたロガーは、指定されたレベル、伝播、ハンドラに従って環境設定されます。

- *root* - これは、ルートロガーへの設定になります。この環境設定の進行は、`propagate` 設定が適用されないことを除き、他のロガーと同じです。
- *incremental* - この環境設定が既存の環境設定に対する増分として解釈されるかどうかです。この値のデフォルトは `False` で、指定された環境設定は、既存の *`fileConfig()`* API によって使われているのと同じ意味上で、既存の環境設定を置き換えます。

  指定された値が `True` なら、環境設定は **増分設定** の節で記述されているように進行します。
- *disable_existing_loggers* - 既存の非ルートロガーをすべて無効にするべきかどうかです。この設定は、*`fileConfig()`* における同じ名前のパラメータと同じです。設定されていなければ、このパラメータのデフォルトは `True` です。この値は、*incremental* が `True` なら無視されます。

増分設定

増分設定に完全な柔軟性を提供するのは難しいです。例えば、フィルタやフォーマッタのようなオブジェクトは匿名なので、一旦環境設定がなされると、設定を拡張するときにそのような匿名オブジェクトを参照することができません。

さらに、一旦環境設定がなされた後、実行時にロガー、ハンドラ、フィルタ、フォーマッタのオブジェクトグラフを任意に変えなければならない例もありません。ロガーとハンドラの冗長性は、レベル (または、ロガーの場合には、伝播フラグ) を設定することによってのみ制御できます。安全な方法でオブジェクトグラフを任意に変えることは、マルチスレッド環境で問題となります。不可能ではないですが、その効用は実装に加えられる複雑さに見合いません。

従って、環境設定辞書の `incremental` キーが与えられ、これが `True` であるとき、システムは `formatters` と `filters` の項目を完全に無視し、`handlers` の項目の `level` 設定と、`loggers` と `root` の項目の `level` と `propagate` 設定のみを処理します。

環境設定辞書の値を使うことで、設定は pickle 化された辞書としてネットワークを通してソケットリスナに送ることができます。これにより、長時間起動するアプリケーションのロギングの冗長性を、アプリケーションを止めて再起動する必要なしに、いつでも変更することができます。

オブジェクトの接続

このスキーマは、ロギングオブジェクトの一揃い - ロガー、ハンドラ、フォーマッタ、フィルタ - について記述します。これらは、オブジェクトグラフ上でお互い接続されます。従って、このスキーマは、オブジェクト間の接続を表現しなければなりません。例えば、環境設定で、特定のロガーが特定のハンドラに取り付けられたとします。この議論では、ロガーとハンドラが、これら 2 つの接続のそれぞれ送信元と送信先であるといえます。もちろん、この設定オブジェクト中では、これはハンドラへの参照を保持しているロガーで表されます。設定辞書中で、これは次のようになされます。まず、送信先オブジェクトを曖昧さなく指定する id を与えます。そして、その id を送信元オブジェクトの環境設定で使い、送信元とその id をもつ送信先が接続されていることを示します。

ですから、例えば、以下の YAML のコード片を例にとると:

```
formatters:
  brief:
    # configuration for formatter with id 'brief' goes here
  precise:
    # configuration for formatter with id 'precise' goes here
handlers:
  h1: #This is an id
   # configuration of handler with id 'h1' goes here
   formatter: brief
  h2: #This is another id
   # configuration of handler with id 'h2' goes here
   formatter: precise
loggers:
  foo.bar.baz:
    # other configuration for logger 'foo.bar.baz'
    handlers: [h1, h2]
```

(注釈: YAML がここで使われているのは、辞書の等価な Python 形式よりもこちらのほうが少し読みやすいからです。)

ロガーの id は、プログラム上でロガーへの参照を得るために使われるロガー名で、たとえば `foo.bar.baz` です。フォーマッタとフィルタの id は、(上の `brief`, `precise` のような) 任意の文字列値にできます。これらは一時的なもので、環境設定辞書の処理にのみ意味があり、オブジェクト間の接続を決定するのに使われます。また、これらは設定の呼び出しが完了したとき、どこにも残りません。

上記のコード片は、`foo.bar.baz` というの名ロガーに、ハンドラ id `h1` と `h2` で表される 2 つのハンドラを接続することを示します。`h1` のフォーマッタは id `brief` で記述されるもので、`h2` のフォーマッタは id `precise` で記述されるものです。

ユーザ定義オブジェクト

このスキーマは、ハンドラ、フィルタ、フォーマッタのための、ユーザ定義オブジェクトをサポートします。(ロガーは、異なるインスタンスに対して異なる型を持つ必要はないので、この環境設定スキーマは、ユーザ定義ロガークラスをサポートしていません。)

設定されるオブジェクトは、それらの設定を詳述する辞書によって記述されます。場所によっては、あるオブジェクトがどのようにインスタンス化されるかというコンテキストを、ロギングシステムが推測できます。しかし、ユーザ定義オブジェクトがインスタンス化されるとき、システムはどのようにこれを行うかを知りません。ユーザ定義オブジェクトのインスタンス化を完全に柔軟なものにするため、ユーザは 'ファクトリ' - 設定辞書を引数として呼ばれ、インスタンス化されたオブジェクトを返す呼び出し可能オブジェクト - を提供する必要があります。これは特殊キー `'()'` で利用できる、ファクトリへの絶対インポートパスによって合図されます。ここに具体的な例を挙げます:

```
formatters:
  brief:
    format: '%(message)s'
  default:
    format: '%(asctime)s %(levelname)-8s %(name)-15s %(message)s'
    datefmt: '%Y-%m-%d %H:%M:%S'
  custom:
      (): my.package.customFormatterFactory
      bar: baz
      spam: 99.9
      answer: 42
```

上記の YAML コード片は 3 つのフォーマッタを定義します。1 つ目は、id が `brief` で、指定されたフォーマット文字列をもつ、標準 *logging.Formatter* インスタンスです。2 つ目は、id が `default` で、長いフォーマットを持ち、時間フォーマットも定義していて、結果はその 2 つのフォーマット文字列で初期化された *logging.Formatter* になります。Python ソース形式で見ると、`brief` と `default` フォーマッタは、それぞれ設定の部分辞書:

```
{
  'format' : '%(message)s'
}
```

および:

```
{
  'format' : '%(asctime)s %(levelname)-8s %(name)-15s %(message)s',
  'datefmt' : '%Y-%m-%d %H:%M:%S'
}
```

を持ち、これらの辞書が特殊キー `'()'` を持たないので、インスタンス化はコンテキストから推測され、結果として標準の *logging.Formatter* インスタンスが生成されます。id が `custom` である、3 つ目のフォーマッタの設定をする部分辞書は:

```
{
  '()' : 'my.package.customFormatterFactory',
```

(次のページに続く)

(前のページからの続き)

```
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42
}
```

で、ユーザ定義のインスタンス化が望まれることを示す特殊キー '()' を含みます。この場合、指定された呼び出し可能ファクトリオブジェクトが使われます。これが実際の呼び出し可能オブジェクトであれば、それが直接使われます - そうではなく、(この例でのように) 文字列を指定したなら、実際の呼び出し可能オブジェクトは、通常のインポート機構を使って検索されます。その呼び出し可能オブジェクトは、環境設定の部分辞書の、**残りの** 要素をキーワード引数として呼ばれます。上記の例では、id が custom のフォーマッタは、以下の呼び出しによって返されるものとみなされます:

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

キー '()' が特殊キーとして使われるのは、キーワードパラメータ名として不正で、呼び出しに使われるキーワード引数と衝突し得ないからです。'()' はまた、対応する値が呼び出し可能オブジェクトであると覚えやすくします。

外部オブジェクトへのアクセス

環境設定が、例えば sys.stderr のような、設定の外部のオブジェクトへの参照を必要とすることがあります。設定辞書が Python コードで構成されていれば話は簡単ですが、これがテキストファイル (JSON, YAML 等) を通して提供されていると問題となります。テキストファイルでは、sys.stderr をリテラル文字列 'sys.stderr' と区別する標準の方法がありません。この区別を容易にするため、環境設定システムは、文字列中の特定の特殊接頭辞を見つけ、それらを特殊に扱います。例えば、リテラル文字列 'ext://sys.stderr' が設定中の値として与えられたら、この ext:// は剥ぎ取られ、この値の残りが普通のインポート機構で処理されます。

このような接頭辞の処理は、プロトコルの処理と同じようになされます。どちらの機構も、正規表現 ^(?P<prefix>[a-z]+)://(?P<suffix>.*)$ にマッチする接頭辞を検索し、それによって prefix が認識されたなら、接頭辞に応じたやり方で suffix が処理され、その処理の結果によって文字列値が置き換えられます。接頭辞が認識されなければ、その文字列値はそのまま残されます。

内部オブジェクトへのアクセス

外部オブジェクトと同様、環境設定内部のオブジェクトへのアクセスを必要とすることもあります。これは、その各オブジェクトを司る環境設定システムによって暗黙に行われます。例えば、ロガーやハンドラの level に対する文字列値 'DEBUG' は、自動的に値 logging.DEBUG に変換されますし、handlers, filters および formatter の項目は、オブジェクト id を取って、適切な送信先オブジェクトを決定します。

しかし、ユーザ定義モジュールには、*logging* モジュールには分からないような、より一般的な機構が必要です。例えば、*logging.handlers.MemoryHandler* があって、委譲する先の別のハンドラである target 引数を取るとします。システムはこのクラスをすでに知っていますから、設定中で、与えられた target は関連するターゲットハンドラのオブジェクト id でさえあればよく、システムはその id からハンドラを決定します。しかし、ユーザが my.package.MyHandler を定義して、それが alternate ハンドラを持つなら、設定

システムは alternate がハンドラを参照していることを知りません。これを知らせるのに、一般的な解析システムで、ユーザはこのように指定できます:

```
handlers:
  file:
    # configuration of file handler goes here

  custom:
    (): my.package.MyHandler
    alternate: cfg://handlers.file
```

リテラル文字列 'cfg://handlers.file' は、ext:// 接頭辞が付いた文字列と同じように分析されますが、インポート名前空間ではなく、環境設定自体が検索されます。この機構は str.format でできるのと同じようにドットやインデックスのアクセスができます。従って、環境設定において以下のコード片が与えられれば:

```
handlers:
  email:
    class: logging.handlers.SMTPHandler
    mailhost: localhost
    fromaddr: my_app@domain.tld
    toaddrs:
      - support_team@domain.tld
      - dev_team@domain.tld
    subject: Houston, we have a problem.
```

文字列 'cfg://handlers' は、キー handlers をもつ辞書であると分析され、文字列 'cfg://handlers.email' は、handlers 辞書内の、email キーをもつ辞書であると分析されます。文字列 'cfg://handlers.email.toaddrs[1] は、'dev_team@domain.tld' と分析され、'cfg://handlers.email.toaddrs[0]' は値 'support_team@domain.tld' と分析されます。subject の値には、'cfg://handlers.email.subject' または等価な 'cfg://handlers.email[subject]' でアクセスできます。後者が必要なのは、キーがスペースや非アルファベット文字を含むときのみです。インデックス値が十進数字のみで構成されているなら、まず対応する整数値を使ってアクセスが試みられ、必要なら文字列値で代替します。

文字列 cfg://handlers.myhandler.mykey.123 が与えられると、これは config_dict['handlers']['myhandler']['mykey']['123'] と分析されます。文字列が cfg://handlers.myhandler.mykey[123] と指定されたら、システムは config_dict['handlers']['myhandler']['mykey'][123] から値を引き出そうとし、失敗したら config_dict['handlers']['myhandler']['mykey']['123'] で代替します。

インポート解決とカスタムインポーター

インポート解決は、デフォルトではインポートを行うために *__import__()* 組み込み関数を使用します。これを独自のインポートメカニズムに置き換えたいと思うかもしれません: もしそうなら、DictConfigurator あるいはその上位クラスである BaseConfigurator クラスの importer 属性を置換することができます。ただし、この関数はクラスからディスクリプタ経由でアクセスされる点に注意する必要があります。インポートを行うために Python callable を使用していて、それをインスタンスレベルではなくクラスレベルで定義したければ、*staticmethod()* でそれをラップする必要があります。例えば:

```
from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)
```

configurator **インスタンス** に対してインポート callable をセットする場合は、*staticmethod()* でラップする必要はありません。

16.7.3 環境設定ファイルの書式

fileConfig() が解釈できる環境設定ファイルの形式は、*configparser* の機能に基づいています。ファイルには、[loggers], [handlers], [formatters] といったセクションが入っていなければならず、各セクションではファイル中で定義されている各タイプのエンティティを名前で指定しています。こうしたエンティティの各々について、そのエンティティをどう設定するかを示した個別のセクションがあります。すなわち、log01 という名前の [loggers] セクションにあるロガーに対しては、対応する詳細設定がセクション [logger_log01] に収められています。同様に、hand01 という名前の [handlers] セクションにあるハンドラは [handler_hand01] と呼ばれるセクションに設定をもつことになり、[formatters] セクションにある form01 は [formatter_form01] というセクションで設定が指定されています。ルートロガーの設定は [logger_root] と呼ばれるセクションで指定されていなければなりません。

注釈: *fileConfig()* API は *dictConfig()* API よりも古く、ロギングのある種の側面についてカバーする機能に欠けています。たとえば *fileConfig()* では数値レベルを超えたメッセージを単に拾うフィルタリングを行う *Filter* オブジェクトを構成出来ません。*Filter* のインスタンスをロギングの設定において持つ必要があるならば、*dictConfig()* を使う必要があるでしょう。設定の機能における将来の拡張は *dictConfig()* に対して行われることに注意してください。ですから、そうするのが便利であるときに新しい API に乗り換えるのは良い考えです。

ファイルにおけるこれらのセクションの例を以下に示します。

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

ルートロガーでは、レベルとハンドラのリストを指定しなければなりません。ルートロガーのセクションの例を以下に示します。

```
[logger_root]
level=NOTSET
handlers=hand01
```

level エントリは DEBUG, INFO, WARNING, ERROR, CRITICAL のうちの一つか、NOTSET になります。ルートロガーの場合にのみ、NOTSET はすべてのメッセージがログ記録されることを意味します。レベル値は logging パッケージの名前空間のコンテキストにおいて *eval()* されます。

handlers エントリはコンマで区切られたハンドラ名からなるリストで、[handlers] セクションになくてはなりません。また、これらの各ハンドラの名前に対応するセクションが設定ファイルに存在しなければなりません。

ルートロガー以外のロガーでは、いくつか追加の情報が必要になります。これは以下の例のように表されます。

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

level および handlers エントリはルートロガーのエントリと同様に解釈されますが、非ルートロガーのレベルが NOTSET に指定された場合、ロギングシステムはロガー階層のより上位のロガーにロガーの実効レベルを問い合わせるところが違います。propagate エントリは、メッセージをロガー階層におけるこのロガーの上位のハンドラに伝播させることを示す 1 に設定されるか、メッセージを階層の上位に伝播 **しない** ことを示す 0 に設定されます。qualname エントリはロガーのチャネル名を階層的に表したもの、すなわちアプリケーションがこのロガーを取得する際に使う名前になります。

ハンドラの環境設定を指定しているセクションは以下の例のようになります。

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

class エントリはハンドラのクラス (logging パッケージの名前空間において *eval()* で決定されます) を示します。level はロガーの場合と同じように解釈され、NOTSET は " すべてを記録する (log everything)" と解釈されます。

formatter エントリはこのハンドラのフォーマッタに対するキー名を表します。空文字列の場合、デフォルトのフォーマッタ (logging._defaultFormatter) が使われます。名前が指定されている場合、その名前は [formatters] セクションになくてはならず、対応するセクションが設定ファイル中になければなりません。

args エントリは、logging パッケージの名前空間のコンテキストで *eval()* される際、ハンドラクラスのコンストラクタに対する引数からなるリストになります。典型的なエントリがどうやって作成されるかについては、対応するハンドラのコンストラクタか、以下の例を参照してください。もし指定しなかった場合にはデフォルトは () となります。

オプションの kwargs エントリーは、*eval()* が logging パッケージの名前空間のコンテキストで利用された時に、ハンドラークラスのコンストラクタに渡されるキーワード引数辞書となります。指定されなかったときのデフォルトは {} です。

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=(('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')
kwargs={'timeout': 10.0}

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
kwargs={'secure': True}
```

フォーマッタの環境設定を指定しているセクションは以下のような形式です。

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s
datefmt=
class=logging.Formatter
```

The `format` entry is the overall format string, and the `datefmt` entry is the `strftime()`-compatible date/time format string. If empty, the package substitutes something which is almost equivalent to specifying the date format string `'%Y-%m-%d %H:%M:%S'`. This format also specifies milliseconds, which are appended to the result of using the above format string, with a comma separator. An example time in this format is `2003-01-23 00:29:50,411`.

`class` エントリはオプションです。これはフォーマッタクラスの名前を (モジュール名とクラス名をドットでつないだもので) 指し示すものです。このオプションは *Formatter* の子クラスをインスタンス化するのに便利です。*Formatter* の子クラスが、展開もしくは要約された形式の例外トレースバックを表示することができます。

注釈: *eval()* を使用していることで、上述のようにソケット経由で設定を送受信するために *listen()* を使用していることに起因する潜在的なセキュリティリスクがあります。そのリスクは、相互に信頼できない多数のユーザが同じマシン上でコードを実行する場合に制限されています; 詳細は *listen()* ドキュメンテーションを参照してください。

参考:

logging **モジュール** logging モジュールの API リファレンス。

logging.handlers **モジュール** logging モジュールに含まれる、便利なハンドラです。

16.8 logging.handlers --- ロギングハンドラ

ソースコード: Lib/logging/handlers.py

Important

このページには、リファレンス情報だけが含まれています。チュートリアルは、以下のページを参照してください

- 基本チュートリアル
- 上級チュートリアル
- ロギングクックブック

このパッケージでは、以下の便利なハンドラが提供されています。なお、これらのハンドラのうち、3 つ (*StreamHandler*, *FileHandler* および *NullHandler*) は、実際には *logging* モジュール自身で定義され

ていますが、他のハンドラと一緒にここでドキュメント化します。

16.8.1 StreamHandler

logging コアパッケージに含まれる *StreamHandler* クラスは、ログ出力を *sys.stdout*, *sys.stderr* あるいは何らかのファイル風 (file-like) オブジェクト (あるいは、より正確に言えば `write()` および `flush()` メソッドをサポートする何らかのオブジェクト) といったストリームに送信します。

class logging.**StreamHandler**(*stream=None*)

StreamHandler クラスの新たなインスタンスを返します。*stream* が指定された場合、インスタンスはログ出力先として指定されたストリームを使います; そうでない場合、*sys.stderr* が使われます。

emit(*record*)

フォーマッタが指定されていれば、フォーマッタを使ってレコードを書式化します。次に、レコードが終端記号とともにストリームに書き込まれます。例外情報が存在する場合、*traceback.print_exception()* を使って書式化され、ストリームの末尾につけられます。

flush()

ストリームの *flush()* メソッドを呼び出してバッファをフラッシュします。`close()` メソッドは *Handler* から継承しているため何も出力を行わないので、*flush()* 呼び出しを明示的に行う必要があるかもしれません。

setStream(*stream*)

このインスタンスの stream と指定された値が異なる場合、指定された値に設定します。新しい stream を設定する前に、古い stream はフラッシュされます。

パラメータ **stream** -- ハンドラがこれから使う stream 。

戻り値 stream が変更された場合は古い stream 、そうでない場合は *None* 。

バージョン 3.7 で追加.

バージョン 3.2 で変更: `StreamHandler` クラスに `terminator` 属性が追加されました (デフォルト値は `'\n'`)。これは、書式化されたレコードをストリームに書き込むときの終端記号として使用されます。このような改行による終端を望まなければ、ハンドラ・インスタンスの `terminator` 属性を空の文字列に設定することができます。初期のバージョンでは、終端記号は `'\n'` としてハードコードされていました。

16.8.2 FileHandler

logging コアパッケージに含まれる *FileHandler* クラスは、ログ出力をディスク上のファイルに送信します。このクラスは出力機能を *StreamHandler* から継承しています。

class logging.**FileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*)

FileHandler クラスの新たなインスタンスを返します。指定されたファイルが開かれ、ログ記録のためのストリームとして使われます。*mode* が指定されなかった場合、`'a'` が使われます。*encoding* が `None` でない場合、その値はファイルを開くときのエンコーディングとして使われます。*delay* が真な

らば、ファイルを開くのは最初の *emit()* 呼び出しまで遅らせられます。デフォルトでは、ファイルは無制限に大きくなりつづけます。

バージョン 3.6 で変更: 文字列値に加え、*Path* オブジェクトも *filename* 引数が受け取るようになりました。

close()
ファイルを閉じます。

emit(*record*)
record をファイルに出力します。

16.8.3 NullHandler

バージョン 3.1 で追加.

logging コアパッケージに含まれる *NullHandler* クラスは、いかなる書式化も出力も行いません。これは本質的には、ライブラリ開発者に使われる 'no-op' ハンドラです。

class logging.**NullHandler**
NullHandler クラスの新しいインスタンスを返します。

emit(*record*)
このメソッドは何もしません。

handle(*record*)
このメソッドは何もしません。

createLock()
アクセスが特殊化される必要がある I/O が下にないので、このメソッドはロックに対して None を返します。

NullHandler の使い方の詳しい情報は、library-config を参照してください。

16.8.4 WatchedFileHandler

logging.handlers モジュールに含まれる *WatchedFileHandler* クラスは、ログ記録先のファイルを監視する FileHandler の一種です。ファイルが変更された場合、ファイルを閉じてからファイル名を使って開き直します。

ファイルはログファイルをローテーションさせる *newsyslog* や *logrotate* のようなプログラムを使うことで変更されることがあります。このハンドラは、Unix/Linux で使われることを意図していますが、ファイルが最後にログを出力してから変わったかどうかを監視します。(ファイルはデバイスや inode が変わることで変わったと判断します。) ファイルが変わったら古いファイルのストリームは閉じて、現在のファイルを新しいストリームを取得するために開きます。

このハンドラを Windows で使うことは適切ではありません。というのも Windows では開いているログファイルを移動したり削除したりできないからです - logging はファイルを排他的ロックを掛けて開きます - そ

のためこうしたハンドラは必要ないのです。さらに、Windows では *ST_INO* がサポートされていません; *stat()* はこの値として常に 0 を返します。

class logging.handlers.**WatchedFileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*)

WatchedFileHandler クラスの新たなインスタンスを返します。指定されたファイルが開かれ、ログ記録のためのストリームとして使われます。*mode* が指定されなかった場合、`'a'` が使われます。*encoding* が `None` でない場合、その値はファイルを開くときのエンコーディングとして使われます。*delay* が真ならば、ファイルを開くのは最初の *emit()* 呼び出しまで遅らせられます。デフォルトでは、ファイルは無制限に大きくなりつづけます。

バージョン 3.6 で変更: 文字列値に加え、*Path* オブジェクトも *filename* 引数が受け取るようになりました。

reopenIfNeeded()

ファイルが変更されていないかチェックします。もし変更されていれば、手始めにレコードをファイルに出力し、既存のストリームはフラッシュして閉じられ、ファイルが再度開かれます。

バージョン 3.6 で追加.

emit(*record*)

レコードをファイルに出力しますが、最初に *reopenIfNeeded()* を呼び出して、変更があった場合はファイルを再度開きます。

16.8.5 BaseRotatingHandler

logging.handlers モジュールに存在する *BaseRotatingHandler* クラスは、ローテートを行うファイルハンドラ *RotatingFileHandler* と *TimedRotatingFileHandler* のベースクラスです。このクラスをインスタンス化する必要はありませんが、オーバーライドすることになるかもしれない属性とメソッドを持っています。

class logging.handlers.**BaseRotatingHandler**(*filename*, *mode*, *encoding=None*, *delay=False*)

パラメータは `FileHandler` と同じです。属性は次の通りです:

namer

この属性に callable がセットされた場合、*rotation_filename()* メソッドはこの callable に委譲されます。callable に渡されるパラメータは *rotation_filename()* に渡されたものです。

注釈: namer 関数はロールオーバー中にかなりの回数呼ばれます。そのため、できるだけ単純で、速くあるべきです。さらに、それは与えられた入力に対しては常に同じ出力を返すべきです。そうでなければ、ロールオーバーの振る舞いは期待通りに動かないかもしれません。

バージョン 3.3 で追加.

rotator

この属性に callable がセットされた場合、*rotate()* メソッドはこの callable に委譲されます。

callable に渡されるパラメータは *rotate()* に渡されたものです。

バージョン 3.3 で追加.

rotation_filename(*default_name*)

ローテートを行う際にログファイルのファイル名を変更します。

このメソッドは、ファイル名をカスタマイズするために提供されます。

デフォルト実装は、ハンドラの 'namer' 属性が callable だった場合、その callable を呼んでデフォルト名を渡します。属性が callable でない場合 (デフォルトは `None` です)、名前は変更せずに返されます。

パラメータ **default_name** -- ログファイルのデフォルトのファイル名。

バージョン 3.3 で追加.

rotate(*source*, *dest*)

ローテートが行われる時、現在のログをローテートします。

デフォルト実装は、ハンドラの 'rotator' 属性が callable だった場合、その callable を呼んで source と dest 引数を渡します。属性が callable でない場合 (デフォルトは `None` です)、単に source が destination に改名されます。

パラメータ

- **source** -- ソースファイル名。これは通常ベースファイル名 、例えば 'test.log' となります。
- **dest** -- 変更先ファイル名。これは通常ソースファイルをローテートしたもの (例えば 'test.log.1') です。

バージョン 3.3 で追加.

これらの属性が存在する理由は、サブクラス化を省略できるようにするためです。*RotatingFileHandler* と *TimedRotatingFileHandler* のインスタンスに対して同じ callable が使えます。もし namer や rotator callable が例外を上げれば、`emit()` 呼び出しで発生した他の例外と同じ方法で、つまりハンドラの `handleError()` メソッドによって扱われます。

ローテート処理に大幅な変更を加える必要があれば、メソッドをオーバーライドすることができます。

例えば、cookbook-rotator-namer を参照してください。

16.8.6 RotatingFileHandler

logging.handlers モジュールに含まれる *RotatingFileHandler* クラスは、ディスク上のログファイルに対するローテーション処理をサポートします。

class logging.handlers.**RotatingFileHandler**(*filename*, *mode='a'*, *maxBytes=0*, *backupCount=0*, *encoding=None*, *delay=False*)

RotatingFileHandler クラスの新たなインスタンスを返します。指定されたファイルが開かれ、ログ記録のためのストリームとして使われます。*mode* が指定されなかった場合、`'a'` が使われます。*encoding* が `None` でない場合、その値はファイルを開くときのエンコーディングとして使われます。*delay* が真ならば、ファイルを開くのは最初の *emit()* 呼び出しまで遅らせられます。デフォルトでは、ファイルは無制限に大きくなりつづけます。

maxBytes および *backupCount* 値を指定することで、あらかじめ決められたサイズでファイルをロールオーバ (*rollover*) させることができます。指定サイズを超えそうになると、ファイルは閉じられ、暗黙のうちに新たなファイルが開かれます。ロールオーバは現在のログファイルの長さが *maxBytes* に近くなると常に起きますが、*maxBytes* または *backupCount* がゼロならロールオーバは起きなくなってしまうので、一般的には *backupCount* を少なくとも 1 に設定し *maxBytes* を非ゼロにするのが良いでしょう。*backupCount* が非ゼロのとき、システムは古いログファイルをファイル名に ".1", ".2" といった拡張子を追加して保存します。例えば、*backupCount* が 5 で、基本のファイル名が `app.log` なら、`app.log`, `app.log.1`, `app.log.2` ... と続き、`app.log.5` までを得ることになります。ログの書き込み対象になるファイルは常に `app.log` です。このファイルが満杯になると、ファイルは閉じられ、`app.log.1` に名前が変更されます。`app.log.1`, `app.log.2` などが存在する場合、それらのファイルはそれぞれ `app.log.2`, `app.log.3` といった具合に名前が変更されます。

バージョン 3.6 で変更: 文字列値に加え、*Path* オブジェクトも *filename* 引数が受け取るようになりました。

doRollover()

上述のような方法でロールオーバを行います。

emit(*record*)

上述のようなロールオーバを行いながら、レコードをファイルに出力します。

16.8.7 TimedRotatingFileHandler

logging.handlers モジュールに含まれる *TimedRotatingFileHandler* クラスは、特定の時間間隔でのログローテーションをサポートしています。

class logging.handlers.**TimedRotatingFileHandler**(*filename*, *when='h'*, *interval=1*, *backupCount=0*, *encoding=None*, *delay=False*, *utc=False*, *atTime=None*)

TimedRotatingFileHandler クラスの新たなインスタンスを返します。*filename* に指定したファイルを開き、ログ出力先のストリームとして使います。ログファイルのローテーション時には、ファイル名に拡張子 (suffix) をつけます。ログファイルのローテーションは *when* および *interval* の積に基づいて行います。

when は *interval* の単位を指定するために使います。使える値は下表の通りです。大小文字の区別は行いません。

値	*interval* の単位	*atTime* の使用有無/使用方法
`'S'`	秒	無視
`'M'`	分	無視
`'H'`	時間	無視
`'D'`	日	無視
`'W0'-'W6'`	曜日 (0=月曜)	初期のロールオーバー時刻の算出に使用
`'midnight'`	*atTime* が指定されなかった場合は深夜に、そうでない場合は *atTime* の時刻にロールオーバーされます	初期のロールオーバー時刻の算出に使用

曜日ベースのローテーションを使う場合は、月曜として 'W0' を、火曜として 'W1' を、…、日曜として 'W6' を指定します。このケースの場合は、*interval* は使われません。

古いログファイルの保存時、ロギングシステムによりファイル名に拡張子が付けられます。ロールオーバ間隔によって、strftime の `%Y-%m-%d_%H-%M-%S` 形式またはその前方の一部を使って、日付と時間に基づいた拡張子が付けられます。

最初に次のロールオーバー時間を計算するとき (ハンドラが生成されるとき)、次のローテーションがいつ起こるかを計算するために、既存のログファイルの最終変更時刻または現在の時間が使用されます。

utc 引数が true の場合時刻は UTC になり、それ以外では現地時間が使われます。

backupCount がゼロでない場合、保存されるファイル数は高々 *backupCount* 個で、それ以上のファイルがロールオーバされる時に作られるならば、一番古いものが削除されます。削除のロジックは interval で決まるファイルを削除するので、interval を変えると古いファイルが残ったままになることもあります。

delay が true なら、ファイルを開くのは *emit()* の最初の呼び出しまで延期されます。

atTime が `None` でない場合、それは `datetime.time` インスタンスでなければなりません。ロールオーバーが「夜中」「特定の曜日」に設定されていて、ロールが発生する時刻を指定します。*atTime* の値は *初期*のロールオーバーの計算に使われますが、後続のロールオーバーは通常の間隔の計算で算出されます。

注釈: 最初のロールオーバーの計算はハンドラーが初期化されたときに行われます。後続のロールオーバーは、ロールオーバーが発生し、ロールオーバーが出力した時にのみ行われます。これを念頭に置いておかないと混乱する可能性があります。例えば、インターバルが **毎分** に設定されていて、1 分ごとにログファイルが常に生成されるとは限りません。アプリケーションが実行されている間、1 分に 1 回以上のログ出力されて **いるとすると** 毎分、分割されたログファイルが出力されますが、そうでなく、5 分ごとに出力される場合は、出力されずにロールオーバーが実行されなかった時間分のギャップが生じます。

バージョン 3.4 で変更: *atTime* パラメータが追加されました。

バージョン 3.6 で変更: 文字列値に加え、*Path* オブジェクトも *filename* 引数が受け取るようになりました。

doRollover()
上述のような方法でロールオーバを行います。

emit(*record*)
上で説明した方法でロールオーバを行いながら、レコードをファイルに出力します。

16.8.8 SocketHandler

logging.handlers モジュールに含まれる *SocketHandler* クラスは、ログ出力をネットワークソケットに送信します。基底クラスでは TCP ソケットを用います。

class logging.handlers.**SocketHandler**(*host*, *port*)
アドレスが *host* および *port* で与えられた遠隔のマシンと通信するようにした *SocketHandler* クラスのインスタンスを生成して返します。

バージョン 3.4 で変更: `port` に `None` を指定すると、Unix ドメインソケットが `host` 値を用いて作られます - そうでない場合は TCP ソケットが作られます。

close()
ソケットを閉じます。

emit()
レコードの属性辞書を pickle して、バイナリ形式でソケットに書き込みます。ソケット操作でエラーが生じた場合、暗黙のうちにパケットは捨てられます。事前に接続が失われていた場合、接続を再度確立します。受信端でレコードを unpickle して *LogRecord* にするには、*makeLogRecord()* 関数を使ってください。

handleError()
emit() の処理中に発生したエラーを処理します。よくある原因は接続の消失です。次のイベント発生時に再試行できるようにソケットを閉じます。

makeSocket()
サブクラスで必要なソケット形式を詳細に定義できるようにするためのファクトリメソッドです。デフォルトの実装では、TCP ソケット (*socket.SOCK_STREAM*) を生成します。

makePickle(*record*)
レコードの属性辞書をバイナリ形式に pickle したものの先頭に長さ情報を付け、ソケットを介して送信できるようにして返します。この操作の詳細は次のコードと同等です:

```
data = pickle.dumps(record_attr_dict, 1)
datalen = struct.pack('>L', len(data))
return datalen + data
```

pickle が完全に安全というわけではないことに注意してください。セキュリティに関して心配なら、より安全なメカニズムを実装するためにこのメソッドをオーバーライドすると良いでしょう。例えば、HMAC を使って pickle に署名して、受け取る側ではそれを検証することができます。あるいはまた、受け取る側でグローバルなオブジェクトの unpickle を無効にすることができます。

send(*packet*)
pickle したバイト文字列 *packet* をソケットに送信します。送信するバイト文字列のフォーマットは、*makePickle()* のドキュメントで解説されています。

この関数はネットワークがビジーの時に発生する部分的送信に対応しています。

createSocket()
ソケットの生成を試みます。失敗時には、指数的な減速アルゴリズムを使います。最初の失敗時には、ハンドラは送ろうとしていたメッセージを落とします。続くメッセージが同じインスタンスで扱われたとき、幾らかの時間が経過するまで接続を試みません。デフォルトのパラメタは、最初の遅延時間が 1 秒で、その遅延時間の後でそれでも接続が確保できないなら、遅延時間は 2 倍づつになり、最大で 30 秒になります。

この働きは、以下のハンドラ属性で制御されます:

- `retryStart` (最初の遅延時間、デフォルトは 1.0 秒)。
- `retryFactor` (乗数、デフォルトは 2.0)。
- `retryMax` (最大遅延時間、デフォルトは 30.0 秒)。

つまり、ハンドラが使われた **後に** リモートリスナが起動した場合、メッセージが失われてしまうことがあります (ハンドラは、遅延時間が経過するまで接続を試みようとさえせず、その遅延時間中に通知なくメッセージを捨てるので)。

16.8.9 DatagramHandler

logging.handlers モジュールに含まれる *DatagramHandler* クラスは、*SocketHandler* を継承しており、UDP ソケットを介したログ記録メッセージの送信をサポートしています。

class logging.handlers.**DatagramHandler**(*host*, *port*)
アドレスが *host* および *port* で与えられた遠隔のマシンと通信するようにした *DatagramHandler* クラスのインスタンスを生成して返します。

バージョン 3.4 で変更: `port` に `None` を指定すると、Unix ドメインソケットが `host` 値を用いて作られます - そうでない場合は UDP ソケットが作られます。

emit()
レコードの属性辞書を pickle して、バイナリ形式でソケットに書き込みます。ソケット操作でエラーが生じた場合、暗黙のうちにパケットは捨てられます。事前に接続が失われていた場合、接続を再度確立します。受信端でレコードを unpickle して *LogRecord* にするには、*makeLogRecord()* 関数を使ってください。

makeSocket()

ここで *SocketHandler* のファクトリメソッドをオーバライドして、UDP ソケット (*socket.SOCK_DGRAM*) を生成しています。

send(*s*)

pickle したバイト文字列をソケットに送信します。送信するバイト文字列のフォーマットは、*SocketHandler.makePickle()* のドキュメントで解説されています。

16.8.10 SysLogHandler

logging.handlers モジュールに含まれる *SysLogHandler* クラスは、ログ記録メッセージを遠隔またはローカルの Unix syslog に送信する機能をサポートしています。

class logging.handlers.**SysLogHandler**(*address=('localhost', SYSLOG_UDP_PORT), facility=LOG_USER, socktype=socket.SOCK_DGRAM*)

遠隔の Unix マシンと通信するための、*SysLogHandler* クラスの新たなインスタンスを返します。マシンのアドレスは `(host, port)` のタプル形式をとる *address* で与えられます。*address* が指定されない場合、`('localhost', 514)` が使われます。アドレスは UDP ソケットを使って開かれます。`(host, port)` のタプル形式の代わりに文字列で "/dev/log" のように与えることもできます。この場合、Unix ドメインソケットが syslog にメッセージを送るのに使われます。*facility* が指定されない場合、`LOG_USER` が使われます。開かれるソケットの型は、*socktype* 引数に依り、デフォルトは *socket.SOCK_DGRAM* で、UDP ソケットを開きます。(rsyslog のような新しい syslog デーモンと使うために) TCP ソケットを開くには、*socket.SOCK_STREAM* の値を指定してください。

使用中のサーバが UDP ポート 514 を待機していない場合、*SysLogHandler* が正常に動作していないように見える場合があります。その場合、ドメインソケットに使うべきアドレスを調べてください。そのアドレスはシステムによって異なります。例えば、Linux システムでは通常 '/dev/log' ですが、OS X では '/var/run/syslog' です。プラットフォームを確認し、適切なアドレスを使う必要があります (アプリケーションを複数のプラットフォーム上で動作させる必要がある場合、実行時に確認する必要があるかもしれません)。Windows では、多くの場合、UDP オプションを使用する必要があります。

バージョン 3.2 で変更: *socktype* が追加されました。

close()

遠隔ホストへのソケットを閉じます。

emit(*record*)

レコードは書式化された後、syslog サーバに送信されます。例外情報が存在しても、サーバには **送信されません** 。

バージョン 3.2.1 で変更: (参照: bpo-12168) 初期のバージョンでは、syslog デーモンに送られるメッセージは常に NUL バイトで終端していました。初期のバージョンの syslog デーモンが NUL 終端されたメッセージを期待していたからです - たとえ、それが適切な仕様 (**RFC 5424**) にはなかったとしても。syslog デーモンの新しいバージョンは NUL バイトを期待せず、代わりにもしそれがある場合は削除します。さらに、より最近のデーモン (RFC 5424 により忠実なバージョン) は、メッセージの一部として NUL バイトを通します。

このような異なるデーモンの振る舞いすべてに対して syslog メッセージの取り扱いをより容易にするため、NUL バイトの追加はクラスレベル属性 `append_nul` を使用して設定できるようになりました。これはデフォルトで `True` (既存の振る舞いを保持) ですが、`SysLogHandler` インスタンスが NUL 終端文字を追加 **しない** ように `False` にセットすることができます。

バージョン 3.3 で変更: (参照: bpo-12419) 以前のバージョンでは、メッセージソースを識別するための "ident" あるいは "tag" プリフィックス機能はありませんでした。これは、今ではクラスレベル属性を使用して指定することができるようになりました。デフォルトでは既存の振る舞いを保持するために `""` ですが、特定の `SysLogHandler` インスタンスが扱うすべてのメッセージに識別子を前置するようにそれをオーバーライドすることができます。識別子はバイトではなくテキストでなければならず、正確にそのままメッセージに前置されることに注意してください。

`encodePriority`(*facility*, *priority*)

ファシリティおよび優先度を整数に符号化します。値は文字列でも整数でも渡すことができます。文字列が渡された場合、内部の対応付け辞書が使われ、整数に変換されます。

シンボリックな `LOG_` 値は *`SysLogHandler`* で定義されています。これは `sys/syslog.h` ヘッダーファイルで定義された値を反映しています。

優先度

名前 (文字列)	シンボル値
`alert`	LOG_ALERT
`crit` or `critical`	LOG_CRIT
`debug`	LOG_DEBUG
`emerg` or `panic`	LOG_EMERG
`err` or `error`	LOG_ERR
`info`	LOG_INFO
`notice`	LOG_NOTICE
`warn` or `warning`	LOG_WARNING

ファシリティ

名前 (文字列)	シンボル値
auth	LOG_AUTH
authpriv	LOG_AUTHPRIV
cron	LOG_CRON
daemon	LOG_DAEMON
ftp	LOG_FTP
kern	LOG_KERN
lpr	LOG_LPR
mail	LOG_MAIL
news	LOG_NEWS
syslog	LOG_SYSLOG
user	LOG_USER
uucp	LOG_UUCP
local0	LOG_LOCAL0
local1	LOG_LOCAL1
local2	LOG_LOCAL2
local3	LOG_LOCAL3
local4	LOG_LOCAL4
local5	LOG_LOCAL5
local6	LOG_LOCAL6
local7	LOG_LOCAL7

mapPriority(*levelname*)

ログレベル名を syslog 優先度名に対応付けます。カスタムレベルを使用している場合や、デフォルトアルゴリズムがニーズに適していない場合には、このメソッドをオーバーライドする必要があるかもしれません。デフォルトアルゴリズムは、DEBUG, INFO, WARNING, ERROR, CRITICAL を等価な syslog 名に、他のすべてのレベル名を "warning" に対応付けます。

16.8.11 NTEventLogHandler

logging.handlers モジュールに含まれる *NTEventLogHandler* クラスは、ログ記録メッセージをローカルな Windows NT, Windows 2000, または Windows XP のイベントログに送信する機能をサポートします。この機能を使えるようにするには、Mark Hammond による Python 用 Win32 拡張パッケージをインストールする必要があります。

class logging.handlers.**NTEventLogHandler**(*appname*, *dllname=None*, *logtype='Application'*)

NTEventLogHandler クラスの新たなインスタンスを返します。*appname* はイベントログに表示する際のアプリケーション名を定義するために使われます。この名前を使って適切なレジストリエントリが生成されます。*dllname* はログに保存するメッセージ定義の入った .dll または .exe ファイルへの完全修飾パス名を与えなければなりません (指定されない場合、'win32service.pyd' が使われます - このライブラリは Win32 拡張とともにインストールされ、いくつかのプレースホルダとなるメッセージ定義を含んでいます)。これらのプレースホルダを利用すると、メッセージの発信源全体がログに記録さ

れるため、イベントログは巨大になるので注意してください。*logtype* は 'Application', 'System', 'Security' のいずれかで、デフォルトは 'Application' です。

close()
現時点では、イベントログエントリの発信源としてのアプリケーション名をレジストリから除去することはできます。しかしこれを行うと、イベントログビューアで意図した通りにログが見えなくなるでしょう - これはイベントログが .dll 名を取得するためにレジストリにアクセスできなければならないからです。現在のバージョンではこの操作を行いません。

emit(*record*)
メッセージ ID、イベントカテゴリ、イベント型を決定し、メッセージを NT イベントログに記録します。

getEventCategory(*record*)
レコードに対するイベントカテゴリを返します。自作のカテゴリを指定したい場合、このメソッドをオーバライドしてください。このクラスのバージョンのメソッドは 0 を返します。

getEventType(*record*)
レコードのイベント型を返します。自作の型を指定したい場合、このメソッドをオーバライドしてください。このクラスのバージョンのメソッドは、ハンドラの *typemap* 属性を使って対応付けを行います。この属性は `__init__()` で初期化され、DEBUG, INFO, WARNING, ERROR, CRITICAL が入っています。自作のレベルを使っているのなら、このメソッドをオーバライドするか、ハンドラの *typemap* 属性に適切な辞書を配置する必要があるでしょう。

getMessageID(*record*)
レコードのメッセージ ID を返します。自作のメッセージを使っているのなら、ロガーに渡される *msg* を書式化文字列ではなく ID にします。その上で、辞書参照を行ってメッセージ ID を得ます。このクラスのバージョンでは 1 を返します。この値は `win32service.pyd` における基本メッセージ ID です。

16.8.12 SMTPHandler

logging.handlers モジュールに含まれる *SMTPHandler* クラスは、SMTP を介したログ記録メッセージの送信機能をサポートします。

class logging.handlers.**SMTPHandler**(*mailhost*, *fromaddr*, *toaddrs*, *subject*, *credentials=None*, *secure=None*, *timeout=1.0*)
新たな *SMTPHandler* クラスのインスタンスを返します。インスタンスは email の from および to アドレス行、および subject 行とともに初期化されます。*toaddrs* は文字列からなるリストでなければなりません。非標準の SMTP ポートを指定するには、*mailhost* 引数に (host, port) のタプル形式を指定します。文字列を使った場合、標準の SMTP ポートが使われます。もし SMTP サーバが認証を必要とするならば、(username, password) のタプル形式を *credentials* 引数に指定することができます。

セキュアプロトコル (TLS) の使用を指定するには *secure* 引数にタプルを渡してください。これは認証情報が渡された場合のみ使用されます。タプルは、空のタプルか、キーファイルの名前を持つ 1 要素のタプルか、またはキーファイルと証明書ファイルの名前を持つ 2 要素のタプルのいずれかでなければなりません。(このタプルは *smtplib.SMTP.starttls()* メソッドに渡されます。)

SMTP サーバとのコミュニケーションのために、*timeout* 引数を使用してタイムアウトを指定することができます。

バージョン 3.3 で追加: *timeout* 引数が追加されました。

emit(*record*)
レコードを書式化し、指定されたアドレスに送信します。

getSubject(*record*)
レコードに応じたサブジェクト行を指定したいなら、このメソッドをオーバライドしてください。

16.8.13 MemoryHandler

logging.handlers モジュールに含まれる *MemoryHandler* は、ログ記録するレコードをメモリ上にバッファリングし、定期的にその内容をターゲット (*target*) となるハンドラにフラッシュする機能をサポートしています。フラッシュ処理はバッファが一杯になるか、ある深刻度かそれ以上のレベルを持つイベントが観測された際に行われます。

MemoryHandler はより一般的な抽象クラス、*BufferingHandler* のサブクラスです。この抽象クラスでは、ログ記録するレコードをメモリ上にバッファリングします。各レコードがバッファに追加される毎に、`shouldFlush()` を呼び出してバッファをフラッシュすべきかどうか調べます。フラッシュする必要がある場合、`flush()` がフラッシュ処理を行うものと想定されます。

class logging.handlers.**BufferingHandler**(*capacity*)
容量つきのバッファーを設定してハンドラーが初期化されます。*capacity* はバッファ可能なログレコード数を意味します。

emit(*record*)
レコードをバッファに追加します。*shouldFlush()* が true を返す場合、バッファを処理するために *flush()* を呼び出します。

flush()
このメソッドをオーバライドして、自作のフラッシュ動作を実装することができます。このクラスのバージョンのメソッドでは、単にバッファの内容を削除して空にします。

shouldFlush(*record*)
バッファが許容量に達している場合に `True` を返します。このメソッドは自作のフラッシュ処理方針を実装するためにオーバライドすることができます。

class logging.handlers.**MemoryHandler**(*capacity*, *flushLevel=ERROR*, *target=None*, *flushOnClose=True*)
MemoryHandler クラスの新たなインスタンスを返します。インスタンスはサイズ *capacity* (バッファされるレコード数) のバッファとともに初期化されます。*flushLevel* が指定されていない場合、`ERROR` が使われます。*target* が指定されていない場合、ハンドラが何らかの意味のある処理を行う前に *setTarget()* でターゲットを指定する必要があります。*flushOnClose* が `False` に指定されていた場合、ハンドラが閉じられるときにバッファはフラッシュ **されません** 。*flushOnClose* が指定されていないか `True` に指定されていた場合、ハンドラが閉じられるときのバッファのフラッシュは以前の挙動になります。

バージョン 3.6 で変更: *flushOnClose* パラメータが追加されました。

close()
flush() を呼び出し、ターゲットを None に設定してバッファを消去します。

flush()
MemoryHandler の場合、フラッシュ処理は単に、バッファされたレコードをターゲットがあれば送信することを意味します。これと異なる動作を行いたい場合、オーバライドしてください。

setTarget(*target*)
ターゲットハンドラをこのハンドラに設定します。

shouldFlush(*record*)
バッファが一杯になっているか、*flushLevel* またはそれ以上のレコードでないかを調べます。

16.8.14 HTTPHandler

logging.handlers モジュールに含まれる *HTTPHandler* クラスは、ログ記録メッセージを GET または POST セマンティクスを使って Web サーバに送信する機能をサポートしています。

class logging.handlers.**HTTPHandler**(*host*, *url*, *method='GET'*, *secure=False*, *credentials=None*, *context=None*)
HTTPHandler クラスの新たなインスタンスを返します。特別なポートを使う必要がある場合、*host* は host:port の形式で使うことができます。*method* が指定されない場合、GET が使われます。*secure* が真の場合、HTTPS 接続が使われます。HTTPS 接続で使用する SSL 設定のために *context* 引数を *ssl.SSLContext* のインスタンスに設定することができます。*credentials* を指定する場合、BASIC 認証の際の HTTP 'Authorization' ヘッダに使われるユーザ ID とパスワードからなる 2 要素タプルを渡してください。credentials を指定する場合、ユーザ ID とパスワードが通信中に平文として剥き出しにならないよう、secure=True も指定すべきです。

バージョン 3.5 で変更: *context* パラメータが追加されました。

mapLogRecord(*record*)
URL エンコードされて Web サーバに送信することになる、record に基づく辞書を供給します。デフォルトの実装では単に record.__dict__ を返します。例えば *LogRecord* のサブセットのみを Web サーバに送信する場合や、サーバーに送信する内容を特別にカスタマイズする必要がある場合には、このメソッドをオーバライドできます。

emit(*record*)
レコードを URL エンコードされた辞書形式で Web サーバに送信します。レコードを送信のために辞書に変換するために *mapLogRecord()* が呼び出されます。

注釈: Web server に送信するためのレコードを準備することは一般的な書式化操作とは同じではありませんので、*setFormatter()* を使って *Formatter* を指定することは、*HTTPHandler* には効果はありません。*format()* を呼び出す代わりに、このハンドラは *mapLogRecord()* を呼び出し、その後その返却辞書を Web server に送信するのに適した様式にエンコードするために *urllib.parse.urlencode()*

を呼び出します。

16.8.15 QueueHandler

バージョン 3.2 で追加.

logging.handlers モジュールに含まれる *QueueHandler* クラスは、*queue* モジュールや *multiprocessing* のモジュールで実装されるようなキューにログメッセージを送信する機能をサポートしています。

QueueListener クラスとともに *QueueHandler* を使うと、ロギングを行うスレッドから分離されたスレッド上でハンドラを動かすことができます。これは、クライアントに対してサービスするスレッドができるだけ速く応答する必要がある一方、別のスレッド上で (*SMTPHandler* によって電子メールを送信するような) 潜在的に遅い操作が行われるような、ウェブアプリケーションおよびその他のサービスアプリケーションにおいて重要です。

class logging.handlers.**QueueHandler**(*queue*)

QueueHandler クラスの新しいインスタンスを返します。インスタンスは、キューにメッセージを送るように初期化されます。キューは任意のキューのようなオブジェクトが可能です; それはそのまま *enqueue()* メソッドによって使用されます。そのメソッドはメッセージを送る方法を知っている必要があります。もしタスクのトラッキング API にキューが必要でない場合はキューとして *SimpleQueue* のインスタンスが使用できます。

emit(*record*)

LogRecord の準備結果をキューに入れます。容量制限のあるキューがいっぱいになったなど、例外が発生した場合は、エラーを処理するために *handleError()* メソッドが呼ばれます。これにより、レコードが応答なく消滅したり（もし `logging.raiseExceptions` が `False` の場合)、`sys.stderr` に出力されます（もし `logging.raiseExceptions` が `True` の場合)。

prepare(*record*)

キューに追加するためレコードを準備します。このメソッドが返したオブジェクトがキューに追加されます。

メッセージと、引数と、もしあれば例外の情報を合成するためにレコードを書式化して、レコードから pickle 不可能なアイテムを in-place で取り除くベース実装です。

レコードを dict や JSON 文字列に変換したい場合や、オリジナルのレコードを変更せずに修正済のコピーを送りたい場合は、このメソッドをオーバーライドすると良いでしょう。

enqueue(*record*)

キューにレコードを `put_nowait()` を使ってエンキューします; ブロッキングやタイムアウト、あるいはなにか特別なキューの実装を使いたければ、これをオーバライドしてみてください。

16.8.16 QueueListener

バージョン 3.2 で追加.

logging.handlers モジュールに含まれる *QueueListener* クラスは、*queue* モジュールや *multiprocessing* のモジュールで実装されるようなキューからログメッセージを受信する機能をサポートしています。メッセージは内部スレッドのキューから受信され、同じスレッド上の複数のハンドラに渡されて処理されます。*QueueListener* それ自体はハンドラではありませんが、*QueueHandler* と連携して動作するのでここで文書化されています。

QueueHandler クラスとともに *QueueListener* を使うと、ロギングを行うスレッドから分離されたスレッド上でハンドラを動かすことができます。これは、クライアントに対してサービスするスレッドができるだけ速く応答する必要がある一方、別のスレッド上で (*SMTPHandler* によって電子メールを送信するような) 潜在的に遅い操作が行われるような、ウェブアプリケーションおよびその他のサービスアプリケーションにおいて重要です。

class logging.handlers.**QueueListener**(*queue*, **handlers*, *respect_handler_level=False*)

QueueListener クラスの新しいインスタンスを返します。インスタンスは、メッセージを送るためのキューと、キューに格納されたエントリーを処理するハンドラーのリストが設定されて初期化されます。キューは任意のキューのようなオブジェクトが可能です; それはそのまま *dequeue()* メソッドによって使用されます。そのメソッドはメッセージを送る方法を知っている必要があります。もしタスクのトラッキング API にキューが必要でない場合はキューとして *SimpleQueue* のインスタンスが使用できます（トラッキング API が使用可能な場合は使用されます)。

もし `respect_handler_level` が `True` なら、メッセージをハンドラーに渡すか決める時に（メッセージのレベルに比べて）ハンドラーのレベルが尊重されます。そうでない場合は依然の Python バージョンと同様にすべてのメッセージは常にハンドラーに渡されます。

バージョン 3.5 で変更: The `respect_handler_level` argument was added.

dequeue(*block*)

キューからレコードを取り除き、それを返します。ブロッキングすることがあります。

ベース実装は `get()` を使用します。タイムアウトを有効にしたい場合や、カスタムのキュー実装を使いたい場合は、このメソッドをオーバーライドすると良いでしょう。

prepare(*record*)

レコードを扱うための準備をします。

この実装は渡されたレコードをそのまま返します。その値をハンドラに渡す前に何らかのカスタムな整列化 (marshalling) あるいはレコードに対する操作を行う必要があれば、このメソッドをオーバーライドすると良いでしょう。

handle(*record*)

レコードを処理します。

これは、ハンドラをループしてそれらに処理すべきレコードを渡します。ハンドラに渡される実際のオブジェクトは、*prepare()* から返されたものです。

start()
リスナーを開始します。

これは、LogRecord を処理するキューを監視するために、バックグラウンドスレッドを開始します。

stop()
リスナーを停止します。

スレッドに終了するように依頼し、終了するまで待ちます。アプリケーションの終了前にこのメソッドを呼ばないと、いくつかのレコードがキューに残り、処理されなくなるかもしれないことに注意してください。

enqueue_sentinel()
リスナーに停止するように指示するためキューに番兵を書き込みます。この実装は put_nowait() を使用します。タイムアウトを有効にしたい場合や、カスタムのキュー実装を使いたい場合は、このメソッドをオーバーライドすると良いでしょう。

バージョン 3.3 で追加.

参考:

logging **モジュール** logging モジュールの API リファレンス。

logging.config **モジュール** logging モジュールの環境設定 API です。

16.9 getpass --- 可搬性のあるパスワード入力機構

ソースコード: Lib/getpass.py

---

getpass モジュールは二つの関数を提供します:

getpass.getpass(*prompt='Password: '*, *stream=None*)
エコーなしでユーザーにパスワードを入力させるプロンプト。ユーザーは *prompt* の文字列をプロンプトに使え、デフォルトは 'Password: ' です。Unix ではプロンプトはファイルに似たオブジェクト *stream* へ、必要なら置き換えられたエラーハンドラを使って出力されます。*stream* のデフォルトは、制御端末 (/dev/tty) か、それが利用できない場合は sys.stderr です (この引数は Windows では無視されます)。

もしエコーなしで入力が利用できない場合は、getpass() は *stream* に警告メッセージを出力し、sys.stdin から読み込み、*GetPassWarning* 警告を発生させます。

注釈: IDLE から getpass を呼び出した場合、入力は IDLE のウィンドウではなく、IDLE を起動したターミナルから行われます。

---

exception getpass.GetPassWarning

UserWarning のサブクラスで、入力がエコーされてしまった場合に発生します。

getpass.getuser()

ユーザーの ” ログイン名” を返します。

この関数は環境変数 LOGNAME USER LNAME USERNAME の順序でチェックして、最初の空ではない文字列が設定された値を返します。もし、なにも設定されていない場合は *pwd* モジュールが提供するシステム上のパスワードデータベースから返します。それ以外は、例外が上がります。

一般的に、この関数は *os.getlogin()* よりも優先されるべきです。

16.10 curses --- 文字セル表示を扱うための端末操作

curses モジュールは、可搬性のある高度な端末操作のデファクトスタンダードである、curses ライブラリへのインタフェースを提供します。

curses が最も広く用いられているのは Unix 環境ですが、Windows、DOS で利用できるバージョンもあり、おそらく他のシステムで利用できるバージョンもあります。この拡張モジュールは Linux および BSD 系の Unix で動作するオープンソースの curses ライブラリである ncurses の API に合致するように設計されています。

注釈: Whenever the documentation mentions a *character* it can be specified as an integer, a one-character Unicode string or a one-byte byte string.

Whenever the documentation mentions a *character string* it can be specified as a Unicode string or a byte string.

注釈: version 5.4 から、ncurses ライブラリは nl_langinfo 関数を利用して非 ASCII データをどう解釈するかを決定するようになりました。これは、アプリケーションは *locale.setlocale()* 関数を呼び出して、Unicode 文字列をシステムの利用可能なエンコーディングのどれかでエンコードする必要があることを意味します。この例では、システムのデフォルトエンコーディングを利用しています:

```
import locale
locale.setlocale(locale.LC_ALL, '')
code = locale.getpreferredencoding()
```

この後、*str.encode()* を呼び出すときに *code* を利用します。

参考:

curses.ascii **モジュール** ロケール設定に関わらず ASCII 文字を扱うためのユーティリティ。

curses.panel **モジュール**　curses ウィンドウにデプス機能を追加するパネルスタック拡張。

curses.textpad **モジュール**　Emacs ライクなキーバインディングをサポートする編集可能な curses 用テキストウィジェット。

curses-howto　Andrew Kuchling および Eric Raymond によって書かれた、curses を Python で使うためのチュートリアルです。

Python ソースコードの Tools/demo/ ディレクトリには、このモジュールで提供されている curses バインディングを使ったプログラム例がいくつか収められています。

16.10.1 関数

curses モジュールでは以下の例外を定義しています:

exception curses.error
curses ライブラリ関数がエラーを返した際に送出される例外です。

注釈: 関数やメソッドにおけるオプションの引数 *x* および *y* がある場合、デフォルト値は常に現在のカーソルになります。オプションの *attr* がある場合、デフォルト値は `A_NORMAL` です。

---

curses では以下の関数を定義しています:

curses.baudrate()
端末の出力速度をビット/秒で返します。ソフトウェア端末エミュレータの場合、これは固定の高い値を持つことになります。この関数は歴史的な理由で入れられています; かつては、この関数は時間遅延を生成するための出力ループを書くために用いられたり、行速度に応じてインタフェースを切り替えたりするために用いられたりしていました。

curses.beep()
注意を促す短い音を鳴らします。

curses.can_change_color()
端末に表示される色をプログラマが変更できるか否かによって、`True` または `False` を返します。

curses.cbreak()
cbreak モードに入ります。cbreak モード ("rare" モードと呼ばれることもあります) では、通常の tty 行バッファリングはオフにされ、文字を一文字一文字読むことができます。ただし、raw モードとは異なり、特殊文字 (割り込み:interrupt、終了:quit、一時停止:suspend、およびフロー制御) については、tty ドライバおよび呼び出し側のプログラムに対する通常の効果をもっています。まず *raw()* を呼び出し、次いで *cbreak()* を呼び出すと、端末を cbreak モードにします。

curses.color_content(*color_number*)
色 *color_number* の赤、緑、および青 (RGB) 要素の強度を返します。*color_number* は `0` から `COLORS - 1` の間でなければなりません。与えられた色の R、G、B、の値からなる三要素のタプルが返されます。この値は `0` (その成分はない) から `1000` (その成分の最大強度) の範囲をとります。

curses.color_pair(*pair_number*)

Return the attribute value for displaying text in the specified color pair. Only the first 256 color pairs are supported. This attribute value can be combined with A_STANDOUT, A_REVERSE, and the other A_* attributes. *pair_number()* is the counterpart to this function.

curses.curs_set(*visibility*)

カーソルの状態を設定します。*visibility* は 0 、1``または ``2 に設定され、それぞれ不可視、通常、または非常に可視、を意味します。要求された可視属性を端末がサポートしている場合、以前のカーソル状態が返されます; そうでなければ例外が送出されます。多くの端末では、” 可視 (通常)” モードは下線カーソルで、” 非常に可視” モードはブロックカーソルです。

curses.def_prog_mode()

現在の端末属性を、稼動中のプログラムが curses を使う際のモードである ” プログラム” モードとして保存します。(このモードの反対は、プログラムが curses を使わない ” シェル” モードです。) その後 *reset_prog_mode()* を呼ぶとこのモードを復旧します。

curses.def_shell_mode()

現在の端末属性を、稼動中のプログラムが curses を使っていないときのモードである ” シェル” モードとして保存します。(このモードの反対は、プログラムが curses 機能を利用している ” プログラム” モードです。) その後 *reset_shell_mode()* を呼ぶとこのモードを復旧します。

curses.delay_output(*ms*)

出力に *ms* ミリ秒の一時停止を入れます。

curses.doupdate()

物理スクリーンを更新します。curses ライブラリは、現在の物理スクリーンの内容と、次の状態として要求されている仮想スクリーンをそれぞれ表す、2 つのデータ構造を保持しています。*doupdate()* は更新を適用し、物理スクリーンを仮想スクリーンに一致させます。

仮想スクリーンは *addstr()* のような書き込み操作をウィンドウに行った後に *noutrefresh()* を呼び出して更新することができます。通常の:meth:*~window.refresh* 呼び出しは、単に noutrefresh() を呼んだ後に doupdate`を呼ぶだけです; 複数のウィンドウを更新しなければならない場合、すべてのウィンドウに対して :meth:`()!noutrefresh‘ を呼び出した後、一度だけ doupdate() を呼ぶことで、パフォーマンスを向上させることができ、おそらくスクリーンのちらつきも押さえることができます。

curses.echo()

echo モードに入ります。echo モードでは、各文字入力はスクリーン上に入力された通りにエコーバックされます。

curses.endwin()

ライブラリの非初期化を行い、端末を通常の状態に戻します。

curses.erasechar()

ユーザの現在の消去文字 (erase character) を 1 バイトの bytes オブジェクトで返します。Unix オペレーティングシステムでは、この値は curses プログラムが制御している端末の属性であり、curses ライブラリ自体では設定されません。

curses.filter()

filter() ルーチンを使う場合、*initscr()* を呼ぶ前に呼び出さなくてはなりません。この手順のもたらす効果は以下の通りです: まず二つの関数の呼び出しの間は、LINES は 1 に設定されます; clear, cup, cud, cud1, cuu1, cuu, vpa は無効化されます; home 文字列は cr の値に設定されます。これにより、カーソルは現在の行に制限されるので、スクリーンの更新も同様に制限されます。この関数は、スクリーンの他の部分に影響を及ぼさずに文字単位の行編集を行う場合に利用できます。

curses.**flash**()
スクリーンを点滅します。すなわち、画面を色反転して、短時間でもとにもどします。人によっては、*beep()* で生成される注意音よりも、このような "目に見えるベル" を好みます。

curses.**flushinp**()
すべての入力バッファをフラッシュします。この関数は、ユーザによってすでに入力されているが、まだプログラムによって処理されていないすべての先行入力文字を破棄します。

curses.**getmouse**()
getch() が KEY_MOUSE を返してマウスイベントを通知した後、この関数を呼んで待ち行列上に置かれているマウスイベントを取得しなければなりません。イベントは (id, x, y, z, bstate) の 5 要素のタプルで表現されています。*id* は複数のデバイスを区別するための ID 値で、*x*, *y*, *z* はイベントの座標値です。(現在 *z* は使われていません) *bstate* は整数値で、その各ビットはイベントのタイプを示す値に設定されています。この値は以下に示す定数のうち一つまたはそれ以上のビット単位 OR になっています。以下の定数の *n* は 1 から 4 のボタン番号を示します: BUTTONn_PRESSED, BUTTONn_RELEASED, BUTTONn_CLICKED, BUTTONn_DOUBLE_CLICKED, BUTTONn_TRIPLE_CLICKED, BUTTON_SHIFT, BUTTON_CTRL, BUTTON_ALT。

curses.**getsyx**()
仮想スクリーンにおける現在のカーソル位置を (y, x) のようなタプルで返します。leaveok() が True に設定されていれば、(-1, -1) が返されます。

curses.**getwin**(*file*)
以前の putwin() 呼び出しでファイルに保存されている、ウィンドウ関連データを読み出します。次に、このルーチンはそのデータを使って新たなウィンドウを生成し初期化して、その新規ウィンドウオブジェクトを返します。

curses.**has_colors**()
端末が色表示を行える場合には True を返します。そうでない場合には False を返します。

curses.**has_ic**()
端末が文字の挿入/削除機能を持つ場合に True を返します。最近の端末エミュレータはどれもこの機能を持っており、この関数は歴史的な理由のためだけに存在しています。

curses.**has_il**()
端末が行の挿入/削除機能を持つ場合に True を返します。最近の端末エミュレータはどれもこの機能を持っていて、この関数は歴史的な理由のためだけに存在しています。

curses.**has_key**(*ch*)
キー値 *ch* をとり、現在の端末タイプがその値のキーを認識できる場合に True を返します。

curses.**halfdelay**(*tenths*)

半遅延モード、すなわち cbreak モードに似た、ユーザが打鍵した文字がすぐにプログラムで利用できるようになるモードで使われます。しかしながら、何も入力されなかった場合、十分の *tenths* 秒後に例外が送出されます。*tenths* の値は 1 から 255 の間でなければなりません。半遅延モードから抜けるには *nocbreak()* を使います。

curses.**init_color**(*color_number*, *r*, *g*, *b*)

色の定義を変更します。変更したい色番号と、その後に 3 つ組みの RGB 値 (赤、緑、青の成分の大きさ) をとります。*color_number* の値は 0 から *COLORS - 1* の間でなければなりません。*r*, *g*, *b* の値は 0 から 1000 の間でなければなりません。*init_color()* を使うと、スクリーン上でカラーが使用されている部分はすべて新しい設定に即時変更されます。この関数はほとんどの端末で何も行いません; *can_change_color()* が True を返す場合にのみ動作します。

curses.**init_pair**(*pair_number*, *fg*, *bg*)

色ペアの定義を変更します。3 つの引数: 変更したい色ペア、前景色の色番号、背景色の色番号、をとります。*pair_number* は 1 から COLOR_PAIRS - 1 の間でなければなりません (0 色ペアは黒色背景に白色前景となるように設定されており、変更することができません)。*fg* および *bg* 引数は 0 と COLORS - 1 の間、または、*use_default_colors()* を呼び出した後では -1 でなければなりません。色ペアが以前に初期化されていれば、スクリーンを更新して、指定された色ペアの部分を新たな設定に変更します。

curses.**initscr**()

ライブラリを初期化します。スクリーン全体をあらわす *window* オブジェクトを返します。

注釈: 端末のオープン時にエラーが発生した場合、curses ライブラリによってインタープリタが終了される場合があります。

---

curses.**is_term_resized**(*nlines*, *ncols*)

resize_term() によってウィンドウ構造が変更されている場合に True を、そうでない場合は False を返します。

curses.**isendwin**()

endwin() がすでに呼び出されている (すなわち、curses ライブラリが非初期化されてしまっている) 場合に True を返します。

curses.**keyname**(*k*)

bytes オブジェクト *k* に番号付けされているキーの名前を返します。印字可能な ASCII 文字を生成するキーの名前はそのキーの文字自体になります。コントロールキーと組み合わせたキーの名前は、キャレット (b'^') の後に対応する ASCII 文字が続く 2 バイトの bytes オブジェクトになります。Alt キーと組み合わせたキー (128--255) の名前は、先頭に b'M-' が付き、その後に対応する ASCII 文字が続く bytes オブジェクトになります。

curses.**killchar**()

Return the user's current line kill character as a one-byte bytes object. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses.longname()
現在の端末について記述している terminfo の長形式 name フィールドが入った bytes オブジェクトを返します。verbose 形式記述の最大長は 128 文字です。この値は *initscr()* 呼び出しの後でのみ定義されています。

curses.meta(*flag*)
flag が True の場合、8 ビット文字の入力を許可します。*flag* が False の場合、7 ビット文字だけを許可します。

curses.mouseinterval(*interval*)
ボタンが押されてから離されるまでの時間をマウスクリック一回として認識する最大の時間間隔をミリ秒で設定します。返り値は以前の内部設定値になります。デフォルトは 200 ミリ秒 (5 分の 1 秒) です。

curses.mousemask(*mousemask*)
報告すべきマウスイベントを設定し、(availmask, oldmask) の組からなるタプルを返します。*availmask* はどの指定されたマウスイベントのどれが報告されるかを示します; どのイベント指定も完全に失敗した場合には 0 が返ります。*oldmask* は与えられたウィンドウの以前のマウスイベントマスクです。この関数が呼ばれない限り、マウスイベントは何も報告されません。

curses.napms(*ms*)
ms ミリ秒間スリープします。

curses.newpad(*nlines*, *ncols*)
与えられた行とカラム数を持つパッド (pad) データ構造を生成し、そのポインタを返します。パッドはウィンドウオブジェクトとして返されます。

パッドはウィンドウと同じようなものですが、スクリーンのサイズによる制限をうけず、スクリーンの特定の部分に関連付けられていなくてもかまいません。大きなウィンドウが必要であり、スクリーンにはそのウィンドウの一部しか一度に表示しない場合に使えます。(スクロールや入力エコーなどによる) パッドに対する再描画は起こりません。パッドに対する *refresh()* および :meth:‘~window.noutrefresh‘メソッドは、パッド中の表示する部分と表示するために利用するスクリーン上の位置を指定する 6 つの引数が必要です。これらの引数は *pminrow*、*pmincol*、*sminrow*、*smincol*、*smaxrow*、*smaxcol* です; *p* で始まる引数はパッド中の表示領域の左上位置で、*s* で始まる引数はパッド領域を表示するスクリーン上のクリップ矩形を指定します。

curses.newwin(*nlines*, *ncols*)
curses.newwin(*nlines*, *ncols*, *begin_y*, *begin_x*)
左上の角が (begin_y, begin_x) で、高さ/幅が *nlines* / *ncols* の新規 **ウィンドウ** を返します。

デフォルトでは、ウィンドウは指定された位置からスクリーンの右下まで広がります。

curses.nl()
newlime モードに入ります。このモードはリターンキーを入力中の改行として変換し、出力時に改行文字を復帰 (return) と改行 (line-feed) に変換します。newline モードは初期化時にはオンになっています。

curses.nocbreak()
cbreak モードを終了します。行バッファリングを行う通常の ”cooked” モードに戻ります。

curses.**noecho**()
: echo モードを終了します。入力のエコーバックはオフにされます。

curses.**nonl**()
: newline モードを終了します。入力時のリターンキーから改行への変換、および出力時の改行から復帰/改行への低レベル変換を無効化します (ただし、`addch('\n')` の振る舞いは変更せず、仮想スクリーン上では常に復帰と改行に等しくなります)。変換をオフにすることで、curses は水平方向の動きを少しだけ高速化できることがあります; また、入力中のリターンキーの検出ができるようになります。

curses.**noqiflush**()
: `noqiflush()` ルーチンを使うと、通常行われている `INTR`、`QUIT` および `SUSP` 文字による入力および出力キューのフラッシュが行われなくなります。シグナルハンドラが終了した際、割り込みが発生しなかったかのように出力を続たい場合、ハンドラ中で `noqiflush()` を呼び出すことができます。

curses.**noraw**()
: raw モードから離れます。行バッファリングを行う通常の "cooked" モードに戻ります。

curses.**pair_content**(*pair_number*)
: 要求された色ペアの色を含むタプル `(fg, bg)` を返します。*pair_number* は `0` から `COLOR_PAIRS - 1` の間でなければなりません。

curses.**pair_number**(*attr*)
: *attr* に対する色ペアセットの番号を返します。*color_pair()* はこの関数の逆に相当します。

curses.**putp**(*str*)
: `tputs(str, 1, putchar)` と等価です; 現在の端末における、指定された terminfo 機能の値を出力します。*putp()* の出力は常に標準出力に送られるので注意して下さい。

curses.**qiflush**([*flag*])
: *flag* が `False` なら、*noqiflush()* を呼ぶのとと同じ効果です。*flag* が `True` か、引数が与えられていない場合、制御文字が読み出された最にキューはフラッシュされます。

curses.**raw**()
: raw モードに入ります。raw モードでは、通常の行バッファリングと割り込み (interrupt)、終了 (quit)、一時停止 (suspend)、およびフロー制御キーはオフになります; 文字は curses 入力関数に一文字づつ渡されます。

curses.**reset_prog_mode**()
: 端末を "program" モードに復旧し、あらかじめ *def_prog_mode()* で保存した内容に戻します。

curses.**reset_shell_mode**()
: 端末を "shell" モードに復旧し、あらかじめ *def_shell_mode()* で保存した内容に戻します。

curses.**resetty**()
: 端末モードの状態を最後に *savetty()* を呼び出した時の状態に戻します。

curses.**resize_term**(*nlines*, *ncols*)
: *resizeterm()* で使用されるバックエンド関数で、ウィンドウサイズを変更する時、*resize_term()* は拡張された領域を埋めます。呼び出したアプリケーションはそれらの領域を適切なデータで埋めなく

てはなりません。resize_term() 関数はすべてのウィンドウのサイズ変更を試みます。ただし、パッド呼び出しの慣例により、アプリケーションとの追加のやり取りを行わないサイズ変更は行えません。

curses.**resizeterm**(*nlines*, *ncols*)

現在の標準ウィンドウのサイズを指定された寸法に変更し、curses ライブラリが使用する、その他のウィンドウサイズを記憶しているデータ (特に SIGWINCH ハンドラ) を調整します。

curses.**savetty**()

resetty() で使用される、バッファ内の端末モードの現在の状態を保存します。

curses.**setsyx**(*y*, *x*)

仮想スクリーンのカーソルを *y*, *x* に設定します。*y* および *x* がどちらも -1 の場合、:meth:‘leaveok ‘ が に “True“に設定されます。

curses.**setupterm**(*term=None*, *fd=-1*)

端末を初期化します。*term* は端末名となる文字列または None です。省略または None の場合、環境変数 TERM の値が使用されます。*fd* は送信される初期化シーケンスへのファイル記述子です。指定されないまたは -1 の場合、sys.stdout のファイル記述子が使用されます。

curses.**start_color**()

プログラマがカラーを利用したい場合で、かつ他の何らかのカラー操作ルーチンを呼び出す前に呼び出さなくてはなりません。この関数は *initscr()* を呼んだ直後に呼ぶようにしておくとよいでしょう。

start_color() は 8 つの基本色 (黒、赤、緑、黄、青、マゼンタ、シアン、および白) と、色数の最大値と端末がサポートする色ペアの最大数が入っている、*curses* モジュールにおける二つのグローバル変数、COLORS および COLOR_PAIRS を初期化します。この関数はまた、色設定を端末のスイッチが入れられたときの状態に戻します。

curses.**termattrs**()

端末がサポートするすべてのビデオ属性を論理和した値を返します。この情報は、curses プログラムがスクリーンの見え方を完全に制御する必要がある場合に便利です。

curses.**termname**()

14 文字以下になるように切り詰められた環境変数 TERM の値を bytes オブジェクトで返します。

curses.**tigetflag**(*capname*)

terminfo 機能名 *capname* に対応する真偽値の機能値を整数で返します。*capname* が真偽値で表せる機能値でない場合 -1 を返し、機能がキャンセルされているか、端末記述上に見つからない場合 0 を返します。

curses.**tigetnum**(*capname*)

terminfo 機能名 *capname* に対応する数値の機能値を整数で返します。*capname* が数値で表せる機能値でない場合 -2 を返し、機能がキャンセルされているか、端末記述上に見つからない場合 -1 を返します。

curses.**tigetstr**(*capname*)

Return the value of the string capability corresponding to the terminfo capability name *capname* as a bytes object. Return None if *capname* is not a terminfo ”string capability”, or is canceled or absent from the terminal description.

curses.**tparm**(*str*[, ...])
str を与えられたパラメタを使って bytes オブジェクトにインスタンス化します。*str* は terminfo データベースから得られたパラメタを持つ文字列でなければなりません。例えば、`tparm(tigetstr("cup"), 5, 3)` は `b'\033[6;4H'` のようになります。厳密には端末の形式によって異なる結果となります。

curses.**typeahead**(*fd*)
先読みチェックに使うためのファイル記述子 *fd* を指定します。*fd* が `-1` の場合、先読みチェックは行われません。

curses ライブラリはスクリーンを更新する間、先読み文字列を定期的に検索することで ” 行はみ出し最適化 (line-breakout optimization)” を行います。入力が得られ、かつ入力は端末からのものである場合、現在行おうとしている更新は refresh や doupdate を再度呼び出すまで先送りにします。この関数は異なるファイル記述子で先読みチェックを行うように指定することができます。

curses.**unctrl**(*ch*)
Return a bytes object which is a printable representation of the character *ch*. Control characters are represented as a caret followed by the character, for example as `b'^C'`. Printing characters are left as they are.

curses.**ungetch**(*ch*)
ch をプッシュし、次に *getch()* を呼び出した時にその値が返るようにします。

注釈: `getch()` を呼び出すまでは *ch* は一つしかプッシュできません。

---

curses.**update_lines_cols**()
`LINES` と `COLS` についての更新。マニュアルでスクリーンのサイズを変更したことを検知するために有用です。

バージョン 3.5 で追加.

curses.**unget_wch**(*ch*)
ch をプッシュし、次に *get_wch()* を呼び出した時にその値が返るようにします。

注釈: `get_wch()` を呼び出すまでは *ch* は一つしかプッシュできません。

バージョン 3.3 で追加.

curses.**ungetmouse**(*id*, *x*, *y*, *z*, *bstate*)
与えられた状態データが関連付けられた `KEY_MOUSE` イベントを入力キューにプッシュします。

curses.**use_env**(*flag*)
この関数を使う場合、*initscr()* または newterm を呼ぶ前に呼び出さなくてはなりません。*flag* が `False` の場合、環境変数 `LINES` および `COLUMNS` の値 (デフォルトで使用されます) の値が設定されていたり、curses がウィンドウ内で動作して (この場合 `LINES` や `COLUMNS` が設定されていないとウィンドウのサイズを使います) いても、terminfo データベースに指定された lines および columns の値を

使います。

curses.use_default_colors()
この機能をサポートしている端末上で、色の値としてデフォルト値を使う設定をします。あなたのアプリケーションで透過性とサポートするためにこの関数を使ってください。デフォルトの色は色番号 -1 に割り当てられます。この関数を呼んだ後、たとえば init_pair(x, curses.COLOR_RED, -1) は色ペア *x* を赤い前景色とデフォルトの背景色に初期化します。

curses.wrapper(*func*, ...)
curses を初期化し、呼び出し可能なオブジェクト *func* (その他の curses を使用するアプリケーション) を呼び出します。アプリケーションが例外を送出した場合、この関数は端末を例外を再送出する前に正常な状態に戻し、トレースバックを作成します。その後、呼び出し可能オブジェクト *func* には、第一引数としてメインウィンドウ 'stdscr' が、続いて wrapper() に渡されたその他の引数が渡されます。*func* を呼び出す前、wrapper() は cbreak モードをオンに、エコーをオフに、端末キーパッドを有効にし、端末が色表示をサポートしている場合は色を初期化します。終了時 (通常終了、例外による終了のどちらでも)、cooked モードに戻し、エコーをオンにし、端末キーパッドを無効にします。

16.10.2 Window オブジェクト

上記の *initscr()* や *newwin()* が返すウィンドウは、以下のメソッドと属性を持ちます:

window.addch(*ch*[, *attr*])
window.addch(*y*, *x*, *ch*[, *attr*])
(y, x) にある文字 *ch* を属性 *attr* で描画します。このときその場所に以前描画された文字は上書きされます。デフォルトでは、文字の位置および属性はウィンドウオブジェクトにおける現在の設定になります。

注釈: Writing outside the window, subwindow, or pad raises a *curses.error*. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the character is printed.

---

window.addnstr(*str*, *n*[, *attr*])
window.addnstr(*y*, *x*, *str*, *n*[, *attr*])
文字列 *str* から最大で *n* 文字を (y, x) に属性 *attr* で描画します。以前ディスプレイにあった内容はすべて上書きされます。

window.addstr(*str*[, *attr*])
window.addstr(*y*, *x*, *str*[, *attr*])
(y, x) に文字列 *str* を属性 *attr* で描画します。以前ディスプレイにあった内容はすべて上書きされます。

注釈:

- Writing outside the window, subwindow, or pad raises *curses.error*. Attempting to write

to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the string is printed.

- A bug in ncurses, the backend for this Python module, can cause SegFaults when resizing windows. This is fixed in ncurses-6.1-20190511. If you are stuck with an earlier ncurses, you can avoid triggering this if you do not call *addstr()* with a *str* that has embedded newlines. Instead, call *addstr()* separately for each line.

---

window.**attroff**(*attr*)

現在のウィンドウに書き込まれたすべての内容に対し "バックグラウンド" に設定された属性 *attr* を除去します。

window.**attron**(*attr*)

現在のウィンドウに書き込まれたすべての内容に対し "バックグラウンド" に属性 *attr* を追加します。

window.**attrset**(*attr*)

"バックグラウンド" の属性セットを *attr* に設定します。初期値は 0 (属性なし) です。

window.**bkgd**(*ch*[, *attr*])

ウィンドウ上の背景プロパティを、*attr* を属性とする文字 *ch* に設定します。変更はそのウィンドウ中のすべての文字に以下のようにして適用されます:

- ウィンドウ中のすべての文字の属性が新たな背景属性に変更されます。
- 以前の背景文字が出現すると、常に新たな背景文字に変更されます。

window.**bkgdset**(*ch*[, *attr*])

ウィンドウの背景を設定します。ウィンドウの背景は、文字と何らかの属性の組み合わせから成り立ちます。背景情報の属性の部分は、ウィンドウ上に描画されている空白でないすべての文字と組み合わされ (OR され) ます。空白文字には文字部分と属性部分の両方が組み合わされます。背景は文字のプロパティとなり、スクロールや行/文字の挿入/削除操作の際には文字と一緒に移動します。

window.**border**([*ls*[, *rs*[, *ts*[, *bs*[, *tl*[, *tr*[, *bl*[, *br*]]]]]]]])

ウィンドウの縁に境界線を描画します。各引数には境界の特定部分を表現するために使われる文字を指定します; 詳細は以下のテーブルを参照してください。

注釈: どの引数も、0 を指定した場合デフォルトの文字が使われるようになります。キーワード引数は使うことが **できません**。デフォルトはテーブル内で示しています:

引数	説明	デフォルト値
ls	左側	ACS_VLINE
rs	右側	ACS_VLINE
ts	上側	ACS_HLINE
bs	下側	ACS_HLINE
tl	左上の角	ACS_ULCORNER
tr	右上の角	ACS_URCORNER
bl	左下の角	ACS_LLCORNER
br	右下の角	ACS_LRCORNER

window.box(*[vertch, horch]*)

border() と同様ですが、*ls* および *rs* は共に *vertch* で、*ts* および *bs* は共に *horch* です。この関数では、角に使われるデフォルト文字が常に使用されます。

window.chgat(*attr*)
window.chgat(*num*, *attr*)
window.chgat(*y*, *x*, *attr*)
window.chgat(*y*, *x*, *num*, *attr*)

Set the attributes of *num* characters at the current cursor position, or at position `(y, x)` if supplied. If *num* is not given or is `-1`, the attribute will be set on all the characters to the end of the line. This function moves cursor to position `(y, x)` if supplied. The changed line will be touched using the *touchline()* method so that the contents will be redisplayed by the next window refresh.

window.clear()

erase() に似ていますが、次に *refresh()* が呼び出された際にすべてのウィンドウを再描画するようにします。

window.clearok(*flag*)

flag が `True` ならば、次の *refresh()* はウィンドウを完全に消去します。

window.clrtobot()

カーソルの位置からウィンドウの端までを消去します: カーソル以降のすべての行が削除されるため、*clrtoeol()* と等価です。

window.clrtoeol()

カーソル位置から行末までを消去します。

window.cursyncup()

ウィンドウのすべての親ウィンドウについて、現在のカーソル位置を反映するよう更新します。

window.delch(*[y, x]*)

`(y, x)` にある文字を削除します。

window.deleteln()

カーソルの下にある行を削除します。後続の行はすべて 1 行上に移動します。

window.**derwin**(*begin_y*, *begin_x*)
window.**derwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)
”derive window (ウィンドウを派生する)” の短縮形です。*derwin()* は *subwin()* と同じですが、*begin_y* および *begin_x* はスクリーン全体の原点ではなく、ウィンドウの原点からの相対位置です。派生したウィンドウオブジェクトが返されます。

window.**echochar**(*ch*[, *attr*])
文字 *ch* に属性 *attr* を付与し、即座に *refresh()* をウィンドウに対して呼び出します。

window.**enclose**(*y*, *x*)
与えられた文字セル座標をスクリーン原点から相対的なものとし、ウィンドウの中に含まれるかを調べて、True または False を返します。スクリーン上のウィンドウの一部がマウスイベントの発生場所を含むかどうかを調べる上で便利です。

window.**encoding**
encode メソッドの引数 (Unicode 文字列および文字) で使用されるエンコーディングです。例えば *window.subwin()* などでサブウィンドウを生成した時、エンコーディング属性は親ウィンドウから継承します。デフォルトでは、そのロケールのエンコーディングが使用されます (*locale.getpreferredencoding()* 参照)。

バージョン 3.3 で追加.

window.**erase**()
ウィンドウをクリアします。

window.**getbegyx**()
左上の角の座標をあらわすタプル (y, x) を返します。

window.**getbkgd**()
与えられたウィンドウの現在の背景文字と属性のペアを返します。

window.**getch**([*y*, *x*])
文字を 1 個取得します。返される整数は ASCII の範囲の値となる **とは限らない** ので注意してください。ファンクションキー、キーパッドのキー等は 256 よりも大きな数字で表されます。無遅延 (no-delay) モードでは、入力がない場合 -1 を返し、そうでなければキーが押されるまで待ちます。

window.**get_wch**([*y*, *x*])
ワイド文字を 1 個取得します。ほとんどのキー、ファンクションキーの数値、キーパッドのキー、およびその他の特殊キーの文字を返します。無遅延モードでは、入力がない場合例外を送出します。

バージョン 3.3 で追加.

window.**getkey**([*y*, *x*])
文字を 1 個取得し、*getch()* が返すような整数ではなく文字列を返します。ファンクションキー、キーパッドのキー、およびその他特殊キーはキー名を含む複数文字を返します。無遅延モードでは、入力がない場合例外を送出します。

window.**getmaxyx**()
ウィンドウの高さおよび幅を表すタプル (y, x) を返します。

window.**getparyx**()
: 親ウィンドウ中におけるウィンドウの開始位置をタプル (y, x) で返します。ウィンドウに親ウィンドウがない場合 -1, -1 を返します。

window.**getstr**()
window.**getstr**(*n*)
window.**getstr**(*y*, *x*)
window.**getstr**(*y*, *x*, *n*)
: 原始的な文字編集機能つきで、ユーザの入力した byte オブヘクトを読み取ります。

window.**getyx**()
: ウィンドウの左上角からの相対で表した現在のカーソル位置をタプル (y, x) で返します。

window.**hline**(*ch*, *n*)
window.**hline**(*y*, *x*, *ch*, *n*)
: (y, x) から始まり、*n* の長さを持つ、文字 *ch* で作られる水平線を表示します。

window.**idcok**(*flag*)
: *flag* が False の場合、curses は端末のハードウェアによる文字挿入/削除機能を使おうとしなくなります; *flag* が True ならば、文字挿入/削除は有効にされます。curses が最初に初期化された際には文字挿入/削除はデフォルトで有効になっています。

window.**idlok**(*flag*)
: *flag* が True であれば、*curses* はハードウェアの行編集機能の利用を試みます。行挿入/削除は無効化されます。

window.**immedok**(*flag*)
: *flag* が True ならば、ウィンドウイメージ内における何らかの変更があるとウィンドウを更新するようになります; すなわち、*refresh()* を自分で呼ばなくても良くなります。とはいえ、wrefresh を繰り返し呼び出すことになるため、この操作はかなりパフォーマンスを低下させます。デフォルトでは無効になっています。

window.**inch**([*y*, *x*])
: ウィンドウの指定の位置の文字を返します。下位 8 ビットが本来の文字で、それより上のビットは属性です。

window.**insch**(*ch*[, *attr*])
window.**insch**(*y*, *x*, *ch*[, *attr*])
: (y, x) に文字 *ch* を属性 *attr* で描画し、行の *x* からの内容を 1 文字分右にずらします。

window.**insdelln**(*nlines*)
: *nlines* 行を指定されたウィンドウの現在の行の上に挿入します。その下にある *nlines* 行は失われます。負の *nlines* を指定すると、カーソルのある行以降の *nlines* を削除し、削除された行の後ろに続く内容が上に来ます。その下にある *nlines* は消去されます。現在のカーソル位置はそのままです。

window.**insertln**()
: カーソルの下に空行を 1 行入れます。それ以降の行は 1 行づつ下に移動します。

window.**insnstr**(*str*, *n*[, *attr*])

window.**insnstr**(*y*, *x*, *str*, *n*[, *attr*])

文字列をカーソルの下にある文字の前に (一行に収まるだけ) 最大 *n* 文字挿入します。*n* がゼロまたは負の値の場合、文字列全体が挿入されます。カーソルの右にあるすべての文字は右に移動し、行の左端にある文字は失われます。カーソル位置は (*y*, *x* が指定されていた場合はそこに移動しますが、その後は) 変化しません。

window.**insstr**(*str*[, *attr*])
window.**insstr**(*y*, *x*, *str*[, *attr*])

キャラクタ文字列を (行に収まるだけ) カーソルより前に挿入します。カーソルの右側にある文字はすべて右にシフトし、行の右端の文字は失われます。カーソル位置は (*y*, *x* が指定されていた場合はそこに移動しますが、その後は) 変化しません。

window.**instr**([*n*])
window.**instr**(*y*, *x*[, *n*])

現在のカーソル位置、または *y*, *x* が指定されている場合にはその場所から始まるキャラクタの bytes オブジェクトをウィンドウから抽出して返します。属性は文字から剥ぎ取られます。*n* が指定された場合、*instr()* は (末尾の NUL 文字を除いて) 最大で *n* 文字までの長さからなる文字列を返します。

window.**is_linetouched**(*line*)

指定した行が、最後に *refresh()* を呼んだ時から変更されている場合に True を返します; そうでない場合には False を返します。*line* が現在のウィンドウ上の有効な行でない場合、*curses.error* 例外を送出します。

window.**is_wintouched**()

指定したウィンドウが、最後に *refresh()* を呼んだ時から変更されている場合に True を返します; そうでない場合には False を返します。

window.**keypad**(*flag*)

flag が True の場合、ある種のキー (キーパッドやファンクションキー) によって生成されたエスケープシーケンスは *curses* で解釈されます。*flag* が False の場合、エスケープシーケンスは入力ストリームにそのままの状態で残されます。

window.**leaveok**(*flag*)

flag が True の場合、カーソルは ” カーソル位置” に移動せず現在の場所にとどめます。これにより、カーソルの移動を減らせる可能性があります。この場合、カーソルは不可視にされます。

flag が False の場合、カーソルは更新の際に常に ” カーソル位置” に移動します。

window.**move**(*new_y*, *new_x*)

カーソルを (new_y, new_x) に移動します。

window.**mvderwin**(*y*, *x*)

ウィンドウを親ウィンドウの中で移動します。ウィンドウのスクリーン相対となるパラメタ群は変化しません。このルーチンは親ウィンドウの一部をスクリーン上の同じ物理位置に表示する際に用いられます。

window.**mvwin**(*new_y*, *new_x*)

ウィンドウの左上角が (new_y, new_x) になるように移動します。

window.**nodelay**(*flag*)
: *flag* が True の場合、*getch()* は非ブロックで動作します。

window.**notimeout**(*flag*)
: *flag* が True の場合、エスケープシーケンスはタイムアウトしなくなります。

  flag が False の場合、数ミリ秒間の間エスケープシーケンスは解釈されず、入力ストリーム中にそのままの状態で残されます。

window.**noutrefresh**()
: 更新をマークはしますが待機します。この関数はウィンドウのデータ構造を表現したい内容を反映するように更新しますが、物理スクリーン上に反映させるための強制更新を行いません。更新を行うためには *doupdate()* を呼び出します。

window.**overlay**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])
: ウィンドウを *destwin* の上に重ね書き (overlay) します。ウィンドウは同じサイズである必要はなく、重なっている領域だけが複写されます。この複写は非破壊的です。これは現在の背景文字が *destwin* の内容を上書きしないことを意味します。

  複写領域をきめ細かく制御するために、*overlay()* の第二形式を使うことができます。*sminrow* および *smincol* は元のウィンドウの左上の座標で、他の変数は *destwin* 内の矩形を表します。

window.**overwrite**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])
: *destwin* の上にウィンドウの内容を上書き (overwrite) します。ウィンドウは同じサイズである必要はなく、重なっている領域だけが複写されます。この複写は破壊的です。これは現在の背景文字が *destwin* の内容を上書きすることを意味します。

  複写領域をきめ細かく制御するために、*overwrite()* の第二形式を使うことができます。*sminrow* および *smincol* は元のウィンドウの左上の座標で、他の変数は *destwin* 内の矩形を表します。

window.**putwin**(*file*)
: ウィンドウに関連付けられているすべてのデータを与えられたファイルオブジェクトに書き込みます。この情報は後に *getwin()* 関数を使って取得することができます。

window.**redrawln**(*beg*, *num*)
: *beg* 行から始まる *num* スクリーン行の表示内容が壊れており、次の *refresh()* 呼び出しで完全に再描画されなければならないことを通知します。

window.**redrawwin**()
: ウィンドウ全体を更新 (touch) し、次の *refresh()* 呼び出しで完全に再描画されるようにします。

window.**refresh**([*pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*])
: ディスプレイを即時更新し (実際のウィンドウとこれまでの描画/削除メソッドの内容とを同期し) ます。

  6 つのオプション引数はウィンドウが *newpad()* で生成された場合にのみ指定することができます。追加の引数はパッドやスクリーンのどの部分が含まれるのかを示すために必要です。*pminrow* および *pmincol* にはパッドが表示されている矩形の左上角を指定します。*sminrow*, *smincol*, *smaxrow*, および *smaxcol* には、スクリーン上に表示される矩形の縁を指定します。パッド内に表示される矩形の右下

角はスクリーン座標から計算されるので、矩形は同じサイズでなければなりません。矩形は両方とも、それぞれのウィンドウ構造内に完全に含まれていなければなりません。*pminrow*, *pmincol*, *sminrow*, または *smincol* に負の値を指定すると、ゼロを指定したものとして扱われます。

window.resize(*nlines*, *ncols*)
curses ウィンドウの記憶域を、指定値のサイズに調整するため再割当てします。サイズが現在の値より大きい場合、ウィンドウのデータは現在の背景設定 (*bkgdset()* で設定) で埋められマージされます。

window.scroll(*[lines=1]*)
スクリーンまたはスクロール領域を上に *lines* 行スクロールします。

window.scrollok(*flag*)
ウィンドウのカーソルが、最下行で改行を行ったり最後の文字を入力したりした結果、ウィンドウやスクロール領域の縁からはみ出して移動した際の動作を制御します。*flag* が `False` の場合、カーソルは最下行にそのままにしておかれます。*flag* が `True` の場合、ウィンドウは 1 行上にスクロールします。端末の物理スクロール効果を得るためには *idlok()* も呼び出す必要があるので注意してください。

window.setscrreg(*top*, *bottom*)
スクロール領域を *top* から *bottom* に設定します。スクロール動作はすべてこの領域で行われます。

window.standend()
A_STANDOUT 属性をオフにします。端末によっては、この操作ですべての属性をオフにする副作用が発生します。

window.standout()
A_STANDOUT 属性をオンにします。

window.subpad(*begin_y*, *begin_x*)
window.subpad(*nlines*, *ncols*, *begin_y*, *begin_x*)
左上の角が `(begin_y, begin_x)` にあり、幅/高さがそれぞれ *ncols* / *nlines* であるようなサブウィンドウを返します。

window.subwin(*begin_y*, *begin_x*)
window.subwin(*nlines*, *ncols*, *begin_y*, *begin_x*)
左上の角が `(begin_y, begin_x)` にあり、幅/高さがそれぞれ *ncols* / *nlines* であるようなサブウィンドウを返します。

デフォルトでは、サブウィンドウは指定された場所からウィンドウの右下角まで広がります。

window.syncdown()
このウィンドウの上位のウィンドウのいずれかで更新 (touch) された各場所をこのウィンドウ内でも更新します。このルーチンは *refresh()* から呼び出されるので、手動で呼び出す必要はほとんどないはずです。

window.syncok(*flag*)
flag が `True` の場合、ウィンドウが変更された際は常に *syncup()* を自動的に呼ぶようになります。

window.syncup()
ウィンドウ内で更新 (touch) した場所を、上位のすべてのウィンドウ内でも更新します。

window.**timeout**(*delay*)
ウィンドウのブロックまたは非ブロック読み込み動作を設定します。*delay* が負の場合、ブロック読み出しが使われ、入力を無期限で待ち受けます。*delay* がゼロの場合、非ブロック読み出しが使われ、入力待ちの文字がない場合 *getch()* は -1 を返します。*delay* が正の値であれば、*getch()* は *delay* ミリ秒間ブロックし、ブロック後の時点で入力がない場合には -1 を返します。

window.**touchline**(*start*, *count*[, *changed*])
start から始まる *count* 行が変更されたかのように振舞わせます。もし *changed* が与えられた場合、その引数は指定された行が変更された (*changed*=True) か、変更されていないか (*changed*) を指定します。

window.**touchwin**()
描画を最適化するために、すべてのウィンドウが変更されたかのように振舞わせます。

window.**untouchwin**()
ウィンドウ内のすべての行を、最後に *refresh()* を呼んだ際から変更されていないものとしてマークします。

window.**vline**(*ch*, *n*)
window.**vline**(*y*, *x*, *ch*, *n*)
(y, x) から始まり、*n* の長さを持つ、文字 *ch* で作られる垂直線を表示します。

16.10.3 定数

curses モジュールでは以下のデータメンバを定義しています:

curses.**ERR**
getch() のような整数を返す curses ルーチンのいくつかは、失敗した際に *ERR* を返します。

curses.**OK**
napms() のような整数を返す curses ルーチンのいくつかは、成功した際に *OK* を返します。

curses.**version**
モジュールの現在のバージョンを表現する bytes オブジェクトです。__version__ でも取得できます。

curses.**ncurses_version**
A named tuple containing the three components of the ncurses library version: *major*, *minor*, and *patch*. All values are integers. The components can also be accessed by name, so curses.ncurses_version[0] is equivalent to curses.ncurses_version.major and so on.

Availability: if the ncurses library is used.

バージョン 3.8 で追加.

Some constants are available to specify character cell attributes. The exact constants available are system dependent.

属性	意味
A_ALTCHARSET	代替文字セットモード
A_BLINK	点滅モード
A_BOLD	太字モード
A_DIM	低輝度モード
A_INVIS	Invisible or blank mode
A_ITALIC	イタリックモード
A_NORMAL	通常の属性
A_PROTECT	Protected mode
A_REVERSE	Reverse background and foreground colors
A_STANDOUT	強調モード
A_UNDERLINE	下線モード
A_HORIZONTAL	Horizontal highlight
A_LEFT	Left highlight
A_LOW	Low highlight
A_RIGHT	Right highlight
A_TOP	Top highlight
A_VERTICAL	Vertical highlight
A_CHARTEXT	Bit-mask to extract a character

バージョン 3.7 で追加: A_ITALIC が追加されました。

Several constants are available to extract corresponding attributes returned by some methods.

Bit-mask	意味
A_ATTRIBUTES	Bit-mask to extract attributes
A_CHARTEXT	Bit-mask to extract a character
A_COLOR	Bit-mask to extract color-pair field information

キーは KEY_ で始まる名前をもつ整数定数です。利用可能なキーキャップはシステムに依存します。

キー定数	キー
KEY_MIN	最小のキー値
KEY_BREAK	ブレークキー (Break, 信頼できません)
KEY_DOWN	下矢印
KEY_UP	上矢印
KEY_LEFT	左矢印
KEY_RIGHT	右矢印
KEY_HOME	ホームキー (Home, または上左矢印)
KEY_BACKSPACE	バックスペース (Backspace, 信頼できません)
KEY_F0	ファンクションキー。64 個までサポートされています。

次のページに続く

表 1 – 前のページからの続き

キー定数	キー
KEY_Fn	ファンクションキー n の値
KEY_DL	行削除 (Delete line)
KEY_IL	行挿入 (Insert line)
KEY_DC	文字削除 (Delete char)
KEY_IC	文字挿入、または文字挿入モードへ入る
KEY_EIC	文字挿入モードから抜ける
KEY_CLEAR	画面消去
KEY_EOS	画面の末端まで消去
KEY_EOL	行末端まで消去
KEY_SF	前に 1 行スクロール
KEY_SR	後ろ (逆方向) に 1 行スクロール
KEY_NPAGE	次のページ (Page Next)
KEY_PPAGE	前のページ (Page Prev)
KEY_STAB	タブ設定
KEY_CTAB	タブリセット
KEY_CATAB	すべてのタブをリセット
KEY_ENTER	入力または送信 (信頼できません)
KEY_SRESET	ソフトウェア (部分的) リセット (信頼できません)
KEY_RESET	リセットまたはハードリセット (信頼できません)
KEY_PRINT	印刷 (Print)
KEY_LL	下ホーム (Home down) または最下行 (左下)
KEY_A1	キーパッドの左上キー
KEY_A3	キーパッドの右上キー
KEY_B2	キーパッドの中央キー
KEY_C1	キーパッドの左下キー
KEY_C3	キーパッドの右下キー
KEY_BTAB	Back tab
KEY_BEG	開始 (Beg)
KEY_CANCEL	キャンセル (Cancel)
KEY_CLOSE	Close [閉じる]
KEY_COMMAND	コマンド (Cmd)
KEY_COPY	Copy [コピー]
KEY_CREATE	生成 (Create)
KEY_END	終了 (End)
KEY_EXIT	Exit [終了]
KEY_FIND	検索 (Find)
KEY_HELP	ヘルプ (Help)
KEY_MARK	マーク (Mark)
KEY_MESSAGE	メッセージ (Message)
KEY_MOVE	移動 (Move)

次のページに続く

表 1 – 前のページからの続き

キー定数	キー
KEY_NEXT	次へ (Next)
KEY_OPEN	開く (Open)
KEY_OPTIONS	オプション
KEY_PREVIOUS	前へ (Prev)
KEY_REDO	Redo [やり直し]
KEY_REFERENCE	参照 (Ref)
KEY_REFRESH	更新 (Refresh)
KEY_REPLACE	置換 (Replace)
KEY_RESTART	再起動 (Restart)
KEY_RESUME	再開 (Resume)
KEY_SAVE	Save [保存]
KEY_SBEG	シフト付き Beg
KEY_SCANCEL	シフト付き Cancel
KEY_SCOMMAND	シフト付き Command
KEY_SCOPY	シフト付き Copy
KEY_SCREATE	シフト付き Create
KEY_SDC	シフト付き Delete char
KEY_SDL	シフト付き Delete line
KEY_SELECT	選択 (Select)
KEY_SEND	シフト付き End
KEY_SEOL	シフト付き Clear line
KEY_SEXIT	シフト付き Exit
KEY_SFIND	シフト付き Find
KEY_SHELP	シフト付き Help
KEY_SHOME	シフト付き Home
KEY_SIC	シフト付き Input
KEY_SLEFT	シフト付き Left arrow
KEY_SMESSAGE	シフト付き Message
KEY_SMOVE	シフト付き Move
KEY_SNEXT	シフト付き Next
KEY_SOPTIONS	シフト付き Options
KEY_SPREVIOUS	シフト付き Prev
KEY_SPRINT	シフト付き Print
KEY_SREDO	シフト付き Redo
KEY_SREPLACE	シフト付き Replace
KEY_SRIGHT	シフト付き Right arrow
KEY_SRSUME	シフト付き Resume
KEY_SSAVE	シフト付き Save
KEY_SSUSPEND	シフト付き Suspend
KEY_SUNDO	シフト付き Undo

次のページに続く

表 1 – 前のページからの続き

キー定数	キー
KEY_SUSPEND	一時停止 (Suspend)
KEY_UNDO	Undo [元に戻す]
KEY_MOUSE	マウスイベント通知
KEY_RESIZE	端末リサイズイベント
KEY_MAX	最大キー値

VT100 や、X 端末エミュレータのようなソフトウェアエミュレーションでは、通常少なくとも 4 つのファンクションキー (KEY_F1, KEY_F2, KEY_F3, KEY_F4) が利用可能で、矢印キーは KEY_UP, KEY_DOWN, KEY_LEFT および KEY_RIGHT が対応付けられています。計算機に PC キーボードが付属している場合、矢印キーと 12 個のファンクションキー (古い PC キーボードには 10 個しかファンクションキーがないかもしれません) が利用できると考えてよいでしょう; また、以下のキーパッド対応付けは標準的なものです:

キーキャップ	定数
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END
Page Up	KEY_PPAGE
Page Down	KEY_NPAGE

代替文字セットを以下の表に列挙します。これらは VT100 端末から継承したものであり、X 端末のようなソフトウェアエミュレーション上で一般に利用可能なものです。グラフィックが利用できない場合、curses は印字可能 ASCII 文字による粗雑な近似出力を行います。

注釈: これらは *initscr()* が呼び出された後でしか利用できません。

ACS コード	意味
ACS_BBSS	右上角の別名
ACS_BLOCK	黒四角ブロック
ACS_BOARD	白四角ブロック
ACS_BSBS	水平線の別名
ACS_BSSB	左上角の別名
ACS_BSSS	上向き T 字罫線の別名
ACS_BTEE	下向き T 字罫線
ACS_BULLET	黒丸 (bullet)
ACS_CKBOARD	チェッカーボードパタン (点描)
ACS_DARROW	下向き矢印
ACS_DEGREE	度記号

次のページに続く

表 2 – 前のページからの続き

ACS コード	意味
ACS_DIAMOND	ダイアモンド
ACS_GEQUAL	大なりイコール
ACS_HLINE	水平線
ACS_LANTERN	ランタン (lantern) シンボル
ACS_LARROW	左向き矢印
ACS_LEQUAL	小なりイコール
ACS_LLCORNER	左下角
ACS_LRCORNER	右下角
ACS_LTEE	左向き T 字罫線
ACS_NEQUAL	不等号
ACS_PI	パイ記号
ACS_PLMINUS	プラスマイナス記号
ACS_PLUS	大プラス記号
ACS_RARROW	右向き矢印
ACS_RTEE	右向き T 字罫線
ACS_S1	スキャンライン 1
ACS_S3	スキャンライン 3
ACS_S7	スキャンライン 7
ACS_S9	スキャンライン 9
ACS_SBBS	右下角の別名
ACS_SBSB	垂直線の別名
ACS_SBSS	右向き T 字罫線の別名
ACS_SSBB	左下角の別名
ACS_SSBS	下向き T 字罫線の別名
ACS_SSSB	左向き T 字罫線の別名
ACS_SSSS	交差罫線または大プラス記号の別名
ACS_STERLING	ポンドスターリング記号
ACS_TTEE	上向き T 字罫線
ACS_UARROW	上向き矢印
ACS_ULCORNER	左上角
ACS_URCORNER	右上角
ACS_VLINE	垂直線

以下のテーブルは定義済みの色を列挙したものです:

定数	色
COLOR_BLACK	黒
COLOR_BLUE	青
COLOR_CYAN	シアン (薄く緑がかった青)
COLOR_GREEN	緑
COLOR_MAGENTA	マゼンタ (紫がかった赤)
COLOR_RED	赤
COLOR_WHITE	白
COLOR_YELLOW	黄色

16.11 curses.textpad --- curses プログラムのためのテキスト入力ウィジェット

curses.textpad モジュールでは、curses ウィンドウ内での基本的なテキスト編集を処理し、Emacs に似た (すなわち Netscape Navigator, BBedit 6.x, FrameMaker, その他諸々のプログラムとも似た) キーバインドをサポートしている *Textbox* クラスを提供します。このモジュールではまた、テキストボックスを枠で囲むなどの目的のために有用な、矩形描画関数を提供しています。

curses.textpad モジュールでは以下の関数を定義しています:

curses.textpad.**rectangle**(*win*, *uly*, *ulx*, *lry*, *lrx*)

矩形を描画します。最初の引数はウィンドウオブジェクトでなければなりません; 残りの引数はそのウィンドウからの相対座標になります。2 番目および 3 番目の引数は描画すべき矩形の左上角の y および x 座標です; 4 番目および 5 番目の引数は右下角の y および x 座標です。矩形は、VT100/IBM PC におけるフォーム文字を利用できる端末 (xterm やその他のほとんどのソフトウェア端末エミュレータを含む) ではそれを使って描画されます。そうでなければ ASCII 文字のダッシュ、垂直バー、およびプラス記号で描画されます。

16.11.1 Textbox オブジェクト

以下のような *Textbox* オブジェクトをインスタンス生成することができます:

class curses.textpad.**Textbox**(*win*)

テキストボックスウィジェットオブジェクトを返します。*win* 引数は、テキストボックスを入れるための **ウィンドウ** オブジェクトでなければなりません。テキストボックスの編集カーソルは、最初はテキストボックスが入っているウィンドウの左上角に配置され、その座標は `(0, 0)` です。インスタンスの *stripspaces* フラグの初期値はオンに設定されます。

Textbox オブジェクトは以下のメソッドを持ちます:

edit(*[validator]*)

普段使うことになるエントリポイントです。終了キーストロークの一つが入力されるまで編集キーストロークを受け付けます。*validator* を与える場合、関数でなければなりません。*validator* は

キーストロークが入力されるたびにそのキーストロークが引数となって呼び出されます; 返された値に対して、コマンドキーストロークとして解釈が行われます。このメソッドはウィンドウの内容を文字列として返します; ウィンドウ内の空白が含められるかどうかは *stripspaces* 属性で決められます。

do_command(*ch*)

単一のコマンドキーストロークを処理します。以下にサポートされている特殊キーストロークを示します:

キーストローク	動作
Control-A	ウィンドウの左端に移動します。
Control-B	カーソルを左へ移動し、必要なら前の行に折り返します。
Control-D	カーソル下の文字を削除します。
Control-E	右端 (stripspaces がオフのとき) または行末 (stripspaces がオンのとき) に移動します。
Control-F	カーソルを右に移動し、必要なら次の行に折り返します。
Control-G	ウィンドウを終了し、その内容を返します。
Control-H	逆方向に文字を削除します。
Control-J	ウィンドウが 1 行であれば終了し、そうでなければ新しい行を挿入します。
Control-K	行が空白行ならその行全体を削除し、そうでなければカーソル以降行末までを消去します。
Control-L	スクリーンを更新します。
Control-N	カーソルを下に移動します; 1 行下に移動します。
Control-O	カーソルの場所に空行を 1 行挿入します。
Control-P	カーソルを上に移動します; 1 行上に移動します。

移動操作は、カーソルがウィンドウの縁にあって移動ができない場合には何も行いません。場合によっては、以下のような同義のキーストロークがサポートされています:

定数	キーストローク
KEY_LEFT	Control-B
KEY_RIGHT	Control-F
KEY_UP	Control-P
KEY_DOWN	Control-N
KEY_BACKSPACE	Control-h

他のキーストロークは、与えられた文字を挿入し、(行折り返し付きで) 右に移動するコマンドとして扱われます。

gather()

ウィンドウの内容を文字列として返します; ウィンドウ内の空白が含められるかどうかは *stripspaces* メンバ変数で決められます。

stripspaces

この属性はウィンドウ内の空白領域の解釈方法を制御するためのフラグです。フラグがオンに設定されている場合、各行の末端にある空白領域は無視されます; すなわち、末端空白領域にカーソルが入ると、その場所の代わりに行の末尾にカーソルが移動します。また、末端の空白領域はウィンドウの内容を取得する際に剥ぎ取られます。

16.12 curses.ascii --- ASCII 文字に関するユーティリティ

curses.ascii モジュールでは、ASCII 文字を指す名前定数と、様々な ASCII 文字区分についてある文字が帰属するかどうかを調べる関数を提供します。このモジュールで提供されている定数は以下の制御文字の名前です:

名前	意味
NUL	
SOH	ヘディング開始、コンソール割り込み
STX	テキスト開始
ETX	テキスト終了
EOT	テキスト伝送終了
ENQ	問い合わせ、ACK フロー制御時に使用
ACK	肯定応答
BEL	ベル
BS	一文字後退
TAB	タブ
HT	TAB の別名: ” 水平タブ”
LF	改行
NL	LF の別名: ” 改行”
VT	垂直タブ
FF	改頁
CR	復帰
SO	シフトアウト、他の文字セットの開始
SI	シフトイン、標準の文字セットに復帰
DLE	データリンクでのエスケープ
DC1	装置制御 1、フロー制御のための XON
DC2	装置制御 2、ブロックモードフロー制御
DC3	装置制御 3、フロー制御のための XOFF
DC4	装置制御 4
NAK	否定応答
SYN	同期信号
ETB	ブロック転送終了
CAN	キャンセル (Cancel)

次のページに続く

表 3 – 前のページからの続き

名前	意味
EM	媒体終端
SUB	代入文字
ESC	エスケープ文字
FS	ファイル区切り文字
GS	グループ区切り文字
RS	レコード区切り文字、ブロックモード終了子
US	単位区切り文字
SP	空白文字
DEL	削除

これらの大部分は、最近は実際に定数の意味通りに使われることがほとんどないので注意してください。これらのニーモニック符号はデジタル計算機より前のテレプリンタにおける慣習から付けられたものです。

このモジュールでは、標準 C ライブラリの関数を雛型とする以下の関数をサポートしています:

curses.ascii.**isalnum**(*c*)
ASCII 英数文字かどうかを調べます; `isalpha(c) or isdigit(c)` と等価です。

curses.ascii.**isalpha**(*c*)
ASCII アルファベット文字かどうかを調べます; `isupper(c) or islower(c)` と等価です。

curses.ascii.**isascii**(*c*)
文字が 7 ビット ASCII 文字に合致するかどうかを調べます。

curses.ascii.**isblank**(*c*)
ASCII 余白文字、すなわち空白または水平タブかどうかを調べます。

curses.ascii.**iscntrl**(*c*)
ASCII 制御文字 (0x00 から 0x1f の範囲または 0x7f) かどうかを調べます。

curses.ascii.**isdigit**(*c*)
ASCII 10 進数字、すなわち `'0'` から `'9'` までの文字かどうかを調べます。`c in string.digits` と等価です。

curses.ascii.**isgraph**(*c*)
空白以外の ASCII 印字可能文字かどうかを調べます。

curses.ascii.**islower**(*c*)
ASCII 小文字かどうかを調べます。

curses.ascii.**isprint**(*c*)
空白文字を含め、ASCII 印字可能文字かどうかを調べます。

curses.ascii.**ispunct**(*c*)
空白または英数字以外の ASCII 印字可能文字かどうかを調べます。

curses.ascii.**isspace**(*c*)
ASCII 余白文字、すなわち空白、改行、復帰、改頁、水平タブ、垂直タブかどうかを調べます。

curses.ascii.**isupper**(*c*)
ASCII 大文字かどうかを調べます。

curses.ascii.**isxdigit**(*c*)
ASCII 16 進数字かどうかを調べます。`c in string.hexdigits` と等価です。

curses.ascii.**isctrl**(*c*)
ASCII 制御文字 (0 から 31 までの値) かどうかを調べます。

curses.ascii.**ismeta**(*c*)
非 ASCII 文字 (0x80 またはそれ以上の値) かどうかを調べます。

これらの関数は数字も 1 文字の文字列も使えます; 引数を文字列にした場合、組み込み関数 *ord()* を使って変換されます。

これらの関数は全て、関数に渡した文字列の文字から得られたビット値を調べるので注意してください; 関数はホスト計算機で使われている文字列エンコーディングについて何ら関知しません。

以下の 2 つの関数は、引数として 1 文字の文字列または整数で表したバイト値のどちらでもとり得ます; これらの関数は引数と同じ型で値を返します。

curses.ascii.**ascii**(*c*)
ASCII 値を返します。*c* の下位 7 ビットに対応します。

curses.ascii.**ctrl**(*c*)
与えた文字に対応する制御文字を返します (0x1f とビット単位で論理積を取ります)。

curses.ascii.**alt**(*c*)
与えた文字に対応する 8 ビット文字を返します (0x80 とビット単位で論理和を取ります)。

以下の関数は 1 文字からなる文字列値または整数値を引数に取り、文字列を返します。

curses.ascii.**unctrl**(*c*)
ASCII 文字 *c* の文字列表現を返します。もし *c* が印字可能文字であれば、返される文字列は *c* そのものになります。もし *c* が制御文字 (0x00--0x1f) であれば、キャレット (`'^'`) と、その後ろに続く *c* に対応した大文字からなる文字列になります。*c* が ASCII 削除文字 (0x7f) であれば、文字列は `'^?'` になります。*c* のメタビット (0x80) がセットされていれば、メタビットは取り去られ、前述のルールが適用され、`'!'` が前につけられます。

curses.ascii.**controlnames**
0 (NUL) から 0x1f (US) までの 32 の ASCII 制御文字と、空白文字 `SP` のニーモニック符号名からなる 33 要素の文字列によるシーケンスです。

16.13 curses.panel --- curses のためのパネルスタック拡張

パネルは深さ (depth) の機能が追加されたウィンドウです。これにより、ウィンドウをお互いに重ね合わせることができ、各ウィンドウの可視部分だけが表示されます。パネルはスタック中に追加したり、スタック内で上下移動させたり、スタックから除去することができます。

16.13.1 関数

curses.panel では以下の関数を定義しています:

curses.panel.**bottom_panel**()
: パネルスタックの最下層のパネルを返します。

curses.panel.**new_panel**(*win*)
: 与えられたウィンドウ *win* に関連付けられたパネルオブジェクトを返します。返されたパネルオブジェクトを参照しておく必要があることに注意してください。もし参照しなければ、パネルオブジェクトはガベージコレクションされてパネルスタックから削除されます。

curses.panel.**top_panel**()
: パネルスタックの最上層のパネルを返します。

curses.panel.**update_panels**()
: 仮想スクリーンをパネルスタック変更後の状態に更新します。この関数では *curses.doupdate()* を呼ばないので、ユーザは自分で呼び出す必要があります。

16.13.2 Panel オブジェクト

上記の *new_panel()* が返す Panel オブジェクトはスタック順の概念を持つウィンドウです。ウィンドウはパネルに関連付けられており、表示する内容を決定している一方、パネルのメソッドはパネルスタック中のウィンドウの深さ管理を担います。

Panel オブジェクトは以下のメソッドを持っています:

Panel.**above**()
: 現在のパネルの上にあるパネルを返します。

Panel.**below**()
: 現在のパネルの下にあるパネルを返します。

Panel.**bottom**()
: パネルをスタックの最下層にプッシュします。

Panel.**hidden**()
: パネルが隠れている (不可視である) 場合に `True` を返し、そうでない場合 `False` を返します。

Panel.**hide**()
パネルを隠します。この操作ではオブジェクトは消去されず、スクリーン上のウィンドウを不可視にするだけです。

Panel.**move**(*y*, *x*)
パネルをスクリーン座標 (`y`, `x`) に移動します。

Panel.**replace**(*win*)
パネルに関連付けられたウィンドウを *win* に変更します。

Panel.**set_userptr**(*obj*)
パネルのユーザポインタを *obj* に設定します。このメソッドは任意のデータをパネルに関連付けるために使われ、任意の Python オブジェクトにすることができます。

Panel.**show**()
(隠れているはずの) パネルを表示します。

Panel.**top**()
パネルをスタックの最上層にプッシュします。

Panel.**userptr**()
パネルのユーザポインタを返します。任意の Python オブジェクトです。

Panel.**window**()
パネルに関連付けられているウィンドウオブジェクトを返します。

16.14 platform --- 実行中プラットフォームの固有情報を参照する

ソースコード: Lib/platform.py

注釈: プラットフォーム毎にアルファベット順に並べています。Linux については Unix セクションを参照してください。

16.14.1 クロスプラットフォーム

platform.**architecture**(*executable=sys.executable*, *bits=''*, *linkage=''*)
executable で指定した実行可能ファイル（省略時は Python インタープリタのバイナリ）の各種アーキテクチャ情報を調べます。

戻り値はタプル (`bits`, `linkage`) で、アーキテクチャのビット数と実行可能ファイルのリンク形式を示します。どちらの値も文字列で返ります。

値を決定できない場合はパラメータプリセットから与えられる値を返します。bits に `''` を与えた場合、サポートされているポインタサイズを知るために `sizeof(pointer)` (Python バージョン < 1.5.2 では `sizeof(long)`) が使用されます。

この関数は、システムの `file` コマンドを使用します。`file` はほとんどの Unix プラットフォームと一部の非 Unix プラットフォームで利用可能ですが、`file` コマンドが利用できず、かつ *executable* が Python インタープリタでない場合には適切なデフォルト値が返ります。

注釈: Mac OS X (とひょっとすると他のプラットフォーム) では、実行可能ファイルは複数のアーキテクチャを含んだユニバーサル形式かもしれません。

現在のインタプリタが "64-bit" であるかどうかを調べるには、*sys.maxsize* の方が信頼できます:

```
is_64bits = sys.maxsize > 2**32
```

`platform.machine()`
: `'i386'` のような、機種を返します。不明な場合は空文字列を返します。

`platform.node()`
: コンピュータのネットワーク名を返します。ネットワーク名は完全修飾名とは限りません。不明な場合は空文字列を返します。

`platform.platform(`*aliased=0*, *terse=0*`)`
: 実行中プラットフォームを識別する文字列を返します。この文字列には、有益な情報をできるだけ多く付加しています。

戻り値は機械で処理しやすい形式ではなく、**人間にとって読みやすい** 形式となっています。異なったプラットフォームでは異なった戻り値となるようになっています。

aliased が真なら、システムの名称として一般的な名称ではなく、別名を使用して結果を返します。たとえば、SunOS は Solaris となります。この機能は *system_alias()* で実装されています。

terse が真なら、プラットフォームを特定するために最低限必要な情報だけを返します。

バージョン 3.8 で変更: macOS では、*mac_ver()* が空でないリリース文字列を返すとき、darwin のバージョンではなく macOS のバージョンを取得するために、この関数は *mac_ver()* を使うようになりました。

`platform.processor()`
: `'amdk6'` のような、(現実の) プロセッサ名を返します。

不明な場合は空文字列を返します。NetBSD のようにこの情報を提供しない、または *machine()* と同じ値しか返さないプラットフォームも多く存在しますので、注意してください。

`platform.python_build()`
: Python のビルド番号と日付を、`(buildno, builddate)` のタプルで返します。

`platform.python_compiler()`
: Python をコンパイルする際に使用したコンパイラを示す文字列を返します。

`platform.python_branch()`
: Python 実装のバージョン管理システム上のブランチを特定する文字列を返します。

platform.**python_implementation**()
Python 実装を指定する文字列を返します。戻り値は: 'CPython', 'IronPython', 'Jython', 'PyPy' のいずれかです。

platform.**python_revision**()
Python 実装のバージョン管理システム上のリビジョンを特定する文字列を返します。

platform.**python_version**()
Python のバージョンを、'major.minor.patchlevel' 形式の文字列で返します。

sys.version と異なり、patchlevel（デフォルトでは 0) も必ず含まれています。

platform.**python_version_tuple**()
Python のバージョンを、文字列のタプル (major, minor, patchlevel) で返します。

sys.version と異なり、patchlevel（デフォルトでは '0') も必ず含まれています。

platform.**release**()
'2.2.0' や 'NT' のような、システムのリリース情報を返します。不明な場合は空文字列を返します。

platform.**system**()
'Linux' 、'Darwin' 、'Java' 、Windows' のような、システム/OS 名を返します。不明な場合は空文字列を返します。

platform.**system_alias**(*system*, *release*, *version*)
マーケティング目的で使われる一般的な別名に変換して (system, release, version) を返します。混乱を避けるために、情報を並べなおす場合があります。

platform.**version**()
'#3 on degas' のような、システムのリリース情報を返します。不明な場合は空文字列を返します。

platform.**uname**()
極めて可搬性の高い uname インタフェースです。*system*, *node*, *release*, *version*, *machine*, *processor* の 6 つの属性を持った *namedtuple()* を返します。

この関数が *os.uname()* の結果には含まれない 6 番目の属性 (*processor*) を追加することに注意してください。さらに、最初の 2 つの属性については属性名が異なります; *os.uname()* はそれらを sysname と nodename と命名します。

不明な項目は '' となります。

バージョン 3.3 で変更: 結果が tuple から namedtuple に変更されました。

16.14.2 Java プラットフォーム

platform.**java_ver**(*release=''*, *vendor=''*, *vminfo=('', '', '')*, *osinfo=('', '', '')*)

Jython 用のバージョンインターフェースです。

タプル (release, vendor, vminfo, osinfo) を返します。*vminfo* はタプル (vm_name, vm_release, vm_vendor)、*osinfo* はタプル (os_name, os_version, os_arch) です。不明な項目は引数で指定した値 (デフォルトは '') となります。

16.14.3 Windows プラットフォーム

platform.**win32_ver**(*release=''*, *version=''*, *csd=''*, *ptype=''*)

Windows レジストリから追加のバージョン情報を取得して、タプル (release, version, csd, ptype) を返します。それぞれ、OS リリース、バージョン番号、CSD レベル (サービスパック)、OS タイプ (マルチ/シングルプロセッサー) を指しています。

参考: *ptype* はシングルプロセッサの NT 上では 'Uniprocessor Free'、マルチプロセッサでは 'Multiprocessor Free' となります。*'Free'* がついている場合はデバッグ用のコードが含まれていないことを示し、*'Checked'* がついていれば引数や範囲のチェックなどのデバッグ用コードが含まれていることを示します。

platform.**win32_edition**()

現在の Windows のエディションの文字列表現を返します。取りうる返り値には 'Enterprise'、'IoTUAP'、'ServerStandard'、'nanoserver' がありますが、これらに限定されません。

バージョン 3.8 で追加.

platform.**win32_is_iot**()

win32_edition() によって返された Windows のエディションが IoT エディションの時 True を返します。

バージョン 3.8 で追加.

16.14.4 Mac OS プラットフォーム

platform.**mac_ver**(*release=''*, *versioninfo=('', '', '')*, *machine=''*)

Mac OS のバージョン情報を、タプル (release, versioninfo, machine) で返します。*versioninfo* は、タプル (version, dev_stage, non_release_version) です。

不明な項目は '' となります。タプルの要素は全て文字列です。

16.14.5 Unix プラットフォーム

platform.libc_ver(*executable=sys.executable*, *lib=''*, *version=''*, *chunksize=16384*)

executable で指定したファイル（省略時は Python インタープリタ）がリンクしている libc バージョンの取得を試みます。戻り値は文字列のタプル（lib, version）で、不明な項目は引数で指定した値となります。

この関数は、実行形式に追加されるシンボルの細かな違いによって、libc のバージョンを特定します。この違いは gcc でコンパイルされた実行可能ファイルでのみ有効だと思われます。

chunksize にはファイルから情報を取得するために読み込むバイト数を指定します。

16.15 errno --- 標準の errno システムシンボル

このモジュールから標準の errno システムシンボルを取得することができます。個々のシンボルの値は errno に対応する整数値です。これらのシンボルの名前は、linux/include/errno.h から借用されており、かなり網羅的なはずです。

errno.errorcode

errno 値を背後のシステムにおける文字列表現に対応付ける辞書です。例えば、errno.errorcode[errno.EPERM] は 'EPERM' に対応付けられます。

数値のエラーコードをエラーメッセージに変換するには、*os.strerror()* を使ってください。

以下のリストの内、現在のプラットフォームで使われていないシンボルはモジュール上で定義されていません。定義されているシンボルだけを挙げたリストは errno.errorcode.keys() として取得することができます。取得できるシンボルには以下のようなものがあります:

errno.EPERM

許可されていない操作です (Operation not permitted)

errno.ENOENT

そのようなファイルまたはディレクトリは存在しません (No such file or directory)

errno.ESRCH

指定したプロセスは存在しません (No such process)

errno.EINTR

システムコールが中断されました (Interrupted system call)

参考:

このエラーは例外 *InterruptedError* にマップされます。

errno.EIO

I/O エラーです (I/O error)

errno.ENXIO
: そのようなデバイスまたはアドレスは存在しません (No such device or address)

errno.E2BIG
: 引数リストが長すぎます (Arg list too long)

errno.ENOEXEC
: 実行形式にエラーがあります (Exec format error)

errno.EBADF
: ファイル番号が間違っています (Bad file number)

errno.ECHILD
: 子プロセスがありません (No child processes)

errno.EAGAIN
: 再試行してください (Try again)

errno.ENOMEM
: 空きメモリがありません (Out of memory)

errno.EACCES
: 許可がありません (Permission denied)

errno.EFAULT
: 不正なアドレスです (Bad address)

errno.ENOTBLK
: ブロックデバイスが必要です (Block device required)

errno.EBUSY
: そのデバイスまたはリソースは使用中です (Device or resource busy)

errno.EEXIST
: ファイルがすでに存在します (File exists)

errno.EXDEV
: デバイスをまたいだリンクです (Cross-device link)

errno.ENODEV
: そのようなデバイスはありません (No such device)

errno.ENOTDIR
: ディレクトリではありません (Not a directory)

errno.EISDIR
: ディレクトリです (Is a directory)

errno.EINVAL
: 無効な引数です (Invalid argument)

errno.ENFILE
: ファイルテーブルがオーバフローしています (File table overflow)

errno.EMFILE
: 開かれたファイルが多すぎます (Too many open files)

errno.ENOTTY
: タイプライタではありません (Not a typewriter)

errno.ETXTBSY
: テキストファイルが使用中です (Text file busy)

errno.EFBIG
: ファイルが大きすぎます (File too large)

errno.ENOSPC
: デバイス上に空きがありません (No space left on device)

errno.ESPIPE
: 不正なシークです (Illegal seek)

errno.EROFS
: リードオンリーのファイルシステムです (Read-only file system)

errno.EMLINK
: リンクが多すぎます (Too many links)

errno.EPIPE
: 壊れたパイプです (Broken pipe)

errno.EDOM
: 数学引数が関数の定義域を越えています (Math argument out of domain of func)

errno.ERANGE
: 表現できない数学演算結果になりました (Math result not representable)

errno.EDEADLK
: リソースのデッドロックが起きます (Resource deadlock would occur)

errno.ENAMETOOLONG
: ファイル名が長すぎます (File name too long)

errno.ENOLCK
: レコードロッキングが利用できません (No record locks available)

errno.ENOSYS
: 実装されていない機能です (Function not implemented)

errno.ENOTEMPTY
: ディレクトリが空ではありません (Directory not empty)

errno.ELOOP
: これ以上シンボリックリンクを追跡できません (Too many symbolic links encountered)

errno.EWOULDBLOCK
: 操作がブロックします (Operation would block)

errno.ENOMSG
: 指定された型のメッセージはありません (No message of desired type)

errno.EIDRM
: 識別子が除去されました (Identifier removed)

errno.ECHRNG
: チャネル番号が範囲を超えました (Channel number out of range)

errno.EL2NSYNC
: レベル 2 で同期がとれていません (Level 2 not synchronized)

errno.EL3HLT
: レベル 3 で終了しました (Level 3 halted)

errno.EL3RST
: レベル 3 でリセットしました (Level 3 reset)

errno.ELNRNG
: リンク番号が範囲を超えています (Link number out of range)

errno.EUNATCH
: プロトコルドライバが接続されていません (Protocol driver not attached)

errno.ENOCSI
: CSI 構造体がありません (No CSI structure available)

errno.EL2HLT
: レベル 2 で終了しました (Level 2 halted)

errno.EBADE
: 無効な変換です (Invalid exchange)

errno.EBADR
: 無効な要求記述子です (Invalid request descriptor)

errno.EXFULL
: 変換テーブルが一杯です (Exchange full)

errno.ENOANO
: 陰極がありません (No anode)

errno.EBADRQC
: 無効なリクエストコードです (Invalid request code)

errno.EBADSLT
: 無効なスロットです (Invalid slot)

errno.EDEADLOCK
: ファイルロックにおけるデッドロックエラーです (File locking deadlock error)

errno.EBFONT
: フォントファイル形式が間違っています (Bad font file format)

errno.ENOSTR
: ストリーム型でないデバイスです (Device not a stream)

errno.ENODATA
: 利用可能なデータがありません (No data available)

errno.ETIME
: 時間切れです (Timer expired)

errno.ENOSR
: ストリームリソースを使い切りました (Out of streams resources)

errno.ENONET
: 計算機はネットワーク上にありません (Machine is not on the network)

errno.ENOPKG
: パッケージがインストールされていません (Package not installed)

errno.EREMOTE
: 対象物は遠隔にあります (Object is remote)

errno.ENOLINK
: リンクが切られました (Link has been severed)

errno.EADV
: Advertise エラーです (Advertise error)

errno.ESRMNT
: Srmount エラーです (Srmount error)

errno.ECOMM
: 送信時の通信エラーです (Communication error on send)

errno.EPROTO
: プロトコルエラーです (Protocol error)

errno.EMULTIHOP
: 多重ホップを試みました (Multihop attempted)

errno.EDOTDOT
: RFS 特有のエラーです (RFS specific error)

errno.EBADMSG
: データメッセージではありません (Not a data message)

errno.EOVERFLOW
: 定義されたデータ型にとって大きすぎる値です (Value too large for defined data type)

errno.ENOTUNIQ
: 名前がネットワーク上で一意でありません (Name not unique on network)

errno.EBADFD
: ファイル記述子の状態が不正です (File descriptor in bad state)

errno.EREMCHG
: 遠隔のアドレスが変更されました (Remote address changed)

errno.ELIBACC
: 必要な共有ライブラリにアクセスできません (Can not access a needed shared library)

errno.ELIBBAD
: 壊れた共有ライブラリにアクセスしています (Accessing a corrupted shared library)

errno.ELIBSCN
: a.out の .lib セクションが壊れています (.lib section in a.out corrupted)

errno.ELIBMAX
: リンクを試みる共有ライブラリが多すぎます (Attempting to link in too many shared libraries)

errno.ELIBEXEC
: 共有ライブラリを直接実行することができません (Cannot exec a shared library directly)

errno.EILSEQ
: 不正なバイト列です (Illegal byte sequence)

errno.ERESTART
: 割り込みシステムコールを復帰しなければなりません (Interrupted system call should be restarted)

errno.ESTRPIPE
: ストリームパイプのエラーです (Streams pipe error)

errno.EUSERS
: ユーザが多すぎます (Too many users)

errno.ENOTSOCK
: 非ソケットに対するソケット操作です (Socket operation on non-socket)

errno.EDESTADDRREQ
: 目的アドレスが必要です (Destination address required)

errno.EMSGSIZE
: メッセージが長すぎます (Message too long)

errno.EPROTOTYPE
ソケットに対して不正なプロトコル型です (Protocol wrong type for socket)

errno.ENOPROTOOPT
利用できないプロトコルです (Protocol not available)

errno.EPROTONOSUPPORT
サポートされていないプロトコルです (Protocol not supported)

errno.ESOCKTNOSUPPORT
サポートされていないソケット型です (Socket type not supported)

errno.EOPNOTSUPP
通信端点に対してサポートされていない操作です (Operation not supported on transport endpoint)

errno.EPFNOSUPPORT
サポートされていないプロトコルファミリです (Protocol family not supported)

errno.EAFNOSUPPORT
プロトコルでサポートされていないアドレスファミリです (Address family not supported by protocol)

errno.EADDRINUSE
アドレスは使用中です (Address already in use)

errno.EADDRNOTAVAIL
要求されたアドレスを割り当てできません (Cannot assign requested address)

errno.ENETDOWN
ネットワークがダウンしています (Network is down)

errno.ENETUNREACH
ネットワークに到達できません (Network is unreachable)

errno.ENETRESET
リセットによってネットワーク接続が切られました (Network dropped connection because of reset)

errno.ECONNABORTED
ソフトウェアによって接続が終了されました (Software caused connection abort)

errno.ECONNRESET
接続がピアによってリセットされました (Connection reset by peer)

errno.ENOBUFS
バッファに空きがありません (No buffer space available)

errno.EISCONN
通信端点がすでに接続されています (Transport endpoint is already connected)

errno.ENOTCONN
通信端点が接続されていません (Transport endpoint is not connected)

errno.ESHUTDOWN
: 通信端点のシャットダウン後は送信できません (Cannot send after transport endpoint shutdown)

errno.ETOOMANYREFS
: 参照が多すぎます: 接続できません (Too many references: cannot splice)

errno.ETIMEDOUT
: 接続がタイムアウトしました (Connection timed out)

errno.ECONNREFUSED
: 接続を拒否されました (Connection refused)

errno.EHOSTDOWN
: ホストはシステムダウンしています (Host is down)

errno.EHOSTUNREACH
: ホストへの経路がありません (No route to host)

errno.EALREADY
: すでに処理中です (Operation already in progress)

errno.EINPROGRESS
: 現在処理中です (Operation now in progress)

errno.ESTALE
: 無効な NFS ファイルハンドルです (Stale NFS file handle)

errno.EUCLEAN
: 構造のクリーニングが必要です (Structure needs cleaning)

errno.ENOTNAM
: XENIX 名前付きファイルではありません (Not a XENIX named type file)

errno.ENAVAIL
: XENIX セマフォは利用できません (No XENIX semaphores available)

errno.EISNAM
: 名前付きファイルです (Is a named type file)

errno.EREMOTEIO
: 遠隔側の I/O エラーです (Remote I/O error)

errno.EDQUOT
: ディスククオータを超えました (Quota exceeded)

16.16 ctypes --- Python のための外部関数ライブラリ

ctypes は Python のための外部関数ライブラリです。このライブラリは C と互換性のあるデータ型を提供し、動的リンク/共有ライブラリ内の関数呼び出しを可能にします。動的リンク/共有ライブラリを純粋な Python でラップするために使うことができます。

16.16.1 ctypes チュートリアル

注意: このチュートリアルのコードサンプルは動作確認のために *doctest* を使います。コードサンプルの中には Linux、Windows、あるいは Mac OS X 上で異なる動作をするものがあるため、サンプルのコメントに doctest 命令を入れてあります。

注意: いくつかのコードサンプルで ctypes の *c_int* 型を参照しています。`sizeof(long) == sizeof(int)` であるようなプラットフォームでは、この型は *c_long* のエイリアスです。そのため、*c_int* 型を想定しているときに *c_long* が表示されたとしても、混乱しないようにしてください --- 実際には同じ型なのです。

動的リンクライブラリをロードする

動的リンクライブラリをロードするために、*ctypes* は *cdll* をエクスポートします。Windows では *windll* と *oledll* オブジェクトをエクスポートします。

これらのオブジェクトの属性としてライブラリにアクセスすることでライブラリをロードします。*cdll* は、標準 `cdecl` 呼び出し規約を用いて関数をエクスポートしているライブラリをロードします。それに対して、*windll* ライブラリは `stdcall` 呼び出し規約を用いる関数を呼び出します。*oledll* も `stdcall` 呼び出し規約を使いますが、関数が Windows `HRESULT` エラーコードを返すことを想定しています。このエラーコードは関数呼び出しが失敗したとき、*OSError* 例外を自動的に送出させるために使われます。

バージョン 3.3 で変更: Windows エラーは以前は *WindowsError* を送出していましたが、これは現在では *OSError* の別名になっています。

Windows 用の例ですが、`msvcrt` はほとんどの標準 C 関数が含まれている MS 標準 C ライブラリであり、cdecl 呼び出し規約を使うことに注意してください:

```
>>> from ctypes import *
>>> print(windll.kernel32)
<WinDLL 'kernel32', handle ... at ...>
>>> print(cdll.msvcrt)
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt
>>>
```

Windows では通常の `.dll` ファイル拡張子を自動的に追加します。

注釈: `cdll.msvcrt` 経由で標準 C ライブラリにアクセスすると、Python が使用しているライブラリとは互換性のない可能性のある、古いバージョンのライブラリが使用されます。可能な場合には、ネイティブ

Python の機能を使用するか、msvcrt モジュールをインポートして使用してください。

Linux ではライブラリをロードするために拡張子を **含む** ファイル名を指定する必要があるので、ロードしたライブラリに対する属性アクセスはできません。dll ローダーの LoadLibrary() メソッドを使うか、コンストラクタを呼び出して CDLL のインスタンスを作ることでライブラリをロードするかのどちらかを行わなければなりません:

```
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

ロードした dll から関数にアクセスする

dll オブジェクトの属性として関数にアクセスします:

```
>>> from ctypes import *
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.GetModuleHandleA)
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.MyOwnFunction)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
>>>
```

kernel32 や user32 のような win32 システム dll は、多くの場合関数の UNICODE バージョンに加えて ANSI バージョンもエクスポートすることに注意してください。UNICODE バージョンは後ろに W が付いた名前でエクスポートされ、ANSI バージョンは A が付いた名前でエクスポートされます。与えられたモジュールの **モジュールハンドル** を返す win32 GetModuleHandle 関数は次のような C プロトタイプを持ちます。UNICODE バージョンが定義されているかどうかにより GetModuleHandle としてどちらか一つを公開するためにマクロが使われます:

```
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

windll は魔法を使ってどちらか一つを選ぶようなことはしません。GetModuleHandleA もしくは GetModuleHandleW を明示的に指定して必要とするバージョンにアクセスし、バイト列か文字列を使ってそれぞれ呼び出さなければなりません。

時には、dll が関数を "??2@YAPAXI@Z" のような Python 識別子として有効でない名前でエクスポートする

ことがあります。このような場合に関数を取り出すには、*getattr()* を使わなければなりません。:

```
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z")
<_FuncPtr object at 0x...>
>>>
```

Windows では、名前ではなく序数によって関数をエクスポートする dll もあります。こうした関数には序数を使って dll オブジェクトにインデックス指定することでアクセスします:

```
>>> cdll.kernel32[1]
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
>>>
```

関数を呼び出す

これらの関数は他の Python 呼び出し可能オブジェクトと同じように呼び出すことができます。この例では `time()` 関数 (Unix エポックからのシステム時間を秒単位で返す) と、`GetModuleHandleA()` 関数 (win32 モジュールハンドルを返す) を使います。

この例は両方の関数を `NULL` ポインタとともに呼び出します (`None` を `NULL` ポインタとして使う必要があります):

```
>>> print(libc.time(None))
1150640792
>>> print(hex(windll.kernel32.GetModuleHandleA(None)))
0x1d000000
>>>
```

`cdecl` 呼び出し規約を使って `stdcall` 関数を呼び出したときには、*ValueError* が送出されます。逆の場合も同様です:

```
>>> cdll.kernel32.GetModuleHandleA(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>>

>>> windll.msvcrt.printf(b"spam")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

正しい呼び出し規約を知るためには、呼び出したい関数についての C ヘッダファイルもしくはドキュメント

を見なければなりません。

Windows では、関数が無効な引数とともに呼び出された場合の一般保護例外によるクラッシュを防ぐために、*ctypes* は win32 構造化例外処理を使います:

```
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: exception: access violation reading 0x00000020
>>>
```

しかしそれでも他に *ctypes* で Python がクラッシュする状況はあるので、どちらにせよ気を配るべきです。クラッシュのデバッグには、*faulthandler* モジュールが役に立つ場合があります (例えば、誤った C ライブラリ呼び出しによって引き起こされたセグメンテーション違反) 。

None 、整数、バイト列オブジェクトおよび (Unicode) 文字列だけが、こうした関数呼び出しにおいてパラメータとして直接使えるネイティブの Python オブジェクトです。None は C の NULL ポインタとして渡され、バイト文字列とユニコード文字列はそのデータを含むメモリブロックへのポインタ (char * または wchar_t *) として渡されます。Python 整数はプラットホームのデフォルトの C int 型として渡され、その値は C int 型に合うようにマスクされます。

他のパラメータ型をもつ関数呼び出しに移る前に、*ctypes* データ型についてさらに学ぶ必要があります。

基本データ型

ctypes ではいくつもの C 互換のプリミティブなデータ型を定義しています:

ctypes の型	C の型	Python の型
c_bool	_Bool	bool (1)
c_char	char	1 文字のバイト列オブジェクト
c_wchar	wchar_t	1 文字の文字列
c_byte	char	int
c_ubyte	unsigned char	int
c_short	short	int
c_ushort	unsigned short	int
c_int	int	int
c_uint	unsigned int	int
c_long	long	int
c_ulong	unsigned long	int
c_longlong	__int64 または long long	int
c_ulonglong	unsigned __int64 または unsigned long long	int
c_size_t	size_t	int
c_ssize_t	ssize_t または Py_ssize_t	int
c_float	float	浮動小数点数
c_double	double	浮動小数点数
c_longdouble	long double	浮動小数点数
c_char_p	char * (NUL 終端)	バイト列オブジェクトまたは None
c_wchar_p	wchar_t * (NUL 終端)	文字列または None
c_void_p	void *	整数または None

(1) コンストラクタは任意のオブジェクトをその真偽値として受け取ります。

これら全ての型はその型を呼び出すことによって作成でき、オプションとして型と値が合っている初期化子を指定することができます:

```
>>> c_int()
c_long(0)
>>> c_wchar_p("Hello, World")
c_wchar_p(140018365411392)
>>> c_ushort(-3)
c_ushort(65533)
>>>
```

これらの型は変更可能であり、値を後で変更することもできます:

```
>>> i = c_int(42)
>>> print(i)
c_long(42)
>>> print(i.value)
42
>>> i.value = -99
```

(次のページに続く)

(前のページからの続き)

```
>>> print(i.value)
-99
>>>
```

新しい値をポインタ型 *c_char_p*, *c_wchar_p* および *c_void_p* のインスタンスへ代入すると、変わるのは指している **メモリ位置** であって、メモリブロックの **内容ではありません** (これは当然で、なぜなら、Python バイト列オブジェクトは変更不可能だからです):

```
>>> s = "Hello, World"
>>> c_s = c_wchar_p(s)
>>> print(c_s)
c_wchar_p(139966785747344)
>>> print(c_s.value)
Hello World
>>> c_s.value = "Hi, there"
>>> print(c_s)              # the memory location has changed
c_wchar_p(139966783348904)
>>> print(c_s.value)
Hi, there
>>> print(s)                # first object is unchanged
Hello, World
>>>
```

しかし、変更可能なメモリを指すポインタであることを想定している関数へそれらを渡さないように注意すべきです。もし変更可能なメモリブロックが必要なら、ctypes には *create_string_buffer()* 関数があり、いろいろな方法で作成することできます。現在のメモリブロックの内容は raw プロパティを使ってアクセス (あるいは変更) することができます。もし現在のメモリブロックに NUL 終端文字列としてアクセスしたいなら、value プロパティを使ってください:

```
>>> from ctypes import *
>>> p = create_string_buffer(3)            # create a 3 byte buffer, initialized to NUL bytes
>>> print(sizeof(p), repr(p.raw))
3 b'\x00\x00\x00'
>>> p = create_string_buffer(b"Hello")     # create a buffer containing a NUL terminated string
>>> print(sizeof(p), repr(p.raw))
6 b'Hello\x00'
>>> print(repr(p.value))
b'Hello'
>>> p = create_string_buffer(b"Hello", 10) # create a 10 byte buffer
>>> print(sizeof(p), repr(p.raw))
10 b'Hello\x00\x00\x00\x00\x00'
>>> p.value = b"Hi"
>>> print(sizeof(p), repr(p.raw))
10 b'Hi\x00lo\x00\x00\x00\x00\x00'
>>>
```

create_string_buffer() 関数は初期の ctypes リリースにあった c_string() 関数だけでなく、(エイリアスとしてはまだ利用できる) c_buffer() 関数をも置き換えるものです。C の型 wchar_t の Unicode 文字を含む変更可能なメモリブロックを作成するには、*create_unicode_buffer()* 関数を使ってください。

続・関数を呼び出す

printf は *sys.stdout* では **なく** 、本物の標準出力チャンネルへプリントすることに注意してください。したがって、これらの例はコンソールプロンプトでのみ動作し、*IDLE* や *PythonWin* では動作しません。:

```
>>> printf = libc.printf
>>> printf(b"Hello, %s\n", b"World!")
Hello, World!
14
>>> printf(b"Hello, %S\n", "World!")
Hello, World!
14
>>> printf(b"%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf(b"%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: Don't know how to convert parameter 2
>>>
```

前に述べたように、必要な C のデータ型へ変換できるようにするためには、整数、文字列およびバイト列オブジェクトを除くすべての Python 型を対応する *ctypes* 型でラップしなければなりません:

```
>>> printf(b"An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
31
>>>
```

自作のデータ型とともに関数を呼び出す

自作のクラスのインスタンスを関数引数として使えるように、*ctypes* 引数変換をカスタマイズすることもできます。*ctypes* は _as_parameter_ 属性を探し出し、関数引数として使います。もちろん、整数、文字列もしくはバイト列オブジェクトの中の一つでなければなりません:

```
>>> class Bottles:
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf(b"%d bottles of beer\n", bottles)
42 bottles of beer
19
>>>
```

_as_parameter_ インスタンス変数にインスタンスのデータを保持したくない場合は、必要に応じて利用できる属性を作る *property* を定義しても構いません。

要求される引数の型を指定する (関数プロトタイプ)

argtypes 属性を設定することによって、DLL からエクスポートされている関数に要求される引数の型を指定することができます。

argtypes は C データ型のシーケンスでなければなりません (この場合 printf 関数はおそらく良い例ではありません。なぜなら、引数の数が可変であり、フォーマット文字列に依存した異なる型のパラメータを取るからです。一方では、この機能の実験にはとても便利です)。:

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf(b"String '%s', Int %d, Double %f\n", b"Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>>
```

(C の関数のプロトタイプのように) 書式を指定すると互換性のない引数型になるのを防ぎ、引数を有効な型へ変換しようとします。:

```
>>> printf(b"%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: wrong type
>>> printf(b"%s %d %f\n", b"X", 2, 3)
X 2 3.000000
13
>>>
```

関数呼び出しへ渡す自作のクラスを定義した場合には、argtypes シーケンスの中で使えるようにするために、そのクラスに from_param() クラスメソッドを実装しなければなりません。from_param() クラスメソッドは関数呼び出しへ渡された Python オブジェクトを受け取り、型チェックもしくはこのオブジェクトが受け入れ可能であると確かめるために必要なことはすべて行ってから、オブジェクト自身、_as_parameter_ 属性、あるいは、この場合に C 関数引数として渡したい何かの値を返さなければなりません。繰り返しになりますが、その返される結果は整数、文字列、バイト列、*ctypes* インスタンス、あるいは _as_parameter_ 属性をもつオブジェクトであるべきです。

戻り値の型

デフォルトでは、関数は C int を返すと仮定されます。他の戻り値の型を指定するには、関数オブジェクトの restype 属性に設定します。

さらに高度な例として、strchr 関数を使います。この関数は文字列ポインタと char を受け取り、文字列へのポインタを返します。:

```
>>> strchr = libc.strchr
>>> strchr(b"abcdef", ord("d"))
8059983
>>> strchr.restype = c_char_p    # c_char_p is a pointer to a string
>>> strchr(b"abcdef", ord("d"))
b'def'
```

(次のページに続く)

(前のページからの続き)

```
>>> print(strchr(b"abcdef", ord("x")))
None
>>>
```

上の ord("x") 呼び出しを避けたいなら、argtypes 属性を設定することができます。二番目の引数が一文字の Python バイト列オブジェクトから C の char へ変換されます:

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr(b"abcdef", b"d")
'def'
>>> strchr(b"abcdef", b"def")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: one character string expected
>>> print(strchr(b"abcdef", b"x"))
None
>>> strchr(b"abcdef", b"d")
'def'
>>>
```

外部関数が整数を返す場合は、restype 属性として呼び出し可能な Python オブジェクト (例えば、関数またはクラス) を使うこともできます。呼び出し可能オブジェクトは C 関数が返す **整数** とともに呼び出され、この呼び出しの結果は関数呼び出しの結果として使われるでしょう。これはエラーの戻り値をチェックして自動的に例外を送出させるために役に立ちます。:

```
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
...
>>>
>>> GetModuleHandle.restype = ValidHandle
>>> GetModuleHandle(None)
486539264
>>> GetModuleHandle("something silly")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in ValidHandle
OSError: [Errno 126] The specified module could not be found.
>>>
```

WinError はエラーコードの文字列表現を得るために Windows の FormatMessage() api を呼び出し、例外を **返す** 関数です。WinError はオプションでエラーコードパラメータを取ります。このパラメータが使われない場合は、エラーコードを取り出すために *GetLastError()* を呼び出します。

errcheck 属性によってもっと強力なエラーチェック機構を利用できることに注意してください。詳細はリファレンスマニュアルを参照してください。

ポインタを渡す (または、パラメータの参照渡し)

時には、C api 関数がパラメータのデータ型として **ポインタ** を想定していることがあります。おそらくパラメータと同一の場所に書き込むためか、もしくはそのデータが大きすぎて値渡しできない場合です。これは **パラメータの参照渡し** としても知られています。

ctypes は *byref()* 関数をエクスポートしており、パラメータを参照渡しするために使用します。*pointer()* 関数を使っても同じ効果が得られます。しかし、*pointer()* は本当のポインタオブジェクトを構築するためより多くの処理を行うことから、Python 側でポインタオブジェクト自体を必要としないならば *byref()* を使う方がより高速です。:

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer(b'\000' * 32)
>>> print(i.value, f.value, repr(s.value))
0 0.0 b''
>>> libc.sscanf(b"1 3.14 Hello", b"%d %f %s",
...             byref(i), byref(f), s)
3
>>> print(i.value, f.value, repr(s.value))
1 3.1400001049 b'Hello'
>>>
```

構造体と共用体

構造体と共用体は *ctypes* モジュールに定義されている *Structure* および *Union* ベースクラスからの派生クラスでなければなりません。それぞれのサブクラスは _fields_ 属性を定義する必要があります。_fields_ は **フィールド名** と **フィールド型** を持つ *2* **要素タプル** のリストでなければなりません。

フィールド型は *c_int* か他の *ctypes* 型 (構造体、共用体、配列、ポインタ) から派生した *ctypes* 型である必要があります。

以下は、x と y という名前の二つの整数からなる簡単な POINT 構造体の例です。コンストラクタで構造体を初期化する方法も説明しています:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
>>> point = POINT(10, 20)
>>> print(point.x, point.y)
10 20
>>> point = POINT(y=5)
>>> print(point.x, point.y)
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

(次のページに続く)

(前のページからの続き)

```
TypeError: too many initializers
>>>
```

しかし、もっと複雑な構造体を構築することもできます。ある構造体は、他の構造体をフィールド型として使うことで、他の構造体を含むことができます。

upperleft と *lowerright* という名前の二つの POINT を持つ RECT 構造体です。:

```
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                 ("lowerright", POINT)]
...
>>> rc = RECT(point)
>>> print(rc.upperleft.x, rc.upperleft.y)
0 5
>>> print(rc.lowerright.x, rc.lowerright.y)
0 0
>>>
```

入れ子になった構造体はいくつかの方法を用いてコンストラクタで初期化することができます。:

```
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

フィールド *descriptor* (記述子) は **クラス** から取り出せます。デバッグするときに役に立つ情報を得ることができます:

```
>>> print(POINT.x)
<Field type=c_long, ofs=0, size=4>
>>> print(POINT.y)
<Field type=c_long, ofs=4, size=4>
>>>
```

警告: *ctypes* では、ビットフィールドのある共用体や構造体の関数への値渡しはサポートしていません。これは 32-bit の x86 環境では動くかもしれませんが、このライブラリでは一般の場合に動作することは保証していません。

構造体/共用体アライメントとバイトオーダー

デフォルトでは、構造体 (Structure) と共用体 (Union) のフィールドは C コンパイラが行うのと同じ方法でアライメントされています。サブクラスを定義するときに _pack_ クラス属性を指定することでこの動作を変えることは可能です。このクラス属性には正の整数を設定する必要があり、フィールドの最大アライメントを指定します。これは MSVC で `#pragma pack(n)` が行っていること同じです。

ctypes は Structure と Union に対してネイティブのバイトオーダーを使います。ネイティブではないバイトオーダーの構造体を作成するには、*BigEndianStructure*, *LittleEndianStructure*, BigEndianUnion

および LittleEndianUnion ベースクラスの中の一つを使います。これらのクラスにポインタフィールドを持たせることはできません。

構造体と共用体におけるビットフィールド

ビットフィールドを含む構造体と共用体を作ることができます。ビットフィールドは整数フィールドに対してのみ作ることができ、ビット幅は _fields_ タプルの第三要素で指定します。:

```
>>> class Int(Structure):
...     _fields_ = [("first_16", c_int, 16),
...                 ("second_16", c_int, 16)]
...
>>> print(Int.first_16)
<Field type=c_long, ofs=0:0, bits=16>
>>> print(Int.second_16)
<Field type=c_long, ofs=0:16, bits=16>
>>>
```

配列

配列 (Array) はシーケンスであり、決まった数の同じ型のインスタンスを持ちます。

推奨されている配列の作成方法はデータ型に正の整数を掛けることです。:

```
TenPointsArrayType = POINT * 10
```

ややわざとらしいデータ型の例になりますが、他のものに混ざって 4 個の POINT がある構造体です:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class MyStruct(Structure):
...     _fields_ = [("a", c_int),
...                 ("b", c_float),
...                 ("point_array", POINT * 4)]
>>>
>>> print(len(MyStruct().point_array))
4
>>>
```

インスタンスはクラスを呼び出す通常の方法で作成します。:

```
arr = TenPointsArrayType()
for pt in arr:
    print(pt.x, pt.y)
```

上記のコードは ０ ０ という行が並んだものを表示します。配列の要素がゼロで初期化されているためです。

正しい型の初期化子を指定することもできます。:

```
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print(ii)
<c_long_Array_10 object at 0x...>
>>> for i in ii: print(i, end=" ")
...
1 2 3 4 5 6 7 8 9 10
>>>
```

ポインタ

ポインタのインスタンスは *ctypes* 型に対して *pointer()* 関数を呼び出して作成します。:

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

次のように、ポインタインスタンスは、ポインタが指すオブジェクト (上の例では `i`) を返す *contents* 属性を持ちます:

```
>>> pi.contents
c_long(42)
>>>
```

ctypes は OOR (original object return 、元のオブジェクトを返すこと) ではないことに注意してください。属性を取り出す度に、新しい同等のオブジェクトを作成しているのです。:

```
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
>>>
```

別の *c_int* インスタンスがポインタの contents 属性に代入されると、これが記憶されているメモリ位置を指すポインタに変化します。:

```
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
c_long(99)
>>>
```

ポインタインスタンスは整数でインデックス指定することもできます。:

```
>>> pi[0]
99
>>>
```

整数インデックスへ代入するとポインタが指す値が変更されます。:

```
>>> print(i)
c_long(99)
>>> pi[0] = 22
>>> print(i)
c_long(22)
>>>
```

0 ではないインデックスを使うこともできますが、C の場合と同じように自分が何をしているかを理解している必要があります。任意のメモリ位置にアクセスもしくは変更できるのです。一般的にこの機能を使うのは、C 関数からポインタを受け取り、そのポインタが単一の要素ではなく実際に配列を指していると **分かっている** 場合だけです。

舞台裏では、*pointer()* 関数は単にポインタインスタンスを作成するという以上のことを行っています。はじめにポインタ **型** を作成する必要があります。これは任意の *ctypes* 型を受け取る *POINTER()* 関数を使って行われ、新しい型を返します:

```
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_long'>
>>> PI(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected c_long instead of int
>>> PI(c_int(42))
<ctypes.LP_c_long object at 0x...>
>>>
```

ポインタ型を引数なしで呼び出すと NULL ポインタを作成します。NULL ポインタは False ブール値を持っています。:

```
>>> null_ptr = POINTER(c_int)()
>>> print(bool(null_ptr))
False
>>>
```

ctypes はポインタの指す値を取り出すときに NULL かどうかを調べます (しかし、NULL でない不正なポインタの指す値の取り出す行為は Python をクラッシュさせるでしょう)。:

```
>>> null_ptr[0]
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>

>>> null_ptr[0] = 1234
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>
```

型変換

たいていの場合、ctypes は厳密な型チェックを行います。これが意味するのは、関数の argtypes リスト内に、もしくは、構造体定義におけるメンバーフィールドの型として POINTER(c_int) がある場合、厳密に同じ型のインスタンスだけを受け取るということです。このルールには ctypes が他のオブジェクトを受け取る場合に例外がいくつかあります。例えば、ポインタ型の代わりに互換性のある配列インスタンスを渡すことができます。このように、POINTER(c_int) に対して、ctypes は c_int の配列を受け取ります。:

```
>>> class Bar(Structure):
...     _fields_ = [("count", c_int), ("values", POINTER(c_int))]
...
>>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print(bar.values[i])
...
1
2
3
>>>
```

それに加えて、argtypes で関数の引数が明示的に (POINTER(c_int) などの) ポインタ型であると宣言されていた場合、ポインタ型が指し示している型のオブジェクト (この場合では c_int) を関数に渡すことができます。この場合 ctypes は、必要となる *byref()* での変換を自動的に適用します。

POINTER 型フィールドを NULL に設定するために、None を代入してもかまいません。:

```
>>> bar.values = None
>>>
```

時には、非互換な型のインスタンスであることもあります。C では、ある型を他の型へキャストすることができます。*ctypes* は同じやり方で使える *cast()* 関数を提供しています。上で定義した Bar 構造体は POINTER(c_int) ポインタまたは *c_int* 配列を values フィールドに対して受け取り、他の型のインスタンスは受け取りません:

```
>>> bar.values = (c_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: incompatible types, c_byte_Array_4 instance instead of LP_c_long instance
>>>
```

このような場合には、*cast()* 関数が便利です。

cast() 関数は ctypes インスタンスを異なる ctypes データ型を指すポインタへキャストするために使えます。*cast()* は二つのパラメータ、ある種のポインタかそのポインタへ変換できる ctypes オブジェクトと、ctypes ポインタ型を取ります。そして、第二引数のインスタンスを返します。このインスタンスは第一引数と同じメモリブロックを参照しています:

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
>>>
```

したがって、*cast()* を Bar 構造体の values フィールドへ代入するために使うことができます:

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print(bar.values[0])
0
>>>
```

不完全型

不完全型 はメンバーがまだ指定されていない構造体、共用体もしくは配列です。C では、前方宣言により指定され、後で定義されます。:

```
struct cell; /* forward declaration */

struct cell {
    char *name;
    struct cell *next;
};
```

ctypes コードへの直接的な変換ではこうなるでしょう。しかし、動作しません:

```
>>> class cell(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
>>>
```

なぜなら、新しい class cell はクラス文自体の中では利用できないからです。*ctypes* では、cell クラスを定義して、_fields_ 属性をクラス文の後で設定することができます。:

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
...                  ("next", POINTER(cell))]
>>>
```

試してみましょう。cell のインスタンスを二つ作り、互いに参照し合うようにします。最後に、つながったポインタを何度かたどります。:

```
>>> c1 = cell()
>>> c1.name = b"foo"
>>> c2 = cell()
>>> c2.name = b"bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print(p.name, end=" ")
...     p = p.next[0]
...
foo bar foo bar foo bar foo bar
>>>
```

コールバック関数

ctypes は C の呼び出し可能な関数ポインタを Python 呼び出し可能オブジェクトから作成できるようにします。これらは **コールバック関数** と呼ばれることがあります。

最初に、コールバック関数のためのクラスを作る必要があります。そのクラスには呼び出し規約、戻り値の型およびこの関数が受け取る引数の数と型についての情報があります。

CFUNCTYPE() ファクトリ関数は通常の `cdecl` 呼び出し規約を用いてコールバック関数のための型を作成します。Windows では、*WINFUNCTYPE()* ファクトリ関数が `stdcall` 呼び出し規約を用いてコールバック関数の型を作成します。

これらのファクトリ関数はともに最初の引数に戻り値の型、残りの引数としてコールバック関数が想定する引数の型を渡して呼び出されます。

標準 C ライブラリの `qsort()` 関数を使う例を示します。これはコールバック関数の助けをかりて要素をソートするために使われます。`qsort()` は整数の配列をソートするために使われます:

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
>>>
```

`qsort()` はソートするデータを指すポインタ、データ配列の要素の数、要素の一つの大きさ、およびコールバック関数である比較関数へのポインタを引数に渡して呼び出さなければなりません。そして、コールバック関数は要素を指す二つのポインタを渡されて呼び出され、一番目が二番目より小さいなら負の数を、等しいならゼロを、それ以外なら正の数を返さなければなりません。

コールバック関数は整数へのポインタを受け取り、整数を返す必要があります。まず、コールバック関数のための `type` を作成します。:

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>>
```

まず初めに、これが受け取った変数を表示するだけのシンプルなコールバックです:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

結果は以下の通りです:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
>>>
```

ここで 2 つの要素を実際に比較し、役に立つ結果を返します:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>>
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

簡単に確認できるように、配列を次のようにソートしました:

```
>>> for i in ia: print(i, end=" ")
...
1 5 7 33 99
>>>
```

関数ファクトリはデコレータファクトリとしても使えるので、次のようにも書けます:

```
>>> @CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
... def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>> qsort(ia, len(ia), sizeof(c_int), py_cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
```

(次のページに続く)

(前のページからの続き)

```
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

注釈: C コードから *CFUNCTYPE()* オブジェクトが使用される限り、そのオブジェクトへの参照を確実に保持してください。*ctypes* は参照を保持しないため、あなたが参照を保持しないと、オブジェクトはガベージコレクションの対象となり、コールバックが行われたときにプログラムをクラッシュさせる場合があります。

同様に、コールバック関数が Python の管理外 (例えば、コールバックを呼び出す外部のコード) で作られたスレッドで呼び出された場合、ctypes は全ての呼び出しごとに新しいダミーの Python スレッドを作成することに注意してください。この動作はほとんどの目的に対して正しいものですが、同じ C スレッドからの呼び出しだったとしても、*threading.local* で格納された値は異なるコールバックをまたいで生存は **しません** 。

---

dll からエクスポートされた値へアクセスする

共有ライブラリの一部は関数だけでなく変数もエクスポートしています。Python ライブラリにある例としては `Py_OptimizeFlag` 、起動時の `-O` または `-OO` フラグに依存して、0 , 1 または 2 が設定される整数があります。

ctypes は型の `in_dll()` クラスメソッドを使ってこのように値にアクセスできます。*pythonapi* は Python C api へアクセスできるようにするための予め定義されたシンボルです。:

```
>>> opt_flag = c_int.in_dll(pythonapi, "Py_OptimizeFlag")
>>> print(opt_flag)
c_long(0)
>>>
```

インタープリタが `-O` を指定されて動き始めた場合、サンプルは `c_long(1)` を表示するでしょうし、`-OO` が指定されたならば `c_long(2)` を表示するでしょう。

ポインタの使い方を説明する拡張例では、Python がエクスポートする `PyImport_FrozenModules` ポインタにアクセスします。

この値のドキュメントから引用すると:

> このポインタは `struct _frozen` のレコードからなり、終端の要素のメンバが `NULL` かゼロになっているような配列を指すよう初期化されます。フリーズされたモジュールをインポートするとき、このテーブルを検索します。サードパーティ製のコードからこのポインタに仕掛けを講じて、動的に生成されたフリーズ化モジュールの集合を提供するようにできます。

これで、このポインタを操作することが役に立つことを証明できるでしょう。例の大きさを制限するために、このテーブルを *ctypes* を使って読む方法だけを示します。:

```
>>> from ctypes import *
>>>
>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("code", POINTER(c_ubyte)),
...                 ("size", c_int)]
...
>>>
```

私たちは struct _frozen データ型を定義済みなので、このテーブルを指すポインタを得ることができます:

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "PyImport_FrozenModules")
>>>
```

table が struct_frozen レコードの配列への pointer なので、その配列に対して反復処理を行えます。しかし、ループが確実に終了するようにする必要があります。なぜなら、ポインタに大きさの情報がないからです。遅かれ早かれ、アクセス違反か何かでクラッシュすることになるでしょう。NULL エントリに達したときはループを抜ける方が良いです:

```
>>> for item in table:
...     if item.name is None:
...         break
...     print(item.name.decode("ascii"), item.size)
...
_frozen_importlib 31764
_frozen_importlib_external 41499
__hello__ 161
__phello__ -161
__phello__.spam 161
>>>
```

標準 Python はフローズンモジュールとフローズンパッケージ (負の size メンバーで表されています) を持っているという事実はあまり知られておらず、テストにだけ使われています。例えば、import __hello__ を試してみてください。

びっくり仰天

There are some edges in *ctypes* where you might expect something other than what actually happens.

次に示す例について考えてみてください。:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
```

(次のページに続く)

(前のページからの続き)

```
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
3 4 3 4
>>>
```

うーん、最後の文に 3 4 1 2 と表示されることを期待していたはずです。何が起きたのでしょうか? 上の行の `rc.a, rc.b = rc.b, rc.a` の各段階はこのようになります。:

```
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
>>>
```

`temp0` と `temp1` は前記の `rc` オブジェクトの内部バッファでまだ使われているオブジェクトです。したがって、`rc.a = temp0` を実行すると `temp0` のバッファ内容が `rc` のバッファへコピーされます。さらに、これは `temp1` の内容を変更します。そのため、最後の代入 `rc.b = temp1` は、期待する結果にはならないのです。

Structure 、Union および Array のサブオブジェクトを取り出しても、そのサブオブジェクトが **コピー** されるわけではなく、ルートオブジェクトの内部バッファにアクセスするラッパーオブジェクトを取り出すことを覚えておいてください。

期待とは違う振る舞いをする別の例はこれです:

```
>>> s = c_char_p()
>>> s.value = b"abc def ghi"
>>> s.value
b'abc def ghi'
>>> s.value is s.value
False
>>>
```

注釈: *c_char_p* からインスタンス化されたオブジェクトは、bytes または整数に設定された値しか持てません。

なぜ `False` と表示されるのでしょうか? ctypes インスタンスはメモリと、メモリの内容にアクセスするいくつかの *descriptor* (記述子) を含むオブジェクトです。メモリブロックに Python オブジェクトを保存してもオブジェクト自身が保存される訳ではなく、オブジェクトの `contents` が保存されます。その contents に再アクセスすると新しい Python オブジェクトがその度に作られます。

可変サイズのデータ型

ctypes は可変サイズの配列と構造体をサポートしています。

resize() 関数は既存の ctypes オブジェクトのメモリバッファのサイズを変更したい場合に使えます。この関数は第一引数にオブジェクト、第二引数に要求されたサイズをバイト単位で指定します。メモリブロックはオブジェクト型で指定される通常のメモリブロックより小さくすることはできません。これをやろうとすると、*ValueError* が送出されます。:

```
>>> short_array = (c_short * 4)()
>>> print(sizeof(short_array))
8
>>> resize(short_array, 4)
Traceback (most recent call last):
    ...
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>>
```

これはこれで上手くいっていますが、この配列の追加した要素へどうやってアクセスするのでしょうか? この型は要素の数が 4 個であるとまだ認識しているので、他の要素にアクセスするとエラーになります。:

```
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
    ...
IndexError: invalid index
>>>
```

ctypes で可変サイズのデータ型を使うもう一つの方法は、必要なサイズが分かった後に Python の動的性質を使って一つ一つデータ型を (再) 定義することです。

16.16.2 ctypes リファレンス

共有ライブラリを見つける

コンパイルされる言語でプログラミングしている場合、共有ライブラリはプログラムをコンパイル/リンクしているときと、そのプログラムが動作しているときにアクセスされます。

ctypes ライブラリローダーはプログラムが動作しているときのように振る舞い、ランタイムローダーを直接呼び出すのに対し、`find_library()` 関数の目的はコンパイラまたはランタイムローダーが行うのと似た方法でライブラリを探し出すことです。(複数のバージョンの共有ライブラリがあるプラットホームでは、一番最近に見つかったものがロードされます)。

`ctypes.util` モジュールはロードするライブラリを決めるのに役立つ関数を提供します。

ctypes.util.find_library(*name*)

ライブラリを見つけてパス名を返そうと試みます。*name* は lib のような接頭辞、.so, .dylib のような接尾辞、あるいは、バージョン番号が何も付いていないライブラリの名前です (これは posix リンカのオプション -l に使われている形式です)。ライブラリが見つからないときは None を返します。

厳密な機能はシステムに依存します。

Linux では、find_library() はライブラリファイルを見つけるために外部プログラム (/sbin/ldconfig, gcc, objdump と ld) を実行しようとします。ライブラリファイルのファイル名を返します。

バージョン 3.6 で変更: Linux では、ライブラリを検索する際に、他の方法でライブラリが見つけられない場合は、LD_LIBRARY_PATH 環境変数の値が使われます

ここに例があります:

```
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
'libbz2.so.1.0'
>>>
```

OS X では、find_library() はライブラリの位置を探すために、予め定義された複数の命名方法とパスを試し、成功すればフルパスを返します。:

```
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
>>>
```

Windows では、find_library() はシステムの探索パスに沿って探し、フルパスを返します。しかし、予め定義された命名方法がないため、find_library("c") のような呼び出しは失敗し、None を返します。

ctypes で共有ライブラリをラップする場合、find_library() を使って実行時にライブラリの場所を特定するのではなく、共有ライブラリの名前を開発時に決めておいて、ラッパーモジュールにハードコードする方が良い **かもしれません** 。

共有ライブラリをロードする

共有ライブラリを Python プロセスへロードする方法はいくつかあります。一つの方法は下記のクラスの一つをインスタンス化することです:

class ctypes.CDLL(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=0*)

このクラスのインスタンスはロードされた共有ライブラリをあらわします。これらのライブラリの関数は標準 C 呼び出し規約を使用し、`int` を返すと仮定されます。

On Windows creating a *CDLL* instance may fail even if the DLL name exists. When a dependent DLL of the loaded DLL is not found, a *OSError* error is raised with the message *"[WinError 126] The specified module could not be found"*. This error message does not contain the name of the missing DLL because the Windows API does not return this information making this error hard to diagnose. To resolve this error and determine which DLL is not found, you need to find the list of dependent DLLs and determine which one is not found using Windows debugging and tracing tools.

参考:

Microsoft DUMPBIN tool -- A tool to find DLL dependents.

class ctypes.OleDLL(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=0*)

Windows 用: このクラスのインスタンスはロードされた共有ライブラリをあらわします。これらのライブラリの関数は `stdcall` 呼び出し規約を使用し、windows 固有の *HRESULT* コードを返すと仮定されます。*HRESULT* 値には関数呼び出しが失敗したのか成功したのかを特定する情報とともに、補足のエラーコードが含まれます。戻り値が失敗を知らせたならば、*OSError* が自動的に送出されます。

バージョン 3.3 で変更: 以前は *WindowsError* を送出していました。

class ctypes.WinDLL(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=0*)

Windows 用: このクラスのインスタンスはロードされた共有ライブラリをあらわします。これらのライブラリの関数は `stdcall` 呼び出し規約を使用し、デフォルトでは `int` を返すと仮定されます。

Windows CE では標準呼び出し規約だけが使われます。便宜上、このプラットフォームでは、*WinDLL* と *OleDLL* が標準呼び出し規約を使用します。

これらのライブラリがエクスポートするどの関数でも呼び出す前に Python *global interpreter lock* は解放され、後でまた獲得されます。

class ctypes.PyDLL(*name*, *mode=DEFAULT_MODE*, *handle=None*)

Python GIL が関数呼び出しの間解放 **されず** 、関数実行の後に Python エラーフラグがチェックされるということを除けば、このクラスのインスタンスは *CDLL* インスタンスのように振る舞います。エラーフラグがセットされた場合、Python 例外が送出されます。

要するに、これは Python C api 関数を直接呼び出すのに便利だというだけです。

これらすべてのクラスは少なくとも一つの引数、すなわちロードする共有ライブラリのパスを渡して呼び出すことでインスタンス化されます。すでにロード済みの共有ライブラリへのハンドルがあるなら、`handle` 名前

付き引数として渡すことができます。土台となっているプラットフォームの dlopen または LoadLibrary 関数がプロセスへライブラリをロードするために使われ、そのライブラリに対するハンドルを得ます。

mode パラメータを使うと、ライブラリがどうやってロードされたかを特定できます。詳細は *dlopen(3)* マニュアルページを参考にしてください。Windows では *mode* は無視されます。POSIX システムでは RTLD_NOW が常に追加され、設定変更はできません。

use_errno 変数が真に設定されたとき、システムの *errno* エラーナンバーに安全にアクセスする ctypes の仕組みが有効化されます。*ctypes* はシステムの *errno* 変数のスレッド限定のコピーを管理します。もし、use_errno=True の状態で作られた外部関数を呼び出したなら、関数呼び出し前の *errno* 変数は ctypes のプライベートコピーと置き換えられ、同じことが関数呼び出しの直後にも発生します。

ctypes.get_errno() 関数は ctypes のプライベートコピーの値を返します。そして、*ctypes.set_errno()* 関数は ctypes のプライベートコピーを置き換え、以前の値を返します。

use_last_error パラメータは、真に設定されたとき、*GetLastError()* と SetLastError() Windows API によって管理される Windows エラーコードに対するのと同じ仕組みが有効化されます。*ctypes.get_last_error()* と *ctypes.set_last_error()* は Windows エラーコードの ctypes プライベートコピーを変更したり要求したりするのに使われます。

The *winmode* parameter is used on Windows to specify how the library is loaded (since *mode* is ignored). It takes any value that is valid for the Win32 API LoadLibraryEx flags parameter. When omitted, the default is to use the flags that result in the most secure DLL load to avoiding issues such as DLL hijacking. Passing the full path to the DLL is the safest way to ensure the correct library and dependencies are loaded.

バージョン 3.8 で変更: *winmode* 引数が追加されました。

ctypes.RTLD_GLOBAL
: *mode* パラメータとして使うフラグ。このフラグが利用できないプラットフォームでは、整数のゼロと定義されています。

ctypes.RTLD_LOCAL
: *mode* パラメータとして使うフラグ。これが利用できないプラットフォームでは、*RTLD_GLOBAL* と同様です。

ctypes.DEFAULT_MODE
: 共有ライブラリをロードするために使われるデフォルトモード。OSX 10.3 では *RTLD_GLOBAL* であり、そうでなければ *RTLD_LOCAL* と同じです。

これらのクラスのインスタンスには公開メソッドはありません。共有ライブラリからエクスポートされた関数は、属性として、もしくは添字でアクセスできます。属性を通した関数へのアクセスは結果がキャッシュされ、従って繰り返しアクセスされると毎回同じオブジェクトを返すことに注意してください。それとは反対に、添字を通したアクセスは毎回新しいオブジェクトを返します:

```
>>> from ctypes import CDLL
>>> libc = CDLL("libc.so.6")  # On Linux
>>> libc.time == libc.time
True
```

(次のページに続く)

(前のページからの続き)

```
>>> libc['time'] == libc['time']
False
```

次に述べる公開属性が利用できます。それらの名前はエクスポートされた関数名に衝突しないように下線で始まります。:

PyDLL._handle
: ライブラリへのアクセスに用いられるシステムハンドル。

PyDLL._name
: コンストラクタに渡されたライブラリの名前。

共有ライブラリは (*LibraryLoader* クラスのインスタンスである) 前もって作られたオブジェクトの一つを使うことによってロードすることもできます。それらの LoadLibrary() メソッドを呼び出すか、ローダーインスタンスの属性としてライブラリを取り出すかのどちらかによりロードします。

class ctypes.LibraryLoader(*dlltype*)
: 共有ライブラリをロードするクラス。*dlltype* は *CDLL* 、*PyDLL* 、*WinDLL* もしくは *OleDLL* 型の一つであるべきです。

  __getattr__() は次のような特別なはたらきをします。ライブラリローダーインスタンスの属性として共有ライブラリにアクセスするとそれがロードされるということを可能にします。結果はキャッシュされます。そのため、繰り返し属性アクセスを行うといつも同じライブラリが返されます。

  LoadLibrary(*name*)
  : 共有ライブラリをプロセスへロードし、それを返します。このメソッドはライブラリの新しいインスタンスを常に返します。

これらの前もって作られたライブラリローダーを利用することができます。:

ctypes.cdll
: *CDLL* インスタンスを作ります。

ctypes.windll
: Windows 用: *WinDLL* インスタンスを作ります。

ctypes.oledll
: Windows 用: *OleDLL* インスタンスを作ります。

ctypes.pydll
: *PyDLL* インスタンスを作ります。

C Python api に直接アクセスするために、すぐに使用できる Python 共有ライブラリオブジェクトが次のように用意されています。

ctypes.pythonapi
: 属性として Python C api 関数を公開する *PyDLL* のインスタンス。これらすべての関数は C int を返すと仮定されますが、もちろん常に正しいとは限りません。そのため、これらの関数を使うためには正しい restype 属性を代入しなければなりません。

引数 name を指定して 監査イベント ctypes.dlopen を送出します。

引数 library, name を指定して 監査イベント ctypes.dlsym を送出します。

引数 handle, name を指定して 監査イベント ctypes.dlsym/handle を送出します。

外部関数

前節で説明した通り、外部関数はロードされた共有ライブラリの属性としてアクセスできます。デフォルトではこの方法で作成された関数オブジェクトはどんな数の引数でも受け取り、引数としてどんな ctypes データのインスタンスをも受け取り、そして、ライブラリローダーが指定したデフォルトの結果の値の型を返します。関数オブジェクトはプライベートクラスのインスタンスです。:

class ctypes._FuncPtr

C の呼び出し可能外部関数のためのベースクラス。

外部関数のインスタンスも C 互換データ型です。それらは C の関数ポインタを表しています。

この振る舞いは外部関数オブジェクトの特別な属性に代入することによって、カスタマイズすることができます。

restype

外部関数の結果の型を指定するために ctypes 型を代入する。何も返さない関数を表す void に対しては None を使います。

ctypes 型ではない呼び出し可能な Python オブジェクトを代入することは可能です。このような場合、関数が C int を返すと仮定され、呼び出し可能オブジェクトはこの整数を引数に呼び出されます。さらに処理を行ったり、エラーチェックをしたりできるようにするためです。これの使用は推奨されません。より柔軟な後処理やエラーチェックのためには *restype* として ctypes 型を使い、*errcheck* 属性へ呼び出し可能オブジェクトを代入してください。

argtypes

関数が受け取る引数の型を指定するために ctypes 型のタプルを代入します。stdcall 呼び出し規約を使う関数はこのタプルの長さと同じ数の引数で呼び出されます。C 呼び出し規約を使う関数は、追加の不特定の引数も取ります。

外部関数が呼ばれたとき、それぞれの実引数は *argtypes* タプルの要素の from_param() クラスメソッドへ渡されます。このメソッドは実引数を外部関数が受け取るオブジェクトに合わせて変えられるようにします。例えば、*argtypes* タプルの *c_char_p* 要素は、ctypes 変換規則にしたがって引数として渡された文字列をバイト列オブジェクトへ変換するでしょう。

新: ctypes 型でない要素を argtypes に入れることができますが、個々の要素は引数として使える値 (整数、文字列、ctypes インスタンス) を返す from_param() メソッドを持っていなければなりません。これにより関数パラメータとしてカスタムオブジェクトを適合するように変更できるアダプタが定義可能となります。

errcheck

Python 関数または他の呼び出し可能オブジェクトをこの属性に代入します。呼び出し可能オブジェクトは三つ以上の引数とともに呼び出されます。

callable(*result*, *func*, *arguments*)

result は外部関数が返すもので、restype 属性で指定されます。

func は外部関数オブジェクト自身で、これにより複数の関数の処理結果をチェックまたは後処理するために、同じ呼び出し可能オブジェクトを再利用できるようになります。

arguments は関数呼び出しに最初に渡されたパラメータが入ったタプルです。これにより使われた引数に基づいた特別な振る舞いをさせることができるようになります。

この関数が返すオブジェクトは外部関数呼び出しから返された値でしょう。しかし、戻り値をチェックして、外部関数呼び出しが失敗しているなら例外を送出させることもできます。

exception ctypes.ArgumentError

この例外は外部関数呼び出しが渡された引数を変換できなかったときに送出されます。

引数 code を指定して 監査イベント ctypes.seh_exception を送出します。

引数 func_pointer, arguments を指定して 監査イベント ctypes.call_function を送出します。

関数プロトタイプ

外部関数は関数プロトタイプをインスタンス化することによって作成されます。関数プロトタイプは C の関数プロトタイプと似ています。実装は定義せずに、関数 (返り値の型、引数の型、呼び出し規約) を記述します。ファクトリ関数は、その関数に要求される返り値の型と引数の型とともに呼び出されます。そしてこの関数はデコレータファクトリとしても使え、@wrapper 構文で他の関数に適用できます。例については コールバック関数 を参照してください。

ctypes.CFUNCTYPE(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)

返された関数プロトタイプは標準 C 呼び出し規約をつかう関数を作成します。関数は呼び出されている間 GIL を解放します。*use_errno* が真に設定されれば、呼び出しの前後で System 変数 *errno* の ctypes プライベートコピーは本当の *errno* の値と交換されます。*use_last_error* も Windows エラーコードに対するのと同様です。

ctypes.WINFUNCTYPE(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)

Windows のみ: 返された関数プロトタイプは stdcall 呼び出し規約を使う関数を作成します。ただし、*WINFUNCTYPE()* が *CFUNCTYPE()* と同じである Windows CE を除きます。関数は呼び出されている間 GIL を解放します。*use_errno* と *use_last_error* は前述と同じ意味を持ちます。

ctypes.PYFUNCTYPE(*restype*, **argtypes*)

返された関数プロトタイプは Python 呼び出し規約を使う関数を作成します。関数は呼び出されている間 GIL を解放 **しません**。

ファクトリ関数によって作られた関数プロトタイプは呼び出しのパラメータの型と数に依存した別の方法でインスタンス化することができます。:

prototype(*address*)

指定されたアドレス (整数でなくてはなりません) の外部関数を返します。

prototype(*callable*)

Python の *callable* から C の呼び出し可能関数 (コールバック関数) を作成します。

prototype(*func_spec*[, *paramflags*])

共有ライブラリがエクスポートしている外部関数を返します。*func_spec* は 2 要素タプル `(name_or_ordinal, library)` でなければなりません。第一要素はエクスポートされた関数の名前である文字列、またはエクスポートされた関数の序数である小さい整数です。第二要素は共有ライブラリインスタンスです。

prototype(*vtbl_index*, *name*[, *paramflags*[, *iid*]])

COM メソッドを呼び出す外部関数を返します。*vtbl_index* は仮想関数テーブルのインデックスで、非負の小さい整数です。*name* は COM メソッドの名前です。*iid* はオプションのインターフェイス識別子へのポインタで、拡張されたエラー情報の提供のために使われます。

COM メソッドは特殊な呼び出し規約を用います。このメソッドは `argtypes` タプルに指定されたパラメータに加えて、第一引数として COM インターフェイスへのポインタを必要とします。

オプションの *paramflags* パラメータは上述した機能より多機能な外部関数ラッパーを作成します。

paramflags は `argtypes` と同じ長さのタプルでなければなりません。

このタプルの個々の要素はパラメータについてのより詳細な情報を持ち、1 、2 もしくは 3 要素を含むタプルでなければなりません。

第一要素はパラメータについてのフラグの組み合わせを含んだ整数です。

1　入力パラメータを関数に指定します。

2　出力パラメータ。外部関数が値を書き込みます。

4　デフォルトで整数ゼロになる入力パラメータ。

オプションの第二要素はパラメータ名の文字列です。これが指定された場合は、外部関数を名前付きパラメータで呼び出すことができます。

オプションの第三要素はこのパラメータのデフォルト値です。

この例では、デフォルトパラメータと名前付き引数をサポートするために Windows の `MessageBoxW` 関数をラップする方法を示します。windows のヘッダファイルの C の宣言は次の通りです:

```
WINUSERAPI int WINAPI
MessageBoxW(
    HWND hWnd,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType);
```

ctypes を使ってラップします。:

```
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCWSTR, UINT
```

(次のページに続く)

(前のページからの続き)

```
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCWSTR, LPCWSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption", "Hello from ctypes"), (1,
↪"flags", 0)
>>> MessageBox = prototype(("MessageBoxW", windll.user32), paramflags)
```

これで外部関数の MessageBox を次のような方法で呼び出すことができるようになりました:

```
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
```

二番目の例は出力パラメータについて説明します。win32 の GetWindowRect 関数は、指定されたウィンドウの大きさを呼び出し側が与える RECT 構造体へコピーすることで取り出します。C の宣言はこうです。:

```
WINUSERAPI BOOL WINAPI
GetWindowRect(
     HWND hWnd,
     LPRECT lpRect);
```

ctypes を使ってラップします。:

```
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(("GetWindowRect", windll.user32), paramflags)
>>>
```

出力パラメータを持つ関数は、単一のパラメータがある場合にはその出力パラメータ値を、複数のパラメータがある場合には出力パラメータ値が入ったタプルを、それぞれ自動的に返します。そのため、GetWindowRect 関数は呼び出されると RECT インスタンスを返します。

さらに出力処理やエラーチェックを行うために、出力パラメータを errcheck プロトコルと組み合わせることができます。win32 GetWindowRect api 関数は成功したか失敗したかを知らせるために BOOL を返します。そのため、この関数はエラーチェックを行って、api 呼び出しが失敗した場合に例外を送出させることができます。:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

errcheck 関数が受け取った引数タプルを変更なしに返した場合、*ctypes* は出力パラメータに対する通常の処理を続けます。RECT インスタンスの代わりに window 座標のタプルを返すには、関数のフィールドを取り出し、代わりにそれらを返すことができます。この場合、通常処理は行われなくなります:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

ユーティリティー関数

ctypes.**addressof**(*obj*)

メモリバッファのアドレスを示す整数を返します。*obj* は ctypes 型のインスタンスでなければなりません。

引数 `obj` を指定して 監査イベント `ctypes.addressof` を送出します。

ctypes.**alignment**(*obj_or_type*)

ctypes 型のアライメントの必要条件を返します。*obj_or_type* は ctypes 型またはインスタンスでなければなりません。

ctypes.**byref**(*obj*[, *offset*])

obj (ctypes 型のインスタンスでなければならない) への軽量ポインタを返します。*offset* はデフォルトでは 0 で、内部ポインターへ加算される整数です。

`byref(obj, offset)` は、C コードとしては、以下のようにみなされます。:

```
(((char *)&obj) + offset)
```

返されるオブジェクトは外部関数呼び出しのパラメータとしてのみ使用できます。`pointer(obj)` と似たふるまいをしますが、作成が非常に速く行えます。

ctypes.**cast**(*obj*, *type*)

この関数は C のキャスト演算子に似ています。*obj* と同じメモリブロックを指している *type* の新しいインスタンスを返します。*type* はポインタ型でなければならず、*obj* はポインタとして解釈できるオブジェクトでなければなりません。

ctypes.**create_string_buffer**(*init_or_size*, *size=None*)

この関数は変更可能な文字バッファを作成します。返されるオブジェクトは *c_char* の ctypes 配列です。

init_or_size は配列のサイズを指定する整数もしくは配列要素を初期化するために使われるバイト列オブジェクトである必要があります。

バイト列オブジェクトが第一引数として指定されていた場合、配列の最後の要素が NUL 終端文字となるように、バイト列オブジェクトの長さより 1 つ長いバッファを作成します。バイト列の長さを使うべきではない場合は、第二引数として整数を渡して、配列の長さを指定することができます。

引数 `init`, `size` を指定して 監査イベント `ctypes.create_string_buffer` を送出します。

ctypes.**create_unicode_buffer**(*init_or_size*, *size=None*)

この関数は変更可能な Unicode 文字バッファを作成します。返されるオブジェクトは *c_wchar* の ctypes 配列です。

init_or_size は配列のサイズを指定する整数もしくは配列要素を初期化するために使われる文字列である必要があります。

第一引数として文字列が指定された場合は、バッファが文字列の長さより一要素分大きく作られます。配列の最後の要素が NUL 終端文字であるためです。文字列の長さを使うべきでない場合は、配列のサイズを指定するために整数を第二引数として渡すことができます。

引数 `init`, `size` を指定して 監査イベント `ctypes.create_unicode_buffer` を送出します。

ctypes.**DllCanUnloadNow**()

Windows 用: この関数は ctypes をつかってインプロセス COM サーバーを実装できるようにするためのフックです。_ctypes 拡張 dll がエクスポートしている DllCanUnloadNow 関数から呼び出されます。

ctypes.**DllGetClassObject**()

Windows 用: この関数は ctypes をつかってインプロセス COM サーバーを実装できるようにするためのフックです。`_ctypes` 拡張 dll がエクスポートしている DllGetClassObject 関数から呼び出されます。

ctypes.util.**find_library**(*name*)

ライブラリを検索し、パス名を返します。*name* は `lib` のような接頭辞、`.so` や `.dylib` のような接尾辞、そして、バージョンナンバーを除くライブラリ名です (これは posix のリンカーオプション `-l` で使われる書式です) 。もしライブラリが見つからなければ、`None` を返します。

厳密な機能はシステムに依存します。

ctypes.util.**find_msvcrt**()

Windows 用: Python と拡張モジュールで使われる VC ランタイムライブラリのファイル名を返します。もしライブラリ名が同定できなければ、`None` を返します。

もし、例えば拡張モジュールにより割り付けられたメモリを `free(void *)` で解放する必要があるなら、メモリ割り付けを行ったのと同じライブラリの関数を使うことが重要です。

ctypes.**FormatError**([*code*])

Windows 用: エラーコード *code* の説明文を返します。エラーコードが指定されない場合は、Windows api 関数 GetLastError を呼び出して、もっとも新しいエラーコードが使われます。

ctypes.**GetLastError**()

Windows 用: 呼び出し側のスレッド内で Windows によって設定された最新のエラーコードを返します。この関数は Windows の *GetLastError()* 関数を直接実行します。ctypes のプライベートなエラーコードのコピーを返したりはしません。

ctypes.**get_errno**()

システムの *errno* 変数の、スレッドローカルなプライベートコピーを返します。

引数無しで 監査イベント `ctypes.get_errno` を送出します。

ctypes.**get_last_error**()
Windows 用: システムの LastError 変数の、スレッドローカルなプライベートコピーを返します。

引数無しで 監査イベント ctypes.get_last_error を送出します。

ctypes.**memmove**(*dst*, *src*, *count*)
標準 C の memmove ライブラリ関数と同じものです。: *count* バイトを *src* から *dst* へコピーします。*dst* と *src* はポインタへ変換可能な整数または ctypes インスタンスでなければなりません。

ctypes.**memset**(*dst*, *c*, *count*)
標準 C の memset ライブラリ関数と同じものです。: アドレス *dst* のメモリブロックを値 *c* を *count* バイト分書き込みます。*dst* はアドレスを指定する整数または ctypes インスタンスである必要があります。

ctypes.**POINTER**(*type*)
このファクトリ関数は新しい ctypes ポインタ型を作成して返します。ポインタ型はキャッシュされ、内部で再利用されます。したがって、この関数を繰り返し呼び出してもコストは小さいです。*type* は ctypes 型でなければなりません。

ctypes.**pointer**(*obj*)
この関数は *obj* を指す新しいポインタインスタンスを作成します。戻り値は POINTER(type(obj)) 型のオブジェクトです。

注意: 外部関数呼び出しへオブジェクトへのポインタを渡したいだけなら、はるかに高速な byref(obj) を使うべきです。

ctypes.**resize**(*obj*, *size*)
この関数は *obj* の内部メモリバッファのサイズを変更します。*obj* は ctypes 型のインスタンスでなければなりません。バッファを sizeof(type(obj)) で与えられるオブジェクト型の本来のサイズより小さくすることはできませんが、バッファを拡大することはできます。

ctypes.**set_errno**(*value*)
システム変数 *errno* の、呼び出し元スレッドでの ctypes のプライベートコピーの現在値を *value* に設定し、前の値を返します。

引数 errno を指定して 監査イベント ctypes.set_errno を送出します。

ctypes.**set_last_error**(*value*)
Windows 用: システム変数 LastError の、呼び出し元スレッドでの ctypes のプライベートコピーの現在値を *value* に設定し、前の値を返します。

引数 error を指定して 監査イベント ctypes.set_last_error を送出します。

ctypes.**sizeof**(*obj_or_type*)
ctypes の型やインスタンスのメモリバッファのサイズをバイト数で返します。C の sizeof 演算子と同様の動きをします。

ctypes.**string_at**(*address*, *size=-1*)
この関数はメモリアドレス *address* から始まる C 文字列を返します。size が指定された場合はサイズとして使われます。指定されなければ、文字列がゼロ終端されていると仮定します。

引数 address, size を指定して 監査イベント ctypes.string_at を送出します。

ctypes.**WinError**(*code=None*, *descr=None*)

Windows 用: この関数はおそらく ctypes の中で最悪の名前でしょう。これは OSError のインスタンスを作成します。*code* が指定されていなかった場合、エラーコードを判別するために GetLastError が呼び出されます。*descr* が指定されていなかった場合、エラーの説明文を得るために *FormatError()* が呼び出されます。

バージョン 3.3 で変更: 以前は *WindowsError* インスタンスが作成されていました。

ctypes.**wstring_at**(*address*, *size=-1*)

この関数は文字列としてメモリアドレス *address* から始まるワイドキャラクタ文字列を返します。*size* が指定されたならば、文字列の文字数として使われます。指定されなければ、文字列がゼロ終端されていると仮定します。

引数 address, size を指定して 監査イベント ctypes.wstring_at を送出します。

データ型

class ctypes.**_CData**

この非公開クラスはすべての ctypes データ型の共通のベースクラスです。他のことはさておき、すべての ctypes 型インスタンスは C 互換データを保持するメモリブロックを内部に持ちます。このメモリブロックのアドレスは *addressof()* ヘルパー関数が返します。別のインスタンス変数が *_objects* として公開されます。これはメモリブロックがポインタを含む場合に存続し続ける必要のある他の Python オブジェクトを含んでいます。

ctypes データ型の共通メソッド、すべてのクラスメソッドが存在します (正確には、メタクラス のメソッドです):

from_buffer(*source*[, *offset*])

このメソッドは *source* オブジェクトのバッファを共有する ctypes のインスタンスを返します。*source* オブジェクトは書き込み可能バッファインターフェースをサポートしている必要があります。オプションの *offset* 引数では *source* バッファのオフセットをバイト単位で指定します。デフォルトではゼロです。もし *source* バッファが十分に大きくなければ、*ValueError* が送出されます。

引数 pointer, size, offset を指定して 監査イベント ctypes.cdata/buffer を送出します。

from_buffer_copy(*source*[, *offset*])

このメソッドは *source* オブジェクトの読み出し可能バッファをコピーすることで、ctypes のインスタンスを生成します。オプションの *offset* 引数では *source* バッファのオフセットをバイト単位で指定します。デフォルトではゼロです。もし *source* バッファが十分に大きくなければ、*ValueError* が送出されます。

引数 pointer, size, offset を指定して 監査イベント ctypes.cdata/buffer を送出します。

from_address(*address*)

このメソッドは *address* で指定されたメモリを使って ctypes 型のインスタンスを返します。

address は整数でなければなりません。

引数 address を指定して **監査イベント** ctypes.cdata を送出します。

from_param(*obj*)
このメソッドは *obj* を ctypes 型に適合させます。外部関数の argtypes タプルに、その型があるとき、外部関数呼び出しで実際に使われるオブジェクトと共に呼び出されます。

すべての ctypes のデータ型は、それが型のインスタンスであれば、*obj* を返すこのクラスメソッドのデフォルトの実装を持ちます。いくつかの型は、別のオブジェクトも受け付けます。

in_dll(*library*, *name*)
このメソッドは、共有ライブラリによってエクスポートされた ctypes 型のインスタンスを返します。*name* はエクスポートされたデータの名前で、*library* はロードされた共有ライブラリです。

ctypes データ型共通のインスタンス変数:

_b_base_
ctypes 型データのインスタンスは、それ自身のメモリブロックを持たず、基底オブジェクトのメモリブロックの一部を共有することがあります。*_b_base_* 読み出し専用属性は、メモリブロックを保持する ctypes の基底オブジェクトです。

_b_needsfree_
この読み出し専用の変数は、ctypes データインスタンスが、それ自身に割り当てられたメモリブロックを持つとき true になります。それ以外の場合は false になります。

_objects
このメンバは None 、または、メモリブロックの内容が正しく保つために、生存させておかなくてはならない Python オブジェクトを持つディクショナリです。このオブジェクトはデバッグでのみ使われます。決してディクショナリの内容を変更しないで下さい。

基本データ型

class ctypes._SimpleCData
この非公開クラスは、全ての基本的な ctypes データ型の基底クラスです。これは基本的な ctypes データ型に共通の属性を持っているので、ここで触れておきます。*_SimpleCData* は *_CData* の子クラスなので、そのメソッドと属性を継承しています。ポインタでないかポインタを含まない ctypes データ型は、現在は pickle 化できます。

インスタンスは一つだけ属性を持ちます:

value
この属性は、インスタンスの実際の値を持ちます。整数型とポインタ型に対しては整数型、文字型に対しては一文字のバイト列オブジェクト、文字へのポインタに対しては Python のバイト列オブジェクトもしくは文字列となります。

value 属性が ctypes インスタンスより参照されたとき、大抵の場合はそれぞれに対し新しいオブジェクトを返します。*ctypes* はオリジナルのオブジェクトを返す実装にはなって **おらず** 新し

いオブジェクトを構築します。同じことが他の ctypes オブジェクトインスタンスに対しても言えます。

基本データ型は、外部関数呼び出しの結果として返されたときや、例えば構造体のフィールドメンバーや配列要素を取り出すときに、ネイティブの Python 型へ透過的に変換されます。言い換えると、外部関数が *c_char_p* の `restype` を持つ場合は、*c_char_p* インスタンスでは **なく** 常に Python バイト列オブジェクトを受け取ることでしょう。

基本データ型のサブクラスはこの振る舞いを継承 **しません** 。したがって、外部関数の `restype` が *c_void_p* のサブクラスならば、関数呼び出しからこのサブクラスのインスタンスを受け取ります。もちろん、`value` 属性にアクセスしてポインタの値を得ることができます。

これらが基本 ctypes データ型です:

class ctypes.c_byte

C の `signed char` データ型を表し、小整数として値を解釈します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class ctypes.c_char

C `char` データ型を表し、単一の文字として値を解釈します。コンストラクタはオプションの文字列初期化子を受け取り、その文字列の長さちょうど一文字である必要があります。

class ctypes.c_char_p

C `char *` データ型を表し、ゼロ終端文字列へのポインタでなければなりません。バイナリデータを指す可能性のある一般的なポインタに対しては `POINTER(c_char)` を使わなければなりません。コンストラクタは整数のアドレスもしくはバイト列オブジェクトを受け取ります。

class ctypes.c_double

C `double` データ型を表します。コンストラクタはオプションの浮動小数点数初期化子を受け取ります。

class ctypes.c_longdouble

C `long double` データ型を表します。コンストラクタはオプションで浮動小数点数初期化子を受け取ります。`sizeof(long double) == sizeof(double)` であるプラットフォームでは *c_double* の別名です。

class ctypes.c_float

C `float` データ型を表します。コンストラクタはオプションの浮動小数点数初期化子を受け取ります。

class ctypes.c_int

C `signed int` データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。`sizeof(int) == sizeof(long)` であるプラットフォームでは、*c_long* の別名です。

class ctypes.c_int8

C 8-bit `signed int` データ型を表します。たいていは、*c_byte* の別名です。

class ctypes.c_int16

C 16-bit `signed int` データ型を表します。たいていは、*c_short* の別名です。

class ctypes.c_int32
C 32-bit signed int データ型を表します。たいていは、*c_int* の別名です。

class ctypes.c_int64
C 64-bit signed int データ型を表します。たいていは、*c_longlong* の別名です。

class ctypes.c_long
C signed long データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class ctypes.c_longlong
C signed long long データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class ctypes.c_short
C signed short データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class ctypes.c_size_t
C size_t データ型を表します。

class ctypes.c_ssize_t
C ssize_t データ型を表します。

バージョン 3.2 で追加.

class ctypes.c_ubyte
C の unsigned char データ型を表し、小さな整数として値を解釈します。コンストラクタはオプションの整数初期化子を受け取ります; オーバーフローのチェックは行われません。

class ctypes.c_uint
C の unsigned int データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります; オーバーフローのチェックは行われません。これは、sizeof(int) == sizeof(long) であるプラットフォームでは *c_ulong* の別名です。

class ctypes.c_uint8
C 8-bit unsigned int データ型を表します。たいていは、*c_ubyte* の別名です。

class ctypes.c_uint16
C 16-bit unsigned int データ型を表します。たいていは、*c_ushort* の別名です。

class ctypes.c_uint32
C 32-bit unsigned int データ型を表します。たいていは、*c_uint* の別名です。

class ctypes.c_uint64
C 64-bit unsigned int データ型を表します。たいていは、*c_ulonglong* の別名です。

class ctypes.c_ulong
C unsigned long データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class ctypes.c_ulonglong
C unsigned long long データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class ctypes.c_ushort
C unsigned short データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class ctypes.c_void_p
C void * データ型を表します。値は整数として表されます。コンストラクタはオプションの整数初期化子を受け取ります。

class ctypes.c_wchar
C wchar_t データ型を表し、値は Unicode 文字列の単一の文字として解釈されます。コンストラクタはオプションの文字列初期化子を受け取り、その文字列の長さはちょうど一文字である必要があります。

class ctypes.c_wchar_p
C wchar_t * データ型を表し、ゼロ終端ワイド文字列へのポインタでなければなりません。コンストラクタは整数のアドレスもしくは文字列を受け取ります。

class ctypes.c_bool
C の bool データ型 (より正確には、C99 以降の _Bool) を表します。True または False の値を持ち、コンストラクタは真偽値と解釈できるオブジェクトを受け取ります。

class ctypes.HRESULT
Windows 用: HRESULT 値を表し、関数またはメソッド呼び出しに対する成功またはエラーの情報を含んでいます。

class ctypes.py_object
C PyObject * データ型を表します。引数なしでこれを呼び出すと NULL PyObject * ポインタを作成します。

ctypes.wintypes モジュールは他の Windows 固有のデータ型を提供します。例えば、HWND, WPARAM, DWORD です。MSG や RECT のような有用な構造体も定義されています。

構造化データ型

class ctypes.Union(*args, **kw)
ネイティブのバイトオーダーでの共用体のための抽象ベースクラス。

class ctypes.BigEndianStructure(*args, **kw)
ビックエンディアン バイトオーダーでの構造体のための抽象ベースクラス。

class ctypes.LittleEndianStructure(*args, **kw)
リトルエンディアン バイトオーダーでの構造体のための抽象ベースクラス。

ネイティブではないバイトオーダーを持つ構造体にポインタ型フィールドあるいはポインタ型フィールドを含む他のどんなデータ型をも入れることはできません。

class ctypes.Structure(*args, **kw)

ネイティブ のバイトオーダーでの構造体のための抽象ベースクラス。

具象構造体型と具象共用体型はこれらの型の一つをサブクラス化することで作らなければなりません。少なくとも、*_fields_* クラス変数を定義する必要があります。*ctypes* は、属性に直接アクセスしてフィールドを読み書きできるようにする **デスクリプタ** を作成するでしょう。これらは、

fields

構造体のフィールドを定義するシーケンス。要素は 2 要素タプルか 3 要素タプルでなければなりません。第一要素はフィールドの名前です。第二要素はフィールドの型を指定します。それはどんな ctypes データ型でも構いません。

c_int のような整数型のために、オプションの第三要素を与えることができます。フィールドのビット幅を定義する正の小整数である必要があります。

一つの構造体と共用体の中で、フィールド名はただ一つである必要があります。これはチェックされません。名前が繰り返しでてきたときにアクセスできるのは一つのフィールドだけです。

Structure サブクラスを定義するクラス文の **後で** 、*_fields_* クラス変数を定義することができます。これにより、次のように自身を直接または間接的に参照するデータ型を作成できるようになります:

```
class List(Structure):
    pass
List._fields_ = [("pnext", POINTER(List)),
                 ...
                ]
```

しかし、*_fields_* クラス変数はその型が最初に使われる (インスタンスが作成される、それに対して *sizeof()* が呼び出されるなど) より前に定義されていなければなりません。その後 *_fields_* クラス変数へ代入すると AttributeError が送出されます。

構造体型のサブクラスのサブクラスを定義することもでき、もしあるならサブクラスのサブクラス内で定義された *_fields_* に加えて、基底クラスのフィールドも継承します。

pack

インスタンスの構造体フィールドのアライメントを上書きできるようにするオプションの小整数。*_pack_* は *_fields_* が代入されたときすでに定義されていなければなりません。そうでなければ、何の効果もありません。

anonymous

無名 (匿名) フィールドの名前が並べあげられたオプションのシーケンス。*_fields_* が代入されたとき、*_anonymous_* がすでに定義されていなければなりません。そうでなければ、何ら影響はありません。

この変数に並べあげられたフィールドは構造体型もしくは共用体型フィールドである必要があります。構造体フィールドまたは共用体フィールドを作る必要なく、入れ子になったフィールドに直接アクセスできるようにするために、*ctypes* は構造体型の中に記述子を作成します。

型の例です (Windows):

```
class _U(Union):
    _fields_ = [("lptdesc", POINTER(TYPEDESC)),
                ("lpadesc", POINTER(ARRAYDESC)),
                ("hreftype", HREFTYPE)]

class TYPEDESC(Structure):
    _anonymous_ = ("u",)
    _fields_ = [("u", _U),
                ("vt", VARTYPE)]
```

TYPEDESC 構造体は COM データ型を表現しており、vt フィールドは共用体フィールドのどれが有効であるかを指定します。u フィールドは匿名フィールドとして定義されているため、TYPEDESC インスタンスから取り除かれてそのメンバーへ直接アクセスできます。td.lptdesc と td.u.lptdesc は同等ですが、前者がより高速です。なぜなら一時的な共用体インスタンスを作る必要がないためです。:

```
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

構造体のサブクラスのサブクラスを定義することができ、基底クラスのフィールドを継承します。サブクラス定義に別の *_fields_* 変数がある場合は、この中で指定されたフィールドは基底クラスのフィールドへ追加されます。

構造体と共用体のコンストラクタは位置引数とキーワード引数の両方を受け取ります。位置引数は *_fields_* の中に現れたのと同じ順番でメンバーフィールドを初期化するために使われます。コンストラクタのキーワード引数は属性代入として解釈され、そのため、同じ名前をもつ *_fields_* を初期化するか、*_fields_* に存在しない名前に対しては新しい属性を作ります。

配列とポインタ

class ctypes.**Array**(**args*)

配列のための抽象基底クラスです。

具象配列型を作成するための推奨される方法は、任意の *ctypes* データ型に正の整数を乗算することです。代わりに、この型のサブクラスを作成し、*_length_* と *_type_* のクラス変数を定義することもできます。配列の要素は、標準の添え字とスライスによるアクセスを使用して読み書きを行うことができます。スライスの読み込みでは、結果のオブジェクト自体は *Array* ではありません。

length

配列の要素数を指定する正の整数。範囲外の添え字を指定すると、*IndexError* が送出されます。*len()* がこの整数を返します。

type

配列内の各要素の型を指定します。

配列のサブクラスのコンストラクタは、位置引数を受け付けて、配列を順番に初期化するために使用し

ます。

class ctypes._Pointer

ポインタのためのプライベートな抽象基底クラスです。

具象ポインタ型は、ポイント先の型を持つ *POINTER()* を呼び出すことで、作成できます。これは、*pointer()* により自動的に行われます。

ポインタが配列を指す場合、その配列の要素は、標準の添え字とスライスによるアクセスを使用して読み書きが行えます。ポインタオブジェクトには、サイズがないため、*len()* 関数は *TypeError* を送出します。負の添え字は、(C と同様に) ポインタの **前** のメモリから読み込み、範囲外の添え字はおそらく (幸運な場合でも) アクセス違反によりクラッシュを起こします。

type

ポイント先の型を指定します。

contents

ポインタが指すオブジェクトを返します。この属性に割り当てると、ポインタが割り当てられたオブジェクトを指すようになります。

第

SEVENTEEN

並行実行

この章で記述されているモジュールは、コードの並行実行のサポートを提供します。ツールの適切な選択は、実行されるタスク (IO bound vs CPU bound) や推奨される開発スタイル (イベントドリブンな協調的マルチタスク vs プリエンプティブマルチタスク) に依存します。ここに概観を示します:

17.1 threading --- スレッドベースの並列処理

ソースコード: Lib/threading.py

このモジュールでは、高水準のスレッドインタフェースをより低水準 な *_thread* モジュールの上に構築しています。*queue* モジュールのドキュメントも参照してください。

バージョン 3.7 で変更: このモジュールは以前はオプションでしたが、常に利用可能なモジュールとなりました。

注釈: ここには載っていませんが、Python 2.x シリーズでこのモジュールの一部のメソッドや関数に使われていた `camelCase` 名は、まだこのモジュールでサポートされます。

CPython implementation detail: CPython は *Global Interpreter Lock* のため、ある時点で Python コードを実行できるスレッドは 1 つに限られます (ただし、いくつかのパフォーマンスが強く求められるライブラリはこの制限を克服しています)。アプリケーションにマルチコアマシンの計算能力をより良く利用させたい場合は、*multiprocessing* モジュールや *concurrent.futures.ProcessPoolExecutor* の利用をお勧めします。ただし、I/O バウンドなタスクを並行して複数走らせたい場合においては、マルチスレッドは正しい選択肢です。

このモジュールは以下の関数を定義しています:

`threading.active_count()`
: 生存中の *Thread* オブジェクトの数を返します。この数は *enumerate()* の返すリストの長さと同じです。

`threading.current_thread()`
: 関数を呼び出している処理のスレッドに対応する *Thread* オブジェクトを返します。関数を呼び出して

いる処理のスレッドが *threading* モジュールで生成したものでない場合、限定的な機能しかもたないダミースレッドオブジェクトを返します。

threading.excepthook(*args*, /)

:func:‘Thread.run‘で発生したキャッチされない例外を処理する。

(実) 引数*args*は以下の属性をもちます:

- *exc_type*: 例外の型
- *exc_value*: 例外の値、“None“の可能性がある。
- *exc_traceback*: Exception traceback, can be `None`.
- *thread*: Thread which raised the exception, can be `None`.

If *exc_type* is *SystemExit*, the exception is silently ignored. Otherwise, the exception is printed out on *sys.stderr*.

If this function raises an exception, *sys.excepthook()* is called to handle it.

threading.excepthook() can be overridden to control how uncaught exceptions raised by *Thread.run()* are handled.

Storing *exc_value* using a custom hook can create a reference cycle. It should be cleared explicitly to break the reference cycle when the exception is no longer needed.

Storing *thread* using a custom hook can resurrect it if it is set to an object which is being finalized. Avoid storing *thread* after the custom hook completes to avoid resurrecting objects.

参考:

sys.excepthook() handles uncaught exceptions.

バージョン 3.8 で追加.

threading.get_ident()

現在のスレッドの ’ スレッド ID’ を返します。非ゼロの整数です。この値は直接の意味を持っていません; 例えばスレッド特有のデータの辞書に索引をつけるためのような、マジッククッキーとして意図されています。スレッドが終了し、他のスレッドが作られたとき、スレッド ID は再利用されるかもしれません。

バージョン 3.3 で追加.

threading.get_native_id()

Return the native integral Thread ID of the current thread assigned by the kernel. This is a non-negative integer. Its value may be used to uniquely identify this particular thread system-wide (until the thread terminates, after which the value may be recycled by the OS).

利用可能な環境: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX。

バージョン 3.8 で追加.

threading.**enumerate**()
現在、生存中の *Thread* オブジェクト全てのリストを返します。リストには、デーモンスレッド (daemonic thread)、*current_thread()* の生成するダミースレッドオブジェクト、そして主スレッドが入ります。終了したスレッドとまだ開始していないスレッドは入りません。

threading.**main_thread**()
main *Thread* オブジェクトを返します。通常の条件では、メインスレッドは Python インタプリタが起動したスレッドを指します。

バージョン 3.4 で追加.

threading.**settrace**(*func*)
threading モジュールを使って開始した全てのスレッドにトレース関数を設定します。*func* は各スレッドの *run()* を呼び出す前にスレッドの *sys.settrace()* に渡されます。

threading.**setprofile**(*func*)
threading モジュールを使って開始した全てのスレッドにプロファイル関数を設定します。*func* は各スレッドの *run()* を呼び出す前にスレッドの *sys.setprofile()* に渡されます。

threading.**stack_size**([*size*])
新しいスレッドを作るときのスレッドスタックサイズを返します。オプションの *size* 引数にはこれ以降に作成するスレッドのスタックサイズを指定し、0 (プラットフォームのデフォルト値または設定されたデフォルト値) か、32,768 (32 KiB) 以上の正の整数でなければなりません。*size* が指定されない場合 0 が使われます。スレッドのスタックサイズの変更がサポートされていない場合、*RuntimeError* を送出します。不正なスタックサイズが指定された場合、*ValueError* を送出して、スタックサイズは変更されません。32 KiB は現在のインタープリタ自身のために十分であると保証された最小のスタックサイズです。いくつかのプラットフォームではスタックサイズに対して制限があることに注意してください。例えば最小のスタックサイズが 32 KiB より大きかったり、システムのメモリページサイズ の整数倍の必要があるなどです。この制限についてはプラットフォームのドキュメントを参照してください (一般的なページサイズは 4 KiB なので、プラットフォームに関する情報がない場合は 4096 の整数倍のスタックサイズを選ぶといいかもしれません)。

Availability: Windows, systems with POSIX threads.

このモジュールでは以下の定数も定義しています:

threading.**TIMEOUT_MAX**
ブロックする関数 (*Lock.acquire()*, *RLock.acquire()*, *Condition.wait()* など) の *timeout* 引数に許される最大値。これ以上の値を timeout に指定すると *OverflowError* が発生します。

バージョン 3.2 で追加.

このモジュールは多くのクラスを定義しています。それらは下記のセクションで詳しく説明されます。

このモジュールのおおまかな設計は Java のスレッドモデルに基づいています。とはいえ、Java がロックと条件変数を全てのオブジェクトの基本的な挙動にしているのに対し、Python ではこれらを別個のオブジェクトに分けています。Python の *Thread* クラスがサポートしているのは Java の Thread クラスの挙動のサブセットにすぎません; 現状では、優先度 (priority) やスレッドグループがなく、スレッドの破壊 (destroy)、中

断 (stop)、一時停止 (suspend)、復帰 (resume)、割り込み (interrupt) は行えません。Java の Thread クラスにおける静的メソッドに対応する機能が実装されている場合にはモジュールレベルの関数になっています。

以下に説明するメソッドは全て原子的 (atomic) に実行されます。

17.1.1 スレッドローカルデータ

スレッドローカルデータは、その値がスレッド固有のデータです。スレッドローカルデータを管理するには、単に *local* (あるいはそのサブクラス) のインスタンスを作成して、その属性に値を設定してください:

```
mydata = threading.local()
mydata.x = 1
```

インスタンスの値はスレッドごとに違った値になります。

class threading.local

スレッドローカルデータを表現するクラス。

詳細と例題については、_threading_local モジュールのドキュメンテーション文字列を参照してください。

17.1.2 Thread オブジェクト

Thread クラスは個別のスレッド中で実行される活動 (activity) を表現します。活動を決める方法は 2 つあり、一つは呼び出し可能オブジェクトをコンストラクタへ渡す方法、もう一つはサブクラスで *run()* メソッドをオーバライドする方法です。(コンストラクタを除く) その他のメソッドは一切サブクラスでオーバライドしてはなりません。言い換えるならば、このクラスの `__init__()` と *run()* メソッド **だけ** をオーバライドしてくださいということです。

ひとたびスレッドオブジェクトを生成すると、スレッドの *start()* メソッドを呼び出して活動を開始しなければなりません。`start()` メソッド はそれぞれのスレッドの *run()* メソッドを起動します。

スレッドの活動が始まると、スレッドは ' 生存中 (alive)' とみなされます。スレッドは、通常 *run()* メソッドが終了するまで、もしくは捕捉されない例外が送出されるまで生存中となります。*is_alive()* メソッドは、スレッドが生存中であるかどうか調べます。

スレッドは他のスレッドの *join()* メソッドを呼び出すことができます。このメソッドは、*join()* メソッドを呼ばれたスレッドが終了するまでメソッドの呼び出し元のスレッドをブロックします。

スレッドは名前を持っています。名前はコンストラクタに渡すことができ、*name* 属性を通して読み出したり変更したりできます。

:meth:‘~Thread.run‘メソッドが例外を発生させた場合、:func:‘threading.excepthook‘が呼び出され、例外を処理します。デフォルトでは、:func:‘threading.excepthook‘は:exc:‘SystemExit‘を黙殺します。

スレッドには " デーモンスレッド (daemon thread)" であるというフラグを立てられます。このフラグには、残っているスレッドがデーモンスレッドだけになった時に Python プログラム全体を終了させるという意味

があります。フラグの初期値はスレッドを生成したスレッドから継承します。フラグの値は *daemon* プロパティまたは *daemon* コンストラクタ引数を通して設定できます。

注釈: デーモンスレッドは終了時にいきなり停止されます。デーモンスレッドで使われたリソース (開いているファイル、データベースのトランザクションなど) は適切に解放されないかもしれません。きちんと (gracefully) スレッドを停止したい場合は、スレッドを非デーモンスレッドにして、*Event* のような適切なシグナル送信機構を使用してください。

スレッドには ” 主スレッド (main thread)” オブジェクトがあります。主スレッドは Python プログラムを最初に制御していたスレッドです。主スレッドはデーモンスレッドではありません。

” ダミースレッド (dummy thread)” オブジェクトを作成することができます。ダミースレッドは、” 外来スレッド (alien thread)” に対応するスレッドオブジェクトです。ダミースレッドは、C コードから直接生成されたスレッドのような、*threading* モジュールの外で開始された処理スレッドです。ダミースレッドオブジェクトには限られた機能しかなく、常に生存中、かつデーモンスレッドであるとみなされ、*join()* できません。また、外来スレッドの終了を検出するのは不可能なので、ダミースレッドは削除できません。

class threading.**Thread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, ***, *daemon=None*)

コンストラクタは常にキーワード引数を使って呼び出さなければなりません。各引数は以下の通りです:

group は None でなければなりません。将来 ThreadGroup クラスが実装されたときの拡張用に予約されている引数です。

target は *run()* メソッドによって起動される呼び出し可能オブジェクトです。デフォルトでは何も呼び出さないことを示す None になっています。

name はスレッドの名前です。デフォルトでは、*N* を小さな 10 進数として、”Thread- *N*” という形式の一意な名前を生成します。

args は *target* を呼び出すときの引数タプルです。デフォルトは () です。

kwargs は *target* を呼び出すときのキーワード引数の辞書です。デフォルトは {} です。

None でない場合、*daemon* はスレッドがデーモンかどうかを明示的に設定します。None の場合 (デフォルト)、デーモン属性は現在のスレッドから継承されます。

サブクラスでコンストラクタをオーバライドした場合、必ずスレッドが何かを始める前に基底クラスのコンストラクタ (Thread.__init__()) を呼び出しておかなくてはなりません。

バージョン 3.3 で変更: *daemon* 引数が追加されました。

start()

スレッドの活動を開始します。

このメソッドは、スレッドオブジェクトあたり一度しか呼び出してはなりません。*start()* は、オブジェクトの *run()* メソッドが個別の処理スレッド中で呼び出されるように調整します。

同じスレッドオブジェクトに対し、このメソッドを 2 回以上呼び出した場合、*RuntimeError* を送出します。

run()

スレッドの活動をもたらすメソッドです。

このメソッドはサブクラスでオーバライドできます。標準の *run()* メソッドでは、オブジェクトのコンストラクタの *target* 引数に呼び出し可能オブジェクトを指定した場合、*args* および *kwargs* の位置引数およびキーワード引数とともに呼び出します。

join(*timeout=None*)

スレッドが終了するまで待機します。このメソッドは、*join()* を呼ばれたスレッドが正常終了あるいは処理されない例外によって終了するか、オプションのタイムアウトが発生するまで、メソッドの呼び出し元のスレッドをブロックします。

timeout 引数が存在して None 以外の場合、それは操作に対するタイムアウト秒 (あるいは秒未満の端数) を表す浮動小数点数でなければなりません。*join()* は常に None を返すので、*join()* の後に *is_alive()* を呼び出してタイムアウトしたかどうかを確認しなければなりません。もしスレッドがまだ生存中であれば、*join()* はタイムアウトしています。

timeout が指定されないかまたは None であるときは、この操作はスレッドが終了するまでブロックします。

一つのスレッドに対して何度でも *join()* できます。

現在のスレッドに対して *join()* を呼び出そうとすると、デッドロックを引き起こすため *RuntimeError* が送出されます。スレッドが開始される前に *join()* を呼び出すことも同様のエラーのため、同じ例外が送出されます。

name

識別のためにのみ用いられる文字列です。名前には機能上の意味づけ (semantics) はありません。複数のスレッドに同じ名前をつけてもかまいません。名前の初期値はコンストラクタで設定されます。

getName()

setName()

name に対する古い getter/setter API; 代わりにプロパティを直接使用してください。

ident

' スレッド識別子' 、または、スレッドが開始されていなければ None です。非ゼロの整数です。*get_ident()* 関数を参照下さい。スレッド識別子は、スレッドが終了した後、新たなスレッドが生成された場合、再利用され得ます。スレッド識別子は、スレッドが終了した後でも利用できます。

native_id

The native integral thread ID of this thread. This is a non-negative integer, or None if the thread has not been started. See the *get_native_id()* function. This represents the Thread ID (TID) as assigned to the thread by the OS (kernel). Its value may be used to uniquely identify this particular thread system-wide (until the thread terminates, after which the value may be recycled by the OS).

注釈: Similar to Process IDs, Thread IDs are only valid (guaranteed unique system-wide) from the time the thread is created until the thread has been terminated.

Availability: Requires *get_native_id()* function.

バージョン 3.8 で追加.

is_alive()
スレッドが生存中かどうかを返します。

このメソッドは、*run()* メソッドが起動する直前から *run()* メソッドが終了する直後までの間 `True` を返します。モジュール関数 *enumerate()* は、全ての生存中のスレッドのリストを返します。

daemon
このスレッドがデーモンスレッドか (True) か否か (False) を示すブール値。この値は *start()* の呼び出し前に設定されなければなりません。さもなければ *RuntimeError* が送出されます。初期値は生成側のスレッドから継承されます; メインスレッドはデーモンスレッドではないので、メインスレッドで作成されたすべてのスレッドは、デフォルトで *daemon* = `False` になります。

デーモンでない生存中のスレッドが全てなくなると、Python プログラム全体が終了します。

isDaemon()
setDaemon()
daemon に対する古い getter/setter API; 代わりにプロパティを直接使用してください。

17.1.3 Lock オブジェクト

プリミティブロックとは、ロックが生じた際に特定のスレッドによって所有されない同期プリミティブです。Python では現在のところ拡張モジュール *_thread* で直接実装されている最も低水準の同期プリミティブを使えます。

プリミティブロックは 2 つの状態、”ロック” または ”アンロック” があります。ロックはアンロック状態で作成されます。ロックには基本となる二つのメソッド、*acquire()* と *release()* があります。ロックの状態がアンロックである場合、*acquire()* は状態をロックに変更して即座に処理を戻します。状態がロックの場合、*acquire()* は他のスレッドが *release()* を呼び出してロックの状態をアンロックに変更するまでブロックします。その後、*acquire()* 呼び出しは状態を再度ロックに設定してから処理を戻します。*release()* メソッドを呼び出すのはロック状態のときでなければなりません; このメソッドはロックの状態をアンロックに変更して、即座に処理を戻します。アンロックの状態のロックを解放しようとすると *RuntimeError* が送出されます。

ロックは **コンテキストマネージメントプロトコル** もサポートします。

複数のスレッドにおいて *acquire()* がアンロック状態への遷移を待っているためにブロックが起きている時に *release()* を呼び出してロックの状態をアンロックにすると、一つのスレッドだけが処理を進行できます。どのスレッドが処理を進行できるのかは定義されておらず、実装によって異なるかもしれません。

全てのメソッドはアトミックに実行されます。

class threading.Lock

プリミティブロック (primitive lock) オブジェクトを実装しているクラスです。スレッドが一度ロックを獲得すると、それ以後のロック獲得の試みはロックが解放されるまでブロックします。どのスレッドでもロックを解放できます。

Lock は実際にはファクトリ関数で、プラットフォームでサポートされる最も効率的なバージョンの具体的な Lock クラスのインスタンスを返すことに注意してください。

acquire(*blocking=True*, *timeout=-1*)

ブロックあり、またはブロックなしでロックを獲得します。

引数 *blocking* を True (デフォルト) に設定して呼び出した場合、ロックがアンロック状態になるまでブロックします。そしてそれをロック状態にしてから True を返します。

引数 *blocking* の値を False にして呼び出すとブロックしません。*blocking* を True にして呼び出した場合にブロックするような状況では、直ちに False を返します。それ以外の場合には、ロックをロック状態にして True を返します。

正の値に設定された浮動小数点の *timeout* 引数とともに起動された場合、ロックを得られなければ最大で *timeout* によって指定された秒数だけブロックします。*timeout* 引数の -1 は無制限の待機を指定します。*blocking* が false の場合に *timeout* を指定することは禁止されています。

ロックを獲得すると True を、ロックを獲得できなかったとき (例えば *timeout* が過ぎた場合) には False を返します。

バージョン 3.2 で変更: 新しい *timeout* 引数。

バージョン 3.2 で変更: Lock acquisition can now be interrupted by signals on POSIX if the underlying threading implementation supports it.

release()

ロックを解放します。これはロックを獲得したスレッドだけでなく、任意のスレッドから呼ぶことができます。

ロックの状態がロックのとき、状態をアンロックにリセットして処理を戻します。他のスレッドがロックがアンロック状態になるのを待ってブロックしている場合、ただ一つのスレッドだけが処理を継続できるようにします。

アンロック状態のロックに対して呼び出された場合、*RuntimeError* が送出されます。

戻り値はありません。

locked()

ロック状態のときに真を返します。

17.1.4 RLock オブジェクト

再入可能ロック (reentrant lock) とは、同じスレッドが複数回獲得できるような同期プリミティブです。再入可能ロックの内部では、プリミティブロックの使うロック／アンロック状態に加え、" 所有スレッド (owning thread)" と " 再帰レベル (recursion level)" という概念を用いています。ロック状態では何らかのスレッドがロックを所有しており、アンロック状態ではいかなるスレッドもロックを所有していません。

このロックの状態をロックにするには、スレッドがロックの *acquire()* メソッドを呼び出します。このメソッドはスレッドがロックを所有すると処理を戻します。ロックの状態をアンロックにするには *release()* メソッドを呼び出します。*acquire()* / *release()* からなるペアの呼び出しはネストできます; 最後に呼び出した *release()* (最も外側の呼び出しペアの *release()*) だけがロックの状態をアンロックにリセットして、*acquire()* でブロック中の別のスレッドの処理を進行させることができます。

再入可能ロックは **コンテキストマネージメントプロトコル** もサポートします。

class threading.RLock

このクラスは再入可能ロックオブジェクトを実装します。再入可能ロックはそれを獲得したスレッドによって解放されなければなりません。いったんスレッドが再入可能ロックを獲得すると、同じスレッドはブロックされずにもう一度それを獲得できます ; そのスレッドは獲得した回数だけ解放しなければいけません。

RLock は実際にはファクトリ関数で、プラットフォームでサポートされる最も効率的なバージョンの具体的な RLock クラスのインスタンスを返すことに注意してください。

acquire(*blocking=True*, *timeout=-1*)

ブロックあり、またはブロックなしでロックを獲得します。

引数なしで呼び出した場合: スレッドが既にロックを所有している場合、再帰レベルをインクリメントして即座に処理を戻します。それ以外の場合、他のスレッドがロックを所有していれば、そのロックの状態がアンロックになるまでブロックします。その後、ロックの状態がアンロックになる (いかなるスレッドもロックを所有しない状態になる) と、ロックの所有権を獲得し、再帰レベルを 1 にセットして処理を戻します。ロックの状態がアンロックになるのを待っているスレッドが複数ある場合、その中の一つだけがロックの所有権を獲得できます。この場合、戻り値はありません。

引数 *blocking* の値を true にして呼び出した場合、引数なしで呼び出したときと同じことを行ない、True を返します。

引数 *blocking* の値を false にして呼び出すとブロックしません。引数なしで呼び出した場合にブロックするような状況であった場合には直ちに False を返します。それ以外の場合には、引数なしで呼び出したときと同じ処理を行い True を返します。

正の値に設定された浮動小数点の *timeout* 引数とともに起動された場合、ロックを得られなければ最大で *timeout* によって指定された秒数だけブロックします。ロックを獲得した場合 True を返し、タイムアウトが過ぎた場合は false を返します。

バージョン 3.2 で変更: 新しい *timeout* 引数。

release()

再帰レベルをデクリメントしてロックを解放します。デクリメント後に再帰レベルがゼロになった

場合、ロックの状態をアンロック (いかなるスレッドにも所有されていない状態) にリセットし、ロックの状態がアンロックになるのを待ってブロックしているスレッドがある場合にはその中のただ一つだけが処理を進行できるようにします。デクリメント後も再帰レベルがゼロでない場合、ロックの状態はロックのままで、呼び出し側のスレッドに所有されたままになります。

呼び出し側のスレッドがロックを所有しているときにのみこのメソッドを呼び出してください。ロックの状態がアンロックの時にこのメソッドを呼び出すと、*RuntimeError* が送出されます。

戻り値はありません。

17.1.5 Condition オブジェクト

条件変数 (condition variable) は、常にある種のロックに関連付けられています; このロックは明示的に渡すことも、デフォルトで生成させることもできます。複数の条件変数で同じロックを共有しなければならない場合には、引渡しによる関連付けが便利です。ロックは条件オブジェクトの一部です: それを別々に扱う必要はありません。

条件変数は **コンテキスト管理プロトコル** に従います: `with` 文を使って囲まれたブロックの間だけ関連付けられたロックを獲得することができます。*acquire()* メソッドと *release()* メソッドは、さらに関連付けられたロックの対応するメソッドを呼び出します。

他のメソッドは、関連付けられたロックを保持した状態で呼び出さなければなりません。*wait()* メソッドはロックを解放します。そして別のスレッドが *notify()* または *notify_all()* を呼ぶことによってスレッドを起こすまでブロックします。一旦起こされたなら、*wait()* は再びロックを得て戻ります。タイムアウトを指定することも可能です。

notify() メソッドは条件変数待ちのスレッドを 1 つ起こします。*notify_all()* メソッドは条件変数待ちの全てのスレッドを起こします。

注意: *notify()* と *notify_all()* はロックを解放しません; 従って、スレッドが起こされたとき、*wait()* の呼び出しは即座に処理を戻すわけではなく、*notify()* または *notify_all()* を呼び出したスレッドが最終的にロックの所有権を放棄したときに初めて処理を返すのです。

条件変数を使う典型的なプログラミングスタイルでは、何らかの共有された状態変数へのアクセスを同期させるためにロックを使います; 状態変数が特定の状態に変化したことを知りたいスレッドは、自分の望む状態になるまで繰り返し *wait()* を呼び出します。その一方で、状態変更を行うスレッドは、前者のスレッドが待ち望んでいる状態であるかもしれないような状態へ変更を行ったときに *notify()* や *notify_all()* を呼び出します。例えば、以下のコードは無制限のバッファ容量のときの一般的な生産者-消費者問題です:

```
# Consume one item
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# Produce one item
with cv:
```

(次のページに続く)

(前のページからの続き)

```
    make_an_item_available()
    cv.notify()
```

アプリケーションの条件をチェックする while ループは必須です。なぜなら、*wait()* が任意の長時間の後で返り、*notify()* 呼び出しを促した条件がもはや真でないことがありえるからです。これはマルチスレッドプログラミングに固有です。条件チェックを自動化するために *wait_for()* メソッドを使うことができ、それはタイムアウトの計算を簡略化します:

```
# Consume an item
with cv:
    cv.wait_for(an_item_is_available)
    get_an_available_item()
```

notify() と *notify_all()* のどちらを使うかは、その状態の変化に興味を持っている待ちスレッドが一つだけなのか、あるいは複数なのかで考えます。例えば、典型的な生産者-消費者問題では、バッファに 1 つの要素を加えた場合には消費者スレッドを 1 つしか起こさなくてかまいません。

class threading.Condition(*lock=None*)

このクラスは条件変数 (condition variable) オブジェクトを実装します。条件変数を使うと、1 つ以上のスレッドを別のスレッドの通知があるまで待機させておけます。

lock に None でない値を指定した場合、その値は *Lock* または *RLock* オブジェクトでなければなりません。この場合、*lock* は根底にあるロックオブジェクトとして使われます。それ以外の場合には、*RLock* オブジェクトを新しく作成して使います。

バージョン 3.3 で変更: ファクトリ関数からクラスに変更されました。

acquire(**args*)

根底にあるロックを獲得します。このメソッドは根底にあるロックの対応するメソッドを呼び出します。そのメソッドの戻り値を返します。

release()

根底にあるロックを解放します。このメソッドは根底にあるロックの対応するメソッドを呼び出します。戻り値はありません。

wait(*timeout=None*)

通知 (notify) を受けるか、タイムアウトするまで待機します。呼び出し側のスレッドがロックを獲得していないときにこのメソッドを呼び出すと *RuntimeError* が送出されます。

このメソッドは根底にあるロックを解放し、他のスレッドが同じ条件変数に対して *notify()* または *notify_all()* を呼び出して現在のスレッドを起こすか、オプションのタイムアウトが発生するまでブロックします。一度スレッドが起こされると、再度ロックを獲得して処理を戻します。

timeout 引数を指定して、None 以外の値にする場合、タイムアウトを秒 (または端数秒) を表す浮動小数点数でなければなりません。

根底にあるロックが *RLock* である場合、*release()* メソッドではロックは解放されません。というのも、ロックが再帰的に複数回獲得されている場合には、*release()* によって実際にアンロッ

クが行われないかもしれないからです。その代わり、ロックが再帰的に複数回獲得されていても確実にアンロックを行える *RLock* クラスの内部インタフェースを使います。その後ロックを再獲得する時に、もう一つの内部インタフェースを使ってロックの再帰レベルを復帰します。

与えられた *timeout* が過ぎていなければ返り値は `True` です。タイムアウトした場合には `False` が返ります。

バージョン 3.2 で変更: 以前は、このメソッドは常に `None` を返していました。

wait_for(*predicate*, *timeout=None*)

条件が真と判定されるまで待ちます。*predicate* は呼び出し可能オブジェクトでなければならず、その結果はブール値として解釈されます。最大の待ち時間を指定する *timeout* を与えることができます。

このユーティリティメソッドは、述語が満たされるかタイムアウトが発生するまで *wait()* を繰り返し呼び出す場合があります。戻り値は述語の最後の戻り値で、もしメソッドがタイムアウトすれば、`False` と評価されます。

タイムアウト機能を無視すれば、このメソッドの呼び出しは以下のように書くのとほぼ等価です:

```
while not predicate():
    cv.wait()
```

したがって、*wait()* と同じルールが適用されます: 呼び出された時にロックを保持していなければならず、戻るときにロックが再度獲得されます。述語はロックを保持した状態で評価されます。

バージョン 3.2 で追加.

notify(*n=1*)

デフォルトで、この条件変数を待っている 1 つのスレッドを起こします。呼び出し側のスレッドがロックを獲得していないときにこのメソッドを呼び出すと *RuntimeError* が送出されます。

何らかの待機中スレッドがある場合、そのうち *n* スレッドを起こします。待機中のスレッドがなければ何もしません。

現在の実装では、少なくとも *n* スレッドが待機中であれば、ちょうど *n* スレッドを起こします。とはいえ、この挙動に依存するのは安全ではありません。将来、実装の最適化によって、複数のスレッドを起こすようになるかもしれないからです。

注意: 起こされたスレッドは実際にロックを再獲得できるまで *wait()* 呼び出しから戻りません。*notify()* はロックを解放しないので、*notify()* 呼び出し側は明示的にロックを解放しなければなりません。

notify_all()

この条件を待っているすべてのスレッドを起こします。このメソッドは *notify()* のように動作しますが、1 つではなくすべての待ちスレッドを起こします。呼び出し側のスレッドがロックを獲得していない場合、*RuntimeError* が送出されます。

17.1.6 Semaphore オブジェクト

セマフォ (semaphore) は、計算機科学史上最も古い同期プリミティブの一つ で、草創期のオランダ計算機科学者 Edsger W. Dijkstra によって発明されました (彼は *acquire()* と *release()* の代わりに `P()` と `V()` を使いました)。

セマフォは *acquire()* でデクリメントされ *release()* でインクリメントされるような内部カウンタを管理します。カウンタは決してゼロより小さくはなりません; *acquire()* は、カウンタがゼロになっている場合、他のスレッドが *release()* を呼び出すまでブロックします。

セマフォは **コンテキストマネージメントプロトコル** もサポートします。

class threading.Semaphore(*value=1*)

このクラスはセマフォ (semaphore) オブジェクトを実装します。セマフォは、*release()* を呼び出した数から *acquire()* を呼び出した数を引き、初期値を足した値を表す極小のカウンタを管理します。*acquire()* メソッドは、カウンタの値を負にせずに処理を戻せるまで必要ならば処理をブロックします。*value* を指定しない場合、デフォルトの値は 1 になります。

オプションの引数には、内部カウンタの初期値を指定します。デフォルトは `1` です。与えられた *value* が 0 より小さい場合、*ValueError* が送出されます。

バージョン 3.3 で変更: ファクトリ関数からクラスに変更されました。

acquire(*blocking=True*, *timeout=None*)

セマフォを獲得します。

When invoked without arguments:

- If the internal counter is larger than zero on entry, decrement it by one and return `True` immediately.
- If the internal counter is zero on entry, block until awoken by a call to *release()*. Once awoken (and the counter is greater than 0), decrement the counter by 1 and return `True`. Exactly one thread will be awoken by each call to *release()*. The order in which threads are awoken should not be relied on.

blocking を false にして呼び出すとブロックしません。引数なしで呼び出した場合にブロックするような状況であった場合には直ちに `False` を返します。それ以外の場合には、引数なしで呼び出したときと同じ処理を行い `True` を返します。

`None` 以外の *timeout* で起動された場合、最大で *timeout* 秒ブロックします。acquire が その間隔の間で完了しなかった場合は `False` が返ります。そうでなければ `True` が返ります。

バージョン 3.2 で変更: 新しい *timeout* 引数。

release()

内部カウンタを 1 インクリメントして、セマフォを解放します。`release()` 処理に入ったときにカウンタがゼロであり、カウンタの値がゼロより大きくなるのを待っている別のスレッドがあった場合、そのスレッドを起こします。

class threading.BoundedSemaphore(*value=1*)

有限セマフォ (bounded semaphore) オブジェクトを実装しているクラスです。有限セマフォは、現在の値が初期値を超過しないようチェックを行います。超過を起こした場合、*ValueError* を送出します。たいていの場合、セマフォは限られた容量のリソースを保護するために使われるものです。従って、あまりにも頻繁なセマフォの解放はバグが生じているしるしです。*value* を指定しない場合、デフォルトの値は 1 になります。

バージョン 3.3 で変更: ファクトリ関数からクラスに変更されました。

Semaphore の例

セマフォはしばしば、容量に限りのある資源、例えばデータベースサーバなどを保護するために使われます。リソースが固定の状況では、常に有限セマフォを使わなければなりません。主スレッドは、作業スレッドを立ち上げる前にセマフォを初期化します:

```
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
```

作業スレッドは、ひとたび立ち上がると、サーバへ接続する必要が生じたときにセマフォの `acquire()` および `release()` メソッドを呼び出します:

```
with pool_sema:
    conn = connectdb()
    try:
        # ... use connection ...
    finally:
        conn.close()
```

有限セマフォを使うと、セマフォを獲得回数以上に解放してしまうというプログラム上の間違いを見逃しにくくします。

17.1.7 Event オブジェクト

イベントは、あるスレッドがイベントを発信し、他のスレッドはそれを待つという、スレッド間で通信を行うための最も単純なメカニズムの一つです。

イベントオブジェクトは内部フラグを管理します。このフラグは *set()* メソッドで値を true に、*clear()* メソッドで値を false にリセットします。*wait()* メソッドはフラグが true になるまでブロックします。

class threading.Event

イベントオブジェクトを実装しているクラスです。イベントは *set()* メソッドを使うと *True* に、*clear()* メソッドを使うと *False* にセットされるようなフラグを管理します。*wait()* メソッドは、全てのフラグが true になるまでブロックするようになっています。フラグの初期値は false です。

バージョン 3.3 で変更: ファクトリ関数からクラスに変更されました。

is_set()
内部フラグが真のとき True を返します。

set()
内部フラグの値を true にセットします。フラグの値が true になるのを待っている全てのスレッドを起こします。一旦フラグが true になると、スレッドが *wait()* を呼び出しても全くブロックしなくなります。

clear()
内部フラグの値を false にリセットします。以降は、*set()* を呼び出して再び内部フラグの値を true にセットするまで、*wait()* を呼び出したスレッドはブロックするようになります。

wait(*timeout=None*)
内部フラグの値が true になるまでブロックします。wait() 処理に入った時点で内部フラグの値が true であれば、直ちに処理を戻します。そうでない場合、他のスレッドが *set()* を呼び出してフラグの値を true にセットするか、オプションのタイムアウトが発生するまでブロックします。

timeout 引数を指定して、None 以外の値にする場合、タイムアウトを秒 (または端数秒) を表す浮動小数点数でなければなりません。

このメソッドは、wait 呼び出しの前あるいは wait が開始した後に、内部フラグが true にセットされている場合に限り True を返します。したがって、タイムアウトが与えられて操作がタイムアウトした場合を除き常に True を返します。

バージョン 3.1 で変更: 以前は、このメソッドは常に None を返していました。

17.1.8 Timer オブジェクト

このクラスは、一定時間経過後に実行される活動、すなわちタイマ活動を表現します。*Timer* は *Thread* のサブクラスであり、自作のスレッドを構築した一例でもあります。

タイマは start() メソッドを呼び出すとスレッドとして作動し始めします。(活動を開始する前に) *cancel()* メソッドを呼び出すと、タイマを停止できます。タイマが活動を実行するまでの待ち時間は、ユーザが指定した待ち時間と必ずしも厳密には一致しません。

例えば:

```
def hello():
    print("hello, world")

t = Timer(30.0, hello)
t.start()  # after 30 seconds, "hello, world" will be printed
```

class threading.Timer(*interval*, *function*, *args=None*, *kwargs=None*)
interval 秒後に引数 *args* キーワード引数 *kwargs* で *function* を実行するようなタイマを生成します。*args* が "None" (デフォルト) なら空のリストが使用されます。*kwargs* が None (デフォルト) なら空の辞書が使用されます。

バージョン 3.3 で変更: ファクトリ関数からクラスに変更されました。

cancel()
タイマをストップして、その動作の実行をキャンセルします。このメソッドはタイマがまだ活動待ち状態にある場合にのみ動作します。

17.1.9 バリアオブジェクト

バージョン 3.2 で追加.

このクラスは、互いを待つ必要のある固定の数のスレッドで使用するための単純な同期プリミティブを提供します。それぞれのスレッドは *wait()* メソッドを呼ぶことによりバリアを通ろうとしてブロックします。すべてのスレッドがそれぞれの:meth:‘~Barrier.wait‘メソッドを呼び出した時点で、すべてのスレッドが同時に解放されます。

バリアは同じ数のスレッドに対して何度でも再利用することができます。

例として、クライアントとサーバの間でスレッドを同期させる単純な方法を紹介します:

```
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
        process_server_connection(connection)

def client():
    b.wait()
    while True:
        connection = make_connection()
        process_client_connection(connection)
```

class threading.**Barrier**(*parties*, *action=None*, *timeout=None*)
parties 個のスレッドのためのバリアオブジェクトを作成します。*action* は、もし提供されるなら呼び出し可能オブジェクトで、スレッドが解放される時にそのうちの 1 つによって呼ばれます。*timeout* は、*wait()* メソッドに対して none が指定された場合のデフォルトのタイムアウト値です。

wait(*timeout=None*)
バリアを通ります。バリアに対するすべてのスレッドがこの関数を呼んだ時に、それらは同時にすべて解放されます。*timeout* が提供される場合、それはクラスコンストラクタに渡された値に優先して使用されます。

返り値は 0 から *parties* -- 1 の範囲の整数で、それぞれのスレッドに対して異なります。これは、特別な後始末 (housekeeping) を行うスレッドを選択するために使用することができます。例えば:

```
i = barrier.wait()
if i == 0:
    # Only one thread needs to print this
    print("passed the barrier")
```

action がコンストラクタに渡されていれば、スレッドのうちの 1 つが解放される前にそれを呼び出します。万一この呼び出しでエラーが発生した場合、バリアは broken な状態に陥ります。

この呼び出しがタイムアウトする場合、バリアは broken な状態に陥ります。

スレッドが待っている間にバリアが broken になるかリセットされた場合、このメソッドは *BrokenBarrierError* 例外を送出するかもしれません。

reset()

バリアをデフォルトの空の状態に戻します。そのバリアの上で待っているすべてのスレッドは *BrokenBarrierError* 例外を受け取ります。

状態が未知の他のスレッドがある場合、この関数を使用するのに何らかの外部同期を必要とするかもしれないことに注意してください。バリアが broken な場合、単にそれをそのままにして新しいものを作成する方がよいでしょう。

abort()

バリアを broken な状態にします。これによって、現在または将来の *wait()* 呼び出しが *BrokenBarrierError* とともに失敗するようになります。これを使うと、例えばスレッドが異常終了する必要がある場合にアプリケーションがデッドロックするのを避けることができます。

スレッドのうちの 1 つが返ってこないことに対して自動的に保護するように、単純に常識的な *timeout* 値でバリアを作成することは望ましいかもしれません。

parties

バリアを通るために要求されるスレッドの数。

n_waiting

現在バリアの中で待っているスレッドの数。

broken

バリアが broken な状態である場合に `True` となるブール値。

exception threading.BrokenBarrierError

Barrier オブジェクトがリセットされるか broken な場合に、この例外 (*RuntimeError* のサブクラス) が送出されます。

17.1.10 with 文でのロック・条件変数・セマフォの使い方

このモジュールのオブジェクトのうち `acquire()` メソッドと `release()` メソッドを備えているものは全て `with` 文のコンテキストマネージャ として使うことができます。`with` 文のブロックに入るときに `acquire()` メソッドが 呼び出され、ブロック脱出時には `release()` メソッドが呼ばれます。したがって、次のコード:

```
with some_lock:
    # do something...
```

は、以下と同じです

```
some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()
```

現在のところ、*Lock* 、*RLock* 、*Condition* 、*Semaphore* 、*BoundedSemaphore* を with 文のコンテキストマネージャとして使うことができます。

17.2 multiprocessing --- プロセスベースの並列処理

ソースコード: Lib/multiprocessing/

17.2.1 はじめに

multiprocessing は、*threading* と似た API で複数のプロセスの生成をサポートするパッケージです。*multiprocessing* パッケージは、ローカルとリモート両方の並行処理を提供します。また、このパッケージはスレッドの代わりにサブプロセスを使用することにより、**グローバルインタープリタロック** の問題を避ける工夫が行われています。このような特徴があるため *multiprocessing* モジュールを使うことで、マルチプロセッサーマシンの性能を最大限に活用することができるでしょう。なお、このモジュールは Unix と Windows の両方で動作します。

multiprocessing モジュールでは、*threading* モジュールには似たものが存在しない API も導入されています。その最たるものが *Pool* オブジェクトです。これは複数の入力データに対して、サブプロセス群に入力データを分配 (データ並列) して関数を並列実行するのに便利な手段を提供します。以下の例では、モジュール内で関数を定義して、子プロセスがそのモジュールを正常にインポートできるようにする一般的な方法を示します。*Pool* を用いたデータ並列の基礎的な例は次の通りです:

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

標準出力に以下が出力されます:

```
[1, 4, 9]
```

Process クラス

multiprocessing モジュールでは、プロセスは以下の手順によって生成されます。はじめに *Process* のオブジェクトを作成し、続いて *start()* メソッドを呼び出します。この *Process* クラスは *threading.Thread* クラスと同様の API を持っています。まずは、簡単な例をもとにマルチプロセスを使用したプログラムについてみていきましょう

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

実行された個々のプロセス ID を表示するために拡張したサンプルコードを以下に示します:

```
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

なぜ `if __name__ == '__main__'` という記述が必要かは **プログラミングガイドライン** を参照してください。

コンテキストと開始方式

プラットフォームにもよりますが、*multiprocessing* はプロセスを開始するために 3 つの方法をサポートしています。それら **開始方式** は以下のとおりです

spawn 親プロセスは新たに python インタープリタープロセスを開始します。子プロセスはプロセスオブジェクトの *run()* メソッドの実行に必要なリソースのみ継承します。特に、親プロセスからの不要なファイル記述子とハンドルは継承されません。この方式を使用したプロセスの開始は *fork* や *forkserver* に比べ遅くなります。

Unix と Windows で利用可能。Windows と macOS でのデフォルト。

fork 親プロセスは *os.fork()* を使用して Python インタープリターをフォークします。子プロセスはそれが開始されるとき、事実上親プロセスと同一になります。親プロセスのリソースはすべて子プロセスに継承されます。マルチスレッドプロセスのフォークは安全性に問題があることに注意してください。

Unix でのみ利用可能。Unix でのデフォルト。

forkserver プログラムを開始するとき *forkserver* 方式を選択した場合、サーバープロセスが開始されます。それ以降、新しいプロセスが必要になったときはいつでも、親プロセスはサーバーに接続し、新しいプロセスのフォークを要求します。フォークサーバープロセスはシングルスレッドなので *os.fork()* の使用に関しても安全です。不要なリソースは継承されません。

Unix パイプを経由したファイル記述子の受け渡しをサポートする Unix で利用可能。

バージョン 3.8 で変更: macOS では、*spawn* 開始方式がデフォルトになりました。*fork* 開始方法は、サブプロセスのクラッシュを引き起こす可能性があるため、安全ではありません。bpo-33725 を参照。

バージョン 3.4 で変更: すべての Unix プラットフォームで *spawn* が、一部のプラットフォームで *forkserver* が追加されました。Windows では親プロセスの継承可能な全ハンドルが子プロセスに継承されることがなくなりました。

Unix で開始方式に *spawn* あるいは *forkserver* を使用した場合は、プログラムのプロセスによって作成されたリンクされていない名前付きシステムリソース (名前付きセマフォや、*SharedMemory* オブジェクト) を追跡する **リソーストラッカー** プロセスも開始されます。全プロセスが終了したときに、リソーストラッカーは残っているすべての追跡していたオブジェクトのリンクを解除します。通常そういったことはないのですが、プロセスがシグナルによって kill されたときに ”漏れた” リソースが発生する場合があります。(この場合、名前付きセマフォと共有メモリセグメントは、システムが再起動されるまでリンク解除されません。名前付きセマフォの個数はシステムによって制限されており、共有メモリセグメントはメインメモリを占有するため、これらは問題になる可能性があります。)

開始方式はメインモジュールの `if __name__ == '__main__'` 節内で、関数 *set_start_method()* によって指定します。以下に例を示します:

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    mp.set_start_method('spawn')
    q = mp.Queue()
    p = mp.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

関数 *set_start_method()* はプログラム内で複数回使用してはいけません。

もうひとつの方法として、*get_context()* を使用してコンテキストオブジェクトを取得することができます。

コンテキストオブジェクトは multiprocessing モジュールと同じ API を持ち、同じプログラム内で複数の開始方式を使用できます。

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    ctx = mp.get_context('spawn')
    q = ctx.Queue()
    p = ctx.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

あるコンテキストに関連したオブジェクトは、異なるコンテキストのプロセスとは互換性がない場合があることに注意してください。特に、*fork* コンテキストを使用して作成されたロックは、*spawn* あるいは *forkserver* を使用して開始されたプロセスに渡すことはできません。

特定の開始方式の使用を要求するライブラリは *get_context()* を使用してライブラリ利用者の選択を阻害しないようにするべきです。

警告: Unix において、'spawn' あるいは 'forkserver' で開始された場合、"frozen" な実行可能形式 (**PyInstaller** や **cx_Freeze** で作成されたバイナリなど) は使用できません。'fork' で開始した場合は動作します。

プロセス間でのオブジェクト交換

multiprocessing モジュールでは、プロセス間通信の手段が 2 つ用意されています。それぞれ以下に詳細を示します:

キュー (Queue)

Queue クラスは *queue.Queue* クラスとほとんど同じように使うことができます。以下に例を示します:

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

キューはスレッドセーフであり、プロセスセーフです。

パイプ (Pipe)

Pipe() 関数はパイプで繋がれたコネクションオブジェクトのペアを返します。デフォルトでは双方向性パイプを返します。以下に例を示します:

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # prints "[42, None, 'hello']"
    p.join()
```

パイプのそれぞれの端を表す 2 つのコネクションオブジェクトが *Pipe()* 関数から返されます。各コネクションオブジェクトには、send()、recv()、その他のメソッドがあります。2 つのプロセス (またはスレッド) がパイプの **同じ** 端で同時に読み込みや書き込みを行うと、パイプ内のデータが破損してしまうかもしれないことに注意してください。もちろん、各プロセスがパイプの別々の端を同時に使用するならば、データが破壊される危険性はありません。

プロセス間の同期

multiprocessing は *threading* モジュールと等価な同期プリミティブを備えています。以下の例では、ロックを使用して、一度に 1 つのプロセスしか標準出力に書き込まないようにしています:

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
    finally:
        l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

ロックを使用しないで標準出力に書き込んだ場合は、各プロセスからの出力がごちゃまぜになってしまいます。

プロセス間での状態の共有

これまでの話の流れで触れたとおり、並行プログラミングを行うときには、できるかぎり状態を共有しないのが定石です。複数のプロセスを使用するときは特にそうでしょう。

しかし、どうしてもプロセス間のデータ共有が必要な場合のために *multiprocessing* モジュールには 2 つの方法が用意されています。

共有メモリ (Shared memory)

データを共有メモリ上に保持するために *Value* クラス、もしくは *Array* クラスを使用することができます。以下のサンプルコードを使って、この機能についてみていきましょう

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

このサンプルコードを実行すると以下のように表示されます

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

num と arr を生成するときに使用されている、引数 'd' と 'i' は *array* モジュールにより使用される種別の型コードです。ここで使用されている 'd' は倍精度浮動小数、'i' は符号付整数を表します。これらの共有オブジェクトは、プロセスセーフでありスレッドセーフです。

共有メモリを使用して、さらに柔軟なプログラミングを行うには *multiprocessing.sharedctypes* モジュールを使用します。このモジュールは共有メモリから割り当てられた任意の ctypes オブジェクトの生成をサポートします。

サーバープロセス (Server process)

Manager() 関数により生成されたマネージャーオブジェクトはサーバープロセスを管理します。マネージャーオブジェクトは Python のオブジェクトを保持して、他のプロセスがプロキシ経由でその Python オブジェクトを操作することができます。

Manager() 関数が返すマネージャは *list*, *dict*, *Namespace*, *Lock*, *RLock*, *Semaphore*, *BoundedSemaphore*, *Condition*, *Event*, *Barrier*, *Queue*, *Value*, *Array* をサポートします。以

下にサンプルコードを示します。

```
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))

        p = Process(target=f, args=(d, l))
        p.start()
        p.join()

        print(d)
        print(l)
```

このサンプルコードを実行すると以下のように表示されます

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

サーバープロセスのマネージャーオブジェクトは共有メモリのオブジェクトよりも柔軟であるといえます。それは、どのような型のオブジェクトでも使えるからです。また、1 つのマネージャーオブジェクトはネットワーク経由で他のコンピューター上のプロセスによって共有することもできます。しかし、共有メモリより動作が遅いという欠点があります。

ワーカープロセスのプールを使用

Pool クラスは、ワーカープロセスをプールする機能を備えています。このクラスには、異なる方法でワーカープロセスへタスクを割り当てるいくつかのメソッドがあります。

例えば:

```
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x

if __name__ == '__main__':
    # start 4 worker processes
    with Pool(processes=4) as pool:
```

(次のページに続く)

(前のページからの続き)

```
        # print "[0, 1, 4,..., 81]"
        print(pool.map(f, range(10)))

        # print same numbers in arbitrary order
        for i in pool.imap_unordered(f, range(10)):
            print(i)

        # evaluate "f(20)" asynchronously
        res = pool.apply_async(f, (20,))      # runs in *only* one process
        print(res.get(timeout=1))             # prints "400"

        # evaluate "os.getpid()" asynchronously
        res = pool.apply_async(os.getpid, ()) # runs in *only* one process
        print(res.get(timeout=1))             # prints the PID of that process

        # launching multiple evaluations asynchronously *may* use more processes
        multiple_results = [pool.apply_async(os.getpid, ()) for i in range(4)]
        print([res.get(timeout=1) for res in multiple_results])

        # make a single worker sleep for 10 secs
        res = pool.apply_async(time.sleep, (10,))
        try:
            print(res.get(timeout=1))
        except TimeoutError:
            print("We lacked patience and got a multiprocessing.TimeoutError")

        print("For the moment, the pool remains available for more work")

    # exiting the 'with'-block has stopped the pool
    print("Now the pool is closed and no longer available")
```

プールオブジェクトのメソッドは、そのプールを作成したプロセスのみが呼び出すべきです。

注釈: このパッケージに含まれる機能を使用するためには、子プロセスから `__main__` モジュールをインポートできる必要があります。このことについては **プログラミングガイドライン** で触れていますが、ここであらためて強調しておきます。なぜかというと、いくつかのサンプルコード、例えば *multiprocessing.pool.Pool* のサンプルはインタラクティブシェル上では動作しないからです。以下に例を示します:

```
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> with p:
...   p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
```

(次のページに続く)

(前のページからの続き)

```
Traceback (most recent call last):
Traceback (most recent call last):
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
```

(もしこのコードを試すなら、実際には 3 つの完全なトレースバックがばらばらの順番で出力されますし、親プロセスを何らかの方法で止める必要があります。)

17.2.2 リファレンス

multiprocessing パッケージは *threading* モジュールの API とほとんど同じです。

Process クラスと例外

class multiprocessing.**Process**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, ***, *daemon=None*)

Process オブジェクトは各プロセスの処理を表します。*Process* クラスは *threading.Thread* クラスのすべてのメソッドと同じインタフェースを提供します。

コンストラクターは必ずキーワード引数で呼び出すべきです。引数 *group* には必ず None を渡してください。この引数は *threading.Thread* クラスとの互換性のためだけに残されています。引数 *target* には、*run()* メソッドから呼び出される callable オブジェクトを渡します。この引数はデフォルトで None となっており、何も呼び出されません。引数 *name* にはプロセス名を渡します (詳細は *name* を見てください)。*args* は対象の呼び出しに対する引数のタプルを渡します。*kwargs* は対象の呼び出しに対するキーワード引数の辞書を渡します。もし提供されれば、キーワード専用の *daemon* 引数はプロセスの *daemon* フラグを True または False にセットします。None の場合 (デフォルト)、このフラグは作成するプロセスから継承されます。

デフォルトでは、*target* には引数が渡されないようになっています。

サブクラスがコンストラクターをオーバーライドする場合は、そのプロセスに対する処理を行う前に基底クラスのコンストラクター (Process.__init__()) を実行しなければなりません。

バージョン 3.3 で変更: *daemon* 引数が追加されました。

run()

プロセスが実行する処理を表すメソッドです。

このメソッドはサブクラスでオーバーライドすることができます。標準の *run()* メソッドは、コンストラクターの target 引数として渡された呼び出し可能オブジェクトを呼び出します。もしコンストラクターに *args* もしくは *kwargs* 引数が渡されていれば、呼び出すオブジェクトにこれらの引数を渡します。

start()

プロセスの処理を開始するためのメソッドです。

各 Process オブジェクトに対し、このメソッドが 2 回以上呼び出されてはいけません。各プロセスでオブジェクトの *run()* メソッドを呼び出す準備を行います。

join([*timeout*])
オプションの引数 *timeout* が None (デフォルト) の場合、*join()* メソッドが呼ばれたプロセスは処理が終了するまでブロックします。*timeout* が正の数である場合、最大 *timeout* 秒ブロックします。プロセスが終了あるいはタイムアウトした場合、メソッドは None を返すことに注意してください。プロセスの *exitcode* を確認し終了したかどうかを判断してください。

1 つのプロセスは何回も join されることができます。

プロセスは自分自身を join することはできません。それはデッドロックを引き起こすことがあるからです。プロセスが start される前に join しようとするとエラーが発生します。

name
プロセスの名前。名前は識別のためだけに使用される文字列です。それ自体には特別な意味はありません。複数のプロセスに同じ名前が与えられても構いません。

最初の名前はコンストラクターによってセットされます。コンストラクターに明示的な名前が渡されない場合、'Process-N_1:N_2:...:N_k' 形式の名前が構築されます。ここでそれぞれの N_k はその親の N 番目の子供です。

is_alive()
プロセスが実行中かを判別します。

おおまかに言って、プロセスオブジェクトは *start()* メソッドを呼び出してから子プロセス終了までの期間が実行中となります。

daemon
デーモンプロセスであるかのフラグであり、ブール値です。この属性は *start()* が呼び出される前に設定されている必要があります。

初期値は作成するプロセスから継承します。

あるプロセスが終了するとき、そのプロセスはその子プロセスであるデーモンプロセスすべてを終了させようとします。

デーモンプロセスは子プロセスを作成できないことに注意してください。もし作成できてしまうと、そのデーモンプロセスの親プロセスが終了したときにデーモンプロセスの子プロセスが孤児になってしまう場合があるからです。さらに言えば、デーモンプロセスは Unix デーモンやサービスでは **なく** 通常のプロセスであり、非デーモンプロセスが終了すると終了されます (そして join されません)。

threading.Thread クラスの API に加えて *Process* クラスのオブジェクトには以下の属性およびメソッドがあります:

pid
プロセス ID を返します。プロセスの生成前は None が設定されています。

exitcode

子プロセスの終了コードです。子プロセスがまだ終了していない場合は None が返されます。負の値 *-N* は子プロセスがシグナル *N* で終了したことを表します。

authkey

プロセスの認証キーです (バイト文字列です)。

multiprocessing モジュールがメインプロセスにより初期化される場合には、*os.urandom()* 関数を使用してランダムな値が設定されます。

Process クラスのオブジェクトの作成時にその親プロセスから認証キーを継承します。もしくは *authkey* に別のバイト文字列を設定することもできます。

詳細は **認証キー** を参照してください。

sentinel

プロセスが終了するときに "ready" となるシステムオブジェクトの数値ハンドル。

multiprocessing.connection.wait() を使用していくつかのイベントを同時に wait したい場合はこの値を使うことができます。それ以外の場合は *join()* を呼ぶ方がより単純です。

Windows においては、これは WaitForSingleObject および WaitForMultipleObjects ファミリーの API 呼び出しで使用可能な OS ハンドルです。Unix においては、これは *select* モジュールのプリミティブで使用可能なファイル記述子です。

バージョン 3.3 で追加.

terminate()

プロセスを終了します。Unix 環境では SIGTERM シグナルを、Windows 環境では TerminateProcess() を使用して終了させます。終了ハンドラーや finally 節などは、実行されないことに注意してください。

このメソッドにより終了するプロセスの子孫プロセスは、終了 **しません** 。そういった子孫プロセスは単純に孤児になります。

> **警告:** このメソッドの使用時に、関連付けられたプロセスがパイプやキューを使用している場合には、使用中のパイプやキューが破損して他のプロセスから使用できなくなる可能性があります。同様に、プロセスがロックやセマフォなどを取得している場合には、このプロセスが終了してしまうと他のプロセスのデッドロックの原因になるでしょう。

kill()

meth:*terminate()* と同様の動作をしますが、Unix では "SIGKILL" シグナルを使用します。

バージョン 3.7 で追加.

close()

Process オブジェクトを閉じ、関連付けられていたすべてのリソースを開放します。中のプロセスが実行中であった場合、*ValueError* を送出します。*close()* が成功した場合、class:*Process* オブジェクトの他のメソッドや属性は、ほとんどが *ValueError* を送出します。

バージョン 3.7 で追加.

プロセスオブジェクトが作成したプロセスのみが *start()*, *join()*, *is_alive()*, *terminate()* と *exitcode* のメソッドを呼び出すべきです。

以下の例では *Process* のメソッドの使い方を示しています:

```
>>> import multiprocessing, time, signal
>>> p = multiprocessing.Process(target=time.sleep, args=(1000,))
>>> print(p, p.is_alive())
<Process ... initial> False
>>> p.start()
>>> print(p, p.is_alive())
<Process ... started> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print(p, p.is_alive())
<Process ... stopped exitcode=-SIGTERM> False
>>> p.exitcode == -signal.SIGTERM
True
```

exception multiprocessing.**ProcessError**

すべての *multiprocessing* 例外の基底クラスです。

exception multiprocessing.**BufferTooShort**

この例外は Connection.recv_bytes_into() によって発生し、バッファーオブジェクトが小さすぎてメッセージが読み込めないことを示します。

e が *BufferTooShort* のインスタンスであるとすると、e.args[0] はそのメッセージをバイト文字列で与えるものです。

exception multiprocessing.**AuthenticationError**

認証エラーがあった場合に送出されます。

exception multiprocessing.**TimeoutError**

タイムアウトをサポートするメソッドでタイムアウトが過ぎたときに送出されます。

パイプ (Pipe) とキュー (Queue)

複数のプロセスを使う場合、一般的にはメッセージパッシングをプロセス間通信に使用し、ロックのような同期プリミティブを使用しないようにします。

メッセージのやりとりのために *Pipe()* (2 つのプロセス間の通信用)、もしくはキュー (複数のメッセージ生成プロセス (producer)、消費プロセス (consumer) の実現用) を使うことができます。

Queue, *SimpleQueue* と *JoinableQueue* 型は複数プロセスから生成/消費を行う FIFO キューです。これらのキューは標準ライブラリの *queue.Queue* を模倣しています。*Queue* には Python 2.5 の *queue.Queue* クラスで導入された *task_done()* と *join()* メソッドがないことが違う点です。

もし *JoinableQueue* を使用するなら、キューから削除される各タスクのために *JoinableQueue.task_done()* を呼び出さなければ **なりません** 。さもないと、いつか完了していないタスクを数えるた

めのセマフォがオーバーフローし、例外を発生させるでしょう。

管理オブジェクトを使用することで共有キューを作成できることも覚えておいてください。詳細は マネージャー を参照してください。

注釈: *multiprocessing* は、タイムアウトを伝えるために、通常の *queue.Empty* と *queue.Full* 例外を使用します。それらは *multiprocessing* の名前空間では利用できないため、*queue* からインポートする必要があります。

注釈: オブジェクトがキューに追加される際、そのオブジェクトは pickle 化されています。そのため、バックグラウンドのスレッドが後になって下位層のパイプに pickle 化されたデータをフラッシュすることがあります。これにより、少し驚くような結果になりますが、実際に問題になることはないはずです。これが問題になるような状況では、かわりに *manager* を使ってキューを作成することができるからです。

(1) 空のキューの中にオブジェクトを追加した後、キューの *empty()* メソッドが *False* を返すまでの間にごくわずかな遅延が起きることがあり、*get_nowait()* が *queue.Empty* を発生させることなく制御が呼び出し元に返ってしまうことがあります。

(2) 複数のプロセスがオブジェクトをキューに詰めている場合、キューの反対側ではオブジェクトが詰められたのとは違う順序で取得される可能性があります。ただし、同一のプロセスから詰め込まれたオブジェクトは、それらのオブジェクト間では、必ず期待どおりの順序になります。

警告: *Queue* を利用しようとしている最中にプロセスを *Process.terminate()* や *os.kill()* で終了させる場合、キューにあるデータは破損し易くなります。終了した後で他のプロセスがキューを利用しようとすると、例外を発生させる可能性があります。

警告: 上述したように、もし子プロセスがキューへ要素を追加するなら (かつ *JoinableQueue.cancel_join_thread* を使用しないなら) そのプロセスはバッファーされたすべての要素がパイプへフラッシュされるまで終了しません。

これは、そのプロセスを join しようとする場合、キューに追加されたすべての要素が消費されたことが確実でないかぎり、デッドロックを発生させる可能性があることを意味します。似たような現象で、子プロセスが非デーモンプロセスの場合、親プロセスは終了時に非デーモンのすべての子プロセスを join しようとしてハングアップする可能性があります。

マネージャーを使用して作成されたキューではこの問題はありません。詳細は プログラミングガイドライン を参照してください。

プロセス間通信におけるキューの使用例を知りたいなら 使用例 を参照してください。

`multiprocessing.Pipe(`[*duplex*]`)`

パイプの両端を表す *Connection* オブジェクトのペア (conn1, conn2) を返します。

duplex が True (デフォルト) ならパイプは双方向性です。*duplex* が False ならパイプは一方向性で、conn1 はメッセージの受信専用、conn2 はメッセージの送信専用になります。

class multiprocessing.**Queue**([*maxsize*])

パイプや 2〜3 個のロック/セマフォを使用して実装されたプロセス共有キューを返します。あるプロセスが最初に要素をキューへ追加するとき、バッファーからパイプの中へオブジェクトを転送する供給スレッドが開始されます。

標準ライブラリの *queue* モジュールの通常の *queue.Empty* や *queue.Full* 例外がタイムアウトを伝えるために送出されます。

Queue は *task_done()* や *join()* を除く *queue.Queue* のすべてのメソッドを実装します。

qsize()

おおよそのキューのサイズを返します。マルチスレッディング/マルチプロセスの特性上、この数値は信用できません。

これは sem_getvalue() が実装されていない Mac OS X のような Unix プラットホーム上で *NotImplementedError* を発生させる可能性があることを覚えておいてください。

empty()

キューが空っぽなら True を、そうでなければ False を返します。マルチスレッディング/マルチプロセシングの特性上、これは信用できません。

full()

キューがいっぱいなら True を、そうでなければ False を返します。マルチスレッディング/マルチプロセシングの特性上、これは信用できません。

put(*obj*[, *block*[, *timeout*]])

キューの中へ obj を追加します。オプションの引数 *block* が True (デフォルト) 且つ *timeout* が None (デフォルト) なら、空きスロットが利用可能になるまで必要であればブロックします。*timeout* が正の数なら、最大 *timeout* 秒ブロックして、その時間内に空きスロットが利用できなかったら *queue.Full* 例外を発生させます。それ以外 (*block* が False) で、空きスロットがすぐに利用可能な場合はキューに要素を追加します。そうでなければ *queue.Full* 例外が発生します (その場合 *timeout* は無視されます)。

バージョン 3.8 で変更: If the queue is closed, *ValueError* is raised instead of *AssertionError*.

put_nowait(*obj*)

put(obj, False) と等価です。

get([*block*[, *timeout*]])

キューから要素を取り出して削除します。オプションの引数 *block* が True (デフォルト) 且つ *timeout* が None (デフォルト) なら、要素が取り出せるまで必要であればブロックします。*timeout* が正の数なら、最大 *timeout* 秒ブロックして、その時間内に要素が取り出せなかったら *queue.Empty* 例外を発生させます。それ以外 (*block* が False) で、要素がすぐに取り出せる場合は要素を返します。そうでなければ *queue.Empty* 例外が発生します (その場合 *timeout* は無視さ

れます)。

バージョン 3.8 で変更: If the queue is closed, *ValueError* is raised instead of *OSError*.

get_nowait()

get(False) と等価です。

multiprocessing.Queue は *queue.Queue* にはない追加メソッドがあります。これらのメソッドは通常、ほとんどのコードに必要ありません:

close()

カレントプロセスからこのキューへそれ以上データが追加されないことを表します。バックグラウンドスレッドはパイプへバッファーされたすべてのデータをフラッシュするとすぐに終了します。これはキューがガベージコレクトされるときに自動的に呼び出されます。

join_thread()

バックグラウンドスレッドを join します。このメソッドは *close()* が呼び出された後でのみ使用されます。バッファーされたすべてのデータがパイプへフラッシュされるのを保証するため、バックグラウンドスレッドが終了するまでブロックします。

デフォルトでは、あるプロセスがキューを作成していない場合、終了時にキューのバックグラウンドスレッドを join しようとします。そのプロセスは *join_thread()* が何もしないように *cancel_join_thread()* を呼び出すことができます。

cancel_join_thread()

join_thread() がブロッキングするのを防ぎます。特にこれはバックグラウンドスレッドがそのプロセスの終了時に自動的に join されるのを防ぎます。詳細は *join_thread()* を参照してください。

このメソッドは allow_exit_without_flush() という名前のほうがよかったかもしれません。キューに追加されたデータが失われてしまいがちなため、このメソッドを使う必要はほぼ確実にないでしょう。本当にこれが必要になるのは、キューに追加されたデータを下位層のパイプにフラッシュすることなくカレントプロセスを直ちに終了する必要があり、かつ失われるデータに関心がない場合です。

注釈: このクラスに含まれる機能には、ホストとなるオペレーティングシステム上で動作している共有セマフォ (shared semaphore) を使用しているものがあります。これが使用できない場合には、このクラスが無効になり、*Queue* をインスタンス化する時に *ImportError* が発生します。詳細は bpo-3770 を参照してください。同様のことが、以下に列挙されている特殊なキューでも成り立ちます。

---

class multiprocessing.SimpleQueue

単純化された *Queue* 型です。ロックされた *Pipe* と非常に似ています。

empty()

キューが空ならば True を、そうでなければ False を返します。

get()

キューから要素を削除して返します。

put(*item*)
item をキューに追加します。

class multiprocessing.JoinableQueue([*maxsize*])
JoinableQueue は *Queue* のサブクラスであり、*task_done()* や *join()* メソッドが追加されているキューです。

task_done()
以前にキューへ追加されたタスクが完了したことを表します。キューのコンシューマによって使用されます。タスクをフェッチするために使用されるそれぞれの *get()* に対して、後続の *task_done()* 呼び出しはタスクの処理が完了したことをキューへ伝えます。

もし *join()* がブロッキング状態なら、すべての要素が処理されたときに復帰します (*task_done()* 呼び出しが すべての要素からキュー内へ *put()* されたと受け取ったことを意味します)。

キューにある要素より多く呼び出された場合 *ValueError* が発生します。

join()
キューにあるすべてのアイテムが取り出されて処理されるまでブロックします。

キューに要素が追加されると未完了タスク数が増えます。コンシューマがキューの要素が取り出されてすべての処理が完了したことを表す *task_done()* を呼び出すと数が減ります。未完了タスク数がゼロになると *join()* はブロッキングを解除します。

その他

multiprocessing.active_children()
カレントプロセスのすべてのアクティブな子プロセスのリストを返します。

これを呼び出すと "join" してすでに終了しているプロセスには副作用があります。

multiprocessing.cpu_count()
システムの CPU 数を返します。

この数は現在のプロセスが使える CPU 数と同じものではありません。使用可能な CPU 数は `len(os.sched_getaffinity(0))` で取得できます。

NotImplementedError を送出するかもしれません。

参考:

os.cpu_count()

multiprocessing.current_process()
カレントプロセスに対応する *Process* オブジェクトを返します。

threading.current_thread() とよく似た関数です。

multiprocessing.parent_process()

Return the *Process* object corresponding to the parent process of the *current_process()*. For the main process, parent_process will be None.

バージョン 3.8 で追加.

multiprocessing.freeze_support()

multiprocessing を使用しているプログラムをフリーズして Windows の実行可能形式を生成するためのサポートを追加します。(**py2exe** , **PyInstaller** や **cx_Freeze** でテストされています。)

メインモジュールの if __name__ == '__main__' の直後にこの関数を呼び出す必要があります。以下に例を示します:

```
from multiprocessing import Process, freeze_support

def f():
    print('hello world!')

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

もし freeze_support() の行がない場合、フリーズされた実行可能形式を実行しようとすると *RuntimeError* を発生させます。

freeze_support() の呼び出しは Windows 以外の OS では効果がありません。さらに、もしモジュールが Windows の通常の Python インタプリタによって実行されているならば (プログラムがフリーズされていなければ) freeze_support() は効果がありません。

multiprocessing.get_all_start_methods()

サポートしている開始方式のリストを返します。先頭の要素がデフォルトを意味します。利用可能な開始方式には 'fork'、'spawn' および 'forkserver' があります。Windows では 'spawn' のみが利用可能です。Unix では 'fork' および 'spawn' は常にサポートされており、'fork' がデフォルトになります。

バージョン 3.4 で追加.

multiprocessing.get_context(*method=None*)

multiprocessing モジュールと同じ属性を持つコンテキストオブジェクトを返します。

method が None の場合、デフォルトのコンテキストが返されます。その他の場合 *method* は 'fork'、'spawn' あるいは 'forkserver' でなければなりません。指定された開始方式が利用できない場合は *ValueError* が送出されます。

バージョン 3.4 で追加.

multiprocessing.get_start_method(*allow_none=False*)

開始するプロセスで使用する開始方式名を返します。

開始方式がまだ確定しておらず、*allow_none* の値が偽の場合、開始方式はデフォルトに確定され、その名前が返されます。開始方式が確定しておらず、*allow_none* の値が真の場合、None が返されます。

The return value can be `'fork'`, `'spawn'`, `'forkserver'` or `None`. `'fork'` is the default on Unix, while `'spawn'` is the default on Windows and macOS.

バージョン 3.8 で変更: macOS では、*spawn* 開始方式がデフォルトになりました。*fork* 開始方法は、サブプロセスのクラッシュを引き起こす可能性があるため、安全ではありません。bpo-33725 を参照。

バージョン 3.4 で追加.

multiprocessing.set_executable()

子プロセスを開始するときに、使用する Python インタープリターのパスを設定します。(デフォルトでは *sys.executable* が使用されます)。コードに組み込むときは、おそらく次のようにする必要があります

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

子プロセスを作成する前に行ってください。

バージョン 3.4 で変更: Unix で開始方式に `'spawn'` を使用している場合にサポートされました。

multiprocessing.set_start_method(*method*)

子プロセスの開始方式を指定します。*method* には `'fork'`、`'spawn'` あるいは `'forkserver'` を指定できます。

これは一度しか呼び出すことができず、その場所もメインモジュールの `if __name__ == '__main__'` 節内で保護された状態でなければなりません。

バージョン 3.4 で追加.

注釈: *multiprocessing* には *threading.active_count()*, *threading.enumerate()*, *threading.settrace()*, *threading.setprofile()*, *threading.Timer* や *threading.local* のような関数はありません。

Connection オブジェクト

Connection オブジェクトは pickle でシリアライズ可能なオブジェクトか文字列を送ったり、受け取ったりします。そういったオブジェクトはメッセージ指向の接続ソケットと考えられます。

Connection オブジェクトは通常は `Pipe` を使用して作成されます。詳細は :ref:`multiprocessing-listeners-clients()` も参照してください。

class multiprocessing.connection.Connection

send(*obj*)

コネクションの相手側へ *recv()* を使用して読み込むオブジェクトを送ります。

オブジェクトは pickle でシリアライズ可能でなければなりません。pickle が極端に大きすぎる (OS にも依りますが、およそ 32 MiB+) と、*ValueError* 例外が送出されることがあります。

recv()
コネクションの相手側から *send()* を使用して送られたオブジェクトを返します。何か受け取るまでブロックします。何も受け取らずにコネクションの相手側でクローズされた場合 *EOFError* が発生します。

fileno()
コネクションが使用するハンドラーか、ファイル記述子を返します。

close()
コネクションをクローズします。

コネクションがガベージコレクトされるときに自動的に呼び出されます。

poll([*timeout*])
読み込み可能なデータがあるかどうかを返します。

timeout が指定されていなければすぐに返します。*timeout* に数値を指定すると、最大指定した秒数をブロッキングします。*timeout* に None を指定するとタイムアウトせずにずっとブロッキングします。

multiprocessing.connection.wait() を使って複数のコネクションオブジェクトを同時にポーリングできることに注意してください。

send_bytes(*buffer*[, *offset*[, *size*]])
bytes-like object から完全なメッセージとしてバイトデータを送ります。

offset が指定されると *buffer* のその位置からデータが読み込まれます。*size* が指定されるとバッファーからその量のデータが読み込まれます。非常に大きなバッファー (OS に依存しますが、およそ 32MiB+) を指定すると、*ValueError* 例外が発生するかもしれません。

recv_bytes([*maxlength*])
コネクションの相手側から送られたバイトデータの完全なメッセージを文字列として返します。何か受け取るまでブロックします。受け取るデータが何も残っておらず、相手側がコネクションを閉じていた場合、*EOFError* が送出されます。

maxlength を指定していて、かつメッセージが *maxlength* より長い場合、*OSError* が発生してコネクションからそれ以上読めなくなります。

バージョン 3.3 で変更: この関数は以前は *IOError* を送出していました。今では *OSError* の別名です。

recv_bytes_into(*buffer*[, *offset*])
コネクションの相手側から送られたバイトデータを *buffer* に読み込み、メッセージのバイト数を返します。何か受け取るまでブロックします。何も受け取らずにコネクションの相手側でクローズされた場合 *EOFError* が発生します。

buffer は書き込み可能な *bytes-like object* でなければなりません。*offset* が与えられたら、その位置からバッファーへメッセージが書き込まれます。オフセットは *buffer* バイトよりも小さい正の数でなければなりません。

バッファーがあまりに小さいと BufferTooShort 例外が発生します。e が例外インスタンスとすると完全なメッセージは e.args[0] で確認できます。

バージョン 3.3 で変更: *Connection.send()* と *Connection.recv()* を使用して Connection オブジェクト自体をプロセス間で転送できるようになりました。

バージョン 3.3 で追加: Connection オブジェクトがコンテキストマネージメント・プロトコルをサポートするようになりました。-- **コンテキストマネージャ型** を参照してください。*__enter__()* は Connection オブジェクトを返します。また *__exit__()* は *close()* を呼び出します。

例えば:

```
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes(b'thank you')
>>> a.recv_bytes()
b'thank you'
>>> import array
>>> arr1 = array.array('i', range(5))
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

警告: *Connection.recv()* メソッドは受信したデータを自動的に unpickle 化します。それはメッセージを送ったプロセスが信頼できる場合を除いてセキュリティリスクになります。

そのため Pipe() を使用してコネクションオブジェクトを生成する場合を除いて、何らかの認証処理を実行した後で *recv()* や *send()* メソッドのみを使用すべきです。詳細は **認証キー** を参照してください。

警告: もしプロセスがパイプの読み込みまたは書き込み中に kill されると、メッセージの境界がどこなのか分からなくなってしまうので、そのパイプ内のデータは破損してしまいがちです。

同期プリミティブ

一般的にマルチプロセスプログラムは、マルチスレッドプログラムほどは同期プリミティブを必要としません。詳細は *threading* モジュールのドキュメントを参照してください。

マネージャーオブジェクトを使用して同期プリミティブを作成できることも覚えておいてください。詳細は マネージャー を参照してください。

class multiprocessing.**Barrier**(*parties*[, *action*[, *timeout*]])
バリアーオブジェクト: *threading.Barrier* のクローンです。

バージョン 3.3 で追加.

class multiprocessing.**BoundedSemaphore**([*value*])
有限セマフォオブジェクト: *threading.BoundedSemaphore* の類似物です。

よく似た *threading.BoundedSemaphore* とは、次の一点だけ異なります。acquire メソッドの第一引数名は *block* で、*Lock.acquire()* と一致しています。

注釈: Mac OS X では sem_getvalue() が実装されていないので *Semaphore* と区別がつきません。

---

class multiprocessing.**Condition**([*lock*])
状態変数: *threading.Condition* の別名です。

lock を指定するなら *multiprocessing* の *Lock* か *RLock* オブジェクトにすべきです。

バージョン 3.3 で変更: *wait_for()* メソッドが追加されました。

class multiprocessing.**Event**
threading.Event のクローンです。

class multiprocessing.**Lock**
再帰しないロックオブジェクトで、*threading.Lock* 相当のものです。プロセスやスレッドがロックをいったん獲得 (acquire) すると、それに続くほかのプロセスやスレッドが獲得しようとする際、それが解放 (release) されるまではブロックされます。解放はどのプロセス、スレッドからも行えます。スレッドに対して適用される *threading.Lock* のコンセプトと振る舞いは、特筆すべきものがない限り、プロセスとスレッドに適用される *multiprocessing.Lock* に引き継がれています。

Lock は実際にはファクトリ関数で、デフォルトコンテキストで初期化された multiprocessing.synchronize.Lock のインスタンスを返すことに注意してください。

Lock は *context manager* プロトコルをサポートしています。つまり with 文で使うことができます。

acquire(*block=True*, *timeout=None*)
ブロックあり、またはブロックなしでロックを獲得します。

引数 *block* を True (デフォルト) に設定して呼び出した場合、ロックがアンロック状態になるまでブロックします。ブロックから抜けるとそれをロック状態にしてから True を返します。*threading.Lock.acquire()* の最初の引数とは名前が違っているので注意してください。

引数 *block* の値を `False` にして呼び出すとブロックしません。現在ロック状態であれば、直ちに `False` を返します。それ以外の場合には、ロックをロック状態にして `True` を返します。

timeout として正の浮動小数点数を与えて呼び出すと、ロックが獲得できない限り、指定された秒数だけブロックします。*timeout* 値に負数を与えると、ゼロを与えた場合と同じになります。*timeout* 値の `None` (デフォルト) を与えると、無限にブロックします。*timeout* 引数の負数と `None` の扱いは、*threading.Lock.acquire()* に実装された動作と異なるので注意してください。*block* が `False` の場合、*timeout* は実際的な意味を持たなくなるので無視されます。ロックを獲得した場合は `True` 、タイムアウトした場合は `False` で戻ります。

release()
ロックを解放します。これはロックを獲得したプロセスやスレッドだけでなく、任意のプロセスやスレッドから呼ぶことができます。

threading.Lock.release() と同じように振舞いますが、ロックされていない場合に呼び出すと *ValueError* となる点だけが違います。

class multiprocessing.RLock
再帰ロックオブジェクトで、*threading.RLock* 相当のものです。再帰ロックオブジェクトはそれを獲得 (acquire) したプロセスやスレッドが解放 (release) しなければなりません。プロセスやスレッドがロックをいったん獲得すると、同じプロセスやスレッドはブロックされずに再度獲得出来ます。そのプロセスやスレッドは獲得した回数ぶん解放しなければなりません。

RLock は実際にはファクトリ関数で、デフォルトコンテキストで初期化された `multiprocessing.synchronize.Lock` のインスタンスを返すことに注意してください。

RLock は *context manager* プロトコルをサポートしています。つまり `with` 文で使うことができます。

acquire(*block=True*, *timeout=None*)
ブロックあり、またはブロックなしでロックを獲得します。

block 引数を `True` にして呼び出した場合、ロックが既にカレントプロセスもしくはカレントスレッドが既に所有していない限りは、アンロック状態 (どのプロセス、スレッドも所有していない状態) になるまでブロックします。ブロックから抜けるとカレントプロセスもしくはカレントスレッドが (既に持っていなければ) 所有権を得て、再帰レベルをインクリメントし、`True` で戻ります。*threading.RLock.acquire()* の実装とはこの最初の引数の振る舞いが、その名前自身を始めとしていくつか違うので注意してください。

block 引数を `False` にして呼び出した場合、ブロックしません。ロックが他のプロセスもしくはスレッドにより獲得済み (つまり所有されている) であれば、カレントプロセスまたはカレントスレッドは所有権を得ず、再帰レベルも変更せずに、`False` で戻ります。ロックがアンロック状態の場合、カレントプロセスもしくはカレントスレッドは所有権を得て再帰レベルがインクリメントされ、`True` で戻ります。(---訳注: *block* の True/False 関係なくここでの説明では「所有権を持っている場合の 2 度目以降の aquire」の説明が欠けています。2 度目以降の acquire では再帰レベルがインクリメントされて即座に返ります。全体読めばわかるとは思いますが一応。---)

timeout 引数の使い方と振る舞いは *Lock.acquire()* と同じです。*timeout* 引数の振る舞いがいくつかの点で *threading.RLock.acquire()* と異なるので注意してください。

release()

再帰レベルをデクリメントしてロックを解放します。デクリメント後に再帰レベルがゼロになった場合、ロックの状態をアンロック (いかなるプロセス、いかなるスレッドにも所有されていない状態) にリセットし、ロックの状態がアンロックになるのを待ってブロックしているプロセスもしくはスレッドがある場合にはその中のただ一つだけが処理を進行できるようにします。デクリメント後も再帰レベルがゼロでない場合、ロックの状態はロックのままで、呼び出し側のプロセスもしくはスレッドに所有されたままになります。

このメソッドは呼び出しプロセスあるいはスレッドがロックを所有している場合に限り呼び出してください。所有者でないプロセスもしくはスレッドによって呼ばれるか、あるいはアンロック (未所有) 状態で呼ばれた場合、*AssertionError* が送出されます。同じ状況での *threading.RLock.release()* 実装とは例外の型が異なるので注意してください。

class multiprocessing.Semaphore([*value*])

セマフォオブジェクト: *threading.Semaphore* のクローンです。

よく似た *threading.BoundedSemaphore* とは、次の一点だけ異なります。acquire メソッドの第一引数名は *block* で、*Lock.acquire()* と一致しています。

注釈: Mac OS X では sem_timedwait がサポートされていないので、acquire() にタイムアウトを与えて呼ぶと、ループ内でスリープすることでこの関数がエミュレートされます。

注釈: メインスレッドが BoundedSemaphore.acquire(), *Lock.acquire()*, *RLock.acquire()*, Semaphore.acquire(), Condition.acquire() 又は Condition.wait() を呼び出してブロッキング状態のときに Ctrl-C で生成される SIGINT シグナルを受け取ると、その呼び出しはすぐに中断されて *KeyboardInterrupt* が発生します。

これは同等のブロッキング呼び出しが実行中のときに SIGINT が無視される *threading* の振る舞いとは違っています。

注釈: このパッケージに含まれる機能には、ホストとなるオペレーティングシステム上で動作している共有セマフォを使用しているものがあります。これが使用できない場合には、multiprocessing.synchronize モジュールが無効になり、このモジュールのインポート時に *ImportError* が発生します。詳細は bpo-3770 を参照してください。

共有 ctypes オブジェクト

子プロセスにより継承される共有メモリを使用する共有オブジェクトを作成することができます。

multiprocessing.**Value**(*typecode_or_type*, **args*, *lock=True*)

共有メモリから割り当てられた *ctypes* オブジェクトを返します。デフォルトでは、返り値は実際のオブジェクトの同期ラッパーです。オブジェクトそれ自身は、*Value* の *value* 属性によってアクセスできます。

typecode_or_type は返されるオブジェクトの型を決めます。それは ctypes の型か *array* モジュールで使用されるような 1 文字の型コードかのどちらか一方です。**args* は型のコンストラクターへ渡されます。

lock が True (デフォルト) なら、値へ同期アクセスするために新たに再帰的なロックオブジェクトが作成されます。*lock* が *Lock* か *RLock* なら値への同期アクセスに使用されます。*lock* が False なら、返されたオブジェクトへのアクセスはロックにより自動的に保護されません。そのため、必ずしも ” プロセスセーフ” ではありません。

+= のような演算は、読み込みと書き込みを含むためアトミックでありません。このため、たとえば自動的に共有の値を増加させたい場合、以下のようにするのでは不十分です

```
counter.value += 1
```

関連するロックが再帰的 (それがデフォルトです) なら、かわりに次のようにします

```
with counter.get_lock():
    counter.value += 1
```

lock はキーワード専用引数であることに注意してください。

multiprocessing.**Array**(*typecode_or_type*, *size_or_initializer*, ***, *lock=True*)

共有メモリから割り当てられた ctypes 配列を返します。デフォルトでは、返り値は実際の配列の同期ラッパーです。

typecode_or_type は返される配列の要素の型を決めます。それは ctypes の型か *array* モジュールで使用されるような 1 文字の型コードかのどちらか一方です。*size_or_initializer* が整数なら、配列の長さを決定し、その配列はゼロで初期化されます。別の使用方法として *size_or_initializer* は配列の初期化に使用されるシーケンスになり、そのシーケンス長が配列の長さを決定します。

lock が True (デフォルト) なら、値へ同期アクセスするために新たなロックオブジェクトが作成されます。*lock* が *Lock* か *RLock* なら値への同期アクセスに使用されます。*lock* が False なら、返されたオブジェクトへのアクセスはロックにより自動的に保護されません。そのため、必ずしも ” プロセスセーフ” ではありません。

lock はキーワード引数としてのみ利用可能なことに注意してください。

ctypes.c_char の配列は文字列を格納して取り出せる *value* と *raw* 属性を持っていることを覚えておいてください。

multiprocessing.sharedctypes モジュール

multiprocessing.sharedctypes モジュールは子プロセスに継承される共有メモリの *ctypes* オブジェクトを割り当てる関数を提供します。

注釈: 共有メモリのポインターを格納することは可能ではありますが、特定プロセスのアドレス空間の位置を参照するということを覚えておいてください。しかし、そのポインターは別のプロセスのコンテキストにおいて無効になる確率が高いです。そして、別のプロセスからそのポインターを逆参照しようとするとクラッシュを引き起こす可能性があります。

---

multiprocessing.sharedctypes.**RawArray**(*typecode_or_type*, *size_or_initializer*)

共有メモリから割り当てられた ctypes 配列を返します。

typecode_or_type は返される配列の要素の型を決めます。それは ctypes の型か *array* モジュールで使用されるような 1 文字の型コードのどちらか一方です。*size_or_initializer* が整数なら、それが配列の長さになり、その配列はゼロで初期化されます。別の使用方法として *size_or_initializer* には配列の初期化に使用されるシーケンスを設定することもでき、その場合はシーケンスの長さが配列の長さになります。

要素を取得したり設定したりすることは潜在的に非アトミックであることに注意してください。ロックを使用して自動的に同期化されたアクセスを保証するには *Array()* を使用してください。

multiprocessing.sharedctypes.**RawValue**(*typecode_or_type*, **args*)

共有メモリから割り当てられた ctypes オブジェクトを返します。

typecode_or_type は返されるオブジェクトの型を決めます。それは ctypes の型か *array* モジュールで使用されるような 1 文字の型コードかのどちらか一方です。**args* は型のコンストラクターへ渡されます。

値を取得したり設定したりすることは潜在的に非アトミックであることに注意してください。ロックを使用して自動的に同期化されたアクセスを保証するには *Value()* を使用してください。

ctypes.c_char の配列は文字列を格納して取り出せる `value` と `raw` 属性を持っていることを覚えておいてください。詳細は *ctypes* を参照してください。

multiprocessing.sharedctypes.**Array**(*typecode_or_type*, *size_or_initializer*, ***, *lock=True*)

RawArray() と同様ですが、*lock* の値によっては ctypes 配列をそのまま返す代わりに、プロセスセーフな同期ラッパーが返されます。

lock が `True` (デフォルト) なら、値へ同期アクセスするために新たな ロックオブジェクトが作成されます。*lock* が *Lock* か *RLock* なら値への同期アクセスに使用されます。*lock* が `False` なら、返された オブジェクトへのアクセスはロックにより自動的に保護されません。そのため、必ずしも ” プロセスセーフ” ではありません。

lock はキーワード専用引数であることに注意してください。

multiprocessing.sharedctypes.**Value**(*typecode_or_type*, **args*, *lock=True*)

RawValue() と同様ですが、*lock* の値によっては ctypes オブジェクトをそのまま返す代わりに、プロセスセーフな同期ラッパーが返されます。

lock が `True` (デフォルト) なら、値へ同期アクセスするために新たな ロックオブジェクトが作成されます。*lock* が *Lock* か *RLock* なら値への同期アクセスに使用されます。*lock* が `False` なら、返された オブジェクトへのアクセスはロックにより自動的に保護されません。そのため、必ずしも ”プロセスセーフ” ではありません。

lock はキーワード専用引数であることに注意してください。

`multiprocessing.sharedctypes.copy`(*obj*)

共有メモリから割り当てられた ctypes オブジェクト *obj* をコピーしたオブジェクトを返します。

`multiprocessing.sharedctypes.synchronized`(*obj*[, *lock*])

同期アクセスに *lock* を使用する ctypes オブジェクトのためにプロセスセーフなラッパーオブジェクトを返します。*lock* が `None` (デフォルト) なら、*multiprocessing.RLock* オブジェクトが自動的に作成されます。

同期ラッパーがラップするオブジェクトに加えて 2 つのメソッドがあります。`get_obj()` はラップされたオブジェクトを返します。`get_lock()` は同期のために使用されるロックオブジェクトを返します。

ラッパー経由で ctypes オブジェクトにアクセスすることは raw ctypes オブジェクトへアクセスするよりずっと遅くなることに注意してください。

バージョン 3.5 で変更: synchronized オブジェクトは **コンテキストマネージャ** プロトコルをサポートしています。

次の表は通常の ctypes 構文で共有メモリから共有 ctypes オブジェクトを作成するための構文を比較します。(`MyStruct` テーブル内には *ctypes.Structure* のサブクラスがあります。)

ctypes	type を使用する sharedctypes	typecode を使用する sharedctypes
c_double(2.4)	RawValue(c_double, 2.4)	RawValue('d', 2.4)
MyStruct(4, 6)	RawValue(MyStruct, 4, 6)	
(c_short * 7)()	RawArray(c_short, 7)	RawArray('h', 7)
(c_int * 3)(9, 2, 8)	RawArray(c_int, (9, 2, 8))	RawArray('i', (9, 2, 8))

以下に子プロセスが多くの ctypes オブジェクトを変更する例を紹介します:

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
```

(次のページに続く)

(前のページからの続き)

```
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
        a.y **= 2

if __name__ == '__main__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', b'hello world', lock=lock)
    A = Array(Point, [(1.875,-6.25), (-5.75,2.0), (2.375,9.5)], lock=lock)

    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()

    print(n.value)
    print(x.value)
    print(s.value)
    print([(a.x, a.y) for a in A])
```

結果は以下のように表示されます

```
49
0.1111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
```

マネージャー

マネージャーは異なるプロセス間で共有されるデータの作成方法を提供します。これには別のマシン上で走るプロセス間のネットワーク越しの共有も含まれます。マネージャーオブジェクトは **共有オブジェクト** を管理するサーバープロセスを制御します。他のプロセスはプロキシ経由で共有オブジェクトへアクセスすることができます。

multiprocessing.**Manager**()

プロセス間でオブジェクトを共有するために使用される *SyncManager* オブジェクトを返します。返されたマネージャーオブジェクトは生成される子プロセスに対応付けられ、共有オブジェクトを作成するメソッドや、共有オブジェクトに対応するプロキシを返すメソッドを持ちます。

マネージャープロセスは親プロセスが終了するか、ガベージコレクトされると停止します。マネージャークラスは *multiprocessing.managers* モジュールで定義されています:

class multiprocessing.managers.**BaseManager**(*[address[, authkey]]*)

BaseManager オブジェクトを作成します。

作成後、*start()* または `get_server().serve_forever()` を呼び出して、マネージャーオブジェク

トが、開始されたマネージャープロセスを確実に参照するようにしてください。

address はマネージャープロセスが新たなコネクションを待ち受けるアドレスです。*address* が None の場合、任意のアドレスが設定されます。

authkey はサーバープロセスへ接続しようとするコネクションの正当性を検証するために 使用される認証キーです。*authkey* が None の場合 `current_process().authkey` が使用されます。*authkey* を使用する場合はバイト文字列でなければなりません。

start([*initializer*[, *initargs*]])
マネージャーを開始するためにサブプロセスを開始します。*initializer* が None でなければ、サブプロセスは開始時に `initializer(*initargs)` を呼び出します。

get_server()
マネージャーの制御下にある実際のサーバーを表す Server オブジェクトを返します。Server オブジェクトは `serve_forever()` メソッドをサポートします:

```
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=('', 50000), authkey=b'abc')
>>> server = manager.get_server()
>>> server.serve_forever()
```

Server はさらに *address* 属性も持っています。

connect()
ローカルからリモートのマネージャーオブジェクトへ接続します:

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 50000), authkey=b'abc')
>>> m.connect()
```

shutdown()
マネージャーが使用するプロセスを停止します。これはサーバープロセスを開始するために *start()* が使用された場合のみ有効です。

これは複数回呼び出すことができます。

register(*typeid*[, *callable*[, *proxytype*[, *exposed*[, *method_to_typeid*[, *create_method*]]]]])
マネージャークラスで呼び出し可能オブジェクト (callable) や型を登録するために使用されるクラスメソッドです。

typeid は特に共有オブジェクトの型を識別するために使用される ”型識別子” です。これは文字列でなければなりません。

callable はこの型識別子のオブジェクトを作成するために使用される呼び出し可能オブジェクトです。マネージャーインスタンスが *connect()* メソッドを使ってサーバーに接続されているか、*create_method* 引数が False の場合は、None でも構いません。

proxytype はこの *typeid* で共有オブジェクトのプロキシを作成するために使用される *BaseProxy* のサブクラスです。None の場合、プロキシクラスは自動的に作成されます。

exposed は *BaseProxy._callmethod()* を使用したアクセスが許されるべき typeid をプロキシするメソッド名のシーケンスを指定するために使用されます (*exposed* が None の場合 proxytype._exposed_ が存在すればそれが代わりに使用されます)。*exposed* リストが指定されない場合、共有オブジェクトのすべての ” パブリックメソッド” がアクセス可能になります。(ここでいう ” パブリックメソッド” とは __call__() メソッドを持つものと名前が '_' で始まらないあらゆる属性を意味します。)

method_to_typeid はプロキシが返す exposed メソッドの返り値の型を指定するために使用されるマッピングで、メソッド名を typeid 文字列にマップします。(*method_to_typeid* が None の場合 proxytype._method_to_typeid_ が存在すれば、それが代わりに使用されます。) メソッド名がこのマッピングのキーではないか、マッピングが None の場合、そのメソッドによって返されるオブジェクトが値として (by value) コピーされます。

create_method は、共有オブジェクトを作成し、それに対するプロキシを返すようサーバープロセスに伝える、名前 *typeid* のメソッドを作成するかを決定します。デフォルトでは True です。

BaseManager インスタンスも読み出し専用属性を 1 つ持っています:

address
マネージャーが使用するアドレスです。

バージョン 3.3 で変更: マネージャーオブジェクトはコンテキストマネージメント・プロトコルをサポートします -- **コンテキストマネージャ型** を参照してください。*__enter__()* は (まだ開始していない場合) サーバープロセスを開始してから、マネージャーオブジェクトを返します。*__exit__()* は *shutdown()* を呼び出します。

旧バージョンでは、*__enter__()* はマネージャーのサーバープロセスがまだ開始していなかった場合でもプロセスを開始しませんでした。

class multiprocessing.managers.**SyncManager**
プロセス間の同期のために使用される *BaseManager* のサブクラスです。multiprocessing.Manager() はこの型のオブジェクトを返します。

Its methods create and return *Proxy* **オブジェクト** for a number of commonly used data types to be synchronized across processes. This notably includes shared lists and dictionaries.

Barrier(*parties*[, *action*[, *timeout*]])
共有 *threading.Barrier* オブジェクトを作成して、そのプロキシを返します。

バージョン 3.3 で追加.

BoundedSemaphore([*value*])
共有 *threading.BoundedSemaphore* オブジェクトを作成して、そのプロキシを返します。

Condition([*lock*])
共有 *threading.Condition* オブジェクトを作成して、そのプロキシを返します。

lock が提供される場合 *threading.Lock* か *threading.RLock* オブジェクトのためのプロキシになります。

バージョン 3.3 で変更: *wait_for()* メソッドが追加されました。

Event()
共有 *threading.Event* オブジェクトを作成して、そのプロキシを返します。

Lock()
共有 *threading.Lock* オブジェクトを作成して、そのプロキシを返します。

Namespace()
共有 *Namespace* オブジェクトを作成して、そのプロキシを返します。

Queue([*maxsize*])
共有 *queue.Queue* オブジェクトを作成して、そのプロキシを返します。

RLock()
共有 *threading.RLock* オブジェクトを作成して、そのプロキシを返します。

Semaphore([*value*])
共有 *threading.Semaphore* オブジェクトを作成して、そのプロキシを返します。

Array(*typecode*, *sequence*)
配列を作成して、そのプロキシを返します。

Value(*typecode*, *value*)
書き込み可能な `value` 属性を作成して、そのプロキシを返します。

dict()
dict(*mapping*)
dict(*sequence*)
共有 *dict* オブジェクトを作成して、そのプロキシを返します。

list()
list(*sequence*)
共有 *list* オブジェクトを作成して、そのプロキシを返します。

バージョン 3.6 で変更: 共有オブジェクトは入れ子もできます。例えば、共有リストのような共有コンテナオブジェクトは、*SyncManager* がまとめて管理し同期を取っている他の共有オブジェクトを保持できます。

class multiprocessing.managers.Namespace
SyncManager に登録することのできる型です。

Namespace オブジェクトにはパブリックなメソッドはありませんが、書き込み可能な属性を持ちます。そのオブジェクト表現はその属性の値を表示します。

しかし、Namespace オブジェクトのためにプロキシを使用するとき `'_'` が先頭に付く属性はプロキシの属性になり、参照対象の属性にはなりません:

```
>>> manager = multiprocessing.Manager()
>>> Global = manager.Namespace()
>>> Global.x = 10
```

(次のページに続く)

(前のページからの続き)

```
>>> Global.y = 'hello'
>>> Global._z = 12.3    # this is an attribute of the proxy
>>> print(Global)
Namespace(x=10, y='hello')
```

カスタマイズされたマネージャー

独自のマネージャーを作成するには、*BaseManager* のサブクラスを作成して、マネージャークラスで呼び出し可能なオブジェクトか新たな型を登録するために *register()* クラスメソッドを使用します。例えば:

```
from multiprocessing.managers import BaseManager

class MathsClass:
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
    pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    with MyManager() as manager:
        maths = manager.Maths()
        print(maths.add(4, 3))         # prints 7
        print(maths.mul(7, 8))         # prints 56
```

リモートマネージャーを使用する

あるマシン上でマネージャーサーバーを実行して、他のマシンからそのサーバーを使用するクライアントを持つことができます (ファイアウォールを通過できることが前提)。

次のコマンドを実行することでリモートクライアントからアクセスを受け付ける 1 つの共有キューのためにサーバーを作成します:

```
>>> from multiprocessing.managers import BaseManager
>>> from queue import Queue
>>> queue = Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda:queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

あるクライアントからサーバーへのアクセスは次のようになります:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.put('hello')
```

別のクライアントもそれを使用することができます:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'
```

ローカルプロセスもそのキューへアクセスすることができます。クライアント上で上述のコードを使用してアクセスします:

```
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super().__init__()
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
...
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

Proxy オブジェクト

プロキシは別のプロセスで (おそらく) 有効な共有オブジェクトを **参照する** オブジェクトです。共有オブジェクトはプロキシの **参照対象** になるということができます。複数のプロキシオブジェクトが同じ参照対象を持つ可能性もあります。

プロキシオブジェクトはその参照対象の対応するメソッドを呼び出すメソッドを持ちます (そうは言っても、参照対象のすべてのメソッドが必ずしもプロキシ経由で利用可能なわけではありません)。この方法で、プロキシオブジェクトはまるでその参照先と同じように使えます:

```
>>> from multiprocessing import Manager
>>> manager = Manager()
>>> l = manager.list([i*i for i in range(10)])
>>> print(l)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print(repr(l))
<ListProxy object, typeid 'list' at 0x...>
>>> l[4]
16
>>> l[2:5]
[4, 9, 16]
```

プロキシに *str()* を適用すると参照対象のオブジェクト表現を返すのに対して、*repr()* を適用するとプロキシのオブジェクト表現を返すことに注意してください。

プロキシオブジェクトの重要な機能は pickle 化ができることで、これによりプロセス間での受け渡しができます。そのため、参照対象が *Proxy* **オブジェクト** を持てます。これによって管理されたリスト、辞書、その他 *Proxy* **オブジェクト** をネストできます:

```
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)         # referent of a now contains referent of b
>>> print(a, b)
[<ListProxy object, typeid 'list' at ...>] []
>>> b.append('hello')
>>> print(a[0], b)
['hello'] ['hello']
```

同様に、辞書とリストのプロキシも他のプロキシの内部に入れてネストできます:

```
>>> l_outer = manager.list([ manager.dict() for i in range(2) ])
>>> d_first_inner = l_outer[0]
>>> d_first_inner['a'] = 1
>>> d_first_inner['b'] = 2
>>> l_outer[1]['c'] = 3
>>> l_outer[1]['z'] = 26
>>> print(l_outer[0])
{'a': 1, 'b': 2}
>>> print(l_outer[1])
{'c': 3, 'z': 26}
```

(プロキシでない) 標準の *list* オブジェクトや *dict* オブジェクトが参照対象に含まれていた場合、それらの可変な値の変更はマネージャーからは伝搬されません。というのも、プロキシには参照対象の中に含まれる値がいつ変更されたかを知る術が無いのです。しかし、コンテナプロキシに値を保存する (これはプロキシオブジェクトの `__setitem__` を起動します) 場合はマネージャーを通して変更が伝搬され、その要素を実際に変更するために、コンテナプロキシに変更後の値が再代入されます:

```
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
```

(次のページに続く)

(前のページからの続き)

```
# now mutate the dictionary
d = lproxy[0]
d['a'] = 1
d['b'] = 2
# at this point, the changes to d are not yet synced, but by
# updating the dictionary, the proxy is notified of the change
lproxy[0] = d
```

This approach is perhaps less convenient than employing nested *Proxy* **オブジェクト** for most use cases but also demonstrates a level of control over the synchronization.

注釈: *multiprocessing* のプロキシ型は値による比較に対して何もサポートしません。そのため、例えば以下のようになります:

```
>>> manager.list([1,2,3]) == [1,2,3]
False
```

比較を行いたいときは参照対象のコピーを使用してください。

---

class multiprocessing.managers.**BaseProxy**

プロキシオブジェクトは *BaseProxy* のサブクラスのインスタンスです。

_callmethod(*methodname*[, *args*[, *kwds*]])

プロキシの参照対象のメソッドの実行結果を返します。

proxy がプロキシで、プロキシ内の参照対象が obj ならこの式

```
proxy._callmethod(methodname, args, kwds)
```

はこの式を評価します

```
getattr(obj, methodname)(*args, **kwds)
```

(マネージャープロセス内の)。

返される値はその呼び出し結果のコピーか、新たな共有オブジェクトに対するプロキシになります。詳細は *BaseManager.register()* の *method_to_typeid* 引数のドキュメントを参照してください。

その呼び出しによって例外が発生した場合、*_callmethod()* によってその例外は再送出されます。他の例外がマネージャープロセスで発生したなら、RemoteError 例外に変換されたものが *_callmethod()* によって送出されます。

特に *methodname* が **公開** されていない場合は例外が発生することに注意してください。

_callmethod() の使用例になります:

```
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
>>> l._callmethod('__getitem__', (slice(2, 7),)) # equivalent to l[2:7]
[2, 3, 4, 5, 6]
>>> l._callmethod('__getitem__', (20,))          # equivalent to l[20]
Traceback (most recent call last):
...
IndexError: list index out of range
```

_getvalue()
参照対象のコピーを返します。

参照対象が unpickle 化できるなら例外を発生します。

__repr__()
プロキシオブジェクトのオブジェクト表現を返します。

__str__()
参照対象のオブジェクト表現を返します。

クリーンアップ

プロキシオブジェクトは弱参照 (weakref) コールバックを使用します。プロキシオブジェクトがガベージコレクトされるときにその参照対象が所有するマネージャーからその登録を取り消せるようにするためです。

共有オブジェクトはプロキシが参照しなくなったときにマネージャープロセスから削除されます。

プロセスプール

Pool クラスでタスクを実行するプロセスのプールを作成することができます。

class multiprocessing.pool.Pool([*processes*[, *initializer*[, *initargs*[, *maxtasksperchild*[, *context*]]]]])
プロセスプールオブジェクトは、ジョブを送り込めるワーカープロセスのプールを制御します。タイムアウトやコールバックのある非同期の実行をサポートし、並列 map 実装を持ちます。

processes は使用するワーカープロセスの数です。*processes* が None の場合 *os.cpu_count()* が返す値を使用します。

initializer が None ではない場合、各ワーカープロセスは開始時に `initializer(*initargs)` を呼び出します。

maxtasksperchild は、ワーカープロセスが exit して新たなワーカープロセスと置き替えられるまでの間に、ワーカープロセスが完了することのできるタスクの数です。この設定により未利用のリソースが解放されるようなります。デフォルトの *maxtasksperchild* は None で、これはワーカープロセスがプールと同じ期間だけ生き続けるということを意味します。

context はワーカープロセスを開始するために使用されるコンテキストの指定に使用できます。通常

プールは関数 multiprocessing.Pool() かコンテキストオブジェクトの *Pool()* メソッドを使用して作成されます。どちらの場合でも *context* は適切に設定されます。

プールオブジェクトのメソッドは、そのプールを作成したプロセスのみが呼び出すべきです。

警告: *multiprocessing.pool* objects have internal resources that need to be properly managed (like any other resource) by using the pool as a context manager or by calling *close()* and *terminate()* manually. Failure to do this can lead to the process hanging on finalization.

Note that is **not correct** to rely on the garbage colletor to destroy the pool as CPython does not assure that the finalizer of the pool will be called (see `object.__del__()` for more information).

バージョン 3.2 で追加: *maxtasksperchild*

バージョン 3.4 で追加: *context*

注釈: *Pool* 中のワーカープロセスは、典型的にはプールのワークキューの存続期間とちょうど同じだけ生き続けます。ワーカーに確保されたリソースを解放するために (Apache, mod_wsgi, などのような) 他のシステムによく見られるパターンは、プール内のワーカーが設定された量だけの仕事を完了したら exit とクリーンアップを行い、古いプロセスを置き換えるために新しいプロセスを生成するというものです。*Pool* の *maxtasksperchild* 引数は、この能力をエンドユーザーに提供します。

---

apply(*func*[, *args*[, *kwds*]])
引数 *args* とキーワード引数 *kwds* を伴って *func* を呼びます。結果が準備できるまでブロックします。このブロックがあるため、*apply_async()* の方が並行作業により適しています。加えて、*func* は、プール内の 1 つのワーカーだけで実行されます。

apply_async(*func*[, *args*[, *kwds*[, *callback*[, *error_callback*]]]])
apply() メソッドの派生版で *AsyncResult* オブジェクトを返します。

callback が指定された場合、それは単一の引数を受け取る呼び出し可能オブジェクトでなければなりません。結果を返せるようになったときに *callback* が結果オブジェクトに対して適用されます。ただし呼び出しが失敗した場合は、代わりに *error_callback* が適用されます。

error_callback が指定された場合、それは単一の引数を受け取る呼び出し可能オブジェクトでなければなりません。対象の関数が失敗した場合、例外インスタンスを伴って *error_callback* が呼ばれます。

コールバックは直ちに完了すべきです。なぜなら、そうしなければ、結果を扱うスレッドがブロックするからです。

map(*func*, *iterable*[, *chunksize*])
map() 組み込み関数の並列版です (*iterable* な引数を 1 つだけサポートするという違いはあります。もしも複数のイテラブルを使いたいのならば:meth:‘starmap‘を参照)。結果が出るまでブロッ

クします。

このメソッドはイテラブルをいくつものチャンクに分割し、プロセスプールにそれぞれ独立したタスクとして送ります。(概算の) チャンクサイズは *chunksize* を正の整数に設定することで指定できます。

Note that it may cause high memory usage for very long iterables. Consider using *imap()* or *imap_unordered()* with explicit *chunksize* option for better efficiency.

map_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])

map() メソッドの派生版で *AsyncResult* オブジェクトを返します。

callback が指定された場合、それは単一の引数を受け取る呼び出し可能オブジェクトでなければなりません。結果を返せるようになったときに *callback* が結果オブジェクトに対して適用されます。ただし呼び出しが失敗した場合は、代わりに *error_callback* が適用されます。

error_callback が指定された場合、それは単一の引数を受け取る呼び出し可能オブジェクトでなければなりません。対象の関数が失敗した場合、例外インスタンスを伴って *error_callback* が呼ばれます。

コールバックは直ちに完了すべきです。なぜなら、そうしなければ、結果を扱うスレッドがブロックするからです。

imap(*func*, *iterable*[, *chunksize*])

map() の遅延評価版です。

chunksize 引数は *map()* メソッドで使用されるものと同じです。引数 iterable がとても長いなら *chunksize* に大きな値を指定して使用する方がデフォルト値の `1` を使用するよりもジョブの完了が **かなり** 速くなります。

また *chunksize* が `1` の場合 *imap()* メソッドが返すイテレーターの `next()` メソッドはオプションで *timeout* パラメーターを持ちます。`next(timeout)` は、その結果が *timeout* 秒以内に返されないときに *multiprocessing.TimeoutError* を発生させます。

imap_unordered(*func*, *iterable*[, *chunksize*])

イテレーターが返す結果の順番が任意の順番で良いと見なされることを除けば *imap()* と同じです。(ワーカープロセスが 1 つしかない場合のみ ” 正しい” 順番になることが保証されます。)

starmap(*func*, *iterable*[, *chunksize*])

iterable の要素が、引数として unpack されるイテレート可能オブジェクトであると期待される以外は、*map()* と似ています。

そのため、*iterable* が `[(1,2), (3, 4)]` なら、結果は `[func(1,2), func(3,4)]` になります。

バージョン 3.3 で追加.

starmap_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])

starmap() と *map_async()* の組み合わせです。イテレート可能オブジェクトの *iterable* をイテレートして、unpack したイテレート可能オブジェクトを伴って *func* を呼び出します。結果オブジェクトを返します。

バージョン 3.3 で追加.

close()
これ以上プールでタスクが実行されないようにします。すべてのタスクが完了した後でワーカープロセスが終了します。

terminate()
実行中の処理を完了させずにワーカープロセスをすぐに停止します。プールオブジェクトがガベージコレクトされるときに *terminate()* が呼び出されます。

join()
ワーカープロセスが終了するのを待ちます。*join()* を使用する前に *close()* か *terminate()* を呼び出さなければなりません。

バージョン 3.3 で追加: Pool オブジェクトがコンテキストマネージメント・プロトコルをサポートするようになりました。-- **コンテキストマネージャ型** を参照してください。*__enter__()* は Pool オブジェクトを返します。また *__exit__()* は *terminate()* を呼び出します。

class multiprocessing.pool.AsyncResult
Pool.apply_async() や *Pool.map_async()* で返される結果のクラスです。

get(*[timeout]*)
結果を受け取ったときに返します。*timeout* が None ではなくて、その結果が *timeout* 秒以内に受け取れない場合 *multiprocessing.TimeoutError* が発生します。リモートの呼び出しが例外を発生させる場合、その例外は *get()* が再発生させます。

wait(*[timeout]*)
その結果が有効になるか *timeout* 秒経つまで待ちます。

ready()
その呼び出しが完了しているかどうかを返します。

successful()
その呼び出しが例外を発生させることなく完了したかどうかを返します。その結果が返せる状態でない場合 *ValueError* が発生します。

バージョン 3.7 で変更: If the result is not ready, *ValueError* is raised instead of *AssertionError*.

次の例はプールの使用例を紹介します:

```
from multiprocessing import Pool
import time

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(processes=4) as pool:         # start 4 worker processes
        result = pool.apply_async(f, (10,)) # evaluate "f(10)" asynchronously in a single␣
↪process
```

(次のページに続く)

(前のページからの続き)

```
        print(result.get(timeout=1))        # prints "100" unless your computer is *very* slow

        print(pool.map(f, range(10)))       # prints "[0, 1, 4,..., 81]"

        it = pool.imap(f, range(10))
        print(next(it))                     # prints "0"
        print(next(it))                     # prints "1"
        print(it.next(timeout=1))           # prints "4" unless your computer is *very* slow

        result = pool.apply_async(time.sleep, (10,))
        print(result.get(timeout=1))        # raises multiprocessing.TimeoutError
```

リスナーとクライアント

通常、プロセス間でメッセージを渡すにはキューを使用するか *Pipe()* が返す *Connection* オブジェクトを使用します。

しかし *multiprocessing.connection* モジュールにはさらに柔軟な仕組みがあります。このモジュールは、基本的にはソケットもしくは Windows の名前付きパイプを扱う高レベルのメッセージ指向 API を提供します。また、*hmac* モジュールを使用した **ダイジェスト認証** や同時の複数接続のポーリングもサポートします。

multiprocessing.connection.**deliver_challenge**(*connection*, *authkey*)

ランダム生成したメッセージをコネクションの相手側へ送信して応答を待ちます。

その応答がキーとして *authkey* を使用するメッセージのダイジェストと一致する場合、コネクションの相手側へ歓迎メッセージを送信します。そうでなければ *AuthenticationError* を発生させます。

multiprocessing.connection.**answer_challenge**(*connection*, *authkey*)

メッセージを受信して、そのキーとして *authkey* を使用するメッセージのダイジェストを計算し、ダイジェストを送り返します。

歓迎メッセージを受け取れない場合 *AuthenticationError* が発生します。

multiprocessing.connection.**Client**(*address*[, *family*[, *authkey*]])

address で渡したアドレスを使用するリスナーに対してコネクションを確立しようとして *Connection* を返します。

コネクション種別は *family* 引数で決定しますが、一般的には *address* のフォーマットから推測できるので、これは指定されません。(**アドレスフォーマット** を参照してください)

If *authkey* is given and not None, it should be a byte string and will be used as the secret key for an HMAC-based authentication challenge. No authentication is done if *authkey* is None. *AuthenticationError* is raised if authentication fails. See **認証キー**.

class multiprocessing.connection.**Listener**([*address*[, *family*[, *backlog*[, *authkey*]]]])

コネクションを ' 待ち受ける ' 束縛されたソケットか Windows の名前付きパイプのラッパーです。

address はリスナーオブジェクトの束縛されたソケットか名前付きパイプが使用するアドレスです。

注釈: '0.0.0.0' のアドレスを使用する場合、Windows 上の終点へ接続することができません。終点へ接続したい場合は '127.0.0.1' を使用すべきです。

family は使用するソケット (名前付きパイプ) の種別です。これは `'AF_INET'` (TCP ソケット), `'AF_UNIX'` (Unix ドメインソケット) または `'AF_PIPE'` (Windows 名前付きパイプ) という文字列のどれか 1 つになります。これらのうち `'AF_INET'` のみが利用可能であることが保証されています。*family* が `None` の場合 *address* のフォーマットから推測されたものが使用されます。*address* も `None` の場合はデフォルトが選択されます。詳細は **アドレスフォーマット** を参照してください。*family* が `'AF_UNIX'` で *address* が `None` の場合 *tempfile.mkstemp()* を使用して作成されたプライベートな一時ディレクトリにソケットが作成されます。

リスナーオブジェクトがソケットを使用する場合、ソケットに束縛されるときに *backlog* (デフォルトでは 1 つ) がソケットの *listen()* メソッドに対して渡されます。

If *authkey* is given and not None, it should be a byte string and will be used as the secret key for an HMAC-based authentication challenge. No authentication is done if *authkey* is None. *AuthenticationError* is raised if authentication fails. See **認証キー**.

`accept()`
: リスナーオブジェクトの名前付きパイプか束縛されたソケット上でコネクションを 受け付けて *Connection* オブジェクトを返します。認証が失敗した場合 *AuthenticationError* が発生します。

`close()`
: リスナーオブジェクトの名前付きパイプか束縛されたソケットをクローズします。これはリスナーがガベージコレクトされるときに自動的に呼ばれます。そうは言っても、明示的に close() を呼び出す方が望ましいです。

リスナーオブジェクトは次の読み出し専用属性を持っています:

`address`
: リスナーオブジェクトが使用中のアドレスです。

`last_accepted`
: 最後にコネクションを受け付けたアドレスです。有効なアドレスがない場合は `None` になります。

バージョン 3.3 で追加: Listener オブジェクトがコンテキストマネージメント・プロトコルをサポートするようになりました。-- **コンテキストマネージャ型** を参照してください。*__enter__()* はリスナーオブジェクトを返します。また *__exit__()* は *close()* を呼び出します。

`multiprocessing.connection.wait(`*object_list*, *timeout=None*`)`
: *object_list* 中のオブジェクトが準備ができるまで待機します。準備ができた *object_list* 中のオブジェクトのリストを返します。*timeout* が浮動小数点なら、最大でその秒数だけ呼び出しがブロックします。*timeout* が `None` の場合、無制限の期間ブロックします。負のタイムアウトは 0 と等価です。

Unix と Windows の両方で、*object_list* には以下のオブジェクトを含めることが出来ます

- 読み取り可能な *Connection* オブジェクト;
- 接続された読み取り可能な *socket.socket* オブジェクト; または
- *Process* オブジェクトの *sentinel* 属性。

読み取ることのできるデータがある場合、あるいは相手側の端が閉じられている場合、コネクションまたはソケットオブジェクトは準備ができています。

Unix: `wait(object_list, timeout)` は `select.select(object_list, [], [], timeout)` とほとんど等価です。違いは、*select.select()* がシグナルによって中断される場合、`EINTR` のエラー番号付きで *OSError* を上げるということです。*wait()* はそのようなことは行いません。

Windows: *object_list* の要素は、(Win32 関数 `WaitForMultipleObjects()` のドキュメントで使われている定義から) wait 可能な整数ハンドルか、ソケットハンドルまたはパイプハンドルを返す `fileno()` メソッドを持つオブジェクトのどちらかでなければなりません。(パイプハンドルとソケットハンドラーは wait 可能なハンドルでは **ない** ことに注意してください。)

バージョン 3.3 で追加.

例

次のサーバーコードは認証キーとして `'secret password'` を使用するリスナーを作成します。このサーバーはコネクションを待ってクライアントへデータを送信します:

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)     # family is deduced to be 'AF_INET'

with Listener(address, authkey=b'secret password') as listener:
    with listener.accept() as conn:
        print('connection accepted from', listener.last_accepted)

        conn.send([2.25, None, 'junk', float])

        conn.send_bytes(b'hello')

        conn.send_bytes(array('i', [42, 1729]))
```

次のコードはサーバーへ接続して、サーバーからデータを受信します:

```
from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)

with Client(address, authkey=b'secret password') as conn:
    print(conn.recv())                  # => [2.25, None, 'junk', float]

    print(conn.recv_bytes())            # => 'hello'
```

(次のページに続く)

(前のページからの続き)

```
    arr = array('i', [0, 0, 0, 0, 0])
    print(conn.recv_bytes_into(arr))    # => 8
    print(arr)                          # => array('i', [42, 1729, 0, 0, 0])
```

次のコードは *wait()* を使って複数のプロセスからのメッセージを同時に待ちます:

```
import time, random
from multiprocessing import Process, Pipe, current_process
from multiprocessing.connection import wait

def foo(w):
    for i in range(10):
        w.send((i, current_process().name))
    w.close()

if __name__ == '__main__':
    readers = []

    for i in range(4):
        r, w = Pipe(duplex=False)
        readers.append(r)
        p = Process(target=foo, args=(w,))
        p.start()
        # We close the writable end of the pipe now to be sure that
        # p is the only process which owns a handle for it.  This
        # ensures that when p closes its handle for the writable end,
        # wait() will promptly report the readable end as being ready.
        w.close()

    while readers:
        for r in wait(readers):
            try:
                msg = r.recv()
            except EOFError:
                readers.remove(r)
            else:
                print(msg)
```

アドレスフォーマット

- `'AF_INET'` アドレスは `(hostname, port)` のタプルになります。*hostname* は文字列で *port* は整数です。
- `'AF_UNIX'` アドレスはファイルシステム上のファイル名の文字列です。
- An `'AF_PIPE'` address is a string of the form `r'\.\pipe{PipeName}'`. To use *Client()* to connect to a named pipe on a remote computer called *ServerName* one should use an address of the form `r'\`*ServerName*`\pipe{PipeName}'` instead.

デフォルトでは、2 つのバックスラッシュで始まる文字列は `'AF_UNIX'` よりも `'AF_PIPE'` として推測され

ることに注意してください。

認証キー

Connection.recv を使用するとき、データは自動的に unpickle されて受信します。信頼できない接続元からのデータを unpickle することはセキュリティリスクがあります。そのため *Listener* や *Client()* はダイジェスト認証を提供するために *hmac* モジュールを使用します。

認証キーはパスワードとして見なされるバイト文字列です。コネクションが確立すると、双方の終点で正しい接続先であることを証明するために 知っているお互いの認証キーを要求します。(双方の終点が同じキーを使用して通信しようとしても、コネクション上でそのキーを送信することは **できません**。)

認証が要求されているにもかかわらず認証キーが指定されていない場合 current_process().authkey の返す値が使用されます。(詳細は *Process* を参照してください。) この値はカレントプロセスを作成する *Process* オブジェクトによって自動的に継承されます。これは (デフォルトでは) 複数プロセスのプログラムの全プロセスが相互にコネクションを 確立するときに使用される 1 つの認証キーを共有することを意味します。

適当な認証キーを *os.urandom()* を使用して生成することもできます。

ログ記録

ロギングのためにいくつかの機能が利用可能です。しかし *logging* パッケージは、(ハンドラー種別に依存して) 違うプロセスからのメッセージがごちゃ混ぜになるので、プロセスの共有ロックを使用しないことに注意してください。

multiprocessing.get_logger()

multiprocessing が使用するロガーを返します。必要に応じて新たなロガーを作成します。

最初に作成するとき、ロガーはレベルに logging.NOTSET が設定されていてデフォルトハンドラーがありません。このロガーへ送られるメッセージはデフォルトではルートロガーへ伝播されません。

Windows 上では子プロセスが親プロセスのロガーレベルを継承しないことに注意してください。さらにその他のロガーのカスタマイズ内容もすべて継承されません。

multiprocessing.log_to_stderr()

この関数は *get_logger()* に対する呼び出しを実行しますが、get_logger によって作成されるロガーを返すことに加えて、'[%(levelname)s/%(processName)s] %(message)s' のフォーマットを使用して *sys.stderr* へ出力を送るハンドラーを追加します。

以下にロギングを有効にした例を紹介します:

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
```

(次のページに続く)

(前のページからの続き)

```
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pymp-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

完全なロギングレベルの表については *logging* モジュールを参照してください。

multiprocessing.dummy モジュール

multiprocessing.dummy は *multiprocessing* の API を複製しますが *threading* モジュールのラッパーでしかありません。

In particular, the `Pool` function provided by *multiprocessing.dummy* returns an instance of *ThreadPool*, which is a subclass of *Pool* that supports all the same method calls but uses a pool of worker threads rather than worker processes.

class multiprocessing.pool.**ThreadPool**([*processes*[, *initializer*[, *initargs*]]])

A thread pool object which controls a pool of worker threads to which jobs can be submitted. *ThreadPool* instances are fully interface compatible with *Pool* instances, and their resources must also be properly managed, either by using the pool as a context manager or by calling *close()* and *terminate()* manually.

processes は使用するワーカースレッドの数です。*processes* が `None` の場合 *os.cpu_count()* が返す値を使用します。

initializer が `None` ではない場合、各ワーカープロセスは開始時に `initializer(*initargs)` を呼び出します。

Unlike *Pool*, *maxtasksperchild* and *context* cannot be provided.

注釈: A *ThreadPool* shares the same interface as *Pool*, which is designed around a pool of processes and predates the introduction of the *concurrent.futures* module. As such, it inherits some operations that don't make sense for a pool backed by threads, and it has its own type for representing the status of asynchronous jobs, *AsyncResult*, that is not understood by any other libraries.

Users should generally prefer to use *concurrent.futures.ThreadPoolExecutor*, which has a simpler interface that was designed around threads from the start, and which returns *concurrent.futures.Future* instances that are compatible with many other libraries, including *asyncio*.

17.2.3 プログラミングガイドライン

multiprocessing を使用するときに守るべき一定のガイドラインとイディオムを挙げます。

すべての開始方式について

以下はすべての開始方式に当てはまります。

共有状態を避ける

できるだけプロセス間で巨大なデータを移動することは避けるようにすべきです。

プロセス間の通信には、*threading* モジュールの低レベルな同期プリミティブを使うのではなく、キューやパイプを使うのが良いでしょう。

pickle 化の可能性

プロキシのメソッドへの引数は、pickle 化できるものにしてください。

プロキシのスレッドセーフ性

1 つのプロキシオブジェクトは、ロックで保護しないかぎり、2 つ以上のスレッドから使用してはいけません。

(異なるプロセスで **同じ** プロキシを使用することは問題ではありません。)

ゾンビプロセスを join する

Unix 上ではプロセスが終了したときに join しないと、そのプロセスはゾンビになります。新たなプロセスが開始する (または *active_children()* が呼ばれる) ときに、join されていないすべての完了プロセスが join されるので、あまり多くにはならないでしょう。また、終了したプロセスの *Process.is_alive* はそのプロセスを join します。そうは言っても、自分で開始したすべてのプロセスを明示的に join することはおそらく良いプラクティスです。

pickle/unpickle より継承する方が良い

開始方式に *spawn* あるいは *forkserver* を使用している場合、*multiprocessing* から多くの型を pickle 化する必要があるため子プロセスはそれらを使うことができます。しかし、一般にパイプやキューを使用して共有オブジェクトを他のプロセスに送信することは避けるべきです。代わりに、共有リソースにアクセスする必要のあるプロセスは上位プロセスからそれらを継承するようにすべきです。

プロセスの強制終了を避ける

あるプロセスを停止するために *Process.terminate* メソッドを使用すると、そのプロセスが現在使用されている (ロック、セマフォ、パイプやキューのような) 共有リソースを破壊したり他のプロセスから利用できない状態を引き起こし易いです。

そのため、共有リソースを使用しないプロセスでのみ *Process.terminate* を使用することを考慮することがおそらく最善の方法です。

キューを使用するプロセスを join する

キューに要素を追加するプロセスは、すべてのバッファーされた要素が "feeder" スレッドによって下位層のパイプに対してフィードされるまで終了を待つということを覚えておいてください。(子プロセスはこの動作を避けるためにキューの *Queue.cancel_join_thread* メソッドを呼ぶことができます。)

これはキューを使用するときに、キューに追加されたすべての要素が最終的にそのプロセスが join される前に削除されていることを確認する必要があることを意味します。そうしないと、そのキューに要素が追加したプロセスの終了を保証できません。デーモンではないプロセスは自動的に join されることも覚えておいてください。

次の例はデッドロックを引き起こします:

```
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()                    # this deadlocks
    obj = queue.get()
```

修正するには最後の 2 行を入れ替えます (または単純に `p.join()` の行を削除します)。

明示的に子プロセスへリソースを渡す

Unix で開始方式に *fork* を使用している場合、子プロセスはグローバルリソースを使用した親プロセス内で作成された共有リソースを使用できます。しかし、オブジェクトを子プロセスのコンストラクターに引数として渡すべきです。

Windows や他の開始方式と (将来的にでも) 互換性のあるコードを書く場合は別として、これは子プロセスが実行中である限りは親プロセス内でオブジェクトがガベージコレクトされないことも保証します。これは親プロセス内でオブジェクトがガベージコレクトされたときに一部のリソースが開放されてしまう場合に重要かもしれません。

そのため、例えば

```
from multiprocessing import Process, Lock

def f():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

は、次のように書き直すべきです

```
from multiprocessing import Process, Lock

def f(l):
    ... do something using "l" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

sys.stdin を file-like オブジェクトに置き換えることに注意する

multiprocessing は元々無条件に:

```
os.close(sys.stdin.fileno())
```

を multiprocessing.Process._bootstrap() メソッドの中で呼び出していました --- これはプロセス内プロセス (processes-in-processes) で問題が起こしてしまいます。そこで、これは以下のように変更されました:

```
sys.stdin.close()
sys.stdin = open(os.open(os.devnull, os.O_RDONLY), closefd=False)
```

これによってプロセス同士が衝突して bad file descripter エラーを起こすという根本的な問題は解決しましたが、アプリケーションの出力バッファーを *sys.stdin()* から "file-like オブジェクト" に置き換えるという潜在的危険を持ち込んでしまいました。危険というのは、複数のプロセスが file-like オブジェクトの *close()* を呼び出すと、オブジェクトに同じデータが何度もフラッシュされ、破損してしまう可能性がある、というものです。

もし file-like オブジェクトを書いて独自のキャッシュを実装するなら、キャッシュするときに常に pid を記録しておき、pid が変わったらキュッシュを捨てることで、フォークセーフにできます。例:

```
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

より詳しい情報は bpo-5155 、bpo-5313 、bpo-5331 を見てください

開始方式が *spawn* および *forkserver* の場合

開始方式に *fork* を適用しない場合にいくつかの追加の制限事項があります。

さらなる pickle 化の可能性

> `Process.__init__()` へのすべての引数は pickle 化できることを確認してください。また *Process* をサブクラス化する場合、そのインスタンスが *Process.start* メソッドが呼ばれたときに pickle 化できるようにしてください。

グローバル変数

> 子プロセスで実行されるコードがグローバル変数にアクセスしようとする場合、子プロセスが見るその値は *Process.start* が呼ばれたときの親プロセスの値と同じではない可能性があります。
>
> しかし、単にモジュールレベルの定数であるグローバル変数なら問題にはなりません。

メインモジュールの安全なインポート

> 新たな Python インタプリタによるメインモジュールのインポートが、意図しない副作用 (新たなプロセスを開始する等) を起こさずできるようにしてください。
>
> 例えば、開始方式に *spawn* あるいは *forkserver* を使用した場合に以下のモジュールを実行すると *RuntimeError* で失敗します:
>
> ```
> from multiprocessing import Process
>
> def foo():
>     print('hello')
>
> p = Process(target=foo)
> p.start()
> ```
>
> 代わりに、次のように `if __name__ == '__main__':` を使用してプログラムの "エントリポイント" を保護すべきです:
>
> ```
> from multiprocessing import Process, freeze_support, set_start_method
>
> def foo():
>     print('hello')
>
> if __name__ == '__main__':
>     freeze_support()
>     set_start_method('spawn')
>     p = Process(target=foo)
>     p.start()
> ```
>
> (プログラムをフリーズせずに通常通り実行するなら `freeze_support()` 行は取り除けます。)
>
> これは新たに生成された Python インタープリターがそのモジュールを安全にインポートして、モジュールの `foo()` 関数を実行します。
>
> プールまたはマネージャーがメインモジュールで作成される場合に似たような制限が適用されます。

17.2.4 使用例

カスタマイズされたマネージャーやプロキシの作成方法と使用方法を紹介します:

```
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo:
    def f(self):
        print('you called Foo.f()')
    def g(self):
        print('you called Foo.g()')
    def _h(self):
        print('you called Foo._h()')

# A simple generator function
def baz():
    for i in range(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ['__next__']
    def __iter__(self):
        return self
    def __next__(self):
        return self._callmethod('__next__')

# Function to return the operator module
def get_operator_module():
    return operator

##

class MyManager(BaseManager):
    pass

# register the Foo class; make `f()` and `g()` accessible via proxy
MyManager.register('Foo1', Foo)

# register the Foo class; make `g()` and `_h()` accessible via proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

# register the generator function baz; use `GeneratorProxy` to make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions accessible via proxy
MyManager.register('operator', get_operator_module)
```

(次のページに続く)

(前のページからの続き)

```
##

def test():
    manager = MyManager()
    manager.start()

    print('-' * 20)

    f1 = manager.Foo1()
    f1.f()
    f1.g()
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print('-' * 20)

    f2 = manager.Foo2()
    f2.g()
    f2._h()
    assert not hasattr(f2, 'f')
    assert sorted(f2._exposed_) == sorted(['g', '_h'])

    print('-' * 20)

    it = manager.baz()
    for i in it:
        print('<%d>' % i, end=' ')
    print()

    print('-' * 20)

    op = manager.operator()
    print('op.add(23, 45) =', op.add(23, 45))
    print('op.pow(2, 94) =', op.pow(2, 94))
    print('op._exposed_ =', op._exposed_)

##

if __name__ == '__main__':
    freeze_support()
    test()
```

Pool を使用する例です:

```
import multiprocessing
import time
import random
import sys

#
# Functions used by test code
```

(次のページに続く)

(前のページからの続き)

```
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % (
        multiprocessing.current_process().name,
        func.__name__, args, result
        )

def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5 * random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5 * random.random())
    return a + b

def f(x):
    return 1.0 / (x - 5.0)

def pow3(x):
    return x ** 3

def noop(x):
    pass

#
# Test code
#

def test():
    PROCESSES = 4
    print('Creating pool with %d processes\n' % PROCESSES)

    with multiprocessing.Pool(PROCESSES) as pool:
        #
        # Tests
        #

        TASKS = [(mul, (i, 7)) for i in range(10)] + \
                [(plus, (i, 8)) for i in range(10)]

        results = [pool.apply_async(calculate, t) for t in TASKS]
        imap_it = pool.imap(calculatestar, TASKS)
        imap_unordered_it = pool.imap_unordered(calculatestar, TASKS)

        print('Ordered results using pool.apply_async():')
        for r in results:
```

(次のページに続く)

(前のページからの続き)

```
            print('\t', r.get())
        print()

        print('Ordered results using pool.imap():')
        for x in imap_it:
            print('\t', x)
        print()

        print('Unordered results using pool.imap_unordered():')
        for x in imap_unordered_it:
            print('\t', x)
        print()

        print('Ordered results using pool.map() --- will block till complete:')
        for x in pool.map(calculatestar, TASKS):
            print('\t', x)
        print()

        #
        # Test error handling
        #

        print('Testing error handling:')

        try:
            print(pool.apply(f, (5,)))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from pool.apply()')
        else:
            raise AssertionError('expected ZeroDivisionError')

        try:
            print(pool.map(f, list(range(10))))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from pool.map()')
        else:
            raise AssertionError('expected ZeroDivisionError')

        try:
            print(list(pool.imap(f, list(range(10)))))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from list(pool.imap())')
        else:
            raise AssertionError('expected ZeroDivisionError')

        it = pool.imap(f, list(range(10)))
        for i in range(10):
            try:
                x = next(it)
            except ZeroDivisionError:
                if i == 5:
```

(次のページに続く)

(前のページからの続き)

```
                    pass
            except StopIteration:
                break
            else:
                if i == 5:
                    raise AssertionError('expected ZeroDivisionError')

        assert i == 9
        print('\tGot ZeroDivisionError as expected from IMapIterator.next()')
        print()

        #
        # Testing timeouts
        #

        print('Testing ApplyResult.get() with timeout:', end=' ')
        res = pool.apply_async(calculate, TASKS[0])
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % res.get(0.02))
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()

        print('Testing IMapIterator.next() with timeout:', end=' ')
        it = pool.imap(calculatestar, TASKS)
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % it.next(0.02))
            except StopIteration:
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()


if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()
```

ワーカープロセスのコレクションに対してタスクをフィードしてその結果をまとめるキューの使い方の例を紹介します:

```
import time
import random
```

(次のページに続く)

(前のページからの続き)

```
from multiprocessing import Process, Queue, current_process, freeze_support

#
# Function run by worker processes
#

def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

#
# Function used to calculate result
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % \
        (current_process().name, func.__name__, args, result)

#
# Functions referenced by tasks
#

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

#
#
#

def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # Create queues
    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)

    # Start worker processes
    for i in range(NUMBER_OF_PROCESSES):
```

(次のページに続く)

(前のページからの続き)

```
        Process(target=worker, args=(task_queue, done_queue)).start()

    # Get and print results
    print('Unordered results:')
    for i in range(len(TASKS1)):
        print('\t', done_queue.get())

    # Add more tasks using `put()`
    for task in TASKS2:
        task_queue.put(task)

    # Get and print some more results
    for i in range(len(TASKS2)):
        print('\t', done_queue.get())

    # Tell child processes to stop
    for i in range(NUMBER_OF_PROCESSES):
        task_queue.put('STOP')


if __name__ == '__main__':
    freeze_support()
    test()
```

17.3 multiprocessing.shared_memory --- 異なるプロセスから参照可能な共有メモリ

ソースコード Lib/multiprocessing/shared_memory.py

バージョン 3.8 で追加.

このモジュールは、マルチコアもしくは対称型マルチプロセッサ (SMP) を持つマシン上で実行される、ひとつ以上のプロセスから参照される共有メモリの確保、管理を行うための:class:*SharedMemory* クラスを提供します。共有メモリのライフサイクル管理、とりわけ複数のプロセスにわたる場合のために、`multiprocessing.managers` モジュールで、*BaseManager* サブクラスと:class:*SharedMemoryManager* が提供されます。

In this module, shared memory refers to "System V style" shared memory blocks (though is not necessarily implemented explicitly as such) and does not refer to "distributed shared memory". This style of shared memory permits distinct processes to potentially read and write to a common (or shared) region of volatile memory. Processes are conventionally limited to only have access to their own process memory space but shared memory permits the sharing of data between processes, avoiding the need to instead send messages between processes containing that data. Sharing data directly via memory can provide significant performance benefits compared to sharing data via disk or socket or other communications requiring the serialization/deserialization and copying of data.

class multiprocessing.shared_memory.**SharedMemory**(*name=None*, *create=False*, *size=0*)

Creates a new shared memory block or attaches to an existing shared memory block. Each shared memory block is assigned a unique name. In this way, one process can create a shared memory block with a particular name and a different process can attach to that same shared memory block using that same name.

As a resource for sharing data across processes, shared memory blocks may outlive the original process that created them. When one process no longer needs access to a shared memory block that might still be needed by other processes, the *close()* method should be called. When a shared memory block is no longer needed by any process, the *unlink()* method should be called to ensure proper cleanup.

name is the unique name for the requested shared memory, specified as a string. When creating a new shared memory block, if `None` (the default) is supplied for the name, a novel name will be generated.

create controls whether a new shared memory block is created (`True`) or an existing shared memory block is attached (`False`).

size specifies the requested number of bytes when creating a new shared memory block. Because some platforms choose to allocate chunks of memory based upon that platform's memory page size, the exact size of the shared memory block may be larger or equal to the size requested. When attaching to an existing shared memory block, the `size` parameter is ignored.

close()

Closes access to the shared memory from this instance. In order to ensure proper cleanup of resources, all instances should call `close()` once the instance is no longer needed. Note that calling `close()` does not cause the shared memory block itself to be destroyed.

unlink()

Requests that the underlying shared memory block be destroyed. In order to ensure proper cleanup of resources, `unlink()` should be called once (and only once) across all processes which have need for the shared memory block. After requesting its destruction, a shared memory block may or may not be immediately destroyed and this behavior may differ across platforms. Attempts to access data inside the shared memory block after `unlink()` has been called may result in memory access errors. Note: the last process relinquishing its hold on a shared memory block may call `unlink()` and *close()* in either order.

buf

A memoryview of contents of the shared memory block.

name

Read-only access to the unique name of the shared memory block.

size

Read-only access to size in bytes of the shared memory block.

The following example demonstrates low-level use of *SharedMemory* instances:

```
>>> from multiprocessing import shared_memory
>>> shm_a = shared_memory.SharedMemory(create=True, size=10)
>>> type(shm_a.buf)
<class 'memoryview'>
>>> buffer = shm_a.buf
>>> len(buffer)
10
>>> buffer[:4] = bytearray([22, 33, 44, 55])  # Modify multiple at once
>>> buffer[4] = 100                           # Modify single byte at a time
>>> # Attach to an existing shared memory block
>>> shm_b = shared_memory.SharedMemory(shm_a.name)
>>> import array
>>> array.array('b', shm_b.buf[:5])  # Copy the data into a new array.array
array('b', [22, 33, 44, 55, 100])
>>> shm_b.buf[:5] = b'howdy'  # Modify via shm_b using bytes
>>> bytes(shm_a.buf[:5])      # Access via shm_a
b'howdy'
>>> shm_b.close()   # Close each SharedMemory instance
>>> shm_a.close()
>>> shm_a.unlink()  # Call unlink only once to release the shared memory
```

The following example demonstrates a practical use of the *SharedMemory* class with NumPy arrays, accessing the same `numpy.ndarray` from two distinct Python shells:

```
>>> # In the first Python interactive shell
>>> import numpy as np
>>> a = np.array([1, 1, 2, 3, 5, 8])  # Start with an existing NumPy array
>>> from multiprocessing import shared_memory
>>> shm = shared_memory.SharedMemory(create=True, size=a.nbytes)
>>> # Now create a NumPy array backed by shared memory
>>> b = np.ndarray(a.shape, dtype=a.dtype, buffer=shm.buf)
>>> b[:] = a[:]  # Copy the original data into shared memory
>>> b
array([1, 1, 2, 3, 5, 8])
>>> type(b)
<class 'numpy.ndarray'>
>>> type(a)
<class 'numpy.ndarray'>
>>> shm.name  # We did not specify a name so one was chosen for us
'psm_21467_46075'

>>> # In either the same shell or a new Python shell on the same machine
>>> import numpy as np
>>> from multiprocessing import shared_memory
>>> # Attach to the existing shared memory block
>>> existing_shm = shared_memory.SharedMemory(name='psm_21467_46075')
>>> # Note that a.shape is (6,) and a.dtype is np.int64 in this example
>>> c = np.ndarray((6,), dtype=np.int64, buffer=existing_shm.buf)
>>> c
array([1, 1, 2, 3, 5, 8])
>>> c[-1] = 888
```

(次のページに続く)

(前のページからの続き)

```
>>> c
array([  1,   1,   2,   3,   5, 888])

>>> # Back in the first Python interactive shell, b reflects this change
>>> b
array([  1,   1,   2,   3,   5, 888])

>>> # Clean up from within the second Python shell
>>> del c  # Unnecessary; merely emphasizing the array is no longer used
>>> existing_shm.close()

>>> # Clean up from within the first Python shell
>>> del b  # Unnecessary; merely emphasizing the array is no longer used
>>> shm.close()
>>> shm.unlink()  # Free and release the shared memory block at the very end
```

class multiprocessing.managers.**SharedMemoryManager**([*address*[, *authkey*]])

A subclass of *BaseManager* which can be used for the management of shared memory blocks across processes.

A call to *start()* on a *SharedMemoryManager* instance causes a new process to be started. This new process's sole purpose is to manage the life cycle of all shared memory blocks created through it. To trigger the release of all shared memory blocks managed by that process, call *shutdown()* on the instance. This triggers a `SharedMemory.unlink()` call on all of the *SharedMemory* objects managed by that process and then stops the process itself. By creating `SharedMemory` instances through a `SharedMemoryManager`, we avoid the need to manually track and trigger the freeing of shared memory resources.

This class provides methods for creating and returning *SharedMemory* instances and for creating a list-like object (*ShareableList*) backed by shared memory.

Refer to *multiprocessing.managers.BaseManager* for a description of the inherited *address* and *authkey* optional input arguments and how they may be used to connect to an existing `SharedMemoryManager` service from other processes.

SharedMemory(*size*)

Create and return a new *SharedMemory* object with the specified `size` in bytes.

ShareableList(*sequence*)

Create and return a new *ShareableList* object, initialized by the values from the input `sequence`.

The following example demonstrates the basic mechanisms of a `SharedMemoryManager`:

```
>>> from multiprocessing.managers import SharedMemoryManager
>>> smm = SharedMemoryManager()
>>> smm.start()  # Start the process that manages the shared memory blocks
>>> sl = smm.ShareableList(range(4))
>>> sl
```

(次のページに続く)

(前のページからの続き)

```
ShareableList([0, 1, 2, 3], name='psm_6572_7512')
>>> raw_shm = smm.SharedMemory(size=128)
>>> another_sl = smm.ShareableList('alpha')
>>> another_sl
ShareableList(['a', 'l', 'p', 'h', 'a'], name='psm_6572_12221')
>>> smm.shutdown()  # Calls unlink() on sl, raw_shm, and another_sl
```

The following example depicts a potentially more convenient pattern for using `SharedMemoryManager` objects via the `with` statement to ensure that all shared memory blocks are released after they are no longer needed:

```
>>> with SharedMemoryManager() as smm:
...     sl = smm.ShareableList(range(2000))
...     # Divide the work among two processes, storing partial results in sl
...     p1 = Process(target=do_work, args=(sl, 0, 1000))
...     p2 = Process(target=do_work, args=(sl, 1000, 2000))
...     p1.start()
...     p2.start()  # A multiprocessing.Pool might be more efficient
...     p1.join()
...     p2.join()   # Wait for all work to complete in both processes
...     total_result = sum(sl)  # Consolidate the partial results now in sl
```

When using a `SharedMemoryManager` in a `with` statement, the shared memory blocks created using that manager are all released when the `with` statement's code block finishes execution.

class multiprocessing.shared_memory.**ShareableList**(*sequence=None*, *, *name=None*)

Provides a mutable list-like object where all values stored within are stored in a shared memory block. This constrains storable values to only the `int`, `float`, `bool`, `str` (less than 10M bytes each), `bytes` (less than 10M bytes each), and `None` built-in data types. It also notably differs from the built-in `list` type in that these lists can not change their overall length (i.e. no append, insert, etc.) and do not support the dynamic creation of new *ShareableList* instances via slicing.

sequence is used in populating a new `ShareableList` full of values. Set to `None` to instead attach to an already existing `ShareableList` by its unique shared memory name.

name is the unique name for the requested shared memory, as described in the definition for *SharedMemory*. When attaching to an existing `ShareableList`, specify its shared memory block's unique name while leaving `sequence` set to `None`.

count(*value*)

`value` が出現する回数を返します。

index(*value*)

Returns first index position of `value`. Raises *ValueError* if `value` is not present.

format

Read-only attribute containing the *struct* packing format used by all currently stored values.

shm

The *SharedMemory* instance where the values are stored.

以下に *ShareableList* インスタンスの利用例を示します。

```
>>> from multiprocessing import shared_memory
>>> a = shared_memory.ShareableList(['howdy', b'HoWdY', -273.154, 100, None, True, 42])
>>> [ type(entry) for entry in a ]
[<class 'str'>, <class 'bytes'>, <class 'float'>, <class 'int'>, <class 'NoneType'>, <class
↪'bool'>, <class 'int'>]
>>> a[2]
-273.154
>>> a[2] = -78.5
>>> a[2]
-78.5
>>> a[2] = 'dry ice'  # Changing data types is supported as well
>>> a[2]
'dry ice'
>>> a[2] = 'larger than previously allocated storage space'
Traceback (most recent call last):
  ...
ValueError: exceeds available storage for existing str
>>> a[2]
'dry ice'
>>> len(a)
7
>>> a.index(42)
6
>>> a.count(b'howdy')
0
>>> a.count(b'HoWdY')
1
>>> a.shm.close()
>>> a.shm.unlink()
>>> del a  # Use of a ShareableList after call to unlink() is unsupported
```

The following example depicts how one, two, or many processes may access the same *ShareableList* by supplying the name of the shared memory block behind it:

```
>>> b = shared_memory.ShareableList(range(5))         # In a first process
>>> c = shared_memory.ShareableList(name=b.shm.name)  # In a second process
>>> c
ShareableList([0, 1, 2, 3, 4], name='...')
>>> c[-1] = -999
>>> b[-1]
-999
>>> b.shm.close()
>>> c.shm.close()
>>> c.shm.unlink()
```

17.4 concurrent パッケージ

現在のところ、このパッケージにはモジュールが 1 つだけあります:

- *concurrent.futures* -- 並列タスク実行

17.5 concurrent.futures -- 並列タスク実行

バージョン 3.2 で追加.

ソースコード: Lib/concurrent/futures/thread.py および Lib/concurrent/futures/process.py

---

concurrent.futures モジュールは、非同期に実行できる呼び出し可能オブジェクトの高水準のインタフェースを提供します。

非同期実行は *ThreadPoolExecutor* を用いてスレッドで実行することも、*ProcessPoolExecutor* を用いて別々のプロセスで実行することもできます. どちらも *Executor* 抽象クラスで定義された同じインターフェースを実装します。

17.5.1 Executor オブジェクト

class concurrent.futures.**Executor**

非同期呼び出しを実行するためのメソッドを提供する抽象クラスです。このクラスを直接使ってはならず、具象サブクラスを介して使います。

submit(*fn*, **args*, ***kwargs*)

呼び出し可能オブジェクト *fn* を、`fn(*args **kwargs)` として実行するようにスケジュールし、呼び出し可能オブジェクトの実行を表現する *Future* オブジェクトを返します。

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

map(*func*, **iterables*, *timeout=None*, *chunksize=1*)

Similar to *map(func, *iterables)* except:

- the *iterables* are collected immediately rather than lazily;
- *func* is executed asynchronously and several calls to *func* may be made concurrently.

返されるイテレータは *concurrent.futures.TimeoutError* を送出します。*timeout* は整数または浮動小数点数です。もし *timeout* が指定されないか の場合、待ち時間に制限はありません。

もし *func* の呼び出しが例外を送出した場合、その例外はイテレータから値を受け取る時に送出されます。

When using *ProcessPoolExecutor*, this method chops *iterables* into a number of chunks which it submits to the pool as separate tasks. The (approximate) size of these chunks

can be specified by setting *chunksize* to a positive integer. For very long iterables, using a large value for *chunksize* can significantly improve performance compared to the default size of 1. With *ThreadPoolExecutor*, *chunksize* has no effect.

バージョン 3.5 で変更: *chunksize* 引数が追加されました。

shutdown(*wait=True*)

executor に対して、現在保留中のフューチャーが実行された後で、使用中のすべての資源を解放するように伝えます。シャットダウンにより後に *Executor.submit()* と *Executor.map()* を呼び出すと *RuntimeError* が送出されます。

wait が `True` の場合、すべての未完了のフューチャの実行が完了して Executor に関連付けられたリソースが解放されるまで、このメソッドは返りません。*wait* が `False` の場合、このメソッドはすぐに返り、すべての未完了のフューチャの実行が完了したときに、Executor に関連付けられたリソースが解放されます。*wait* の値に関係なく、すべての未完了のフューチャの実行が完了するまで Python プログラム全体は終了しません。

`with` 文を使用することで、このメソッドを明示的に呼ばないようにできます。`with` 文は *Executor* をシャットダウンします (*wait* を `True` にセットして *Executor.shutdown()* が呼ばれたかのように待ちます)。

```
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

17.5.2 ThreadPoolExecutor

ThreadPoolExecutor はスレッドのプールを使用して非同期に呼び出しを行う、*Executor* のサブクラスです。

Future に関連づけられた呼び出し可能オブジェクトが、別の *Future* の結果を待つ時にデッドロックすることがあります。例:

```
import time
def wait_on_b():
    time.sleep(5)
    print(b.result())  # b will never complete because it is waiting on a.
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result())  # a will never complete because it is waiting on b.
    return 6
```

(次のページに続く)

(前のページからの続き)

```
executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
```

以下でも同様です:

```
def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # This will never complete because there is only one worker thread and
    # it is executing this function.
    print(f.result())

executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```

class concurrent.futures.**ThreadPoolExecutor**(*max_workers=None*, *thread_name_prefix=''*, *initializer=None*, *initargs=()*)

最大で *max_workers* 個のスレッドを非同期実行に使う *Executor* のサブクラスです。

initializer is an optional callable that is called at the start of each worker thread; *initargs* is a tuple of arguments passed to the initializer. Should *initializer* raise an exception, all currently pending jobs will raise a *BrokenThreadPool*, as well as any attempt to submit more jobs to the pool.

バージョン 3.5 で変更: *max_workers* が `None` か指定されない場合のデフォルト値はマシンのプロセッサの数に 5 を掛けたものになります。これは、*ThreadPoolExecutor* は CPU の処理ではなく I/O をオーバーラップするのによく使用されるため、*ProcessPoolExecutor* のワーカーの数よりもこのワーカーの数を増やすべきであるという想定に基づいています。

バージョン 3.6 で追加: *thread_name_prefix* 引数が追加され、デバッグしやすくなるようにプールから作られたワーカスレッド *threading.Thread* の名前を管理できるようになりました。

バージョン 3.7 で変更: *initializer* と *initargs* 引数が追加されました。

バージョン 3.8 で変更: Default value of *max_workers* is changed to `min(32, os.cpu_count() + 4)`. This default value preserves at least 5 workers for I/O bound tasks. It utilizes at most 32 CPU cores for CPU bound tasks which release the GIL. And it avoids using very large resources implicitly on many-core machines.

ThreadPoolExecutor now reuses idle worker threads before starting *max_workers* worker threads too.

ThreadPoolExecutor の例

```
import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://nonexistant-subdomain.python.org/']

# Retrieve a single page and report the URL and contents
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

# We can use a with statement to ensure threads are cleaned up promptly
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    # Start the load operations and mark each future with its URL
    future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
        else:
            print('%r page is %d bytes' % (url, len(data)))
```

17.5.3 ProcessPoolExecutor

ProcessPoolExecutor はプロセスプールを使って非同期呼び出しを実施する *Executor* のサブクラスです。*ProcessPoolExecutor* は *multiprocessing* モジュールを利用します。このため *Global Interpreter Lock* を回避することができますが、pickle 化できるオブジェクトしか実行したり返したりすることができません。

`__main__` モジュールはワーカサブプロセスでインポート可能でなければなりません。すなわち、*ProcessPoolExecutor* は対話的インタープリタでは動きません。

ProcessPoolExecutor に渡された呼び出し可能オブジェクトから *Executor* や *Future* メソッドを呼ぶとデッドロックに陥ります。

class concurrent.futures.**ProcessPoolExecutor**(*max_workers=None*, *mp_context=None*, *initializer=None*, *initargs=()*)

An *Executor* subclass that executes calls asynchronously using a pool of at most *max_workers* processes. If *max_workers* is `None` or not given, it will default to the number of processors on the machine. If *max_workers* is less than or equal to `0`, then a *ValueError* will be raised. On Windows, *max_workers* must be less than or equal to `61`. If it is not then *ValueError* will be raised. If *max_workers* is `None`, then the default chosen will be at most `61`, even if more processors are available. *mp_context* can be a multiprocessing context or None. It will be used to launch the

workers. If *mp_context* is `None` or not given, the default multiprocessing context is used.

initializer is an optional callable that is called at the start of each worker process; *initargs* is a tuple of arguments passed to the initializer. Should *initializer* raise an exception, all currently pending jobs will raise a *BrokenProcessPool*, as well as any attempt to submit more jobs to the pool.

バージョン 3.3 で変更: ワーカプロセスの 1 つが突然終了した場合、`BrokenProcessPool` エラーが送出されるようになりました。以前は挙動は未定義でしたが、executor や futures がフリーズしたりデッドロックを起こすことがしばしばでした。

バージョン 3.7 で変更: The *mp_context* argument was added to allow users to control the start_method for worker processes created by the pool.

initializer と *initargs* 引数が追加されました。

ProcessPoolExecutor の例

```
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print('%d is prime: %s' % (number, prime))

if __name__ == '__main__':
    main()
```

17.5.4 Future オブジェクト

Future クラスは呼び出し可能オブジェクトの非同期実行をカプセル化します。*Future* のインスタンスは *Executor.submit()* によって生成されます。

class concurrent.futures.Future

呼び出し可能オブジェクトの非同期実行をカプセル化します。*Future* インスタンスは *Executor.submit()* で生成され、テストを除いて直接生成すべきではありません。

cancel()

Attempt to cancel the call. If the call is currently being executed or finished running and cannot be cancelled then the method will return `False`, otherwise the call will be cancelled and the method will return `True`.

cancelled()

呼び出しが正常にキャンセルされた場合 `True` を返します。

running()

現在呼び出しが実行中でキャンセルできない場合 `True` を返します。

done()

呼び出しが正常にキャンセルされたか終了した場合 `True` を返します。

result(*timeout=None*)

呼び出しによって返された値を返します。呼び出しがまだ完了していない場合、このメソッドは *timeout* 秒の間待機します。呼び出しが *timeout* 秒間の間に完了しない場合、*concurrent.futures.TimeoutError* が送出されます。*timeout* には int か float を指定できます。*timeout* が指定されていないか、`None` である場合、待機時間に制限はありません。

future が完了する前にキャンセルされた場合 *CancelledError* が送出されます。

呼び出しが例外を送出した場合、このメソッドは同じ例外を送出します。

exception(*timeout=None*)

呼び出しによって送出された例外を返します。呼び出しがまだ完了していない場合、このメソッドは *timeout* 秒だけ待機します。呼び出しが *timeout* 秒の間に完了しない場合、*concurrent.futures.TimeoutError* が送出されます。*timeout* には int か float を指定できます。*timeout* が指定されていないか、`None` である場合、待機時間に制限はありません。

future が完了する前にキャンセルされた場合 *CancelledError* が送出されます。

呼び出しが例外を送出することなく完了した場合、`None` を返します。

add_done_callback(*fn*)

呼び出し可能な *fn* オブジェクトを future にアタッチします。future がキャンセルされたか、実行を終了した際に、future をそのただ一つの引数として *fn* が呼び出されます。

追加された呼び出し可能オブジェクトは、追加された順番で呼びだされ、追加を行ったプロセスに属するスレッド中で呼び出されます。もし呼び出し可能オブジェクトが *Exception* のサ

ブクラスを送出した場合、それはログに記録され無視されます。呼び出し可能オブジェクトが *BaseException* のサブクラスを送出した場合の動作は未定義です。

もし future がすでに完了しているか、キャンセル済みであれば、*fn* は即座に実行されます。

以下の *Future* メソッドは、ユニットテストでの使用と *Executor* を実装することを意図しています。

set_running_or_notify_cancel()

このメソッドは、*Future* に関連付けられたワークやユニットテストによるワークの実行前に、*Executor* の実装によってのみ呼び出してください。

このメソッドが `False` を返す場合、*Future* はキャンセルされています。つまり、*Future.cancel()* が呼び出されて *True* が返っています。*Future* の完了を (*as_completed()* または *wait()* により) 待機するすべてのスレッドが起動します。

このメソッドが `True` を返す場合、*Future* はキャンセルされて、実行状態に移行されています。つまり、*Future.running()* を呼び出すと *True* が返ります。

このメソッドは、一度だけ呼び出すことができ、*Future.set_result()* または *Future.set_exception()* がキャンセルされた後には呼び出すことができません。

set_result(*result*)

Future に関連付けられたワークの結果を *result* に設定します。

このメソッドは、*Executor* の実装またはユニットテストによってのみ使用してください。

バージョン 3.8 で変更: This method raises *concurrent.futures.InvalidStateError* if the *Future* is already done.

set_exception(*exception*)

Future に関連付けられたワークの結果を *Exception* *exception* に設定します。

このメソッドは、*Executor* の実装またはユニットテストによってのみ使用してください。

バージョン 3.8 で変更: This method raises *concurrent.futures.InvalidStateError* if the *Future* is already done.

17.5.5 モジュール関数

concurrent.futures.wait(*fs*, *timeout=None*, *return_when=ALL_COMPLETED*)

Wait for the *Future* instances (possibly created by different *Executor* instances) given by *fs* to complete. Returns a named 2-tuple of sets. The first set, named `done`, contains the futures that completed (finished or cancelled futures) before the wait completed. The second set, named `not_done`, contains the futures that did not complete (pending or running futures).

timeout で結果を返すまで待機する最大秒数を指定できます。*timeout* は整数か浮動小数点数をとります。*timeout* が指定されないか `None` の場合、無期限に待機します。

return_when でこの関数がいつ結果を返すか指定します。指定できる値は以下の 定数のどれか一つです:

定数	説明
FIRST_COMPLETED	いずれかのフューチャが終了したかキャンセルされたときに返します。
FIRST_EXCEPTION	いずれかのフューチャが例外の送出で終了した場合に返します。例外を送出したフューチャがない場合は、ALL_COMPLETED と等価になります。
ALL_COMPLETED	すべてのフューチャが終了したかキャンセルされたときに返します。

concurrent.futures.**as_completed**(*fs*, *timeout=None*)

Returns an iterator over the *Future* instances (possibly created by different *Executor* instances) given by *fs* that yields futures as they complete (finished or cancelled futures). Any futures given by *fs* that are duplicated will be returned once. Any futures that completed before *as_completed()* is called will be yielded first. The returned iterator raises a *concurrent.futures.TimeoutError* if *__next__()* is called and the result isn't available after *timeout* seconds from the original call to *as_completed()*. *timeout* can be an int or float. If *timeout* is not specified or None, there is no limit to the wait time.

参考:

PEP 3148 -- futures - execute computations asynchronously　この機能を Python 標準ライブラリに含めることを述べた提案です。

17.5.6 例外クラス

exception concurrent.futures.**CancelledError**

future がキャンセルされたときに送出されます。

exception concurrent.futures.**TimeoutError**

future の操作が与えられたタイムアウトを超過したときに送出されます。

exception concurrent.futures.**BrokenExecutor**

Derived from *RuntimeError*, this exception class is raised when an executor is broken for some reason, and cannot be used to submit or execute new tasks.

バージョン 3.7 で追加.

exception concurrent.futures.**InvalidStateError**

Raised when an operation is performed on a future that is not allowed in the current state.

バージョン 3.8 で追加.

exception concurrent.futures.thread.**BrokenThreadPool**

Derived from *BrokenExecutor*, this exception class is raised when one of the workers of a ThreadPoolExecutor has failed initializing.

バージョン 3.7 で追加.

exception concurrent.futures.process.**BrokenProcessPool**

BrokenExecutor から派生しています。この例外クラスは ProcessPoolExecutor のワーカの 1 つが

正常に終了されなかったとき (例えば外部から kill されたとき) に送出されます。

バージョン 3.3 で追加.

17.6 subprocess --- サブプロセス管理

ソースコード: Lib/subprocess.py

---

subprocess モジュールは新しいプロセスの開始、入力/出力/エラーパイプの接続、リターンコードの取得を可能とします。このモジュールは以下の古いモジュールや関数を置き換えることを目的としています：

```
os.system
os.spawn*
```

これらのモジュールや関数の代わりに、*subprocess* モジュールをどのように使うかについてを以下の節で説明します。

参考:

PEP 324 -- subprocess モジュールを提案している PEP

17.6.1 subprocess モジュールを使う

サブプロセスを起動するために推奨される方法は、すべての用法を扱える *run()* 関数を使用することです。より高度な用法では下層の *Popen* インターフェースを直接使用することもできます。

run() 関数は Python 3.5 で追加されました; 過去のバージョンとの互換性の維持が必要な場合は、**古い高水準** *API* 節をご覧ください。

subprocess.run(*args*, ***, *stdin=None*, *input=None*, *stdout=None*, *stderr=None*, *capture_output=False*, *shell=False*, *cwd=None*, *timeout=None*, *check=False*, *encoding=None*, *errors=None*, *text=None*, *env=None*, *universal_newlines=None*, ***other_popen_kwargs*)

args で指定されたコマンドを実行します。コマンドの完了を待って、*CompletedProcess* インスタンスを返します。

上記の引数は、もっともよく使われるものだけ示しており、後述の **よく使われる引数** で説明されています (そのためここではキーワード専用引数の表記に省略されています)。関数の完全な使用法を説明しても大部分が *Popen* コンストラクターの内容と同じになります - この関数のほとんどの引数は *Popen* インターフェイスに渡されます。(*timeout*、*input* および *check* は除く。)

capture_output を true に設定すると、stdout および stderr が捕捉されます。このとき、内部では `stdout=PIPE` および `stderr=PIPE` に設定された *Popen* オブジェクトが自動的に作られます。なお、*stdout* または *stderr* 引数を *capture_output* と同時に指定することはできません。両者をまとめてひとつのストリームとして捕捉したい場合は、*capture_output* を指定するのではなく、`stdout=PIPE` and `stderr=STDOUT` を指定してください。

引数 *timeout* は *Popen.communicate()* に渡されます。タイムアウトが発生すると、子プロセスは kill され、待機されます。子プロセスが中断されたあと *TimeoutExpired* が再び送出されます。

引数 *input* は *Popen.communicate()* を経由し、サブプロセスの標準入力に渡されます。これはバイト列であるか、(encoding もしくは errors が指定されているか、text 引数が true である場合は) 文字列でなければなりません。この引数が指定されると、内部で *Popen* オブジェクトが stdin=PIPE で自動的に作成され、引数 *stdin* は使用されません。

check に真を指定した場合、プロセスが非ゼロの終了コードで終了すると *CalledProcessError* 例外が送出されます。この例外の属性には、引数、終了コード、標準出力および標準エラー出力が捕捉できた場合に格納されます。

encoding または *errors* 引数が指定されるか、*text* 引数が true である場合、stdin, stdout および stderr のためのファイルオブジェクトはテキストモードでオープンされます。その際には指定された *encoding* および *errors* が使われるか、デフォルトの *io.TextIOWrapper* になります。*universal_newlines* 引数は *text* 引数と等価であり、後方互換性のために提供されています。そうでない場合、デフォルトでこれらのファイルオブジェクトはバイナリモードでオープンされます。

env が None 以外の場合、これは新しいプロセスでの環境変数を定義します。デフォルトでは、子プロセスは現在のプロセスの環境変数を引き継ぎます。*Popen* に直接渡されます。

例:

```
>>> subprocess.run(["ls", "-l"])  # doesn't capture output
CompletedProcess(args=['ls', '-l'], returncode=0)

>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1

>>> subprocess.run(["ls", "-l", "/dev/null"], capture_output=True)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n', stderr=b'')
```

バージョン 3.5 で追加.

バージョン 3.6 で変更: *encoding* と *error* が引数に追加されました。

バージョン 3.7 で変更: *universal_newlines* 引数のよりわかりやすい名前として、*text* 引数が追加されました。*capture_output* 引数が追加されました。

バージョン 3.8.17 で変更: Changed Windows shell search order for shell=True. The current directory and %PATH% are replaced with %COMSPEC% and %SystemRoot%\System32\cmd.exe. As a result, dropping a malicious program named cmd.exe into a current directory no longer works.

class subprocess.**CompletedProcess**

run() の戻り値。プロセスが終了したことを表します。

args

プロセスを起動するときに使用された引数。1 個のリストか 1 個の文字列になります。

returncode
子プロセスの終了コード。一般に、終了ステータス 0 はプロセスが正常に終了したことを示します。

負の値 -N は子プロセスがシグナル N により中止させられたことを示します (POSIX のみ)。

stdout
子プロセスから補足された標準出力です。バイトシーケンス、もしくは *run()* でエンコーディングが指定された場合、エラーの場合、text=True が指定された場合は文字列が出力されます。標準出力が補足されなかった場合は None になります。

プロセスが stderr=subprocess.STDOUT で実行された場合、標準出力と標準エラー出力が混合されたものがこの属性に格納され、*stderr* は None になります。

stderr
子プロセスから補足された標準エラー出力です。バイト列、もしくは *run()* でエンコーディングが指定された場合、エラーの場合、text=True が指定された場合は文字列です。標準エラー出力が補足できなかったら None になります。

check_returncode()
returncode が非ゼロの場合、*CalledProcessError* が送出されます。

バージョン 3.5 で追加.

subprocess.DEVNULL
Popen の *stdin*, *stdout*, *stderr* 引数に渡して、標準入出力を *os.devnull* から入出力するように指定するための特殊値です。

バージョン 3.3 で追加.

subprocess.PIPE
Popen の *stdin*, *stdout*, *stderr* 引数に渡して、標準ストリームに対するパイプを開くことを指定するための特殊値です。*Popen.communicate()* に非常に有用です。

subprocess.STDOUT
Popen の *stderr* 引数に渡して、標準エラー出力が標準出力と同じハンドルに出力されるように指定するための特殊値です。

exception subprocess.SubprocessError
このモジュールの他のすべての例外のための基底クラスです。

バージョン 3.3 で追加.

exception subprocess.TimeoutExpired
SubprocessError のサブクラスです。子プロセスの終了を待機している間にタイムアウトが発生した場合に送出されます。

cmd
子プロセスの生成に使用されるコマンド本文。

timeout
タイムアウト秒数。

output
run() または *check_output()* によって捕捉された子プロセスの出力。捕捉されなかったら None になります。

stdout
output の別名。*stderr* と対になります。

stderr
run() によって捕捉された子プロセスの標準エラー出力。捕捉されなかったら None になります。

バージョン 3.3 で追加.

バージョン 3.5 で変更: 属性 *stdout* および *stderr* が追加されました。

exception subprocess.CalledProcessError
SubprocessError のサブクラスです。*check_call()* または *check_output()* によって実行されたプロセスが非ゼロの終了ステータスを返した場合に送出されます。

returncode
子プロセスの終了ステータスです。もしプロセスがシグナルによって終了したなら、これは負のシグナル番号になります。

cmd
子プロセスの生成に使用されるコマンド本文。

output
run() または *check_output()* によって捕捉された子プロセスの出力。捕捉されなかったら None になります。

stdout
output の別名。*stderr* と対になります。

stderr
run() によって捕捉された子プロセスの標準エラー出力。捕捉されなかったら None になります。

バージョン 3.5 で変更: 属性 *stdout* および *stderr* が追加されました。

よく使われる引数

幅広い使用例をサポートするために、*Popen* コンストラクター (とその他の簡易関数) は、多くのオプション引数を受け付けます。一般的な用法については、これらの引数の多くはデフォルト値のままで問題ありません。通常必要とされる引数は以下の通りです:

args はすべての呼び出しに必要で、文字列あるいはプログラム引数のシーケンスでなければなりません。一般に、引数のシーケンスを渡す方が望ましいです。なぜなら、モジュールが必要な引数のエスケープやクオート (例えばファイル名中のスペースを許すこと) の面倒を見ることができるためです。単一の文字列を渡す場合、*shell* は *True* でなければなりません (以下を参照)。もしくは、その文字列は引数を指定せずに実行される単なるプログラムの名前でなければなりません。

stdin, *stdout* および *stderr* には、実行するプログラムの標準入力、標準出力、および標準エラー出力のファイルハンドルをそれぞれ指定します。有効な値は *PIPE*、*DEVNULL*、既存のファイル記述子 (正の整数)、既存のファイルオブジェクトおよび None です。*PIPE* を指定すると新しいパイプが子プロセスに向けて作られます。*DEVNULL* を指定すると特殊ファイル *os.devnull* が使用されます。デフォルト設定の None を指定するとリダイレクトは起こりません。子プロセスのファイルハンドルはすべて親から受け継がれます。加えて、*stderr* を *STDOUT* にすると、子プロセスの stderr からの出力は *stdout* と同じファイルハンドルに出力されます。

If *encoding* or *errors* are specified, or *text* (also known as *universal_newlines*) is true, the file objects *stdin*, *stdout* and *stderr* will be opened in text mode using the *encoding* and *errors* specified in the call or the defaults for *io.TextIOWrapper*.

stdin については、入力での行末文字 '\n' はデフォルトの行セパレーター *os.linesep* に変換されます。*stdout* と *stderr* については、出力での行末はすべて '\n' に変換されます。詳細は *io.TextIOWrapper* クラスのドキュメントでコンストラクターの引数 *newline* が None である場合を参照してください。

If text mode is not used, *stdin*, *stdout* and *stderr* will be opened as binary streams. No encoding or line ending conversion is performed.

バージョン 3.6 で追加: *encoding* と *error* が引数に追加されました。

バージョン 3.7 で追加: *universal_newlines* の別名として、*text* 引数が追加されました。

注釈: ファイルオブジェクト *Popen.stdin*、*Popen.stdout* ならびに *Popen.stderr* の改行属性は *Popen.communicate()* メソッドで更新されません。

---

shell が True なら、指定されたコマンドはシェルによって実行されます。あなたが Python を主として (ほとんどのシステムシェル以上の) 強化された制御フローのために使用していて、さらにシェルパイプ、ファイル名ワイルドカード、環境変数展開、~ のユーザーホームディレクトリへの展開のような他のシェル機能への簡単なアクセスを望むなら、これは有用かもしれません。しかしながら、Python 自身が多くのシェル的な機能の実装を提供していることに注意してください (特に *glob*, *fnmatch*, *os.walk()*, *os.path.expandvars()*, *os.path.expanduser()*, *shutil*)。

バージョン 3.3 で変更: *universal_newlines* が True の場合、クラスはエンコーディング locale.getpreferredencoding() の代わりに *locale.getpreferredencoding(False)* を使用します。この変更についての詳細は、*io.TextIOWrapper* クラスを参照してください。

注釈: shell=True を使う前に **セキュリティで考慮すべき点** を読んでください。

これらのオプションは、他のすべてのオプションとともに *Popen* コンストラクターのドキュメントの中でより詳細に説明されています。

Popen コンストラクター

このモジュールの中で、根底のプロセス生成と管理は *Popen* クラスによって扱われます。簡易関数によってカバーされないあまり一般的でないケースを開発者が扱えるように、*Popen* クラスは多くの柔軟性を提供しています。

class subprocess.Popen(*args*, *bufsize=-1*, *executable=None*, *stdin=None*, *stdout=None*, *stderr=None*, *preexec_fn=None*, *close_fds=True*, *shell=False*, *cwd=None*, *env=None*, *universal_newlines=None*, *startupinfo=None*, *creationflags=0*, *restore_signals=True*, *start_new_session=False*, *pass_fds=()*, ***, *encoding=None*, *errors=None*, *text=None*)

新しいプロセスで子のプログラムを実行します。POSIX においては、子のプログラムを実行するために、このクラスは *os.execvp()* のような挙動を使用します。Windows においては、このクラスは Windows の `CreateProcess()` 関数を使用します。*Popen* への引数は以下の通りです。

args はプログラム引数のシーケンスか、単一の文字列または *path-like* **オブジェクト** でなければなりません。デフォルトでは、*args* がシーケンスの場合に実行されるプログラムは *args* の最初の要素です。*args* が文字列の場合、解釈はプラットフォーム依存であり、下記に説明されます。デフォルトの挙動からの追加の違いについては *shell* および *executable* 引数を参照してください。特に明記されない限り、*args* をシーケンスとして渡すことが推奨されます。

An example of passing some arguments to an external program as a sequence is:

```
Popen(["/usr/bin/git", "commit", "-m", "Fixes a bug."])
```

POSIX 上では、*args* が文字列の場合、その文字列は実行すべきプログラムの名前またはパスとして解釈されます。しかし、これはプログラムに引数を渡さない場合にのみ可能です。

注釈: It may not be obvious how to break a shell command into a sequence of arguments, especially in complex cases. *shlex.split()* can illustrate how to determine the correct tokenization for *args*:

```
>>> import shlex, subprocess
>>> command_line = input()
/bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
>>> args = shlex.split(command_line)
>>> print(args)
['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd', "echo '$MONEY'
↪"]
>>> p = subprocess.Popen(args) # Success!
```

特に注意すべき点は、シェル内でスペースで区切られたオプション (*-input* など) と引数 (*eggs.txt* など) はリストの別々の要素になるのに対し、シェル内で (上記のスペースを含むファイル名や *echo* コマンドのように) クォーティングやバックスラッシュエスケープが必要なものは単一のリスト要素であることです。

Windows 上では、*args* がシーケンスなら *Windows* **における引数シーケンスから文字列への変換** に記述された方法で文字列に変換されます。これは根底の `CreateProcess()` が文字列上で動作するから

です。

バージョン 3.6 で変更: *args* parameter accepts a *path-like object* if *shell* is `False` and a sequence containing path-like objects on POSIX.

バージョン 3.8 で変更: *args* parameter accepts a *path-like object* if *shell* is `False` and a sequence containing bytes and path-like objects on Windows.

shell 引数 (デフォルトでは `False`) は、実行するプログラムとしてシェルを使用するかどうかを指定します。*shell* が `True` の場合、*args* をシーケンスとしてではなく文字列として渡すことが推奨されます。

POSIX で `shell=True` の場合、シェルのデフォルトは `/bin/sh` になります。*args* が文字列の場合、この文字列はシェルを介して実行されるコマンドを指定します。したがって、文字列は厳密にシェルプロンプトで打つ形式と一致しなければなりません。例えば、文字列の中にスペースを含むファイル名がある場合は、クォーティングやバックスラッシュエスケープが必要です。*args* がシーケンスの場合には、最初の要素はコマンド名を表わす文字列として、残りの要素は追加の引数としてシェルに渡されます。つまり、以下の *Popen* と等価ということです:

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

Windows で `shell=True` とすると、`COMSPEC` 環境変数がデフォルトシェルを指定します。Windows で `shell=True` を指定する必要があるのは、実行したいコマンドがシェルに組み込みの場合だけです (例えば `dir` や `copy`)。バッチファイルやコンソールベースの実行ファイルを実行するために `shell=True` は必要ありません。

注釈: `shell=True` を使う前に **セキュリティで考慮すべき点** を読んでください。

---

bufsize は標準入力/標準出力/標準エラー出力パイプファイルオブジェクトを生成するときに *open()* 関数の対応する引数に渡されます:

- `0` はバッファーされないことを意味します (読み込みおよび書き出しのたびにシステムコールが行われ、すぐに復帰します)。
- `1` はラインバッファーを意味します (`universal_newlines=True`、すなわちテキストモードの場合のみ使用可能)。
- それ以外の正の整数はバッファーのおよそのサイズになることを意味します。
- 負のサイズ (デフォルト) は io.DEFAULT_BUFFER_SIZE のシステムデフォルトが使用されることを意味します。

バージョン 3.3.1 で変更: ほとんどのコードが期待する振る舞いに合わせてデフォルトでバッファリングが有効となるよう *bufsize* のデフォルト値が -1 になりました。Python 3.2.4 および 3.3.1 より前のバージョンでは、誤ってバッファーされず短い読み込みを許可する `0` がデフォルトになっていました。これは意図したものではなく、ほとんどのコードが期待する Python 2 での振る舞いとも一致していませんでした。

executable 引数は、実行する置換プログラムを指定します。これが必要になるのは極めて稀です。

shell=False のときは、*executable* は *args* で指定されている実行プログラムを置換します。しかし、オリジナルの *args* は依然としてプログラムに渡されます。ほとんどのプログラムは、*args* で指定されたプログラムをコマンド名として扱います。そして、それは実際に実行されたプログラムとは異なる可能性があります。POSIX において、**ps** のようなユーティリティの中では、*args* 名が実行ファイルの表示名になります。shell=True の場合、POSIX において *executable* 引数はデフォルトの /bin/sh に対する置換シェルを指定します。

バージョン 3.6 で変更: *executable* 引数が POSIX で *path-like object* を受け付けるようになりました。

バージョン 3.8 で変更: *executable* 引数が Windows で *path-like object* を受け付けるようになりました。

バージョン 3.8.17 で変更: Changed Windows shell search order for shell=True. The current directory and %PATH% are replaced with %COMSPEC% and %SystemRoot%\System32\cmd.exe. As a result, dropping a malicious program named cmd.exe into a current directory no longer works.

stdin、*stdout* および *stderr* には、実行するプログラムの標準入力、標準出力、および標準エラー出力のファイルハンドルをそれぞれ指定します。有効な値は *PIPE*, *DEVNULL*, 既存のファイル記述子 (正の整数)、既存の **ファイルオブジェクト** 、そして None です。*PIPE* を指定すると新しいパイプが子プロセスに向けて作られます。*DEVNULL* を指定すると特殊ファイル *os.devnull* が使用されます。デフォルト設定の None を指定するとリダイレクトは起こりません。子プロセスのファイルハンドルはすべて親から受け継がれます。加えて、*stderr* を *STDOUT* にすると、子プロセスの標準エラー出力からの出力は標準出力と同じファイルハンドルに出力されます。

preexec_fn に呼び出し可能オブジェクトが指定されている場合、このオブジェクトは子プロセスが実行される直前 (fork されたあと、exec される直前) に子プロセス内で呼ばれます。(POSIX のみ)

警告: アプリケーション中に複数のスレッドが存在する状態で *preexec_fn* 引数を使用するのは安全ではありません。exec が呼ばれる前に子プロセスがデッドロックを起こすことがあります。それを使用しなければならない場合、プログラムを自明なものにしておいてください! 呼び出すライブラリの数を最小にしてください。

注釈: 子プロセスのために環境を変更する必要がある場合は、*preexec_fn* の中でそれをするのではなく *env* 引数を使用します。*start_new_session* 引数は、子プロセスの中で os.setsid() を呼ぶ過去の一般的な *preexec_fn* の使用方法の代わりになります。

バージョン 3.8 で変更: The *preexec_fn* parameter is no longer supported in subinterpreters. The use of the parameter in a subinterpreter raises *RuntimeError*. The new restriction may affect applications that are deployed in mod_wsgi, uWSGI, and other embedded environments.

If *close_fds* is true, all file descriptors except 0, 1 and 2 will be closed before the child process is executed. Otherwise when *close_fds* is false, file descriptors obey their inheritable flag as described in **ファイル記述子の継承**.

On Windows, if *close_fds* is true then no handles will be inherited by the child process unless explicitly passed in the `handle_list` element of *`STARTUPINFO.lpAttributeList`*, or by standard handle redirection.

バージョン 3.2 で変更: *close_fds* のデフォルトは、*`False`* から上記のものに変更されました。

バージョン 3.7 で変更: On Windows the default for *close_fds* was changed from *`False`* to *`True`* when redirecting the standard handles. It's now possible to set *close_fds* to *`True`* when redirecting the standard handles.

pass_fds はオプションで、親と子の間で開いたままにしておくファイル記述子のシーケンスを指定します。何らかの *pass_fds* を渡した場合、*close_fds* は強制的に *`True`* になります。(POSIX のみ)

バージョン 3.2 で変更: *pass_fds* 引数が追加されました。

If *cwd* is not `None`, the function changes the working directory to *cwd* before executing the child. *cwd* can be a string, bytes or *path-like* object. In particular, the function looks for *executable* (or for the first item in *args*) relative to *cwd* if the executable path is a relative path.

バージョン 3.6 で変更: *cwd* 引数が POSIX で *path-like object* を受け付けるようになりました。

バージョン 3.7 で変更: *cwd* 引数が Windows で *path-like object* を受け付けるようになりました。

バージョン 3.8 で変更: *cwd* 引数が Windows で bytes オブジェクトを受け付けるようになりました。

restore_signals が真の場合 (デフォルト)、Python が SIG_IGN に設定したすべてのシグナルは子プロセスが exec される前に子プロセスの SIG_DFL に格納されます。現在これには SIGPIPE, SIGXFZ および SIGXFSZ シグナルが含まれています。(POSIX のみ)

バージョン 3.2 で変更: *restore_signals* が追加されました。

start_new_session が真の場合、サブプロセスの実行前に子プロセス内で setsid() システムコールが作成されます。(POSIX のみ)

バージョン 3.2 で変更: *start_new_session* が追加されました。

env が `None` 以外の場合、これは新しいプロセスでの環境変数を定義します。デフォルトでは、子プロセスは現在のプロセスの環境変数を引き継ぎます。

注釈: *env* を指定する場合、プログラムを実行するのに必要な変数すべてを与えなければなりません。Windows で Side-by-Side アセンブリ を実行するためには、*env* は正しい `SystemRoot` を **含まなければなりません** 。

If *encoding* or *errors* are specified, or *text* is true, the file objects *stdin*, *stdout* and *stderr* are opened in text mode with the specified encoding and *errors*, as described above in **よく使われる引数**. The *universal_newlines* argument is equivalent to *text* and is provided for backwards compatibility. By default, file objects are opened in binary mode.

バージョン 3.6 で追加: *encoding* と *errors* が追加されました。

バージョン 3.7 で追加: *text* が、*universal_newlines* のより読みやすい別名として追加されました。

startupinfo は、基底の CreateProcess 関数に渡される *STARTUPINFO* オブジェクトになります。*creationflags* は、与えられるなら、以下のフラグのうち一つ以上にできます:

- *CREATE_NEW_CONSOLE*
- *CREATE_NEW_PROCESS_GROUP*
- *ABOVE_NORMAL_PRIORITY_CLASS*
- *BELOW_NORMAL_PRIORITY_CLASS*
- *HIGH_PRIORITY_CLASS*
- *IDLE_PRIORITY_CLASS*
- *NORMAL_PRIORITY_CLASS*
- *REALTIME_PRIORITY_CLASS*
- *CREATE_NO_WINDOW*
- *DETACHED_PROCESS*
- *CREATE_DEFAULT_ERROR_MODE*
- *CREATE_BREAKAWAY_FROM_JOB*

Popen オブジェクトは with 文によってコンテキストマネージャーとしてサポートされます: 終了時には標準ファイル記述子が閉じられ、プロセスを待機します:

```
with Popen(["ifconfig"], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

引数 executable, args, cwd, env を指定して **監査イベント** subprocess.Popen を送出します。

バージョン 3.2 で変更: コンテキストマネージャーサポートが追加されました。

バージョン 3.6 で変更: Popen destructor now emits a *ResourceWarning* warning if the child process is still running.

バージョン 3.8 で変更: Popen can use *os.posix_spawn()* in some cases for better performance. On Windows Subsystem for Linux and QEMU User Emulation, Popen constructor using *os.posix_spawn()* no longer raise an exception on errors like missing program, but the child process fails with a non-zero *returncode*.

例外

Exceptions raised in the child process, before the new program has started to execute, will be re-raised in the parent.

もっとも一般的に起こる例外は *OSError* です。これは、たとえば存在しないファイルを実行しようとしたときなどに発生します。アプリケーションは *OSError* 例外に備えておかなければなりません。

不正な引数で *Popen* が呼ばれた場合は *ValueError* が発生します。

呼び出されたプロセスが非ゼロのリターンコードを返した場合 *check_call()* や *check_output()* は *CalledProcessError* を送出します。

call() や *Popen.communicate()* のような *timeout* 引数を受け取るすべての関数とメソッドは、プロセスが終了する前にタイムアウトが発生した場合に *TimeoutExpired* を送出します。

このモジュールで定義されたすべての例外は *SubprocessError* を継承しています。

バージョン 3.3 で追加: *SubprocessError* 基底クラスが追加されました。

17.6.2 セキュリティで考慮すべき点

その他一部の popen 関数と異なり、この実装は暗黙的にシステムシェルを呼び出すことはありません。これはシェルのメタ文字を含むすべての文字が子プロセスに安全に渡されることを意味します。`shell=True` でシェルを明示的に呼びだした場合、シェルインジェクション の脆弱性に対処するための、すべての空白やメタ文字の適切なクオートの保証はアプリケーションの責任になります。

`shell=True` を使用するときは、シェルコマンドで使用される文字列の空白やメタ文字は *shlex.quote()* 関数を使うと正しくエスケープできます。

17.6.3 Popen オブジェクト

Popen クラスのインスタンスには、以下のようなメソッドがあります:

`Popen.poll()`

子プロセスが終了しているかどうかを調べます。*returncode* 属性を設定して返します。そうでなければ `None` を返します。

`Popen.wait(`*timeout=None*`)`

子プロセスが終了するまで待ちます。*returncode* 属性を設定して返します。

プロセスが *timeout* 秒後に終了してない場合、*TimeoutExpired* 例外を送出します。この例外を捕捉して wait を再試行するのは安全です。

注釈: `stdout=PIPE` や `stderr=PIPE` を使っていて、より多くのデータを受け入れるために OS のパイプバッファーをブロックしているパイプに子プロセスが十分な出力を生成した場合、デッドロックが

発生します。これを避けるには *Popen.communicate()* を使用してください。

注釈: この関数はビジーループ (非ブロック化呼び出しと短いスリープ) を使用して実装されています。非同期の待機には *asyncio* モジュールを使用してください (*asyncio.create_subprocess_exec* を参照)。

バージョン 3.3 で変更: *timeout* が追加されました

Popen.**communicate**(*input=None*, *timeout=None*)

Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate and set the *returncode* attribute. The optional *input* argument should be data to be sent to the child process, or `None`, if no data should be sent to the child. If streams were opened in text mode, *input* must be a string. Otherwise, it must be bytes.

communicate() returns a tuple `(stdout_data, stderr_data)`. The data will be strings if streams were opened in text mode; otherwise, bytes.

子プロセスの標準入力にデータを送りたい場合は、Popen オブジェクトを `stdin=PIPE` と指定して作成しなければなりません。同じく、戻り値のタプルから `None` ではない値を取得するためには、`stdout=PIPE` かつ/または `stderr=PIPE` を指定しなければなりません。

プロセスが *timeout* 秒後に終了してない場合、*TimeoutExpired* 例外が送出されます。この例外を捕捉して通信を再試行しても出力データは失われません。

タイムアウトが発生した場合子プロセスは kill されません。したがって、適切にクリーンアップを行うために、正常に動作するアプリケーションは子プロセスを kill して通信を終了すべきです:

```
proc = subprocess.Popen(...)
try:
    outs, errs = proc.communicate(timeout=15)
except TimeoutExpired:
    proc.kill()
    outs, errs = proc.communicate()
```

注釈: 受信したデータはメモリにバッファーされます。そのため、返されるデータが大きいかあるいは制限がないような場合はこのメソッドを使うべきではありません。

バージョン 3.3 で変更: *timeout* が追加されました

Popen.**send_signal**(*signal*)

signal シグナルを子プロセスに送ります。

注釈: Windows では、SIGTERM は *terminate()* の別名です。CTRL_C_EVENT と CTRL_BREAK_EVENT を、*CREATE_NEW_PROCESS_GROUP* を含む *creationflags* で始

まった、プロセスに送れます。

Popen.**terminate**()
子プロセスを停止します。POSIX OS では、このメソッドは SIGTERM シグナルを子プロセスに送ります。Windows では、Win32 API の `TerminateProcess()` 関数を利用して子プロセスを止めます。

Popen.**kill**()
子プロセスを kill します。POSIX OS では SIGKILL シグナルを子プロセスに送ります。Windows では、*kill()* は *terminate()* の別名です。

以下の属性も利用可能です:

Popen.**args**
Popen に渡された引数 *args* です -- プログラム引数のシーケンスまたは 1 個の文字列になります。

バージョン 3.3 で追加.

Popen.**stdin**
If the *stdin* argument was *PIPE*, this attribute is a writeable stream object as returned by *open()*. If the *encoding* or *errors* arguments were specified or the *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stdin* argument was not *PIPE*, this attribute is `None`.

Popen.**stdout**
If the *stdout* argument was *PIPE*, this attribute is a readable stream object as returned by *open()*. Reading from the stream provides output from the child process. If the *encoding* or *errors* arguments were specified or the *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stdout* argument was not *PIPE*, this attribute is `None`.

Popen.**stderr**
If the *stderr* argument was *PIPE*, this attribute is a readable stream object as returned by *open()*. Reading from the stream provides error output from the child process. If the *encoding* or *errors* arguments were specified or the *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stderr* argument was not *PIPE*, this attribute is `None`.

警告: *.stdin.write*, *.stdout.read*, *.stderr.read* を利用すると、別のパイプの OS パイプバッファーがいっぱいになってデッドロックが発生する恐れがあります。これを避けるためには *communicate()* を利用してください。

Popen.**pid**
子プロセスのプロセス ID が入ります。

shell 引数を `True` に設定した場合は、生成されたシェルのプロセス ID になります。

Popen.**returncode**
poll() か *wait()* (か、間接的に *communicate()*) から設定された、子プロセスの終了ステータスが

入ります。None はまだその子プロセスが終了していないことを示します。

負の値 -N は子プロセスがシグナル N により中止させられたことを示します (POSIX のみ)。

17.6.4 Windows Popen ヘルパー

STARTUPINFO クラスと以下の定数は、Windows のみで利用できます。

class subprocess.STARTUPINFO(**, dwFlags=0, hStdInput=None, hStdOutput=None, hStdError=None, wShowWindow=0, lpAttributeList=None*)

Partial support of the Windows STARTUPINFO structure is used for *Popen* creation. The following attributes can be set by passing them as keyword-only arguments.

バージョン 3.7 で変更: キーワード専用引数のサポートが追加されました。

dwFlags

特定の *STARTUPINFO* の属性が、プロセスがウィンドウを生成するときに使われるかを決定するビットフィールドです:

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_USESHOWWINDOW
```

hStdInput

dwFlags が *STARTF_USESTDHANDLES* を指定すれば、この属性がプロセスの標準入力処理です。*STARTF_USESTDHANDLES* が指定されなければ、標準入力のデフォルトはキーボードバッファーです。

hStdOutput

dwFlags が *STARTF_USESTDHANDLES* を指定すれば、この属性がプロセスの標準出力処理です。そうでなければ、この属性は無視され、標準出力のデフォルトはコンソールウィンドウのバッファーです。

hStdError

dwFlags が *STARTF_USESTDHANDLES* を指定すれば、この属性がプロセスの標準エラー処理です。そうでなければ、この属性は無視され、標準エラー出力のデフォルトはコンソールウィンドウのバッファーです。

wShowWindow

dwFlags が *STARTF_USESHOWWINDOW* を指定すれば、この属性は ShowWindow 関数の nCmdShow 引数で指定された値なら、SW_SHOWDEFAULT 以外の任意のものにできます。しかし、この属性は無視されます。

この属性には *SW_HIDE* が提供されています。これは、*Popen* が shell=True として呼び出されたときに使われます。

lpAttributeList

A dictionary of additional attributes for process creation as given in STARTUPINFOEX, see UpdateProcThreadAttribute.

Supported attributes:

handle_list Sequence of handles that will be inherited. *close_fds* must be true if non-empty.

The handles must be temporarily made inheritable by *os.set_handle_inheritable()* when passed to the *Popen* constructor, else *OSError* will be raised with Windows error ERROR_INVALID_PARAMETER (87).

> **警告:** In a multithreaded process, use caution to avoid leaking handles that are marked inheritable when combining this feature with concurrent calls to other process creation functions that inherit all handles such as *os.system()*. This also applies to standard handle redirection, which temporarily creates inheritable handles.

バージョン 3.7 で追加.

Windows Constants

subprocess モジュールは、以下の定数を公開しています。

subprocess.STD_INPUT_HANDLE
: 標準入力デバイスです。この初期値は、コンソール入力バッファ、CONIN$ です。

subprocess.STD_OUTPUT_HANDLE
: 標準出力デバイスです。この初期値は、アクティブコンソールスクリーン、CONOUT$ です。

subprocess.STD_ERROR_HANDLE
: 標準エラーデバイスです。この初期値は、アクティブコンソールスクリーン、CONOUT$ です。

subprocess.SW_HIDE
: ウィンドウを隠します。別のウィンドウがアクティブになります。

subprocess.STARTF_USESTDHANDLES
: 追加情報を保持する、*STARTUPINFO.hStdInput*, *STARTUPINFO.hStdOutput*, および *STARTUPINFO.hStdError* 属性を指定します。

subprocess.STARTF_USESHOWWINDOW
: 追加情報を保持する、*STARTUPINFO.wShowWindow* 属性を指定します。

subprocess.CREATE_NEW_CONSOLE
: 新しいプロセスが、親プロセスのコンソールを継承する (デフォルト) のではなく、新しいコンソールを持ちます。

subprocess.CREATE_NEW_PROCESS_GROUP
: 新しいプロセスグループが生成されることを指定する *Popen* creationflags パラメーターです。このフラグは、サブプロセスで *os.kill()* を使うのに必要です。

  CREATE_NEW_CONSOLE が指定されていたら、このフラグは無視されます。

subprocess.ABOVE_NORMAL_PRIORITY_CLASS

A *Popen* creationflags parameter to specify that a new process will have an above average priority.

バージョン 3.7 で追加.

subprocess.BELOW_NORMAL_PRIORITY_CLASS

A *Popen* creationflags parameter to specify that a new process will have a below average priority.

バージョン 3.7 で追加.

subprocess.HIGH_PRIORITY_CLASS

A *Popen* creationflags parameter to specify that a new process will have a high priority.

バージョン 3.7 で追加.

subprocess.IDLE_PRIORITY_CLASS

A *Popen* creationflags parameter to specify that a new process will have an idle (lowest) priority.

バージョン 3.7 で追加.

subprocess.NORMAL_PRIORITY_CLASS

A *Popen* creationflags parameter to specify that a new process will have an normal priority. (default)

バージョン 3.7 で追加.

subprocess.REALTIME_PRIORITY_CLASS

A *Popen* creationflags parameter to specify that a new process will have realtime priority. You should almost never use REALTIME_PRIORITY_CLASS, because this interrupts system threads that manage mouse input, keyboard input, and background disk flushing. This class can be appropriate for applications that "talk" directly to hardware or that perform brief tasks that should have limited interruptions.

バージョン 3.7 で追加.

subprocess.CREATE_NO_WINDOW

A *Popen* creationflags parameter to specify that a new process will not create a window.

バージョン 3.7 で追加.

subprocess.DETACHED_PROCESS

A *Popen* creationflags parameter to specify that a new process will not inherit its parent's console. This value cannot be used with CREATE_NEW_CONSOLE.

バージョン 3.7 で追加.

subprocess.CREATE_DEFAULT_ERROR_MODE

A *Popen* creationflags parameter to specify that a new process does not inherit the error mode of the calling process. Instead, the new process gets the default error mode. This feature is particularly useful for multithreaded shell applications that run with hard errors disabled.

バージョン 3.7 で追加.

subprocess.CREATE_BREAKAWAY_FROM_JOB
A *Popen* creationflags parameter to specify that a new process is not associated with the job.

バージョン 3.7 で追加.

17.6.5 古い高水準 API

Python 3.5 より前のバージョンでは、サブプロセスに対して以下の 3 つの関数からなる高水準 API が用意されていました。現在多くの場合 *run()* の使用で済みますが、既存の多くのコードではこれらの関数が使用されています。

subprocess.call(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, ***other_popen_kwargs*)
args で指定されたコマンドを実行します。コマンドの終了を待ち、*returncode* 属性を返します。

Code needing to capture stdout or stderr should use *run()* instead:

```
run(...).returncode
```

To suppress stdout or stderr, supply a value of *DEVNULL*.

上記の引数は、よく使われるものだけ示しています。関数の全使用法は *Popen* コンストラクターの内容と同じになります - この関数は、このインターフェースに直接指定される *timeout* 以外は与えられた全引数を渡します。

注釈: この関数を使用する際は stdout=PIPE および stderr=PIPE を使用しないでください。子プロセスが OS のパイプバッファーを埋めてしまうほどの出力データを生成した場合、パイプからは読み込まれないので、子プロセスがブロックされることがあります。

バージョン 3.3 で変更: *timeout* が追加されました

バージョン 3.8.17 で変更: Changed Windows shell search order for shell=True. The current directory and %PATH% are replaced with %COMSPEC% and %SystemRoot%\System32\cmd.exe. As a result, dropping a malicious program named cmd.exe into a current directory no longer works.

subprocess.check_call(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, ***other_popen_kwargs*)
指定された引数でコマンドを実行し、完了を待ちます。コマンドのリターンコードがゼロならば返りますが、非ゼロなら *CalledProcessError* 例外が送出されます。*CalledProcessError* オブジェクトにはリターンコードが *returncode* 属性として格納されています。

Code needing to capture stdout or stderr should use *run()* instead:

```
run(..., check=True)
```

To suppress stdout or stderr, supply a value of *DEVNULL*.

上記の引数は、よく使われるものだけ示しています。関数の全使用法は *Popen* コンストラクターの内容と同じになります - この関数は、このインターフェースに直接指定される *timeout* 以外は与えられた全引数を渡します。

注釈: この関数を使用する際は `stdout=PIPE` および `stderr=PIPE` を使用しないでください。子プロセスが OS のパイプバッファーを埋めてしまうほどの出力データを生成した場合、パイプからは読み込まれないので、子プロセスがブロックされることがあります。

バージョン 3.3 で変更: *timeout* が追加されました

バージョン 3.8.17 で変更: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

`subprocess.check_output`(*args*, ***, *stdin=None*, *stderr=None*, *shell=False*, *cwd=None*, *encoding=None*, *errors=None*, *universal_newlines=None*, *timeout=None*, *text=None*, ***other_popen_kwargs*)

引数でコマンドを実行し、その出力を返します。

コマンドのリターンコードが非ゼロならば *CalledProcessError* 例外が送出されます。*CalledProcessError* オブジェクトには、リターンコードが *returncode* 属性に、コマンドからの出力が *output* 属性に、それぞれ格納されています。

これは次と等価です:

```
run(..., check=True, stdout=PIPE).stdout
```

The arguments shown above are merely some common ones. The full function signature is largely the same as that of *run()* - most arguments are passed directly through to that interface. One API deviation from *run()* behavior exists: passing `input=None` will behave the same as `input=b''` (or `input=''`, depending on other arguments) rather than using the parent's standard input file handle.

デフォルトで、この関数はデータをエンコードされたバイトとして返します。出力されたデータの実際のエンコードは起動されているコマンドに依存するため、テキストへのデコードは通常アプリケーションレベルで扱う必要があります。

This behaviour may be overridden by setting *text*, *encoding*, *errors*, or *universal_newlines* to `True` as described in **よく使われる引数** and *run()*.

標準エラー出力も結果に含めるには、`stderr=subprocess.STDOUT` を使います:

```
>>> subprocess.check_output(
...     "ls non_existent_file; exit 0",
...     stderr=subprocess.STDOUT,
```

(次のページに続く)

(前のページからの続き)

```
...     shell=True)
'ls: non_existent_file: No such file or directory\n'
```

バージョン 3.1 で追加.

バージョン 3.3 で変更: *timeout* が追加されました

バージョン 3.4 で変更: キーワード引数 *input* が追加されました。

バージョン 3.6 で変更: *encoding* and *errors* were added. See *run()* for details.

バージョン 3.7 で追加: *text* が、*universal_newlines* のより読みやすい別名として追加されました。

バージョン 3.8.17 で変更: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

17.6.6 古い関数を subprocess モジュールで置き換える

この節では、"a becomes b" と書かれているものは a の代替として b が使えるということを表します。

注釈: この節で紹介されている "a" 関数は全て、実行するプログラムが見つからないときは (おおむね) 静かに終了します。それに対して "b" 代替手段は *OSError* 例外を送出します。

また、要求された操作が非ゼロの終了コードを返した場合、*check_output()* を使用した置き換えは *CalledProcessError* で失敗します。その出力は、送出された例外の *output* 属性として利用可能です。

以下の例では、適切な関数が *subprocess* モジュールからすでにインポートされていることを前提としています。

Replacing /bin/sh shell command substitution

```
output=$(mycmd myarg)
```

これは以下のようになります:

```
output = check_output(["mycmd", "myarg"])
```

シェルのパイプラインを置き換える

```
output=$(dmesg | grep hda)
```

これは以下のようになります:

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close()  # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
```

p2 を開始した後の p1.stdout.close() の呼び出しは、p1 が p2 の前に存在した場合に、p1 が SIGPIPE を受け取るために重要です。

あるいは、信頼された入力に対しては、シェル自身のパイプラインサポートを直接使用することもできます:

```
output=$(dmesg | grep hda)
```

これは以下のようになります:

```
output=check_output("dmesg | grep hda", shell=True)
```

os.system() を置き換える

```
sts = os.system("mycmd" + " myarg")
# becomes
sts = call("mycmd" + " myarg", shell=True)
```

注釈:

- このプログラムは普通シェル経由で呼び出す必要はありません。

より現実的な例ではこうなるでしょう:

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print("Child was terminated by signal", -retcode, file=sys.stderr)
    else:
        print("Child returned", retcode, file=sys.stderr)
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```

os.spawn 関数群を置き換える

P_NOWAIT の例:

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

P_WAIT の例:

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

シーケンスを使った例:

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

環境変数を使った例:

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

os.popen(), os.popen2(), os.popen3() を置き換える

```
(child_stdin, child_stdout) = os.popen2(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

```
(child_stdin,
 child_stdout,
 child_stderr) = os.popen3(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin,
 child_stdout,
 child_stderr) = (p.stdin, p.stdout, p.stderr)
```

```
(child_stdin, child_stdout_and_stderr) = os.popen4(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```

終了コードハンドリングは以下のように解釈します:

```
pipe = os.popen(cmd, 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print("There were some errors")
==>
process = Popen(cmd, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print("There were some errors")
```

popen2 モジュールの関数群を置き換える

注釈: popen2 関数の cmd 引数が文字列の場合、コマンドは /bin/sh によって実行されます。リストの場合、コマンドは直接実行されます。

```
(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize, mode)
==>
p = Popen("somestring", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

```
(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"], bufsize, mode)
==>
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

popen2.Popen3 および popen2.Popen4 は以下の点を除けば、基本的に *subprocess.Popen* と同じです:

- *Popen* は実行が失敗した場合に例外を送出します。
- *capturestderr* 引数は *stderr* 引数に代わりました。
- stdin=PIPE および stdout=PIPE を指定する必要があります。
- popen2 はデフォルトですべてのファイル記述子を閉じます。しかし、全てのプラットフォーム上で、あるいは過去の Python バージョンでこの挙動を保証するためには、*Popen* に対して close_fds=True を指定しなければなりません。

17.6.7 レガシーなシェル呼び出し関数

このモジュールでは、以下のような 2.x commands モジュールからのレガシー関数も提供しています。これらの操作は、暗黙的にシステムシェルを起動します。また、セキュリティに関して上述した保証や例外処理一貫性は、これらの関数では有効ではありません。

subprocess.getstatusoutput(*cmd*)

シェル中の *cmd* を実行して (exitcode, output) を返します。

Execute the string *cmd* in a shell with Popen.check_output() and return a 2-tuple (exitcode, output). The locale encoding is used; see the notes on よく使われる引数 for more details.

A trailing newline is stripped from the output. The exit code for the command can be interpreted as the return code of subprocess. Example:

```
>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(1, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(127, 'sh: /bin/junk: not found')
>>> subprocess.getstatusoutput('/bin/kill $$')
(-15, '')
```

利用可能な環境: POSIX と Windows。

バージョン 3.3.4 で変更: Windows のサポートが追加されました。

The function now returns (exitcode, output) instead of (status, output) as it did in Python 3.3.3 and earlier. exitcode has the same value as *returncode*.

subprocess.getoutput(*cmd*)

シェル中の *cmd* を実行して出力 (stdout と stderr) を返します。

getstatusoutput() に似ていますが、終了コードは無視され、コマンドの出力のみを返します。例えば:

```
>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'
```

利用可能な環境: POSIX と Windows。

バージョン 3.3.4 で変更: Windows で利用可能になりました

17.6.8 注釈

Windows における引数シーケンスから文字列への変換

Windows では、*args* シーケンスは以下の (MS C ランタイムで使われる規則に対応する) 規則を使って解析できる文字列に変換されます:

1. 引数は、スペースかタブのどちらかの空白で分けられます。
2. ダブルクオーテーションマークで囲まれた文字列は、空白が含まれていたとしても 1 つの引数として解釈されます。クオートされた文字列は引数に埋め込めます。
3. バックスラッシュに続くダブルクオーテーションマークは、リテラルのダブルクオーテーションマークと解釈されます。
4. バックスラッシュは、ダブルクオーテーションが続かない限り、リテラルとして解釈されます。
5. 複数のバックスラッシュにダブルクオーテーションマークが続くなら、バックスラッシュ 2 つで 1 つのバックスラッシュ文字と解釈されます。バックスラッシュの数が奇数なら、最後のバックスラッシュは規則 3 に従って続くダブルクオーテーションマークをエスケープします。

参考:

shlex コマンドラインを解析したりエスケープしたりする関数を提供するモジュール。

17.7 sched --- イベントスケジューラ

ソースコード: Lib/sched.py

---

sched モジュールは一般的な目的のためのイベントスケジューラを実装するクラスを定義します:

class sched.scheduler(*timefunc=time.monotonic*, *delayfunc=time.sleep*)

scheduler クラスはイベントをスケジュールするための一般的なインタフェースを定義します。それは ” 外の世界” を実際に扱うための 2 つの関数を必要とします --- *timefunc* は引数なしで呼ばれて 1 つの数値を返す callable オブジェクトでなければなりません (戻り値は任意の単位で「時間」を表します)。*delayfunc* は 1 つの引数を持つ callable オブジェクトでなければならず、その時間だけ遅延する必要があります (引数は *timefunc* の出力と互換)。*delayfunc* は、各々のイベントが実行された後に引数 0 で呼ばれることがあります。これは、マルチスレッドアプリケーションの中で他のスレッドが実行する機会を与えるためです。

バージョン 3.3 で変更: *timefunc* と *delayfunc* がオプション引数になりました。

バージョン 3.3 で変更: *scheduler* クラスをマルチスレッド環境で安全に使用出来るようになりました。

以下はプログラム例です:

```
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(a='default'):
...     print("From print_time", time.time(), a)
...
>>> def print_some_times():
...     print(time.time())
...     s.enter(10, 1, print_time)
...     s.enter(5, 2, print_time, argument=('positional',))
...     s.enter(5, 1, print_time, kwargs={'a': 'keyword'})
...     s.run()
...     print(time.time())
...
>>> print_some_times()
930343690.257
From print_time 930343695.274 positional
From print_time 930343695.275 keyword
From print_time 930343700.273 default
930343700.276
```

17.7.1 スケジューラオブジェクト

scheduler インスタンスは以下のメソッドと属性を持っています:

scheduler.**enterabs**(*time*, *priority*, *action*, *argument=()*, *kwargs={}*)

新しいイベントをスケジュールします。引数 *time* は、コンストラクタへ渡された *timefunc* の戻り値と互換な数値型でなければいけません。同じ *time* によってスケジュールされたイベントは、それらの *priority* によって実行されます。数値の小さい方が高い優先度となります。

イベントを実行することは、`action(*argument, **kwargs)` を実行することを意味します。*argument* は *action* のための位置引数を保持するシーケンスでなければいけません。*kwargs* は *action* のためのキーワード引数を保持する辞書でなければいけません。

戻り値は、後にイベントをキャンセルする時に使われる可能性のあるイベントです (*cancel()* を参照)。

バージョン 3.3 で変更: *argument* 引数が任意になりました。

バージョン 3.3 で変更: *kwargs* 引数が追加されました。

scheduler.**enter**(*delay*, *priority*, *action*, *argument=()*, *kwargs={}*)

時間単位以上の *delay* でイベントをスケジュールします。相対的時間以外の、引数、効果、戻り値は、*enterabs()* に対するものと同じです。

バージョン 3.3 で変更: *argument* 引数が任意になりました。

バージョン 3.3 で変更: *kwargs* 引数が追加されました。

scheduler.**cancel**(*event*)

キューからイベントを消去します。もし *event* がキューにある現在のイベントでないならば、このメソッドは *ValueError* を送出します。

scheduler.empty()
: もしイベントキューが空ならば、True を返します。

scheduler.run(*blocking=True*)
: すべてのスケジュールされたイベントを実行します。このメソッドは次のイベントを待ち、それを実行し、スケジュールされたイベントがなくなるまで同じことを繰り返します。(イベントの待機は、コンストラクタへ渡された関数 delayfunc() を使うことで行います。)

  blocking が False の場合、最も早く期限が来るスケジュールされたイベントを (存在する場合) 実行し、スケジューラ内で次にスケジュールされた呼び出しの期限を (存在する場合) 返します。

  action あるいは *delayfunc* は例外を投げることができます。いずれの場合も、スケジューラは一貫した状態を維持し、例外を伝播するでしょう。例外が *action* によって投げられる場合、イベントは *run()* への呼出しを未来に行なわないでしょう。

  イベントのシーケンスが、次イベントの前に、利用可能時間より実行時間が長いと、スケジューラは単に遅れることになるでしょう。イベントが落ちることはありません; 呼出しコードはもはや適切でないキャンセルイベントに対して責任があります。

  バージョン 3.3 で変更: *blocking* 引数が追加されました。

scheduler.queue
: 読み出し専用の属性で、これから起こるイベントが実行される順序で格納されたリストを返します。各イベントは、次の属性 time, priority, action, argument, kwargs を持った *named tuple* の形式になります。

17.8 queue --- 同期キュークラス

ソースコード: Lib/queue.py

---

queue モジュールは、複数プロデューサ-複数コンシューマ (multi-producer, multi-consumer) キューを実装します。これは、複数のスレッドの間で情報を安全に交換しなければならないときのマルチスレッドプログラミングで特に有益です。このモジュールの *Queue* クラスは、必要なすべてのロックセマンティクスを実装しています。

このモジュールでは 3 種類のキューが実装されています。それらはキューから取り出されるエントリの順番だけが違います。FIFO キューでは、最初に追加されたエントリが最初に取り出されます。LIFO キューでは、最後に追加されたエントリが最初に取り出されます (スタックのように振る舞います)。優先順位付きキュー (priority queue) では、エントリは (*heapq* モジュールを利用して) ソートされ、最も低い値のエントリが最初に取り出されます。

内部的には、これらの 3 種類のキューは競争スレッドを一時的にブロックするためにロックを使っています; しかし、スレッド内での再入を扱うようには設計されていません。

In addition, the module implements a ”simple” FIFO queue type, *SimpleQueue*, whose specific implementation provides additional guarantees in exchange for the smaller functionality.

queue モジュールは以下のクラスと例外を定義します:

class queue.Queue(*maxsize=0*)

FIFO キューのコンストラクタです。*maxsize* はキューに入れられる要素数の上限を設定する整数です。いったんこの大きさに達したら、挿入処理はキューの要素が消費されるまでブロックされます。*maxsize* が 0 以下の場合は、キューの大きさは無限です。

class queue.LifoQueue(*maxsize=0*)

LIFO キューのコンストラクタです。*maxsize* はキューに入れられる要素数の上限を設定する整数です。いったんこの大きさに達したら、挿入処理はキューの要素が消費されるまでブロックされます。*maxsize* が 0 以下の場合は、キューの大きさは無限です。

class queue.PriorityQueue(*maxsize=0*)

優先順位付きキューのコンストラクタです。*maxsize* はキューに置くことのできる要素数の上限を設定する整数です。いったんこの大きさに達したら、挿入はキューの要素が消費されるまでブロックされます。もし *maxsize* が 0 以下であるならば、キューの大きさは無限です。

最小の値を持つ要素が最初に検索されます (最小の値を持つ値は、`sorted(list(entries))[0]` によって返されるものです)。典型的な要素のパターンは、`(priority_number, data)` 形式のタプルです。

If the *data* elements are not comparable, the data can be wrapped in a class that ignores the data item and only compares the priority number:

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

class queue.SimpleQueue

Constructor for an unbounded FIFO queue. Simple queues lack advanced functionality such as task tracking.

バージョン 3.7 で追加.

exception queue.Empty

空の *Queue* オブジェクトで、非ブロックメソッド *get()* (または *get_nowait()*) が呼ばれたとき、送出される例外です。

exception queue.Full

満杯の *Queue* オブジェクトで、非ブロックメソッド *put()* (または *put_nowait()*) が呼ばれたとき、送出される例外です。

17.8.1 キューオブジェクト

キューオブジェクト (*Queue*, *LifoQueue*, *PriorityQueue*) は、以下の public メソッドを提供しています。

Queue.qsize()

キューの近似サイズを返します。ここで、qsize() > 0 は後続の get() がブロックしないことを保証しないこと、また qsize() < maxsize が put() がブロックしないことを保証しないことに注意してください。

Queue.empty()

キューが空の場合は `True` を返し、そうでなければ `False` を返します。empty() が `True` を返しても、後続の put() の呼び出しがブロックしないことは保証されません。同様に、empty() が `False` を返しても、後続の get() の呼び出しがブロックしないことは保証されません。

Queue.full()

キューが一杯の場合は `True` を返し、そうでなければ `False` を返します。full() が `True` を返しても、後続の get() の呼び出しがブロックしないことは保証されません。同様に、full() が `False` を返しても、後続の put() の呼び出しがブロックしないことは保証されません。

Queue.put(*item*, *block=True*, *timeout=None*)

item をキューに入れます。もしオプション引数 *block* が真で *timeout* が `None` (デフォルト) の場合は、必要であればフリースロットが利用可能になるまでブロックします。*timeout* が正の数の場合は、最大で *timeout* 秒間ブロックし、その時間内に空きスロットが利用可能にならなければ、例外 *Full* を送出します。そうでない場合 (*block* が偽) は、空きスロットが直ちに利用できるならば、キューにアイテムを置きます。できないならば、例外 *Full* を送出します (この場合 *timeout* は無視されます)。

Queue.put_nowait(*item*)

`put(item, False)` と等価です。

Queue.get(*block=True*, *timeout=None*)

キューからアイテムを取り除き、それを返します。オプション引数 *block* が真で *timeout* が `None` (デフォルト) の場合は、必要であればアイテムが取り出せるようになるまでブロックします。もし *timeout* が正の数の場合は、最大で *timeout* 秒間ブロックし、その時間内でアイテムが取り出せるようにならなければ、例外 *Empty* を送出します。そうでない場合 (*block* が偽) は、直ちにアイテムが取り出せるならば、それを返します。できないならば、例外 *Empty* を送出します (この場合 *timeout* は無視されます)。

Prior to 3.0 on POSIX systems, and for all versions on Windows, if *block* is true and *timeout* is `None`, this operation goes into an uninterruptible wait on an underlying lock. This means that no exceptions can occur, and in particular a SIGINT will not trigger a *KeyboardInterrupt*.

Queue.get_nowait()

`get(False)` と等価です。

キューに入れられたタスクが全てコンシューマスレッドに処理されたかどうかを追跡するために 2 つのメソッドが提供されます。

Queue.task_done()

過去にキューに入れられたタスクが完了した事を示します。キューのコンシューマスレッドに利用されます。タスクの取り出しに使われた各 *get()* の後に *task_done()* を呼び出すと、取り出したタスクに対する処理が完了した事をキューに教えます。

join() がブロックされていた場合、全 item が処理された (キューに *put()* された全ての item に対して *task_done()* が呼び出されたことを意味します) 時に復帰します。

キューにある要素より多く呼び出された場合 *ValueError* が発生します。

Queue.join()

キューにあるすべてのアイテムが取り出されて処理されるまでブロックします。

キューに item が追加される度に、未完了タスクカウントが増やされます。コンシューマスレッドが *task_done()* を呼び出して、item を受け取ってそれに対する処理が完了した事を知らせる度に、未完了タスクカウントが減らされます。未完了タスクカウントが 0 になったときに、*join()* のブロックが解除されます。

キューに入れたタスクが完了するのを待つ例:

```
import threading, queue

q = queue.Queue()

def worker():
    while True:
        item = q.get()
        print(f'Working on {item}')
        print(f'Finished {item}')
        q.task_done()

# turn-on the worker thread
threading.Thread(target=worker, daemon=True).start()

# send thirty task requests to the worker
for item in range(30):
    q.put(item)
print('All task requests sent\n', end='')

# block until all tasks are done
q.join()
print('All work completed')
```

17.8.2 SimpleQueue オブジェクト

SimpleQueue オブジェクトは以下の public メソッドを提供しています。

SimpleQueue.**qsize**()
: キューの近似サイズを返します。ここで、qsize() > 0 であるからといって、後続の get() の呼び出しがブロックしないことが保証されないことに注意してください。

SimpleQueue.**empty**()
: キューが空の場合は `True` を返し、そうでなければ `False` を返します。empty() が `False` を返しても、後続の get() の呼び出しがブロックしないことは保証されません。

SimpleQueue.**put**(*item*, *block=True*, *timeout=None*)
: Put *item* into the queue. The method never blocks and always succeeds (except for potential low-level errors such as failure to allocate memory). The optional args *block* and *timeout* are ignored and only provided for compatibility with *Queue.put()*.

  CPython implementation detail: This method has a C implementation which is reentrant. That is, a `put()` or `get()` call can be interrupted by another `put()` call in the same thread without deadlocking or corrupting internal state inside the queue. This makes it appropriate for use in destructors such as `__del__` methods or *weakref* callbacks.

SimpleQueue.**put_nowait**(*item*)
: `put(item)` と等価です。:meth:‘Queue.put_nowait‘との互換性のためのメソッドです。

SimpleQueue.**get**(*block=True*, *timeout=None*)
: キューからアイテムを取り除き、それを返します。オプション引数 *block* が真で *timeout* が `None` (デフォルト) の場合は、必要であればアイテムが取り出せるようになるまでブロックします。もし *timeout* が正の数の場合は、最大で *timeout* 秒間ブロックし、その時間内でアイテムが取り出せるようにならなければ、例外 *Empty* を送出します。そうでない場合 (*block* が偽) は、直ちにアイテムが取り出せるならば、それを返します。できないならば、例外 *Empty* を送出します (この場合 *timeout* は無視されます)。

SimpleQueue.**get_nowait**()
: `get(False)` と等価です。

参考:

multiprocessing.Queue **クラス** (マルチスレッドではなく) マルチプロセスの文脈で使用されるキュークラス。

collections.deque is an alternative implementation of unbounded queues with fast atomic *append()* and *popleft()* operations that do not require locking and also support indexing.

17.9 contextvars --- コンテキスト変数

このモジュールは、コンテキストローカルな状態を管理し、保持し、アクセスするための API を提供します。*ContextVar* クラスは **コンテキスト変数** を宣言し、取り扱うために使われます。非同期フレームワークで現時点のコンテキストを管理するのには、*copy_context()* 関数と *Context* クラスを使うべきです。

状態を持っているコンテキストマネージャは *threading.local()* ではなくコンテキスト変数を使い、並行処理のコードから状態が意図せず他のコードへ漏れ出すのを避けるべきです。

より詳しくは **PEP 567** を参照をしてください。

バージョン 3.7 で追加.

17.9.1 コンテキスト変数

class contextvars.**ContextVar**(*name*[, *, *default*])

このクラスは新しいコンテキスト変数を宣言するのに使われます。例えば、次の通りです:

```
var: ContextVar[int] = ContextVar('var', default=42)
```

必須のパラメータの *name* は内観やデバッグの目的で使われます。

オプションのキーワード専用引数 *default* は、現在のコンテキストにその変数の値が見付からなかったときに *ContextVar.get()* から返されます。

重要: コンテキスト変数は、モジュールのトップレベルで生成する必要があり、クロージャの中で作成すべきではありません。*Context* オブジェクトはコンテキスト変数への強参照を持っており、コンテキスト変数がガーベジコレクトされるのを防ぎます。

name

変数の名前。読み出し専用のプロパティです。

バージョン 3.7.1 で追加.

get([*default*])

現在のコンテキストのコンテキスト変数の値を返します。

現在のコンテキストのコンテキスト変数に値がなければ、メソッドは:

- メソッドの *default* 引数に値が指定されていればその値を返します。さもなければ
- コンテキスト変数が生成された時にデフォルト値が渡されていれば、その値を返します。さもなければ
- *LookupError* を送出します。

set(*value*)

現在のコンテキストのコンテキスト変数に新しい値を設定する際に呼び出します。

value は必須の引数で、コンテキスト変数の新しい値を指定します。

Token オブジェクトを返します。このオブジェクトを *ContextVar.reset()* メソッドに渡すことで、以前の値に戻すことができます。

reset(*token*)

コンテキスト変数の値を、*token* を生成した *ContextVar.set()* が呼び出される前の値にリセットします。

例えば:

```
var = ContextVar('var')

token = var.set('new value')
# code that uses 'var'; var.get() returns 'new value'.
var.reset(token)

# After the reset call the var has no value again, so
# var.get() would raise a LookupError.
```

class contextvars.**Token**

Token オブジェクトは、*ContextVar.set()* メソッドによって返されるオブジェクトです。このオブジェクトを *ContextVar.reset()* メソッドに渡すことで、対応する *set* を呼び出す前のコンテキスト変数の値に戻せます。

var

読み出し専用のプロパティです。トークンを生成した *ContextVar* オブジェクトを指します。

old_value

読み出し専用のプロパティです。このトークンを返した *ContextVar.set()* メソッドの呼び出し前に設定されていた値を返します。もし呼び出しの前に値が設定されていなければ *Token.MISSING* を返します。

MISSING

Token.old_value で利用されるマーカーオブジェクトです。

17.9.2 マニュアルでのコンテキスト管理

contextvars.**copy_context**()

現在の *Context* オブジェクトのコピーを返します。

次のスニペットは、現在のコンテキストのコピーを取得し、コンテキストに設定されているすべての変数とその値を表示します:

```
ctx: Context = copy_context()
print(list(ctx.items()))
```

この関数の複雑性は O(1) です。つまり、少数のコンテキスト変数を持つコンテキストと多くの変数を持つコンテキストで同程度の速度で動作します。

class contextvars.Context

ContextVars とその値の対応付け。

Context() は、値を持たない空のコンテキストを生成します。現在のコンテキストのコピーを得るには、*copy_context()* 関数を利用します。

Context は、*collections.abc.Mapping* インタフェースを実装します。

run(*callable*, **args*, ***kwargs*)

callable(*args, **kwargs) を *run* メソッドが呼ばれたコンテキストオブジェクトの中で実行します。実行した結果を返すか、例外が発生した場合はその例外を伝播します。

callable が行ったコンテキスト変数へのいかなる変更も、コンテキストオブジェクトに格納されます:

```
var = ContextVar('var')
var.set('spam')

def main():
    # 'var' was set to 'spam' before
    # calling 'copy_context()' and 'ctx.run(main)', so:
    # var.get() == ctx[var] == 'spam'

    var.set('ham')

    # Now, after setting 'var' to 'ham':
    # var.get() == ctx[var] == 'ham'

ctx = copy_context()

# Any changes that the 'main' function makes to 'var'
# will be contained in 'ctx'.
ctx.run(main)

# The 'main()' function was run in the 'ctx' context,
# so changes to 'var' are contained in it:
# ctx[var] == 'ham'

# However, outside of 'ctx', 'var' is still set to 'spam':
# var.get() == 'spam'
```

2 つ以上の OS スレッドから同一のコンテキストオブジェクトを呼び出すか、再帰的に呼び出したとき、メソッドは *RuntimeError* を送出します。

copy()

コンテキストオブジェクトの浅いコピーを返します。

var in context

context に *var* の値が設定されていた場合 True を返します; そうでない場合は False を返します。

context[var]

ContextVar *var* の値を返します。コンテキストオブジェクト内で変数が設定されていない場合は、*KeyError* を送出します。

get(*var*[, *default*])
var がコンテキストオブジェクトの中に値を持てば、その値を返します。さもなければ、*default* を返します。*default* を指定していなければ、None を返します。

iter(context)
コンテキストオブジェクトに格納されている変数群のイテレータを返します。

len(proxy)
コンテキストオブジェクトに格納されている変数の数を返します。

keys()
コンテキストオブジェクト中のすべての変数のリストを返します。

values()
コンテキストオブジェクト中のすべての変数の値のリストを返します。

items()
コンテキストオブジェクト中のすべての変数について、変数とその値からなる 2 要素のタプルのリストを返します。

17.9.3 asyncio サポート

コンテキスト変数は、追加の設定なしに *asyncio* をサポートします。例えば、次の単純な echo サーバーは、クライアントを扱う Task の中でリモートクライアントのアドレスが利用できるように、コンテキスト変数を利用します:

```
import asyncio
import contextvars

client_addr_var = contextvars.ContextVar('client_addr')

def render_goodbye():
    # The address of the currently handled client can be accessed
    # without passing it explicitly to this function.

    client_addr = client_addr_var.get()
    return f'Good bye, client @ {client_addr}\n'.encode()

async def handle_request(reader, writer):
    addr = writer.transport.get_extra_info('socket').getpeername()
    client_addr_var.set(addr)

    # In any code that we call is now possible to get
    # client's address by calling 'client_addr_var.get()'.

    while True:
```

(次のページに続く)

(前のページからの続き)

```
        line = await reader.readline()
        print(line)
        if not line.strip():
            break
        writer.write(line)

    writer.write(render_goodbye())
    writer.close()

async def main():
    srv = await asyncio.start_server(
        handle_request, '127.0.0.1', 8081)

    async with srv:
        await srv.serve_forever()

asyncio.run(main())

# To test it you can use telnet:
#     telnet 127.0.0.1 8081
```

以下のモジュールは上記のサービスの一部で使われるサポートモジュールです:

17.10 _thread --- 低水準の スレッド API

このモジュールはマルチスレッド (別名 **軽量プロセス** (*light-weight processes*) または **タスク** (*tasks*)) に用いられる低水準プリミティブを提供します --- グローバルデータ空間を共有するマルチスレッドを制御します。同期のための単純なロック (別名 *mutexes* またはバイナリセマフォ (*binary semaphores*)) が提供されています。*threading* モジュールは、このモジュール上で、より使い易く高級なスレッディングの API を提供します。

バージョン 3.7 で変更: このモジュールは以前はオプションでしたが、常に利用可能なモジュールとなりました。

このモジュールでは以下の定数および関数を定義しています:

exception _thread.error

スレッド固有の例外です。

バージョン 3.3 で変更: 現在は組み込みの *RuntimeError* の別名です。

_thread.LockType

これはロックオブジェクトのタイプです。

_thread.start_new_thread(*function*, *args*[, *kwargs*])

新しいスレッドを開始して、その ID を返します。スレッドは引数リスト *args* (タプルでなければな

りません) の関数 *function* を実行します。オプション引数 *kwargs* はキーワード引数の辞書を指定します。

関数が戻るとき、スレッドは静かに終了します。

When the function terminates with an unhandled exception, *sys.unraisablehook()* is called to handle the exception. The *object* attribute of the hook argument is *function*. By default, a stack trace is printed and then the thread exits (but other threads continue to run).

When the function raises a *SystemExit* exception, it is silently ignored.

バージョン 3.8 で変更: *sys.unraisablehook()* is now used to handle unhandled exceptions.

_thread.interrupt_main()

Simulate the effect of a *signal.SIGINT* signal arriving in the main thread. A thread can use this function to interrupt the main thread.

signal.SIGINT が Python に対処されなかった (*signal.SIG_DFL* または *signal.SIG_IGN* に設定されていた) 場合、この関数は何もしません。

_thread.exit()

SystemExit を送出します。それが捕えられないときは、静かにスレッドを終了させます。

_thread.allocate_lock()

新しいロックオブジェクトを返します。ロックのメソッドはこの後に記述されます。ロックは初期状態としてアンロック状態です。

_thread.get_ident()

現在のスレッドの ' スレッド ID' を返します。非ゼロの整数です。この値は直接の意味を持っていません; 例えばスレッド特有のデータの辞書に索引をつけるためのような、マジッククッキーとして意図されています。スレッドが終了し、他のスレッドが作られたとき、スレッド ID は再利用されるかもしれません。

_thread.get_native_id()

Return the native integral Thread ID of the current thread assigned by the kernel. This is a non-negative integer. Its value may be used to uniquely identify this particular thread system-wide (until the thread terminates, after which the value may be recycled by the OS).

利用可能な環境: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX。

バージョン 3.8 で追加.

_thread.stack_size([*size*])

新しいスレッドを作るときのスレッドスタックサイズを返します。オプションの *size* 引数にはこれ以降に作成するスレッドのスタックサイズを指定し、0 (プラットフォームのデフォルト値または設定されたデフォルト値) か、32,768 (32 KiB) 以上の正の整数でなければなりません。*size* が指定されない場合 0 が使われます。スレッドのスタックサイズの変更がサポートされていない場合、*RuntimeError* を送出します。不正なスタックサイズが指定された場合、*ValueError* を送出して、スタックサイズは変更されません。32 KiB は現在のインタープリタ自身のために十分であると保証された最小のスタックサイズです。いくつかのプラットフォームではスタックサイズに対して制限があることに注意してく

ださい。例えば最小のスタックサイズが 32 KiB より大きかったり、システムのメモリページサイズ の整数倍の必要があるなどです。この制限についてはプラットフォームのドキュメントを参照してください (一般的なページサイズは 4 KiB なので、プラットフォームに関する情報がない場合は 4096 の整数倍のスタックサイズを選ぶといいかもしれません)。

Availability: Windows, systems with POSIX threads.

_thread.TIMEOUT_MAX

`Lock.acquire()` の *timeout* 引数に許される最大値です。これ以上の値を timeout に指定すると *OverflowError* を発生させます。

バージョン 3.2 で追加.

ロックオブジェクトは次のようなメソッドを持っています:

lock.acquire(*waitflag=1*, *timeout=-1*)

オプションの引数なしで使用すると、このメソッドは他のスレッドがロックしているかどうかにかかわらずロックを獲得します。ただし、他のスレッドがすでにロックしている場合には解除されるまで待ってからロックを獲得します (同時にロックを獲得できるスレッドはひとつだけであり、これこそがロックの存在理由です)。

整数の引数 *waitflag* を指定すると、その値によって動作が変わります。引数が `0` のときは、待たずにすぐ獲得できる場合にだけロックを獲得します。`0` 以外の値を与えると、先の例と同様、ロックの状態にかかわらず獲得をおこないます。

timeout 引数に正の float 値が指定された場合、返る前に待つ最大の時間を秒数で指定します。負の *timeout* 引数は無制限に待つことを指定します。*waitflag* が 0 の時は *timeout* を指定することはできません。

なお、ロックを獲得できた場合は `True`、できなかった場合は `False` を返します。

バージョン 3.2 で変更: 新しい *timeout* 引数。

バージョン 3.2 で変更: POSIX ではロックの取得がシグナルに割り込まれるようになりました。

lock.release()

ロックを解放します。そのロックは既に獲得されたものでなければなりませんが、しかし同じスレッドによって獲得されたものである必要はありません。

lock.locked()

ロックの状態を返します: 同じスレッドによって獲得されたものなら `True` 、違うのなら `False` を返します。

これらのメソッドに加えて、ロックオブジェクトは `with` 文を通じて以下の例のように使うこともできます。

```
import _thread

a_lock = _thread.allocate_lock()

with a_lock:
    print("a_lock is locked while this executes")
```

警告:

- スレッドは割り込みと奇妙な相互作用をします: *KeyboardInterrupt* 例外は任意のスレッドによって受け取られます。(*signal* モジュールが利用可能なとき、割り込みは常にメインスレッドへ行きます。)
- *sys.exit()* を呼び出す、あるいは *SystemExit* 例外を送出することは、*_thread.exit()* を呼び出すことと同じです。
- ロックの `acquire()` メソッドに割り込むことはできません --- *KeyboardInterrupt* 例外は、ロックが獲得された後に発生します。
- メインスレッドが終了したとき、他のスレッドが生き残るかどうかは、システムに依存します。多くのシステムでは、`try ... finally` 節や、オブジェクトデストラクタを実行せずに終了されます。
- メインスレッドが終了したとき、それの通常のクリーンアップは行なわれず、(`try ... finally` 節が尊重されることは除きます)、標準 I/O ファイルはフラッシュされません。

17.11 _dummy_thread --- _thread の代替モジュール

ソースコード: Lib/_dummy_thread.py

バージョン 3.7 で非推奨: Python は常にスレッドが使えるようになりました。代わりに mod:*_thread* (あるいは、それより良い *threading*) を使ってください。

このモジュールは *_thread* モジュールのインターフェースをそっくりまねるものです。*_thread* モジュールがサポートされていなかったプラットフォームで import することを意図して作られたものでした。

生成するスレッドが、他のブロックしたスレッドを待ち、デッドロック発生の可能性がある場合には、このモジュールを使わないようにしてください。ブロッキング I/O を使っている場合によく起きます。

17.12 dummy_threading --- threading の代替モジュール

ソースコード: Lib/dummy_threading.py

バージョン 3.7 で非推奨: Python は常にスレッドが使えるようになりました。代わりに mod:*threading* を使ってください。

このモジュールは *threading* モジュールのインターフェースをそっくりまねるものです。*_thread* モジュールがサポートされていなかったプラットフォームで import することを意図して作られたものでした。

生成するスレッドが、他のブロックしたスレッドを待ち、デッドロック発生の可能性がある場合には、このモジュールを使わないようにしてください。ブロッキング I/O を使っている場合によく起きます。

第

EIGHTEEN

ネットワーク通信とプロセス間通信

この章で解説しているモジュールはネットワーク通信とプロセス間通信の仕組みを提供しています。

例えば *signal* や *mmap* のように、同じマシン上の 2 つのプロセスでしか使えないモジュールがあります。その他のモジュールは 2 つ以上のプロセスを使ってマシン間で通信できるネットワークプロトコルをサポートします。

この章で解説されるモジュールのリスト:

18.1 asyncio --- 非同期 I/O

Hello World!

```
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

# Python 3.7+
asyncio.run(main())
```

asyncio は **async/await** 構文を使い **並行処理の** コードを書くためのライブラリです。

asyncio は、高性能なネットワークとウェブサーバ、データベース接続ライブラリ、分散タスクキューなどの複数の非同期 Python フレームワークの基盤として使われています。

asyncio は多くの場合、IO バウンドだったり高レベルの **構造化された** ネットワークコードに完璧に適しています。

asyncio は次の目的で **高レベル** API を提供しています:

- 並行に *Python* **コルーチンを起動** し、実行全体を管理する
- **ネットワーク** *IO* **と** *IPC* を執り行う

- *subprocesses* を管理する
- **キュー** を使ってタスクを分散する
- 並列処理のコードを **同期** させる

これに加えて、**ライブラリやフレームワークの開発者** が次のことをするための **低レベル** API があります:

- **ネットワーク通信** 、**サブプロセス** の実行、*OS* **シグナル** の取り扱いなどのための非同期 API を提供する **イベントループ** の作成と管理を行う
- *Transport* を使った効率的な protocol を実装します
- コールバックを用いたライブラリと async/await 構文を使ったコードの **橋渡し**

リファレンス

18.1.1 コルーチンと Task

この節では、コルーチンと Task を利用する高レベルの asyncio の API の概略を解説します。

- **コルーチン**
- *Awaitable*
- **非同期プログラムの実行**
- *Task* **の作成**
- **スリープ**
- **並行な** *Task* **実行**
- **キャンセルからの保護**
- **タイムアウト**
- **要素の終了待機**
- **外部スレッドからのスケジュール**
- **イントロスペクション**
- *Task* **オブジェクト**
- *Generator-based Coroutines*

コルーチン

async/await 構文で宣言された **コルーチン** は、asyncio を使ったアプリケーションを書くのに推奨される方法です。例えば、次のコードスニペット (Python 3.7 以降が必要) は "hello" を出力し、そこから 1 秒待って "world" を出力します:

```
>>> import asyncio

>>> async def main():
...     print('hello')
...     await asyncio.sleep(1)
...     print('world')

>>> asyncio.run(main())
hello
world
```

単にコルーチンを呼び出しただけでは、コルーチンの実行スケジュールは予約されていないことに注意してください:

```
>>> main()
<coroutine object main at 0x1053bb7c8>
```

実際にコルーチンを走らせるために、asyncio は 3 つの機構を提供しています:

- 最上位のエントリーポイントである "main()" 関数を実行する *asyncio.run()* 関数 (上の例を参照してください。)
- コルーチンを await すること。次のコード片は 1 秒間待機した後に "hello" と出力し、**更に** 2 秒間待機してから "world" と出力します:

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")

    await say_after(1, 'hello')
    await say_after(2, 'world')

    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

予想される出力:

```
started at 17:13:52
hello
world
finished at 17:13:55
```

- asyncio の *Tasks* としてコルーチンを並行して走らせる *asyncio.create_task()* 関数。

  上のコード例を編集して、ふたつの say_after コルーチンを **並行して** 走らせてみましょう:

```
async def main():
    task1 = asyncio.create_task(
        say_after(1, 'hello'))

    task2 = asyncio.create_task(
        say_after(2, 'world'))

    print(f"started at {time.strftime('%X')}")

    # Wait until both tasks are completed (should take
    # around 2 seconds.)
    await task1
    await task2

    print(f"finished at {time.strftime('%X')}")
```

  予想される出力が、スニペットの実行が前回よりも 1 秒早いことを示していることに注意してください:

```
started at 17:14:32
hello
world
finished at 17:14:34
```

Awaitable

あるオブジェクトを await 式の中で使うことができる場合、そのオブジェクトを **awaitable** オブジェクトと言います。多くの asyncio API は awaitable を受け取るように設計されています。

awaitable オブジェクトには主に 3 つの種類があります: **コルーチン**, **Task**, そして **Future** です

コルーチン

Python のコルーチンは *awaitable* であり、そのため他のコルーチンを待機させられます:

```
import asyncio

async def nested():
    return 42
```

(次のページに続く)

(前のページからの続き)

```
async def main():
    # Nothing happens if we just call "nested()".
    # A coroutine object is created but not awaited,
    # so it *won't run at all*.
    nested()

    # Let's do it differently now and await it:
    print(await nested())  # will print "42".

asyncio.run(main())
```

重要: このドキュメントにおいて「コルーチン」という用語は以下 2 つの密接に関連した概念に対して使用できます:

- **コルーチン関数**: async def 関数;
- **コルーチンオブジェクト**: **コルーチン関数** を呼び出すと返ってくるオブジェクト.

---

asyncio は、古くからある **ジェネレータベース** のコルーチンもサポートしています。

Task

Task は、コルーチンを **並行に** スケジュールするのに使います。

asyncio.create_task() のような関数で、コルーチンが *Task* にラップされているとき、自動的にコルーチンは即時実行されるようにスケジュールされます:

```
import asyncio

async def nested():
    return 42

async def main():
    # Schedule nested() to run soon concurrently
    # with "main()".
    task = asyncio.create_task(nested())

    # "task" can now be used to cancel "nested()", or
    # can simply be awaited to wait until it is complete:
    await task

asyncio.run(main())
```

Future

Future は、非同期処理の **最終結果** を表現する特別な **低レベルの** awaitable オブジェクトです。

Future オブジェクトが他の awaitable を **待機させている** と言うときは、ある場所で Future が解決されるまでコルーチンが待機するということです。

asyncio の Future オブジェクトを使うと、async/await とコールバック形式のコードを併用できます。

通常、アプリケーション水準のコードで Future オブジェクトを作る **必要はありません** 。

Future オブジェクトはライブラリや asyncio の API で表に出ることもあり、他の awaitable を待機させられます:

```
async def main():
    await function_that_returns_a_future_object()

    # this is also valid:
    await asyncio.gather(
        function_that_returns_a_future_object(),
        some_python_coroutine()
    )
```

Future オブジェクトを返す低レベル関数の良い例は *loop.run_in_executor()* です。

非同期プログラムの実行

asyncio.**run**(*coro*, ***, *debug=False*)

coroutine *coro* を実行し、結果を返します。

この関数は、非同期イベントループの管理と **非同期ジェネレータの終了処理** を行いながら、渡されたコルーチンを実行します。

この関数は、同じスレッドで他の非同期イベントループが実行中のときは呼び出せません。

debug が `True` の場合、イベントループはデバッグモードで実行されます。

この関数は常に新しいイベントループを作成し、終了したらそのイベントループを閉じます。この関数は非同期プログラムのメインのエントリーポイントとして使われるべきで、理想的には 1 回だけ呼び出されるべきです。

以下はプログラム例です:

```
async def main():
    await asyncio.sleep(1)
    print('hello')

asyncio.run(main())
```

バージョン 3.7 で追加.

注釈: asyncio.run() のソースコードは Lib/asyncio/runners.py にあります。

Task の作成

asyncio.**create_task**(*coro*, ***, *name=None*)

coro *coroutine* を *Task* でラップし、その実行をスケジュールします。Task オブジェクトを返します。

もし *name* が None でない場合、*Task.set_name()* を使用し、*name* がタスクの名前として設定されます。

その Task オブジェクトは *get_running_loop()* から返されたループの中で実行されます。現在のスレッドに実行中のループが無い場合は、*RuntimeError* が送出されます。

この関数は **Python 3.7 で追加** されました。Python 3.7 より前では、代わりに低レベルの *asyncio.ensure_future()* 関数が使えます:

```
async def coro():
    ...

# In Python 3.7+
task = asyncio.create_task(coro())
...

# This works in all Python versions but is less readable
task = asyncio.ensure_future(coro())
...
```

バージョン 3.7 で追加.

バージョン 3.8 で変更: name パラメータが追加されました。

スリープ

coroutine asyncio.**sleep**(*delay*, *result=None*, ***, *loop=None*)

delay 秒だけ停止します。

result が提供されている場合は、コルーチン完了時にそれが呼び出し元に返されます。

sleep() は常に現在の Task を一時中断し、他の Task が実行されるのを許可します。

Deprecated since version 3.8, will be removed in version 3.10: *loop* パラメータ。 現在の時刻を 5 秒間、毎秒表示するコルーチンの例:

```
import asyncio
import datetime

async def display_date():
    loop = asyncio.get_running_loop()
```

(次のページに続く)

(前のページからの続き)

```
    end_time = loop.time() + 5.0
    while True:
        print(datetime.datetime.now())
        if (loop.time() + 1.0) >= end_time:
            break
        await asyncio.sleep(1)

asyncio.run(display_date())
```

並行な Task 実行

awaitable asyncio.gather(**aws*, *loop=None*, *return_exceptions=False*)

aws シーケンスにある *awaitable* **オブジェクト** を **並行** 実行します。

aws にある awaitable がコルーチンである場合、自動的に Task としてスケジュールされます。

全ての awaitable が正常終了した場合、その結果は返り値を集めたリストになります。返り値の順序は、*aws* での awaitable の順序に相当します。

return_exceptions が `False` である場合 (デフォルト)、`gather()` で await しているタスクに対して、最初の例外が直接伝えられます。*aws* に並んでいる他の awaitable は、**キャンセルされずに** 引き続いて実行されます。

return_exceptions が `True` だった場合、例外は成功した結果と同じように取り扱われ、結果リストに集められます。

`gather()` が **キャンセル** された場合、起動された全ての (未完了の) awaitable も **キャンセル** されます。

aws シーケンスにある Task あるいは Future が **キャンセル** された場合、*CancelledError* を送出したかのうように扱われます。つまり、この場合 `gather()` 呼び出しはキャンセル **されません**。これは、起動された 1 つの Task あるいは Future のキャンセルが、他の Task あるいは Future のキャンセルを引き起こすのを避けるためです。

Deprecated since version 3.8, will be removed in version 3.10: *loop* パラメータ。 以下はプログラム例です:

```
import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute factorial({i})...")
        await asyncio.sleep(1)
        f *= i
    print(f"Task {name}: factorial({number}) = {f}")

async def main():
    # Schedule three calls *concurrently*:
```

(次のページに続く)

(前のページからの続き)

```
    await asyncio.gather(
        factorial("A", 2),
        factorial("B", 3),
        factorial("C", 4),
    )

asyncio.run(main())

# Expected output:
#
#     Task A: Compute factorial(2)...
#     Task B: Compute factorial(2)...
#     Task C: Compute factorial(2)...
#     Task A: factorial(2) = 2
#     Task B: Compute factorial(3)...
#     Task C: Compute factorial(3)...
#     Task B: factorial(3) = 6
#     Task C: Compute factorial(4)...
#     Task C: factorial(4) = 24
```

注釈: *return_exceptions* が False の場合、いったん完了状態となった gather() をキャンセルしても起動された awaitables がキャンセルされないことがあります。例えば、gather は例外を呼び出し元に送出したあと完了状態になることがあるため、(起動した awaitable のいずれかから送出された) gather からの例外をキャッチした後で `gather.cancel()` を呼び出しても、他の awaitable がキャンセルされない可能性があります。

バージョン 3.7 で変更: *gather* 自身がキャンセルされた場合は、*return_exceptions* の値に関わらずキャンセルが伝搬されます。

キャンセルからの保護

awaitable asyncio.shield(*aw*, ***, *loop=None*)

キャンセル から *awaitable* **オブジェクト** を保護します。

aw がコルーチンだった場合、自動的に Task としてスケジュールされます。

文:

```
res = await shield(something())
```

は、以下と同じです

```
res = await something()
```

それを含むコルーチンがキャンセルされた場合を **除き**、`something()` 内で動作している Task はキャンセルされません。`something()` 側から見るとキャンセルは発生しません。呼び出し元がキャンセル

された場合でも、”await” 式は *CancelledError* を送出します。

注意: `something()` が他の理由 (例えば、原因が自分自身) でキャンセルされた場合は `shield()` でも保護できません。

完全にキャンセルを無視したい場合 (推奨はしません) は、`shield()` 関数は次のように try/except 節と組み合わせることになるでしょう:

```
try:
    res = await shield(something())
except CancelledError:
    res = None
```

Deprecated since version 3.8, will be removed in version 3.10: *loop* パラメータ。

タイムアウト

coroutine asyncio.**wait_for**(*aw*, *timeout*, ***, *loop=None*)

aw *awaitable* が、完了するかタイムアウトになるのを待ちます。

aw がコルーチンだった場合、自動的に Task としてスケジュールされます。

timeout には `None` もしくは待つ秒数の浮動小数点数か整数を指定できます。*timeout* が `None` の場合、Future が完了するまで待ちます。

タイムアウトが起きた場合は、Task をキャンセルし *asyncio.TimeoutError* を送出します。

Task の **キャンセル** を避けるためには、*shield()* の中にラップしてください。

この関数は、Future が実際にキャンセルされるまで待つので、全体の待ち時間は *timeout* を超えることもあります。

待機が中止された場合 *aw* も中止されます。

Deprecated since version 3.8, will be removed in version 3.10: *loop* パラメータ。 以下はプログラム例です:

```
async def eternity():
    # Sleep for one hour
    await asyncio.sleep(3600)
    print('yay!')

async def main():
    # Wait for at most 1 second
    try:
        await asyncio.wait_for(eternity(), timeout=1.0)
    except asyncio.TimeoutError:
        print('timeout!')

asyncio.run(main())
```

(次のページに続く)

(前のページからの続き)

```
# Expected output:
#
#     timeout!
```

バージョン 3.7 で変更: *aw* がタイムアウトでキャンセルされたとき、`wait_for` は *aw* がキャンセルされるまで待ちます。以前は、すぐに *asyncio.TimeoutError* を送出していました。

要素の終了待機

coroutine asyncio.wait(*aws*, ***, *loop=None*, *timeout=None*, *return_when=ALL_COMPLETED*)

Run *awaitable objects* in the *aws* iterable concurrently and block until the condition specified by *return_when*.

Task と Future の 2 つ (`done`, `pending`) を返します。

使い方:

```
done, pending = await asyncio.wait(aws)
```

timeout (浮動小数点数または整数) が指定されていたら、処理を返すのを待つ最大秒数を制御するのに使われます。

この関数は *asyncio.TimeoutError* を送出しないことに注意してください。タイムアウトが起きたときに完了していなかった Future や Task は、2 つ目の集合の要素として返されるだけです。

return_when でこの関数がいつ結果を返すか指定します。指定できる値は以下の 定数のどれか一つです:

定数	説明
`FIRST_COMPLETED`	いずれかの Future が終了したかキャンセルされたときに返します。
`FIRST_EXCEPTION`	いずれかの Future が例外の送出で終了した場合に返します。例外を送出したフューチャがない場合は、`ALL_COMPLETED` と等価になります。
`ALL_COMPLETED`	すべての Future が終了したかキャンセルされたときに返します。

wait_for() と異なり、`wait()` はタイムアウトが起きたときに Future をキャンセルしません。

バージョン 3.8 で非推奨: *aws* にある awaitable のどれかがコルーチンの場合、自動的に Task としてスケジュールされます。コルーチンオブジェクトを `wait()` に直接渡すのは **紛らわしい振る舞い** を引き起こすため非推奨です。

Deprecated since version 3.8, will be removed in version 3.10: *loop* パラメータ。

注釈: `wait()` は自動的にコルーチンを Task としてスケジュールし、その後、暗黙的に作成された Task オブジェクトを組になった集合 (`done`, `pending`) に入れて返します。従って、次のコードは予想した通りには動作しません:

```
async def foo():
    return 42

coro = foo()
done, pending = await asyncio.wait({coro})

if coro in done:
    # This branch will never be run!
```

上のスクリプト片は次のように修正できます:

```
async def foo():
    return 42

task = asyncio.create_task(foo())
done, pending = await asyncio.wait({task})

if task in done:
    # Everything will work as expected now.
```

バージョン 3.8 で非推奨: `wait()` にコルーチンオブジェクトを直接渡すのは非推奨です。

asyncio.**as_completed**(*aws*, ***, *loop=None*, *timeout=None*)

イテラブル *aws* 内の *awaitable* **オブジェクト** を並列実行します。コルーチンのイテレータを返します。戻り値の各コルーチンは、残りの awaitable のうちで最も早く得られた結果を待ち受けることができます。

全フューチャが終了する前にタイムアウトが発生した場合 *asyncio.TimeoutError* を送出します。

Deprecated since version 3.8, will be removed in version 3.10: *loop* パラメータ。

以下はプログラム例です:

```
for coro in as_completed(aws):
    earliest_result = await coro
    # ...
```

外部スレッドからのスケジュール

asyncio.**run_coroutine_threadsafe**(*coro*, *loop*)

与えられたイベントループにコルーチンを送ります。この処理は、スレッドセーフです。

他の OS スレッドから結果を待つための *concurrent.futures.Future* を返します。

この関数は、イベントループが動作しているスレッドとは異なる OS スレッドから呼び出すためのものです。例えば次のように使います:

```
# Create a coroutine
coro = asyncio.sleep(1, result=3)

# Submit the coroutine to a given loop
future = asyncio.run_coroutine_threadsafe(coro, loop)

# Wait for the result with an optional timeout argument
assert future.result(timeout) == 3
```

コルーチンから例外が送出された場合、返された Future に通知されます。これはイベントループの Task をキャンセルするのにも使えます:

```
try:
    result = future.result(timeout)
except asyncio.TimeoutError:
    print('The coroutine took too long, cancelling the task...')
    future.cancel()
except Exception as exc:
    print(f'The coroutine raised an exception: {exc!r}')
else:
    print(f'The coroutine returned: {result!r}')
```

このドキュメントの *asyncio-multithreading* 節を参照してください。

他の asyncio 関数とは異なり、この関数は明示的に渡される *loop* 引数を必要とします。

バージョン 3.5.1 で追加.

イントロスペクション

asyncio.**current_task**(*loop=None*)

現在実行中の *Task* インスタンスを返します。実行中の Task が無い場合は None を返します。

loop が None の場合、*get_running_loop()* が現在のループを取得するのに使われます。

バージョン 3.7 で追加.

asyncio.**all_tasks**(*loop=None*)

ループで実行された *Task* オブジェクトでまだ完了していないものの集合を返します。

loop が None の場合、*get_running_loop()* は現在のループを取得するのに使われます。

バージョン 3.7 で追加.

Task オブジェクト

class asyncio.Task(*coro*, ***, *loop=None*, *name=None*)

Python コルーチン を実行する *Future* 類 オブジェクトです。スレッドセーフではありません。

Task はイベントループのコルーチンを実行するのに使われます。Future でコルーチンが待機している場合、Task は自身のコルーチンの実行を一時停止させ、Future の完了を待ちます。Future が **完了** したら、Task が内包しているコルーチンの実行を再開します。

イベントループは協調スケジューリングを使用します。つまり、イベントループは同時に 1 つの Task のみ実行します。Task が Future の完了を待っているときは、イベントループは他の Task やコールバックを動作させるか、IO 処理を実行します。

Task を作成するには高レベルの *asyncio.create_task()* 関数、あるいは低レベルの *loop.create_task()* 関数や *ensure_future()* 関数を使用してください。手作業での Task の実装は推奨されません。

実行中のタスクをキャンセルするためには、*cancel()* メソッドを使用します。このメソッドを呼ぶと、タスクはそれを内包するコルーチンに対して *CancelledError* 例外を送出します。キャンセルの際にコルーチンが Future オブジェクトを待っていた場合、その Future オブジェクトはキャンセルされます。

cancelled() は、タスクがキャンセルされたかを調べるのに使用できます。タスクを内包するコルーチンで *CancelledError* 例外が抑制されておらず、かつタスクが実際にキャンセルされている場合に、このメソッドは `True` を変えます。

asyncio.Task は、*Future.set_result()* と *Future.set_exception()* を除いて、*Future* の API をすべて継承しています。

Task は *contextvars* モジュールをサポートします。Task が作られたときに現在のコンテキストがコピーされ、のちに Task のコルーチンを実行する際に、コピーされたコンテキストが使用されます。

バージョン 3.7 で変更: *contextvars* モジュールのサポートを追加。

バージョン 3.8 で変更: `name` パラメータが追加されました。

Deprecated since version 3.8, will be removed in version 3.10: *loop* パラメータ。

cancel()

このタスクに、自身のキャンセルを要求します。

このメソッドは、イベントループの次のステップにおいて、タスクがラップしているコルーチン内で *CancelledError* 例外が送出されるように準備します。

コルーチン側では `try` ... ... `except CancelledError` ... `finally` ブロックで例外を処理することにより、クリーンアップ処理を行なったり、リクエストを拒否したりする機会が与えられます。この特性を使ってキャンセル処理を完全に抑え込むことも可能であることから、*Future.cancel()* と異なり、*Task.cancel()* は Task が実際にキャンセルされることを保証しません。ただしそのような処理は一般的ではありませんし、そのような処理をしないことが望ましいです。

以下の例は、コルーチンがどのようにしてキャンセルのリクエストを阻止するかを示しています:

```
async def cancel_me():
    print('cancel_me(): before sleep')

    try:
        # Wait for 1 hour
        await asyncio.sleep(3600)
    except asyncio.CancelledError:
        print('cancel_me(): cancel sleep')
        raise
    finally:
        print('cancel_me(): after sleep')

async def main():
    # Create a "cancel_me" Task
    task = asyncio.create_task(cancel_me())

    # Wait for 1 second
    await asyncio.sleep(1)

    task.cancel()
    try:
        await task
    except asyncio.CancelledError:
        print("main(): cancel_me is cancelled now")

asyncio.run(main())

# Expected output:
#
#     cancel_me(): before sleep
#     cancel_me(): cancel sleep
#     cancel_me(): after sleep
#     main(): cancel_me is cancelled now
```

cancelled()

Task が **キャンセルされた** 場合に True を返します。

cancel() メソッドによりキャンセルがリクエストされ、かつ Task がラップしているコルーチンが内部で送出された *CancelledError* 例外を伝達したとき、Task は実際に **キャンセル** されます。

done()

Task が **完了** しているなら True を返します。

Task がラップしているコルーチンが値を返すか、例外を送出するか、または Task がキャンセルされたとき、Task は **完了** します。

result()

Task の結果を返します。

Task が **完了** している場合、ラップしているコルーチンの結果が返されます (コルーチンが例外を送出された場合、その例外が例外が再送出されます)

Task が **キャンセル** されている場合、このメソッドは *CancelledError* 例外を送出します。

Task の結果がまだ未設定の場合、このメソッドは *InvalidStateError* 例外を送出します。

exception()
Task の例外を返します。

ラップされたコルーチンが例外を送出した場合、その例外が返されます。ラップされたコルーチンが正常終了した場合、このメソッドは None を返します。

Task が **キャンセル** されている場合、このメソッドは *CancelledError* 例外を送出します。

Task がまだ **完了** していない場合、このメソッドは *InvalidStateError* 例外を送出します。

add_done_callback(*callback*, ***, *context=None*)
Task が **完了** したときに実行されるコールバックを追加します。

このメソッドは低水準のコールバックベースのコードでのみ使うべきです。

詳細については *Future.add_done_callback()* のドキュメントを参照してください。

remove_done_callback(*callback*)
コールバックリストから *callback* を削除します。

このメソッドは低水準のコールバックベースのコードでのみ使うべきです。

詳細については *Future.remove_done_callback()* のドキュメントを参照してください。

get_stack(***, *limit=None*)
このタスクのスタックフレームのリストを返します。

コルーチンが完了していない場合、これはサスペンドされた時点でのスタックを返します。コルーチンが正常に処理を完了したか、キャンセルされていた場合は空のリストを返します。コルーチンが例外で終了した場合はトレースバックフレームのリストを返します。

フレームは常に古いものから新しい物へ並んでいます。

サスペンドされているコルーチンの場合スタックフレームが 1 個だけ返されます。

オプション引数 *limit* は返すフレームの最大数を指定します; デフォルトでは取得可能な全てのフレームを返します。返されるリストの順番は、スタックが返されるか、トレースバックが返されるかによって変わります: スタックでは新しい順に並んだリストが返されますが、トレースバックでは古い順に並んだリストが返されます（これは traceback モジュールの振る舞いと一致します）。

print_stack(***, *limit=None*, *file=None*)
このタスクのスタックまたはトレースバックを出力します。

このメソッドは *get_stack()* によって取得されるフレームに対し、traceback モジュールと同じような出力を生成します。

引数 *limit* は *get_stack()* にそのまま渡されます。

引数 *file* は出力を書き込む I/O ストリームを指定します; デフォルトでは出力は標準エラー出力 *sys.stderr* に書き込まれます。

get_coro()

Task がラップしているコルーチンオブジェクトを返します。

バージョン 3.8 で追加.

get_name()

Task の名前を返します。

Task に対して明示的に名前が設定されていない場合, デフォルトの asyncio Task 実装はタスクをインスタンス化する際にデフォルトの名前を生成します。

バージョン 3.8 で追加.

set_name(*value*)

Task に名前を設定します。

引数 *value* は文字列に変換可能なオブジェクトであれば何でもかまいません。

Task のデフォルト実装では、名前はオブジェクトの *repr()* メソッドの出力で確認できます。

バージョン 3.8 で追加.

classmethod all_tasks(*loop=None*)

イベントループのすべての Task の集合を返します。

By default all tasks for the current event loop are returned. If *loop* is `None`, the *get_event_loop()* function is used to get the current loop.

Deprecated since version 3.7, will be removed in version 3.9: Do not call this as a task method. Use the *asyncio.all_tasks()* function instead.

classmethod current_task(*loop=None*)

Return the currently running task or `None`.

If *loop* is `None`, the *get_event_loop()* function is used to get the current loop.

Deprecated since version 3.7, will be removed in version 3.9: Do not call this as a task method. Use the *asyncio.current_task()* function instead.

Generator-based Coroutines

注釈: Support for generator-based coroutines is **deprecated** and is scheduled for removal in Python 3.10.

Generator-based coroutines predate async/await syntax. They are Python generators that use `yield from` expressions to await on Futures and other coroutines.

Generator-based coroutines should be decorated with *@asyncio.coroutine*, although this is not enforced.

@asyncio.coroutine

Decorator to mark generator-based coroutines.

This decorator enables legacy generator-based coroutines to be compatible with async/await code:

```
@asyncio.coroutine
def old_style_coroutine():
    yield from asyncio.sleep(1)

async def main():
    await old_style_coroutine()
```

This decorator should not be used for `async def` coroutines.

Deprecated since version 3.8, will be removed in version 3.10: Use `async def` instead.

asyncio.iscoroutine(*obj*)

obj が **コルーチンオブジェクト** であれば `True` を返します。

This method is different from *inspect.iscoroutine()* because it returns `True` for generator-based coroutines.

asyncio.iscoroutinefunction(*func*)

Return `True` if *func* is a *coroutine function*.

This method is different from *inspect.iscoroutinefunction()* because it returns `True` for generator-based coroutine functions decorated with *@coroutine*.

18.1.2 ストリーム

ソースコード: Lib/asyncio/streams.py

ストリームはネットワークコネクションと合わせて動作する高水準の async/await 可能な基本要素です。ストリームはコールバックや低水準のプロトコルやトランスポートを使うことなくデータを送受信することを可能にします。

以下は asyncio ストリームを使って書いた TCP エコークライアントの例です:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())
```

(次のページに続く)

(前のページからの続き)

```
    await writer.drain()

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_echo_client('Hello World!'))
```

下記の 使用例 節も参照してください。

ストリーム関数

以下の asyncio のトップレベル関数はストリームの作成や操作を行うことができます:

coroutine asyncio.open_connection(*host=None*, *port=None*, ***, *loop=None*, *limit=None*, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*)

ネットワークコネクションを確立し、(reader, writer) のオブジェクトのペアを返します。

戻り値の *reader* と *writer* はそれぞれ *StreamReader* と *StreamWriter* クラスのインスタンスです。

The *loop* argument is optional and can always be determined automatically when this function is awaited from a coroutine.

limit は戻り値の *StreamReader* インスタンスが利用するバッファのサイズの上限値を設定します。デフォルトでは *limit* は 64 KiB に設定されます。

残りの引数は直接 *loop.create_connection()* に渡されます。

バージョン 3.7 で追加: The *ssl_handshake_timeout* parameter.

coroutine asyncio.start_server(*client_connected_cb*, *host=None*, *port=None*, ***, *loop=None*, *limit=None*, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*, *ssl_handshake_timeout=None*, *start_serving=True*)

ソケットサーバーを起動します。

client_connected_cb コールバックは新しいクライアントコネクションが確立されるたびに呼び出されます。このコールバックは *StreamReader* と *StreamWriter* クラスのインスタンスのペア (reader, writer) を 2 つの引数として受け取ります。

client_connected_cb には単純な呼び出し可能オブジェクトか、または コルーチン関数 を指定します; コルーチン関数が指定された場合、コールバックの呼び出しは自動的に *Task* としてスケジュールされます。

The *loop* argument is optional and can always be determined automatically when this method is awaited from a coroutine.

limit は戻り値の *StreamReader* インスタンスが利用するバッファのサイズの上限値を設定します。デフォルトでは *limit* は 64 KiB に設定されます。

残りの引数は直接 *loop.create_server()* に渡されます。

バージョン 3.7 で追加: The *ssl_handshake_timeout* and *start_serving* parameters.

Unix ソケット

coroutine asyncio.open_unix_connection(*path=None*, ***, *loop=None*, *limit=None*, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*)

Unix ソケットコネクションを確立し、`(reader, writer)` のオブジェクトのペアを返します。

この関数は *open_connection()* と似ていますが Unix ソケットに対して動作します。

loop.create_unix_connection() のドキュメントも参照してください。

利用可能な環境: Unix。

バージョン 3.7 で追加: The *ssl_handshake_timeout* parameter.

バージョン 3.7 で変更: The *path* parameter can now be a *path-like object*

coroutine asyncio.start_unix_server(*client_connected_cb*, *path=None*, ***, *loop=None*, *limit=None*, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *start_serving=True*)

Unix のソケットサーバーを起動します。

start_server() と似ていますが Unix ソケットに対して動作します。

loop.create_unix_server() のドキュメントも参照してください。

利用可能な環境: Unix。

バージョン 3.7 で追加: The *ssl_handshake_timeout* and *start_serving* parameters.

バージョン 3.7 で変更: The *path* parameter can now be a *path-like object*.

StreamReader

class asyncio.StreamReader

IO ストリームからデータを読み出すための API を提供するリーダーオブジェクトを表します。

StreamReader オブジェクトを直接インスタンス化することは推奨されません; 代わりに *open_connection()* や *start_server()* を使ってください。

coroutine read(*n=-1*)

n バイト読み込みます。*n* が指定されないか `-1` が指定されていた場合 EOF になるまで読み込み、

全データを返します。

EOF を受信し、かつ内部バッファーが空の場合、空の bytes オブジェクトを返します。

coroutine readline()

1 行読み込みます。" 行" とは、\n で終了するバイト列のシーケンスです。

EOF を受信し、かつ \n が見つからない場合、このメソッドは部分的に読み込んだデータを返します。

EOF を受信し、かつ内部バッファーが空の場合、空の bytes オブジェクトを返します。

coroutine readexactly(*n*)

厳密に *n* バイトのデータを読み出します。

n バイトを読み出す前に EOF に達した場合 *IncompleteReadError* を送出します。部分的に読み出したデータを取得するには *IncompleteReadError.partial* 属性を使ってください。

coroutine readuntil(*separator=b'\n'*)

separator が見つかるまでストリームからデータを読み出します。

成功時には、データと区切り文字は内部バッファから削除されます (消費されます)。返されるデータの最後には区切り文字が含まれます。

読み出したデータの量が設定したストリームの上限を超えると *LimitOverrunError* 例外が送出されます。このときデータは内部バッファーに残され、再度読み出すことができます。

完全な区切り文字が見つかる前に EOF に達すると *IncompleteReadError* 例外が送出され、内部バッファーがリセットされます。このとき *IncompleteReadError.partial* 属性は区切り文字の一部を含むかもしれません。

バージョン 3.5.2 で追加.

at_eof()

バッファーが空で feed_eof() が呼ばれていた場合 True を返します。

StreamWriter

class asyncio.StreamWriter

IO ストリームにデータを書き込むための API を提供するライターオブジェクトを表します。

StreamWriter オブジェクトを直接インスタンス化することは推奨されません; 代わりに *open_connection()* や *start_server()* を使ってください。

write(*data*)

このメソッドは、背後にあるソケットにデータ *data* を即座に書き込みます。書き込みに失敗した場合、データは送信可能になるまで内部の書き込みバッファーに格納されて待機します。

このメソッドは drain() メソッドと組み合わせて使うべきです:

```
stream.write(data)
await stream.drain()
```

writelines(*data*)

このメソッドは、背後にあるソケットにバイトデータのリスト (またはイテラブル) を即座に書き込みます。書き込みに失敗した場合、データは送信可能になるまで内部の書き込みバッファーに格納されて待機します。

このメソッドは `drain()` メソッドと組み合わせて使うべきです:

```
stream.writelines(lines)
await stream.drain()
```

close()

このメソッドはストリームと背後にあるソケットをクローズします。

このメソッドは `wait_closed()` メソッドと組み合わせて使うべきです:

```
stream.close()
await stream.wait_closed()
```

can_write_eof()

背後にあるトランスポートが *write_eof()* メソッドをサポートしている場合 `True` を返し、そうでない場合は `False` を返します。

write_eof()

バッファーされた書き込みデータを全て書き込んでから、ストリームの書き込み側終端をクローズします。

transport

背後にある asyncio トランスポートを返します。

get_extra_info(*name*, *default=None*)

オプションのトランスポート情報にアクセスします。詳細は *BaseTransport.get_extra_info()* を参照してください。

coroutine drain()

ストリームへの書き込み再開に適切な状態になるまで待ちます。使用例:

```
writer.write(data)
await writer.drain()
```

このメソッドは背後にある IO 書き込みバッファーとやりとりを行うフロー制御メソッドです。バッファーのサイズが最高水位点に達した場合、*drain()* はバッファのサイズが最低水位点を下回るまで減量され、書き込み再開可能になるまで書き込みをブロックします。待ち受けの必要がない場合、*drain()* は即座にリターンします。

is_closing()

ストリームがクローズされたか、またはクローズ処理中の場合に `True` を返します。

バージョン 3.7 で追加.

coroutine wait_closed()

ストリームがクローズされるまで待機します。

このメソッドは、*close()* を呼び出した後に、コネクションがクローズされるまで待機するために呼び出すべきです。

バージョン 3.7 で追加.

使用例

ストリームを使った TCP Echo クライアント

asyncio.open_connection() 関数を使った TCP Echo クライアントです:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()

asyncio.run(tcp_echo_client('Hello World!'))
```

参考:

TCP エコークライアントプロトコル の例は低水準の *loop.create_connection()* メソッドを使っています。

ストリームを使った TCP Echo サーバー

asyncio.start_server() 関数を使った TCP Echo サーバーです:

```
import asyncio

async def handle_echo(reader, writer):
    data = await reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')

    print(f"Received {message!r} from {addr!r}")
```

(次のページに続く)

(前のページからの続き)

```
    print(f"Send: {message!r}")
    writer.write(data)
    await writer.drain()

    print("Close the connection")
    writer.close()

async def main():
    server = await asyncio.start_server(
        handle_echo, '127.0.0.1', 8888)

    addr = server.sockets[0].getsockname()
    print(f'Serving on {addr}')

    async with server:
        await server.serve_forever()

asyncio.run(main())
```

参考:

TCP **エコーサーバープロトコル** の例は *loop.create_server()* メソッドを使っています。

HTTP ヘッダーの取得

コマンドラインから渡された URL の HTTP ヘッダーを問い合わせる簡単な例です:

```
import asyncio
import urllib.parse
import sys

async def print_http_headers(url):
    url = urllib.parse.urlsplit(url)
    if url.scheme == 'https':
        reader, writer = await asyncio.open_connection(
            url.hostname, 443, ssl=True)
    else:
        reader, writer = await asyncio.open_connection(
            url.hostname, 80)

    query = (
        f"HEAD {url.path or '/'} HTTP/1.0\r\n"
        f"Host: {url.hostname}\r\n"
        f"\r\n"
    )

    writer.write(query.encode('latin-1'))
    while True:
        line = await reader.readline()
```

(次のページに続く)

(前のページからの続き)

```
        if not line:
            break

        line = line.decode('latin1').rstrip()
        if line:
            print(f'HTTP header> {line}')

    # Ignore the body, close the socket
    writer.close()

url = sys.argv[1]
asyncio.run(print_http_headers(url))
```

使い方:

```
python example.py http://example.com/path/page.html
```

または HTTPS を使用:

```
python example.py https://example.com/path/page.html
```

ストリームを使ってデータを待つオープンソケットの登録

open_connection() 関数を使ってソケットがデータを受信するまで待つコルーチンです:

```
import asyncio
import socket

async def wait_for_data():
    # Get a reference to the current event loop because
    # we want to access low-level APIs.
    loop = asyncio.get_running_loop()

    # Create a pair of connected sockets.
    rsock, wsock = socket.socketpair()

    # Register the open socket to wait for data.
    reader, writer = await asyncio.open_connection(sock=rsock)

    # Simulate the reception of data from the network
    loop.call_soon(wsock.send, 'abc'.encode())

    # Wait for data
    data = await reader.read(100)

    # Got data, we are done: close the socket
    print("Received:", data.decode())
    writer.close()
```

(次のページに続く)

(前のページからの続き)

```
    # Close the second socket
    wsock.close()

asyncio.run(wait_for_data())
```

参考:

プロトコルを使ってオープンしたソケットをデータ待ち受けのために登録する 例では、低水準のプロトコルと *loop.create_connection()* メソッドを使っています。

ファイル記述子の読み出しイベントを監視する 例では、低水準の *loop.add_reader()* メソッドを使ってファイル記述子を監視しています。

18.1.3 同期プリミティブ

ソースコード: Lib/asyncio/locks.py

---

asyncio の同期プリミティブは *threading* モジュールのそれと類似するようにデザインされていますが、2つの重要な注意事項があります:

- asyncio の同期プリミティブはスレッドセーフではありません。従って OS スレッドの同期に使うべきではありません (代わりに *threading* を使ってください);
- 同期プリミティブのメソッドは *timeout* 引数を受け付けません; タイムアウトを伴う操作を実行するには *asyncio.wait_for()* 関数を使ってください。

asyncio モジュールは以下の基本的な同期プリミティブを持っています:

- *Lock*
- *Event*
- *Condition*
- *Semaphore*
- *BoundedSemaphore*

Lock

class asyncio.**Lock**(**, loop=None*)

asyncio タスクのためのミューテックスロックを実装しています。スレッドセーフではありません。

asyncio ロックは、共有リソースに対する排他的なアクセスを保証するために使うことができます。

Lock の望ましい使用方法は、`async with` 文と組み合わせて使うことです:

```
lock = asyncio.Lock()

# ... later
async with lock:
    # access shared state
```

これは以下のコードと等価です:

```
lock = asyncio.Lock()

# ... later
await lock.acquire()
try:
    # access shared state
finally:
    lock.release()
```

Deprecated since version 3.8, will be removed in version 3.10: *loop* パラメータ。

coroutine acquire()

ロックを獲得します。

このメソッドはロックが **解除される** まで待機し、ロックを **ロック状態** に変更して `True` を返します。

複数のコルーチンが *acquire()* メソッドによりロックの解除を待ち受けている場合、最終的にただひとつのコルーチンが実行されます。

ロックの獲得は **公平** です: すなわちロックを獲得して実行されるコルーチンは、最初にロックの待ち受けを開始したコルーチンです。

release()

ロックを解放します。

ロックが **ロック状態** の場合、ロックを **解除状態** にしてリターンします。

ロックが **解除状態** の場合、*RuntimeError* 例外が送出されます。

locked()

ロック状態 の場合に `True` を返します。

Event

class asyncio.Event(*, *loop=None*)

イベントオブジェクトです。スレッドセーフではありません。

asyncio イベントは、複数の asyncio タスクに対して何らかのイベントが発生したことを通知するために使うことができます。

Event オブジェクトは内部フラグを管理します。フラグの値は *set()* メソッドにより *true* に、また *clear()* メソッドにより *false* に設定することができます。*wait()* メソッドはフラグが *true* になるまで処理をブロックします。フラグの初期値は *false* です。

Deprecated since version 3.8, will be removed in version 3.10: *loop* パラメータ。 以下はプログラム例です:

```
async def waiter(event):
    print('waiting for it ...')
    await event.wait()
    print('... got it!')

async def main():
    # Create an Event object.
    event = asyncio.Event()

    # Spawn a Task to wait until 'event' is set.
    waiter_task = asyncio.create_task(waiter(event))

    # Sleep for 1 second and set the event.
    await asyncio.sleep(1)
    event.set()

    # Wait until the waiter task is finished.
    await waiter_task

asyncio.run(main())
```

coroutine wait()

イベントがセットされるまで待機します。

イベントがセットされると、即座に True を返します。そうでなければ、他のタスクが *set()* メソッドを呼び出すまで処理をブロックします。

set()

イベントをセットします。

イベントがセットされるまで待機している全てのタスクは、即座に通知を受けて実行を再開します。

clear()

イベントをクリア (アンセット) します

wait() メソッドで待ち受けを行うタスクは *set()* メソッドが再度呼び出されるまで処理をブ

ロックします。

is_set()
イベントがセットされている場合 True を返します。

Condition

class asyncio.Condition(*lock=None*, ***, *loop=None*)
条件変数オブジェクトです。スレッドセーフではありません。

asyncio 条件プリミティブは何らかのイベントが発生するのを待ち受け、そのイベントを契機として共有リソースへの排他的なアクセスを得るために利用することができます。

本質的に、Condition オブジェクトは *Event* と a *Lock* の 2 つのクラスの機能を組み合わせたものです。複数の Condition オブジェクトが単一の Lock を共有することでができます。これにより、共有リソースのそれぞれの状態に関連する異なるタスクの間で、そのリソースへの排他的アクセスを調整することが可能になります。

オプション引数 *lock* は *Lock* または None でなければなりません。後者の場合自動的に新しい Lock オブジェクトが生成されます。

Deprecated since version 3.8, will be removed in version 3.10: *loop* パラメータ。

Condition の望ましい使用方法は async with 文と組み合わせて使うことです:

```
cond = asyncio.Condition()

# ... later
async with cond:
    await cond.wait()
```

これは以下のコードと等価です:

```
cond = asyncio.Condition()

# ... later
await cond.acquire()
try:
    await cond.wait()
finally:
    cond.release()
```

coroutine acquire()
下層でのロックを獲得します。

このメソッドは下層のロックが **解除される** まで待機し、ロックを **ロック状態** に変更して True を返します。

notify(*n=1*)
この条件を待ち受けている最大で *n* 個のタスク (*n* のデフォルト値は 1) を起動します。待ち受けているタスクがいない場合、このメソッドは何もしません。

このメソッドが呼び出される前にロックを獲得しておかなければなりません。また、メソッド呼び出し後速やかにロックを解除しなければなりません。**解除された** ロックとと共に呼び出された場合、*RuntimeError* 例外が送出されます。

locked()
下層のロックを獲得していれば True を返します。

notify_all()
この条件を待ち受けている全てのタスクを起動します。

このメソッドは *notify()* と同じように振る舞いますが、待ち受けている全てのタスクを起動します。

このメソッドが呼び出される前にロックを獲得しておかなければなりません。また、メソッド呼び出し後速やかにロックを解除しなければなりません。**解除された** ロックとと共に呼び出された場合、*RuntimeError* 例外が送出されます。

release()
下層のロックを解除します。

アンロック状態のロックに対して呼び出された場合、*RuntimeError* が送出されます。

coroutine wait()
通知を受けるまで待機します。

このメソッドが呼び出された時点で呼び出し元のタスクがロックを獲得していない場合、*RuntimeError* 例外が送出されます。

このメソッドは下層のロックを解除し、その後 *notify()* または *notify_all()* の呼び出しによって起動されるまで処理をブロックします。いったん起動されると、Condition は再びロックを獲得し、メソッドは True を返します。

coroutine wait_for(*predicate*)
引数 predicate の条件が **真** になるまで待機します。

引数 predicate は戻り値が真偽地として解釈可能な呼び出し可能オブジェクトでなければなりません。predicate の最終的な値が戻り値になります。

Semaphore

class asyncio.Semaphore(*value=1*, ***, *loop=None*)
セマフォオブジェクトです。スレッドセーフではありません。

セマフォは内部のカウンターを管理しています。カウンターは *acquire()* メソッドの呼び出しによって減算され、*release()* メソッドの呼び出しによって加算されます。カウンターがゼロを下回ることはありません。*acquire()* メソッドが呼び出された時にカウンターがゼロになっていると、セマフォは処理をブロックし、他のタスクが *release()* メソッドを呼び出すまで待機します。

オプション引数 *value* は内部カウンターの初期値を与えます (デフォルトは 1 です)。指定された値が 0 より小さい場合、*ValueError* 例外が送出されます。

Deprecated since version 3.8, will be removed in version 3.10: *loop* パラメータ。

Semaphore 望ましい使用方法は、`async with` 文と組み合わせて使うことです:

```
sem = asyncio.Semaphore(10)

# ... later
async with sem:
    # work with shared resource
```

これは以下のコードと等価です:

```
sem = asyncio.Semaphore(10)

# ... later
await sem.acquire()
try:
    # work with shared resource
finally:
    sem.release()
```

coroutine acquire()
セマフォを獲得します。

内部カウンターがゼロより大きい場合、カウンターを 1 つ減算して即座に `True` を返します。内部カウンターがゼロの場合、*release()* が呼び出されるまで待機してから `True` を返します。

locked()
セマフォを直ちに獲得できない場合 `True` を返します。

release()
セマフォを解放し、内部カウンターを 1 つ加算します。セマフォ待ちをしているタスクを起動する可能性があります。

BoundedSemaphore と異なり、*Semaphore* は `release()` を `acquire()` よりも多く呼び出すことを許容します。

BoundedSemaphore

class asyncio.BoundedSemaphore(*value=1*, ***, *loop=None*)
有限セマフォオブジェクトです。スレッドセーフではありません。

有限セマフォは *Semaphore* の一種で、*release()* メソッドの呼び出しにより内部カウンターが **初期値** よりも増加してしまう場合は *ValueError* 例外を送出します。

Deprecated since version 3.8, will be removed in version 3.10: *loop* パラメータ。

バージョン 3.7 で非推奨: Acquiring a lock using `await lock` or `yield from lock` and/or `with` statement (`with await lock`, `with (yield from lock)`) is deprecated. Use `async with lock` instead.

18.1.4 サブプロセス

ソースコード: Lib/asyncio/subprocess.py, Lib/asyncio/base_subprocess.py

---

このセクションはサブプロセスの生成と管理を行う高水準の async/await asyncio API について説明します。

以下は、asyncio モジュールがどのようにシェルコマンドを実行し、結果を取得できるかを示す例です:

```
import asyncio

async def run(cmd):
    proc = await asyncio.create_subprocess_shell(
        cmd,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE)

    stdout, stderr = await proc.communicate()

    print(f'[{cmd!r} exited with {proc.returncode}]')
    if stdout:
        print(f'[stdout]\n{stdout.decode()}')
    if stderr:
        print(f'[stderr]\n{stderr.decode()}')

asyncio.run(run('ls /zzz'))
```

このサンプルコードは以下を出力します:

```
['ls /zzz' exited with 1]
[stderr]
ls: /zzz: No such file or directory
```

全ての asyncio のサブプロセス関数は非同期ですが、asyncio モジュールはこれらの非同期関数と協調するための多くのツールを提供しているので、複数のサブプロセスを並列に実行して監視することは簡単です。実際、上記のサンプル小 0 どを複数のコマンドを同時に実行するように修正するのはきわめて単純です:

```
async def main():
    await asyncio.gather(
        run('ls /zzz'),
        run('sleep 1; echo "hello"'))

asyncio.run(main())
```

使用例 節も参照してください。

サブプロセスの生成

coroutine asyncio.**create_subprocess_exec**(*program*, **args*, *stdin=None*, *stdout=None*, *stderr=None*, *loop=None*, *limit=None*, ***kwds*)

サブプロセスを作成します。

引数 *limit* は (*subprocess.PIPE* を *stdout* と *stderr* に設定した場合の) Process.stdout と Process.stderr のための *StreamReader* ラッパーのバッファー上限値を設定します。

Process のインスタンスを返します。

他のパラメータについては *loop.subprocess_exec()* を参照してください。

Deprecated since version 3.8, will be removed in version 3.10: *loop* パラメータ。

coroutine asyncio.**create_subprocess_shell**(*cmd*, *stdin=None*, *stdout=None*, *stderr=None*, *loop=None*, *limit=None*, ***kwds*)

シェルコマンド *cmd* を実行します。

引数 *limit* は (*subprocess.PIPE* を *stdout* と *stderr* に設定した場合の) Process.stdout と Process.stderr のための *StreamReader* ラッパーのバッファー上限値を設定します。

Process のインスタンスを返します。

他のパラメータについては *loop.subprocess_shell()* のドキュメントを参照してください。

重要: シェルインジェクション の脆弱性を回避するために全ての空白文字および特殊文字を適切にクオートすることは、アプリケーション側の責任で確実に行ってください。シェルコマンドを構成する文字列内の空白文字と特殊文字のエスケープは、*shlex.quote()* 関数を使うと適切に行うことができます。

Deprecated since version 3.8, will be removed in version 3.10: *loop* パラメータ。

注釈: サブプロセスは、*ProactorEventLoop* を使えば Windows でも利用可能です。詳しくは *Windows* **におけるサブプロセスのサポート** を参照してください。

参考:

asyncio は以下に挙げるサブプロセスと協調するための **低水準の** API も持っています: *loop.subprocess_exec()*, *loop.subprocess_shell()*, *loop.connect_read_pipe()*, *loop.connect_write_pipe()* および *Subprocess Transports* と *Subprocess Protocols*。

定数

asyncio.subprocess.PIPE
stdin, *stdout* または *stderr* に渡すことができます。

PIPE が *stdin* 引数に渡された場合、*Process.stdin* 属性は *StreamWriter* インスタンスを指します。

PIPE が *stdout* や *stderr* 引数に渡された場合、*Process.stdout* と *Process.stderr* 属性は *StreamReader* インスタンスを指します。

asyncio.subprocess.STDOUT
stderr 引数に対して利用できる特殊な値で、標準エラー出力が標準出力にリダイレクトされることを意味します。

asyncio.subprocess.DEVNULL
プロセスを生成する関数の *stdin*, *stdout* または *stderr* 引数に利用できる特殊な値です。対応するサブプロセスのストリームに特殊なファイル *os.devnull* が使われることを意味します。

サブプロセスとやりとりする

create_subprocess_exec() と *create_subprocess_shell()* の 2 つの関数はどちらも *Process* クラスのインスタンスを返します。*Process* クラスはサブプロセスと通信したり、サブプロセスの完了を監視したりするための高水準のラッパーです。

class asyncio.subprocess.Process
関数 *create_subprocess_exec()* や *create_subprocess_shell()* によって生成された OS のプロセスをラップするオブジェクトです。

このクラスは *subprocess.Popen* クラスと同様の API を持つように設計されていますが、いくつかの注意すべき違いがあります:

- Popen と異なり、Process インスタンスは *poll()* メソッドに相当するメソッドを持っていません;
- the *communicate()* and *wait()* methods don't have a *timeout* parameter: use the *wait_for()* function;
- *subprocess.Popen.wait()* メソッドが同期処理のビジーループとして実装されているのに対して、*Process.wait()* メソッドは非同期処理です;
- *universal_newlines* パラメータはサポートされていません。

このクラスは **スレッド安全ではありません**。

サブプロセスとスレッド 節も参照してください。

coroutine wait()
子プロセスが終了するのを待ち受けます。

returncode 属性を設定し、その値を返します。

注釈: stdout=PIPE または stderr=PIPE を使っており、OS パイプバッファがさらなるデータを受け付けるようになるまで子プロセスをブロックするほど大量の出力を生成場合、このメソッドはデッドロックする可能性があります。この条件を避けるため、パイプを使用する場合は *communicate()* メソッドを使ってください。

coroutine communicate(*input=None*)

プロセスとのやりとりを行います:

1. *stdin* にデータを送信します (*input* が None でない場合);
2. EOF に達するまで *stdout* および *stderr* からデータを読み出します;
3. プロセスが終了するまで待ち受けます。

input オプション引数は子プロセスに送信されるデータ (*bytes* オブジェクト) です。

(stdout_data, stderr_data) のタプルを返します。

input を標準入力 *stdin* に書き込んでいる時に *BrokenPipeError* または *ConnectionResetError* 例外が送出された場合、例外は無視されます。このような条件は、全てのデータが *stdin* に書き込まれる前にプロセスが終了した場合に起こります。

子プロセスの標準入力 *stdin* にデータを送りたい場合、プロセスは stdin=PIPE を設定して生成する必要があります。同様に、None 以外の何らかのデータを戻り値のタプルで受け取りたい場合、プロセスは stdout=PIPE と stderr=PIPE のいずれかまたは両方を指定して生成しなければなりません。

プロセスから受信したデータはメモリ上にバッファーされることに注意してください。そのため、返されるデータのサイズが大きいかまたは無制限の場合はこのメソッドを使わないようにしてください。

send_signal(*signal*)

子プロセスにシグナル *signal* を送信します。

注釈: Windows では、SIGTERM は *terminate()* の別名になります。CTRL_C_EVENT および CTRL_BREAK_EVENT で、*creationflags* で始まり、CREATE_NEW_PROCESS_GROUP を含むパラメータをプロセスに送信することができます。

terminate()

子プロセスを停止します。

POSIX システムでは、このメソッドは子プロセスに *signal.SIGTERM* シグナルを送信します

Windows では、子プロセスを停止するために Win32 API 関数 TerminateProcess() を呼び出します。

kill()
子プロセスを強制終了 (kill) します。

POSIX システムの場合、このメソッドは子プロセスに SIGKILL シグナルを送信します。

Windows では、このメソッドは *terminate()* のエイリアスです。

stdin
標準入力ストリーム (*StreamWriter*) です。プロセスが stdin=None で生成された場合は None になります。

stdout
標準出力ストリーム (*StreamReader*) です。プロセスが stdout=None で生成された場合は None になります。

stderr
標準エラー出力ストリーム (*StreamReader*) です。プロセスが stderr=None で生成された場合は None になります。

警告: Use the *communicate()* method rather than *process.stdin.write()*, *await process.stdout.read()* or *await process.stderr.read*. This avoids deadlocks due to streams pausing reading or writing and blocking the child process.

pid
子プロセスのプロセス番号 (PID) です。

create_subprocess_shell() 関数によって生成されたプロセスの場合、この属性は生成されたシェルの PID になることに注意してください。

returncode
プロセスが終了した時の終了ステータスを返します。

この属性が None であることは、プロセスがまだ終了していないことを示しています。

負の値 -N は子プロセスがシグナル N により中止させられたことを示します (POSIX のみ)。

サブプロセスとスレッド

標準的な asyncio のイベントループは、異なるスレッドからサブプロセスを実行するのをデフォルトでサポートしています。

Windows のサブプロセスは *ProactorEventLoop* (デフォルト) のみ提供され、*SelectorEventLoop* はサブプロセスをサポートしていません。

UNIX の *child watchers* はサブプロセスの終了を待ち受けるために使われます。より詳しい情報については **プロセスのウオッチャー** を参照してください。

バージョン 3.8 で変更: UNIX では、異なるスレッドから何らの制限なくサブプロセスを生成するために *ThreadedChildWatcher* を使うようになりました。

現在の子プロセスのウオッチャーが **アクティブでない** 場合にサブプロセスを生成すると *RuntimeError* 例外が送出されます。

標準で提供されない別のイベントループ実装の場合、固有の制限がある可能性があります; それぞれの実装のドキュメントを参照してください。

参考:

asyncio-multithreading

使用例

サブプロセスの制御のために *Process* クラスを使い、サブプロセスの標準出力を読み出すために *StreamReader* を使う例です。

サブプロセスは *create_subprocess_exec()* 関数により生成されます:

```
import asyncio
import sys

async def get_date():
    code = 'import datetime; print(datetime.datetime.now())'

    # Create the subprocess; redirect the standard output
    # into a pipe.
    proc = await asyncio.create_subprocess_exec(
        sys.executable, '-c', code,
        stdout=asyncio.subprocess.PIPE)

    # Read one line of output.
    data = await proc.stdout.readline()
    line = data.decode('ascii').rstrip()

    # Wait for the subprocess exit.
    await proc.wait()
    return line

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

低水準の API を使って書かれた **同様の例** も参照してください。

18.1.5 キュー

ソースコード: Lib/asyncio/queues.py

---

asyncio キューは *queue* モジュールのクラス群と同じ形になるように設計されています。asyncio キューはスレッドセーフではありませんが、それらは async/await コードから使われるために特別に設計されています。

asyncio キューのメソッドは *timeout* パラメータを持たないことに注意してください; タイムアウトを伴うキューを使った処理を行うには *asyncio.wait_for()* 関数を使ってください。

下記の **使用例** 節も参照してください。

Queue

class asyncio.Queue(*maxsize=0*, ***, *loop=None*)

先入れ先出し (FIFO) キューです。

maxsize がゼロ以下の場合、キューは無限長になります。0 より大きい整数の場合、キューが *maxsize* に達すると `await put()` は *get()* によってキューの要素が除去されるまでブロックします。

標準ライブラリにおけるスレッドベースの *queue* モジュールと異なり、キューのサイズは常に既知であり、*qsize()* メソッドを呼び出すことによって取得することができます。

Deprecated since version 3.8, will be removed in version 3.10: *loop* パラメータ。

このクラスは **スレッド安全ではありません**。

maxsize

キューに追加できるアイテム数です。

empty()

キューが空ならば `True` を、そうでなければ `False` を返します。

full()

キューに要素が *maxsize* 個あれば `True` を返します。

キューが `maxsize=0` (デフォルト値) で初期化された場合、*full()* メソッドが `True` を返すことはありません。

coroutine get()

キューから要素を削除して返します。キューが空の場合項目が利用可能になるまで待機します。

get_nowait()

直ちに利用できるアイテムがあるときはそれを、そうでなければ *QueueEmpty* を返します。

coroutine join()

キューにある全ての要素が取得され、処理されるまでブロックします。

未完了のタスクのカウント値は、キューにアイテムが追加されるときは常に加算され、キューの要素を消費するコルーチンが要素を取り出し、処理を完了したことを通知するために *task_done()*

を呼び出すと減算されます。未完了のタスクのカウント値がゼロになると、*join()* のブロックは解除されます。

coroutine put(*item*)
要素をキューに入力します。キューが満杯の場合、要素を追加する前に空きスロットが利用できるようになるまで待機します。

put_nowait(*item*)
ブロックせずにアイテムをキューに追加します。

直ちに利用できるスロットがない場合、*QueueFull* を送出します。

qsize()
キュー内の要素数を返します。

task_done()
キューに入っていたタスクが完了したことを示します。

キューコンシューマーによって使用されます。タスクの取得に *get()* を使用し、その後の *task_done()* の呼び出しでタスクの処理が完了したことをキューに通知します。

join() が現在ブロック中だった場合、全アイテムが処理されたとき (*put()* でキューに追加された全アイテムの *task_done()* の呼び出しを受信したとき) に再開します。

キューに追加されているアイテム数以上の呼び出しが行われたときに *ValueError* を送出します。

優先度付きのキュー

class asyncio.PriorityQueue
Queue の変種です; 優先順位にしたがって要素を取り出します (最低順位が最初に取り出されます)。

項目は典型的には `(priority_number, data)` 形式のタプルです。

LIFO キュー

class asyncio.LifoQueue
Queue の変種で、最後に追加された項目を最初に取り出します (後入れ先出し、またはスタック)。

例外

exception asyncio.QueueEmpty
この例外は *get_nowait()* メソッドが空のキューに対して呼ばれたときに送出されます。

exception asyncio.QueueFull
サイズが *maxsize* に達したキューに対して *put_nowait()* メソッドが 呼ばれたときに送出される例外です。

使用例

キューを使って、並行処理を行う複数のタスクにワークロードを分散させることができます:

```
import asyncio
import random
import time


async def worker(name, queue):
    while True:
        # Get a "work item" out of the queue.
        sleep_for = await queue.get()

        # Sleep for the "sleep_for" seconds.
        await asyncio.sleep(sleep_for)

        # Notify the queue that the "work item" has been processed.
        queue.task_done()

        print(f'{name} has slept for {sleep_for:.2f} seconds')


async def main():
    # Create a queue that we will use to store our "workload".
    queue = asyncio.Queue()

    # Generate random timings and put them into the queue.
    total_sleep_time = 0
    for _ in range(20):
        sleep_for = random.uniform(0.05, 1.0)
        total_sleep_time += sleep_for
        queue.put_nowait(sleep_for)

    # Create three worker tasks to process the queue concurrently.
    tasks = []
    for i in range(3):
        task = asyncio.create_task(worker(f'worker-{i}', queue))
        tasks.append(task)

    # Wait until the queue is fully processed.
    started_at = time.monotonic()
    await queue.join()
    total_slept_for = time.monotonic() - started_at

    # Cancel our worker tasks.
    for task in tasks:
        task.cancel()
    # Wait until all worker tasks are cancelled.
    await asyncio.gather(*tasks, return_exceptions=True)

    print('====')
    print(f'3 workers slept in parallel for {total_slept_for:.2f} seconds')
```

(次のページに続く)

(前のページからの続き)

```
    print(f'total expected sleep time: {total_sleep_time:.2f} seconds')


asyncio.run(main())
```

18.1.6 例外

ソースコード: Lib/asyncio/exceptions.py

---

exception asyncio.TimeoutError

The operation has exceeded the given deadline.

重要: This exception is different from the builtin *TimeoutError* exception.

---

exception asyncio.CancelledError

処理がキャンセルされました。

asyncio タスクがキャンセルされた場合の処理をカスタマイズするために、この例外を一旦キャッチすることができます。ほとんどの場合、キャッチした例外は再度送出しなければなりません。

バージョン 3.8 で変更: *CancelledError* は *BaseException* の派生クラスになりました。

exception asyncio.InvalidStateError

Task または *Future* の内部状態が不正になりました。

すでに結果の値が設定されている *Future* オブジェクトに対してさらに結果の値を再び設定しようとする場合などに送出されることがあります。

exception asyncio.SendfileNotAvailableError

与えられたソケットまたはファイルタイプに対して "sendfile" システムコールが利用可能ではありません。

RuntimeError の派生クラスです。

exception asyncio.IncompleteReadError

要求された読み込み処理が完了できませんでした。

asyncio ストリーム *API* から送出されます。

この例外は *EOFError* の派生クラスです。

expected

期待される総バイト数 (*int*) です。

partial

ストリームの終端に達するまでに読み込んだ *bytes* 文字列です。

exception asyncio.LimitOverrunError
区切り文字を探している間にバッファサイズの上限に到達しました。

asyncio ストリーム *API* から送出されます。

consumed
未消費のバイトの合計数です。

18.1.7 イベントループ

ソースコード: Lib/asyncio/profile.py と Lib/asyncio/pstats.py

まえがき

イベントループは全ての asyncio アプリケーションの中核をなす存在です。イベントループは非同期タスクやコールバックを実行し、ネットワーク I/O を処理し、サブプロセスを実行します。

アプリケーション開発者は通常 *asyncio.run()* のような高水準の ayncio 関数だけを利用し、ループオブジェクトを参照したり、ループオブジェクトのメソッドを呼び出したりすることはほとんどありません。この節は、イベントループの振る舞いに対して細かい調整が必要な、低水準のコード、ライブラリ、フレームワークの開発者向けです。

イベントループの取得

以下の低水準関数はイベントループの取得、設定、生成するために使います:

asyncio.get_running_loop()
現在の OS スレッドで実行中のイベントループを取得します。

実行中のイベントループがない場合は *RuntimeError* 例外を送出します。この関数はコルーチンまたはコールバックからのみ呼び出し可能です。

バージョン 3.7 で追加.

asyncio.get_event_loop()
現在のイベントループを取得します。

OS スレッドに現在のイベントループが未設定で、OS スレッドがメインスレッドであり、かつ *set_event_loop()* がまだ呼び出されていない場合、asyncio は新しいイベントループを生成し、それを現在のイベントループに設定します。

この関数の振る舞いは (特にイベントループポリシーをカスタマイズした場合) 複雑なため、コルーチンやコールバックでは *get_event_loop()* よりも *get_running_loop()* を使うほうが好ましいと考えられます。

また、低水準の関数を使って手作業でイベントループの管理をするかわりに、*asyncio.run()* を使うことを検討してください。

asyncio.**set_event_loop**(*loop*)

loop を OS スレッドの現在のイベントループに設定します。

asyncio.**new_event_loop**()

Create a new event loop object.

get_event_loop(), *set_event_loop()*, および *new_event_loop()* 関数の振る舞いは、**カスタムイベントループポリシーを設定する** ことにより変更することができます。

内容

このページは以下の節から構成されます:

- **イベントループのメソッド** 節は、イベントループ API のリファレンスです。
- **コールバックハンドル** 節は *loop.call_soon()* や *loop.call_later()* などのスケジューリングメソッドが返す *Handle* や *TimerHandle* インスタンスについて解説しています。
- **サーバーオブジェクト** 節は *loop.create_server()* のようなメソッドが返す型について解説しています。
- **イベントループの実装** 節は *SelectorEventLoop* と *ProactorEventLoop* の 2 つのクラスについて解説しています。
- **使用例** 節ではイベントループ API の具体的な使い方を紹介しています。

イベントループのメソッド

イベントループは以下の **低水準な** API を持っています:

- **ループの開始と停止**
- **コールバックのスケジューリング**
- **遅延コールバックのスケジューリング**
- **フューチャーとタスクの生成**
- **ネットワーク接続の確立**
- **ネットワークサーバの生成**
- **ファイルの転送**
- *TLS* **へのアップグレード**
- **ファイル記述子の監視**
- **ソケットオブジェクトと直接やりとりする**

- *DNS*
- パイプとやりとりする
- *Unix* シグナル
- スレッドまたはプロセスプールでコードを実行する
- エラーハンドリング *API*
- デバッグモードの有効化
- サブプロセスの実行

ループの開始と停止

loop.**run_until_complete**(*future*)

フューチャー (*Future* インスタンス) が完了するまで実行します。

引数が **コルーチンオブジェクト** の場合、暗黙のうちに *asyncio.Task* として実行されるようにスケジュールされます。

Future の結果を返すか、例外を送出します。

loop.**run_forever**()

stop() が呼び出されるまでイベントループを実行します。

run_forever() メソッドが呼ばれるより前に *stop()* メソッドが呼ばれた場合、イベントループはタイムアウトをゼロにして一度だけ I/O セレクタの問い合わせ処理を行い、I/O イベントに対してスケジュールされた全てのコールバック (および既にスケジュール済みのコールバック) を実行したのち、終了します。

run_forever() メソッドを実行中に *stop()* メソッドが呼び出された場合、イベントループは現在処理されているすべてのコールバックを実行してから終了します。この場合、コールバックにより新たにスケジュールされるコールバックは実行されないことに注意してください; これら新たにスケジュールされたコールバックは、次に *run_forever()* または *run_until_complete()* が呼び出されたときに実行されます。

loop.**stop**()

イベントループを停止します。

loop.**is_running**()

イベントループが現在実行中の場合 `True` を返します。

loop.**is_closed**()

イベントループが閉じられていた場合 `True` を返します。

loop.**close**()

イベントループをクローズします。

この関数が呼び出される時点で、イベントループが実行中であってはいけません。保留中のコールバックはすべて破棄されます。

このメソッドは全てのキューをクリアし、エグゼキューターが実行完了するのを待たずにシャットダウンします。

このメソッドはべき等 (何回実行しても結果は同じ) であり取り消せません。イベントループがクローズされた後、他のいかなるメソッドも呼び出すべきではありません。

coroutine loop.**shutdown_asyncgens**()

現在オープンになっているすべての *asynchronous generator* (非同期ジェネレータ) オブジェクトをスケジュールし、aclose() メソッドを呼び出すことでそれらをクローズします。このメソッドの呼び出し後に新しい非同期ジェネレータがイテレートされると、イベントループは警告を発します。このメソッドはスケジュールされたすべての非同期ジェネレータの終了処理を確実に行うために使用すべきです。

asyncio.run() を使った場合はこの関数を呼び出す必要はありません。

以下はプログラム例です:

```
try:
    loop.run_forever()
finally:
    loop.run_until_complete(loop.shutdown_asyncgens())
    loop.close()
```

バージョン 3.6 で追加.

コールバックのスケジューリング

loop.**call_soon**(*callback*, **args*, *context=None*)

イベントループの次のイテレーションで *callback* に指定したコールバック (*callback*) を *args* 引数で呼び出すようにスケジュールします。

コールバックは登録された順に呼び出されます。各コールバックは厳密に 1 回だけ呼び出されます。

オプションのキーワード引数 *context* を使って、コールバック*callback* を実行する際のコンテキスト *contextvars.Context* を設定することができます。コンテキスト *context* が指定されない場合は現在のコンテキストが使われます。

asyncio.Handle のインスタンスを返します。このインスタンスを使ってスケジュールしたコールバックをキャンセルすることができます。

このメソッドはスレッドセーフではありません。

loop.**call_soon_threadsafe**(*callback*, **args*, *context=None*)

call_soon() のスレッドセーフ版です。必ず **別のスレッドから** コールバックをスケジュールする際に使ってください。

このドキュメントの *asyncio-multithreading* 節を参照してください。

バージョン 3.7 で変更: キーワード引数 *context* が追加されました。詳細は **PEP 567** を参照してください。

注釈: ほとんどの *asyncio* モジュールのスケジューリング関数は、キーワード引数をコールバックに渡すことを許していません。キーワード引数を渡すためには *functools.partial()* を使ってください:

```
# will schedule "print("Hello", flush=True)"
loop.call_soon(
    functools.partial(print, "Hello", flush=True))
```

asyncio は partial オブジェクトのデバッグメッセージやエラーメッセージをよりよく可視化することができるため、通常はラムダ式よりも partial オブジェクトを使う方が便利です。

遅延コールバックのスケジューリング

イベントループは、コールバック関数を未来のある時点で呼び出されるようにスケジュールする仕組みを提供します。イベントループは時刻が戻らない単調な時計 (monotonic clock) を使って時刻を追跡します。

`loop.call_later`(*delay*, *callback*, **args*, *context=None*)

delay 秒経過後にコールバック関数 *callback* を呼び出すようにスケジュールします。*delay* には整数または浮動小数点数を指定します。

asyncio.TimerHandle のインスタンスを返します。このインスタンスを使ってスケジュールしたコールバックをキャンセルすることができます。

callback は厳密に一度だけ呼び出されます。2 つのコールバックが完全に同じ時間にスケジュールされた場合、呼び出しの順序は未定義です。

オプションの位置引数 *args* はコールバックが呼び出されるときに位置引数として渡されます。キーワード引数を指定してコールバックを呼び出したい場合は *functools.partial()* を使用してください。

オプションのキーワード引数 *context* を使って、コールバック*callback* を実行する際のコンテキスト *contextvars.Context* を設定することができます。コンテキスト *context* が指定されない場合は現在のコンテキストが使われます。

バージョン 3.7 で変更: キーワード引数 *context* が追加されました。詳細は **PEP 567** を参照してください。

バージョン 3.8 で変更: Python 3.7 またはそれ以前のバージョンでは、デフォルトイベントループの実装を利用した場合に遅延時間 *delay* が 1 日を超えることができませんでした。この問題は Python 3.8 で修正されました。

`loop.call_at`(*when*, *callback*, **args*, *context=None*)

絶対値の時刻 *when* (整数または浮動小数点数) にコールバックを呼び出すようにスケジュールします。*loop.time()* と同じ参照時刻を使用します。

このメソッドの振る舞いは *call_later()* と同じです。

asyncio.TimerHandle のインスタンスを返します。このインスタンスを使ってスケジュールしたコールバックをキャンセルすることができます。

バージョン 3.7 で変更: キーワード引数 *context* が追加されました。詳細は **PEP 567** を参照してください。

バージョン 3.8 で変更: Python 3.7 またはそれ以前のバージョンでは、デフォルトイベントループの実装を利用した場合に現在の時刻と *when* との差が 1 日を超えることができませんでした。この問題は Python 3.8 で修正されました。

loop.**time**()
現在の時刻を *float* 値で返します。時刻はイベントループが内部で参照している時刻が戻らない単調な時計 (monotonic clock) に従います。

注釈: バージョン 3.8 で変更: Python 3.7 またはそれ以前のバージョンでは、タイムアウト (相対値 *delay* もしくは絶対値 *when*) は 1 日を超えることができませんでした。この問題は Python 3.8 で修正されました。

参考:

関数 *asyncio.sleep()*。

フューチャーとタスクの生成

loop.**create_future**()
イベントループに接続した *asyncio.Future* オブジェクトを生成します。

asyncio でフューチャーオブジェクトを作成するために推奨される方法です。このメソッドにより、サードパーティ製のイベントループが Futures クラスの (パフォーマンスや計測方法が優れた) 代替実装を提供することを可能にします。

バージョン 3.5.2 で追加.

loop.**create_task**(*coro*, ***, *name=None*)
コルーチン の実行をスケジュールします。*Task* オブジェクトを返します。

サードパーティのイベントループは相互運用のための自身の *Task* のサブクラスを使用できます。この場合、結果は *Task* のサブクラスになります。

name 引数が指定され、値が None でない場合、*Task.set_name()* メソッドにより *name* がタスクの名前として設定されます。

バージョン 3.8 で変更: name パラメータが追加されました。

loop.**set_task_factory**(*factory*)
loop.create_task() が使用するタスクファクトリーを設定します。

factory が None の場合、デフォルトのタスクファクトリーが設定されます。そうでなければ、*factory* は (loop, coro) に一致する関数シグネチャを持った **呼び出し可能オブジェクト** でなければなりま

せん。ここで *loop* はアクティブなイベントループへの参照であり、*coro* はコルーチンオブジェクトです。呼び出し可能オブジェクトは *asyncio.Future* と互換性のあるオブジェクトを返さなければなりません。

loop.**get_task_factory**()
タスクファクトリを返します。デフォルトのタスクファクトリを使用中の場合は None を返します。

ネットワーク接続の確立

coroutine loop.**create_connection**(*protocol_factory*, *host=None*, *port=None*, ***, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *happy_eyeballs_delay=None*, *interleave=None*)
host と *port* で指定されたアドレスとのストリーミングトランスポート接続をオープンします。

ソケットファミリーは *host* (または *family* 引数が与えられた場合は *family*) に依存し、*AF_INET* か *AF_INET6* のいずれかを指定します。

ソケットタイプは *SOCK_STREAM* になります。

protocol_factory は *asyncio* **プロトコル** の実装を返す呼び出し可能オブジェクトでなければなりません。

このメソッドはバックグラウンドで接続の確立を試みます。成功した場合、メソッドは (transport, protocol) のペアを返します。

時系列での下層処理の概要は以下のとおりです:

1. 接続を確立し、その接続に対する **トランスポート** が生成されます。
2. *protocol_factory* が引数なしで呼び出され、ファクトリが **プロトコル** インスタンスを返すよう要求します。
3. プロトコルインスタンスが *connection_made()* メソッドを呼び出すことにより、トランスポートと紐付けられます。
4. 成功すると (transport, protocol) タプルが返されます。

作成されたトランスポートは実装依存の双方向ストリームです。

その他の引数:

- *ssl*: 偽値以外が与えられた場合、SSL/TLS トランスポートが作成されます (デフォルトでは暗号化なしの TCP トランスポートが作成されます)。*ssl* が *ssl.SSLContext* オブジェクトの場合、このコンテキストがトランスポートを作成するために使用されます; *ssl* が *True* の場合、*ssl.create_default_context()* が返すデフォルトのコンテキストが使われます。

  参考:

  SSL/TLS **セキュリティについての考察**

- *server_hostname* は対象サーバーの証明書との一致を確認するためのホスト名を設定または上書きします。この引数は *ssl* が `None` でない場合のみ設定すべきです。デフォルトでは *host* に指定したサーバー名が使用されます。*host* が空の文字列の場合のデフォルト値は設定されていません。その場合、*server_hostname* を必ず指定してください。*server_hostname* も空の文字列の場合は、ホスト名の一致確認は行われません (これは深刻なセキュリティリスクであり、中間者攻撃を受ける可能性があります)。

- *family*, *proto*, *flags* は任意のアドレスファミリであり、*host* 解決のための getaddrinfo() 経由で渡されるプロトコルおよびフラグになります。このオプションが与えられた場合、これらはすべて *socket* モジュール定数に従った整数でなければなりません。

- *happy_eyeballs_delay* が設定されると、この接続に対して Happy Eyeballs が有効化されます。設定する値は浮動小数点数であり、次の接続試行を開始する前に、現在の接続試行が完了するのを待つ時間を秒単位で表現します。この値は **RFC 8305** で定義されている " 接続試行遅延" に相当します。RFC で推奨されている実用的なデフォルト値は `0.25` (250 ミリ秒) です。

- *interleave* はホスト名が複数の IP アドレスに名前解決される場合のアドレスの並べ替えを制御します。`0` または未指定の場合並べ替えは行われず、*getaddrinfo()* が返す順番にしたがってアドレスへの接続を試行します。正の整数が指定されると、アドレスはアドレスファミリに応じてインターリーブされます。このとき、与えられた整数は **RFC 8305** で定義される " 最初のアドレスファミリカウント (First Address Family Count)" として解釈されます。デフォルト値は、*happy_eyeballs_delay* が指定されない場合は `0` であり、指定された場合は `1` です。

- *sock* を与える場合、トランスポートに使用される、既存の、かつ接続済の *socket.socket* オブジェクトを指定します。*sock* を指定する場合、*host*、*port*、*family*、*proto*、*flags*、*happy_eyeballs_delay*、*interleave* および *local_addr* のいずれも指定してはいけません。

- *local_addr*, if given, is a `(local_host, local_port)` tuple used to bind the socket to locally. The *local_host* and *local_port* are looked up using `getaddrinfo()`, similarly to *host* and *port*.

- *ssl_handshake_timeout* は TLS ハンドシェイクが完了するまでの (TLS 接続のための) 待ち時間を秒単位で指定します。指定した待ち時間を超えると接続は中断します。`None` が与えられた場合はデフォルト値 `60.0` が使われます。

バージョン 3.8 で追加: *happy_eyeballs_delay* と *interleave* が追加されました。

Happy Eyeballs Algorithm: Success with Dual-Stack Hosts. When a server's IPv4 path and protocol are working, but the server's IPv6 path and protocol are not working, a dual-stack client application experiences significant connection delay compared to an IPv4-only client. This is undesirable because it causes the dual- stack client to have a worse user experience. This document specifies requirements for algorithms that reduce this user-visible delay and provides an algorithm.

詳しくは右記を参照してください: https://tools.ietf.org/html/rfc6555

バージョン 3.7 で追加: The *ssl_handshake_timeout* parameter.

バージョン 3.6 で変更: 全ての TCP 接続に対してデフォルトでソケットオプション `TCP_NODELAY` が設定されるようになりました。

バージョン 3.5 で変更: *ProactorEventLoop* において SSL/TLS のサポートが追加されました。

参考:

open_connection() 関数は高水準の代替 API です。この関数は (*StreamReader*, *StreamWriter*) のペアを返し、async/await コードから直接使うことができます。

coroutine loop.create_datagram_endpoint(*protocol_factory*, *local_addr=None*, *remote_addr=None*, ***, *family=0*, *proto=0*, *flags=0*, *reuse_address=None*, *reuse_port=None*, *allow_broadcast=None*, *sock=None*)

注釈: SO_REUSEADDR の利用が UDP に対して重大なセキュリティ上の懸念をもたらすため、*reuse_address* パラメータはサポートされなくなりました。明示的に reuse_address=True を設定すると例外を送出します。

SO_REUSEADDR を使って、同一の UDP ソケットアドレスに対して複数のプロセスが異なる UID でソケットを割り当てている場合、受信パケットは複数のソケット間にランダムに分散する可能性があります。

サポートされているプラットフォームでは、*reuse_port* が同様の機能に対する代用品として利用できます。*reuse_port* は代替機能として SO_REUSEPORT を使っており、複数のプロセスが異なる UID で同一のソケットに対して割り当てられるのを明確に禁止します。

データグラム接続 (UDP) を生成します。

ソケットファミリーは *host* (または *family* 引数が与えられた場合は *family*) に依存し、*AF_INET*、*AF_INET6*、*AF_UNIX* のいずれかを指定します。

ソケットタイプは *SOCK_DGRAM* になります。

protocol_factory は *asyncio* **プロトコル** の実装を返す呼び出し可能オブジェクトでなければなりません。

成功すると (transport, protocol) タプルが返されます。

その他の引数:

- *local_addr* が指定される場合、(local_host, local_port) のタプルで、ソケットをローカルで束縛するために使用されます。*local_host* と *local_port* は *getaddrinfo()* を使用して検索されます。
- *remote_addr* が指定される場合、(remote_host, remote_por) のタプルで、ソケットをリモートアドレスに束縛するために使用されます。*remote_host* と *remote_port* は *getaddrinfo()* を使用して検索されます。
- *family*, *proto*, *flags* は任意のアドレスファミリです。これらのファミリ、プロトコル、フラグは、*host* 解決のため *getaddrinfo()* 経由でオプションで渡されます。これらのオプションを指定する場合、すべて *socket* モジュール定数に従った整数でなければなりません。

- *reuse_port* は、同じポートにバインドされた既存の端点すべてがこのフラグを設定して生成されている場合に限り、この端点を既存の端点と同じポートにバインドすることを許可するよう、カーネルに指示します（訳註: ソケットのオプション SO_REUSEPORT を使用します)。このオプションは、Windows やいくつかの UNIX システムではサポートされていません。`SO_REUSEPORT` 定数が定義されていなければ、この機能はサポートされません。
- *allow_broadcast* は、カーネルに、このエンドポイントがブロードキャストアドレスにメッセージを送信することを許可するように指示します。
- オプションの *sock* を指定することで、既存の、すでに接続されている *socket.socket* をトランスポートで使用することができます。このオプションを使用する場合、*local_addr* と *remote_addr* は省略してください (*None* でなければなりません)。

UDP echo **クライアントプロトコル** および *UDP echo* **サーバープロトコル** の例を参照してください。

バージョン 3.4.4 で変更: *family*, *proto*, *flags*, *reuse_address*, *reuse_port*, **allow_broadcast*, *sock* パラメータが追加されました。

バージョン 3.8.1 で変更: セキュリティ上の懸念により、*reuse_address* パラメータはサポートされなくなりました。

バージョン 3.8 で変更: Windows サポートが追加されました。

coroutine loop.**create_unix_connection**(*protocol_factory*, *path=None*, ***, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*)

Unix 接続を生成します。

ソケットファミリーは *AF_UNIX* になります; また、ソケットタイプは *SOCK_STREAM* になります。

成功すると `(transport, protocol)` タプルが返されます。

path は Unix ドメインソケット名で、*sock* パラメータが指定されない場合は必須です。抽象 Unix ソケット、*str*、*bytes*、and *Path* 形式でのパスがサポートされています。

このメソッドの引数についての詳細は *loop.create_connection()* メソッドのドキュメントを参照してください。

利用可能な環境: Unix。

バージョン 3.7 で追加: The *ssl_handshake_timeout* parameter.

バージョン 3.7 で変更: The *path* parameter can now be a *path-like object*.

ネットワークサーバの生成

coroutine loop.**create_server**(*protocol_factory*, *host=None*, *port=None*, ***, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*, *ssl_handshake_timeout=None*, *start_serving=True*)

アドレス *host* のポート *port* をリッスンする (ソケットタイプが *SOCK_STREAM* である) TCP サーバーを生成します。

Server オブジェクトを返します。

引数:

- *protocol_factory* は *asyncio* **プロトコル** の実装を返す呼び出し可能オブジェクトでなければなりません。
- *host* パラメータはいくつかの方法で指定することができ、その値によってサーバーがどこをリッスンするかが決まります。
  - *host* が文字列の場合、TCP サーバーは *host* で指定した単一のネットワークインターフェースに束縛されます。
  - *host* が文字列のシーケンスである場合、TCP サーバーはそのシーケンスで指定された全てのネットワークインターフェースに束縛されます。
  - *host* が空の文字列か None の場合、すべてのインターフェースが想定され、複合的なソケットのリスト (通常は一つが IPv4、もう一つが IPv6) が返されます。
- *family* に *socket.AF_INET* または *AF_INET6* を指定することにより、ソケットでそれぞれ IPv4 または IPv6 の使用を強制することができます。設定されない場合、*family* はホスト名から決定されます (socket.AF_UNSPEC がデフォルトになります)。
- *flags* は *getaddrinfo()* のためのビットマスクになります。
- サーバーで既存のソケットオブジェクトを使用するために、オプションの引数 *sock* にソケットオブジェクトを設定することができます。指定した場合、*host* と *port* を指定してはいけません。
- *backlog* は *listen()* に渡される、キューに入るコネクションの最大数になります (デフォルトは 100)。
- 確立した接続の上で TLS を有効化するために、*ssl* に *SSLContext* のインスタンスを指定することができます。
- *reuse_address* は、TIME_WAIT 状態にあるローカルソケットを、その状態が自然にタイムアウトするのを待つことなく再利用するようカーネルに指示します (訳註: ソケットのオプション SO_REUSEADDR を使用します)。指定しない場合、UNIX では自動的に True が設定されます。
- *reuse_port* は、同じポートにバインドされた既存の端点すべてがこのフラグを設定して生成されている場合に限り、この端点を既存の端点と同じポートにバインドすることを許可するよう、カー

ネルに指示します（訳註: ソケットのオプション SO_REUSEPORT を使用します)。このオプションは、Windows ではサポートされていません。

- *ssl_handshake_timeout* は TLS ハンドシェイクが完了するまでの (TLS サーバーのための) 待ち時間を秒単位で指定します。指定した待ち時間を超えると接続は中断します。`None` が与えられた場合はデフォルト値 `60.0` が使われます。
- *start_serving* が `True` に設定された場合 (これがデフォルトです)、生成されたサーバーは即座に接続の受け付けを開始します。`False` が指定された場合、ユーザーは接続の受け付けを開始するために *Server.start_serving()* または *Server.serve_forever()* を待ち受け (await) る必要があります。

バージョン 3.7 で追加: Added *ssl_handshake_timeout* and *start_serving* parameters.

バージョン 3.6 で変更: 全ての TCP 接続に対してデフォルトでソケットオプション `TCP_NODELAY` が設定されるようになりました。

バージョン 3.5 で変更: *ProactorEventLoop* において SSL/TLS のサポートが追加されました。

バージョン 3.5.1 で変更: *host* パラメータに文字列のシーケンスを指定できるようになりました。

参考:

start_server() 関数は高水準の代替 API です。この関数は *StreamReader* と *StreamWriter* のペアを返し、async/await コードから使うことができます。

coroutine loop.**create_unix_server**(*protocol_factory*, *path=None*, ***, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *start_serving=True*)

loop.create_server() と似ていますが、*AF_UNIX* ソケットファミリーとともに動作します。

path は Unix ドメインソケット名で、*sock* パラメータが指定されない場合は必須です。抽象 Unix ソケット、*str*、*bytes*、and *Path* 形式でのパスがサポートされています。

このメソッドの引数についての詳細は *loop.create_server()* メソッドのドキュメントを参照してください。

利用可能な環境: Unix。

バージョン 3.7 で追加: The *ssl_handshake_timeout* and *start_serving* parameters.

バージョン 3.7 で変更: The *path* parameter can now be a *Path* object.

coroutine loop.**connect_accepted_socket**(*protocol_factory*, *sock*, ***, *ssl=None*, *ssl_handshake_timeout=None*)

すでに確立した接続を transport と protocol のペアでラップします。

このメソッドは asyncio の範囲外で確立された接続を使うサーバーに対しても使えますが、その場合でも接続は asyncio を使って処理されます。

引数:

- *protocol_factory* は *asyncio* **プロトコル** の実装を返す呼び出し可能オブジェクトでなければなりません。
- *sock* は *socket.accept* メソッドが返す既存のソケットオブジェクトです。
- *ssl* には *SSLContext* を指定できます。指定すると、受け付けたコネクション上での SSL を有効にします。
- *ssl_handshake_timeout* は SSL ハンドシェイクが完了するまでの (SSL 接続のための) 待ち時間を秒単位で指定します。None が与えられた場合はデフォルト値 60.0 が使われます。

(transport, protocol) のペアを返します。

バージョン 3.7 で追加: The *ssl_handshake_timeout* parameter.

バージョン 3.5.3 で追加.

ファイルの転送

coroutine loop.**sendfile**(*transport*, *file*, *offset=0*, *count=None*, ***, *fallback=True*)

transport を通じて *file* を送信します。送信したデータの総バイト数を返します。

このメソッドは、もし利用可能であれば高性能な *os.sendfile()* を利用します。

file はバイナリモードでオープンされた通常のファイルオブジェクトでなければなりません。

offset はファイルの読み込み開始位置を指定します。*count* が指定された場合、ファイルの EOF までファイルを送信する代わりに、*count* で指定された総バイト数の分だけ送信します。ファイルオブジェクトが指し示す位置は、メソッドがエラーを送出した場合でも更新されます。この場合実際に送信されたバイト数は *file.tell()* メソッドで取得することができます。

fallback を True に指定することで、asyncio がプラットフォームが sendfile システムコールをサポートしていない場合 (たとえば Windows や Unix の SSL ソケットなど) に別の方法でファイルの読み込みと送信を行うようにすることができます。

システムが *sendfile* システムコールをサポートしておらず、かつ *fallback* が False の場合、*SendfileNotAvailableError* 例外を送出します。

バージョン 3.7 で追加.

TLS へのアップグレード

coroutine loop.**start_tls**(*transport*, *protocol*, *sslcontext*, ***, *server_side=False*, *server_hostname=None*, *ssl_handshake_timeout=None*)

既存のトランスポートベースの接続を TLS にアップグレードします。

新しいトランスポートのインスタンスを返します。*protocol* は必ず *await* 直後に利用を開始しなければなりません。*start_tls* メソッドに渡した *transport* は、このメソッドの呼び出し以後決して使ってはいけません。

引数:

- *transport* と *protocol* には、*create_server()* や *create_connection()* が返すものと同等のインスタンスを指定します。
- *sslcontext*: 構成済みの *SSLContext* インスタンスです。
- (*create_server()* で生成されたような) サーバーサイドの接続をアップグレードする場合は *server_side* に `True` を渡します。
- *server_hostname*: 対象のサーバーの証明書との照合に使われるホスト名を設定または上書きします。
- *ssl_handshake_timeout* は TLS ハンドシェイクが完了するまでの (TLS 接続のための) 待ち時間を秒単位で指定します。指定した待ち時間を超えると接続は中断します。`None` が与えられた場合はデフォルト値 `60.0` が使われます。

バージョン 3.7 で追加.

ファイル記述子の監視

loop.add_reader(*fd*, *callback*, **args*)

ファイル記述子 *fd* に対する読み込みが可能かどうかの監視を開始し、*fd* が読み込み可能になると、指定した引数でコールバック *callback* を呼び出します。

loop.remove_reader(*fd*)

ファイル記述子 *fd* に対する読み込みが可能かどうかの監視を停止します。

loop.add_writer(*fd*, *callback*, **args*)

ファイル記述子 *fd* に対する書き込みが可能かどうかの監視を開始し、*fd* が書き込み可能になると、指定した引数でコールバック *callback* を呼び出します。

コールバック *callback* に **キーワード引数を渡す** 場合は *functools.partial()* を使ってください。

loop.remove_writer(*fd*)

ファイル記述子 *fd* に対する書き込みが可能かどうかの監視を停止します。

これらのメソッドに対する制限事項については **プラットフォームのサポート状況** 節も参照してください。

ソケットオブジェクトと直接やりとりする

一般に、*loop.create_connection()* や *loop.create_server()* のようなトランスポートベースの API を使ったプロトコルの実装はソケットと直接やり取りする実装に比べて高速です。しかしながら、パフォーマンスが重要でなく、直接 *socket* オブジェクトとやりとりした方が便利なユースケースがいくつかあります。

coroutine loop.sock_recv(*sock*, *nbytes*)

nbytes で指定したバイト数までのデータをソケット *sock* から受信します。このメソッドは *socket.recv()* の非同期版です。

受信したデータをバイトオブジェクトとして返します。

sock はノンブロッキングソケットでなければなりません。

バージョン 3.7 で変更: このメソッドは常にコルーチンメソッドとしてドキュメントに記載されてきましたが、Python 3.7 以前のリリースでは *Future* オブジェクトを返していました。Python 3.7 からは async def メソッドになりました。

coroutine loop.**sock_recv_into**(*sock*, *buf*)

ソケット *sock* からデータを受信してバッファ *buf* に格納します。ブロッキングコードの *socket.recv_into()* メソッドをモデルとしています。

バッファに書き込んだデータのバイト数を返します。

sock はノンブロッキングソケットでなければなりません。

バージョン 3.7 で追加.

coroutine loop.**sock_sendall**(*sock*, *data*)

データ *data* をソケット *sock* に送信します。*socket.sendall()* メソッドの非同期版です。

このメソッドは *data* をすべて送信し終えるか、またはエラーが起きるまでデータをソケットに送信し続けます。送信に成功した場合 None を返します。エラーの場合は例外が送出されます。エラーとなった場合、接続の受信側で正しく処理されたデータの総量を特定する方法はありません。

sock はノンブロッキングソケットでなければなりません。

バージョン 3.7 で変更: このメソッドは常にコルーチンメソッドとしてドキュメントに記載されてきましたが、Python 3.7 以前のリリースでは *Future* オブジェクトを返していました。Python 3.7 からは async def メソッドになりました。

coroutine loop.**sock_connect**(*sock*, *address*)

ソケット *sock* をアドレス *address* のリモートソケットに接続します。

socket.connect() の非同期版です。

sock はノンブロッキングソケットでなければなりません。

バージョン 3.5.2 で変更: address を名前解決する必要はなくなりました。sock_connect は *socket.inet_pton()* を呼び出して *address* が解決済みかどうかを確認します。未解決の場合、*address* の名前解決には *loop.getaddrinfo()* メソッドが使われます。

参考:

loop.create_connection() および *asyncio.open_connection()*。

coroutine loop.**sock_accept**(*sock*)

接続を受け付けます。ブロッキングコールの *socket.accept()* メソッドをモデルとしています。

ソケットはアドレスに束縛済みで、接続を listen 中である必要があります。戻り値は (conn, address) のペアで、*conn* は接続を通じてデータの送受信を行うための **新しい** ソケットオブジェクト、*address* は接続先の端点でソケットに束縛されているアドレスを示します。

sock はノンブロッキングソケットでなければなりません。

バージョン 3.7 で変更: このメソッドは常にコルーチンメソッドとしてドキュメントに記載されてきま

したが、Python 3.7 以前のリリースでは *Future* オブジェクトを返していました。Python 3.7 からは async def メソッドになりました。

参考:

loop.create_server() および *start_server()*。

coroutine loop.**sock_sendfile**(*sock*, *file*, *offset=0*, *count=None*, ***, *fallback=True*)

ファイルを送信します。利用可能なら高性能な *os.sendfile* を使います。送信したデータの総バイト数を返します。

socket.sendfile() メソッドの非同期版です。

sock は *socket.SOCK_STREAM* タイプのノンブロッキングな *socket* でなければなりません。

file はバイナリモードでオープンされた通常のファイルオブジェクトでなければなりません。

offset はファイルの読み込み開始位置を指定します。*count* が指定された場合、ファイルの EOF までファイルを送信する代わりに、*count* で指定された総バイト数の分だけ送信します。ファイルオブジェクトが指し示す位置は、メソッドがエラーを送出した場合でも更新されます。この場合実際に送信されたバイト数は *file.tell()* メソッドで取得することができます。

fallback が True に設定された場合、プラットフォームが sendfile システムコールをサポートしていない場合 (たとえば Windows や Unix の SSL ソケットなど) に asyncio が別の方法でファイルの読み込みと送信を行うようにすることができます。

システムが *sendfile* システムコールをサポートしておらず、かつ *fallback* が False の場合、*SendfileNotAvailableError* 例外を送出します。

sock はノンブロッキングソケットでなければなりません。

バージョン 3.7 で追加.

DNS

coroutine loop.**getaddrinfo**(*host*, *port*, ***, *family=0*, *type=0*, *proto=0*, *flags=0*)

socket.getaddrinfo() の非同期版です。

coroutine loop.**getnameinfo**(*sockaddr*, *flags=0*)

socket.getnameinfo() の非同期版です。

バージョン 3.7 で変更: *getaddrinfo* と *getnameinfo* の 2 つのメソッドは、いずれも常にコルーチンメソッドとしてドキュメントに記載されてきましたが、Python 3.7 以前のリリースでは、実際には *asyncio.Future* オブジェクトを返していました。Python 3.7 からはどちらのメソッドもコルーチンになりました。

パイプとやりとりする

coroutine loop.**connect_read_pipe**(*protocol_factory*, *pipe*)

イベントループの読み込み側終端に *pipe* を登録します。

protocol_factory は *asyncio* **プロトコル** の実装を返す呼び出し可能オブジェクトでなければなりません。

pipe には *file-like* **オブジェクト** を指定します。

(transport, protocol) のペアを返します。ここで *transport* は *ReadTransport* のインターフェースをサポートし、*protocol* は *protocol_factory* ファクトリでインスタンス化されたオブジェクトです。

SelectorEventLoop イベントループの場合、*pipe* は非ブロックモードに設定されていなければなりません。

coroutine loop.**connect_write_pipe**(*protocol_factory*, *pipe*)

pipe の書き込み側終端をイベントループに登録します。

protocol_factory は *asyncio* **プロトコル** の実装を返す呼び出し可能オブジェクトでなければなりません。

pipe は *file-like* **オブジェクト** です。

(transport, protocol) のペアを返します。ここで *transport* は *WriteTransport* のインスタンスであり、*protocol* は *protocol_factory* ファクトリでインスタンス化されたオブジェクトです。

SelectorEventLoop イベントループの場合、*pipe* は非ブロックモードに設定されていなければなりません。

注釈: *SelectorEventLoop* は Windows 上で上記のメソッドをサポートしていません。Windows では代わりに *ProactorEventLoop* を使ってください。

参考:

loop.subprocess_exec() および *loop.subprocess_shell()* メソッド。

Unix シグナル

loop.**add_signal_handler**(*signum*, *callback*, **args*)

コールバック *callback* をシグナル *signum* に対するハンドラに設定します。

コールバックは *loop*、登録された他のコールバック、およびイベントループの実行可能なコルーチンから呼び出されます。*signal.signal()* を使って登録されたシグナルハンドラと異なり、この関数で登録されたコールバックはイベントループと相互作用することが可能です。

シグナルナンバーが誤っているか捕捉不可能な場合 *ValueError* が送出されます。ハンドラーの設定に問題があった場合 *RuntimeError* が送出されます。

コールバック *callback* に **キーワード引数を渡す** 場合は *functools.partial()* を使ってください。

signal.signal() と同じく、この関数はメインスレッドから呼び出されなければなりません。

loop.**remove_signal_handler**(*sig*)

シグナル *sig* に対するハンドラを削除します。

シグナルハンドラが削除された場合 True を返します。シグナルに対してハンドラが設定されていない場合には False を返します。

利用可能な環境: Unix。

参考:

signal モジュール。

スレッドまたはプロセスプールでコードを実行する

awaitable loop.**run_in_executor**(*executor*, *func*, **args*)

指定したエグゼキュータで関数 *func* が実行されるように準備します。

引数 *executor* は *concurrent.futures.Executor* のインスタンスでなければなりません。*executor* が None の場合はデフォルトのエグゼキュータが使われます。

以下はプログラム例です:

```
import asyncio
import concurrent.futures

def blocking_io():
    # File operations (such as logging) can block the
    # event loop: run them in a thread pool.
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)

def cpu_bound():
    # CPU-bound operations will block the event loop:
    # in general it is preferable to run them in a
    # process pool.
    return sum(i * i for i in range(10 ** 7))

async def main():
    loop = asyncio.get_running_loop()

    ## Options:

    # 1. Run in the default loop's executor:
    result = await loop.run_in_executor(
        None, blocking_io)
    print('default thread pool', result)

    # 2. Run in a custom thread pool:
```

(次のページに続く)

(前のページからの続き)

```
    with concurrent.futures.ThreadPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, blocking_io)
        print('custom thread pool', result)

    # 3. Run in a custom process pool:
    with concurrent.futures.ProcessPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, cpu_bound)
        print('custom process pool', result)

asyncio.run(main())
```

このメソッドは *asyncio.Future* オブジェクトを返します。

関数 *func* に **キーワード引数を渡す** 場合は *functools.partial()* を使ってください。

バージョン 3.5.3 で変更: *loop.run_in_executor()* は内部で生成するスレッドプールエグゼキュータの `max_workers` を設定せず、代わりにスレッドプールエグゼキュータ (*ThreadPoolExecutor*) にデフォルト値を設定させるようになりました。

`loop.set_default_executor(`*executor*`)`

executor を *run_in_executor()* が使うデフォルトのエグゼキュータに設定します。*executor* は *ThreadPoolExecutor* のインスタンスでなければなりません。

バージョン 3.8 で非推奨: *ThreadPoolExecutor* のインスタンスでないエグゼキュータの使用は非推奨となり、Python 3.9 ではエラーになります。

executor は *concurrent.futures.ThreadPoolExecutor* のインスタンスでなければなりません。

エラーハンドリング API

イベントループ内での例外の扱い方をカスタマイズできます。

`loop.set_exception_handler(`*handler*`)`

handler を新しいイベントループ例外ハンドラーとして設定します。

handler が `None` の場合、デフォルトの例外ハンドラが設定されます。そうでなければ、*handler* は `(loop, context)` に一致する関数シグネチャを持った呼び出し可能オブジェクトでなければなりません。ここで `loop` はアクティブなイベントループへの参照であり、`context` は例外の詳細な記述からなる `dict` オブジェクトです (*context* についての詳細は *call_exception_handler()* メソッドのドキュメントを参照してください)。

`loop.get_exception_handler()`

現在の例外ハンドラを返します。カスタム例外ハンドラが設定されていない場合は `None` を返します。

バージョン 3.5.2 で追加.

`loop.default_exception_handler(`*context*`)`

デフォルトの例外ハンドラーです。

デフォルト例外ハンドラは、例外ハンドラが未設定の場合、例外が発生した時に呼び出されます。デフォルト例外ハンドラの挙動を受け入れるために、カスタム例外ハンドラから呼び出すことも可能です。

引数 *context* の意味は *call_exception_handler()* と同じです。

loop.**call_exception_handler**(*context*)
現在のイベントループ例外ハンドラーを呼び出します。

context は以下のキーを含む `dict` オブジェクトです (将来の Python バージョンで新しいキーが追加される可能性があります):

- 'message': エラーメッセージ;
- 'exception' (任意): 例外オブジェクト;
- 'future' (任意): *asyncio.Future* インスタンス;
- 'handle' (任意): *asyncio.Handle* インスタンス;
- 'protocol' (任意): **プロトコル** インスタンス;
- 'transport' (任意): **トランスポート** インスタンス;
- 'socket' (任意): *socket.socket* インスタンス;

注釈: このメソッドはイベントループの派生クラスでオーバーロードされてはいけません。カスタム例外ハンドラの設定には *set_exception_handler()* メソッドを使ってください。

デバッグモードの有効化

loop.**get_debug**()
イベントループのデバッグモード (*bool*) を取得します。

環境変数 `PYTHONASYNCIODEBUG` に空でない文字列が設定されている場合のデフォルト値は `True`、そうでない場合は `False` になります。

loop.**set_debug**(*enabled: bool*)
イベントループのデバッグモードを設定します。

バージョン 3.7 で変更: The new `-X dev` command line option can now also be used to enable the debug mode.

参考:

asyncio **のデバッグモード**。

サブプロセスの実行

この節で解説しているのは低水準のメソッドです。通常の async/await コードでは、高水準の関数である *asyncio.create_subprocess_shell()* や *asyncio.create_subprocess_exec()* を代わりに使うことを検討してください。

注釈: The default asyncio event loop on **Windows** does not support subprocesses. See *Subprocess Support on Windows* for details.

---

coroutine loop.**subprocess_exec**(*protocol_factory*, **args*, *stdin=subprocess.PIPE*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, ***kwargs*)

args で指定されたひとつの、または複数の文字列引数からサブプロセスを生成します。

args は下記のいずれかに当てはまる文字列のリストでなければなりません:

- *str*;
- または **ファイルシステムのエンコーディング** にエンコードされた *bytes*

引数の最初の文字列はプログラムの実行ファイルを指定します。それに続く残りの文字列は引数を指定し、そのプログラムに対する `argv` を構成します。

このメソッドは標準ライブラリの *subprocess.Popen* クラスを、`shell=False` かつ最初の引数に文字列のリストを渡して呼び出した場合に似ています。しかし、*Popen* クラスは文字列のリストを引数としてひとつだけ取るのに対して、*subprocess_exec* は複数の文字列引数をとることができます。

protocol_factory は *asyncio.SubprocessProtocol* クラスの派生クラスを返す呼び出し可能オブジェクトでなければなりません。

その他の引数:

- *stdin* 下記のいずれかをとることができます:
  - *connect_write_pipe()* メソッドを使ってサブプロセスの標準入力ストリームに接続されたパイプを表す file-like オブジェクト
  - デフォルト値は *subprocess.PIPE* 定数で、この場合新規にパイプを生成して接続します。
  - `None` が設定された場合、サブプロセスは元のプロセスのファイルデスクリプタを引き継ぎます。
  - *subprocess.DEVNULL* 定数を設定すると、特別なファイル *os.devnull* を使います。
- *stdout* は下記のいずれかをとることができます:
  - *connect_write_pipe()* メソッドを使ってサブプロセスの標準出力ストリームに接続されたパイプを表す file-like オブジェクト
  - デフォルト値は *subprocess.PIPE* 定数で、この場合新規にパイプを生成して接続します。

  - None が設定された場合、サブプロセスは元のプロセスのファイルデスクリプタを引き継ぎます。
  - *subprocess.DEVNULL* 定数を設定すると、特別なファイル *os.devnull* を使います。

- *stderr* は下記のいずれかをとることができます:
  - *connect_write_pipe()* メソッドを使ってサブプロセスの標準エラー出力ストリームに接続されたパイプを表す file-like オブジェクト
  - デフォルト値は *subprocess.PIPE* 定数で、この場合新規にパイプを生成して接続します。
  - None が設定された場合、サブプロセスは元のプロセスのファイルデスクリプタを引き継ぎます。
  - *subprocess.DEVNULL* 定数を設定すると、特別なファイル *os.devnull* を使います。
  - *subprocess.STDOUT* 定数を設定すると、標準エラー出力ストリームをプロセスの標準出力ストリームに接続します。
- その他のすべてのキーワード引数は解釈されずにそのまま *subprocess.Popen* に渡されます。ただし、*bufsize*、*universal_newlines*、*shell*、*text*、*encoding* および *errors* は指定してはいけません。

  asyncio のサブプロセス API はストリームからテキストへのデコードをサポ 0 としていません。ストリームからテキストに変換するには *bytes.decode()* 関数を使ってください。

他の引数についての詳細は *subprocess.Popen* クラスのコンストラクタを参照してください。

(transport, protocol) のペアを返します。ここで *transport* は *asyncio.SubprocessTransport* 基底クラスに適合するオブジェクトで、*protocol* は *protocol_factory* によりインスタンス化されたオブジェクトです。

coroutine loop.**subprocess_shell**(*protocol_factory*, *cmd*, ***, *stdin=subprocess.PIPE*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, ***kwargs*)

コマンド *cmd* からプラットフォームの ” シェル” シンタックスを使ってサブプロセスを生成します。*cmd* は *str* 文字列もしくは **ファイルシステムのエンコーディング** でエンコードされた *bytes* 文字列です。

これは標準ライブラリの *subprocess.Popen* クラスを shell=True で呼び出した場合と似ています。

protocol_factory は *SubprocessProtocol* の派生クラスを返す呼び出し可能オブジェクトでなければなりません。

その他の引数についての詳細は *subprocess_exec()* メソッドを参照してください。

“(transport, protocol)“のペアを返します。ここで *transport* は *SubprocessTransport* 基底クラスに適合するオブジェクトで、*protocol* は *protocol_factory* によりインスタンス化されたオブジェクトです。

注釈: シェルインジェクション の脆弱性を回避するために全ての空白文字および特殊文字を適切にクオート

することは、アプリケーション側の責任で確実に行ってください。シェルコマンドを構成する文字列内の空白文字と特殊文字のエスケープは、*shlex.quote()* 関数を使うと適切に行うことができます。

コールバックのハンドル

class asyncio.**Handle**

loop.call_soon() や *loop.call_soon_threadsafe()* が返すコールバックのラッパーです。

cancel()

コールバックをキャンセルします。コールバックがキャンセル済みまたは実行済みの場合、このメソッドは何の影響もありません。

cancelled()

コールバックがキャンセルされた場合 True を返します。

バージョン 3.7 で追加.

class asyncio.**TimerHandle**

A callback wrapper object returned by *loop.call_later()*, and *loop.call_at()*.

このクラスは *Handle* の派生クラスです。

when()

コールバックのスケジュール時刻を秒単位の *float* で返します。

戻り値の時刻は絶対値で、*loop.time()* と同じ参照時刻を使って定義されています。

バージョン 3.7 で追加.

Server オブジェクト

Server オブジェクトは *loop.create_server()*、*loop.create_unix_server()*、*start_server()* および *start_unix_server()* 関数により生成されます。

クラスを直接インスタンス化しないでください。

class asyncio.**Server**

Server オブジェクトは非同期のコンテキストマネージャです。async with 文の中で使われた場合、async with 文が完了した時に Server オブジェクトがクローズされること、およびそれ以降に接続を受け付けないことが保証されます。

```
srv = await loop.create_server(...)

async with srv:
    # some code

# At this point, srv is closed and no longer accepts new connections.
```

バージョン 3.7 で変更: Python 3.7 から、Server オブジェクトは非同期のコンテキストマネージャになりました。

close()

サーバーを停止します: 待機しているソケットをクローズし *sockets* 属性に None を設定します。

既存の受信中のクライアントとの接続を表すソケットはオープンのままです。

サーバーは非同期に停止されます。サーバーの停止を待ちたい場合は *wait_closed()* コルーチンを使用します。

get_loop()

サーバオブジェクトに付随するイベントループを返します。

バージョン 3.7 で追加.

coroutine start_serving()

接続の受け付けを開始します。

このメソッドはべき等です。すなわちサーバがすでにサービスを開始した後でも呼び出すことができます。

キーワード専用のパラメータ *start_serving* を *loop.create_server()* や *asyncio.start_server()* メソッドに対して使用することにより、初期に接続を受け付けない Server オブジェクトを生成することができます。この場合 Server.start_serving() または *Server.serve_forever()* メソッドを使ってオブジェクトが接続の受け付けを開始するようにすることができます。

バージョン 3.7 で追加.

coroutine serve_forever()

接続の受け入れを開始し、コルーチンがキャンセルされるまで継続します。serve_forever タスクのキャンセルによりサーバーもクローズされます。

このメソッドはサーバーがすでに接続の受け入れを開始していても呼び出し可能です。ひとつの *Server* オブジェクトにつき serve_forever タスクはひとつだけ存在できます。

以下はプログラム例です:

```
async def client_connected(reader, writer):
    # Communicate with the client with
    # reader/writer streams.  For example:
    await reader.readline()

async def main(host, port):
    srv = await asyncio.start_server(
        client_connected, host, port)
    await srv.serve_forever()

asyncio.run(main('127.0.0.1', 0))
```

バージョン 3.7 で追加.

is_serving()
サーバーが新規に接続の受け入れを開始した場合 `True` を返します。

バージョン 3.7 で追加.

coroutine wait_closed()
close() メソッドが完了するまで待ちます。

sockets
サーバーがリッスンしている *socket.socket* オブジェクトのリストです。

バージョン 3.7 で変更: Python 3.7 より前のバージョンでは、`Server.sockets` は内部に持っているサーバーソケットのリストを直接返していました。Python 3.7 ではリストのコピーが返されるようになりました。

イベントループの実装

asyncio は 2 つの異なるイベントループの実装、class:*SelectorEventLoop* と *ProactorEventLoop*、を提供します:

デフォルトでは、asyncio は Unix では *SelectorEventLoop* 、Windows では *ProactorEventLoop* 、をそれぞれ使うように構成されています。

class asyncio.SelectorEventLoop
selectors に基づくイベントループです。

プラットフォーム上で利用可能な最も効率の良い *selector* を使います。特定のセレクタ実装を使うように手動で構成することも可能です:

```
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

Availability: Unix, Windows.

class asyncio.ProactorEventLoop
”I/O 完了ポート” (IOCP) を使った Windows 向けのイベントループです。

利用可能な環境: Windows 。

参考:

I/O 完了ポートに関する MSDN のドキュメント.

class asyncio.AbstractEventLoop
asyncio に適合するイベントループの抽象基底クラスです。

イベントループのメソッド 節は、AbstractEventLoop の代替実装が定義すべき全てのメソッドを列挙しています。

使用例

この節の全ての使用例は **意図的に** *loop.run_forever()* や *loop.call_soon()* のような 低水準のイベントループ API の使用法を示しています。一方で現代的な asyncio アプリケーションはここに示すような方法をほとんど必要としません。*asyncio.run()* のような高水準の関数の使用を検討してください。

call_soon() を使った Hello World

loop.call_soon() メソッドを使ってコールバックをスケジュールする例です。コールバックは "Hello World" を出力しイベントループを停止します:

```
import asyncio

def hello_world(loop):
    """A callback to print 'Hello World' and stop the event loop"""
    print('Hello World')
    loop.stop()

loop = asyncio.get_event_loop()

# Schedule a call to hello_world()
loop.call_soon(hello_world, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

参考:

コルーチンと *run()* 関数を使用した同じような *Hello World* の例。

call_later() で現在の日時を表示する

毎秒現在時刻を表示するコールバックの例です。コールバックは *loop.call_later()* メソッドを使って自身を 5 秒後に実行するよう再スケジュールし、イベントループを停止します:

```
import asyncio
import datetime

def display_date(end_time, loop):
    print(datetime.datetime.now())
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, display_date, end_time, loop)
```

(次のページに続く)

(前のページからの続き)

```
    else:
        loop.stop()

loop = asyncio.get_event_loop()

# Schedule the first call to display_date()
end_time = loop.time() + 5.0
loop.call_soon(display_date, end_time, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

参考:

コルーチンと *run()* 関数を使用した同じような **現在時刻出力** の例。

読み込みイベント用ファイル記述子の監視

ファイル記述子が *loop.add_reader()* メソッドを使って何らかのデータを受信するまで待機し、その後イベントループをクローズします:

```
import asyncio
from socket import socketpair

# Create a pair of connected file descriptors
rsock, wsock = socketpair()

loop = asyncio.get_event_loop()

def reader():
    data = rsock.recv(100)
    print("Received:", data.decode())

    # We are done: unregister the file descriptor
    loop.remove_reader(rsock)

    # Stop the event loop
    loop.stop()

# Register the file descriptor for read event
loop.add_reader(rsock, reader)

# Simulate the reception of data from the network
loop.call_soon(wsock.send, 'abc'.encode())

try:
    # Run the event loop
```

(次のページに続く)

(前のページからの続き)

```
    loop.run_forever()
finally:
    # We are done. Close sockets and the event loop.
    rsock.close()
    wsock.close()
    loop.close()
```

参考:

- トランスポート、プロトコル、および *loop.create_connection()* メソッドを使用した同じような例。
- 高水準の *asyncio.open_connection()* 関数とストリームを使用したもうひとつの 実装例。

SIGINT および SIGTERM 用のシグナルハンドラーの設定

(ここに挙げる signals の例は Unix でのみ動きます。)

loop.add_signal_handler() メソッドを使用して SIGINT と SIGTERM の 2 つのシグナルに対するハンドラを登録します:

```
import asyncio
import functools
import os
import signal

def ask_exit(signame, loop):
    print("got signal %s: exit" % signame)
    loop.stop()

async def main():
    loop = asyncio.get_running_loop()

    for signame in {'SIGINT', 'SIGTERM'}:
        loop.add_signal_handler(
            getattr(signal, signame),
            functools.partial(ask_exit, signame, loop))

    await asyncio.sleep(3600)

print("Event loop running for 1 hour, press Ctrl+C to interrupt.")
print(f"pid {os.getpid()}: send SIGINT or SIGTERM to exit.")

asyncio.run(main())
```

18.1.8 Future

ソースコード: Lib/asyncio/futures.py, Lib/asyncio/base_futures.py

Future オブジェクトは **低水準のコールバックベースのコード** と高水準の async/await コードとの間を橋渡しします。

Future の関数

asyncio.**isfuture**(*obj*)

オブジェクト *obj* が下記のいずれかであれば `True` を返します:

- *asyncio.Future* クラスのインスタンス
- *asyncio.Task* クラスのインスタンス
- `_asyncio_future_blocking` 属性を持った Future 的なオブジェクト

バージョン 3.5 で追加.

asyncio.**ensure_future**(*obj*, ***, *loop=None*)

下記のいずれかを返します:

- *obj* が *Future* オブジェクト、*Task* オブジェクト、または Future 的なオブジェクト (*isfuture()* 関数を使って検査します) である場合には *obj* そのもの
- 引数 *obj* がコルーチンである (*iscoroutine()* 関数を使って検査します) 場合には *obj* をラップした *Task* オブジェクト; この場合コルーチンは `ensure_future()` によりスケジュールされます。
- 引数 *obj* が awaitable である (*inspect.isawaitable()* 関数を使って検査します) 場合には *obj* を await する *Task* オブジェクト

引数 *obj* が上記のいずれにもあてはまらない場合は *TypeError* 例外を送出します。

重要: Task を生成するより好ましい方法である *create_task()* 関数も参照してください。

バージョン 3.5.1 で変更: この関数はどんな *awaitable* なオブジェクトでも受け入れるようになりました。

asyncio.**wrap_future**(*future*, ***, *loop=None*)

concurrent.futures.Future オブジェクトを *asyncio.Future* オブジェクトでラップします。

Future オブジェクト

class asyncio.Future(*, *loop=None*)

Future は非同期処理の最終結果を表すクラスです。スレッドセーフではありません。

Future は *awaitable* オブジェクトです。コルーチンは Future オブジェクトが結果を返すか、例外をセットするか、もしくはキャンセルされるまで待機する (await) ことができます。

典型的には、Future は低水準のコールバックベースのコード (たとえば asyncio の *transports* を使って実装されたプロトコル) が高水準の async/await と相互運用することを可能にするために利用されます。

経験則は Future オブジェクトをユーザー向けの API であらわに利用しないことです。推奨される Future オブジェクトの生成方法は *loop.create_future()* を呼び出すことです。これによりイベントループの代替実装は、自身に最適化された Future オブジェクトの実装を合わせて提供することができます。

バージョン 3.7 で変更: *contextvars* モジュールのサポートを追加。

result()

Future の結果を返します。

Future が **完了** していて、*set_result()* メソッドにより設定された結果を持っている場合は結果の値を返します。

Future が **完了** していて、*set_exception()* メソッドにより設定された例外を持っている場合はその例外を送出します。

Future が **キャンセルされた** 場合、このメソッドは *CancelledError* 例外を送出します。

Future の結果が未設定の場合、このメソッドは *InvalidStateError* 例外を送出します。

set_result(*result*)

Future を **完了** とマークし、結果を設定します。

Future がすでに **完了** している場合 *InvalidStateError* 例外を送出します。

set_exception(*exception*)

Future を **完了** とマークし、例外を設定します。

Future がすでに **完了** している場合 *InvalidStateError* 例外を送出します。

done()

Future が **完了** しているなら True を返します。

Future は **キャンセルされた** か、または *set_result()* メソッドや *set_exception()* メソッドの呼び出しにより結果や例外が設定された場合に **完了** とみなされます。

cancelled()

Future が **キャンセルされた** 場合に True を返します。

通常の場合、このメソッドは Future に処理結果や例外を設定する前に Future が **キャンセルされていない** ことを確認するために使用されます:

```
if not fut.cancelled():
    fut.set_result(42)
```

add_done_callback(*callback*, *, *context=None*)

Future が **完了** したときに実行されるコールバックを追加します。

callback は Future オブジェクトだけを引数にとって呼び出されます。

このメソッドが呼び出される時点ですでに Future が **完了** している場合、コールバックは *loop.call_soon()* メソッドによりスケジュールされます。

オプションのキーワード引数 *context* を使って、コールバック*callback* を実行する際のコンテキスト *contextvars.Context* を設定することができます。コンテキスト *context* が指定されない場合は現在のコンテキストが使われます。

コールバックにパラメータを渡すには、次の例のように *functools.partial()* を使うことができます:

```
# Call 'print("Future:", fut)' when "fut" is done.
fut.add_done_callback(
    functools.partial(print, "Future:"))
```

バージョン 3.7 で変更: キーワード引数 *context* が追加されました。詳細は **PEP 567** を参照してください。

remove_done_callback(*callback*)

コールバックリストから *callback* を削除します。

削除されたコールバックの数を返します。コールバックが複数回追加されていない限り、通常は 1 が返ります。

cancel()

Future をキャンセルし、コールバックをスケジュールします。

Future がすでに **完了** または **キャンセル** された場合、`False` を返します。そうでない場合 Future の状態を **キャンセル** に変更した上でコールバックをスケジュールし、`True` を返します。

exception()

この Future オブジェクトに設定された例外を返します。

例外 (または例外が設定されていないときは `None`) は Future が **完了** している場合のみ返されます。

Future が **キャンセルされた** 場合、このメソッドは *CancelledError* 例外を送出します。

Future が **未完了** の場合、このメソッドは *InvalidStateError* 例外を送出します。

get_loop()

Future オブジェクトが束縛されているイベントループを返します。

バージョン 3.7 で追加.

この例は Future オブジェクトを生成し、Future に結果を設定するための非同期タスクを生成してスケジュールし、そして Future に結果が設定されるまで待機します:

```
async def set_after(fut, delay, value):
    # Sleep for *delay* seconds.
    await asyncio.sleep(delay)

    # Set *value* as a result of *fut* Future.
    fut.set_result(value)

async def main():
    # Get the current event loop.
    loop = asyncio.get_running_loop()

    # Create a new Future object.
    fut = loop.create_future()

    # Run "set_after()" coroutine in a parallel Task.
    # We are using the low-level "loop.create_task()" API here because
    # we already have a reference to the event loop at hand.
    # Otherwise we could have just used "asyncio.create_task()".
    loop.create_task(
        set_after(fut, 1, '... world'))

    print('hello ...')

    # Wait until *fut* has a result (1 second) and print it.
    print(await fut)

asyncio.run(main())
```

重要: Future オブジェクトは *concurrent.futures.Future* を模倣してデザインされました。両者の重要な違いは以下の通りです:

- asyncio の Futures と異なり、*concurrent.futures.Future* インスタンスは待ち受けできません。
- *asyncio.Future.result()* と *asyncio.Future.exception()* は *timeout* 引数を取りません。
- *asyncio.Future.result()* と *asyncio.Future.exception()* は Future が **未完了** の場合に *InvalidStateError* 例外を送出します。
- *asyncio.Future.add_done_callback()* メソッドによって登録されたコールバックは、即座に呼び出されません。代わりにコールバックは *loop.call_soon()* によりスケジュールされます。
- asyncio の Future は *concurrent.futures.wait()* および *concurrent.futures.as_completed()* との互換性がありません。

18.1.9 トランスポートとプロトコル

まえがき

トランスポートとプロトコルは *loop.create_connection()* のような **低水準の** イベントループ API から使われます。これらはコールバックに基づくプログラミングスタイルを使うことでネットワークや IPC プロトコル (HTTP など) の高性能な実装を可能にします。

基本的にトランスポートとプロトコルはライブラリやフレームワークからのみ使われるべきであり、高水準の asyncio アプリケーションから使われるものではありません。

このドキュメントは *Transports* と *Protocols* を扱います。

はじめに

最上位の観点からは、トランスポートは **どのように** バイトデータを送信するかに影響を与え、いっぽうプロトコルは **どの** バイトデータを送信するかを決定します (また、ある程度は **いつ** も決定します) 。

同じことを違う言い方で表現します: トランスポートはソケット (または同様の I/O 端点) の抽象化であり、プロトコルはトランスポートから見たときのアプリケーションの抽象化です。

さらにもう一つの見方として、トランスポートとプロトコルの 2 つのインターフェースは、協調してネットワーク I/O やプロセス間 I/O の抽象インターフェースを定義しています。

トランスポートオブジェクトとプロトコルオブジェクトの間には常に 1 対 1 の関係があります: プロトコルはデータを送信するためにトランスポートのメソッドを呼び出し、トランスポートは受信したデータを渡すためにプロトコルのメソッドを呼び出します。

ほとんどの接続に基づくイベントループメソッド (*loop.create_connection()* など) は通常 *protocol_factory* 引数を受け取り、*Transport* オブジェクトで表現される確立した接続に対する *Protocol* オブジェクトを生成するために使います。そのようなメソッドは通常 `(transport, protocol)` タプルを返します。

内容

このページは以下の節から構成されます:

- *Transports* 節は asyncio の *BaseTransport*, *ReadTransport*, *WriteTransport*, *Transport*, *DatagramTransport*, および *SubprocessTransport* クラスについて記述しています。
- *Protocols* 節は asyncio の *BaseProtocol*, *Protocol*, *BufferedProtocol*, *DatagramProtocol*, および *SubprocessProtocol* クラスについて記述しています。
- *Examples* 節はトランスポート、プロトコル、および低水準のイベントループ API の利用方法を紹介しています。

トランスポート

ソースコード: Lib/asyncio/transports.py

トランスポートはさまざまな通信方法を抽象化するために *asyncio* が提供するクラス群です。

トランスポートオブジェクトは常に *asyncio* **イベントループ** によってインスタンス化されます。

asyncio は TCP, UDP, SSL, およびサブプロセスパイプのトランスポートを実装しています。利用可能なトランスポートのメソッドはトランスポートの種類に依存します。

トランスポートクラスは **スレッド安全ではありません**。

トランスポートのクラス階層構造

class asyncio.**BaseTransport**

全てのトランスポートの基底クラスです。すべての asyncio トランスポートが共有するメソッドを含んでいます。

class asyncio.**WriteTransport**(*BaseTransport*)

書き込み専用の接続に対する基底トランスポートクラスです。

WriteTransport クラスのインスタンスは *loop.connect_write_pipe()* イベントループメソッドから返され、*loop.subprocess_exec()* のようなサブプロセスに関連するメソッドから利用されます。

class asyncio.**ReadTransport**(*BaseTransport*)

読み込み専用の接続に対する基底トランスポートクラスです。

Instances of the *ReadTransport* class are returned from the *loop.connect_read_pipe()* event loop method and are also used by subprocess-related methods like *loop.subprocess_exec()*.

class asyncio.**Transport**(*WriteTransport*, *ReadTransport*)

TCP 接続のような、読み出しと書き込みの双方向のトランスポートを表現するインターフェースです。

ユーザーはトランスポートを直接インスタンス化することはありません; ユーザーは、ユーティリティ関数にプロトコルファクトリとその他トランスポートとプロトコルを作成するために必要な情報を渡して呼び出します。

Transport クラスのインスタンスは、*loop.create_connection()*, *loop.create_unix_connection()*, *loop.create_server()*, *loop.sendfile()* などのイベントループメソッドから返されたり、これらのメソッドから利用されたりします。

class asyncio.**DatagramTransport**(*BaseTransport*)

データグラム (UDP) 接続のためのトランスポートです。

DatagramTransport クラスのインスタンスは *loop.create_datagram_endpoint()* イベントループメソッドから返されます。

class asyncio.**SubprocessTransport**(*BaseTransport*)

親プロセスとその子プロセスの間の接続を表現する抽象クラスです。

SubprocessTransport クラスのインスタンスは *loop.subprocess_shell()* と *loop.subprocess_exec()* の 2 つのイベントループメソッドから返されます。

基底トランスポート

BaseTransport.close()
トランスポートをクローズします。

トランスポートが発信データのバッファーを持っていた場合、バッファーされたデータは非同期にフラッシュされます。それ以降データは受信されません。バッファーされていたデータがすべてフラッシュされた後、そのプロトコルの *protocol.connection_lost()* メソッドが引数 *None* で呼び出されます。

BaseTransport.is_closing()
トランスポートを閉じている最中か閉じていた場合 True を返します。

BaseTransport.get_extra_info(*name*, *default=None*)
トランスポートまたはそれが背後で利用しているリソースの情報を返します。

name は取得するトランスポート特有の情報を表す文字列です。

default は、取得したい情報が取得可能でなかったり、サードパーティのイベントループ実装や現在のプラットフォームがその情報の問い合わせをサポートしていない場合に返される値です。

例えば、以下のコードはトランスポート内のソケットオブジェクトを取得しようとします:

```
sock = transport.get_extra_info('socket')
if sock is not None:
    print(sock.getsockopt(...))
```

いくつかのトランスポートで問い合わせ可能な情報のカテゴリを示します:

- ソケット:
  - 'peername': ソケットが接続されているリモートアドレスで、*socket.socket.getpeername()* の結果になります (エラーのときは None)
  - 'socket': *socket.socket* のインスタンスになります
  - 'sockname': ソケット自身のアドレスで、*socket.socket.getsockname()* の結果になります
- SSL ソケット:
  - 'compression': 圧縮アルゴリズムで、*ssl.SSLSocket.compression()* の結果になります。圧縮されていないときは None になります
  - 'cipher': 3 個の値 (使用されている暗号アルゴリズムの名称、使用が定義されている SSL プロトコルのバージョン、および使用されている秘密鍵のビット数) からなるタプルで、*ssl.SSLSocket.cipher()* の結果になります

  - 'peercert': ピアの証明書で、*ssl.SSLSocket.getpeercert()* の結果になります
  - 'sslcontext': *ssl.SSLContext* のインスタンスになります
  - 'ssl_object': *ssl.SSLObject* または *ssl.SSLSocket* インスタンス

- パイプ:
  - 'pipe': パイプオブジェクトです

- サブプロセス:
  - 'subprocess': *subprocess.Popen* のインスタンスになります

BaseTransport.**set_protocol**(*protocol*)
トランスポートに新しいプロトコルを設定します。

プロトコルの切り替えは、両方のプロトコルのドキュメントで切り替えがサポートされている場合にのみ行うべきです。

BaseTransport.**get_protocol**()
現在のプロトコルを返します。

読み出し専用のトランスポート

ReadTransport.**is_reading**()
トランスポートが新しいデータを受信中の場合 True を返します。

バージョン 3.7 で追加.

ReadTransport.**pause_reading**()
トランスポートの受信側を一時停止します。*resume_reading()* メソッドが呼び出されるまでプロトコルの *protocol.data_received()* メソッドにデータは渡されません。

バージョン 3.7 で変更: このメソッドはべき等です。すなわちトランスポートがすでに停止していたりクローズしていても呼び出すことができます。

ReadTransport.**resume_reading**()
受信を再開します。データが読み込み可能になるとプロトコルの *protocol.data_received()* メソッドが再び呼び出されるようになります。

バージョン 3.7 で変更: このメソッドはべき等です。すなわちトランスポートがすでにデータを読み込み中であっても呼び出すことができます。

書き込み専用のトランスポート

WriteTransport.**abort**()
: 未完了の処理が完了するのを待たず、即座にトランスポートをクローズします。バッファーされているデータは失われます。このメソッドの呼び出し以降データは受信されません。最終的にプロトコルの *protocol.connection_lost()* メソッドが引数 *None* で呼び出されます。

WriteTransport.**can_write_eof**()
: トランスポートが *write_eof()* メソッドをサポートしている場合 *True* を返し、そうでない場合は *False* を返します。

WriteTransport.**get_write_buffer_size**()
: トランスポートで使用されている出力バッファーの現在のサイズを返します。

WriteTransport.**get_write_buffer_limits**()
: 書き込みフロー制御の **最高** および **最低** 水位点を取得します。(low, high) タプルを返します。ここで *low* と *high* はバイト数をあらわす正の整数です。

  水位点の設定は *set_write_buffer_limits()* で行います。

  バージョン 3.4.2 で追加.

WriteTransport.**set_write_buffer_limits**(*high=None*, *low=None*)
: 書き込みフロー制御の **最高** および **最低** 水位点を設定します。

  (バイト数をあらわす) これら 2 つの値はプロトコルの *protocol.pause_writing()* と *protocol.resume_writing()* の 2 つのメソッドがいつ呼ばれるかを制御します。指定する場合、*low* は *high* と等しいかまたは *high* より小さくなければなりません。また、*high* も *low* も負の値を指定することはできません。

  pause_writing() はバッファーサイズが *high* の値以上になった場合に呼び出されます。書き込みが一時停止している場合、バッファーサイズが *low* の値以下になると *resume_writing()* メソッドが呼び出されます。

  デフォルト値は実装固有になります。*high* のみ与えられた場合、*low* は *high* 以下の実装固有のデフォルト値になります。*high* をゼロに設定すると *low* も強制的にゼロになり、バッファーが空でなくなるとすぐに *pause_writing()* メソッドが呼び出されるようになります。*low* をゼロに設定すると、バッファーが空にな *resume_writing()* が呼び出されるようになります。どちらかにゼロを設定することは I/O と計算を並行に実行する機会を減少させるため、一般に最適ではありません。

  上限値と下限値を取得するには *get_write_buffer_limits()* メソッドを使ってください。

WriteTransport.**write**(*data*)
: トランスポートにバイト列 *data* を書き込みます。

  このメソッドはブロックしません; データをバッファーし、非同期に送信する準備を行います。

WriteTransport.**writelines**(*list_of_data*)
: バイト列のデータのリスト (またはイテラブル) をトランスポートに書き込みます。この振る舞いはイ

テラブルを yield して各要素で *write()* を呼び出すことと等価ですが、より効率的な実装となる場合があります。

WriteTransport.**write_eof**()
バッファーされた全てのデータをフラッシュした後トランスポートの送信側をクローズします。送信側をクローズした後もデータを受信することは可能です。

このメソッドはトランスポート (例えば SSL) がハーフクローズドな接続をサポートしていない場合 *NotImplementedError* を送出します。

データグラムトランスポート

DatagramTransport.**sendto**(*data*, *addr=None*)
リモートピア *addr* (トランスポート依存の対象アドレス) にバイト列 *data* を送信します。*addr* が *None* の場合、データはトランスポートの作成時に指定された送信先に送られます。

このメソッドはブロックしません; データをバッファーし、非同期に送信する準備を行います。

DatagramTransport.**abort**()
未完了の処理が完了するのを待たず、即座にトランスポートをクローズします。バッファーされているデータは失われます。このメソッドの呼び出し以降データは受信されません。最終的にプロトコルの *protocol.connection_lost()* メソッドが引数 *None* で呼び出されます。

サブプロセス化されたトランスポート

SubprocessTransport.**get_pid**()
サブプロセスのプロセス ID (整数) を返します。

SubprocessTransport.**get_pipe_transport**(*fd*)
整数のファイル記述子 *fd* に該当する通信パイプのトランスポートを返します:

- 0: 標準入力 (*stdin*) の読み込み可能ストリーミングトランスポート。サブプロセスが `stdin=PIPE` で作成されていない場合は *None*
- 1: 標準出力 (*stdout*) の書き込み可能ストリーミングトランスポート。サブプロセスが `stdout=PIPE` で作成されていない場合は *None*
- 2: 標準エラー出力 (*stderr*) の書き込み可能ストリーミングトランスポート。サブプロセスが `stderr=PIPE` で作成されていない場合は *None*
- その他の *fd*: *None*

SubprocessTransport.**get_returncode**()
サブプロセスのリターンコードを整数で返します。サブプロセスがリターンしなかった場合は *None* を返します。*subprocess.Popen.returncode* 属性と同じです。

SubprocessTransport.**kill**()
サブプロセスを強制終了 (kill) します。

POSIX システムでは、この関数はサブプロセスに SIGKILL を送信します。Windows では、このメソッドは *terminate()* の別名です。

subprocess.Popen.kill() も参照してください。

SubprocessTransport.**send_signal**(*signal*)
サブプロセスにシグナル *signal* を送信します。*subprocess.Popen.send_signal()* と同じです。

SubprocessTransport.**terminate**()
サブプロセスを停止します。

POSIX システムでは、このメソッドはサブプロセスに SIGTERM を送信します。Windows では、Windows API 関数 TerminateProcess() が呼び出されます。

subprocess.Popen.terminate() も参照してください。

SubprocessTransport.**close**()
kill() メソッドを呼び出すことでサブプロセスを強制終了します。

サブプロセスがまだリターンしていない場合、*stdin*, *stdout*, および *stderr* の各パイプのトランスポートをクローズします。

プロトコル

ソースコード: Lib/asyncio/protocols.py

---

asyncio はネットワークプロトコルを実装するために使う抽象基底クラス群を提供します。これらのクラスは **トランスポート** と組み合わせて使うことが想定されています。

抽象基底プロトコルクラスの派生クラスはメソッドの一部または全てを実装することができます。これらのメソッドは全てコールバックです: それらは、データを受信した、などの決まったイベントに対してトランスポートから呼び出されます。基底プロトコルメソッドは対応するトランスポートから呼び出されるべきです。

基底プロトコル

class asyncio.**BaseProtocol**
全てのプロトコルクラスが共有する全てのメソッドを持った基底プロトコルクラスです。

class asyncio.**Protocol**(*BaseProtocol*)
ストリーミングプロトコル (TCP, Unix ソケットなど) を実装するための基底クラスです。

class asyncio.**BufferedProtocol**(*BaseProtocol*)
受信バッファーを手動で制御するストリーミングプロトコルを実装するための基底クラスです。

class asyncio.**DatagramProtocol**(*BaseProtocol*)
データグラム (UDP) プロトコルを実装するための基底クラスです。

class asyncio.**SubprocessProtocol**(*BaseProtocol*)
子プロセスと (一方向パイプを通じて) 通信するプロトコルを実装するための基底クラスです。

基底プロトコル

全ての asyncio プロトコルは基底プロトコルのコールバックを実装することができます。

通信のコールバック

コネクションコールバックは全てのプロトコルから、成功したコネクションそれぞれにつきただ一度だけ呼び出されます。その他の全てのプロトコルコールバックはこれら 2 つのメソッドの間に呼び出すことができます。

BaseProtocol.**connection_made**(*transport*)

コネクションが作成されたときに呼び出されます。

引数 *transport* はコネクションをあらわすトランスポートです。プロトコルはトランスポートへの参照を保存する責任を負います。

BaseProtocol.**connection_lost**(*exc*)

コネクションが失われた、あるいはクローズされたときに呼び出されます。

引数は例外オブジェクトまたは *None* になります。*None* のとき、通常の EOF が受信されたか、あるいはコネクションがこちら側から中止またはクローズされたことを意味します。

フロー制御コールバック

フロー制御コールバックは、プロトコルによって実行される書き込み処理を停止または再開するためにトランスポートから呼び出されます。

詳しくは *set_write_buffer_limits()* メソッドのドキュメントを参照してください。

BaseProtocol.**pause_writing**()

トランスポートのバッファーサイズが最高水位点 (high watermark) を超えたときに呼び出されます。

BaseProtocol.**resume_writing**()

トランスポートのバッファーサイズが最低水位点 (low watermark) に達したきに呼び出されます。

バッファーサイズが最高水位点と等しい場合、*pause_writing()* は呼び出されません: バッファーサイズは必ず制限値を超えなければなりません。

それに対して、*resume_writing()* はバッファーサイズが最低水位点と等しいかそれよりも小さい場合に呼び出されます。これらの境界条件は、どちらの基準値もゼロである場合の処理が期待通りとなることを保証するために重要です。

ストリーミングプロトコル

loop.create_server(), *loop.create_unix_server()*, *loop.create_connection()*, *loop.create_unix_connection()*, *loop.connect_accepted_socket()*, *loop.connect_read_pipe()*, そして *loop.connect_write_pipe()* などのイベントメソッドはストリーミングプロトコルを返すファクトリを受け付けます。

Protocol.**data_received**(*data*)

データを受信したときに呼び出されます。*data* は受信したデータを含む空ではないバイト列オブジェクトになります。

データがバッファーされるか、チャンキングされるか、または再構築されるかはトランスポートに依存します。一般には、特定のセマンティクスを信頼するべきではなく、代わりにデータのパースを全般的かつ柔軟に行うべきです。ただし、データは常に正しい順序で受信されます。

このメソッドは、コネクションがオープンである間は何度でも呼び出すことができます。

いっぽうで、*protocol.eof_received()* メソッドは最大でも一度だけ呼び出されます。いったん *eof_received()* が呼び出されると、それ以降 `data_received()` は呼び出されません。

Protocol.**eof_received**()

コネクションの相手方がこれ以上データを送信しないことを伝えてきたとき (例えば相手方が asyncio を使用しており、*transport.write_eof()* を呼び出した場合) に呼び出されます。

このメソッドは (`None` を含む) 偽値 を返すことがあり、その場合トランスポートは自身をクローズします。一方メソッドが真値を返す場合は、利用しているプロトコルがトランスポートをクローズするかどうかを決めます。デフォルトの実装は `None` を返すため、コネクションは暗黙のうちにクローズされます。

SSL を含む一部のトランスポートはハーフクローズ接続をサポートしません。そのような場合このメソッドが真値を返すとコネクションはクローズされます。

ステートマシン:

```
start -> connection_made
    [-> data_received]*
    [-> eof_received]?
-> connection_lost -> end
```

バッファリングされたストリーミングプロトコル

バージョン 3.7 で追加: **Important:** this has been added to asyncio in Python 3.7 *on a provisional basis*! This is as an experimental API that might be changed or removed completely in Python 3.8.

バッファー付きプロトコルは ストリーミングプロトコル をサポートするイベントループメソッドで利用することができます。

`BufferedProtocol` 実装は受信バッファーの手動での明示的な割り当てや制御を可能にします。イベントループはプロトコルにより提供されるバッファを利用することにより不要なデータのコピーを避けることがで

きます。これにより大量のデータを受信するプロトコルにおいて顕著なパフォーマンスの向上をもたらします。精巧なプロトコル実装によりバッファー割り当ての数を劇的に減少させることができます。

以下に示すコールバックは *BufferedProtocol* インスタンスに対して呼び出されます:

BufferedProtocol.**get_buffer**(*sizehint*)

新しい受信バッファを割り当てるために呼び出します。

sizehint は返されるバッファーの推奨される最小サイズです。*sizehint* によって推奨された値より小さい、または大きいサイズのバッファーを返すことは容認されています。-1 がセットされた場合、バッファーサイズは任意となります。サイズがゼロのバッファーを返すとエラーになります。

get_buffer() は バッファープロトコル を実装したオブジェクトを返さなければなりません。

BufferedProtocol.**buffer_updated**(*nbytes*)

受信データによりバッファが更新された場合に呼び出されます。

nbytes はバッファに書き込まれた総バイト数です。

BufferedProtocol.**eof_received**()

protocol.eof_received() メソッドのドキュメントを参照してください。

コネクションの間、*get_buffer()* は何度でも呼び出すことができます。しかし *protocol.eof_received()* が呼び出されるのは最大でも 1 回で、もし呼び出されると、それ以降 *get_buffer()* と *buffer_updated()* が呼び出されることはありません。

ステートマシン:

```
start -> connection_made
    [-> get_buffer
        [-> buffer_updated]?
    ]*
    [-> eof_received]?
-> connection_lost -> end
```

データグラムプロトコル

データグラムプロトコルのインスタンスは *loop.create_datagram_endpoint()* メソッドに渡されたプロトコルファクトリによって生成されるべきです。

DatagramProtocol.**datagram_received**(*data*, *addr*)

データグラムを受信したときに呼び出されます。*data* は受信データを含むバイトオブジェクトです。*addr* はデータを送信するピアのアドレスです; 正確な形式はトランスポートに依存します。

DatagramProtocol.**error_received**(*exc*)

直前の送信あるいは受信が *OSError* を送出したときに呼び出されます。*exc* は *OSError* のインスタンスになります。

このメソッドが呼ばれるのは、トランスポート (UDP など) がデータグラムを受信側に配信できなかったことが検出されたなどの、まれな場合においてのみです。ほとんどの場合、データグラムが配信でき

なければそのまま通知されることなく破棄されます。

注釈: BSD システム (macOS, FreeBSD など) ではフロー制御はサポートされていません。これは非常に多くのパケットを書き込もうとしたことによる送信の失敗を検出する信頼できる方法が存在しないためです。

ソケットは常に ' 準備ができた状態' のように振る舞いますが、超過したパケットは破棄されます。この場合 errno を *errno.ENOBUFS* に設定した *OSError* 例外が送出されることがあります。もし例外が送出された場合は *DatagramProtocol.error_received()* に通知されますが、送出されない場合は単に無視されます。

サブプロセスプロトコル

Datagram Protocol instances should be constructed by protocol factories passed to the *loop.subprocess_exec()* and *loop.subprocess_shell()* methods.

SubprocessProtocol.**pipe_data_received**(*fd*, *data*)

子プロセスが標準出力または標準エラー出力のパイプにデータを書き込んだ時に呼び出されます。

fd はパイプのファイル記述子を表す整数です。

data は受信データを含む空でないバイトオブジェクトです。

SubprocessProtocol.**pipe_connection_lost**(*fd*, *exc*)

子プロセスと通信するパイプのいずれかがクローズされたときに呼び出されます。

fd はクローズされたファイル記述子を表す整数です。

SubprocessProtocol.**process_exited**()

子プロセスが終了したときに呼び出されます。

使用例

TCP エコーサーバー

loop.create_server() メソッドを使って TCP エコーサーバーを生成し、受信したデータをそのまま送り返して、最後にコネクションをクローズします:

```
import asyncio


class EchoServerProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info('peername')
        print('Connection from {}'.format(peername))
        self.transport = transport

    def data_received(self, data):
        message = data.decode()
```

(次のページに続く)

(前のページからの続き)

```
        print('Data received: {!r}'.format(message))

        print('Send: {!r}'.format(message))
        self.transport.write(data)

        print('Close the client socket')
        self.transport.close()


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    server = await loop.create_server(
        lambda: EchoServerProtocol(),
        '127.0.0.1', 8888)

    async with server:
        await server.serve_forever()


asyncio.run(main())
```

参考:

ストリームを使った *TCP* エコーサーバー の例では高水準の *asyncio.start_server()* 関数を使っています。

TCP エコークライアント

loop.create_connection() メソッドを使った TCP エコークライアントは、データを送信したあとコネクションがクローズされるまで待機します:

```
import asyncio


class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        transport.write(self.message.encode())
        print('Data sent: {!r}'.format(self.message))

    def data_received(self, data):
        print('Data received: {!r}'.format(data.decode()))

    def connection_lost(self, exc):
```

(次のページに続く)

(前のページからの続き)

```
        print('The server closed the connection')
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = 'Hello World!'

    transport, protocol = await loop.create_connection(
        lambda: EchoClientProtocol(message, on_con_lost),
        '127.0.0.1', 8888)

    # Wait until the protocol signals that the connection
    # is lost and close the transport.
    try:
        await on_con_lost
    finally:
        transport.close()


asyncio.run(main())
```

参考:

ストリームを使った *TCP* エコークライアント の例では高水準の *asyncio.open_connection()* 関数を使っています。

UDP エコーサーバー

loop.create_datagram_endpoint() メソッドを使った UDP エコーサーバーは受信したデータをそのまま送り返します:

```
import asyncio


class EchoServerProtocol:
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        message = data.decode()
        print('Received %r from %s' % (message, addr))
        print('Send %r to %s' % (message, addr))
        self.transport.sendto(data, addr)
```

(次のページに続く)

(前のページからの続き)

```
async def main():
    print("Starting UDP server")

    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    # One protocol instance will be created to serve all
    # client requests.
    transport, protocol = await loop.create_datagram_endpoint(
        lambda: EchoServerProtocol(),
        local_addr=('127.0.0.1', 9999))

    try:
        await asyncio.sleep(3600)  # Serve for 1 hour.
    finally:
        transport.close()


asyncio.run(main())
```

UDP エコークライアント

loop.create_datagram_endpoint() メソッドを使った UDP エコークライアントはデータを送信し、応答を受信するとトランスポートをクローズします:

```
import asyncio


class EchoClientProtocol:
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost
        self.transport = None

    def connection_made(self, transport):
        self.transport = transport
        print('Send:', self.message)
        self.transport.sendto(self.message.encode())

    def datagram_received(self, data, addr):
        print("Received:", data.decode())

        print("Close the socket")
        self.transport.close()

    def error_received(self, exc):
        print('Error received:', exc)

    def connection_lost(self, exc):
```

(次のページに続く)

(前のページからの続き)

```
        print("Connection closed")
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = "Hello World!"

    transport, protocol = await loop.create_datagram_endpoint(
        lambda: EchoClientProtocol(message, on_con_lost),
        remote_addr=('127.0.0.1', 9999))

    try:
        await on_con_lost
    finally:
        transport.close()


asyncio.run(main())
```

既存のソケットへの接続

プロトコルを設定した *loop.create_connection()* メソッドを使ってソケットがデータを受信するまで待機します:

```
import asyncio
import socket


class MyProtocol(asyncio.Protocol):

    def __init__(self, on_con_lost):
        self.transport = None
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        self.transport = transport

    def data_received(self, data):
        print("Received:", data.decode())

        # We are done: close the transport;
        # connection_lost() will be called automatically.
        self.transport.close()

    def connection_lost(self, exc):
```

(次のページに続く)

(前のページからの続き)

```
        # The socket has been closed
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()
    on_con_lost = loop.create_future()

    # Create a pair of connected sockets
    rsock, wsock = socket.socketpair()

    # Register the socket to wait for data.
    transport, protocol = await loop.create_connection(
        lambda: MyProtocol(on_con_lost), sock=rsock)

    # Simulate the reception of data from the network.
    loop.call_soon(wsock.send, 'abc'.encode())

    try:
        await protocol.on_con_lost
    finally:
        transport.close()
        wsock.close()

asyncio.run(main())
```

参考:

ファイル記述子の読み込みイベントを監視する 例では低レベルの *loop.add_reader()* メソッドを使ってファイル記述子 (FD) を登録しています。

ストリームを使ってデータを待ち受けるオープンなソケットを登録する 例ではコルーチン内で *open_connection()* 関数によって生成されたストリームを使っています。

loop.subprocess_exec() と SubprocessProtocol

サブプロセスからの出力を受け取り、サブプロセスが終了するまで待機するために使われるサブプロセスプロトコルの例です。

サブプロセスは *loop.subprocess_exec()* メソッドにより生成されます:

```
import asyncio
import sys

class DateProtocol(asyncio.SubprocessProtocol):
    def __init__(self, exit_future):
        self.exit_future = exit_future
        self.output = bytearray()
```

(次のページに続く)

(前のページからの続き)

```
    def pipe_data_received(self, fd, data):
        self.output.extend(data)

    def process_exited(self):
        self.exit_future.set_result(True)

async def get_date():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    code = 'import datetime; print(datetime.datetime.now())'
    exit_future = asyncio.Future(loop=loop)

    # Create the subprocess controlled by DateProtocol;
    # redirect the standard output into a pipe.
    transport, protocol = await loop.subprocess_exec(
        lambda: DateProtocol(exit_future),
        sys.executable, '-c', code,
        stdin=None, stderr=None)

    # Wait for the subprocess exit using the process_exited()
    # method of the protocol.
    await exit_future

    # Close the stdout pipe.
    transport.close()

    # Read the output which was collected by the
    # pipe_data_received() method of the protocol.
    data = bytes(protocol.output)
    return data.decode('ascii').rstrip()

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

高水準の API を使って書かれた **同様の例** も参照してください。

18.1.10 ポリシー

An event loop policy is a global per-process object that controls the management of the event loop. Each event loop has a default policy, which can be changed and customized using the policy API.

A policy defines the notion of *context* and manages a separate event loop per context. The default policy defines *context* to be the current thread.

By using a custom event loop policy, the behavior of *get_event_loop()*, *set_event_loop()*, and *new_event_loop()* functions can be customized.

ポリシーオブジェクトは *AbstractEventLoopPolicy* 抽象基底クラスで定義された API を実装しなければなりません。

ポリシーの取得と設定

以下の関数は現在のプロセスに対するポリシーの取得や設定をするために使われます:

asyncio.get_event_loop_policy()
プロセス全体にわたる現在のポリシーを返します。

asyncio.set_event_loop_policy(*policy*)
プロセス全体にわたる現在のポリシーを *policy* に設定します。

policy が None の場合、デフォルトポリシーが現在のポリシーに戻されます。

ポリシーオブジェクト

イベントループポリシーの抽象基底クラスは以下のように定義されています:

class asyncio.AbstractEventLoopPolicy
asyncio ポリシーの抽象基底クラスです。

get_event_loop()
現在のコンテキストのイベントループを取得します。

AbstractEventLoop のインターフェースを実装したイベントループオブジェクトを返します。

このメソッドは None を返してはいけません。

バージョン 3.6 で変更.

set_event_loop(*loop*)
現在のコンテキストにイベントループ *loop* を設定します。

new_event_loop()
新しいイベントループオブジェクトを生成して返します。

このメソッドは None を返してはいけません。

get_child_watcher()
子プロセスを監視するウオッチャーオブジェクトを返します。

AbstractChildWatcher のインターフェースを実装したウオッチャーオブジェクトを返します。

この関数は Unix 特有です。

set_child_watcher(*watcher*)
子プロセスに対する現在のウオッチャーオブジェクトを *watcher* に設定します。

この関数は Unix 特有です。

asyncio は以下の組み込みポリシーを提供します:

class asyncio.DefaultEventLoopPolicy

デフォルトの asyncio ポリシーです。Unix では *SelectorEventLoop* 、Windows では *ProactorEventLoop* を使います。

デフォルトのポリシーを手動でインストールする必要はありません。asyncio はデフォルトポリシーを使うように自動的に構成されます。

バージョン 3.8 で変更: Windows では *ProactorEventLoop* がデフォルトで使われるようになりました。

class asyncio.WindowsSelectorEventLoopPolicy

SelectorEventLoop イベントループ実装を使った別のイベントループポリシーです。

利用可能な環境: Windows 。

class asyncio.WindowsProactorEventLoopPolicy

ProactorEventLoop イベントループ実装を使った別のイベントループポリシーです。

利用可能な環境: Windows 。

プロセスのウオッチャー

プロセスのウオッチャーは Unix 上でイベントループが子プロセスを監視する方法をカスタマイズすることを可能にします。特に、子プロセスがいつ終了したかをイベントループは知る必要があります。

asyncio では、子プロセスは *create_subprocess_exec()* や *loop.subprocess_exec()* 関数により生成されます。

asyncio は、子プロセスのウオッチャーが実装すべき *AbstractChildWatcher* 抽象基底クラスを定義しており、さらに異なる 4 つの実装クラスを提供しています: *ThreadedChildWatcher* (デフォルトでこのクラスが使われるように構成されます), *MultiLoopChildWatcher*, *SafeChildWatcher*, そして *FastChildWatcher* です。

サブプロセスとスレッド 節も参照してください。

以下の 2 つの関数は asyncio のイベントループが使う子プロセスのウオッチャーの実装をカスタマイズするために使うことができます:

asyncio.get_child_watcher()

現在のポリシーにおける子プロセスのウオッチャーを返します。

asyncio.set_child_watcher(*watcher***)**

現在ポリシーにおける子プロセスのウオッチャーを *watcher* に設定します。*watcher* は *AbstractChildWatcher* 基底クラスで定義されたメソッドを実装していなければなりません。

注釈: サードパーティのイベントループ実装は子プロセスのウオッチャーのカスタマイズをサポートしていない可能性があります。そのようなイベントループでは、*set_child_watcher()* 関数の利用は禁止されているか、または何の効果もありません。

---

class asyncio.**AbstractChildWatcher**

add_child_handler(*pid*, *callback*, **args*)
新しい子プロセスのハンドラを登録します。

プロセス ID (PID) が *pid* であるプロセスが終了した時に callback(pid, returncode, *args) コールバック関数が呼び出されるように手配します。同じプロセスに対して別のコールバックを登録した場合、以前登録したハンドラを置き換えます。

callback はスレッドセーフな呼び出し可能オブジェクトでなければなりません。

remove_child_handler(*pid*)
プロセス ID (PID) が *pid* であるプロセスに対して登録されたハンドラを削除します。

ハンドラが正しく削除された場合 True を返します。削除するハンドラがない場合は False を返します。

attach_loop(*loop*)
ウオッチャーをイベントループに接続します。

ウオッチャーがイベントループに接続されている場合、新しいイベントループに接続される前に接続済みのイベントループから切り離されます。

注: 引数は None をとることができます。

is_active()
ウオッチャーが利用可能な状態なら True を返します。

現在の子プロセスのウオッチャーが **アクティブでない** 場合にサブプロセスを生成すると *RuntimeError* 例外が送出されます。

バージョン 3.8 で追加.

close()
ウオッチャーをクローズします。

このメソッドは、ウオッチャーの背後にあるリソースを確実にクリーンアップするために必ず呼び出さなければなりません。

class asyncio.**ThreadedChildWatcher**
この実装は、各サブプロセスの生成時に新しい待ち受けスレッドを開始します。

このクラスは asyncio イベントループがメインでない OS スレッド上で実行されていても期待通りに動きます。

大量の子プロセスを処理する際に顕著なオーバーヘッドはありません (子プロセスが終了するごとに *O(1)* 程度です) が、各プロセスに対してスレッドを開始するための追加のメモリが必要になります。

このウオッチャーはデフォルトで使われます。

バージョン 3.8 で追加.

class asyncio.MultiLoopChildWatcher

この実装はインスタンス化の際に SIGCHLD シグナルハンドラを登録します。これにより、独自の SIGCHLD シグナルハンドラをインストールするようなサードパーティのコードを壊す可能性があります。

このウオッチャーは、各プロセスに明示的に SIGCHLD シグナルをポーリングさせることにより、プロセスを生成する他のコードを中断させないようにします。

いったんウオッチャーがインストールされると、異なるスレッドからのサブプロセスの実行について特に制限はありません。

このソリューションは安全ですが、大量の子プロセスを処理する際に非常に大きなオーバーヘッドを伴います (SIGCHLD シグナルを受信するごとに *O(n)* 程度)。

バージョン 3.8 で追加.

class asyncio.SafeChildWatcher

この実装はメインスレッドでアクティブなイベントループを使って SIGCHLD シグナルを処理します。メインスレッドでイベントループが実行中でない場合、別のスレッドからサブプロセスを生成することはできません (*RuntimeError* 例外が送出されます)。

このウオッチャーは、各プロセスに明示的に SIGCHLD シグナルをポーリングさせることにより、プロセスを生成する他のコードを中断させないようにします。

このソリューションは *MultiLoopChildWatcher* と同じように安全で、同程度の *O(N)* オーバーヘッドがあります。一方で、このソリューションはメインスレッドで実行中のイベントループが必要です。

class asyncio.FastChildWatcher

この実装は終了した子プロセスを得るために直接 os.waitpid(-1) を呼び出します。これにより、プロセスを生成してその終了を待ち受ける別のコードを壊す可能性があります。

大量の子プロセスを処理する際に顕著なオーバーヘッドはありません (子プロセスが終了するごとに *O(1)* 程度です)。

このソリューションは、*SafeChildWatcher* と同様にメインスレッドで実行中のイベントループが必要です。

ポリシーのカスタマイズ

新しいイベントループのポリシーを実装するためには、以下に示すように *DefaultEventLoopPolicy* を継承して振る舞いを変更したいメソッドをオーバーライドすることが推奨されます。:

```
class MyEventLoopPolicy(asyncio.DefaultEventLoopPolicy):

    def get_event_loop(self):
        """Get the event loop.

        This may be None or an instance of EventLoop.
        """
```

(次のページに続く)

(前のページからの続き)

```
        loop = super().get_event_loop()
        # Do something with loop ...
        return loop

asyncio.set_event_loop_policy(MyEventLoopPolicy())
```

18.1.11 プラットフォームでのサポート

asyncio モジュールは可搬的であるようにデザインされていますが、いくつかのプラットフォームでは、その根底にあるアーキテクチャや性能による微妙な動作の違いや制限があります。

全てのプラットホーム

- *loop.add_reader()* と *loop.add_writer()* をファイル I/O を監視するためには使えません。

Windows

ソースコード: Lib/asyncio/proactor_events.py, Lib/asyncio/windows_events.py, Lib/asyncio/windows_utils.py

---

バージョン 3.8 で変更: Windows では *ProactorEventLoop* がデフォルトのイベントループになりました。

全ての Windows 上のイベントループは以下のメソッドをサポートしません:

- *loop.create_unix_connection()* と *loop.create_unix_server()* はサポートされません。*socket.AF_UNIX* ソケットファミリーは Unix 固有です。
- *loop.add_signal_handler()* と *loop.remove_signal_handler()* はサポートされていません。

SelectorEventLoop は以下の制限があります:

- *SelectSelector* はソケットイベントの待ち受けに使われます: このクラスはソケットをサポートしますが 512 ソケットまでに制限されています。
- *loop.add_reader()* と *loop.add_writer()* はソケットハンドルのみを受け付けます (たとえばパイプファイル記述子はサポートされていません)。
- パイプはサポートされていません。従って *loop.connect_read_pipe()* と *loop.connect_write_pipe()* の 2 つのメソッドは未実装です。
- *Subprocesses* はサポートされていません。すなわち *loop.subprocess_exec()* と *loop.subprocess_shell()* の 2 つのメソッドは未実装です。

ProactorEventLoop は以下の制限があります:

- *loop.add_reader()* と *loop.add_writer()* はサポートされていません。

The resolution of the monotonic clock on Windows is usually around 15.6 msec. The best resolution is 0.5 msec. The resolution depends on the hardware (availability of HPET) and on the Windows configuration.

Windows におけるサブプロセスのサポート

Windows において、デフォルトのイベントループ *ProactorEventLoop* はサブプロセスをサポートしますが、*SelectorEventLoop* はサポートしません。

policy.set_child_watcher() 関数もサポートされません。*ProactorEventLoop* は子プロセスを監視するための異なる仕組みを持っています。

macOS

最近の macOS バージョンは完全にサポートされています。

10.8 以前の macOS

macOS 10.6, 10.7 および 10.8 では、デフォルトイベントループは *selectors.KqueueSelector* をしていますが、このクラスはこれらの macOS バージョンのキャラクターデバイスをサポートしていません。これらの macOS バージョンでキャラクターデバイスをサポートするためには *SelectorEventLoop* で *SelectSelector* または *PollSelector* を使うように手動で設定します。以下はその例です:

```
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

18.1.12 高水準の API インデックス

このページには、すべての高水準の 非同期/待機 可能な asyncio API が一覧になっています。

タスク

ユーティリティは asyncio プログラムを実行し、タスクを作成し、タイムアウトのある複数の機能を待っています。

run()	イベントループを作成し、コルーチンを実行し、ループを閉じます。
create_task()	asyncio タスクを開始します。
await *sleep()*	数秒間スリープします。
await *gather()*	並行してスケジュールして、待ちます。
await *wait_for()*	タイムアウトで実行します。
await *shield()*	取り消しから保護します。
await *wait()*	完了かどうかを監視します。
current_task()	現在のタスクを返します。
all_tasks()	イベントループのすべてのタスクを返します。
Task	Task オブジェクト
run_coroutine_threadsafe()	別の OS スレッドからコルーチンの実行をスケジュールします。
for in *as_completed()*	for ループ向けにコルーチンの完了を監視します。

使用例

- *asyncio.gather()* **を使って複数の処理を並列に実行する** 。
- *asyncio.wait_for()* **を使って強制的にタイムアウトする** 。
- **非同期処理をキャンセルする** 。
- *asyncio.sleep()* **を使う** 。
- *Tasks* **のドキュメント** も参照してください。

キュー

キューは複数の非同期タスクの分散処理、コネクションプールや pub/sub パターンの実装に適しています。

Queue	FIFO キューです。
PriorityQueue	優先度付きのキューです。
LifoQueue	LIFO キュー (スタック) です。

使用例

- *asyncio.Queue* を使って複数のタスクを分散処理する.
- *Queue* のドキュメント も参照してください。

サブプロセス

サブプロセスを生成したり、シェルコマンドを実行するためのユーティリティです。

await *create_subprocess_exec()*	サブプロセスを作成します。
await *create_subprocess_shell()*	シェルコマンドを実行します。

使用例

- シェルコマンドを実行する。
- サブプロセス *API* のドキュメントも参照してください。

ストリーム

ネットワーク IO を利用するための高水準の APIs です。

await *open_connection()*	TCP コネクションを確立します。
await *open_unix_connection()*	Unix のソケット接続を確立します。
await *start_server()*	TCP サーバーを起動します。
await *start_unix_server()*	Unix のソケットサーバーを起動します。
StreamReader	ネットワークからデータを受信するための高水準の async/await オブジェクトです。
StreamWriter	ネットワークにデータを送信するための高水準の async/await オブジェクトです。

使用例

- *TCP* クライアントの例.
- ストリーム *API* のドキュメントも参照してください。

同期

タスク内で利用できるスレッド並列処理に似た同期プリミティブです。

Lock	ミューテックスロックです。
Event	イベントオブジェクトです。
Condition	条件変数オブジェクトです。
Semaphore	セマフォ (semaphore) です。
BoundedSemaphore	有限セマフォ (bounded semaphore) です。

使用例

- *asyncio.Event* **の使用例** 。
- asyncio の **同期プリミティブ** についてのドキュメントも参照してください。

例外

asyncio.TimeoutError	Raised on timeout by functions like *wait_for()*. Keep in mind that `asyncio.TimeoutError` is **unrelated** to the built-in *TimeoutError* exception.
asyncio.CancelledError	タスクがキャンセルされた場合に送出されます。*Task.cancel()* も参照してください。

使用例

- *CancelledError* **を処理してキャンセル要求に対応するコードを実行する**.
- *asyncio* **に特有な例外の完全なリスト** も参照してください。

18.1.13 低水準の API インデックス

このページでは低水準の asyncio API を全てリストしています。

イベントループの取得

asyncio.get_running_loop()	実行中のイベントループを取得するために **利用が推奨される** 関数です。
asyncio.get_event_loop()	Get an event loop instance (current or via the policy).
asyncio.set_event_loop()	ポリシーに基づいて引数のイベントループを ” カレント” (current event loop) に設定します。
asyncio.new_event_loop()	新しいイベントループのインスタンスを生成します。

使用例

- :ref:‘asyncio.get_running_loop() を使う <asyncio_example_future> ‘。

イベントループのメソッド

See also the main documentation section about the *event loop methods*.

ライフサイクル

loop.run_until_complete()	Future/Task/awaitable が完了するまで実行します。
loop.run_forever()	イベントループを永久に実行します。
loop.stop()	イベントループを停止します。
loop.close()	イベントループをクローズします。
loop.is_running()	イベントループが実行中の場合 `True` を返します。
loop.is_closed()	イベントループがクローズされている場合 `True` を返します。
`await` *loop.shutdown_asyncgens()*	非同期ジェネレータをクローズします。

デバッグ

loop.set_debug()	デバッグモードを有効化または無効化します。
loop.get_debug()	現在のデバッグモードを取得します。

コールバックのスケジューリング

loop.call_soon()	コールバックを即座に実行します。
loop.call_soon_threadsafe()	*loop.call_soon()* のスレッドセーフ版です。
loop.call_later()	与えられた遅延時間の **経過後** にコールバックを実行します。
loop.call_at()	**与えられた時刻に** コールバックを実行します。

スレッドプール／プロセスプール

await *loop.run_in_executor()*	CPU バウンドなブロッキング関数、またはその他のブロッキング関数を *concurrent.futures* 実行オブジェクト (executor) 上で実行します。
loop.set_default_executor()	*loop.run_in_executor()* のデフォルト実行オブジェクト (executor) を設定します。

タスクとフューチャー

loop.create_future()	*Future* オブジェクトを生成します。
loop.create_task()	コルーチンを *Task* としてスケジュールします。
loop.set_task_factory()	*loop.create_task()* が *Tasks* を生成する際に使われるファクトリを設定します。
loop.get_task_factory()	*loop.create_task()* が *Tasks* を生成するファクトリを取得します。

DNS

await *loop.getaddrinfo()*	*socket.getaddrinfo()* の非同期版です。
await *loop.getnameinfo()*	*socket.getnameinfo()* の非同期版です。

ネットワークとプロセス間通信 (IPC)

await *loop.create_connection()*	TCP 接続を確立します。
await *loop.create_server()*	TCP サーバーを起動します。
await *loop.create_unix_connection()*	Unix のソケット接続を確立します。
await *loop.create_unix_server()*	Create a Unix socket server.
await *loop.connect_accepted_socket()*	*socket* を (transport, protocol) のペアでラップします。
await *loop.create_datagram_endpoint()*	データグラム (UDP) 接続を確立します。
await *loop.sendfile()*	確立した接続 (transport) を通じてファイルを送信します。
await *loop.start_tls()*	既存の接続を TLS にアップグレードします。
await *loop.connect_read_pipe()*	パイプの読み出し側を (transport, protocol) のペアでラップします。
await *loop.connect_write_pipe()*	パイプの書き込み側を (transport, protocol) のペアでラップします。

ソケット

await *loop.sock_recv()*	*socket* からデータを受信します。
await *loop.sock_recv_into()*	*socket* からデータを受信し、バッファに送信します。
await *loop.sock_sendall()*	*socket* にデータを送信します。
await *loop.sock_connect()*	*socket* を接続します。
await *loop.sock_accept()*	*socket* の接続を受け入れます。
await *loop.sock_sendfile()*	Send a file over the *socket*.
loop.add_reader()	ファイル記述子が読み込み可能かどうかの監視を開始します。
loop.remove_reader()	ファイル記述子が読み込み可能かどうかの監視を停止します。
loop.add_writer()	ファイル記述子が書き込み可能かどうかの監視を開始します。
loop.remove_writer()	ファイル記述子が書き込み可能かどうかの監視を停止します。

Unix シグナル

loop.add_signal_handler()	*signal* 用のハンドラーを追加します。
loop.remove_signal_handler()	*signal* 用のハンドラーを削除します。

サブプロセス

loop.subprocess_exec()	サブプロセスを生成します。
loop.subprocess_shell()	シェルコマンドからサブプロセスを生成します。

エラー処理

loop.call_exception_handler()	例外ハンドラを呼び出します。
loop.set_exception_handler()	新しい例外ハンドラーを設定します。
loop.get_exception_handler()	現在の例外ハンドラーを取得します。
loop.default_exception_handler()	デフォルトの例外ハンドラー実装です。

使用例

- *Using asyncio.get_event_loop() and loop.run_forever()*.
- *loop.call_later()* を使う。
- `loop.create_connection()` を使って *an echo-client* を実装する。
- `loop.create_connection()` を使って **ソケットに接続する**。
- *add_reader()* を使ってファイルデスクリプタの読み込みイベントを監視する。
- *loop.add_signal_handler()* を使う。
- *loop.subprocess_exec()* を使う。

トランスポート

全てのトランスポートは以下のメソッドを実装します:

transport.close()	トランスポートをクローズします。
transport.is_closing()	トランスポートを閉じている最中か閉じていた場合 `True` を返します。
transport.get_extra_info()	トランスポートについての情報をリクエストします。
transport.set_protocol()	トランスポートに新しいプロトコルを設定します。
transport.get_protocol()	現在のプロトコルを返します。

データを受信できるトランスポート (TCP 接続、Unix 接続、パイプなど) のメソッドです。該当するトランスポートは *loop.create_connection()*, *loop.create_unix_connection()*, *loop.connect_read_pipe()* などの戻り値です:

読み込みトランスポート

transport.is_reading()	トランスポートがデータを受信中の場合 True を返します。
transport.pause_reading()	データの受信を停止します。
transport.resume_reading()	データの受信を再開します。

データを送信できるトランスポート (TCP 接続、Unix 接続、パイプなど) のメソッドです。該当するトランスポートは *loop.create_connection()*, *loop.create_unix_connection()*, *loop.connect_write_pipe()* などの戻り値です:

トランスポートにデータを書き込みます。

transport.write()	トランスポートにデータを書き込みます。
transport.writelines()	トランスポートにバッファの内容を書き込みます。
transport.can_write_eof()	トランスポートが 終端 (EOF) の送信をサポートしている場合 *True* を返します。
transport.write_eof()	バッファに残っているデータをフラッシュしてから終端 (EOF) を送信して、トランスポートをクローズします。
transport.abort()	トランスポートを即座にクローズします。
transport.get_write_buffer_size()	書き込みフロー制御の高水位点と低水位点を取得します。
transport.set_write_buffer_limits()	書き込みフロー制御の高水位点と低水位点を設定します。

loop.create_datagram_endpoint() が返すトランスポート:

データグラムトランスポート

transport.sendto()	リモートピアにデータを送信します。
transport.abort()	トランスポートを即座にクローズします。

サブプロセスに対するトランスポートの低レベルな抽象化です。*loop.subprocess_exec()* や *loop.subprocess_shell()* の戻り値です:

サブプロセス化されたトランスポート

transport.get_pid()	サブプロセスのプロセス ID を返します。
transport.get_pipe_transport()	リクエストされた通信パイプ (標準入力 *stdin*, 標準出力 *stdout*, または標準エラー出力 *stderr*) のためのトランスポートを返します。
transport.get_returncode()	サブプロセスの終了ステータスを返します。
transport.kill()	サブプロセスを強制終了 (kill) します。
transport.send_signal()	サブプロセスにシグナルを送信します。
transport.terminate()	サブプロセスを停止します。
transport.close()	サブプロセスを強制終了 (kill) し、全てのパイプをクローズします。

プロトコル

プロトコルクラスは以下の **コールバックメソッド** を実装することができます:

callback *connection_made()*	コネクションが作成されたときに呼び出されます。
callback *connection_lost()*	コネクションが失われた、あるいはクローズされたときに呼び出されます。
callback *pause_writing()*	トランスポートのバッファーサイズが最高水位点 (High-Water Mark) を超えたときに呼び出されます。
callback *resume_writing()*	トランスポートのバッファーサイズが最低水位点 (Low-Water Mark) に達したきに呼び出されます。

ストリーミングプロトコル (TCP, Unix ソケット, パイプ)

callback *data_received()*	データを受信したときに呼び出されます。
callback *eof_received()*	終端 (EOF) を受信したときに呼び出されます。

バッファリングされたストリーミングプロトコル

callback *get_buffer()*	新しい受信バッファを割り当てるために呼び出します。
callback *buffer_updated()*	受信データによりバッファが更新された場合に呼び出されます。
callback *eof_received()*	終端 (EOF) を受信したときに呼び出されます。

データグラムプロトコル

callback *datagram_received()*	データグラムを受信したときに呼び出されます。
callback *error_received()*	直前の送信あるいは受信が *OSError* を送出したときに呼び出されます。

サブプロセスプロトコル

callback *pipe_data_received()*	子プロセスが標準出力 (*stdout*) または標準エラー出力 (*stderr*) のパイプにデータを書き込んだときに呼び出されます。
callback *pipe_connection_lost()*	子プロセスと通信するパイプのいずれかがクローズされたときに呼び出されます。
callback *process_exited()*	子プロセスが終了したときに呼び出されます。

イベントループのポリシー

ポリシーは *asyncio.get_event_loop()* などの関数の振る舞いを変更する低レベルなメカニズムです。詳細は **ポリシーについてのセクション** を参照してください。

ポリシーへのアクセス

asyncio.get_event_loop_policy()	プロセス全体にわたる現在のポリシーを返します。
asyncio.set_event_loop_policy()	新たなプロセス全体にわたるポリシーを設定します。
AbstractEventLoopPolicy	ポリシーオブジェクトの基底クラスです。

18.1.14 asyncio での開発

非同期プログラミングは伝統的な ” 同期的” プログラミングとは異なります。

このページはよくある間違いや落とし穴を列挙し、それらを回避する方法を説明します。

デバッグモード

asyncio はデフォルトで本運用モードで実行されます。いっぽう、開発を容易にするために asyncio は ” デバッグモード” を持っています。

asyncio のデバッグモードを有効化する方法はいくつかあります:

- PYTHONASYNCIODEBUG 環境変数の値を 1 に設定する。
- Using the -X dev Python command line option.
- *asyncio.run()* 実行時に debug=True を設定する。
- *loop.set_debug()* を呼び出す。

デバッグモードを有効化することに加え、以下も検討してください:

- *asyncio* **ロガー** のログレベルを logging.DEBUG に設定します。例えばアプリケーションの起動時に以下を実行します:

```
logging.basicConfig(level=logging.DEBUG)
```

- *warnings* モジュールが *ResourceWarning* 警告を表示するように設定します。やり方のひとつは -W default コマンドラインオプションを使うことです。

デバッグモードが有効化されたときの動作:

- asyncio は **待ち受け処理** *(await)* **を伴わないコルーチン** がないかをチェックし、それらを記録します; これにより ” 待ち受け忘れ” の落とし穴にはまる可能性を軽減します。
- スレッドセーフでない asyncio APIs の多く (*loop.call_soon()* や *loop.call_at()* など) は、誤ったスレッドから呼び出されたときに例外を送出します。
- I/O セレクタが I/O 処理を実行する時間が長すぎる場合、その実行時間が記録されます。
- Callbacks taking longer than 100ms are logged. The loop.slow_callback_duration attribute can be used to set the minimum execution duration in seconds that is considered ”slow”.

並行処理とマルチスレッド処理

イベントループはスレッド（典型的にはメインスレッド）内で動作し、すべてのコールバックとタスクをそのスレッド内で実行します。ひとつのタスクがイベントループ内で実行される間、他のタスクを同じスレッド内で実行することはできません。タスクが await 式を実行すると、実行中のタスクはサスペンドされ、イベントループは次のタスクを実行します。

別の OS スレッドからのコールバック (*callback*) をスケジュールする場合、*loop.call_soon_threadsafe()* メソッドを使ってください。例:

```
loop.call_soon_threadsafe(callback, *args)
```

ほぼ全ての非同期オブジェクトはスレッドセーフではありませんが、タスクやコールバックの外側で非同期オブジェクトを使うコードが存在しない限り、それが問題にはなることはほとんどありません。も

しそのような目的で低レベルの asyncio API を呼び出すようなコードを書く必要がある場合、*loop.call_soon_threadsafe()* メソッドを使ってください。例:

```
loop.call_soon_threadsafe(fut.cancel)
```

別の OS スレッドからコルーチンオブジェクトをスケジュールする場合は、*run_coroutine_threadsafe()* メソッドを使ってください。*run_coroutine_threadsafe()* は結果にアクセスするための *concurrent.futures.Future* オブジェクトを返します:

```
async def coro_func():
    return await asyncio.sleep(1, 42)

# Later in another OS thread:

future = asyncio.run_coroutine_threadsafe(coro_func(), loop)
# Wait for the result:
result = future.result()
```

シグナルの処理やサブプロセスの実行を行うには、イベントループはメインスレッド内で実行しなければなりません。

The *loop.run_in_executor()* メソッドを *concurrent.futures.ThreadPoolExecutor* とともに使用することで、イベントループの OS スレッドをブロックすることなく、別の OS スレッド内でブロッキングコードを実行することができます。

There is currently no way to schedule coroutines or callbacks directly from a different process (such as one started with *multiprocessing*). The *Event Loop Methods* section lists APIs that can read from pipes and watch file descriptors without blocking the event loop. In addition, asyncio's *Subprocess* APIs provide a way to start a process and communicate with it from the event loop. Lastly, the aforementioned *loop.run_in_executor()* method can also be used with a *concurrent.futures.ProcessPoolExecutor* to execute code in a different process.

ブロッキングコードの実行

ブロッキングコード (CPU バウンドなコード) を直接呼び出すべきではありません。たとえば、CPU 負荷の高い関数を 1 秒実行したとすると、並行に処理されている全ての非同期タスクと I/O 処理は 1 秒遅れる可能性があります。

エグゼキューターを使用することにより、イベントループの OS スレッドをブロックすることなく、別のスレッドや別のプロセス上でタスクを実行することができます。詳しくは *loop.run_in_executor()* メソッドを参照してください。

ログ記録

asyncio は *logging* モジュールを利用し、全てのログ記録は "asyncio" ロガーを通じて行われます。

デフォルトのログレベルは logging.INFO ですが、これは簡単に調節できます:

```
logging.getLogger("asyncio").setLevel(logging.WARNING)
```

待ち受け処理を伴わないコルーチンの検出

コルーチンが呼び出されただけで、待ち受け処理がない場合 (たとえば await coro() のかわりに coro() と書いてしまった場合) 、またはコルーチンが *asyncio.create_task()* を使わずにスケジュールされた場合、asyncio は *RuntimeWarning* 警告を送出します:

```
import asyncio

async def test():
    print("never scheduled")

async def main():
    test()

asyncio.run(main())
```

出力:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
  test()
```

デバッグモードの出力:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
Coroutine created at (most recent call last)
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

  < .. >

  File "../t.py", line 7, in main
    test()
  test()
```

通常の修正方法はコルーチンを待ち受ける (await) か、*asyncio.create_task()* 関数を呼び出すことです:

```
async def main():
    await test()
```

回収されない例外の検出

もし *Future.set_exception()* メソッドが呼び出されても、その Future オブジェクトを待ち受けていなければ、例外は決してユーザーコードまで伝播しません。この場合 asyncio は、Future オブジェクトがガベージコレクションの対象となったときにログメッセージを送出することがあります。

処理されない例外の例:

```
import asyncio

async def bug():
    raise Exception("not consumed")

async def main():
    asyncio.create_task(bug())

asyncio.run(main())
```

出力:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
  exception=Exception('not consumed')>

Traceback (most recent call last):
  File "test.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

タスクが生成された箇所を特定するには、**デバッグモードを有効化して** トレースバックを取得してください:

```
asyncio.run(main(), debug=True)
```

デバッグモードの出力:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
    exception=Exception('not consumed') created at asyncio/tasks.py:321>

source_traceback: Object created at (most recent call last):
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

< .. >

Traceback (most recent call last):
  File "../t.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

注釈: asyncio のソースコードは Lib/asyncio/ にあります。

18.2 socket --- 低水準ネットワークインターフェース

ソースコード: Lib/socket.py

このモジュールは BSD の **ソケット** *(socket)* インターフェイスへのアクセスを提供します。これは、近代的な Unix システム、Windows、MacOS、その他多くのプラットフォームで動作します。

注釈: いくつかの挙動はプラットフォームに依存します。オペレーティングシステムのソケット API を呼び出しているためです。

Python インターフェースは、Unix のソケット用システムコールとライブラリインターフェースを、そのまま Python のオブジェクト指向スタイルに変換したものです。各種ソケット関連のシステムコールは、*socket()* 関数で生成される *socket* **オブジェクト** のメソッドとして実装されています。メソッドの引数は C のインターフェイスよりも多少高水準で、例えばファイルに対する `read()` や `write()` メソッドと同様に、受信時のバッファ確保は自動的に処理され、送信時のバッファ長は暗黙的に決まります。

参考:

Module *socketserver* ネットワークサーバの開発を省力化するためのクラス群。

Module *ssl* ソケットオブジェクトに対する TLS/SSL ラッパー.

18.2.1 ソケットファミリー

どのシステムで実行するかとビルドオプションに依存しますが、このモジュールによって多様なソケットファミリーをサポートします。

特定のソケットオブジェクトによって必要とされるアドレスフォーマットは、ソケットオブジェクトが生成されたときに指定されたアドレスファミリーを元に自動的に選択されます。ソケットアドレスは次の通りです。

- ファイルシステム上のノードに束縛された *AF_UNIX* ソケットのアドレスは、ファイルシステムエンコーディングと `'surrogateescape'` エラーハンドラ (**PEP 383** を参照) を使って文字列として表現されます。Linux の抽象名前空間のアドレスは、先頭が null バイトとなる *bytes-like object* として返されます。この名前空間のソケットは通常のファイルシステム上のソケットと通信できるので、Linux 上で動作することを意図したプログラムは両方のアドレスを扱う必要がある可能性があります。文字列と bytes-like オブジェクトはどちらのタイプのアドレスにも引数として渡すことができます。

  バージョン 3.3 で変更: これまでは *AF_UNIX* ソケットパスは UTF-8 エンコーディングを使用するものとされていました。

バージョン 3.5 で変更: 書き込み可能な *bytes-like object* を使用できるようになりました。

- *AF_INET* アドレスファミリーには、(host, port) ペアがアドレスとして利用されます。*host* はホスト名か 'daring.cwi.nl' のようなインターネットドメインか、'100.50.200.5' のような IPv4 アドレスで、*port* は整数です。

  – IPv4 ではホストアドレスのほかに 2 つの特別な形式が使用できます。''``はすべてのインターフェイスにバインドされるために使われる:const:`INADDR_ANY`を表し、'<broadcast>'"は:const:'INADDR_BROADCAST'を表します。これらの動作は IPv6 と互換性がありません。そのためもしもあなたが Python プログラムで IPv6 をサポートする予定があるのならばこれらを避けたほうが良いでしょう。

- *AF_INET6* アドレスファミリーでは、(host, port, flowinfo, scopeid) の 4 要素のタプルが利用されます。*flowinfo* と *scopeid* はそれぞれ C 言語の struct sockaddr_in6 の sin6_flowinfo と sin6_scope_id メンバーを表します。*socket* モジュールのメソッドでは、後方互換性のために *flowinfo* と *scopeid* の省略を許しています。しかし、*scopeid* を省略すると scope された IPv6 アドレスを操作するときに問題が起こる場合があることに注意してください。

  バージョン 3.7 で変更: For multicast addresses (with *scopeid* meaningful) *address* may not contain %scope (or zone id) part. This information is superfluous and may be safely omitted (recommended).

- AF_NETLINK ソケットのアドレスは (pid, groups) のペアで表されます。

- Linux 限定で、AF_TIPC アドレスファミリーを用いて TIPC がサポートされます。TIPC は、クラスタコンピューティング環境のために設計された、IP ベースではないオープンなネットワークプロトコルです。アドレスはタプルで表現され、フィールドはアドレスタイプに依存します。一般的なタプルの形式は (addr_type, v1, v2, v3 [, scope]) で、それぞれは次の通りです:

  – *addr_type* は TIPC_ADDR_NAMESEQ, TIPC_ADDR_NAME, TIPC_ADDR_ID の 1 つ。

  – *scope* は TIPC_ZONE_SCOPE, TIPC_CLUSTER_SCOPE, TIPC_NODE_SCOPE の 1 つ。

  – *addr_type* が TIPC_ADDR_NAME の場合、*v1* はサーバータイプ、*v2* はポート ID (the port identifier)、そして *v3* は 0 であるべきです。

    addr_type が TIPC_ADDR_NAMESEQ の場合、*v1* はサーバータイプ、*v2* はポート番号下位 (lower port number)、*v3* はポート番号上位 (upper port number) です。

    addr_type が TIPC_ADDR_ID の場合、*v1* はノード、*v2* は参照、*v3* は 0 であるべきです。

- *AF_CAN* アドレスファミリーには (interface,) というタプルを利用します。*interface* は 'can0' のようなネットワークインタフェース名を表す文字列です。このファミリーの全てのネットワークインタフェースからパケットを受信するために、ネットワークインタフェース名 '' を利用できます。

  – *CAN_ISOTP* protocol require a tuple (interface, rx_addr, tx_addr) where both additional parameters are unsigned long integer that represent a CAN identifier (standard or extended).

- 文字列またはタプル (id, unit) は PF_SYSTEM ファミリーの SYSPROTO_CONTROL プロトコルのために使用されます。この文字列は、動的に割り当てられた ID によるカーネルコントロールの名前です。

このタプルは、カーネルコントロールの ID とユニット番号が既知の場合、または登録済み ID が使用中の場合に使用することができます。

バージョン 3.3 で追加.

- AF_BLUETOOTH は以下のプロトコルとアドレスフォーマットをサポートしています。
  - BTPROTO_L2CAP は (bdaddr, psm) を受け取ります。bdaddr は Bluetooth アドレスを表す文字列で、psm は整数です。
  - BTPROTO_RFCOMM は (bdaddr, channel) を受け取ります。bdaddr は Bluetooth アドレスを表す文字列で、channel は整数です。
  - BTPROTO_HCI は (device_id,) を受け取ります。device_id は、数値またはインターフェイスの Bluetooth アドレスを表す文字列です。(OS に依存します。NetBSD と DragonFlyBSD は Bluetooth アドレスを期待しますが、その他すべての OS は、数値を期待します。)

    バージョン 3.2 で変更: NetBSD と DragonFlyBSD のサポートが追加されました。
  - BTPROTO_SCO は bdaddr を受け取ります。ここで、bdaddr は Bluetooth アドレスを文字列形式で持つ *bytes* オブジェクトです (例: b'12:23:34:45:56:67')。このプロトコルは、FreeBSD ではサポートされていません。
- *AF_ALG* はカーネル暗号へのソケットベースのインターフェイスで、Linux でのみ使用できます。アルゴリズムソケットは、2 つから 4 つの要素を持つタプル (type, name [, feat [, mask]]) で構成されます。各要素の意味は、以下の通りです。
  - *type* はアルゴリズムタイプを示す文字列です。例: aead, hash, skcipher または rng。
  - *name* はアルゴリズム名及び操作モードを示す文字列です。例: sha256, hmac(sha256), cbc(aes) または drbg_nopr_ctr_aes256。
  - *feat* と *mask* は、符号を持たない 32 ビットの整数です。

  Availability: Linux 2.6.38, some algorithm types require more recent Kernels.

  バージョン 3.6 で追加.
- *AF_VSOCK* allows communication between virtual machines and their hosts. The sockets are represented as a (CID, port) tuple where the context ID or CID and port are integers.

  Availability: Linux >= 4.8 QEMU >= 2.8 ESX >= 4.0 ESX Workstation >= 6.5.

  バージョン 3.7 で追加.
- *AF_PACKET* is a low-level interface directly to network devices. The packets are represented by the tuple (ifname, proto[, pkttype[, hatype[, addr]]]) where:
  - *ifname* - デバイス名を指定する文字列。
  - *proto* - An in network-byte-order integer specifying the Ethernet protocol number.
  - *pkttype* - パケットタイプを指定するオプションの整数:

        * PACKET_HOST (the default) - Packet addressed to the local host.
        * PACKET_BROADCAST - Physical-layer broadcast packet.
        * PACKET_MULTIHOST - Packet sent to a physical-layer multicast address.
        * PACKET_OTHERHOST - Packet to some other host that has been caught by a device driver in promiscuous mode.
        * PACKET_OUTGOING - Packet originating from the local host that is looped back to a packet socket.
      - *hatype* - Optional integer specifying the ARP hardware address type.
      - *addr* - Optional bytes-like object specifying the hardware physical address, whose interpretation depends on the device.
- *AF_QIPCRTR* is a Linux-only socket based interface for communicating with services running on co-processors in Qualcomm platforms. The address family is represented as a `(node, port)` tuple where the *node* and *port* are non-negative integers.

  バージョン 3.8 で追加.

IPv4/v6 ソケットの *host* 部にホスト名を指定すると、処理結果が一定ではない場合があります。これは Python は DNS から取得したアドレスのうち最初のアドレスを使用するので、DNS の処理やホストの設定によって異なる IPv4/6 アドレスを取得する場合があるためです。常に同じ結果が必要であれば、*host* に数値のアドレスを指定してください。

全てのエラーは例外を発生させます。引数型のエラーやメモリ不足の場合には通常の例外が発生し、ソケットやアドレス関連のエラーは Python 3.3 からは *OSError* かそのサブクラスを発生させます (Python 3.3 以前は *socket.error* を発生させていました)。

setblocking() メソッドで、非ブロッキングモードを使用することができます。また、より汎用的に *settimeout()* メソッドでタイムアウトを指定する事ができます。

18.2.2 モジュールの内容

socket モジュールは以下の要素を公開しています。

例外

exception socket.error

OSError の非推奨のエイリアスです。

バージョン 3.3 で変更: **PEP 3151** に基づき、このクラスは *OSError* のエイリアスになりました。

exception socket.herror

OSError のサブクラス。この例外はアドレス関連のエラー、つまり *gethostbyname_ex()* と *gethostbyaddr()* などの、POSIX C API の *h_errno* を利用する関数のために利用されます。

例外に付随する (h_errno, string) ペアはライブラリの呼び出しによって返されたエラーを表します。*h_errno* は数値で、*string* は、hstrerror() C 関数によって返される *h_errno* を説明する文字列です。

バージョン 3.3 で変更: このクラスは *OSError* のサブクラスになりました。

exception socket.gaierror

OSError のサブクラスです。この例外は *getaddrinfo()* と *getnameinfo()* でアドレス関連のエラーが発生した場合に送出されます。例外の値は (error, string) のペアで、ライブラリの呼び出し結果を返します。*string* は C 関数 gai_strerror() で取得した、*error* の意味を示す文字列です。*error* の値は、このモジュールで定義される EAI_* 定数のどれかとなります。

バージョン 3.3 で変更: このクラスは *OSError* のサブクラスになりました。

exception socket.timeout

OSError のサブクラスです。この例外は、あらかじめ *settimeout()* を呼び出して (あるいは *setdefaulttimeout()* を利用して暗黙に) タイムアウトを有効にしてあるソケットでタイムアウトが生じた際に送出されます。例外に付属する値は文字列で、その内容は現状では常に "timed out" となります。

バージョン 3.3 で変更: このクラスは *OSError* のサブクラスになりました。

定数

AF_* 定数と SOCK_* 定数は、AddressFamily と SocketKind *IntEnum* collection になりました。

バージョン 3.4 で追加.

socket.AF_UNIX
socket.AF_INET
socket.AF_INET6

アドレス (およびプロトコル) ファミリーを示す定数で、*socket()* の 最初の引数に指定することができます。*AF_UNIX* ファミリーをサポート しないプラットフォームでは、*AF_UNIX* は未定義となります。システムによってはこれら以外の定数が定義されているかもしれません。

socket.SOCK_STREAM
socket.SOCK_DGRAM
socket.SOCK_RAW
socket.SOCK_RDM
socket.SOCK_SEQPACKET

ソケットタイプを示す定数で、*socket()* の 2 番目の引数に指定することができます。システムによってはこれら以外の定数が定義されているかもしれません。(ほとんどの場合、*SOCK_STREAM* と *SOCK_DGRAM* 以外は必要ありません。)

socket.SOCK_CLOEXEC
socket.SOCK_NONBLOCK

この 2 つの定数が定義されていた場合、ソケットタイプと組み合わせていくつかの flags をアトミックに設定することができます (別の呼び出しを不要にして競合状態を避ける事ができます)。

参考:

より完全な説明は Secure File Descriptor Handling を参照してください。

利用可能な環境: Linux 2.6.27 以上。

バージョン 3.2 で追加.

SO_*
socket.**SOMAXCONN**
MSG_*
SOL_*
SCM_*
IPPROTO_*
IPPORT_*
INADDR_*
IP_*
IPV6_*
EAI_*
AI_*
NI_*
TCP_*

Unix のソケット・IP プロトコルのドキュメントで定義されている各種定数。ソケットオブジェクトの `setsockopt()` や `getsockopt()` で使用します。ほとんどのシンボルは Unix のヘッダファイルに従っています。一部のシンボルには、デフォルト値を定義してあります。

バージョン 3.6 で変更: `SO_DOMAIN`, `SO_PROTOCOL`, `SO_PEERSEC`, `SO_PASSSEC`, `TCP_USER_TIMEOUT`, `TCP_CONGESTION` が追加されました。

バージョン 3.6.5 で変更: Windows では、実行時の Windows がサポートしているならば `TCP_FASTOPEN`、“TCP_KEEPCNT“が表示されます。

バージョン 3.7 で変更: “TCP_NOTSENT_LOWAT“が追加されました。

Windows では、実行時の Windows がサポートしているならば `TCP_KEEPIDLE`, `TCP_KEEPINTVL` が表示されます。

socket.AF_CAN
socket.PF_CAN
SOL_CAN_*
CAN_*

Linux ドキュメントにあるこの形式の定数は socket モジュールでも定義されています。

利用可能な環境: Linux 2.6.25 以上。

バージョン 3.3 で追加.

socket.CAN_BCM
CAN_BCM_*

CAN プロトコルファミリーの CAN_BCM は、ブロードキャストマネージャー (BCM) プロトコルです。Linux ドキュメントにあるこの形式の定数は、socket モジュールでも定義されています。

利用可能な環境: Linux 2.6.25 以上。

注釈: The `CAN_BCM_CAN_FD_FRAME` flag is only available on Linux >= 4.8.

バージョン 3.4 で追加.

socket.CAN_RAW_FD_FRAMES

CAN_RAW ソケットで CAN FD をサポートします。これはデフォルトで無効になっています。これにより、アプリケーションが CAN フレームと CAN FD フレームを送信できるようになります。ただし、ソケットからの読み出し時に、CAN と CAN FD の両方のフレームを受け入れなければなりません。

この定数は、Linux のドキュメンテーションで説明されています。

利用可能な環境: Linux 3.6 以上。

バージョン 3.5 で追加.

socket.CAN_ISOTP

CAN_ISOTP, in the CAN protocol family, is the ISO-TP (ISO 15765-2) protocol. ISO-TP constants, documented in the Linux documentation.

利用可能な環境: Linux 2.6.25 以上。

バージョン 3.7 で追加.

socket.AF_PACKET

socket.PF_PACKET

PACKET_*

Linux ドキュメントにあるこの形式の定数は socket モジュールでも定義されています。

利用可能な環境: Linux 2.2 以上。

socket.AF_RDS

socket.PF_RDS

socket.SOL_RDS

RDS_*

Linux ドキュメントにあるこの形式の定数は socket モジュールでも定義されています。

利用可能な環境: Linux 2.6.30 以上。

バージョン 3.3 で追加.

socket.SIO_RCVALL

socket.SIO_KEEPALIVE_VALS

socket.SIO_LOOPBACK_FAST_PATH

RCVALL_*

Windows の WSAIoctl() のための定数です。この定数はソケットオブジェクトの *ioctl()* メソッドに引数として渡されます。

バージョン 3.6 で変更: SIO_LOOPBACK_FAST_PATH が追加されました。

TIPC_*

TIPC 関連の定数で、C のソケット API が公開しているものにマッチします。詳しい情報は TIPC のドキュメントを参照してください。

socket.AF_ALG

socket.SOL_ALG

ALG_*

Linux カーネル暗号用の定数です。

利用可能な環境: Linux 2.6.38 以上。

バージョン 3.6 で追加.

socket.AF_VSOCK

socket.IOCTL_VM_SOCKETS_GET_LOCAL_CID

VMADDR*

SO_VM*

Constants for Linux host/guest communication.

利用可能な環境: Linux 4.8 以上。

バージョン 3.7 で追加.

socket.AF_LINK

利用可能な環境: BSD, OSX。

バージョン 3.4 で追加.

socket.has_ipv6

現在のプラットフォームで IPv6 がサポートされているか否かを示す真偽値。

socket.BDADDR_ANY

socket.BDADDR_LOCAL

これらは、特別な意味を持つ Bluetooth アドレスを含む文字列定数です。例えば、*BDADDR_ANY* を使用すると、BTPROTO_RFCOMM で束縛ソケットを指定する際に、任意のアドレスを指し示すことができます。

socket.HCI_FILTER

socket.HCI_TIME_STAMP

socket.HCI_DATA_DIR

BTPROTO_HCI で使用します。*HCI_FILTER* は NetBSD または DragonFlyBSD では使用できません。*HCI_TIME_STAMP* と *HCI_DATA_DIR* は FreeBSD, NetBSD, DragonFlyBSD では使用できません。

socket.AF_QIPCRTR

Constant for Qualcomm's IPC router protocol, used to communicate with service providing remote processors.

利用可能な環境: Linux 4.7 以上。

関数

ソケットの作成

以下の関数は全て *socket object* を生成します。

socket.socket(*family=AF_INET*, *type=SOCK_STREAM*, *proto=0*, *fileno=None*)

Create a new socket using the given address family, socket type and protocol number. The address family should be *AF_INET* (the default), *AF_INET6*, *AF_UNIX*, *AF_CAN*, *AF_PACKET*, or *AF_RDS*. The socket type should be *SOCK_STREAM* (the default), *SOCK_DGRAM*, *SOCK_RAW* or perhaps one of the other `SOCK_` constants. The protocol number is usually zero and may be omitted or in the case where the address family is *AF_CAN* the protocol should be one of `CAN_RAW`, *CAN_BCM* or *CAN_ISOTP*.

If *fileno* is specified, the values for *family*, *type*, and *proto* are auto-detected from the specified file descriptor. Auto-detection can be overruled by calling the function with explicit *family*, *type*, or *proto* arguments. This only affects how Python represents e.g. the return value of *socket.getpeername()* but not the actual OS resource. Unlike *socket.fromfd()*, *fileno* will return the same socket and not a duplicate. This may help close a detached socket using *socket.close()*.

新たに作成されたソケットは 継承不可 です。

引数 `self`, `family`, `type`, `protocol` 付きで 監査イベント `socket.__new__` を送出します。

バージョン 3.3 で変更: AF_CAN, AF_RDS ファミリーが追加されました。

バージョン 3.4 で変更: CAN_BCM プロトコルが追加されました。

バージョン 3.4 で変更: 返されるソケットは継承不可になりました。

バージョン 3.7 で変更: CAN_ISOTP プロトコルが追加されました。

バージョン 3.7 で変更: When *SOCK_NONBLOCK* or *SOCK_CLOEXEC* bit flags are applied to *type* they are cleared, and *socket.type* will not reflect them. They are still passed to the underlying system *socket()* call. Therefore,

```
sock = socket.socket(
    socket.AF_INET,
    socket.SOCK_STREAM | socket.SOCK_NONBLOCK)
```

will still create a non-blocking socket on OSes that support `SOCK_NONBLOCK`, but `sock.type` will be set to `socket.SOCK_STREAM`.

socket.socketpair([*family*[, *type*[, *proto*]]])

指定されたアドレスファミリー、ソケットタイプ、プロトコル番号から、接続されたソケットオブジェクトのペアを作成します。アドレスファミリー、ソケットタイプ、プロトコル番号は *socket()* 関

数と同様に指定します。デフォルトのアドレスファミリは、プラットフォームで定義されている場合 *AF_UNIX* 、そうでなければ *AF_INET* が使われます。

新たに作成されたソケットは **継承不可** です。

バージョン 3.2 で変更: 返されるソケットオブジェクトが、サブセットではなく完全なソケット API を提供するようになりました。

バージョン 3.4 で変更: 返されるソケットの組は、どちらも継承不可になりました。

バージョン 3.5 で変更: Windows のサポートが追加されました。

socket.create_connection(*address*[, *timeout*[, *source_address*]])

address (`(host, port)` ペア) で listen している TCP サービスに接続し、ソケットオブジェクトを返します。これは *socket.connect()* を高級にした関数です。*host* が数値でないホスト名の場合、*AF_INET* と *AF_INET6* の両方で名前解決を試み、得られた全てのアドレスに対して成功するまで接続を試みます。この関数を使って IPv4 と IPv6 に両対応したクライアントを簡単に書くことができます。

オプションの *timeout* 引数を指定すると、接続を試みる前にソケットオブジェクトのタイムアウトを設定します。*timeout* が指定されない場合、*getdefaulttimeout()* が返すデフォルトのタイムアウト設定値を利用します。

source_address は接続する前にバインドするソースアドレスを指定するオプション引数で、指定する場合は `(host, port)` の 2 要素タプルでなければなりません。host や port が '' か 0 だった場合は、OS のデフォルトの動作になります。

バージョン 3.2 で変更: *source_address* が追加されました。

socket.create_server(*address*, *, *family=AF_INET*, *backlog=None*, *reuse_port=False*, *dualstack_ipv6=False*)

Convenience function which creates a TCP socket bound to *address* (a 2-tuple `(host, port)`) and return the socket object.

family should be either *AF_INET* or *AF_INET6*. *backlog* is the queue size passed to *socket.listen()*; when `0` a default reasonable value is chosen. *reuse_port* dictates whether to set the `SO_REUSEPORT` socket option.

If *dualstack_ipv6* is true and the platform supports it the socket will be able to accept both IPv4 and IPv6 connections, else it will raise *ValueError*. Most POSIX platforms and Windows are supposed to support this functionality. When this functionality is enabled the address returned by *socket.getpeername()* when an IPv4 connection occurs will be an IPv6 address represented as an IPv4-mapped IPv6 address. If *dualstack_ipv6* is false it will explicitly disable this functionality on platforms that enable it by default (e.g. Linux). This parameter can be used in conjunction with *has_dualstack_ipv6()*:

```
import socket

addr = ("", 8080)  # all interfaces, port 8080
if socket.has_dualstack_ipv6():
    s = socket.create_server(addr, family=socket.AF_INET6, dualstack_ipv6=True)
```

(次のページに続く)

(前のページからの続き)

```
else:
    s = socket.create_server(addr)
```

注釈: On POSIX platforms the SO_REUSEADDR socket option is set in order to immediately reuse previous sockets which were bound on the same *address* and remained in TIME_WAIT state.

バージョン 3.8 で追加.

socket.**has_dualstack_ipv6**()

Return True if the platform supports creating a TCP socket which can handle both IPv4 and IPv6 connections.

バージョン 3.8 で追加.

socket.**fromfd**(*fd*, *family*, *type*, *proto=0*)

ファイル記述子 (ファイルオブジェクトの fileno() メソッドが返す整数) *fd* を複製して、ソケットオブジェクトを構築します。アドレスファミリとプロトコル番号は *socket()* と同様に指定します。ファイル記述子 はソケットを指していなければなりませんが、実際にソケットであるかどうかのチェックは行っていません。このため、ソケット以外のファイル記述子 を指定するとその後の処理が失敗する場合があります。この関数が必要な事はあまりありませんが、(Unix の inet デーモンに起動されるプログラムのように) ソケットを標準入力や標準出力として使用するプログラムでソケットオプションの取得や設定を行うために使われます。この関数で使用するソケットは、ブロッキングモードと想定しています。

新たに作成されたソケットは **継承不可** です。

バージョン 3.4 で変更: 返されるソケットは継承不可になりました。

socket.**fromshare**(*data*)

socket.share() メソッドから取得した data からソケットオブジェクトを生成します。ソケットはブロッキングモードだと仮定されます。

利用可能な環境: Windows 。

バージョン 3.3 で追加.

socket.**SocketType**

ソケットオブジェクトの型を示す型オブジェクト。type(socket(...)) と同じです。

その他の関数

socket モジュールはネットワーク関連のサービスを提供しています:

socket.close(*fd*)

Close a socket file descriptor. This is like *os.close()*, but for sockets. On some platforms (most noticeable Windows) *os.close()* does not work for socket file descriptors.

バージョン 3.7 で追加.

socket.getaddrinfo(*host*, *port*, *family=0*, *type=0*, *proto=0*, *flags=0*)

host / *port* 引数の指すアドレス情報を、そのサービスに接続されたソケットを作成するために必要な全ての引数が入った 5 要素のタプルに変換します。*host* はドメイン名、IPv4/v6 アドレスの文字列、または None です。*port* は 'http' のようなサービス名文字列、ポート番号を表す数値、または None です。*host* と *port* に None を指定すると C API に NULL を渡せます。

オプションの *family*, *type*, *proto* 引数を指定すると、返されるアドレスのリストを絞り込むことができます。これらの引数の値として 0 を渡すと絞り込まない結果を返します。*flags* 引数には AI_* 定数のうち 1 つ以上が指定でき、結果の取り方を変えることができます。例えば、AI_NUMERICHOST を指定するとドメイン名解決を行わないようにし、*host* がドメイン名だった場合には例外を送出します。

この関数は以下の構造をとる 5 要素のタプルのリストを返します:

(family, type, proto, canonname, sockaddr)

このタプルにある *family*, *type*, *proto* は、*socket()* 関数を呼び出す際に指定する値と同じ整数です。AI_CANONNAME を含んだ *flags* を指定した場合、*canonname* は *host* の canonical name を示す文字列です。そうでない場合は *canonname* は空文字列です。*sockaddr* は、ソケットアドレスを *family* に依存した形式で表すタプルで、(*AF_INET* の場合は 2 要素のタプル (address, port)、*AF_INET6* の場合は 4 要素のタプル (address, port, flow info, scope id)) *socket.connect()* メソッドに渡すためのものです。

引数 host, port, family, type, protocol 付きで **監査イベント** socket.getaddrinfo を送出します。

次の例では example.org の 80 番ポートポートへの TCP 接続を得るためのアドレス情報を取得しようとしています。(結果は IPv6 をサポートしているかどうかで変わります):

```
>>> socket.getaddrinfo("example.org", 80, proto=socket.IPPROTO_TCP)
[(<AddressFamily.AF_INET6: 10>, <SocketType.SOCK_STREAM: 1>,
 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (<AddressFamily.AF_INET: 2>, <SocketType.SOCK_STREAM: 1>,
 6, '', ('93.184.216.34', 80))]
```

バージョン 3.2 で変更: パラメータをキーワード引数で渡すことができるようになりました。

バージョン 3.7 で変更: for IPv6 multicast addresses, string representing an address will not contain %scope part.

socket.**getfqdn**([*name*])

name の完全修飾ドメイン名を返します。*name* が空または省略された場合、ローカルホストを指定したとみなします。完全修飾ドメイン名の取得にはまず *gethostbyaddr()* でチェックし、次に可能であればエイリアスを調べ、名前にピリオドを含む最初の名前を値として返します。完全修飾ドメイン名を取得できない場合、*gethostname()* で返されるホスト名を返します。

socket.**gethostbyname**(*hostname*)

ホスト名を `'100.50.200.5'` のような IPv4 形式のアドレスに変換します。ホスト名として IPv4 アドレスを指定した場合、その値は変換せずにそのまま返ります。*gethostbyname()* API へのより完全なインターフェイスが必要であれば、*gethostbyname_ex()* を参照してください。*gethostbyname()* は、IPv6 名前解決をサポートしていません。IPv4/ v6 のデュアルスタックをサポートする場合は *getaddrinfo()* を使用します。

引数 `hostname` を指定して **監査イベント** `socket.gethostbyname` を送出します。

socket.**gethostbyname_ex**(*hostname*)

ホスト名から、IPv4 形式の各種アドレス情報を取得します。戻り値は `(hostname, aliaslist, ipaddrlist)` のタプルで、*hostname* は *ip_address* で指定したホストの正式名、*aliaslist* は同じアドレスの別名のリスト (空の場合もある)、*ipaddrlist* は同じホスト上の同一インターフェイスの IPv4 アドレスのリスト (ほとんどの場合は単一のアドレスのみ) を示します。*gethostbyname_ex()* は、IPv6 名前解決をサポートしていません。IPv4/v6 のデュアルスタックをサポートする場合は *getaddrinfo()* を使用します。

引数 `hostname` を指定して **監査イベント** `socket.gethostbyname` を送出します。

socket.**gethostname**()

Python インタープリタを現在実行しているマシンのホスト名を含む文字列を返します。

引数無しで **監査イベント** `socket.gethostname` を送出します。

注意: *gethostname()* は完全修飾ドメイン名を返すとは限りません。完全修飾ドメイン名が必要であれば、*getfqdn()* を使用してください。

socket.**gethostbyaddr**(*ip_address*)

`(hostname, aliaslist, ipaddrlist)` のタプルを返し、*hostname* は *ip_address* で指定したホストの正式名、*aliaslist* は同じアドレスの別名のリスト (空の場合もある)、*ipaddrlist* は同じホスト上の同一インターフェイスの IPv4 アドレスのリスト (ほとんどの場合は単一のアドレスのみ) を示します。完全修飾ドメイン名が必要であれば、*getfqdn()* を使用してください。*gethostbyaddr()* は、IPv4/IPv6 の両方をサポートしています。

引数 `ip_address` を指定して **監査イベント** `socket.gethostbyaddr` を送出します。

socket.**getnameinfo**(*sockaddr*, *flags*)

ソケットアドレス *sockaddr* から、`(host, port)` のタプルを取得します。*flags* の設定に従い、*host* は完全修飾ドメイン名または数値形式アドレスとなります。同様に、*port* は文字列のポート名または数値のポート番号となります。

For IPv6 addresses, `%scope` is appended to the host part if *sockaddr* contains meaningful *scopeid*. Usually this happens for multicast addresses.

For more information about *flags* you can consult *getnameinfo(3)*.

引数 sockaddr を指定して 監査イベント socket.getnameinfo を送出します。

socket.**getprotobyname**(*protocolname*)

('icmp' のような) インターネットプロトコル名を、*socket()* の 第三引数として指定する事ができる定数に変換します。これは主にソケットを "raw" モード (*SOCK_RAW*) でオープンする場合には必要ですが、通常の ソケットモードでは第三引数に 0 を指定するか省略すれば正しいプロトコルが自動的に選択されます。

socket.**getservbyname**(*servicename*[, *protocolname*])

インターネットサービス名とプロトコルから、そのサービスのポート番号を取得します。省略可能なプロトコル名として、'tcp' か 'udp' のどちらかを指定することができます。指定がなければどちらのプロトコルにもマッチします。

引数 servicename, protocolname を指定して 監査イベント socket.getservbyname を送出します。

socket.**getservbyport**(*port*[, *protocolname*])

インターネットポート番号とプロトコル名から、サービス名を取得します。省略可能なプロトコル名として、'tcp' か 'udp' のどちらかを指定することができます。指定がなければどちらのプロトコルにもマッチします。

引数 port, protocolname を指定して 監査イベント socket.getservbyport を送出します。

socket.**ntohl**(*x*)

32 ビットの正の整数のバイトオーダを、ネットワークバイトオーダからホストバイトオーダに変換します。ホストバイトオーダとネットワークバイトオーダが一致するマシンでは、この関数は何もしません。それ以外の場合は 4 バイトのスワップを行います。

socket.**ntohs**(*x*)

16 ビットの正の整数のバイトオーダを、ネットワークバイトオーダからホストバイトオーダに変換します。ホストバイトオーダとネットワークバイトオーダが一致するマシンでは、この関数は何もしません。それ以外の場合は 2 バイトのスワップを行います。

バージョン 3.7 で非推奨: In case *x* does not fit in 16-bit unsigned integer, but does fit in a positive C int, it is silently truncated to 16-bit unsigned integer. This silent truncation feature is deprecated, and will raise an exception in future versions of Python.

socket.**htonl**(*x*)

32 ビットの正の整数のバイトオーダを、ホストバイトオーダからネットワークバイトオーダに変換します。ホストバイトオーダとネットワークバイトオーダが一致するマシンでは、この関数は何もしません。それ以外の場合は 4 バイトのスワップを行います。

socket.**htons**(*x*)

16 ビットの正の整数のバイトオーダを、ホストバイトオーダからネットワークバイトオーダに変換します。ホストバイトオーダとネットワークバイトオーダが一致するマシンでは、この関数は何もしません。それ以外の場合は 2 バイトのスワップを行います。

バージョン 3.7 で非推奨: In case *x* does not fit in 16-bit unsigned integer, but does fit in a positive C

int, it is silently truncated to 16-bit unsigned integer. This silent truncation feature is deprecated, and will raise an exception in future versions of Python.

socket.inet_aton(*ip_string*)

ドット記法による IPv4 アドレス (`'123.45.67.89'` など) を 32 ビットにパックしたバイナリ形式に変換し、長さ 4 のバイト列オブジェクトとして返します。この関数が返す値は、標準 C ライブラリの `struct in_addr` 型を使用する関数に渡す事ができます。

inet_aton() はドットが 3 個以下の文字列も受け取ります; 詳細については Unix のマニュアル *inet(3)* を参照してください。

IPv4 アドレス文字列が不正であれば、*OSError* が発生します。このチェックは、この関数で使用している C の実装 `inet_aton()` で行われます。

inet_aton() は、IPv6 をサポートしません。IPv4/v6 のデュアルスタックをサポートする場合は *inet_pton()* を使用します。

socket.inet_ntoa(*packed_ip*)

32 ビットにパックされた IPv4 アドレス (長さ 4 バイトの *bytes-like object*) を、標準的なドット記法による 4 桁の文字列 (`'123.45.67.89'` など) に変換します。この関数は、`struct in_addr` 型を使用する標準 C ライブラリのプログラムとやりとりする場合に便利です。`struct in_addr` 型は、この関数が引数として受け取る 32 ビットにパックされたバイナリデータに対する C の型です。

この関数に渡すバイトシーケンスの長さが 4 バイト以外であれば、*OSError* が発生します。*inet_ntoa()* は、IPv6 をサポートしません。IPv4/v6 のデュアルスタックをサポートする場合は *inet_ntop()* を使用します。

バージョン 3.5 で変更: 書き込み可能な *bytes-like object* を使用できるようになりました。

socket.inet_pton(*address_family*, *ip_string*)

IP アドレスを、アドレスファミリ固有の文字列からパックしたバイナリ形式に変換します。*inet_pton()* は、`struct in_addr` 型 (*inet_aton()* と同様) や `struct in6_addr` を使用するライブラリやネットワークプロトコルを呼び出す際に使用することができます。

現在サポートされている *address_family* は、*AF_INET* と *AF_INET6* です。*ip_string* に不正な IP アドレス文字列を指定すると、*OSError* が発生します。有効な *ip_string* は、*address_family* と `inet_pton()` の実装によって異なります。

利用可能な環境: Unix (一部のプラットフォームを除く)、Windows。

バージョン 3.4 で変更: Windows で利用可能になりました

socket.inet_ntop(*address_family*, *packed_ip*)

パックした IP アドレス (数バイトからなる *bytes-like* **オブジェクト**) を、`'7.10.0.5'` や `'5aef:2b::8'` などの標準的な、アドレスファミリ固有の文字列形式に変換します。*inet_ntop()* は (*inet_ntoa()* と同様に)、`struct in_addr` 型や `struct in6_addr` 型のオブジェクトを返すライブラリやネットワークプロトコル等で使用することができます。

現在サポートされている *address_family* の値は、*AF_INET* と *AF_INET6* です。バイトオブジェクトの *packed_ip* の長さが、指定したアドレスファミリで適切な長さでない場合、*ValueError* が発生し

ます。*inet_ntop()* の呼び出しでエラーが起こると、*OSError* が発生します。

利用可能な環境: Unix (一部のプラットフォームを除く)、Windows。

バージョン 3.4 で変更: Windows で利用可能になりました

バージョン 3.5 で変更: 書き込み可能な *bytes-like object* を使用できるようになりました。

socket.**CMSG_LEN**(*length*)

指定された *length* にある制御メッセージ（CMSG）から、末尾のパディングを除いた全体の長さを返します。この値は多くの場合、*recvmsg()* が制御メッセージの一連の要素を受信するためのバッファサイズとして使用できますが、バッファの末尾が要素である場合であってもパディングは含まれるので、バッファサイズを取得するには **RFC 3542** で求められているように、*CMSG_SPACE()* を使用した移植可能なアプリケーションが必要です。通常 *length* は定数であり、許容範囲外の値が指定された場合は *OverflowError* 例外が送出されます。

ref:利用可能な環境 *<availability>*: 主な Unix で利用できます。他のプラットフォームでも、利用できる場合があります。

バージョン 3.3 で追加.

socket.**CMSG_SPACE**(*length*)

指定された *length* の制御メッセージ（CMSG）の要素を *recvmsg()* が受信するために必要な、パディングを含めたバッファサイズを返します。複数の項目を受信するために必要なバッファスペースは、*CMSG_SPACE()* が返すそれぞれの要素の長さの合計です。通常 *length* は定数であり、許容範囲外の値が指定された場合は *OverflowError* 例外が送出されます。

一部のシステムではこの関数を提供せずに制御メッセージをサポートする可能性があることに注意してください。また、この関数の返り値を使用して設定するバッファサイズは、受信する制御メッセージの量を正確に規定しないことがあり、その後に受信するデータがパディング領域に合う場合があることに注意してください。

ref:利用可能な環境 *<availability>*: 主な Unix で利用できます。他のプラットフォームでも、利用できる場合があります。

バージョン 3.3 で追加.

socket.**getdefaulttimeout**()

新規に生成されたソケットオブジェクトの、デフォルトのタイムアウト値を浮動小数点形式の秒数で返します。タイムアウトを使用しない場合には None を返します。最初に socket モジュールがインポートされた時の初期値は None です。

socket.**setdefaulttimeout**(*timeout*)

新規に生成されるソケットオブジェクトの、デフォルトのタイムアウト値を秒数 (float 型) で設定します。最初に socket モジュールがインポートされた時の初期値は None です。指定可能な値とその意味については *settimeout()* メソッドを参照してください。

socket.**sethostname**(*name*)

マシンのホスト名を *name* に設定します。必要な権限がない場合は *OSError* を送出します。

引数 name を指定して 監査イベント socket.sethostname を送出します。

利用可能な環境: Unix。

バージョン 3.3 で追加.

socket.if_nameindex()

ネットワークインターフェース情報 (index int, name string) のタプルを返します。システムコールが失敗した場合、*OSError* 例外を送出します。

Availability: Unix, Windows。

バージョン 3.3 で追加.

バージョン 3.8 で変更: Windows のサポートが追加されました。

注釈: On Windows network interfaces have different names in different contexts (all names are examples):

- UUID: `{FB605B73-AAC2-49A6-9A2F-25416AEA0573}`
- name: `ethernet_32770`
- friendly name: `vEthernet (nat)`
- description: `Hyper-V Virtual Ethernet Adapter`

This function returns names of the second form from the list, `ethernet_32770` in this example case.

---

socket.if_nametoindex(*if_name*)

インターフェース名 *if_name* に対応するネットワークインターフェースのインデックス番号を返します。対応するインターフェースが存在しない場合は *OSError* 例外を送出します。

Availability: Unix, Windows。

バージョン 3.3 で追加.

バージョン 3.8 で変更: Windows のサポートが追加されました。

参考:

”Interface name” is a name as documented in *if_nameindex()*.

socket.if_indextoname(*if_index*)

インターフェースインデックス番号 *if_index* に対応するネットワークインターフェース名を返します。対応するインターフェースが存在しない場合は *OSError* 例外を送出します。

Availability: Unix, Windows。

バージョン 3.3 で追加.

バージョン 3.8 で変更: Windows のサポートが追加されました。

参考:

”Interface name” is a name as documented in *if_nameindex()*.

18.2.3 socket オブジェクト

ソケットオブジェクトは以下のメソッドを持ちます。*makefile()* 以外のメソッドは、Unix のソケット用システムコールに対応しています。

バージョン 3.2 で変更: *context manager* プロトコルのサポートが追加されました。コンテキストマネージャを終了することは、*close()* を呼ぶことと同一です。

socket.accept()
: 接続を受け付けます。ソケットはアドレスに bind 済みで、listen 中である必要があります。戻り値は (conn, address) のペアで、*conn* は接続を通じてデータの送受信を行うための **新しい** ソケットオブジェクト、*address* は接続先でソケットに bind しているアドレスを示します。

  新たに作成されたソケットは **継承不可** です。

  バージョン 3.4 で変更: ソケットが **継承不可** になりました。

  バージョン 3.5 で変更: システムコールが中断されシグナルハンドラが例外を送出しなかった場合、このメソッドは *InterruptedError* 例外を送出する代わりにシステムコールを再試行するようになりました (論拠については **PEP 475** を参照してください)。

socket.bind(*address*)
: ソケットを *address* に bind します。bind 済みのソケットを再バインドする事はできません。(*address* のフォーマットはアドレスファミリによって異なります -- 前述。)

  引数 self, address を指定して **監査イベント** socket.bind を送出します。

socket.close()
: ソケットを閉じられたものとしてマークします。*makefile()* が返したファイルオブジェクトを閉じる時、対応する下層のシステムリソース（例：ファイル記述子）もすべて閉じます。一度この操作をすると、その後、このソケットオブジェクトに対するすべての操作が失敗します。キューに溜まったデータがフラッシュされた後は、リモート側の端点ではそれ以上のデータを受信しません。

  ソケットはガベージコレクション時に自動的にクローズされます。しかし、明示的に *close()* するか、with 文の中でソケットを使うことを推奨します。

  バージョン 3.6 で変更: 下層の close() が呼び出される時、*OSError* が送出されるようになりました。

注釈: *close()* は接続に関連付けられたリソースを解放しますが、接続をすぐに切断するとは限りません。接続を即座に切断したい場合は、*close()* の前に *shutdown()* を呼び出してください。

---

socket.connect(*address*)
: *address* で示されるリモートソケットに接続します。(*address* のフォーマットはアドレスファミリによって異なります --- 前述。)

  接続が信号によって中断された場合、このメソッドは接続が完了するまで待機するか、タイムアウト時に *socket.timeout* を送出します。タイムアウトは、信号ハンドラが例外を送出せず、ソケットがブロックするかタイムアウトが設定されている場合に起こります。非ブロックソケットでは、接続が信号によって中断された場合 (あるいは信号ハンドラにより例外が送出された場合)、このメソッドは *InterruptedError* 例外を送出します。

  引数 `self`, `address` を指定して **監査イベント** `socket.connect` を送出します。

  バージョン 3.5 で変更: このメソッドは、接続が信号によって中断され、信号ハンドラが例外を送出せず、ソケットがブロックであるかタイムアウトが設定されている場合、*InterruptedError* 例外を送出する代わりに、接続を完了するまで待機するようになりました (論拠については **PEP 475** を参照してください)。

socket.connect_ex(*address*)
: `connect(address)` と同様ですが、C 言語の `connect()` 関数の呼び出しでエラーが発生した場合には例外を送出せずにエラーを戻り値として返します。(これ以外の、"host not found," 等のエラーの場合には例外が発生します。) 処理が正常に終了した場合には `0` を返し、エラー時には `errno` の値を返します。この関数は、非同期接続をサポートする場合などに使用することができます。

  引数 `self`, `address` を指定して **監査イベント** `socket.connect` を送出します。

socket.detach()
: 実際にファイル記述子を閉じることなく、ソケットオブジェクトを閉じた状態にします。ファイル記述子は返却され、他の目的に再利用することができます。

  バージョン 3.2 で追加.

socket.dup()
: ソケットを複製します。

  新たに作成されたソケットは **継承不可** です。

  バージョン 3.4 で変更: ソケットが **継承不可** になりました。

socket.fileno()
: ソケットのファイル記述子を短い整数型で返します。失敗時には、-1 を返します。ファイル記述子は、*select.select()* などで使用します。

  Windows ではこのメソッドで返された小整数をファイル記述子を扱う箇所 (*os.fdopen()* など) で利用できません。Unix にはこの制限はありません。

socket.get_inheritable()
: ソケットのファイル記述子またはソケットのハンドルの **継承可能フラグ** を取得します。ソケットが子プロセスへ継承可能なら `True` 、継承不可なら `False` を返します。

  バージョン 3.4 で追加.

socket.**getpeername**()
ソケットが接続しているリモートアドレスを返します。この関数は、リモート IPv4/v6 ソケットのポート番号を調べる場合などに使用します。*address* のフォーマットはアドレスファミリによって異なります (前述)。この関数をサポートしていないシステムも存在します。

socket.**getsockname**()
ソケット自身のアドレスを返します。この関数は、IPv4/v6 ソケットのポート番号を調べる場合などに使用します。(*address* のフォーマットはアドレスファミリによって異なります --- 前述。)

socket.**getsockopt**(*level*, *optname*[, *buflen*])
ソケットに指定されたオプションを返します (Unix のマニュアルページ *getsockopt(2)* を参照)。SO_* 等のシンボルは、このモジュールで定義しています。*buflen* を省略した場合、取得するオブションは整数とみなし、整数型の値を戻り値とします。*buflen* を指定した場合、長さ *buflen* のバッファでオプションを受け取り、このバッファをバイト列オブジェクトとして返します。このバッファは、呼び出し元プログラムで *struct* モジュール等を利用して内容を読み取ることができます。

socket.**getblocking**()
Return True if socket is in blocking mode, False if in non-blocking.

This is equivalent to checking socket.gettimeout() == 0.

バージョン 3.7 で追加.

socket.**gettimeout**()
ソケットに指定されたタイムアウト値を取得します。タイムアウト値が設定されている場合には浮動小数点型で秒数が、設定されていなければ None が返ります。この値は、最後に呼び出された *setblocking()* または *settimeout()* によって設定されます。

socket.**ioctl**(*control*, *option*)

プラットフォーム Windows

ioctl() メソッドは WSAIoctl システムインタフェースへの制限されたインタフェースです。詳しい情報については、Win32 documentation を参照してください。

他のプラットフォームでは一般的な *fcntl.fcntl()* と *fcntl.ioctl()* が使われるでしょう; これらの関数は第 1 引数としてソケットオブジェクトを取ります。

現在、以下のコントロールコードのみがサポートされています。SIO_RCVALL, SIO_KEEPALIVE_VALS, SIO_LOOPBACK_FAST_PATH。

バージョン 3.6 で変更: SIO_LOOPBACK_FAST_PATH が追加されました。

socket.**listen**([*backlog*])
サーバーを有効にして、接続を受け付けるようにします。*backlog* が指定されている場合、少なくとも 0 以上でなければなりません (それより低い場合、0 に設定されます)。システムが新しい接続を拒否するまでに許可する未受付の接続の数を指定します。指定しない場合、デフォルトの妥当な値が選択されます。

バージョン 3.5 で変更: *backlog* 引数が任意になりました。

socket.**makefile**(*mode='r'*, *buffering=None*, ***, *encoding=None*, *errors=None*, *newline=None*)

ソケットに関連付けられた **ファイルオブジェクト** を返します。戻り値の正確な型は、*makefile()* に指定した引数によります。これらの引数は、組み込み関数 *open()* の引数と同様に解釈されます。ただし、*mode* の値は `'r'` (デフォルト), `'w'`, `'b'` のみがサポートされています。

ソケットはブロッキングモードでなければなりません。タイムアウトを設定することはできますが、タイムアウトが発生すると、ファイルオブジェクトの内部バッファが矛盾した状態になることがあります。

makefile() でファイルオブジェクトにソケットを関連づけた場合、ソケットを閉じるには、関連づけられたすべてのファイルオブジェクトを閉じたあとで、元のソケットの *socket.close()* を呼び出さなければなりません。

注釈: Windows では *subprocess.Popen()* の stream 引数などファイルディスクリプタつき file オブジェクトが期待されている場所では、*makefile()* によって作成される file-like オブジェクトは使用できません。

---

socket.**recv**(*bufsize*[, *flags*])

ソケットからデータを受信し、結果を bytes オブジェクトで返します。一度に受信するデータは、最大でも *bufsize* で指定した量です。オプション引数 *flags* に指定するフラグの意味については、Unix のマニュアルページ *recv(2)* を参照してください。*flags* のデフォルトは 0 です。

注釈: ハードウェアおよびネットワークの現実に最大限マッチするように、*bufsize* の値は比較的小さい 2 の累乗、たとえば 4096、にすべきです。

バージョン 3.5 で変更: システムコールが中断されシグナルハンドラが例外を送出しなかった場合、このメソッドは *InterruptedError* 例外を送出する代わりにシステムコールを再試行するようになりました (論拠については **PEP 475** を参照してください)。

socket.**recvfrom**(*bufsize*[, *flags*])

ソケットからデータを受信し、結果をタプル `(bytes, address)` として返します。*bytes* は受信データの bytes オブジェクトで、*address* は送信元のアドレスを示します。オプション引数 *flags* については、Unix のマニュアルページ *recv(2)* を参照してください。デフォルトは 0 です。(*address* のフォーマットはアドレスファミリによって異なります (前述))

バージョン 3.5 で変更: システムコールが中断されシグナルハンドラが例外を送出しなかった場合、このメソッドは *InterruptedError* 例外を送出する代わりにシステムコールを再試行するようになりました (論拠については **PEP 475** を参照してください)。

バージョン 3.7 で変更: For multicast IPv6 address, first item of *address* does not contain `%scope` part anymore. In order to get full IPv6 address use *getnameinfo()*.

socket.**recvmsg**(*bufsize*[, *ancbufsize*[, *flags*]])

ソケットから通常のデータ (最大 *bufsize* バイト) と補助的なデータを受信します。*ancbufsize* 引数

により、補助的なデータの受信に使用される内部バッファのバイト数として、サイズが設定されます。このデフォルトは 0 で、補助的なデータを受信しないことを意味します。*CMSG_SPACE()* または *CMSG_LEN()* を使用して、補助的なデータの適切なサイズを計算することができ、バッファ内に収まらないアイテムは、短縮されるか破棄されます。*flags* 引数はデフォルトでは 0 で、*recv()* での意味と同じ意味を持ちます。

戻り値は 4 要素のタプル (data, ancdata, msg_flags, address) です。*data* アイテムは、受信した非付属的データを保持する *bytes* オブジェクトです。*ancdata* アイテムは、ゼロ以上のタプル (cmsg_level, cmsg_type, cmsg_data) からなるリストで、受信する付属的なデータ (制御メッセージ) を表します。*cmsg_level* と *cmsg_type* はそれぞれ、プロトコルレベルとプロトコル固有のタイプを指定する整数で、*cmsg_data* は関連するデータを保持する *bytes* オブジェクトです。*msg_flags* アイテムは、受信したメッセージの条件を示す様々なフラグのビット OR です。詳細は、システムのドキュメントを参照してください。受信ソケットが接続されていない場合、*address* は、送信ソケットが利用できる場合にはそのアドレスで、利用できない場合、その値は未指定になります。

一部のシステムでは、*sendmsg()* と *recvmsg()* を使用して、プロセス間で *AF_UNIX* ソケットを経由してファイル記述子を渡すことができます。この機能を使用する場合 (しばしば *SOCK_STREAM* ソケットに限定されます)、*recvmsg()* は、付属的なデータ中に、(socket.SOL_SOCKET, socket.SCM_RIGHTS, fds) という形式のアイテムを返します。ここで、*fds* は、新しいファイル記述子をネイティブ C の int 型のバイナリ配列として表します。システムコールが返った後 *recvmsg()* が例外を送出する場合、まずこのメカニズムを経由して受信したファイル記述子を全て閉じようと試みます。

一部のシステムでは、部分的に受信した付属的なデータアイテムの短縮された長さが示されません。アイテムがバッファの末尾を超えているようである場合、*recvmsg()* は *RuntimeWarning* を送出し、関連するデータの開始位置より前で途切れていない場合、バッファ内の付属的なデータの一部を返します。

SCM_RIGHTS メカニズムをサポートするシステム上では、次の関数が最大 *maxfds* のファイル記述子を受信し、メッセージデータと記述子を含むリストを返しま (無関係な制御メッセージを受信した場合など、予期しない条件は無視します)。*sendmsg()* も参照してください。

```
import socket, array

def recv_fds(sock, msglen, maxfds):
    fds = array.array("i")   # Array of ints
    msg, ancdata, flags, addr = sock.recvmsg(msglen, socket.CMSG_LEN(maxfds * fds.
↪itemsize))
    for cmsg_level, cmsg_type, cmsg_data in ancdata:
        if cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_RIGHTS:
            # Append data, ignoring any truncated integers at the end.
            fds.frombytes(cmsg_data[:len(cmsg_data) - (len(cmsg_data) % fds.itemsize)])
    return msg, list(fds)
```

ref:**利用可能な環境** *<availability>*: 主な Unix で利用できます。他のプラットフォームでも、利用できる場合があります。

バージョン 3.3 で追加.

バージョン 3.5 で変更: システムコールが中断されシグナルハンドラが例外を送出しなかった場合、こ

のメソッドは *InterruptedError* 例外を送出する代わりにシステムコールを再試行するようになりました (論拠については **PEP 475** を参照してください)。

socket.**recvmsg_into**(*buffers*[, *ancbufsize*[, *flags*]])

recvmsg() と同様に動作してソケットから通常のデータと付属的なデータを受信しますが、非付属的データは新しいバイトオブジェクトとして返すのではなく、一連のバッファとして返します。*buffers* 引数は書き込み可能なバッファをエクスポートするオブジェクトのイテラブルでなければなりません (例: *bytearray* オブジェクト)。これらは、全てに書き込まれるか、残りバッファがなくなるまで、非付属的データの連続チャンクで埋められます。オペレーティングシステムによって、使用できるバッファの数が制限 (*sysconf()* 値 SC_IOV_MAX) されている場合があります。*ancbufsize* 引数と *flags* 引数は、*recvmsg()* での意味と同じ意味を持ちます。

戻り値は 4 要素のタプル (nbytes, ancdata, msg_flags, address) です。ここで、*nbytes* はバッファに書き込まれた非付属的データの総数で、*ancdata*、*msg_flags*、*address* は *recvmsg()* と同様です。

以下はプログラム例です:

```
>>> import socket
>>> s1, s2 = socket.socketpair()
>>> b1 = bytearray(b'----')
>>> b2 = bytearray(b'0123456789')
>>> b3 = bytearray(b'--------------')
>>> s1.send(b'Mary had a little lamb')
22
>>> s2.recvmsg_into([b1, memoryview(b2)[2:9], b3])
(22, [], 0, None)
>>> [b1, b2, b3]
[bytearray(b'Mary'), bytearray(b'01 had a 9'), bytearray(b'little lamb---')]
```

ref:**利用可能な環境** *<availability>*: 主な Unix で利用できます。他のプラットフォームでも、利用できる場合があります。

バージョン 3.3 で追加.

socket.**recvfrom_into**(*buffer*[, *nbytes*[, *flags*]])

ソケットからデータを受信し、そのデータを新しいバイト文字列として返す代わりに *buffer* に書きます。戻り値は (nbytes, address) のペアで、*nbytes* は受信したデータのバイト数を、*address* はデータを送信したソケットのアドレスです。オプション引数 *flags* (デフォルト:0) の意味については、Unix マニュアルページ *recv(2)* を参照してください。(*address* のフォーマットは前述のとおりアドレスファミリーに依存します。)

socket.**recv_into**(*buffer*[, *nbytes*[, *flags*]])

nbytes バイトまでのデータをソケットから受信して、そのデータを新しいバイト文字列にするのではなく *buffer* に保存します。*nbytes* が指定されない (あるいは 0 が指定された) 場合、*buffer* の利用可能なサイズまで受信します。受信したバイト数を返り値として返します。オプション引数 *flags* (デフォルト:0) の意味については、Unix マニュアルページ *recv(2)* を参照してください。

socket.**send**(*bytes*[, *flags*])

ソケットにデータを送信します。ソケットはリモートソケットに接続済みでなければなりません。オプション引数 *flags* の意味は、上記 *recv()* と同じです。戻り値として、送信したバイト数を返します。アプリケーションでは、必ず戻り値をチェックし、全てのデータが送られた事を確認する必要があります。データの一部だけが送信された場合、アプリケーションで残りのデータを再送信してください。ソケットプログラミング HOWTO に、さらに詳しい情報があります。

バージョン 3.5 で変更: システムコールが中断されシグナルハンドラが例外を送出しなかった場合、このメソッドは *InterruptedError* 例外を送出する代わりにシステムコールを再試行するようになりました (論拠については **PEP 475** を参照してください)。

socket.**sendall**(*bytes*[, *flags*])

ソケットにデータを送信します。ソケットはリモートソケットに接続済みでなければなりません。オプション引数 *flags* の意味は、上記 *recv()* と同じです。*send()* と異なり、このメソッドは *bytes* の全データを送信するか、エラーが発生するまで処理を継続します。正常終了の場合は None を返し、エラー発生時には例外が発生します。エラー発生時、送信されたバイト数を調べる事はできません。

バージョン 3.5 で変更: ソケットのタイムアウトは、データが正常に送信される度にリセットされなくなりました。ソケットのタイムアウトは、すべてのデータを送る最大の合計時間となります。

バージョン 3.5 で変更: システムコールが中断されシグナルハンドラが例外を送出しなかった場合、このメソッドは *InterruptedError* 例外を送出する代わりにシステムコールを再試行するようになりました (論拠については **PEP 475** を参照してください)。

socket.**sendto**(*bytes*, *address*)

socket.**sendto**(*bytes*, *flags*, *address*)

ソケットにデータを送信します。このメソッドでは接続先を *address* で指定するので、接続済みではいけません。オプション引数 *flags* の意味は、上記 *recv()* と同じです。戻り値として、送信したバイト数を返します。(*address* のフォーマットはアドレスファミリによって異なります --- 前述。)

引数 self, address を指定して **監査イベント** socket.sendto を送出します。

バージョン 3.5 で変更: システムコールが中断されシグナルハンドラが例外を送出しなかった場合、このメソッドは *InterruptedError* 例外を送出する代わりにシステムコールを再試行するようになりました (論拠については **PEP 475** を参照してください)。

socket.**sendmsg**(*buffers*[, *ancdata*[, *flags*[, *address*]]])

非付属的なデータを一連のバッファから集め、単一のメッセージにまとめることで、通常のデータと付属的なデータをソケットに送信します。*buffers* 引数は、非付属的なデータを *bytes-like objects* (例: *bytes* オブジェクト) のイテラブルとして指定します。オペレーティングシステムによって、使用できるバッファの数が制限 (*sysconf()* 値 SC_IOV_MAX) されている場合があります。*ancdata* 引数は付属的なデータ (制御メッセージ) をゼロ以上のタプル (cmsg_level, cmsg_type, cmsg_data) のイテラブルとして指定します。ここで、*cmsg_level* と *cmsg_type* はそれぞれプロトコルレベルとプロトコル固有のタイプを指定する整数で、*cmsg_data* は関連データを保持するバイトライクオブジェクトです。一部のシステム (特に *CMSG_SPACE()* を持たないシステム) では、一度の呼び出しで一つの制御メッセージの送信しかサポートされていない場合があります。*flags* 引数のデフォルトは 0 であり、*send()* での意味と同じ意味を持ちます。None 以外の *address* が渡された場合、メッセージの目的地のアドレスを設定します。戻り値は、送信された非付属的データのバイト数です。

以下の関数は、SCM_RIGHTS メカニズムをサポートするシステムで、ファイル記述子 *fds* を *AF_UNIX* ソケット経由で送信します。*recvmsg()* も参照してください。

```
import socket, array

def send_fds(sock, msg, fds):
    return sock.sendmsg([msg], [(socket.SOL_SOCKET, socket.SCM_RIGHTS, array.array("i",␣
↪fds))])
```

ref:**利用可能な環境** *<availability>*: 主な Unix で利用できます。他のプラットフォームでも、利用できる場合があります。

引数 `self`, `address` を指定して **監査イベント** `socket.sendmsg` を送出します。

バージョン 3.3 で追加.

バージョン 3.5 で変更: システムコールが中断されシグナルハンドラが例外を送出しなかった場合、このメソッドは *InterruptedError* 例外を送出する代わりにシステムコールを再試行するようになりました (論拠については **PEP 475** を参照してください)。

socket.sendmsg_afalg([*msg*], *, *op*[, *iv*[, *assoclen*[, *flags*]]])

sendmsg() の *AF_ALG* ソケット用に特化したバージョンです。*AF_ALG* ソケットの、モード、IV、AEAD に関連づけられたデータ長、フラグを設定します。

利用可能な環境: Linux 2.6.38 以上。

バージョン 3.6 で追加.

socket.sendfile(*file*, *offset=0*, *count=None*)

高性能の *os.sendfile* を使用して、ファイルを EOF まで送信し、送信されたバイトの総数を返します。*file* は、バイナリモードで開かれた標準的なファイルオブジェクトです。*os.sendfile* が使用できない場合 (例: Windows)、または *file* が標準的なファイルでない場合、代わりに *send()* が使用されます。*offset* は、ファイルの読み出し開始位置を指定します。*count* が指定されている場合、ファイルを EOF まで送信するのではなく、転送するバイトの総数を指定します。ファイルの位置は、返る時に更新されます。あるいは、エラー時には *file.tell()* を使用して送信されたバイトの数を確認することができます。ソケットは *SOCK_STREAM* タイプでなければなりません。非ブロックソケットはサポートされていません。

バージョン 3.5 で追加.

socket.set_inheritable(*inheritable*)

ソケットのファイル記述子、またはソケットのハンドルの、**継承可能フラグ** を立てます。

バージョン 3.4 で追加.

socket.setblocking(*flag*)

ソケットをブロッキングモード、または非ブロッキングモードに設定します。*flag* が False の場合にはソケットは非ブロッキングモードになり、True の場合にはブロッキングモードになります。

このメソッドは、次の *settimeout()* 呼び出しの省略表記です:

- `sock.setblocking(True)` は `sock.settimeout(None)` と等価です
- `sock.setblocking(False)` は `sock.settimeout(0.0)` と等価です

バージョン 3.7 で変更: The method no longer applies *SOCK_NONBLOCK* flag on *socket.type*.

socket.settimeout(*value*)

ブロッキングソケットの処理のタイムアウト値を指定します。*value* には float 型で非負の秒数を指定するか、None を指定します。ゼロ以外の値を指定した場合、ソケットの処理が完了する前に *value* で指定した秒数が経過すれば *timeout* 例外を送出します。ゼロを指定した場合、ソケットは非ブロッキングモード状態に置かれます。None を指定した場合、ソケットのタイムアウトを無効にします。

詳しくは **ソケットタイムアウトの注意事項** を参照してください。

バージョン 3.7 で変更: The method no longer toggles *SOCK_NONBLOCK* flag on *socket.type*.

socket.setsockopt(*level*, *optname*, *value: int*)

socket.setsockopt(*level*, *optname*, *value: buffer*)

socket.setsockopt(*level*, *optname*, *None*, *optlen: int*)

指定されたソケットオプションの値を設定します (Unix のマニュアルページ *setsockopt(2)* を参照)。必要なシンボリック定数は、*socket* モジュール (SO_* など) で定義されています。この値は、整数、None、またはバッファを表す *bytes-like object* のいずれかです。バイトライクオブジェクトの場合、バイト文字列に適切なビットが含まれていることを確認するのは呼び出し元の仕事です (C 構造をバイト文字列としてエンコードする方法については、オプションの組み込みモジュール *struct* を参照)。値が None に設定されている場合、optlen 引数が必須です。これは、optval=NULL と optlen=optlen で setsockopt() C 関数を呼び出すのと同一です。

バージョン 3.5 で変更: 書き込み可能な *bytes-like object* を使用できるようになりました。

バージョン 3.6 で変更: setsockopt(level, optname, None, optlen: int) の形式が追加されました。

socket.shutdown(*how*)

接続の片方向、または両方向を切断します。*how* が SHUT_RD の場合、以降は受信を行えません。*how* が SHUT_WR の場合、以降は送信を行えません。*how* が SHUT_RDWR の場合、以降は送受信を行えません。

socket.share(*process_id*)

ソケットを複製し、対象のプロセスと共有するための bytes オブジェクトを返します。対象のプロセスを *process_id* で指定しなければなりません。戻り値の bytes オブジェクトは、何らかのプロセス間通信を使って対象のプロセスに伝えます。対象のプロセス側では、*fromshare()* を使って複製されたソケットをとらえます。オペレーティング・システムは対象のプロセスに対してソケットを複製するため、このメソッドを呼び出した後であれば、元のソケットをクローズしても、対象のプロセスに渡ったソケットには影響がありません。

利用可能な環境: Windows 。

バージョン 3.3 で追加.

read() メソッドと write() メソッドは存在しませんので注意してください。代わりに *flags* を省略した *recv()* と *send()* を使うことができます。

ソケットオブジェクトには以下の *socket* コンストラクタに渡された値に対応した (読み出し専用) 属性があります。

socket.family
ソケットファミリー。

socket.type
ソケットタイプ。

socket.proto
ソケットプロトコル。

18.2.4 ソケットタイムアウトの注意事項

ソケットオブジェクトは、ブロッキングモード、非ブロッキングモード、タイムアウトモードのうち、いずれか 1 つのモードをとります。デフォルトでは、ソケットは常にブロッキングモードで作成されますが、*setdefaulttimeout()* で標準のモードを変更することができます。

- **ブロッキングモード** での操作は、完了するか、または（接続がタイムアウトするなどして）システムがエラーを返すまで、ブロックされます。
- **非ブロッキングモード** での操作は、ただちに完了できない場合、例外を送出して失敗します。この場合の例外の種類は、システムに依存するため、ここに記すことができません。*select* モジュールの関数を使って、ソケットの読み書きが利用可能かどうか、可能な場合はいつ利用できるかを調べることができます。
- **タイムアウトモード** での操作は、指定されたタイムアウトの時間内に完了しなければ、*timeout* 例外を送出します。タイムアウトの時間内にシステムがエラーを返した場合は、そのエラーを返します。

注釈: オペレーティング・システムのレベルでは、**タイムアウトモード** のソケットには、内部的に非ブロッキングモードが設定されています。またブロッキングモードとタイムアウトモードの指定は、ファイル記述子と、「そのファイル記述子と同じネットワーク端点を参照するソケットオブジェクト」との間で共有されます。このことは、例えばソケットの *fileno()* を使うことにした場合に、明らかな影響を与えます。

タイムアウトと connect メソッド

connect() もタイムアウト設定に従います。一般的に、*settimeout()* を *connect()* の前に呼ぶか、*create_connection()* にタイムアウト引数を渡すことが推奨されます。ただし、システムのネットワークスタックが Python のソケットタイムアウトの設定を無視して、自身の接続タイムアウトエラーを返すこともあります。

タイムアウトと accept メソッド

getdefaulttimeout() が *None* でない場合、*accept()* メソッドが返すソケットでは、そのタイムアウトが継承されます。*None* である場合、待機中のソケットの設定によって動作は異なります。

- 待機中のソケットが **ブロッキングモード** または **タイムアウトモード** である場合、*accept()* が返すソケットは、**ブロッキングモード** になります。
- 待機中のソケットが **非ブロッキングモード** である場合、*accept()* が返すソケットは、オペレーティングシステムによってブロッキングモードまたは非ブロッキングモードになります。クロスプラットフォームの動作を確保したい場合、この設定を手動でオーバーライドすることをお勧めします。

18.2.5 使用例

以下は TCP/IP プロトコルの簡単なサンプルとして、受信したデータをクライアントにそのまま返送するサーバ (接続可能なクライアントは一件のみ) と、サーバに接続するクライアントの例を示します。サーバでは、*socket()* ・ *bind()* ・ *listen()* ・ *accept()* を実行し (複数のクライアントからの接続を受け付ける場合、*accept()* を複数回呼び出します)、クライアントでは *socket()* と *connect()* だけを呼び出しています。サーバでは *sendall()* / *recv()* メソッドは listen 中のソケットで実行するのではなく、*accept()* で取得したソケットに対して実行している点にも注意してください。

次のクライアントとサーバは、IPv4 のみをサポートしています。

```
# Echo server program
import socket

HOST = ''                 # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data: break
            conn.sendall(data)
```

```
# Echo client program
import socket

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

次のサンプルは上記のサンプルとほとんど同じですが、IPv4 と IPv6 の両方をサポートしています。サーバでは、IPv4/v6 の両方ではなく、利用可能な最初のアドレスファミリだけを listen しています。ほとんどの IPv6 対応システムでは IPv6 が先に現れるため、サーバは IPv4 には応答しません。クライアントでは名前解決の結果として取得したアドレスに順次接続を試み、最初に接続に成功したソケットにデータを送信しています。

```
# Echo server program
import socket
import sys

HOST = None               # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                              socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data: break
        conn.send(data)
```

```
# Echo client program
import socket
import sys

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
with s:
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

次の例は、Windows で raw socket を利用して非常にシンプルなネットワークスニファーを書きます。このサンプルを実行するには、インタフェースを操作するための管理者権限が必要です:

```
import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a package
print(s.recvfrom(65565))

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

次の例では、ソケットインターフェースを使用してローソケットプロトコルを使用する CAN ネットワークと通信する方法を説明します。ブロードキャストマネージャプロトコルで CAN を使用するには、以下でソケットを開きます。

```
socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_BCM)
```

ソケットの束縛 (CAN_RAW) または (*CAN_BCM*) 接続を行ったあと、ソケットオブジェクトで *socket.send()* と *socket.recv()* 操作 (とそのカウンターパート) を通常通りに使用することができます。

最後の例では、特権が必要になるかもしれません:

```
import socket
import struct


# CAN frame packing/unpacking (see 'struct can_frame' in <linux/can.h>)

can_frame_fmt = "=IB3x8s"
can_frame_size = struct.calcsize(can_frame_fmt)

def build_can_frame(can_id, data):
    can_dlc = len(data)
    data = data.ljust(8, b'\x00')
    return struct.pack(can_frame_fmt, can_id, can_dlc, data)

def dissect_can_frame(frame):
    can_id, can_dlc, data = struct.unpack(can_frame_fmt, frame)
    return (can_id, can_dlc, data[:can_dlc])


# create a raw socket and bind it to the 'vcan0' interface
s = socket.socket(socket.AF_CAN, socket.SOCK_RAW, socket.CAN_RAW)
s.bind(('vcan0',))

while True:
    cf, addr = s.recvfrom(can_frame_size)

    print('Received: can_id=%x, can_dlc=%x, data=%s' % dissect_can_frame(cf))

    try:
        s.send(cf)
    except OSError:
        print('Error sending CAN frame')

    try:
        s.send(build_can_frame(0x01, b'\x01\x02\x03'))
    except OSError:
        print('Error sending CAN frame')
```

この例を、ほとんど間を空けずに複数回実行すると、以下のエラーが発生する場合があります:

```
OSError: [Errno 98] Address already in use
```

これは以前の実行がソケットを TIME_WAIT 状態のままにし、すぐには再利用できないことで起こります。

これを防ぐのに、*socket* フラグの socket.SO_REUSEADDR があります:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))
```

SO_REUSEADDR フラグは、TIME_WAIT 状態にあるローカルソケットをそのタイムアウト期限が自然に切れるのを待つことなく再利用することをカーネルに伝えます。

参考:

C 言語によるソケットプログラミングの基礎については、以下の資料を参照してください。

- *An Introductory 4.3BSD Interprocess Communication Tutorial*, by Stuart Sechrest
- *An Advanced 4.3BSD Interprocess Communication Tutorial*, by Samuel J. Leffler et al,

両書とも UNIX Programmer's Manual, Supplementary Documents 1 (PS1:7 章 PS1:8 章)。ソケットの詳細については、各プラットフォームのソケット関連システムコールに関するドキュメントも参照してください。Unix ではマニュアルページ、Windows では WinSock (または WinSock2) 仕様書をご覧ください。IPv6 対応の API については、**RFC 3493** "Basic Socket Interface Extensions for IPv6" を参照してください。

18.3 ssl --- ソケットオブジェクトに対する TLS/SSL ラッパー

Source code: Lib/ssl.py

このモジュールは Transport Layer Security ("Secure Sockets Layer" という名前でよく知られています) 暗号化と、クライアントサイド、サーバサイド両方のネットワークソケットのためのピア認証の仕組みを提供しています。このモジュールは OpenSSL ライブラリを利用しています。OpenSSL は、すべてのモダンな Unix システム、Windows 、Mac OS X 、その他幾つかの OpenSSL がインストールされているプラットフォームで利用できます。

注釈: OS のソケット API に対して実装されているので、幾つかの挙動はプラットフォーム依存になるかもしれません。インストールされている OpenSSL のバージョンの違いも挙動の違いの原因になるかもしれません。例えば、TLSv1.1, TLSv1.2 は openssl version 1.0.1 以降でのみ利用できます。

警告: セキュリティで考慮すべき点 を読まずにこのモジュールを使用しないでください。SSL のデフォルト設定はアプリケーションに十分ではないので、読まない場合はセキュリティに誤った意識を持ってしまうかもしれません。

このセクションでは、ssl モジュールのオブジェクトと関数を解説します。TLS, SSL, 証明書に関するより一般的な情報は、末尾にある "See Also" のセクションを参照してください。

このモジュールは *ssl.SSLSocket* クラスを提供します。このクラスは *socket.socket* クラスを継承していて、ソケットで通信されるデータを SSL で暗号化・復号するソケットに似たラッパーになります。また、このクラスは、接続の相手側からの証明書を取得する getpeercert() メソッドや、セキュア接続で使うための暗号方式を取得する cipher() メソッドのような追加のメソッドをサポートしています。

より洗練されたアプリケーションのために、*ssl.SSLContext* クラスが設定と証明書の管理の助けとなるでしょう。それは *SSLContext.wrap_socket()* メソッドを通して SSL ソケットを作成することで引き継がれます。

バージョン 3.5.3 で変更: Updated to support linking with OpenSSL 1.1.0

バージョン 3.6 で変更: OpenSSL 0.9.8, 1.0.0, 1.0.1 は廃止されており、もはやサポートされていません。ssl モジュールは、将来的に OpenSSL 1.0.2 または 1.1.0 を必要とするようになります。

18.3.1 関数、定数、例外

ソケットの作成

Since Python 3.2 and 2.7.9, it is recommended to use the *SSLContext.wrap_socket()* of an *SSLContext* instance to wrap sockets as *SSLSocket* objects. The helper functions *create_default_context()* returns a new context with secure default settings. The old *wrap_socket()* function is deprecated since it is both inefficient and has no support for server name indication (SNI) and hostname matching.

Client socket example with default context and IPv4/IPv6 dual stack:

```
import socket
import ssl

hostname = 'www.python.org'
context = ssl.create_default_context()

with socket.create_connection((hostname, 443)) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

Client socket example with custom context and IPv4:

```
hostname = 'www.python.org'
# PROTOCOL_TLS_CLIENT requires valid cert chain and hostname
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.load_verify_locations('path/to/cabundle.pem')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

Server socket example listening on localhost IPv4:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain('/path/to/certchain.pem', '/path/to/private.key')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    sock.bind(('127.0.0.1', 8443))
    sock.listen(5)
    with context.wrap_socket(sock, server_side=True) as ssock:
        conn, addr = ssock.accept()
        ...
```

コンテキストの作成

コンビニエンス関数が、共通の目的で使用される *SSLContext* オブジェクトを作成するのに役立ちます。

ssl.create_default_context(*purpose=Purpose.SERVER_AUTH*, *cafile=None*, *capath=None*, *cadata=None*)

新規の *SSLContext* オブジェクトを、与えられた *purpose* のデフォルト設定で返します。設定は *ssl* モジュールで選択され、通常は *SSLContext* のコンストラクタを直接呼び出すよりも高いセキュリティレベルを表現します。

cafile, *capath*, *cadata* は証明書の検証で信用するオプションの CA 証明書で、*SSLContext.load_verify_locations()* のものと同じです。これら 3 つすべてが *None* であれば、この関数は代わりにシステムのデフォルトの CA 証明書を信用して選択することができます。

設定は、*PROTOCOL_TLS*, *OP_NO_SSLv2*, RC4 と非認証暗号化スイート以外の、高度暗号化スイートを利用した *OP_NO_SSLv3* です。*SERVER_AUTH* を *purpose* として渡すと、*verify_mode* を *CERT_REQUIRED* に設定し、CA 証明書をロードする (*cafile*, *capath*, *cadata* の少なくとも 1 つが与えられている場合) か、*SSLContext.load_default_certs()* を使用してデフォルトの CA 証明書をロードします。

When *keylog_filename* is supported and the environment variable SSLKEYLOGFILE is set, *create_default_context()* enables key logging.

注釈: プロトコル、オプション、暗号方式その他の設定は、事前に非推奨の状態にすることなく、もっと制限の強い値に変更される場合があります。これらの値は、互換性と安全性との妥当なバランスをとって決められます。

もしもあなたのアプリケーションが特定の設定を必要とする場合、*SSLContext* を作って自分自身で設定を適用すべきです。

注釈: ある種の古いクライアントやサーバが接続しようと試みてきた場合に、この関数で作られた *SSLContext* が "Protocol or cipher suite mismatch" で始まるエラーを起こすのを目撃したらそれは、この関数が *OP_NO_SSLv3* を使って除外している SSL 3.0 しかサポートしていないのでしょう。SSL 3.0 は 完璧にぶっ壊れている ことが広く知られています。それでもまだこの関数を使って、ただし SSL 3.0 接続を許可したいと望むならば、これをこのように再有効化できます:

```
ctx = ssl.create_default_context(Purpose.CLIENT_AUTH)
ctx.options &= ~ssl.OP_NO_SSLv3
```

バージョン 3.4 で追加.

バージョン 3.4.4 で変更: デフォルトの暗号設定から RC4 が除かれました。

バージョン 3.6 で変更: デフォルトの暗号化文字列に ChaCha20/Poly1305 が追加されました。

デフォルトの暗号化文字列から 3DES が除かれました。

バージョン 3.8 で変更: Support for key logging to `SSLKEYLOGFILE` was added.

例外

exception ssl.SSLError

(現在のところ OpenSSL ライブラリによって提供されている) 下層の SSL 実装からのエラーを伝えるための例外です。このエラーは、低レベルなネットワークの上に載っている、高レベルな暗号化と認証レイヤーでの問題を通知します。このエラーは *OSError* のサブタイプです。*SSLError* インスタンスのエラーコードとメッセージは OpenSSL ライブラリによるものです。

バージョン 3.3 で変更: *SSLError* は以前は *socket.error* のサブタイプでした。

library

エラーが起こった OpenSSL サブモジュールを示すニーモニック文字列で、`SSL`, `PEM`, `X509` などです。取り得る値は OpenSSL のバージョンに依存します。

バージョン 3.3 で追加.

reason

エラーが起こった原因を示すニーモニック文字列で、`CERTIFICATE_VERIFY_FAILED` などです。取り得る値は OpenSSL のバージョンに依存します。

バージョン 3.3 で追加.

exception ssl.SSLZeroReturnError

読み出しあるいは書き込みを試みようとした際に SSL コネクションが行儀よく閉じられてしまった場合に送出される *SSLError* サブクラス例外です。これは下層の転送 (read TCP) が閉じたことは意味しないことに注意してください。

バージョン 3.3 で追加.

exception ssl.SSLWantReadError

読み出しあるいは書き込みを試みようとした際に、リクエストが遂行される前に下層の TCP 転送で受け取る必要があるデータが不足した場合に *non-blocking SSL socket* によって送出される *SSLError* サブクラス例外です。

バージョン 3.3 で追加.

exception ssl.SSLWantWriteError
: 読み出しあるいは書き込みを試みようとした際に、リクエストが遂行される前に下層の TCP 転送が送信する必要があるデータが不足した場合に *non-blocking SSL socket* によって送出される *SSLError* サブクラス例外です。

  バージョン 3.3 で追加.

exception ssl.SSLSyscallError
: SSL ソケット上で操作を遂行しようとしていてシステムエラーが起こった場合に送出される *SSLError* サブクラス例外です。残念ながら元となった errno 番号を調べる簡単な方法はありません。

  バージョン 3.3 で追加.

exception ssl.SSLEOFError
: SSL コネクションが唐突に打ち切られた際に送出される *SSLError* サブクラス例外です。一般的に、このエラーが起こったら下層の転送を再利用しようと試みるべきではありません。

  バージョン 3.3 で追加.

exception ssl.SSLCertVerificationError
: A subclass of *SSLError* raised when certificate validation has failed.

  バージョン 3.7 で追加.

  verify_code
  : A numeric error number that denotes the verification error.

  verify_message
  : A human readable string of the verification error.

exception ssl.CertificateError
: *SSLCertVerificationError* の別名です。

  バージョン 3.7 で変更: 例外は *SSLCertVerificationError* の別名になりました。

乱数生成

ssl.RAND_bytes(*num*)
: 暗号学的に強固な擬似乱数の *num* バイトを返します。擬似乱数生成器に十分なデータでシードが与えられていない場合や、現在の RANDOM メソッドに操作がサポートされていない場合は *SSLError* を送出します。*RAND_status()* を使って擬似乱数生成器の状態をチェックできます。*RAND_add()* を使って擬似乱数生成器にシードを与えることができます。

  ほとんどすべてのアプリケーションでは *os.urandom()* が望ましいです。

  暗号論的擬似乱数生成器に要求されることについては Wikipedia の記事 Cryptographically secure pseudorandom number generator (CSPRNG) (日本語版: 暗号論的擬似乱数生成器) を参照してください。

  バージョン 3.3 で追加.

ssl.RAND_pseudo_bytes(*num*)

(bytes, is_cryptographic) タプルを返却: bytes は長さ *num* の擬似乱数バイト列、is_cryptographic は、生成されたバイト列が暗号として強ければ `True` 。操作が現在使われている RAND メソッドでサポートされていなければ、*SSLError* が送出されます。

生成される擬似乱数バイトシーケンスは十分な長さであれば一意にはなるでしょうが、必ずしも予測不可能とは言えません。これは非暗号目的、あるいは暗号化プロトコルでの若干の用途に使われますが、普通は鍵生成などには使いません。

ほとんどすべてのアプリケーションでは *os.urandom()* が望ましいです。

バージョン 3.3 で追加.

バージョン 3.6 で非推奨: OpenSSL は *ssl.RAND_pseudo_bytes()* を廃止しました。代わりに *ssl.RAND_bytes()* を使用してください。

ssl.RAND_status()

SSL 擬似乱数生成器が十分なランダム性 (randomness) を受け取っている時に `True` を、それ以外の場合は *False* を返します。*ssl.RAND_egd()* と *ssl.RAND_add()* を使って擬似乱数生成機にランダム性を加えることができます。

ssl.RAND_egd(*path*)

もしエントロピー収集デーモン (EGD=entropy-gathering daemon) が動いていて、*path* が EGD へのソケットのパスだった場合、この関数はそのソケットから 256 バイトのランダム性を読み込み、SSL 擬似乱数生成器にそれを渡すことで、生成される暗号鍵のセキュリティを向上させることができます。これは、より良いランダム性のソースが無いシステムでのみ必要です。

エントロピー収集デーモンについては、http://egd.sourceforge.net/ や http://prngd.sourceforge.net/ を参照してください。

利用可能な環境: LibreSSL および 11.0 を超えるバージョンの OpenSSL は利用できません。

ssl.RAND_add(*bytes*, *entropy*)

与えられた *bytes* を SSL 擬似乱数生成器に混ぜます。*entropy* 引数 (float 値) は、その文字列に含まれるエントロピーの下限 (lower bound) です。(なので、いつでも `0.0` を使うことができます。) エントロピーのソースについてのより詳しい情報は、**RFC 1750** を参照してください。

バージョン 3.5 で変更: 書き込み可能な *bytes-like object* を使用できるようになりました。

証明書の取り扱い

ssl.match_hostname(*cert*, *hostname*)

(*SSLSocket.getpeercert()* が返してきたようなデコードされたフォーマットの) *cert* が、与えられた *hostname* に合致するかを検証します。HTTPS サーバの身元をチェックするために適用されるルールは **RFC 2818**, **RFC 5280**, **RFC 6125** で概説されているものです。HTTPS に加え、この関数は他の SSL ベースのプロトコル、例えば FTPS, IMAPS, POPS などのサーバの身元をチェックするのに相応しいはずです。

失敗すれば *CertificateError* が送出されます。成功すれば、この関数は何も返しません:

```
>>> cert = {'subject': ((('commonName', 'example.com'),),)}
>>> ssl.match_hostname(cert, "example.com")
>>> ssl.match_hostname(cert, "example.org")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/py3k/Lib/ssl.py", line 130, in match_hostname
ssl.CertificateError: hostname 'example.org' doesn't match 'example.com'
```

バージョン 3.2 で追加.

バージョン 3.3.3 で変更: この関数は **RFC 6125** の section 6.4.3 に従うようになりましたので、マルチプルワイルドカード (例. `*.*.com` や `*a*.example.org`) にも国際化ドメイン名 (IDN=internationalized domain name) フラグメント内部に含まれるワイルドカードのどちらにも合致しません。`www*.xn--pthon-kva.org` のような IDN A-labels はまだサポートしますが、`x*.python.org` はもはや `xn--tda.python.org` には合致しません。

バージョン 3.5 で変更: 認定書の subjectAltName フィールドで提示されている場合、IP アドレスの一致がサポートされるようになりました。

バージョン 3.7 で変更: The function is no longer used to TLS connections. Hostname matching is now performed by OpenSSL.

Allow wildcard when it is the leftmost and the only character in that segment. Partial wildcards like `www*.example.com` are no longer supported.

バージョン 3.7 で非推奨.

ssl.cert_time_to_seconds(*cert_time*)

`cert_time` として証明書内の "notBefore" や "notAfter" の `"%b %d %H:%M:%S %Y %Z"` strptime フォーマット (C locale) 日付を渡すと、エポックからの積算秒を返します。

例です。:

```
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan  5 09:34:43 2018 GMT")
>>> timestamp
1515144883
>>> from datetime import datetime
>>> print(datetime.utcfromtimestamp(timestamp))
2018-01-05 09:34:43
```

"notBefore" や "notAfter" の日付には GMT を使わなければなりません (**RFC 5280**)。

バージョン 3.5 で変更: 入力文字列に指定された 'GMT' タイムゾーンを UTC として解釈するようになりました。以前はローカルタイムで解釈していました。また、整数を返すようになりました (入力に含まれる秒の端数を含まない)。

ssl.get_server_certificate(*addr*, *ssl_version=PROTOCOL_TLS*, *ca_certs=None*)

SSL で保護されたサーバーのアドレス `addr` を (*hostname*, *port-number*) の形で受け取り、そのサーバーから証明書を取得し、それを PEM エンコードされた文字列として返します。`ssl_version` が

指定された場合は、サーバーに接続を試みるときにそのバージョンの SSL プロトコルを利用します。ca_certs が指定された場合、それは *SSLContext.wrap_socket()* の同名の引数と同じフォーマットで、ルート証明書のリストを含むファイルでなければなりません。この関数はサーバー証明書をルート証明書リストに対して認証し、認証が失敗した場合にこの関数も失敗します。

バージョン 3.3 で変更: この関数は IPv6 互換になりました。

バージョン 3.5 で変更: *ssl_version* のデフォルトが、最近のサーバへの最大限の互換性のために *PROTOCOL_SSLv3* から *PROTOCOL_TLS* に変更されました。

ssl.**DER_cert_to_PEM_cert**(*DER_cert_bytes*)

DER エンコードされたバイト列として与えられた証明書から、PEM エンコードされたバージョンの同じ証明書を返します。

ssl.**PEM_cert_to_DER_cert**(*PEM_cert_string*)

PEM 形式の ASCII 文字列として与えられた証明書から、同じ証明書を DER エンコードしたバイト列を返します。

ssl.**get_default_verify_paths**()

OpenSSL デフォルトの cafile, capath を指すパスを名前付きタプルで返します。パスは *SSLContext.set_default_verify_paths()* で使われるものと同じです。戻り値は *named tuple* DefaultVerifyPaths です:

- cafile - cafile の解決済みパス、またはファイルが存在しない場合は None
- capath - capath の解決済みパス、またはディレクトリが存在しない場合は None
- openssl_cafile_env - cafile を指す OpenSSL の環境変数
- openssl_cafile - OpenSSL にハードコードされた cafile のパス
- openssl_capath_env - capath を指す OpenSSL の環境変数
- openssl_capath - OpenSSL にハードコードされた capath のパス

利用可能な環境: LibreSSL では環境変数 openssl_cafile_env と openssl_capath_env が無視されます

バージョン 3.4 で追加.

ssl.**enum_certificates**(*store_name*)

Windows のシステム証明書ストアより証明書を抽出します。*store_name* は CA, ROOT, MY のうちどれか一つでしょう。Windows は追加の証明書ストアを提供しているかもしれません。

この関数はタプル (cert_bytes, encoding_type, trust) のリストで返します。encoding_type は cert_bytes のエンコーディングを表します。X.509 ASN.1 に対する x509_asn か PKCS#7 ASN.1 データに対する pkcs_7_asn のいずれかです。trust は、証明書の目的を、OIDS を内容に持つ set として表すか、または証明書がすべての目的で信頼できるならば True です。

以下はプログラム例です:

```
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509_asn', {'1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2'}),
 (b'data...', 'x509_asn', True)]
```

利用可能な環境: Windows 。

バージョン 3.4 で追加.

`ssl.enum_crls`(*store_name*)

Windows のシステム証明書ストアより CRLs を抽出します。*store_name* は `CA`, `ROOT`, `MY` のうちどれか一つでしょう。Windows は追加の証明書ストアを提供しているかもしれません。

この関数はタプル (cert_bytes, encoding_type, trust) のリストで返します。encoding_type は cert_bytes のエンコーディングを表します。X.509 ASN.1 に対する `x509_asn` か PKCS#7 ASN.1 データに対する `pkcs_7_asn` のいずれかです。

利用可能な環境: Windows 。

バージョン 3.4 で追加.

`ssl.wrap_socket`(*sock*, *keyfile=None*, *certfile=None*, *server_side=False*, *cert_reqs=CERT_NONE*, *ssl_version=PROTOCOL_TLS*, *ca_certs=None*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *ciphers=None*)

socket.socket のインスタンス `sock` を受け取り、*socket.socket* のサブタイプである *ssl.SSLSocket* のインスタンスを返します。*ssl.SSLSocket* は低レイヤのソケットを SSL コンテキストでラップします。`sock` は *SOCK_STREAM* ソケットでなければなりません; ほかのタイプのソケットはサポートされていません。

Internally, function creates a *SSLContext* with protocol *ssl_version* and *SSLContext.options* set to *cert_reqs*. If parameters *keyfile*, *certfile*, *ca_certs* or *ciphers* are set, then the values are passed to *SSLContext.load_cert_chain()*, *SSLContext.load_verify_locations()*, and *SSLContext.set_ciphers()*.

server_side, *do_handshake_on_connect*, *suppress_ragged_eofs* 引数は *SSLContext.wrap_socket()* と同じ意味を持ちます。

バージョン 3.7 で非推奨: Since Python 3.2 and 2.7.9, it is recommended to use the *SSLContext.wrap_socket()* instead of *wrap_socket()*. The top-level function is limited and creates an insecure client socket without server name indication or hostname matching.

定数

すべての定数が *enum.IntEnum* コレクションまたは *enum.IntFlag* コレクションになりました。

バージョン 3.6 で追加.

ssl.CERT_NONE

Possible value for *SSLContext.verify_mode*, or the `cert_reqs` parameter to *wrap_socket()*. Except for *PROTOCOL_TLS_CLIENT*, it is the default mode. With client-side sockets, just about any cert is accepted. Validation errors, such as untrusted or expired cert, are ignored and do not abort the TLS/SSL handshake.

In server mode, no certificate is requested from the client, so the client does not send any for client cert authentication.

このドキュメントの下の方の、**セキュリティで考慮すべき点** に関する議論を参照してください。

ssl.CERT_OPTIONAL

Possible value for *SSLContext.verify_mode*, or the `cert_reqs` parameter to *wrap_socket()*. In client mode, *CERT_OPTIONAL* has the same meaning as *CERT_REQUIRED*. It is recommended to use *CERT_REQUIRED* for client-side sockets instead.

In server mode, a client certificate request is sent to the client. The client may either ignore the request or send a certificate in order perform TLS client cert authentication. If the client chooses to send a certificate, it is verified. Any verification error immediately aborts the TLS handshake.

この設定では、正当な CA 証明書のセットを *SSLContext.load_verify_locations()* または *wrap_socket()* の `ca_certs` パラメータのどちらかに渡す必要があります。

ssl.CERT_REQUIRED

Possible value for *SSLContext.verify_mode*, or the `cert_reqs` parameter to *wrap_socket()*. In this mode, certificates are required from the other side of the socket connection; an *SSLError* will be raised if no certificate is provided, or if its validation fails. This mode is **not** sufficient to verify a certificate in client mode as it does not match hostnames. *check_hostname* must be enabled as well to verify the authenticity of a cert. *PROTOCOL_TLS_CLIENT* uses *CERT_REQUIRED* and enables *check_hostname* by default.

With server socket, this mode provides mandatory TLS client cert authentication. A client certificate request is sent to the client and the client must provide a valid and trusted certificate.

この設定では、正当な CA 証明書のセットを *SSLContext.load_verify_locations()* または *wrap_socket()* の `ca_certs` パラメータのどちらかに渡す必要があります。

class ssl.VerifyMode

CERT_* 定数の *enum.IntEnum* コレクションです。

バージョン 3.6 で追加.

ssl.VERIFY_DEFAULT

SSLContext.verify_flags に渡せる値です。このモードでは、証明書失効リスト (CRLs) はチェッ

クされません。デフォルトでは OpenSSL は CRLs を必要ともしませんし検証にも使いません。

バージョン 3.4 で追加.

ssl.VERIFY_CRL_CHECK_LEAF
: *SSLContext.verify_flags* に渡せる値です。このモードでは、接続先の証明書のみがチェックされ、仲介の CA 証明書はチェックされません。接続先証明書の発行者 (その CA の直接の祖先) によって署名された妥当な CRL が必要です。*SSLContext.load_verify_locations* で相応しい CRL をロードしていなければ、検証は失敗します。

バージョン 3.4 で追加.

ssl.VERIFY_CRL_CHECK_CHAIN
: *SSLContext.verify_flags* に渡せる値です。このモードでは、接続先の証明書チェイン内のすべての証明書についての CRLs がチェックされます。

バージョン 3.4 で追加.

ssl.VERIFY_X509_STRICT
: *SSLContext.verify_flags* に渡せる値で、壊れた X.509 証明書に対するワークアラウンドを無効にします。

バージョン 3.4 で追加.

ssl.VERIFY_X509_TRUSTED_FIRST
: *SSLContext.verify_flags* に渡せる値です。OpenSSL に対し、証明書検証のために信頼チェインを構築する際、信頼できる証明書を選ぶように指示します。これはデフォルトで有効にされています。

バージョン 3.4.4 で追加.

class ssl.VerifyFlags
: VERIFY_* 定数の *enum.IntFlag* コレクションです。

バージョン 3.6 で追加.

ssl.PROTOCOL_TLS
: クライアントとサーバの両方がサポートするプロトコルバージョンのうち、最も大きなものを選択します。名前に反して、このオプションは ”SSL” と ”TLS” プロトコルのいずれも選択できます。

バージョン 3.6 で追加.

ssl.PROTOCOL_TLS_CLIENT
: *PROTOCOL_TLS* のような最高のプロトコルバージョンを自動的にネゴシエートしますが、クライアントサイドの *SSLSocket* 接続しかサポートしません。このプロトコルでは、デフォルトで *CERT_REQUIRED* と *check_hostname* が有効になっています。

バージョン 3.6 で追加.

ssl.PROTOCOL_TLS_SERVER
: *PROTOCOL_TLS* のような最高のプロトコルバージョンを自動的にネゴシエートしますが、サーバサイドの *SSLSocket* 接続しかサポートしません。

バージョン 3.6 で追加.

ssl.PROTOCOL_SSLv23
PROTOCOL_TLS のエイリアスです。

バージョン 3.6 で非推奨: 代わりに *PROTOCOL_TLS* を使用してください。

ssl.PROTOCOL_SSLv2
チャンネル暗号化プロトコルとして SSL バージョン 2 を選択します。

このプロトコルは、OpenSSL が OPENSSL_NO_SSL2 フラグが有効な状態でコンパイルされている場合には利用できません。

警告: SSL version 2 は非セキュアです。このプロトコルは強く非推奨です。

バージョン 3.6 で非推奨: OpenSSL は SSLv2 へのサポートを打切りました。

ssl.PROTOCOL_SSLv3
チャンネル暗号化プロトコルとして SSL バージョン 3 を選択します。

このプロトコルは、OpenSSL が OPENSSL_NO_SSLv3 フラグが有効な状態でコンパイルされている場合には利用できません。

警告: SSL version 3 は非セキュアです。このプロトコルは強く非推奨です。

バージョン 3.6 で非推奨: OpenSSL は全てのバージョン固有のプロトコルを廃止しました。デフォルトプロトコルの *PROTOCOL_TLS* に *OP_NO_SSLv3* などのフラグをつけて使用してください。

ssl.PROTOCOL_TLSv1
チャンネル暗号化プロトコルとして TLS バージョン 1.0 を選択します。

バージョン 3.6 で非推奨: OpenSSL は全てのバージョン固有のプロトコルを廃止しました。デフォルトプロトコルの *PROTOCOL_TLS* に *OP_NO_SSLv3* などのフラグをつけて使用してください。

ssl.PROTOCOL_TLSv1_1
チャンネル暗号化プロトコルとして TLS バージョン 1.1 を選択します。openssl version 1.0.1+ のみで利用可能です。

バージョン 3.4 で追加.

バージョン 3.6 で非推奨: OpenSSL は全てのバージョン固有のプロトコルを廃止しました。デフォルトプロトコルの *PROTOCOL_TLS* に *OP_NO_SSLv3* などのフラグをつけて使用してください。

ssl.PROTOCOL_TLSv1_2
チャンネル暗号化プロトコルとして TLS バージョン 1.2 を選択します。これは最も現代的で、接続の両サイドが利用できる場合は、たぶん最も安全な選択肢です。openssl version 1.0.1+ のみで利用可能です。

バージョン 3.4 で追加.

バージョン 3.6 で非推奨: OpenSSL は全てのバージョン固有のプロトコルを廃止しました。デフォルトプロトコルの *PROTOCOL_TLS* に *OP_NO_SSLv3* などのフラグをつけて使用してください。

ssl.OP_ALL
相手にする SSL 実装のさまざまなバグを回避するためのワークアラウンドを有効にします。このオプションはデフォルトで有効です。これを有効にする場合 OpenSSL 用の同じ意味のフラグ SSL_OP_ALL をセットする必要はありません。

バージョン 3.2 で追加.

ssl.OP_NO_SSLv2
SSLv2 接続が行われないようにします。このオプションは *PROTOCOL_TLS* と組み合わされている場合にのみ適用されます。ピアがプロトコルバージョンとして SSLv2 を選択しないようにします。

バージョン 3.2 で追加.

バージョン 3.6 で非推奨: SSLv2 は非推奨です

ssl.OP_NO_SSLv3
SSLv3 接続が行われないようにします。このオプションは *PROTOCOL_TLS* と組み合わされている場合にのみ適用されます。ピアがプロトコルバージョンとして SSLv3 を選択しないようにします。

バージョン 3.2 で追加.

バージョン 3.6 で非推奨: SSLv3 は非推奨です

ssl.OP_NO_TLSv1
TLSv1 接続が行われないようにします。このオプションは *PROTOCOL_TLS* と組み合わされている場合にのみ適用されます。ピアがプロトコルバージョンとして TLSv1 を選択しないようにします。

バージョン 3.2 で追加.

バージョン 3.7 で非推奨: The option is deprecated since OpenSSL 1.1.0, use the new *SSLContext.minimum_version* and *SSLContext.maximum_version* instead.

ssl.OP_NO_TLSv1_1
TLSv1.1 接続が行われないようにします。このオプションは *PROTOCOL_TLS* と組み合わされている場合にのみ適用されます。ピアがプロトコルバージョンとして TLSv1.1 を選択しないようにします。openssl バージョン 1.0.1 以降でのみ利用できます。

バージョン 3.4 で追加.

バージョン 3.7 で非推奨: The option is deprecated since OpenSSL 1.1.0.

ssl.OP_NO_TLSv1_2
TLSv1.2 接続が行われないようにします。このオプションは *PROTOCOL_TLS* と組み合わされている場合にのみ適用されます。ピアがプロトコルバージョンとして TLSv1.2 を選択しないようにします。openssl バージョン 1.0.1 以降でのみ利用できます。

バージョン 3.4 で追加.

バージョン 3.7 で非推奨: The option is deprecated since OpenSSL 1.1.0.

ssl.OP_NO_TLSv1_3

Prevents a TLSv1.3 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing TLSv1.3 as the protocol version. TLS 1.3 is available with OpenSSL 1.1.1 or later. When Python has been compiled against an older version of OpenSSL, the flag defaults to *0*.

バージョン 3.7 で追加.

バージョン 3.7 で非推奨: The option is deprecated since OpenSSL 1.1.0. It was added to 2.7.15, 3.6.3 and 3.7.0 for backwards compatibility with OpenSSL 1.0.2.

ssl.OP_NO_RENEGOTIATION

Disable all renegotiation in TLSv1.2 and earlier. Do not send HelloRequest messages, and ignore renegotiation requests via ClientHello.

このオプションは OpenSSL 1.1.0h 以降のみで使用できます。

バージョン 3.7 で追加.

ssl.OP_CIPHER_SERVER_PREFERENCE

暗号の優先順位として、クライアントのものではなくサーバのものを使います。このオプションはクライアントソケットと SSLv2 のサーバソケットでは効果はありません。

バージョン 3.3 で追加.

ssl.OP_SINGLE_DH_USE

SSL セッションを区別するのに同じ DH 鍵を再利用しないようにします。これはセキュリティを向上させますが、より多くの計算機リソースを必要とします。このオプションはサーバソケットに適用されます。

バージョン 3.3 で追加.

ssl.OP_SINGLE_ECDH_USE

SSL セッションを区別するのに同じ ECDH 鍵を再利用しないようにします。これはセキュリティを向上させますが、より多くの計算機リソースを必要とします。このオプションはサーバソケットに適用されます。

バージョン 3.3 で追加.

ssl.OP_ENABLE_MIDDLEBOX_COMPAT

Send dummy Change Cipher Spec (CCS) messages in TLS 1.3 handshake to make a TLS 1.3 connection look more like a TLS 1.2 connection.

このオプションは OpenSSL 1.1.1 以降のみで使用できます。

バージョン 3.8 で追加.

ssl.OP_NO_COMPRESSION

SSL チャネルでの圧縮を無効にします。これはアプリケーションのプロトコルが自身の圧縮方法をサ

ポートする場合に有用です。

このオプションは OpenSSL 1.0.0 以降のみで使用できます。

バージョン 3.3 で追加.

class ssl.Options

OP_* 定数の *enum.IntFlag* コレクションです。

ssl.OP_NO_TICKET

クライアントサイドがセッションチケットをリクエストしないようにします。

バージョン 3.6 で追加.

ssl.OP_IGNORE_UNEXPECTED_EOF

Ignore unexpected shutdown of TLS connections.

このオプションは OpenSSL 3.0.0 以降のみで使用できます。

バージョン 3.10 で追加.

ssl.HAS_ALPN

OpenSSL ライブラリが、組み込みで **RFC 7301** で記述されている *Application-Layer Protocol Negotiation* TLS 拡張をサポートしているかどうか。

バージョン 3.5 で追加.

ssl.HAS_NEVER_CHECK_COMMON_NAME

Whether the OpenSSL library has built-in support not checking subject common name and *SSLContext.hostname_checks_common_name* is writeable.

バージョン 3.7 で追加.

ssl.HAS_ECDH

OpenSSL ライブラリが、組み込みの楕円曲線ディフィー・ヘルマン鍵共有をサポートしているかどうか。これは、ディストリビュータが明示的に無効にしていない限りは、真であるはずです。

バージョン 3.3 で追加.

ssl.HAS_SNI

OpenSSL ライブラリが、組み込みで (**RFC 6066** で記述されている) *Server Name Indication* 拡張をサポートしているかどうか。

バージョン 3.2 で追加.

ssl.HAS_NPN

OpenSSL ライブラリが、組み込みで、Application Layer Protocol Negotiation で記述されている *Next Protocol Negotiation* をサポートしているかどうか。true であれば、サポートしたいプロトコルを *SSLContext.set_npn_protocols()* メソッドで提示することができます。

バージョン 3.3 で追加.

ssl.HAS_SSLv2
: OpenSSL ライブラリが、組み込みで SSL 2.0 プロトコルをサポートしているかどうか。

  バージョン 3.7 で追加.

ssl.HAS_SSLv3
: OpenSSL ライブラリが、組み込みで SSL 3.0 プロトコルをサポートしているかどうか。

  バージョン 3.7 で追加.

ssl.HAS_TLSv1
: OpenSSL ライブラリが、組み込みで TLS 1.0 プロトコルをサポートしているかどうか。

  バージョン 3.7 で追加.

ssl.HAS_TLSv1_1
: OpenSSL ライブラリが、組み込みで TLS 1.1 プロトコルをサポートしているかどうか。

  バージョン 3.7 で追加.

ssl.HAS_TLSv1_2
: OpenSSL ライブラリが、組み込みで TLS 1.2 プロトコルをサポートしているかどうか。

  バージョン 3.7 で追加.

ssl.HAS_TLSv1_3
: OpenSSL ライブラリが、組み込みで TLS 1.3 プロトコルをサポートしているかどうか。

  バージョン 3.7 で追加.

ssl.CHANNEL_BINDING_TYPES
: サポートされている TLS のチャネルバインディングのタイプのリスト。リスト内の文字列は *SSLSocket.get_channel_binding()* の引数に渡せます。

  バージョン 3.3 で追加.

ssl.OPENSSL_VERSION
: インタプリタによってロードされた OpenSSL ライブラリのバージョン文字列:

```
>>> ssl.OPENSSL_VERSION
'OpenSSL 1.0.2k  26 Jan 2017'
```

  バージョン 3.2 で追加.

ssl.OPENSSL_VERSION_INFO
: OpenSSL ライブラリのバージョン情報を表す 5 つの整数のタプル:

```
>>> ssl.OPENSSL_VERSION_INFO
(1, 0, 2, 11, 15)
```

  バージョン 3.2 で追加.

ssl.OPENSSL_VERSION_NUMBER

1 つの整数の形式の、OpenSSL ライブラリの生のバージョン番号:

```
>>> ssl.OPENSSL_VERSION_NUMBER
268443839
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x100020bf'
```

バージョン 3.2 で追加.

ssl.ALERT_DESCRIPTION_HANDSHAKE_FAILURE

ssl.ALERT_DESCRIPTION_INTERNAL_ERROR

ALERT_DESCRIPTION_*

RFC 5246 その他からのアラートの種類です。IANA TLS Alert Registry にはこのリストとその意味が定義された RFC へのリファレンスが含まれています。

SSLContext.set_servername_callback() でのコールバック関数の戻り値として使われます。

バージョン 3.4 で追加.

class ssl.AlertDescription

ALERT_DESCRIPTION_* 定数の *enum.IntEnum* コレクションです。

バージョン 3.6 で追加.

Purpose.SERVER_AUTH

create_default_context() と *SSLContext.load_default_certs()* に渡すオプションです。この値はコンテキストが Web サーバの認証に使われることを示します (ですので、クライアントサイドのソケットを作るのに使うことになるでしょう)。

バージョン 3.4 で追加.

Purpose.CLIENT_AUTH

create_default_context() と *SSLContext.load_default_certs()* に渡すオプションです。この値はコンテキストが Web クライアントの認証に使われることを示します (ですので、サーバサイドのソケットを作るのに使うことになるでしょう)。

バージョン 3.4 で追加.

class ssl.SSLErrorNumber

SSL_ERROR_* 定数の *enum.IntEnum* コレクションです。

バージョン 3.6 で追加.

class ssl.TLSVersion

enum.IntEnum collection of SSL and TLS versions for *SSLContext.maximum_version* and *SSLContext.minimum_version*.

バージョン 3.7 で追加.

TLSVersion.MINIMUM_SUPPORTED

TLSVersion.MAXIMUM_SUPPORTED
: The minimum or maximum supported SSL or TLS version. These are magic constants. Their values don't reflect the lowest and highest available TLS/SSL versions.

TLSVersion.SSLv3

TLSVersion.TLSv1

TLSVersion.TLSv1_1

TLSVersion.TLSv1_2

TLSVersion.TLSv1_3
: SSL 3.0 to TLS 1.3.

18.3.2 SSL ソケット

class ssl.SSLSocket(*socket.socket*)
: SSL ソケットは *socket* **オブジェクト** の以下のメソッドを提供します:

- *accept()*
- *bind()*
- *close()*
- *connect()*
- *detach()*
- *fileno()*
- *getpeername()*, *getsockname()*
- *getsockopt()*, *setsockopt()*
- *gettimeout()*, *settimeout()*, *setblocking()*
- *listen()*
- *makefile()*
- *recv()*, *recv_into()* (非ゼロの `flags` は渡せません)
- *send()*, *sendall()* (非ゼロの `flags` は渡せません)
- *sendfile()* (ただし、*os.sendfile* は平文ソケットにのみ使用されます。それ以外の場合には、*send()* が使用されます。)
- *shutdown()*

SSL(および TLS) プロトコルは TCP の上に独自の枠組みを持っているので、SSL ソケットの抽象化は、いくつかの点で通常の OS レベルのソケットの仕様から逸脱することがあります。特に **ノンブロッキングソケットについての注釈** を参照してください。

SSLSocket のインスタンスは *SSLContext.wrap_socket()* メソッドを使用して作成されなければなりません。

バージョン 3.5 で変更: `sendfile()` メソッドが追加されました。

バージョン 3.5 で変更: `shutdown()` は、バイトが送受信されるたびにソケットのタイムアウトをリセットしません。ソケットのタイムアウトは、シャットダウンの最大合計時間になりました。

バージョン 3.6 で非推奨: *SSLSocket* インスタンスを直接作成することは非推奨です。ソケットをラップするために *SSLContext.wrap_socket()* を使用してください。

バージョン 3.7 で変更: *SSLSocket* instances must to created with *wrap_socket()*. In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

SSL ソケットには、以下に示す追加のメソッドと属性もあります:

`SSLSocket.read`(*len=1024*, *buffer=None*)

SSL ソケットからデータの *len* バイトまでを読み出し、読み出した結果を `bytes` インスタンスで返します。*buffer* を指定すると、結果は代わりに buffer に読み込まれ、読み込んだバイト数を返します。

ソケットが *non-blocking* で読み出しがブロックすると、*SSLWantReadError* もしくは *SSLWantWriteError* が送出されます。

再ネゴシエーションがいつでも可能なので、*read()* の呼び出しは書き込み操作も引き起こしえます。

バージョン 3.5 で変更: ソケットのタイムアウトは、バイトが送受信されるたびにリセットされません。ソケットのタイムアウトは、最大 *len* バイトを読むのにかかる最大合計時間になりました。

バージョン 3.6 で非推奨: *read()* の代わりに `recv()` を使用してください。

`SSLSocket.write`(*buf*)

buf を SSL ソケットに書き込み、書き込んだバイト数を返します。*buf* 引数はバッファインターフェイスをサポートするオブジェクトでなければなりません。

ソケットが *non-blocking* で書き込みがブロックすると、*SSLWantReadError* もしくは *SSLWantWriteError* が送出されます。

再ネゴシエーションがいつでも可能なので、*write()* の呼び出しは読み出し操作も引き起こしえます。

バージョン 3.5 で変更: ソケットのタイムアウトは、バイトが送受信されるたびにリセットされません。ソケットのタイムアウトは、*buf* を書き込むのにかかる最大合計時間になりました。

バージョン 3.6 で非推奨: *write()* の代わりに `send()` を使用してください。

注釈: *read()*, *write()* メソッドは下位レベルのメソッドであり、暗号化されていないアプリケーションレベルのデータを読み書きし、それを復号/暗号化して暗号化された書き込みレベルのデータにします。これら

のメソッドはアクティブな SSL 接続つまり、ハンドシェイクが完了していて、*SSLSocket.unwrap()* が呼ばれていないことを必要とします。

通常はこれらのメソッドの代わりに *recv()* や *send()* のようなソケット API メソッドを使うべきです。

SSLSocket.**do_handshake**()

SSL セットアップのハンドシェイクを実行します。

バージョン 3.4 で変更: ソケットの *context* の属性 *check_hostname* が真の場合に、ハンドシェイクメソッドが *match_hostname()* を実行するようになりました。

バージョン 3.5 で変更: ソケットのタイムアウトは、バイトが送受信されるたびにリセットされません。ソケットのタイムアウトは、ハンドシェイクにかかる最大合計時間になりました。

バージョン 3.7 で変更: Hostname or IP address is matched by OpenSSL during handshake. The function *match_hostname()* is no longer used. In case OpenSSL refuses a hostname or IP address, the handshake is aborted early and a TLS alert message is send to the peer.

SSLSocket.**getpeercert**(*binary_form=False*)

接続先に証明書が無い場合、None を返します。SSL ハンドシェイクがまだ行われていない場合は、*ValueError* が送出されます。

binary_form が *False* で接続先から証明書を取得した場合、このメソッドは *dict* のインスタンスを返します。証明書が認証されていない場合、辞書は空です。証明書が認証されていた場合いくつかのキーを持った辞書を返し、subject (証明書が発行された principal), issuer (証明書を発行した principal) を含みます。証明書が *Subject Alternative Name* 拡張 (**RFC 3280** を参照) のインスタンスを格納していた場合、subjectAltName キーも辞書に含まれます。

subject, issuer フィールドは、証明書のそれぞれのフィールドについてのデータ構造で与えられる RDN (relative distinguishued name) のシーケンスを格納したタプルで、各 RDN は name-value ペアのシーケンスです。現実世界での例をお見せします:

```
{'issuer': ((('countryName', 'IL'),),
            (('organizationName', 'StartCom Ltd.'),),
            (('organizationalUnitName',
              'Secure Digital Certificate Signing'),),
            (('commonName',
              'StartCom Class 2 Primary Intermediate Server CA'),)),
 'notAfter': 'Nov 22 08:15:19 2013 GMT',
 'notBefore': 'Nov 21 03:09:52 2011 GMT',
 'serialNumber': '95F0',
 'subject': ((('description', '571208-SLe257oHY9fVQ07Z'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'San Francisco'),),
             (('organizationName', 'Electronic Frontier Foundation, Inc.'),),
             (('commonName', '*.eff.org'),),
             (('emailAddress', 'hostmaster@eff.org'),)),
 'subjectAltName': (('DNS', '*.eff.org'), ('DNS', 'eff.org')),
 'version': 3}
```

注釈: 特定のサービスのために証明書の検証がしたければ、*match_hostname()* 関数を使うことができます。

binary_form 引数が *True* だった場合、証明書が渡されていればこのメソッドは DER エンコードされた証明書全体をバイト列として返し、接続先が証明書を提示しなかった場合は *None* を返します。接続先が証明書を提供するかどうかは SSL ソケットの役割に依存します:

- クライアント SSL ソケットでは、認証が要求されているかどうかに関わらず、サーバは常に証明書を提供します。
- サーバ SSL ソケットでは、クライアントはサーバによって認証が要求されている場合にのみ証明書を提供します。したがって、(*CERT_OPTIONAL* や *CERT_REQUIRED* ではなく) *CERT_NONE* を使用した場合 *getpeercert()* は *None* を返します。

バージョン 3.2 で変更: 返される辞書に issuer, notBefore のような追加アイテムを含むようになりました。

バージョン 3.4 で変更: ハンドシェイクが済んでいなければ *ValueError* を投げるようになりました。返される辞書に crlDistributionPoints, caIssuers, OCSP URI のような X509v3 拡張アイテムを含むようになりました。

バージョン 3.8.1 で変更: IPv6 address strings no longer have a trailing new line.

SSLSocket.**cipher**()

利用されている暗号の名前、その暗号の利用を定義している SSL プロトコルのバージョン、利用されている鍵の bit 長の 3 つの値を含むタプルを返します。もし接続が確立されていない場合、None を返します。

SSLSocket.**shared_ciphers**()

ハンドシェイク中にクライアントにより共有される暗号方式のリストを返します。返されるリストの各要素は 3 つの値を含むタプルで、その値はそれぞれ、暗号方式の名前、その暗号の利用を定義している SSL プロトコルのバージョン、暗号で使用される秘密鍵のビット長です。接続が確立されていないか、ソケットがクライアントソケットである場合、meth:*~SSLSocket.shared_ciphers* は None を返します。

バージョン 3.5 で追加.

SSLSocket.**compression**()

使われている圧縮アルゴリズムを文字列で返します。接続が圧縮されていなければ None を返します。

上位レベルのプロトコルが自身で圧縮メカニズムをサポートする場合、SSL レベルでの圧縮を *OP_NO_COMPRESSION* を使って無効にできます。

バージョン 3.3 で追加.

SSLSocket.**get_channel_binding**(*cb_type="tls-unique"*)

現在の接続におけるチャネルバインディングのデータを取得します。未接続あるいはハンドシェイクが完了していなければ None を返します。

cb_type パラメータにより、望みのチャネルバインディングのタイプを選択できます。チャネルバインディングのタイプの妥当なものは *CHANNEL_BINDING_TYPES* でリストされています。現在のところは **RFC 5929** で定義されている 'tls-unique' のみがサポートされています。未サポートのチャネルバインディングのタイプが要求された場合、*ValueError* を送出します。

バージョン 3.3 で追加.

SSLSocket.**selected_alpn_protocol**()

TLS ハンドシェイクで選択されたプロトコルを返します。*SSLContext.set_alpn_protocols()* が呼ばれていない場合、相手側が ALPN をサポートしていない場合、クライアントが提案したプロトコルのどれもソケットがサポートしない場合、あるいはハンドシェイクがまだ行われていない場合には、None が返されます。

バージョン 3.5 で追加.

SSLSocket.**selected_npn_protocol**()

TLS/SSL ハンドシェイクで選択された上位レベルのプロトコルを返します。*SSLContext.set_npn_protocols()* が呼ばれていない場合、相手側が NPN をサポートしていない場合、あるいはハンドシェイクがまだ行われていない場合には、None が返されます。

バージョン 3.3 で追加.

SSLSocket.**unwrap**()

SSL シャットダウンハンドシェイクを実行します。これは下位レイヤーのソケットから TLS レイヤーを取り除き、下位レイヤーのソケットオブジェクトを返します。これは暗号化されたオペレーションから暗号化されていない接続に移行するときに利用されます。以降の通信には、オリジナルのソケットではなくこのメソッドが返したソケットのみを利用するべきです。

SSLSocket.**verify_client_post_handshake**()

Requests post-handshake authentication (PHA) from a TLS 1.3 client. PHA can only be initiated for a TLS 1.3 connection from a server-side socket, after the initial TLS handshake and with PHA enabled on both sides, see *SSLContext.post_handshake_auth*.

The method does not perform a cert exchange immediately. The server-side sends a CertificateRequest during the next write event and expects the client to respond with a certificate on the next read event.

If any precondition isn't met (e.g. not TLS 1.3, PHA not enabled), an *SSLError* is raised.

注釈: Only available with OpenSSL 1.1.1 and TLS 1.3 enabled. Without TLS 1.3 support, the method raises *NotImplementedError*.

バージョン 3.8 で追加.

SSLSocket.**version**()

コネクションによって実際にネゴシエイトされた SSL プロトコルバージョンを文字列で、または、セキュアなコネクションが確立していなければ None を返します。これを書いている時点では、"SSLv2",

"SSLv3", "TLSv1", "TLSv1.1", "TLSv1.2" などが返ります。最新の OpenSSL はもっと色々な値を定義しているかもしれません。

バージョン 3.5 で追加.

SSLSocket.**pending**()
接続において既に復号済みで読み出し可能で保留になっているバイト列の数を返します。

SSLSocket.**context**
この SSL ソケットに結び付けられた *SSLContext* オブジェクトです。SSL ソケットが (*SSLContext.wrap_socket()* ではなく) 非推奨の *wrap_socket()* 関数を使って作られた場合、これはこの SSL ソケットのために作られたカスタムコンテキストオブジェクトです。

バージョン 3.2 で追加.

SSLSocket.**server_side**
サーバサイドのソケットに対して True 、クライアントサイドのソケットに対して False となる真偽値です。

バージョン 3.2 で追加.

SSLSocket.**server_hostname**
サーバのホスト名: *str* 型、またはサーバサイドのソケットの場合とコンストラクタで hostname が指定されなかった場合は None

バージョン 3.2 で追加.

バージョン 3.7 で変更: The attribute is now always ASCII text. When server_hostname is an internationalized domain name (IDN), this attribute now stores the A-label form ("xn--pythn-mua.org"), rather than the U-label form ("pythön.org").

SSLSocket.**session**
この SSL 接続に対する *SSLSession* です。このセッションは、TLS ハンドシェイクの実行後、クライアントサイドとサーバサイドのソケットで使用できます。クライアントソケットでは、このセッションを *do_handshake()* が呼ばれる前に設定して、セッションを再利用できます。

バージョン 3.6 で追加.

SSLSocket.**session_reused**
バージョン 3.6 で追加.

18.3.3 SSL コンテキスト

バージョン 3.2 で追加.

SSL コンテキストは、SSL 構成オプション、証明書 (群) や秘密鍵 (群) などのような、一回の SSL 接続よりも長生きするさまざまなデータを保持します。これはサーバサイドソケットの SSL セッションのキャッシュも管理し、同じクライアントからの繰り返しの接続時の速度向上に一役買います。

class ssl.SSLContext(*protocol=PROTOCOL_TLS*)

Create a new SSL context. You may pass *protocol* which must be one of the `PROTOCOL_*` constants defined in this module. The parameter specifies which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server's choice. Most of the versions are not interoperable with the other versions. If not specified, the default is *PROTOCOL_TLS*; it provides the most compatibility with other versions.

次のテーブルは、どのクライアントのバージョンがどのサーバのバージョンに接続できるかを示しています:

client / **server**	**SSLv2**	**SSLv3**	**TLS**[*3]	**TLSv1**	**TLSv1.1**	**TLSv1.2**
SSLv2	yes	no	no[*1]	no	no	no
SSLv3	no	yes	no[*2]	no	no	no
TLS (SSLv23)[*3]	no[*1]	no[*2]	yes	yes	yes	yes
TLSv1	no	no	yes	yes	no	no
TLSv1.1	no	no	yes	no	yes	no
TLSv1.2	no	no	yes	no	no	yes

脚注

参考:

create_default_context() は *ssl* モジュールに、目的に合ったセキュリティ設定を選ばせます。

バージョン 3.6 で変更: このコンテキストは、安全性の高いデフォルト値で作成されます。デフォルト設定されるオプションは、*OP_NO_COMPRESSION*, *OP_CIPHER_SERVER_PREFERENCE*, *OP_SINGLE_DH_USE*, *OP_SINGLE_ECDH_USE*, *OP_NO_SSLv2* (*PROTOCOL_SSLv2* 以外), *OP_NO_SSLv3* (*PROTOCOL_SSLv3* 以外) です。初期の暗号方式スイートリストには `HIGH` 暗号のみが含まれており、`NULL` 暗号および `MD5` 暗号は含まれません (*PROTOCOL_SSLv2* 以外)。

SSLContext オブジェクトは以下のメソッドと属性を持っています:

SSLContext.cert_store_stats()

ロードされた X.509 証明書の数、CA 証明書で活性の X.509 証明書の数、証明書失効リストの数、についての統計情報を辞書として取得します。

一つの CA と他の一つの証明書を持ったコンテキストでの例です:

```
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

バージョン 3.4 で追加.

[*3] TLS 1.3 protocol will be available with *PROTOCOL_TLS* in OpenSSL >= 1.1.1. There is no dedicated PROTOCOL constant for just TLS 1.3.

[*1] *SSLContext* では、デフォルトで *OP_NO_SSLv2* により SSLv2 が無効になっています。

[*2] *SSLContext* では、デフォルトで *OP_NO_SSLv3* により SSLv3 が無効になっています。

SSLContext.load_cert_chain(*certfile*, *keyfile=None*, *password=None*)

秘密鍵と対応する証明書をロードします。*certfile* は、証明書と、証明書認証で必要とされる任意の数の CA 証明書を含む、PEM フォーマットの単一ファイルへのパスでなければなりません。*keyfile* 文字列を指定する場合、秘密鍵が含まれるファイルを指すものでなければなりません。指定しない場合、秘密鍵も *certfile* から取得されます。*certfile* への証明書の格納についての詳細は、**証明書** の議論を参照してください。

password 引数に、秘密鍵を復号するためのパスワードを返す関数を与えることができます。その関数は秘密鍵が暗号化されていて、なおかつパスワードが必要な場合にのみ呼び出されます。その関数は引数なしで呼び出され、string, bytes, または bytearray を返さなければなりません。戻り値が string の場合は鍵を復号化するのに使う前に UTF-8 でエンコードされます。string の代わりに bytes や bytearray を返した場合は *password* 引数に直接供給されます。秘密鍵が暗号化されていなかったりパスワードを必要としない場合は、指定は無視されます。

password が与えられず、そしてパスワードが必要な場合には、OpenSSL 組み込みのパスワード問い合わせメカニズムが、ユーザに対話的にパスワードを問い合わせます。

秘密鍵が証明書に合致しなければ、*SSLError* が送出されます。

バージョン 3.3 で変更: 新しいオプション引数 *password*。

SSLContext.load_default_certs(*purpose=Purpose.SERVER_AUTH*)

デフォルトの場所から ” 認証局” (CA=certification authority) 証明書ファイル一式をロードします。Windows では、CA 証明書はシステム記憶域の CA と ROOT からロードします。それ以外のシステムでは、この関数は *SSLContext.set_default_verify_paths()* を呼び出します。将来的にはこのメソッドは、他の場所からも CA 証明書をロードするかもしれません。

purpose フラグでどの種類の CA 証明書をロードするかを指定します。デフォルトの *Purpose.SERVER_AUTH* は TLS web サーバの認証のために活性かつ信頼された証明書をロードします (クライアントサイドのソケット)。*Purpose.CLIENT_AUTH* はクライアント証明書の正当性検証をサーバサイドで行うための CA 証明書をロードします。

バージョン 3.4 で追加.

SSLContext.load_verify_locations(*cafile=None*, *capath=None*, *cadata=None*)

verify_mode が *CERT_NONE* でない場合に接続先の証明書ファイルの正当性検証に使われる ” 認証局” (CA=certification authority) 証明書ファイル一式をロードします。少なくとも *cafile* か *capath* のどちらかは指定しなければなりません。

このメソッドは PEM または DER フォーマットの証明書失効リスト (CRLs=certification revocation lists) もロードできます。CRLs のために使うには、*SSLContext.verify_flags* を適切に設定しなければなりません。

cafile を指定する場合は、PEM フォーマットで CA 証明書が結合されたファイルへのパスを指定してください。このファイル内で証明書をどのように編成すれば良いのかについての詳しい情報については、**証明書** の議論を参照してください。

capath を指定する場合は、PEM フォーマットの CA 証明書が含まれる、‘OpenSSL specific layout‘_ に従ったディレクトリへのパスを指定してください。

cadata オブジェクトを指定する場合は、PEM エンコードの証明書一つ以上の ASCII 文字列か、DER エンコードの証明書の *bytes-like object* オブジェクトのどちらかを指定してください。PEM エンコードの証明書の周囲の余分な行は無視されますが、少なくとも一つの証明書が含まれている必要があります。

バージョン 3.4 で変更: 新しいオプション引数 *cadata* 。

SSLContext.**get_ca_certs**(*binary_form=False*)

ロードされた ” 認証局” (CA=certification authority) 証明書のリストを取得します。binary_form 引数が *False* である場合、リストのそれぞれのエントリは *SSLSocket.getpeercert()* が出力するような辞書になります。True である場合、このメソッドは、DER エンコード形式の証明書のリストを返します。返却されるリストには、SSL 接続によって証明書がリクエストおよびロードされない限り、*capath* からの証明書は含まれません。

注釈: capath ディレクトリ内の証明書は一度でも使われない限りはロードされません。

バージョン 3.4 で追加.

SSLContext.**get_ciphers**()

有効な暗号化のリストを取得します。リストは暗号化優先度順に並びます。*SSLContext.set_ciphers()* を参照してください。

以下はプログラム例です:

```
>>> ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> ctx.set_ciphers('ECDHE+AESGCM:!ECDSA')
>>> ctx.get_ciphers()  # OpenSSL 1.0.x
[{'alg_bits': 256,
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'id': 50380848,
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 256},
 {'alg_bits': 128,
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
  'id': 50380847,
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 128}]
```

OpenSSL 1.1 以降では、暗号化辞書に以下のフィールドが追加されました。

```
>>> ctx.get_ciphers()  # OpenSSL 1.1+
[{'aead': True,
  'alg_bits': 256,
  'auth': 'auth-rsa',
```

(次のページに続く)

(前のページからの続き)

```
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'digest': None,
  'id': 50380848,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1.2',
  'strength_bits': 256,
  'symmetric': 'aes-256-gcm'},
 {'aead': True,
  'alg_bits': 128,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
  'digest': None,
  'id': 50380847,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1.2',
  'strength_bits': 128,
  'symmetric': 'aes-128-gcm'}]
```

利用可能な環境: OpenSSL 1.0.2 以上

バージョン 3.6 で追加.

SSLContext.set_default_verify_paths()

デフォルトの ” 認証局” (CA=certification authority) 証明書を、OpenSSL ライブラリがビルドされた際に定義されたファイルシステム上のパスからロードします。残念ながらこのメソッドが成功したかどうかを知るための簡単な方法はありません: 証明書が見つからなくてもエラーは返りません。OpenSSL ライブラリがオペレーティングシステムの一部として提供されている際にはどうやら適切に構成できるようですが。

SSLContext.set_ciphers(*ciphers*)

このコンテキストによって作られるソケットで利用できる暗号を設定します。OpenSSL cipher list format に書かれている形式の文字列でなければなりません。(OpenSSL のコンパイル時オプションや他の設定がそれらすべての暗号の使用を禁止しているなどの理由で) どの暗号も選べない場合、*SSLError* が送出されます。

注釈: 接続時に SSL ソケットの *SSLSocket.cipher()* メソッドが、現在選択されているその暗号を使います。

OpenSSL 1.1.1 has TLS 1.3 cipher suites enabled by default. The suites cannot be disabled with *set_ciphers()*.

SSLContext.set_alpn_protocols(*protocols*)

SSL/TLS ハンドシェイク時にソケットが提示すべきプロトコルを指定します。['http/1.1', 'spdy/

2'] のような推奨順に並べた ASCII 文字列のリストでなければなりません。プロトコルの選択は **RFC 7301** に従いハンドシェイク中に行われます。ハンドシェイクが正常に終了した後、*SSLSocket.selected_alpn_protocol()* メソッドは合意されたプロトコルを返します。

このメソッドは *HAS_ALPN* が `False` の場合 *NotImplementedError* を送出します。

OpenSSL 1.1.0 to 1.1.0e will abort the handshake and raise *SSLError* when both sides support ALPN but cannot agree on a protocol. 1.1.0f+ behaves like 1.0.2, *SSLSocket.selected_alpn_protocol()* returns None.

バージョン 3.5 で追加.

`SSLContext.set_npn_protocols`(*protocols*)

SSL/TLS ハンドシェイク時にソケットが提示すべきプロトコルを指定します。['http/1.1', 'spdy/2'] のような推奨順に並べた文字列のリストでなければなりません。プロトコルの選択は Application Layer Protocol Negotiation に従いハンドシェイク中に行われます。ハンドシェイクが正常に終了した後、*SSLSocket.selected_alpn_protocol()* メソッドは合意されたプロトコルを返します。

このメソッドは *HAS_NPN* が `False` の場合 *NotImplementedError* を送出します。

バージョン 3.3 で追加.

`SSLContext.sni_callback`

TLS クライアントがサーバ名表示を指定した際の、SSL/TLS サーバによって TLS Client Hello ハンドシェイクメッセージが受け取られたあとで呼び出されるコールバック関数を登録します。サーバ名表示メカニズムは **RFC 6066** セクション 3 - Server Name Indication で述べられています。

`SSLContext` ごとに一つだけコールバックをセットできます。*sni_callback* を `None` にすればコールバックは無効になります。この関数を続けて呼ぶと、以前に登録されたコールバックを上書きします。

The callback function will be called with three arguments; the first being the *ssl.SSLSocket*, the second is a string that represents the server name that the client is intending to communicate (or *None* if the TLS Client Hello does not contain a server name) and the third argument is the original *SSLContext*. The server name argument is text. For internationalized domain name, the server name is an IDN A-label (`"xn--pythn-mua.org"`).

このコールバックの典型的な利用方法は、*ssl.SSLSocket* の *SSLSocket.context* 属性を、サーバ名に合致する証明書チェインを持つ新しい *SSLContext* オブジェクトに変更することです。

TLS 接続の初期ネゴシエーションのフェーズなので、*SSLSocket.selected_alpn_protocol()*, *SSLSocket.context* のような限られたメソッドと属性のみ使えます。*SSLSocket.getpeercert()*, *SSLSocket.getpeercert()*, *SSLSocket.cipher()*, `SSLSocket.compress()` メソッドは TLS 接続が TLS Client Hello よりも先に進行していることを必要としますから、これらは意味のある値を返しませんし、安全に呼び出すこともできません。

TLS ネゴシエーションを継続させるならば、*sni_callback* 関数は `None` を返さなければなりません。TLS が失敗することを必要とするなら、constant *ALERT_DESCRIPTION_** を返してください。ここにない値を返すと、致命エラー *ALERT_DESCRIPTION_INTERNAL_ERROR* を引き起こします。

sni_callback 関数が例外を送出した場合、TLS 接続は TLS の致命的アラートメッセージ *ALERT_DESCRIPTION_HANDSHAKE_FAILURE* とともに終了します。

このメソッドは OpenSSL ライブラリが OPENSSL_NO_TLSEXT を定義してビルドされている場合、*NotImplementedError* を送出します。

バージョン 3.7 で追加.

SSLContext.**set_servername_callback**(*server_name_callback*)

This is a legacy API retained for backwards compatibility. When possible, you should use *sni_callback* instead. The given *server_name_callback* is similar to *sni_callback*, except that when the server hostname is an IDN-encoded internationalized domain name, the *server_name_callback* receives a decoded U-label (`"pythön.org"`).

サーバ名に対するデコードエラーがあれば、TLS 接続はクライアントに対する TLS の致命的アラートメッセージ *ALERT_DESCRIPTION_INTERNAL_ERROR* とともに終了します。

バージョン 3.4 で追加.

SSLContext.**load_dh_params**(*dhfile*)

ディフィー・ヘルマン (DH) 鍵交換のための鍵生成パラメータをロードします。DH 鍵交換を用いることは、(サーバ、クライアントともに) 計算機リソースに高い処理負荷をかけますがセキュリティを向上させます。*dhfile* パラメータは PEM フォーマットの DH パラメータを含んだファイルへのパスでなければなりません。

この設定はクライアントソケットには適用されません。さらにセキュリティを改善するのに *OP_SINGLE_DH_USE* オプションも利用できます。

バージョン 3.3 で追加.

SSLContext.**set_ecdh_curve**(*curve_name*)

楕円曲線ディフィー・ヘルマン (ECDH) 鍵交換の曲線名を指定します。ECDH はもとの DH に較べて、ほぼ間違いなく同程度に安全である一方で、顕著に高速です。*curve_name* パラメータは既知の楕円曲線を表す文字列でなければなりません。例えば `prime256v1` が広くサポートされている曲線です。

この設定はクライアントソケットには適用されません。さらにセキュリティを改善するのに *OP_SINGLE_ECDH_USE* オプションも利用できます。

このメソッドは *HAS_ECDH* が `False` の場合は利用できません。

バージョン 3.3 で追加.

参考:

'SSL/TLS & Perfect Forward Secrecy '_ Vincent Bernat.

SSLContext.**wrap_socket**(*sock*, *server_side=False*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *server_hostname=None*, *session=None*)

Wrap an existing Python socket *sock* and return an instance of *SSLContext.sslsocket_class*

(default *SSLSocket*). The returned SSL socket is tied to the context, its settings and certificates. *sock* must be a *SOCK_STREAM* socket; other socket types are unsupported.

server_side 引数は真偽値で、このソケットがサーバサイドとクライアントサイドのどちらの動作をするのかを指定します。

クライアントサイドソケットにおいて、コンテキストの生成は遅延されます。つまり、低レイヤのソケットがまだ接続されていない場合、コンテキストの生成はそのソケットの connect() メソッドが呼ばれた後に行われます。サーバサイドソケットの場合、そのソケットに接続先が居なければそれは listen 用ソケットだと判断されます。accept() メソッドで生成されるクライアント接続に対してのサーバサイド SSL ラップは自動的に行われます。メソッドは *SSLError* を送出することがあります。

クライアントからの接続では、*server_hostname* で接続先サービスのホスト名を指定できます。これは HTTP バーチャルホストにかなり似て、シングルサーバで複数の SSL ベースのサービスを別々の証明書でホストしているようなサーバに対して使えます。*server_side* が True の場合に *server_hostname* を指定すると *ValueError* を送出します。

do_handshake_on_connect 引数は、socket.connect() の後に自動的に SSL ハンドシェイクを行うか、それともアプリケーションが明示的に *SSLSocket.do_handshake()* メソッドを実行するかを指定します。*SSLSocket.do_handshake()* を明示的に呼びだすことで、ハンドシェイクによるソケット I/O のブロッキング動作を制御できます。

suppress_ragged_eofs 引数は、SSLSocket.recv() メソッドが、接続先から予期しない EOF を受け取った時に通知する方法を指定します。*True* (デフォルト) の場合、下位のソケットレイヤーから予期せぬ EOF エラーが来た場合、通常の EOF (空のバイト列オブジェクト) を返します。*False* の場合、呼び出し元に例外を投げて通知します。

session, *session* を参照してください。

バージョン 3.5 で変更: OpenSSL が SNI をサポートしなくても server_hostname を許容するようになりました。

バージョン 3.6 で変更: *session* 引数が追加されました。

バージョン 3.7 で変更: The method returns on instance of *SSLContext.sslsocket_class* instead of hard-coded *SSLSocket*.

SSLContext.sslsocket_class

The return type of *SSLContext.wrap_socket()*, defaults to *SSLSocket*. The attribute can be overridden on instance of class in order to return a custom subclass of *SSLSocket*.

バージョン 3.7 で追加.

SSLContext.wrap_bio(*incoming*, *outgoing*, *server_side=False*, *server_hostname=None*, *session=None*)

Wrap the BIO objects *incoming* and *outgoing* and return an instance of *SSLContext.sslobject_class* (default *SSLObject*). The SSL routines will read input data from the incoming BIO and write data to the outgoing BIO.

server_side、*server_hostname* 、*session* 引数は、*SSLContext.wrap_socket()* での意味と同じ意

味を持ちます。

バージョン 3.6 で変更: *session* 引数が追加されました。

バージョン 3.7 で変更: The method returns on instance of *SSLContext.sslobject_class* instead of hard-coded *SSLObject*.

SSLContext.sslobject_class

The return type of *SSLContext.wrap_bio()*, defaults to *SSLObject*. The attribute can be overridden on instance of class in order to return a custom subclass of *SSLObject*.

バージョン 3.7 で追加.

SSLContext.session_stats()

Get statistics about the SSL sessions created or managed by this context. A dictionary is returned which maps the names of each piece of information to their numeric values. For example, here is the total number of hits and misses in the session cache since the context was created:

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

SSLContext.check_hostname

Whether to match the peer cert's hostname in *SSLSocket.do_handshake()*. The context's *verify_mode* must be set to *CERT_OPTIONAL* or *CERT_REQUIRED*, and you must pass *server_hostname* to *wrap_socket()* in order to match the hostname. Enabling hostname checking automatically sets *verify_mode* from *CERT_NONE* to *CERT_REQUIRED*. It cannot be set back to *CERT_NONE* as long as hostname checking is enabled. The *PROTOCOL_TLS_CLIENT* protocol enables hostname checking by default. With other protocols, hostname checking must be enabled explicitly.

以下はプログラム例です:

```
import socket, ssl

context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = context.wrap_socket(s, server_hostname='www.verisign.com')
ssl_sock.connect(('www.verisign.com', 443))
```

バージョン 3.4 で追加.

バージョン 3.7 で変更: *verify_mode* is now automatically changed to *CERT_REQUIRED* when hostname checking is enabled and *verify_mode* is *CERT_NONE*. Previously the same operation would have failed with a *ValueError*.

注釈: この機能には OpenSSL0.9.8f 以降が必要です。

SSLContext.**keylog_filename**

Write TLS keys to a keylog file, whenever key material is generated or received. The keylog file is designed for debugging purposes only. The file format is specified by NSS and used by many traffic analyzers such as Wireshark. The log file is opened in append-only mode. Writes are synchronized between threads, but not between processes.

バージョン 3.8 で追加.

注釈: この機能には OpenSSL 1.1.1 以降が必要です。

SSLContext.**maximum_version**

A *TLSVersion* enum member representing the highest supported TLS version. The value defaults to *TLSVersion.MAXIMUM_SUPPORTED*. The attribute is read-only for protocols other than *PROTOCOL_TLS*, *PROTOCOL_TLS_CLIENT*, and *PROTOCOL_TLS_SERVER*.

The attributes *maximum_version*, *minimum_version* and *SSLContext.options* all affect the supported SSL and TLS versions of the context. The implementation does not prevent invalid combination. For example a context with *OP_NO_TLSv1_2* in *options* and *maximum_version* set to *TLSVersion.TLSv1_2* will not be able to establish a TLS 1.2 connection.

注釈: This attribute is not available unless the ssl module is compiled with OpenSSL 1.1.0g or newer.

バージョン 3.7 で追加.

SSLContext.**minimum_version**

Like *SSLContext.maximum_version* except it is the lowest supported version or *TLSVersion.MINIMUM_SUPPORTED*.

注釈: This attribute is not available unless the ssl module is compiled with OpenSSL 1.1.0g or newer.

バージョン 3.7 で追加.

SSLContext.**num_tickets**

Control the number of TLS 1.3 session tickets of a `TLS_PROTOCOL_SERVER` context. The setting has no impact on TLS 1.0 to 1.2 connections.

注釈: This attribute is not available unless the ssl module is compiled with OpenSSL 1.1.1 or newer.

バージョン 3.8 で追加.

SSLContext.options

このコンテキストで有効になっている SSL オプションを表す整数。デフォルトの値は *OP_ALL* ですが、*OP_NO_SSLv2* のような他の値をビット OR 演算で指定できます。

注釈: With versions of OpenSSL older than 0.9.8m, it is only possible to set options, not to clear them. Attempting to clear an option (by resetting the corresponding bits) will raise a *ValueError*.

バージョン 3.6 で変更: *SSLContext.options* は次のように *Options* のフラグを返します。

```
>>> ssl.create_default_context().options
<Options.OP_ALL|OP_NO_SSLv3|OP_NO_SSLv2|OP_NO_COMPRESSION: 2197947391>
```

SSLContext.post_handshake_auth

Enable TLS 1.3 post-handshake client authentication. Post-handshake auth is disabled by default and a server can only request a TLS client certificate during the initial handshake. When enabled, a server may request a TLS client certificate at any time after the handshake.

When enabled on client-side sockets, the client signals the server that it supports post-handshake authentication.

When enabled on server-side sockets, *SSLContext.verify_mode* must be set to *CERT_OPTIONAL* or *CERT_REQUIRED*, too. The actual client cert exchange is delayed until *SSLSocket.verify_client_post_handshake()* is called and some I/O is performed.

注釈: Only available with OpenSSL 1.1.1 and TLS 1.3 enabled. Without TLS 1.3 support, the property value is None and can't be modified

バージョン 3.8 で追加.

SSLContext.protocol

コンテキストの構築時に選択されたプロトコルバージョン。この属性は読み出し専用です。

SSLContext.hostname_checks_common_name

Whether *check_hostname* falls back to verify the cert's subject common name in the absence of a subject alternative name extension (default: true).

注釈: Only writeable with OpenSSL 1.1.0 or higher.

バージョン 3.7 で追加.

バージョン 3.9.3 で変更: The flag had no effect with OpenSSL before version 1.1.1k. Python 3.8.9, 3.9.3, and 3.10 include workarounds for previous versions.

SSLContext.**verify_flags**

証明書の検証操作のためのフラグです。*VERIFY_CRL_CHECK_LEAF* などのフラグをビット OR 演算でセットできます。デフォルトでは OpenSSL は証明書失効リスト (CRLs) を必要としませんし検証にも使いません。openssl version 0.9.8+ でのみ利用可能です。

バージョン 3.4 で追加.

バージョン 3.6 で変更: *SSLContext.verify_flags* は次のように *VerifyFlags* のフラグを返します。

```
>>> ssl.create_default_context().verify_flags
<VerifyFlags.VERIFY_X509_TRUSTED_FIRST: 32768>
```

SSLContext.**verify_mode**

接続先の証明書の検証を試みるかどうか、また、検証が失敗した場合にどのように振舞うべきかを制御します。この属性は *CERT_NONE*, *CERT_OPTIONAL*, *CERT_REQUIRED* のうちどれか一つでなければなりません。

バージョン 3.6 で変更: *SSLContext.verify_mode* は次のように *VerifyMode* enum (列挙) を返します。

```
>>> ssl.create_default_context().verify_mode
<VerifyMode.CERT_REQUIRED: 2>
```

18.3.4 証明書

証明書を大まかに説明すると、公開鍵/秘密鍵システムの一種です。このシステムでは、各 *principal* (これはマシン、人、組織などです) は、ユニークな 2 つの暗号鍵を割り当てられます。1 つは公開され、**公開鍵** *(public key)* と呼ばれます。もう一方は秘密にされ、**秘密鍵** *(private key)* と呼ばれます。2 つの鍵は関連しており、片方の鍵で暗号化したメッセージは、もう片方の鍵 **のみ** で復号できます。

証明書は 2 つの principal の情報を含んでいます。証明書は *subject* 名とその公開鍵を含んでいます。また、もう一つの principal である **発行者** *(issuer)* からの、subject が本人であることと、その公開鍵が正しいことの宣言 (statement) を含んでいます。発行者からの宣言は、その発行者の秘密鍵で署名されています。発行者の秘密鍵は発行者しか知りませんが、誰もがその発行者の公開鍵を利用して宣言を復号し、証明書内の別の情報と比較することで認証することができます。証明書はまた、その証明書が有効である期限に関する情報も含んでいます。この期限は "notBefore" と "notAfter" と呼ばれる 2 つのフィールドで表現されています。

Python において証明書を利用する場合、クライアントもサーバーも自分を証明するために証明書を利用することができます。ネットワーク接続の相手側に証明書の提示を要求する事ができ、そのクライアントやサーバーが認証を必要とするならその証明書を認証することができます。認証が失敗した場合、接続は例外を発生させます。認証は下位層の OpenSSL フレームワークが自動的に行います。アプリケーションは認証機構につ

いて意識する必要はありません。しかし、アプリケーションは認証プロセスのために幾つかの証明書を提供する必要があるかもしれません。

Python は証明書を格納したファイルを利用します。そのファイルは "PEM" (**RFC 1422** 参照) フォーマットという、ヘッダー行とフッター行の間に base-64 エンコードされた形をとっている必要があります。

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

証明書チェイン

Python が利用する証明書を格納したファイルは、ときには **証明書チェイン** *(certificate chain)* と呼ばれる証明書のシーケンスを格納します。このチェインの先頭には、まずクライアントやサーバーである principal の証明書を置き、それ以降には、その証明書の発行者 (issuer) の証明書などを続け、最後に証明対象 (subject) と発行者が同じ **自己署名** *(self-signed)* 証明書で終わります。この最後の証明書は **ルート証明書** *(root certificate* と呼ばれます。これらの証明書チェインは単純に 1 つの証明書ファイルに結合してください。例えば、3 つの証明書からなる証明書チェインがある場合、私たちのサーバーの証明書から、私たちのサーバーに署名した認証局の証明書、そして認証局の証明書を発行した機関のルート証明書と続きます:

```
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the root certificate for the CA's issuer)...
-----END CERTIFICATE-----
```

CA 証明書

もし相手から送られてきた証明書の認証をしたい場合、信頼している各発行者の証明書チェインが入った "CA certs" ファイルを提供する必要があります。繰り返しますが、このファイルは単純に、各チェインを結合しただけのものです。認証のために、Python はそのファイルの中の最初にマッチしたチェインを利用します。*SSLContext.load_default_certs()* を呼び出すことでプラットフォームの証明書ファイルも使われますが、これは *create_default_context()* によって自動的に行われます。

秘密鍵と証明書の組み合わせ

多くの場合、証明書と同じファイルに秘密鍵も格納されています。この場合、*SSLContext.load_cert_chain()*, *wrap_socket()* には certfile 引数だけが必要とされます。秘密鍵が証明書ファイルに格納されている場合、秘密鍵は証明書チェインの最初の証明書よりも先にないといけません。

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

自己署名証明書

SSL 暗号化接続サービスを提供するサーバーを建てる場合、適切な証明書を取得するには、認証局から買うなどの幾つかの方法があります。また、自己署名証明書を作るケースもあります。OpenSSL を使って自己署名証明書を作るには、次のようにします。

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
Generating a 1024 bit RSA private key
.......++++++
.............................++++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization.com
Email Address []:ops@myserver.mygroup.myorganization.com
%
```

自己署名証明書の欠点は、それ自身がルート証明書であり、他の人はその証明書を持っていない (そして信頼しない) ことです。

18.3.5 使用例

SSL サポートをテストする

インストールされている Python が SSL をサポートしているかどうかをテストするために、ユーザーコードは次のイディオムを利用することができます。

```
try:
    import ssl
except ImportError:
    pass
else:
    ...  # do something that requires SSL support
```

クライアントサイドの処理

この例では、自動的に証明書の検証を行うことを含む望ましいセキュリティ設定でクライアントソケットの SSL コンテキストを作ります:

```
>>> context = ssl.create_default_context()
```

自分自身でセキュリティ設定を調整したい場合、コンテキストを一から作ることはできます (ただし、正しくない設定をしてしまいがちなことに注意してください):

```
>>> context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(このスニペットはすべての CA 証明書が `/etc/ssl/certs/ca-bundle.crt` にバンドルされていることを仮定しています; もし違っていればエラーになりますので、適宜修正してください)

The *PROTOCOL_TLS_CLIENT* protocol configures the context for cert validation and hostname verification. *verify_mode* is set to *CERT_REQUIRED* and *check_hostname* is set to `True`. All other protocols create SSL contexts with insecure defaults.

When you use the context to connect to a server, *CERT_REQUIRED* and *check_hostname* validate the server certificate: it ensures that the server certificate was signed with one of the CA certificates, checks the signature for correctness, and verifies other properties like validity and identity of the hostname:

```
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
...                            server_hostname="www.python.org")
>>> conn.connect(("www.python.org", 443))
```

そして証明書を持ってくることができます:

```
>>> cert = conn.getpeercert()
```

証明書が、期待しているサービス (つまり、HTTPS ホスト `www.python.org`) の身元を特定していることを視覚的に点検してみましょう:

```
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2ExtendedValidationServerCA.crt',),
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ev-server-g1.crl',
                           'http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'DigiCert Inc'),),
            (('organizationalUnitName', 'www.digicert.com'),),
            (('commonName', 'DigiCert SHA2 Extended Validation Server CA'),)),
 'notAfter': 'Sep  9 12:00:00 2016 GMT',
 'notBefore': 'Sep  5 00:00:00 2014 GMT',
 'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
 'subject': ((('businessCategory', 'Private Organization'),),
             (('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
             (('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
             (('serialNumber', '3359300'),),
             (('streetAddress', '16 Allen Rd'),),
             (('postalCode', '03894-4801'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'NH'),),
             (('localityName', 'Wolfeboro'),),
             (('organizationName', 'Python Software Foundation'),),
             (('commonName', 'www.python.org'),)),
 'subjectAltName': (('DNS', 'www.python.org'),
                    ('DNS', 'python.org'),
                    ('DNS', 'pypi.org'),
                    ('DNS', 'docs.python.org'),
                    ('DNS', 'testpypi.org'),
                    ('DNS', 'bugs.python.org'),
                    ('DNS', 'wiki.python.org'),
                    ('DNS', 'hg.python.org'),
                    ('DNS', 'mail.python.org'),
                    ('DNS', 'packaging.python.org'),
                    ('DNS', 'pythonhosted.org'),
                    ('DNS', 'www.pythonhosted.org'),
                    ('DNS', 'test.pythonhosted.org'),
                    ('DNS', 'us.pycon.org'),
                    ('DNS', 'id.python.org')),
 'version': 3}
```

SSL チャネルは今や確立されて証明書が検証されているので、サーバとのお喋りを続けることができます:

```
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
```

(次のページに続く)

(前のページからの続き)

```
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
 b'X-Cache: HIT',
 b'X-Cache-Hits: 11',
 b'Vary: Cookie',
 b'Strict-Transport-Security: max-age=63072000; includeSubDomains',
 b'Connection: close',
 b'',
 b'']
```

このドキュメントの下の方の、**セキュリティで考慮すべき点** に関する議論を参照してください。

サーバサイドの処理

サーバサイドの処理では、通常、サーバー証明書と秘密鍵がそれぞれファイルに格納された形で必要です。最初に秘密鍵と証明書が保持されたコンテキストを作成し、クライアントがあなたの信憑性をチェックできるようにします。そののちにソケットを開き、ポートにバインドし、そのソケットの `listen()` を呼び、クライアントからの接続を待ちます。

```
import socket, ssl

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="mycertfile", keyfile="mykeyfile")

bindsocket = socket.socket()
bindsocket.bind(('myaddr.mydomain.com', 10023))
bindsocket.listen(5)
```

クライアントが接続してきた場合、`accept()` を呼んで新しいソケットを作成し、接続のためにサーバサイドの SSL ソケットを、コンテキストの *SSLContext.wrap_socket()* メソッドで作ります:

```
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

そして、connstream からデータを読み、クライアントと切断する (あるいはクライアントが切断してくる) まで何か処理をします。

```
def deal_with_client(connstream):
    data = connstream.recv(1024)
    # empty data means the client is finished with us
    while data:
        if not do_something(connstream, data):
```

(次のページに続く)

(前のページからの続き)

```
            # we'll assume do_something returns False
            # when we're finished with client
            break
        data = connstream.recv(1024)
    # finished with client
```

そして新しいクライアント接続のために listen に戻ります。(もちろん現実のサーバは、おそらく個々のクライアント接続ごとに別のスレッドで処理するか、ソケットを **ノンブロッキングモード** にし、イベントループを使うでしょう。)

18.3.6 ノンブロッキングソケットについての注意事項

SSL ソケットはノンブロッキングモードにおいては、普通のソケットとは少し違った振る舞いをします。ですのでノンブロッキングソケットとともに使う場合、いくつか気をつけなければならない事項があります:

- ほとんどの *SSLSocket* のメソッドは I/O 操作がブロックすると *BlockingIOError* ではなく *SSLWantWriteError* か *SSLWantReadError* のどちらかを送出します。*SSLWantReadError* は下層のソケットで読み出しが必要な場合に送出され、*SSLWantWriteError* は下層のソケットで書き込みが必要な場合に送出されます。SSL ソケットに対して **書き込み** を試みると下層のソケットから最初に **読み出す** 必要があるかもしれず、SSL ソケットに対して **読み出し** を試みると下層のソケットに先に **書き込む** 必要があるかもしれないことに注意してください。

  バージョン 3.5 で変更: 以前の Python バージョンでは、SSLSocket.send() メソッドは *SSLWantWriteError* または *SSLWantReadError* を送出するのではなく、ゼロを返していました。

- *select()* 呼び出しは OS レベルでのソケットが読み出し可能 (または書き込み可能) になったことを教えてくれますが、上位の SSL レイヤーでの十分なデータがあることを意味するわけではありません。例えば、SSL フレームの一部が届いただけかもしれません。ですから、SSLSocket.recv() と SSLSocket.send() の失敗を処理することに備え、ほかの *select()* 呼び出し後にリトライしなければなりません。

- 反対に、SSL レイヤーは独自の枠組みを持っているため、*select()* が気付かない読み出し可能なデータを SSL ソケットが持っている場合があります。したがって、入手可能な可能性のあるデータをすべて引き出すために最初に SSLSocket.recv() を呼び出し、次にそれでもまだ必要な場合にだけ *select()* 呼び出しでブロックすべきです。

  (当然のことながら、ほかのプリミティブ、例えば *poll()* や *selectors* モジュール内のものを使う際にも似た但し書きが付きます)

- SSL ハンドシェイクそのものがノンブロッキングになります: *SSLSocket.do_handshake()* メソッドは成功するまでリトライしなければなりません。*select()* を用いてソケットの準備が整うのを待つためには、およそ以下のようにします:

```
while True:
    try:
        sock.do_handshake()
```

(次のページに続く)

(前のページからの続き)

```
        break
    except ssl.SSLWantReadError:
        select.select([sock], [], [])
    except ssl.SSLWantWriteError:
        select.select([], [sock], [])
```

参考:

asyncio モジュールは **ノンブロッキング *SSL* ソケット** をサポートし、より高いレベルの API を提供しています。*selectors* モジュールを使ってイベントを poll し、*SSLWantWriteError*, *SSLWantReadError*, *BlockingIOError* 例外を処理します。SSL ハンドシェイクも非同期に実行します。

18.3.7 メモリ BIO サポート

バージョン 3.5 で追加.

Python 2.6 で SSL モジュールが導入されて以降、*SSLSocket* クラスは、以下の互いに関連するが別々の機能を提供してきました。

- SSL プロトコル処理
- ネットワーク IO

ネットワーク IO API は、*socket.socket* が提供するものと同じです。*SSLSocket* も、そのクラスから継承しています。これにより、SSL ソケットは標準のソケットをそっくりそのまま置き換えるものとして使用できるため、既存のアプリケーションを SSL に対応させるのが非常に簡単になります。

SSL プロトコルの処理とネットワーク IO を組み合わせた場合、通常は問題なく動作しますが、問題が発生する場合があります。一例を挙げると、非同期 IO フレームワークが別の多重化モデルを使用する場合、これは *socket.socket* と内部 OpenSSL ソケット IO ルーティンが想定する「ファイル記述子上の select/poll」モデル（準備状態ベース）とは異なります。これは、このモデルが非効率的になる Windows などのプラットフォームに主に該当します。そのため、スコープを限定した *SSLSocket* の変種、*SSLObject* が提供されています。

class ssl.SSLObject

ネットワーク IO メソッドを含まない SSL プロトコルインスタンスを表す、スコープを限定した *SSLSocket* の変種です。一般的にこ、のクラスを使用するのは、メモリバッファを通じて SSL のための非同期 IO を実装するフレームワーク作成者です。

このクラスは、OpenSSL が実装する低水準 SSL オブジェクトの上にインターフェースを実装します。このオブジェクトは SSL 接続の状態をキャプチャしますが、ネットワーク IO 自体は提供しません。IO は、OpenSSL の IO 抽象レイヤである別の「BIO」オブジェクトを通じて実行する必要があります。

このクラスには公開されたコンストラクタがありません。*SSLObject* インスタンスは、*wrap_bio()* メソッドを使用して作成しなければなりません。このメソッドは、*SSLObject* インスタンスを作成し、2 つの BIO に束縛します。*incoming* BIO は、Python から SSL プロトコルインスタンスにデータを渡すために使用され、*outgoing* BIO は、データを反対向きに渡すために使用されます。

次のメソッドがサポートされています:

- *context*
- *server_side*
- *server_hostname*
- *session*
- *session_reused*
- *read()*
- *write()*
- *getpeercert()*
- *selected_alpn_protocol()*
- *selected_npn_protocol()*
- *cipher()*
- *shared_ciphers()*
- *compression()*
- *pending()*
- *do_handshake()*
- *verify_client_post_handshake()*
- *unwrap()*
- *get_channel_binding()*
- *version()*

SSLSocket と比較すると、このオブジェクトでは以下の機能が不足しています。

- Any form of network IO; `recv()` and `send()` read and write only to the underlying *MemoryBIO* buffers.
- *do_handshake_on_connect* 機構はありません。必ず手動で *do_handshake()* を呼んで、ハンドシェイクを開始する必要があります。
- *suppress_ragged_eofs* は処理されません。プロトコルに違反するファイル末尾状態は、*SSLEOFError* 例外を通じて報告されます。
- *unwrap()* メソッドの呼び出しは、下層のソケットを返す SSL ソケットとは異なり、何も返しません。

- *SSLContext.set_servername_callback()* に渡される *server_name_callback* コールバックは、1 つ目の引数として *SSLSocket* インスタンスではなく *SSLObject* インスタンスを受け取ります。

SSLObject の使用に関する注意:

- *SSLObject* 上のすべての IO は *non-blocking* です。例えば、*read()* は入力 BIO が持つデータよりも多くのデータを必要とする場合、*SSLWantReadError* を送出します。
- *wrap_socket()* に対して存在するような、モジュールレベルの `wrap_bio()` 呼び出しは存在しません。*SSLObject* は、常に *SSLContext* を経由して作成されます。

バージョン 3.7 で変更: *SSLObject* instances must to created with *wrap_bio()*. In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

SSLObject は、メモリバッファを使用して外界と通信します。*MemoryBIO* クラスは、以下のように OpenSSL メモリ BIO (Basic IO) オブジェクトをラップし、この目的に使用できるメモリバッファを提供します。

class ssl.MemoryBIO

Python と SSL プロトコルインスタンス間でデータをやり取りするために使用できるメモリバッファ。

pending

現在メモリバッファ中にあるバイト数を返します。

eof

メモリ BIO が現在ファイルの末尾にあるかを表す真偽値です。

read(*n=-1*)

メモリバッファから最大 *n* 読み取ります。*n* が指定されていないか、負値の場合、すべてのバイトが返されます。

write(*buf*)

buf からメモリ BIO にバイトを書き込みます。*buf* 引数は、バッファプロトコルをサポートするオブジェクトでなければなりません。

戻り値は、書き込まれるバイト数であり、常に *buf* の長さと等しくなります。

write_eof()

EOF マーカーをメモリ BIO に書き込みます。このメソッドが呼び出された後に *write()* を呼ぶことはできません。*eof* 属性は、バッファ内のすべてのデータが読み出された後に True になります。

18.3.8 SSL セッション

バージョン 3.6 で追加.

class ssl.SSLSession

session が使用するセッションオブジェクトです。

id

time

timeout

ticket_lifetime_hint

has_ticket

18.3.9 セキュリティで考慮すべき点

最善のデフォルト値

クライアントでの使用 では、セキュリティポリシーによる特殊な要件がない限りは、*create_default_context()* 関数を使用して SSL コンテキストを作成することを強くお勧めします。この関数は、システムの信頼済み CA 証明書をロードし、証明書の検証とホスト名のチェックを有効化し、十分にセキュアなプロトコルと暗号を選択しようとします。

例として、*smtplib.SMTP* クラスを使用して SMTP サーバーに対して信頼できるセキュアな接続を行う方法を以下に示します:

```
>>> import ssl, smtplib
>>> smtp = smtplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

接続にクライアントの証明書が必要な場合、*SSLContext.load_cert_chain()* によって追加できます。

対照的に、自分自身で *SSLContext* クラスのコンストラクタを呼び出すことによって SSL コンテキストを作ると、デフォルトでは証明書検証もホスト名チェックも有効になりません。自分で設定を行う場合は、十分なセキュリティレベルを達成するために、以下のパラグラフをお読みください。

手動での設定

証明書の検証

SSLContext のコンストラクタを直接呼び出した場合、*CERT_NONE* がデフォルトとして使われます。これは接続先の身元特定をしないので安全ではありませんし、特にクライアントモードでは大抵相手となるサーバの信憑性を保障したいでしょう。ですから、クライアントモードでは *CERT_REQUIRED* を強くお勧めします。ですが、それだけでは不十分です; *SSLSocket.getpeercert()* を呼び出してサーバ証明書が望んだサービスと合致するかのチェックもしなければなりません。多くのプロトコルとアプリケーションにとって、サービスはホスト名で特定されます; この場合、*match_hostname()* が使えます。これらの共通的なチェックは *SSLContext.check_hostname* が有効な場合、自動的に行われます。

バージョン 3.7 で変更: Hostname matchings is now performed by OpenSSL. Python no longer uses *match_hostname()*.

サーバモードにおいて、(より上位のレベルでの認証メカニズムではなく) SSL レイヤーを使ってあなたのクライアントを認証したいならば、*CERT_REQUIRED* を指定して同じようにクライアントの証明書を検証すべきでしょう。

プロトコルのバージョン

SSL バージョン 2 と 3 は安全性に欠けると考えられており、使用するのは危険です。クライアントとサーバ間の互換性を最大限に確保したい場合、プロトコルバージョンとして *PROTOCOL_TLS_CLIENT* または *PROTOCOL_TLS_SERVER* を使用してください。SSLv2 と SSLv3 はデフォルトで無効になっています。

```
>>> client_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> client_context.options |= ssl.OP_NO_TLSv1
>>> client_context.options |= ssl.OP_NO_TLSv1_1
```

上記で作成した SSL コンテキストは、TLSv1.2 以降 (システムでサポートされている場合) でのサーバへの接続のみを許可します。*PROTOCOL_TLS_CLIENT* は、デフォルトで証明書の検証とホスト名のチェックを意味します。コンテキスト中に証明書をロードする必要があります。

暗号の選択

高度なセキュリティが要求されている場合、SSL セッションのネゴシエーションで有効になる暗号の微調整が *SSLContext.set_ciphers()* によって可能です。Python 3.2.3 以降、ssl モジュールではデフォルトで特定の弱い暗号化が無効になっていますが、暗号方式の選択をさらに厳しく制限したい場合もあるでしょう。OpenSSL ドキュメントの *cipher list format* '_ **を注意深く読んでください。与えられた暗号方式リストによって有効になる暗号方式をチェックするには、***:meth:'SSLContext.get_ciphers* メソッドまたは `openssl ciphers` コマンドをシステム上で実行してください。

マルチプロセス化

(例えば *multiprocessing* や *concurrent.futures* を使って、) マルチプロセスアプリケーションの一部としてこのモジュールを使う場合、OpenSSL の内部の乱数発生器は fork したプロセスを適切に処理しないことに気を付けて下さい。SSL の機能を *os.fork()* とともに使う場合、アプリケーションは親プロセスの PRNG 状態を変更しなければなりません。*RAND_add()*, *RAND_bytes()*, *RAND_pseudo_bytes()* のいずれかの呼び出し成功があれば十分です。

18.3.10 TLS 1.3

バージョン 3.7 で追加.

Python has provisional and experimental support for TLS 1.3 with OpenSSL 1.1.1. The new protocol behaves slightly differently than previous version of TLS/SSL. Some new TLS 1.3 features are not yet available.

- TLS 1.3 uses a disjunct set of cipher suites. All AES-GCM and ChaCha20 cipher suites are enabled by default. The method *SSLContext.set_ciphers()* cannot enable or disable any TLS 1.3 ciphers yet, but *SSLContext.get_ciphers()* returns them.
- Session tickets are no longer sent as part of the initial handshake and are handled differently. *SSLSocket.session* and *SSLSession* are not compatible with TLS 1.3.
- Client-side certificates are also no longer verified during the initial handshake. A server can request a certificate at any time. Clients process certificate requests while they send or receive application data from the server.
- TLS 1.3 features like early data, deferred TLS client cert request, signature algorithm configuration, and rekeying are not supported yet.

18.3.11 LibreSSL support

LibreSSL is a fork of OpenSSL 1.0.1. The ssl module has limited support for LibreSSL. Some features are not available when the ssl module is compiled with LibreSSL.

- LibreSSL >= 2.6.1 no longer supports NPN. The methods *SSLContext.set_npn_protocols()* and *SSLSocket.selected_npn_protocol()* are not available.
- *SSLContext.set_default_verify_paths()* ignores the env vars `SSL_CERT_FILE` and `SSL_CERT_PATH` although *get_default_verify_paths()* still reports them.

参考:

socket.socket **クラス** 下位レイヤーの *socket* クラスのドキュメント

SSL/TLS Strong Encryption: An Introduction Apache HTTP サーバのドキュメンテーションのイントロ

RFC 1422: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management
Steve Kent

RFC 4086: Randomness Requirements for Security Donald E., Jeffrey I. Schiller

RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile
D. Cooper

RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2 T. Dierks et. al.

RFC 6066: Transport Layer Security (TLS) Extensions D. Eastlake

IANA TLS: Transport Layer Security (TLS) Parameters IANA

RFC 7525: Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Se
IETF

'Mozilla's Server Side TLS recommendations '_ Mozilla

18.4 select --- I/O 処理の完了を待機する

このモジュールでは、ほとんどのオペレーティングシステムで利用可能な `select()` および `poll()` 関数、Solaris やその派生で利用可能な `devpoll()` 、Linux 2.5+ で利用可能な `epoll()` 、多くの BSD で利用可能な `kqueue()` 関数に対するアクセスを提供しています。Windows 上ではソケットに対してしか動作しないので注意してください; その他のオペレーティングシステムでは、他のファイル形式でも (特に Unix ではパイプにも) 動作します。通常のファイルに対して適用し、最後にファイルを読み出した時から内容が増えているかを決定するために使うことはできません。

注釈: *selectors* モジュールにより、*select* モジュールプリミティブに基づく高水準かつ効率的な I/O の多重化が行うことが出来ます。OS レベルプリミティブを使用した正確な制御を求めない限り、このモジュールの使用が推奨されます。

このモジュールは以下を定義します:

exception select.error

OSError の非推奨のエイリアスです。

バージョン 3.3 で変更: **PEP 3151** に基づき、このクラスは *OSError* のエイリアスになりました。

select.devpoll()

(Solaris およびその派生でのみサポートされています) `/dev/poll` ポーリングオブジェクトを返します。ポーリングオブジェクトが提供しているメソッドについては **ポーリングオブジェクト** 節を参照してください。

`devpoll()` オブジェクトはインスタンス化時に許されるファイル記述子の数にリンクされます。プログラムがこの値を減らす場合 `devpoll()` は失敗します。プログラムがこの値を増やす場合 `devpoll()`

は有効なファイル記述子の不完全なリストを返すことがあります。

新しいファイル記述子は **継承不可** です。

バージョン 3.3 で追加.

バージョン 3.4 で変更: 新しいファイル記述子が継承不可になりました。

select.epoll(*sizehint=-1*, *flags=0*)

(Linux 2.5.44 以降でのみサポート) エッジポーリング (edge polling) オブジェクトを返します。このオブジェクトは、I/O イベントのエッジトリガもしくはレベルトリガインタフェースとして使えます。

sizehint informs epoll about the expected number of events to be registered. It must be positive, or *-1* to use the default. It is only used on older systems where `epoll_create1()` is not available; otherwise it has no effect (though its value is still checked).

flags is deprecated and completely ignored. However, when supplied, its value must be `0` or `select.EPOLL_CLOEXEC`, otherwise `OSError` is raised.

エッジポーリングオブジェクトが提供しているメソッドについては **エッジおよびレベルトリガポーリング** *(epoll)* **オブジェクト** 節を参照してください。

`epoll` オブジェクトはコンテキストマネジメントプロトコルをサポートしています。`with` 文内で使用された場合、新たなファイル記述子はブロックの最後で自動的に閉じられます。

新しいファイル記述子は **継承不可** です。

バージョン 3.3 で変更: *flags* 引数が追加されました。

バージョン 3.4 で変更: `with` 文のサポートが追加されました。新しいファイル記述子が継承不可になりました。

バージョン 3.4 で非推奨: *flags* パラメータ。現在ではデフォルトで `select.EPOLL_CLOEXEC` が使われます。ファイルディスクリプタを継承可能にするには *os.set_inheritable()* を使ってください。

select.poll()

(全てのオペレーティングシステムでサポートされているわけではありません) ポーリングオブジェクトを返します。このオブジェクトはファイル記述子を登録したり登録解除したりすることができ、ファイル記述子に対する I/O イベント発生をポーリングすることができます; ポーリングオブジェクトが提供しているメソッドについては **ポーリングオブジェクト** 節を参照してください。

select.kqueue()

(BSD でのみサポート) カーネルキュー (kernel queue) オブジェクトを返します。カーネルキューオブジェクトが提供しているメソッドについては、*kqueue* **オブジェクト** 節を参照してください。

新しいファイル記述子は **継承不可** です。

バージョン 3.4 で変更: 新しいファイル記述子が継承不可になりました。

select.kevent(*ident*, *filter=KQ_FILTER_READ*, *flags=KQ_EV_ADD*, *fflags=0*, *data=0*, *udata=0*)

(BSD でのみサポート) カーネルイベント (kernel event) オブジェクトを返します。カーネルイベント

オブジェクトが提供しているメソッドについては、*kevent* **オブジェクト** 節を参照してください。

`select.select`(*rlist*, *wlist*, *xlist*[, *timeout*])

Unix の `select()` システムコールに対する直接的なインタフェースです。最初の 3 つの引数は ' 待機可能オブジェクト' からなるイテラブルです: 待機可能オブジェクトとは、ファイル記述子を表す整数値か、そのような整数を返す引数なしメソッド *fileno()* を持つオブジェクトです。

- *rlist*: 読み込み可能になるまで待機
- *wlist*: 書き込み可能になるまで待機
- *xlist*: " 例外状態 (exceptional condition)" になるまで待機 (" 例外状態" については、システムのマニュアルページを参照してください)

引数に空のイテラブルを指定してもかまいませんが、3 つの引数全てを空のイテラブルにしてもよいかどうかはプラットフォームに依存します (Unix では動作し、Windows では動作しないことが知られています)。オプションの *timeout* 引数にはタイムアウトまでの秒数を浮動小数点数で指定します。*timeout* 引数が省略された場合、関数は少なくとも一つのファイル記述子が何らかの準備完了状態になるまでブロックします。*timeout* に 0 を指定した場合は、ポーリングを行いブロックしないことを示します。

戻り値は準備完了状態のオブジェクトからなる 3 つのリストです: したがってこのリストはそれぞれ関数の最初の 3 つの引数のサブセットになります。ファイル記述子のいずれも準備完了にならないままタイムアウトした場合、3 つの空のリストが返されます。

イテラブルの中に含めることのできるオブジェクトは Python **ファイルオブジェクト** (例えば `sys.stdin` や、*open()* または *os.popen()* が返すオブジェクト)、*socket.socket()* が返すソケットオブジェクトです。**ラッパー** *<wrapper>* クラスを自分で定義することもできます。この場合、適切な (まったくデタラメな数ではなく本物のファイル記述子を返す) *fileno()* メソッドを持つ必要があります。

注釈: *select()* は Windows のファイルオブジェクトを受理しませんが、ソケットは受理します。Windows では、背後の `select()` 関数は WinSock ライブラリで提供されており、WinSock によって生成されたものではないファイル記述子を扱うことができないのです。

バージョン 3.5 で変更: この関数は、シグナルによって中断された時に、*InterruptedError* を上げる代わりに再計算されたタイムアウトによってリトライするようになりました。ただし、シグナルハンドラが例外を起こした場合を除きます (この論理的根拠については **PEP 475** を見てください)。

`select.PIPE_BUF`

select(), *poll()* またはこのモジュールの別のインタフェースによってパイプが書き込む準備ができていると報告された時に、ブロックせずにパイプに書き込むことのできる最小のバイト数。これはソケットなどの他の種類の file-like オブジェクトには適用されません。

この値は POSIX により少なくとも 512 であることが保証されています。

利用可能な環境: Unix

バージョン 3.2 で追加.

18.4.1 /dev/poll ポーリングオブジェクト

Solaris とその派生は、/dev/poll を持っています。select() が O(最大のファイル記述子) 、poll() が O(ファイル記述子の数) である一方、/dev/poll は O(アクティブなファイル記述子) です。

/dev/poll の挙動は標準的な poll() オブジェクトに非常に近いです。

devpoll.close()
ポーリングオブジェクトのファイル記述子を閉じます。

バージョン 3.4 で追加.

devpoll.closed
ポーリングオブジェクトが閉じている場合 True です。

バージョン 3.4 で追加.

devpoll.fileno()
ポーリングオブジェクトのファイル記述子番号を返します。

バージョン 3.4 で追加.

devpoll.register(*fd*[, *eventmask*])
ファイル記述子をポーリングオブジェクトに登録します。これ以降の *poll()* メソッド呼び出しでは、そのファイル記述子に処理待ち中の I/O イベントがあるかどうかを監視します。*fd* は整数か、整数値を返す *fileno()* メソッドを持つオブジェクトを取ります。ファイルオブジェクトも fileno() を実装しているので、引数として使うことができます。

eventmask はオプションのビットマスクで、どの種類の I/O イベントを監視したいかを記述します。poll() オブジェクトと同じ定数が使われます。デフォルト値は定数 POLLIN 、POLLPRI 、および POLLOUT の組み合わせです。

> **警告:** 登録済みのファイル記述子を登録してもエラーにはなりませんが、結果は未定義です。適切なアクションは、最初に unregister するか modify することです。これは poll() と比較した場合の重要な違いです。

devpoll.modify(*fd*[, *eventmask*])
このメソッドは *unregister()* に続いて *register()* を行います。同じことを明示的に行うよりも (少し) 効率的です。

devpoll.unregister(*fd*)
ポーリングオブジェクトによって追跡中のファイル記述子を登録解除します。*register()* メソッドと同様に、*fd* は整数か、整数値を返す *fileno()* メソッドを持つオブジェクトを取ります。

登録されていないファイル記述子の削除を試みるのは安全に無視されます。

devpoll.poll(*[timeout]*)

登録されたファイル記述子に対してポーリングを行い、報告すべき I/O イベントまたはエラーの発生したファイル記述子毎に 2 要素のタプル (fd, event) からなるリストを返します。リストは空になることもあります。*fd* はファイル記述子で、*event* は該当するファイル記述子について報告されたイベントを表すビットマスクです --- 例えば POLLIN は入力待ちを示し、POLLOUT はファイル記述子に対する書き込みが可能を示す、などです。空のリストは呼び出しがタイムアウトしたか、報告すべきイベントがどのファイル記述子でも発生しなかったことを示します。*timeout* が与えられた場合、処理を戻すまで待機する時間の長さをミリ秒単位で指定します。*timeout* が省略されたり、-1 であったり、あるいは *None* の場合、そのポーリングオブジェクトが監視している何らかのイベントが発生するまでブロックします。

バージョン 3.5 で変更: この関数は、シグナルによって中断された時に、*InterruptedError* を上げる代わりに再計算されたタイムアウトによってリトライするようになりました。ただし、シグナルハンドラが例外を起こした場合を除きます (この論理的根拠については **PEP 475** を見てください)。

18.4.2 エッジおよびレベルトリガポーリング (epoll) オブジェクト

https://linux.die.net/man/4/epoll

eventmask

定数	意味
EPOLLIN	読み込み可能
EPOLLOUT	書き込み可能
EPOLLPRI	緊急の読み出しデータ
EPOLLERR	設定された fd にエラー状態が発生した
EPOLLHUP	設定された fd がハングアップした
EPOLLET	エッジトリガ動作に設定する。デフォルトではレベルトリガ動作
EPOLLONESHOT	ワンショット動作に設定する。1 回イベントが取り出されたら、その fd が内部で無効になる
EPOLLEXCLUSIVE	関連づけられた fd にイベントがある場合、1 つの epoll オブジェクトのみを起こします。デフォルトでは (このフラグが設定されていない場合には)、fd に対してポーリングするすべての epoll オブジェクトを起こします。
EPOLLRDHUP	ストリームソケットの他端が接続を切断したか、接続の書き込み側のシャットダウンを行った。
EPOLLRDNORM	EPOLLIN と同じ
EPOLLRDBAND	優先データバンドを読み込める。
EPOLLWRNORM	EPOLLOUT と同じ
EPOLLWRBAND	優先データに書き込みできる。
EPOLLMSG	無視される。

バージョン 3.6 で追加: EPOLLEXCLUSIVE was added. It's only supported by Linux Kernel 4.5 or later.

epoll.close()

epoll オブジェクトの制御用ファイル記述子を閉じます。

epoll.closed
epoll オブジェクトが閉じている場合 True です。

epoll.fileno()
制御用ファイル記述子の番号を返します。

epoll.fromfd(*fd*)
fd から epoll オブジェクトを作成します。

epoll.register(*fd*[, *eventmask*])
epoll オブジェクトにファイル記述子 *fd* を登録します。

epoll.modify(*fd*, *eventmask*)
登録されたファイル記述子変更します。

epoll.unregister(*fd*)
epoll オブジェクトから登録されたファイル記述子 *fd* を削除します。

epoll.poll(*timeout=None*, *maxevents=-1*)
イベントを待機します。*timeout* はタイムアウト時間で、単位は秒 (float 型) です。

バージョン 3.5 で変更: この関数は、シグナルによって中断された時に、*InterruptedError* を上げる代わりに再計算されたタイムアウトによってリトライするようになりました。ただし、シグナルハンドラが例外を起こした場合を除きます (この論理的根拠については **PEP 475** を見てください)。

18.4.3 ポーリングオブジェクト

`poll()` システムコールはほとんどの Unix システムでサポートされており、非常に多数のクライアントに同時にサービスを提供するようなネットワークサーバが高いスケーラビリティを持てるようにしています。`poll()` は対象のファイル記述子を列挙するだけでよいため、良くスケールします。一方、`select()` はビット対応表を構築し、対象ファイルの記述子に対応するビットを立て、その後全ての対応表の全てのビットを線形探索します。`select()` は O(最大のファイル記述子番号) なのに対し、`poll()` は O(対象とするファイル記述子の数) で済みます。

poll.register(*fd*[, *eventmask*])
ファイル記述子をポーリングオブジェクトに登録します。これ以降の *poll()* メソッド呼び出しでは、そのファイル記述子に処理待ち中の I/O イベントがあるかどうかを監視します。*fd* は整数か、整数値を返す *fileno()* メソッドを持つオブジェクトを取ります。ファイルオブジェクトも `fileno()` を実装しているので、引数として使うことができます。

eventmask はオプションのビットマスクで、どの種類の I/O イベントを監視したいかを記述します。この値は以下の表で述べる定数 `POLLIN` 、`POLLPRI` 、および `POLLOUT` の組み合わせにすることができます。ビットマスクを指定しない場合、標準の値が使われ、3 種類のイベント全てに対して監視が行われます。

定数	意味
POLLIN	読み出し可能なデータが存在する
POLLPRI	緊急の読み出し可能なデータが存在する
POLLOUT	書き出しの準備ができている: 書き出し処理がブロックしない
POLLERR	何らかのエラー状態
POLLHUP	ハングアップ
POLLRDHUP	ストリームソケットの他端が接続を切断したか、接続の書き込み側のシャットダウンを行った。
POLLNVAL	無効な要求: 記述子が開かれていない

登録済みのファイル記述子を登録してもエラーにはならず、一度だけ登録した場合と同じ効果になります。

poll.**modify**(*fd*, *eventmask*)

登録されているファイル記述子 *fd* を変更する。これは、`register(fd, eventmask)` と同じ効果を持ちます。登録されていないファイル記述子に対してこのメソッドを呼び出すと、errno `ENOENT` で *OSError* 例外が発生します。

poll.**unregister**(*fd*)

ポーリングオブジェクトによって追跡中のファイル記述子を登録解除します。*register()* メソッドと同様に、*fd* は整数か、整数値を返す *fileno()* メソッドを持つオブジェクトを取ります。

登録されていないファイル記述子を登録解除しようとすると *KeyError* 例外が送出されます。

poll.**poll**(*[timeout]*)

登録されたファイル記述子に対してポーリングを行い、報告すべき I/O イベントまたはエラーの発生したファイル記述子毎に 2 要素のタプル `(fd, event)` からなるリストを返します。リストは空になることもあります。*fd* はファイル記述子で、*event* は該当するファイル記述子について報告されたイベントを表すビットマスクです --- 例えば `POLLIN` は入力待ちを示し、`POLLOUT` はファイル記述子に対する書き込みが可能を示す、などです。空のリストは呼び出しがタイムアウトしたか、報告すべきイベントがどのファイル記述子でも発生しなかったことを示します。*timeout* が与えられた場合、処理を戻すまで待機する時間の長さをミリ秒単位で指定します。*timeout* が省略されたり、負の値であったり、あるいは *None* の場合、そのポーリングオブジェクトが監視している何らかのイベントが発生するまでブロックします。

バージョン 3.5 で変更: この関数は、シグナルによって中断された時に、*InterruptedError* を上げる代わりに再計算されたタイムアウトによってリトライするようになりました。ただし、シグナルハンドラが例外を起こした場合を除きます (この論理的根拠については **PEP 475** を見てください)。

18.4.4 kqueue オブジェクト

kqueue.close()
kqueue オブジェクトの制御用ファイル記述子を閉じる。

kqueue.closed
kqueue オブジェクトが閉じている場合 True です。

kqueue.fileno()
制御用ファイル記述子の番号を返します。

kqueue.fromfd(*fd*)
与えられたファイル記述子から、kqueue オブジェクトを作成する。

kqueue.control(*changelist*, *max_events*[, *timeout*]) → eventlist
kevent に対する低水準のインタフェース

- *changelist* は kevent オブジェクトのイテラブルまたは None
- *max_events* は 0 または正の整数
- timeout in seconds (floats possible); the default is None, to wait forever

バージョン 3.5 で変更: この関数は、シグナルによって中断された時に、*InterruptedError* を上げる代わりに再計算されたタイムアウトによってリトライするようになりました。ただし、シグナルハンドラが例外を起こした場合を除きます (この論理的根拠については **PEP 475** を見てください)。

18.4.5 kevent オブジェクト

https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

kevent.ident
イベントを特定するための値。この値は、フィルタにもよりますが、大抵の場合はファイル記述子です。コンストラクタでは、ident として、整数値か *fileno()* メソッドを持ったオブジェクトを渡せます。kevent は内部で整数値を保存します。

kevent.filter
カーネルフィルタの名前。

定数	意味
KQ_FILTER_READ	記述子を受け取り、読み込めるデータが存在する時に戻る
KQ_FILTER_WRITE	記述子を受け取り、書き込み可能な時に戻る
KQ_FILTER_AIO	AIO リクエスト
KQ_FILTER_VNODE	*fflag* で監視されたイベントが 1 つ以上発生したときに戻る
KQ_FILTER_PROC	プロセス ID 上のイベントを監視する
KQ_FILTER_NETDEV	ネットワークデバイス上のイベントを監視する (Mac OS X では利用不可)
KQ_FILTER_SIGNAL	監視しているシグナルがプロセスに届いたときに戻る
KQ_FILTER_TIMER	任意のタイマを設定する

kevent.flags
: フィルタアクション。

定数	意味
KQ_EV_ADD	イベントを追加または修正する
KQ_EV_DELETE	キューからイベントを取り除く
KQ_EV_ENABLE	control() がイベントを返すのを許可する
KQ_EV_DISABLE	イベントを無効にする
KQ_EV_ONESHOT	イベントを最初の発生後無効にする
KQ_EV_CLEAR	イベントを受け取った後で状態をリセットする
KQ_EV_SYSFLAGS	内部イベント
KQ_EV_FLAG1	内部イベント
KQ_EV_EOF	フィルタ依存の EOF 状態
KQ_EV_ERROR	戻り値を参照

kevent.fflags
: フィルタ依存のフラグ。

KQ_FILTER_READ と KQ_FILTER_WRITE フィルタのフラグ:

定数	意味
KQ_NOTE_LOWAT	ソケットバッファの最低基準値

KQ_FILTER_VNODE フィルタのフラグ:

定数	意味
KQ_NOTE_DELETE	*unlink()* が呼ばれた
KQ_NOTE_WRITE	書き込みが発生した
KQ_NOTE_EXTEND	ファイルのサイズが拡張された
KQ_NOTE_ATTRIB	属性が変更された
KQ_NOTE_LINK	リンクカウントが変更された
KQ_NOTE_RENAME	ファイル名が変更された
KQ_NOTE_REVOKE	ファイルアクセスが破棄された

KQ_FILTER_PROC フィルタフラグ:

定数	意味
KQ_NOTE_EXIT	プロセスが終了した
KQ_NOTE_FORK	プロセスが *fork()* を呼び出した
KQ_NOTE_EXEC	プロセスが新しいプロセスを実行した
KQ_NOTE_PCTRLMASK	内部フィルタフラグ
KQ_NOTE_PDATAMASK	内部フィルタフラグ
KQ_NOTE_TRACK	*fork()* の呼び出しを超えてプロセスを監視する
KQ_NOTE_CHILD	*NOTE_TRACK* に対して子プロセスに渡される
KQ_NOTE_TRACKERR	子プロセスにアタッチできなかった

KQ_FILTER_NETDEV フィルタフラグ (Mac OS X では利用不可):

定数	意味
KQ_NOTE_LINKUP	リンクアップしている
KQ_NOTE_LINKDOWN	リンクダウンしている
KQ_NOTE_LINKINV	リンク状態が不正

kevent.data
フィルタ固有のデータ。

kevent.udata
ユーザー定義値。

18.5 selectors --- 高水準の I/O 多重化

バージョン 3.4 で追加.

ソースコード: Lib/selectors.py

18.5.1 はじめに

このモジュールにより、*select* モジュールプリミティブに基づく高水準かつ効率的な I/O の多重化が行えます。OS 水準のプリミティブを使用した正確な制御を求めない限り、このモジュールの使用が推奨されます。

このモジュールは *BaseSelector* 抽象基底クラスと、いくつかの具象実装 (*KqueueSelector*, *EpollSelector*...) を定義しており、これらは複数のファイルオブジェクトの I/O の準備状況の通知の待機に使用できます。以下では、”ファイルオブジェクト”は、fileno() メソッドを持つあらゆるオブジェクトか、あるいは Raw ファイル記述子を意味します。ファイルオブジェクト を参照してください。

DefaultSelector は、現在のプラットフォームで利用できる、もっとも効率的な実装の別名になります: これはほとんどのユーザーにとってのデフォルトの選択になるはずです。

注釈: プラットフォームごとにサポートされているファイルオブジェクトのタイプは異なります: Windows ではソケットはサポートされますが、パイプはされません。Unix では両方がサポートされます (その他の fifo やスペシャルファイルデバイスなどのタイプもサポートされます)。

参考:

select 低水準の I/O 多重化モジュールです。

18.5.2 クラス

クラス階層:

```
BaseSelector
+-- SelectSelector
+-- PollSelector
+-- EpollSelector
+-- DevpollSelector
+-- KqueueSelector
```

以下では、*events* は与えられたファイルオブジェクトを待機すべき I/O イベントを示すビット単位のマスクになります。これには以下のモジュール定数の組み合わせを設定できます:

定数	意味
EVENT_READ	読み込み可能
EVENT_WRITE	書き込み可能

class selectors.SelectorKey

SelectorKey はそれの下層のファイルディスクリプタ、選択したイベントマスク、および付属データへのファイルオブジェクトの関連付けに使用される *namedtuple* です。いくつかの *BaseSelector* メソッドを返します。

fileobj
登録されたファイルオブジェクトです。

fd
下層のファイル記述子です。

events
このファイルオブジェクトで待機しなければならないイベントです。

data
このファイルオブジェクトに関連付けられたオプションの不透明型 (Opaque) データです。例えば、これはクライアントごとのセッション ID を格納するために使用できます。

class selectors.BaseSelector

BaseSelector は複数のファイルオブジェクトの I/O イベントの準備状況の待機に使用されます。こ

れはファイルストリームを登録、登録解除、およびこれらのストリームでの I/O イベントを待機 (オプションでタイムアウト) するメソッドをサポートします。これは抽象基底クラスであるため、インスタンスを作成できません。使用する実装を明示的に指定したい、そしてプラットフォームがそれをサポートしている場合は、代わりに *DefaultSelector* を使用するか、*SelectSelector* や *KqueueSelector* などの一つを使用します。*BaseSelector* とその具象実装は **コンテキストマネージャー** プロトコルをサポートしています。

abstractmethod register(*fileobj*, *events*, *data=None*)

I/O イベントを監視するファイルオブジェクトをセレクションに登録します。

fileobj は監視するファイルオブジェクトです。これは整数のファイル記述子か、`fileno()` メソッドを持つオブジェクトのどちらかになります。*events* は監視するイベントのビット幅マスクになります。*data* は不透明型 (Opaque) オブジェクトです。

これは新しい *SelectorKey* インスタンスを返します。不正なイベントマスク化ファイル記述子のときは *ValueError* が、ファイルオブジェクトがすでに登録済みのときは *KeyError* が送出されます。

abstractmethod unregister(*fileobj*)

ファイルオブジェクトのセレクション登録を解除し、監視対象から外します。ファイルオブジェクトの登録解除はそのクローズより前に行われます。

fileobj は登録済みのファイルオブジェクトでなければなりません。

関連付けられた *SelectorKey* インスタンスを返します。*fileobj* が登録されていない場合 *KeyError* を送出します。*fileobj* が不正な場合 (例えば *fileobj* に `fileno()` メソッドが無い場合や `fileno()` メソッドの戻り値が不正な場合) *ValueError* を送出します。

modify(*fileobj*, *events*, *data=None*)

登録されたファイルオブジェクトの監視されたイベントや付属データを変更します。

より効率的に実装できる点を除けば、`BaseSelector.unregister(fileobj)()` に続けて `BaseSelector.register(fileobj, events, data)()` を行うのと等価です。

新たな *SelectorKey* インスタンスを返します。イベントマスクやファイル記述子が不正な場合は *ValueError* を、ファイルオブジェクトが登録されていない場合は *KeyError* を送出します。

abstractmethod select(*timeout=None*)

登録されたいくつかのファイルオブジェクトが準備できたか、タイムアウトするまで待機します。

`timeout > 0` の場合、最大待機時間を秒で指定します。`timeout <= 0` の場合、この関数の呼び出しはブロックせず、現在準備できているファイルオブジェクトを報告します。*timeout* が `None` の場合、監視しているファイルオブジェクトの一つが準備できるまでブロックします。

この関数は `(key, events)` タプルのリストを返します。準備できたファイルオブジェクトにつき 1 タプルです。

key は準備状態のファイルオブジェクトに対応する *SelectorKey* インスタンスです。*events* はそのファイルオブジェクトで準備が完了したイベントのビットマスクです。

注釈: このメソッドは、現在のプロセスで信号を受信した場合、どのファイルオブジェクトも準備完了にならないうちに、またはタイムアウトが経過する前に返ることがあります。その場合、空のリストが返されます。

バージョン 3.5 で変更: このセレクタは、シグナルによって中断された時に、シグナルハンドラが例外を起こさなかった場合、空のイベントリストを返すのではなく、再計算されたタイムアウトによってリトライするようになりました (この論拠については **PEP 475** を参照してください)。

close()
セレクタを閉じます。

下層のリソースがすべて解放されたことを確かめるために呼ばれなければなりません。一旦閉じられたセレクタは使ってはいけません。

get_key(*fileobj*)
登録されたファイルオブジェクトに関連付けられたキーを返します。

そのファイルオブジェクトに関連付けられた *SelectorKey* インスタンスを返します。そのファイルオブジェクトが登録されていない場合 *KeyError* を送出します。

abstractmethod get_map()
ファイルオブジェクトからセレクタキーへのマッピングを返します。

これは、登録済みのファイルオブジェクトを、それらに関連づけられた *SelectorKey* インスタンスにマッピングする *Mapping* のインスタンスを返します。

class selectors.DefaultSelector
デフォルトの selector クラスで、現在のプラットフォームで利用できる最も効率的な実装を使用しています。大半のユーザはこれをデフォルトにすべきです。

class selectors.SelectSelector
select.select() を基底とするセレクタです。

class selectors.PollSelector
select.poll() を基底とするセレクタです。

class selectors.EpollSelector
select.epoll() を基底とするセレクタです。

fileno()
下層の *select.epoll()* オブジェクトが使用しているファイル記述子を返します。

class selectors.DevpollSelector
select.devpoll() を基底とするセレクタです。

fileno()
下層の *select.devpoll()* オブジェクトが使用しているファイル記述子を返します。

バージョン 3.5 で追加.

class selectors.KqueueSelector

select.kqueue() を基底とするセレクタです。

fileno()

下層の *select.kqueue()* オブジェクトが使用しているファイル記述子を返します。

18.5.3 使用例

簡単なエコーサーバの実装です:

```
import selectors
import socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept()  # Should be ready
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)

def read(conn, mask):
    data = conn.recv(1000)  # Should be ready
    if data:
        print('echoing', repr(data), 'to', conn)
        conn.send(data)  # Hope it won't block
    else:
        print('closing', conn)
        sel.unregister(conn)
        conn.close()

sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

18.6 asyncore --- 非同期ソケットハンドラ

ソースコード: Lib/asyncore.py

バージョン 3.6 で非推奨: 代わりに *asyncio* を使ってください。

注釈: このモジュールは後方互換性のためだけに存在します。新しいコードでは *asyncio* を利用することを推奨します。

このモジュールは、非同期ソケットサービスのクライアント・サーバを開発するための基盤として使われます。

CPU が一つしかない場合、プログラムが" 二つのことを同時に" 実行する方法は二つしかありません。もっとも簡単で一般的なのはマルチスレッドを利用する方法ですが、これとはまったく異なるテクニックで、一つのスレッドだけでマルチスレッドと同じような効果を得られるテクニックがあります。このテクニックは I/O 処理が中心である場合にのみ有効で、CPU 負荷の高いプログラムでは効果が無く、この場合にはプリエンプティブなスケジューリングが可能なスレッドが有効でしょう。しかし、多くの場合、ネットワークサーバでは CPU 負荷よりは IO 負荷が問題となります。

もし OS の I/O ライブラリがシステムコール `select()` をサポートしている場合（ほとんどの場合はサポートされている)、I/O 処理は" バックグラウンド" で実行し、その間に他の処理を実行すれば、複数の通信チャネルを同時にこなすことができます。一見、この戦略は奇妙で複雑に思えるかもしれませんが、いろいろな面でマルチスレッドよりも理解しやすく、制御も容易です。*asyncore* は多くの複雑な問題を解決済みなので、洗練され、パフォーマンスにも優れたネットワークサーバとクライアントを簡単に開発することができます。とくに、*asynchat* のような、対話型のアプリケーションやプロトコルには非常に有効でしょう。

基本的には、この二つのモジュールを使う場合は一つ以上のネットワーク **チャネル** を *asyncore.dispatcher* クラス、または *asynchat.async_chat* のインスタンスとして作成します。作成されたチャネルはグローバルマップに登録され、*loop()* 関数で参照されます。*loop()* には、専用の **マップ** を渡す事も可能です。

チャネルを生成後、*loop()* を呼び出すとチャネル処理が開始し、最後のチャネル（非同期処理中にマップに追加されたチャネルを含む）が閉じるまで継続します。

`asyncore.loop`([*timeout*[, *use_poll*[, *map*[, *count*]]]])

ポーリングループを開始し、count 回が過ぎるか、全てのオープン済みチャネルがクローズされた場合のみ終了します。全ての引数はオプションです。引数 *count* のデフォルト値は `None` で、ループは全てのチャネルがクローズされた場合のみ終了します。引数 *timeout* は *select()* または *poll()* の引数 timeout として渡され、秒単位で指定します。デフォルト値は 30 秒です。引数 *use_poll* が真の場合、*select()* ではなく *poll()* が使われます (デフォルト値は `False` です)。

引数 *map* には、監視するチャネルをアイテムとして格納した辞書を指定します。チャネルがクローズされた時に *map* からそのチャネルが削除されます。*map* が省略された場合、グローバルなマップが使用されます。チャネル (*asyncore.dispatcher*, *asynchat.async_chat* とそのサブクラス) は自由に混ぜて map に入れることができます。

`class asyncore.dispatcher`

dispatcher クラスは、低レベルソケットオブジェクトの薄いラッパーです。便宜上、非同期ループか

ら呼び出されるイベント処理メソッドを追加していますが、これ以外の点では、non-blocking なソケットと同様です。

非同期ループ内で低レベルイベントが発生した場合、発生のタイミングやコネクションの状態から特定の高レベルイベントへと置き換えることができます。例えばソケットを他のホストに接続する場合、最初の書き込み可能イベントが発生すれば接続が完了した事が分かります (この時点で、ソケットへの書き込みは成功すると考えられる)。このように判定できる高レベルイベントを以下に示します:

Event	説明
`handle_connect()`	最初に read もしくは write イベントが発生した時
`handle_close()`	読み込み可能なデータなしで read イベントが発生した時
`handle_accepted()`	listen 中のソケットで read イベントが発生した時

非同期処理中、マップに登録されたチャネルの *`readable()`* メソッドと *`writable()`* メソッドが呼び出され、`select()` か `poll()` で read/write イベントを検出するリストに登録するか否かを判定します。

このようにして、チャネルでは低レベルなソケットイベントの種類より多くの種類のイベントを検出する事ができます。以下にあげるイベントは、サブクラスでオーバライドすることが可能です:

`handle_read()`
非同期ループで、チャネルのソケットの `read()` メソッドの呼び出しが成功した時に呼び出されます。

`handle_write()`
非同期ループで、書き込み可能ソケットが実際に書き込み可能になった時に呼び出されます。このメソッドでは、しばしばパフォーマンスの向上のために必要なバッファリングを実装します。例:

```
def handle_write(self):
    sent = self.send(self.buffer)
    self.buffer = self.buffer[sent:]
```

`handle_expt()`
out of band (OOB) データが検出された時に呼び出されます。OOB はあまりサポートされておらず、また滅多に使われないので、`handle_expt()` が呼び出されることはほとんどありません。

`handle_connect()`
ソケットの接続が確立した時に呼び出されます。”welcome” バナーの送信、プロトコルネゴシエーションの初期化などを行います。

`handle_close()`
ソケットが閉じた時に呼び出されます。

`handle_error()`
捕捉されない例外が発生した時に呼び出されます。デフォルトでは、短縮したトレースバック情報が出力されます。

handle_accept()

listen 中のチャネル (受動的にオープンしたもの) がリモートホストからの *connect()* で接続され、接続が確立した時に呼び出されます。バージョン 3.2 で非推奨になりました; 代わりに *handle_accepted()* を使ってください。

バージョン 3.2 で非推奨.

handle_accepted(*sock*, *addr*)

listen 中のチャネル (受動的にオープンしたもの) がリモートホストからの *connect()* で接続され、接続が確立した時に呼び出されます。*sock* はその接続でデータを送受信するのに使える **新しい** ソケットオブジェクトで、*addr* は接続の対向のソケットに bind されているアドレスです。

バージョン 3.2 で追加.

readable()

非同期ループ中に呼び出され、read イベントの監視リストに加えるか否かを決定します。デフォルトのメソッドでは `True` を返し、read イベントの発生を監視します。

writable()

非同期ループ中に呼び出され、write イベントの監視リストに加えるか否かを決定します。デフォルトのメソッドでは `True` を返し、write イベントの発生を監視します。

さらに、チャネルにはソケットのメソッドとほぼ同じメソッドがあり、チャネルはソケットのメソッドの多くを委譲・拡張しており、ソケットとほぼ同じメソッドを持っています。

create_socket(*family=socket.AF_INET*, *type=socket.SOCK_STREAM*)

引数も含め、通常のソケット生成と同一です。ソケットの生成については、*socket* モジュールのドキュメントを参照してください。

バージョン 3.3 で変更: *family* 引数と *type* 引数が省略可能になりました。

connect(*address*)

通常のソケットオブジェクトと同様、*address* には一番目の値が接続先ホスト、2 番目の値がポート番号であるタプルを指定します。

send(*data*)

リモート側の端点に *data* を送出します。

recv(*buffer_size*)

リモート側の端点より、最大 *buffer_size* バイトのデータを読み込みます。長さ 0 のバイト列オブジェクトが返ってきた場合、チャネルはリモートから切断された事を示します。

select.select() や *select.poll()* がソケットが読み込みできる状態にあると報告したとしても、*recv()* が *BlockingIOError* を送出する場合があります。

listen(*backlog*)

ソケットへの接続を待ちます。引数 *backlog* は、キューに追加できるコネクションの最大数 (1 以上) を指定します。最大値はシステムに依存します（通常は 5)。

bind(*address*)

ソケットを *address* にバインドします。ソケットはバインド済みであってはなりません。(*address* の形式は、アドレスファミリに依存します。*socket* モジュールを参照のこと。) ソケットを再利用可能にする (SO_REUSEADDR オプションを設定する) には、*dispatcher* オブジェクトの set_reuse_addr() メソッドを呼び出してください。

accept()
接続を受け入れます。ソケットはアドレスにバインド済みであり、listen() で接続待ち状態でなければなりません。戻り値は None か (conn, address) のペアで、*conn* はデータの送受信を行う **新しい** ソケットオブジェクト、*address* は接続先ソケットがバインドされているアドレスです。None が返された場合、接続が起こらなかったことを意味します。その場合、サーバーはこのイベントを無視して後続の接続を待ち続けるべきです。

close()
ソケットをクローズします。以降の全ての操作は失敗します。リモート端点では、キューに溜まったデータ以外、これ以降のデータ受信は行えません。ソケットはガベージコレクト時に自動的にクローズされます。

class asyncore.dispatcher_with_send
dispatcher のサブクラスで、シンプルなバッファされた出力機能を持ちます。シンプルなクライアントプログラムに適しています。もっと高レベルな場合には *asynchat.async_chat* を利用してください。

class asyncore.file_dispatcher
file_dispatcher はファイルデスクリプタか **ファイルオブジェクト** とオプションとして map を引数にとって、poll() か loop() 関数で利用できるようにラップします。与えられたファイルオブジェクトなどが fileno() メソッドを持っているとき、そのメソッドが呼び出されて戻り値が *file_wrapper* のコンストラクタに渡されます。

利用可能な環境: Unix。

class asyncore.file_wrapper
file_wrapper は整数のファイルデスクリプタを受け取って *os.dup()* を呼び出してハンドルを複製するので、元のハンドルは file_wrapper と独立して close されます。このクラスは *file_dispatcher* クラスが使うために必要なソケットをエミュレートするメソッドを実装しています。

利用可能な環境: Unix。

18.6.1 asyncore の例: 簡単な HTTP クライアント

基本的なサンプルとして、以下に非常に単純な HTTP クライアントを示します。この HTTP クライアントは *dispatcher* クラスでソケットを利用しています:

```
import asyncore

class HTTPClient(asyncore.dispatcher):

    def __init__(self, host, path):
```

(次のページに続く)

(前のページからの続き)

```
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.connect( (host, 80) )
        self.buffer = bytes('GET %s HTTP/1.0\r\nHost: %s\r\n\r\n' %
                            (path, host), 'ascii')

    def handle_connect(self):
        pass

    def handle_close(self):
        self.close()

    def handle_read(self):
        print(self.recv(8192))

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]


client = HTTPClient('www.python.org', '/')
asyncore.loop()
```

18.6.2 基本的な echo サーバーの例

この例の基本的な echo サーバーは、*dispatcher* を利用して接続を受けつけ、接続をハンドラーにディスパッチします:

```
import asyncore

class EchoHandler(asyncore.dispatcher_with_send):

    def handle_read(self):
        data = self.recv(8192)
        if data:
            self.send(data)

class EchoServer(asyncore.dispatcher):

    def __init__(self, host, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.set_reuse_addr()
        self.bind((host, port))
        self.listen(5)
```

(次のページに続く)

(前のページからの続き)

```
    def handle_accepted(self, sock, addr):
        print('Incoming connection from %s' % repr(addr))
        handler = EchoHandler(sock)

server = EchoServer('localhost', 8080)
asyncore.loop()
```

18.7 asynchat --- 非同期ソケットコマンド/レスポンスハンドラ

ソースコード: Lib/asynchat.py

バージョン 3.6 で非推奨: 代わりに *asyncio* を使ってください。

注釈: このモジュールは後方互換性のためだけに存在します。新しいコードでは *asyncio* を利用することを推奨します。

---

asynchat を使うと、*asyncore* を基盤とした非同期なサーバ・クライアントをより簡単に開発する事ができます。*asynchat* では、プロトコルの要素が任意の文字列で終了するか、または可変長の文字列であるようなプロトコルを容易に制御できるようになっています。*asynchat* は、抽象クラス *async_chat* を定義しており、*async_chat* を継承して `collect_incoming_data()` メソッドと `found_terminator()` メソッドを実装すれば使うことができます。*async_chat* と *asyncore* は同じ非同期ループを使用しており、*asyncore.dispatcher* も *asynchat.async_chat* も同じチャネルマップに登録する事ができます。通常、*asyncore.dispatcher* はサーバチャネルとして使用し、リクエストの受け付け時に *asynchat.async_chat* オブジェクトを生成します。

class asynchat.async_chat

このクラスは、*asyncore.dispatcher* から継承した抽象クラスです。使用する際には *async_chat* のサブクラスを作成し、*collect_incoming_data()* と *found_terminator()* を定義しなければなりません。*asyncore.dispatcher* のメソッドを使用する事もできますが、メッセージ/レスポンス処理を中心に行う場合には使えないメソッドもあります。

asyncore.dispatcher と同様に、*async_chat* も `select()` 呼出し後のソケットの状態からイベントを生成します。ポーリングループ開始後、イベント処理フレームワークが自動的に *async_chat* のメソッドを呼び出しますので、プログラマが処理を記述する必要はありません。

パフォーマンスの向上やメモリの節約のために、2 つのクラス属性を調整することができます。

ac_in_buffer_size

非同期入力バッファサイズ (デフォルト値: 4096)。

ac_out_buffer_size

非同期出力バッファサイズ (デフォルト値: 4096)。

asyncore.dispatcher と違い、*async_chat* では *producer* の FIFO キューを作成する事ができます。producer は `more()` メソッドを必ず持ち、このメソッドでチャネル上に送出するデータを返します。producer が枯渇状態 (*i.e.* これ以上のデータを持たない状態) にある場合、`more()` は空のバイトオブジェクトを返します。この時、*async_chat* は枯渇状態にある producer をキューから除去し、次の producer が存在すればその producer を使用します。キューに producer が存在しない場合、`handle_write()` は何もしません。リモート端点からの入力の終了や重要な中断点を検出する場合は、*set_terminator()* に記述します。

async_chat のサブクラスでは、入力メソッド *collect_incoming_data()* と *found_terminator()* を定義し、チャネルが非同期に受信するデータを処理します。これらのメソッドについては後ろで解説します。

async_chat.close_when_done()
producer キューのトップに `None` をプッシュします。この producer がキューからポップされると、チャネルがクローズします。

async_chat.collect_incoming_data(*data*)
チャネルが受信した不定長のデータを *data* に指定して呼び出されます。このメソッドは必ずオーバライドする必要があり、デフォルトの実装では、*NotImplementedError* 例外を送出します。

async_chat.discard_buffers()
非常用のメソッドで、全ての入出力バッファと producer キューを廃棄します。

async_chat.found_terminator()
入力データストリームが、*set_terminator()* で指定した終了条件と一致した場合に呼び出されます。このメソッドは必ずオーバライドする必要があり、デフォルトの実装では、*NotImplementedError* 例外を送出します。入力データを参照する必要がある場合でも引数としては与えられないため、入力バッファをインスタンス属性として参照しなければなりません。

async_chat.get_terminator()
現在のチャネルの終了条件を返します。

async_chat.push(*data*)
チャネルのキューにデータをプッシュして転送します。データをチャネルに書き出すために必要なのはこれだけですが、データの暗号化やチャンク化などを行う場合には独自の producer を使用する事もできます。

async_chat.push_with_producer(*producer*)
指定した producer オブジェクトをチャネルのキューに追加します。これより前に push された producer が全て枯渇した後、チャネルはこの producer から `more()` メソッドでデータを取得し、リモート端点に送信します。

async_chat.set_terminator(*term*)
チャネルで検出する終了条件を設定します。`term` は入力プロトコルデータの処理方式によって以下の 3 つの型の何れかを指定します。

term	説明
string	入力ストリーム中で string が検出された時、*found_terminator()* を呼び出します
integer	指定された文字数が読み込まれた時、*found_terminator()* を呼び出します
None	永久にデータを読み込みます

終了条件が成立しても、その後に続くデータは、*found_terminator()* の呼出し後に再びチャネルを読み込めば取得する事ができます。

18.7.1 asynchat 使用例

以下のサンプルは、*async_chat* で HTTP リクエストを読み込む処理の一部です。Web サーバは、クライアントからの接続毎に http_request_handler オブジェクトを作成します。最初はチャネルの終了条件に空行を指定して HTTP ヘッダの末尾までを検出し、その後ヘッダ読み込み済みを示すフラグを立てています。

ヘッダ読み込んだ後、リクエストの種類が POST であればデータが入力ストリームに流れるため、Content-Length: ヘッダの値を数値として終了条件に指定し、適切な長さのデータをチャネルから読み込みます。

必要な入力データを全て入手したら、チャネルの終了条件に None を指定して残りのデータを無視するようにしています。この後、handle_request() が呼び出されます。

```
import asynchat

class http_request_handler(asynchat.async_chat):

    def __init__(self, sock, addr, sessions, log):
        asynchat.async_chat.__init__(self, sock=sock)
        self.addr = addr
        self.sessions = sessions
        self.ibuffer = []
        self.obuffer = b""
        self.set_terminator(b"\r\n\r\n")
        self.reading_headers = True
        self.handling = False
        self.cgi_data = None
        self.log = log

    def collect_incoming_data(self, data):
        """Buffer the data"""
        self.ibuffer.append(data)

    def found_terminator(self):
        if self.reading_headers:
            self.reading_headers = False
            self.parse_headers(b"".join(self.ibuffer))
            self.ibuffer = []
            if self.op.upper() == b"POST":
                clen = self.headers.getheader("content-length")
```

(次のページに続く)

(前のページからの続き)

```
                self.set_terminator(int(clen))
            else:
                self.handling = True
                self.set_terminator(None)
                self.handle_request()
        elif not self.handling:
            self.set_terminator(None)  # browsers sometimes over-send
            self.cgi_data = parse(self.headers, b"".join(self.ibuffer))
            self.handling = True
            self.ibuffer = []
            self.handle_request()
```

18.8 signal --- 非同期イベントにハンドラを設定する

このモジュールでは Python でシグナルハンドラを使うための機構を提供します。

18.8.1 一般的なルール

The *signal.signal()* function allows defining custom handlers to be executed when a signal is received. A small number of default handlers are installed: *SIGPIPE* is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions) and *SIGINT* is translated into a *KeyboardInterrupt* exception if the parent process has not changed it.

特定のシグナルに対するハンドラが一度設定されると、明示的にリセットしないかぎり設定されたままになります (Python は背後の実装系に関係なく BSD 形式のインタフェースをエミュレートします)。例外は *SIGCHLD* のハンドラで、この場合は背後の実装系の仕様に従います。

Python のシグナルハンドラの実行

Python のシグナルハンドラは、低水準 (C 言語) のシグナルハンドラ内で実行されるわけではありません。代わりに、低水準のシグナルハンドラが *virtual machine* が対応する Python のシグナルハンドラを後から (例えば次の *bytecode* 命令時に) 実行するようにフラグを立てます:

- It makes little sense to catch synchronous errors like *SIGFPE* or *SIGSEGV* that are caused by an invalid operation in C code. Python will return from the signal handler to the C code, which is likely to raise the same signal again, causing Python to apparently hang. From Python 3.3 onwards, you can use the *faulthandler* module to report on synchronous errors.
- 完全に C で実装された長時間かかる計算 (大きいテキストに対する正規表現のマッチなど) は、どのシグナルを受信しても中断されないまま長時間実行され続ける可能性があります。Python のシグナルハンドラはその計算が終了してから呼び出されます。

シグナルとスレッド

Python のシグナルハンドラは、もしシグナルを受け取ったのが別のスレッドだったとしても、常に Python のメインスレッドで実行されます。このためシグナルをスレッド間通信に使うことはできません。代わりに *threading* モジュールが提供している同期プリミティブを利用できます。

また、メインスレッドだけが新しいシグナルハンドラを登録できます。

18.8.2 モジュールの内容

バージョン 3.5 で変更: signal (SIG*), handler (*SIG_DFL*, *SIG_IGN*) and sigmask (*SIG_BLOCK*, *SIG_UNBLOCK*, *SIG_SETMASK*) related constants listed below were turned into *enums*. *getsignal()*, *pthread_sigmask()*, *sigpending()* and *sigwait()* functions return human-readable *enums*.

以下に *signal* モジュールで定義されている変数を示します:

signal.SIG_DFL
二つある標準シグナル処理オプションのうちの一つです; 単純にシグナルに対する標準の関数を実行します。例えば、ほとんどのシステムでは、SIGQUIT に対する標準の動作はコアダンプと終了で、*SIGCHLD* に対する標準の動作は単にシグナルの無視です。

signal.SIG_IGN
もう一つの標準シグナル処理オプションで、受け取ったシグナルを単に無視します。

signal.SIGABRT
Abort signal from *abort(3)*.

signal.SIGALRM
Timer signal from *alarm(2)*.

利用可能な環境: Unix。

signal.SIGBREAK
Interrupt from keyboard (CTRL + BREAK).

利用可能な環境: Windows 。

signal.SIGBUS
Bus error (bad memory access).

利用可能な環境: Unix。

signal.SIGCHLD
Child process stopped or terminated.

利用可能な環境: Unix。

signal.SIGCLD
SIGCHLD のエイリアスです。

signal.SIGCONT
: Continue the process if it is currently stopped

  利用可能な環境: Unix。

signal.SIGFPE
: Floating-point exception. For example, division by zero.

  参考:

  ZeroDivisionError is raised when the second argument of a division or modulo operation is zero.

signal.SIGHUP
: Hangup detected on controlling terminal or death of controlling process.

  利用可能な環境: Unix。

signal.SIGILL
: Illegal instruction.

signal.SIGINT
: Interrupt from keyboard (CTRL + C).

  Default action is to raise *KeyboardInterrupt*.

signal.SIGKILL
: Kill signal.

  It cannot be caught, blocked, or ignored.

  利用可能な環境: Unix。

signal.SIGPIPE
: Broken pipe: write to pipe with no readers.

  Default action is to ignore the signal.

  利用可能な環境: Unix。

signal.SIGSEGV
: Segmentation fault: invalid memory reference.

signal.SIGTERM
: Termination signal.

signal.SIGUSR1
: User-defined signal 1.

  利用可能な環境: Unix。

signal.SIGUSR2
: User-defined signal 2.

  利用可能な環境: Unix。

signal.SIGWINCH
: Window resize signal.

  利用可能な環境: Unix。

SIG*
: 全てのシグナル番号はシンボル定義されています。例えば、ハングアップシグナルは *signal.SIGHUP* で定義されています; 変数名は C 言語のプログラムで使われているのと同じ名前で、<signal.h> にあります。'signal()' に関する Unix マニュアルページでは、システムで定義されているシグナルを列挙しています (あるシステムではリストは *signal(2)* に、別のシステムでは *signal(7)* に列挙されています)。全てのシステムで同じシグナル名のセットを定義しているわけではないので注意してください; このモジュールでは、システムで定義されているシグナル名だけを定義しています。

signal.CTRL_C_EVENT
: CTRL+C キーストロークに該当するシグナル。このシグナルは *os.kill()* でだけ利用できます。

  利用可能な環境: Windows 。

  バージョン 3.2 で追加.

signal.CTRL_BREAK_EVENT
: CTRL+BREAK キーストロークに該当するシグナル。このシグナルは *os.kill()* でだけ利用できます。

  利用可能な環境: Windows 。

  バージョン 3.2 で追加.

signal.NSIG
: 最も大きいシグナル番号に 1 を足した値です。

signal.ITIMER_REAL
: 実時間でデクリメントするインターバルタイマーです。タイマーが発火したときに *SIGALRM* を送ります。

signal.ITIMER_VIRTUAL
: プロセスの実行時間だけデクリメントするインターバルタイマーです。タイマーが発火したときに SIGVTALRM を送ります。

signal.ITIMER_PROF
: プロセスの実行中と、システムがそのプロセスのために実行している時間だけデクリメントするインターバルタイマーです。ITIMER_VIRTUAL と組み合わせて、このタイマーはよくアプリケーションがユーザー空間とカーネル空間で消費した時間のプロファイリングに利用されます。タイマーが発火したときに SIGPROF を送ります。

signal.SIG_BLOCK
: *pthread_sigmask()* の *how* 引数に渡せる値で、シグナルがブロックされることを意味します。

  バージョン 3.3 で追加.

signal.SIG_UNBLOCK
: *pthread_sigmask()* の *how* 引数に渡せる値で、シグナルがブロック解除されることを意味します。

バージョン 3.3 で追加.

signal.SIG_SETMASK
: *pthread_sigmask()* の *how* 引数に渡せる値で、シグナルが置換されることを意味します。

  バージョン 3.3 で追加.

signal モジュールは 1 つの例外を定義しています:

exception signal.ItimerError
: 背後の *setitimer()* または *getitimer()* 実装からエラーを通知するために送出されます。無効なインタバルタイマーや負の時間が *setitimer()* に渡された場合、このエラーを予期してください。このエラーは *OSError* を継承しています。

  バージョン 3.3 で追加: このエラーは以前は *IOError* のサブタイプでしたが、*OSError* のエイリアスになりました。

signal モジュールでは以下の関数を定義しています:

signal.alarm(*time*)
: *time* がゼロでない値の場合、この関数は *time* 秒後頃に *SIGALRM* をプロセスに送るように要求します。それ以前にスケジュールしたアラームはキャンセルされます (常に一つのアラームしかスケジュールできません)。この場合、戻り値は以前に設定されたアラームシグナルが通知されるまであと何秒だったかを示す値です。*time* がゼロの場合、アラームは一切スケジュールされず、現在スケジュールされているアラームがキャンセルされます。戻り値がゼロの場合、現在アラームがスケジュールされていないことを示します。

  利用可能な環境: Unix。さらに詳しい情報についてはオンラインマニュアルページ *alarm(2)* を参照してください。

signal.getsignal(*signalnum*)
: シグナル *signalnum* に対する現在のシグナルハンドラを返します。戻り値は呼び出し可能な Python オブジェクトか、*signal.SIG_IGN*、*signal.SIG_DFL*、および *None* といった特殊な値のいずれかです。ここで *signal.SIG_IGN* は以前そのシグナルが無視されていたことを示し、*signal.SIG_DFL* は以前そのシグナルの標準の処理方法が使われていたことを示し、None はシグナルハンドラがまだ Python によってインストールされていないことを示します。

signal.strsignal(*signalnum*)
: Return the system description of the signal *signalnum*, such as "Interrupt", "Segmentation fault", etc. Returns *None* if the signal is not recognized.

  バージョン 3.8 で追加.

signal.valid_signals()
: Return the set of valid signal numbers on this platform. This can be less than `range(1, NSIG)` if some signals are reserved by the system for internal use.

  バージョン 3.8 で追加.

signal.pause()

シグナルを受け取るまでプロセスを一時停止します; その後、適切なハンドラが呼び出されます。戻り値はありません。

利用可能な環境: Unix。さらに詳しい情報についてはオンラインマニュアルページ *signal(2)* を参照してください。

sigwait(), *sigwaitinfo()*, *sigtimedwait()* *sigpending()* も参照してください。

signal.**raise_signal**(*signum*)

Sends a signal to the calling process. Returns nothing.

バージョン 3.8 で追加.

signal.**pthread_kill**(*thread_id*, *signalnum*)

Send the signal *signalnum* to the thread *thread_id*, another thread in the same process as the caller. The target thread can be executing any code (Python or not). However, if the target thread is executing the Python interpreter, the Python signal handlers will be *executed by the main thread*. Therefore, the only point of sending a signal to a particular Python thread would be to force a running system call to fail with *InterruptedError*.

Use *threading.get_ident()* or the *ident* attribute of *threading.Thread* objects to get a suitable value for *thread_id*.

If *signalnum* is 0, then no signal is sent, but error checking is still performed; this can be used to check if the target thread is still running.

引数 thread_id, signalnum を指定して **監査イベント** signal.pthread_kill を送出します。

利用可能な環境: Unix。さらに詳しい情報についてはオンラインマニュアルページ *pthread_kill(3)* を参照してください。

os.kill() を参照してください。

バージョン 3.3 で追加.

signal.**pthread_sigmask**(*how*, *mask*)

これを呼び出すスレッドにセットされているシグナルマスクを取り出したり変更したりします。シグナルマスクは、呼び出し側のために現在どのシグナルの配送がブロックされているかを示す集合 (set) です。呼び出し前のもとのシグナルマスクを集合として返却します。

この関数の振る舞いは *how* に依存して以下のようになります。

- *SIG_BLOCK*: *mask* で指定されるシグナルが現時点のシグナルマスクに追加されます。
- *SIG_UNBLOCK*: *mask* で指定されるシグナルが現時点のシグナルマスクから取り除かれます。もともとブロックされていないシグナルをブロック解除しようとしても問題ありません。
- *SIG_SETMASK*: シグナルマスク全体を *mask* としてセットします。

mask はシグナル番号の集合です (例えば {*signal.SIGINT*, *signal.SIGTERM*})。全てのシグナルを含む全集合として *valid_signals()* を使うことが出来ます。

呼び出しスレッドにセットされたシグナルマスクを問い合わせるには例えば `signal.pthread_sigmask(signal.SIG_BLOCK, [])` とします。

SIGKILL and SIGSTOP cannot be blocked.

利用可能な環境：Unix。さらに詳しい情報についてはオンラインマニュアルページ *sigprocmask(3)* と *pthread_sigmask(3)* を参照してください。

pause(), *sigpending()*, *sigwait()* も参照して下さい。

バージョン 3.3 で追加.

signal.setitimer(*which*, *seconds*, *interval=0.0*)

which で指定されたタイマー (*signal.ITIMER_REAL*, *signal.ITIMER_VIRTUAL*, *signal.ITIMER_PROF* のどれか) を、*seconds* 秒後と (*alarm()* と異なり、float を指定できます)、それから (*interval* が 0 でなければ) *interval* 秒間隔で起動するように設定します。*seconds* に 0 を指定すると、*which* で指定されたタイマーをクリアすることができます。

インターバルタイマーが起動したとき、シグナルがプロセスに送られます。送られるシグナルは利用されたタイマーの種類に依存します。*signal.ITIMER_REAL* の場合は *SIGALRM* が、*signal.ITIMER_VIRTUAL* の場合は SIGVTALRM が、*signal.ITIMER_PROF* の場合は SIGPROF が送られます。

以前の値が (delay, interval) のタプルとして返されます。

無効なインターバルタイマーを渡すと *ItimerError* 例外が発生します。

利用可能な環境: Unix。

signal.getitimer(*which*)

which で指定されたインターバルタイマーの現在の値を返します。

利用可能な環境: Unix。

signal.set_wakeup_fd(*fd*, ***, *warn_on_full_buffer=True*)

Set the wakeup file descriptor to *fd*. When a signal is received, the signal number is written as a single byte into the fd. This can be used by a library to wakeup a poll or select call, allowing the signal to be fully processed.

The old wakeup fd is returned (or -1 if file descriptor wakeup was not enabled). If *fd* is -1, file descriptor wakeup is disabled. If not -1, *fd* must be non-blocking. It is up to the library to remove any bytes from *fd* before calling poll or select again.

スレッドが有効な場合、この関数はメインスレッドからしか実行できません。それ以外のスレッドからこの関数を実行しようとすると *ValueError* 例外が発生します。

There are two common ways to use this function. In both approaches, you use the fd to wake up when a signal arrives, but then they differ in how they determine *which* signal or signals have arrived.

In the first approach, we read the data out of the fd's buffer, and the byte values give you the signal numbers. This is simple, but in rare cases it can run into a problem: generally the fd will

have a limited amount of buffer space, and if too many signals arrive too quickly, then the buffer may become full, and some signals may be lost. If you use this approach, then you should set `warn_on_full_buffer=True`, which will at least cause a warning to be printed to stderr when signals are lost.

In the second approach, we use the wakeup fd *only* for wakeups, and ignore the actual byte values. In this case, all we care about is whether the fd's buffer is empty or non-empty; a full buffer doesn't indicate a problem at all. If you use this approach, then you should set `warn_on_full_buffer=False`, so that your users are not confused by spurious warning messages.

バージョン 3.5 で変更: Windows で、この関数はソケットハンドルをサポートするようになりました。

バージョン 3.7 で変更: “warn_on_full_buffer“引数が追加されました。

`signal.siginterrupt`(*signalnum*, *flag*)

システムコールのリスタートの動作を変更します。*flag* が *False* の場合、*signalnum* シグナルに中断されたシステムコールは再実行されます。それ以外の場合、システムコールは中断されます。戻り値はありません。

利用可能な環境: Unix。さらに詳しい情報についてはオンラインマニュアルページ *siginterrupt(3)* を参照してください。

signal() を使ってシグナルハンドラを設定したときに、暗黙のうちに *flag* に true を指定して `siginterrupt()` が実行されるため、中断に対するリスタートの動作がリセットされることに注意してください。

`signal.signal`(*signalnum*, *handler*)

シグナル *signalnum* に対するハンドラを関数 *handler* にします。*handler* は二つの引数 (下記参照) を取る呼び出し可能な Python オブジェクトか、*signal.SIG_IGN* あるいは *signal.SIG_DFL* といった特殊な値にすることができます。以前に使われていたシグナルハンドラが返されます (上記の *getsignal()* の記述を参照してください)。(さらに詳しい情報については Unix マニュアルページ *signal(2)* を参照してください。)

スレッドが有効な場合、この関数はメインスレッドからしか実行できません。それ以外のスレッドからこの関数を実行しようとすると *ValueError* 例外が発生します。

handler は二つの引数とともに呼び出されます: シグナル番号、および現在のスタックフレーム (`None` またはフレームオブジェクト; フレームオブジェクトについての記述は 標準型の階層における説明 か、*inspect* モジュールの属性の説明を参照してください)。

On Windows, *signal()* can only be called with *SIGABRT*, *SIGFPE*, *SIGILL*, *SIGINT*, *SIGSEGV*, *SIGTERM*, or *SIGBREAK*. A *ValueError* will be raised in any other case. Note that not all systems define the same set of signal names; an *AttributeError* will be raised if a signal name is not defined as `SIG*` module level constant.

`signal.sigpending`()

呼び出しスレッドで配送が保留されているシグナル (つまり配送がブロックされている間に発生したシグナル) の集合を調べます。保留中のシグナルの集合を返します。

利用可能な環境: Unix。さらに詳しい情報についてはオンラインマニュアルページ *sigpending(2)* を参照してください。

pause(), *pthread_sigmask()*, *sigwait()* も参照して下さい。

バージョン 3.3 で追加.

signal.**sigwait**(*sigset*)

sigset 集合で指定されたシグナルのうちどれか一つが届くまで呼び出しスレッドを一時停止します。この関数はそのシグナルを受け取ると (それを保留シグナルリストから取り除いて) そのシグナル番号を返します。

利用可能な環境: Unix。さらに詳しい情報についてはオンラインマニュアルページ *sigwait(3)* を参照してください。

pause(), *pthread_sigmask()*, *sigpending()*, *sigwaitinfo()*, *sigtimedwait()* も参照して下さい。

バージョン 3.3 で追加.

signal.**sigwaitinfo**(*sigset*)

Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal and removes it from the pending list of signals. If one of the signals in *sigset* is already pending for the calling thread, the function will return immediately with information about that signal. The signal handler is not called for the delivered signal. The function raises an *InterruptedError* if it is interrupted by a signal that is not in *sigset*.

The return value is an object representing the data contained in the `siginfo_t` structure, namely: `si_signo`, `si_code`, `si_errno`, `si_pid`, `si_uid`, `si_status`, `si_band`.

利用可能な環境: Unix。さらに詳しい情報についてはオンラインマニュアルページ *sigwaitinfo(2)* を参照してください。

pause(), *sigwait()*, *sigtimedwait()* も参照して下さい。

バージョン 3.3 で追加.

バージョン 3.5 で変更: The function is now retried if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see **PEP 475** for the rationale).

signal.**sigtimedwait**(*sigset*, *timeout*)

Like *sigwaitinfo()*, but takes an additional *timeout* argument specifying a timeout. If *timeout* is specified as `0`, a poll is performed. Returns *None* if a timeout occurs.

利用可能な環境: Unix。さらに詳しい情報についてはオンラインマニュアルページ *sigtimedwait(2)* を参照してください。

pause(), *sigwait()*, *sigwaitinfo()* も参照して下さい。

バージョン 3.3 で追加.

バージョン 3.5 で変更: The function is now retried with the recomputed *timeout* if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see **PEP 475** for the rationale).

18.8.3 使用例

以下は最小限のプログラム例です。この例では *alarm()* を使ってファイルを開く処理を待つのに費やす時間を制限します; 例えば、電源の入っていないシリアルデバイスを開こうとすると、通常 *os.open()* は未定義の期間ハングアップしてしまいますが、この方法はそうした場合に便利です。ここではファイルを開くまで 5 秒間のアラームを設定することで解決しています; ファイルを開く処理が長くかかりすぎると、アラームシグナルが送信され、ハンドラが例外を送出するようになっています。

```
import signal, os

def handler(signum, frame):
    print('Signal handler called with signal', signum)
    raise OSError("Couldn't open device!")

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)          # Disable the alarm
```

18.8.4 Note on SIGPIPE

Piping output of your program to tools like *head(1)* will cause a *SIGPIPE* signal to be sent to your process when the receiver of its standard output closes early. This results in an exception like `BrokenPipeError: [Errno 32] Broken pipe`. To handle this case, wrap your entry point to catch this exception as follows:

```
import os
import sys

def main():
    try:
        # simulate large output (your code replaces this loop)
        for x in range(10000):
            print("y")
        # flush output here to force SIGPIPE to be triggered
        # while inside this try block.
        sys.stdout.flush()
    except BrokenPipeError:
        # Python flushes standard streams on exit; redirect remaining output
        # to devnull to avoid another BrokenPipeError at shutdown
```

(次のページに続く)

(前のページからの続き)

```
        devnull = os.open(os.devnull, os.O_WRONLY)
        os.dup2(devnull, sys.stdout.fileno())
        sys.exit(1)  # Python exits with error code 1 on EPIPE

if __name__ == '__main__':
    main()
```

Do not set *SIGPIPE*'s disposition to *SIG_DFL* in order to avoid *BrokenPipeError*. Doing that would cause your program to exit unexpectedly also whenever any socket connection is interrupted while your program is still writing to it.

18.9 mmap --- メモリマップファイル

メモリにマップされたファイルオブジェクトは、*bytearray* と **ファイルオブジェクト** の両方のように振舞います。しかし通常の文字列オブジェクトとは異なり、これらは可変です。*bytearray* が期待されるほとんどの場所で mmap オブジェクトを利用できます。例えば、メモリマップファイルを探索するために *re* モジュールを使うことができます。それらは可変なので、`obj[index] = 97` のように文字を変換できますし、スライスを使うことで `obj[i1:i2] = b'...'` のように部分文字列を変換することができます。現在のファイル位置をデータの始めとする読込みや書込み、ファイルの異なる位置へ `seek()` することもできます。

メモリマップドファイルは Unix と Windows で異なる *mmap* コンストラクタで生成されます。どちらの場合も、更新用に開かれたファイルディスクリプタを渡さなければなりません。既存の Python ファイルオブジェクトをマップしたければ、`fileno()` メソッドを使って *fileno* パラメータの正しい値を取得してください。そうでなければ、*os.open()* 関数を使ってファイルを開けます。この関数はファイルディスクリプタを直接返します (処理が終わったら、やはりファイルを閉じる必要があります)。

注釈: 書き込み可能でバッファされたファイルへのメモリマップファイルを作りたいのであれば、まず最初にファイルの *flush()* を呼び出すべきです。これはバッファへのローカルな修正がマッピングで実際に利用可能になることを保障するために必要です。

Unix バージョンと Windows バージョンのどちらのコンストラクタについても、オプションのキーワード・パラメータとして *access* を指定できます。*access* は 4 つの値の内の 1 つを受け入れます。`ACCESS_READ` は読み出し専用、`ACCESS_WRITE` は書き込み可能、`ACCESS_COPY` はコピーした上での書き込み、`ACCESS_DEFAULT` は *prot* に従います。*access* は Unix と Windows の両方で使用することができます。*access* が指定されない場合、Windows の mmap は書き込み可能マップを返します。3 つのアクセス型すべてに対する初期メモリ値は、指定されたファイルから得られます。`ACCESS_READ` 型のメモリマップに対して書き込むと *TypeError* 例外を送出します。`ACCESS_WRITE` 型のメモリマップへの書き込みはメモリと元のファイルの両方に影響を与えます。`ACCESS_COPY` 型のメモリマップへの書き込みはメモリに影響を与えますが、元のファイルを更新することはありません。

バージョン 3.7 で変更: 定数 `ACCESS_DEFAULT` が追加されました。

無名メモリ (anonymous memory) にマップするためには fileno として -1 を渡し、length を与えてください。

class mmap.**mmap**(*fileno*, *length*, *tagname=None*, *access=ACCESS_DEFAULT*[, *offset*])

(Windows バージョン) ファイルハンドル *fileno* によって指定されたファイルから *length* バイトをマップして、mmap オブジェクトを生成します。*length* が現在のファイルサイズより大きな場合、ファイルサイズは *length* を含む大きさにまで拡張されます。*length* が 0 の場合、マップの最大の長さは現在のファイルサイズになります。ただし、ファイル自体が空のときは Windows が例外を送出します (Windows では空のマップを作成することができません)。

tagname は、None 以外で指定された場合、マップのタグ名を与える文字列となります。Windows は同じファイルに対する様々なマップを持つことを可能にします。既存のタグの名前を指定すればそのタグがオープンされ、そうでなければこの名前の新しいタグが作成されます。もしこのパラメータを省略したり None を与えたりしたならば、マップは名前なしで作成されます。タグ・パラメータの使用の回避は、あなたのコードを Unix と Windows の間で移植可能にしておくのを助けてくれるでしょう。

offset は非負整数のオフセットとして指定できます。mmap の参照はファイルの先頭からのオフセットに相対的になります。*offset* のデフォルトは 0 です。*offset* は ALLOCATIONGRANULARITY の倍数でなければなりません。

引数 fileno, length, access, offset 付きで 監査イベント mmap.__new__ を送出します。

class mmap.**mmap**(*fileno*, *length*, *flags=MAP_SHARED*, *prot=PROT_WRITE|PROT_READ*, *access=ACCESS_DEFAULT*[, *offset*])

(Unix バージョン) ファイルディスクリプタ *fileno* で指定されたファイルから *length* バイトをマップし、mmap オブジェクトを返します。*length* が 0 の場合、マップの最大の長さは *mmap* が呼ばれた時点でのファイルサイズになります。

flags はマップの種類を指定します。MAP_PRIVATE はプライベートな copy-on-write(書込み時コピー) のマップを作成します。従って、mmap オブジェクトの内容への変更はこのプロセス内にのみ有効です。MAP_SHARED はファイルの同じ領域をマップする他のすべてのプロセスと共有されたマップを作成します。デフォルトは MAP_SHARED です。

prot が指定された場合、希望のメモリ保護を与えます。2 つの最も有用な値は、PROT_READ と PROT_WRITE です。これは、読込み可能または書込み可能を指定するものです。*prot* のデフォルトは PROT_READ | PROT_WRITE です。

access はオプションのキーワード・パラメータとして、*flags* と *prot* の代わりに指定してもかまいません。*flags*, *prot* と *access* の両方を指定することは間違っています。このパラメータの使用法についての情報は、先に述べた *access* の記述を参照してください。

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of ALLOCATIONGRANULARITY which is equal to PAGESIZE on Unix systems.

Mac OS X と OpenVMS において、作成された memory mapping の正当性を確実にするために *fileno* で指定されたファイルディスクリプタは内部で自動的に物理的なストレージ (physical backing store) と同期されます。

この例は *mmap* の簡潔な使い方を示すものです:

```
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write(b"Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)
    # read content via standard file methods
    print(mm.readline())  # prints b"Hello Python!\n"
    # read content via slice notation
    print(mm[:5])  # prints b"Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = b" world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print(mm.readline())  # prints b"Hello  world!\n"
    # close the map
    mm.close()
```

mmap は with 文の中でコンテキストマネージャとしても使えます:

```
import mmap

with mmap.mmap(-1, 13) as mm:
    mm.write(b"Hello world!")
```

バージョン 3.2 で追加: コンテキストマネージャのサポート。

次の例では無名マップを作り親プロセスと子プロセスの間でデータのやりとりをしてみせます:

```
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write(b"Hello world!")

pid = os.fork()

if pid == 0:  # In a child process
    mm.seek(0)
    print(mm.readline())

    mm.close()
```

引数 fileno, length, access, offset 付きで 監査イベント mmap.__new__ を送出します。

メモリマップファイルオブジェクトは以下のメソッドをサポートしています:

close()

メモリマップファイルを閉じます。この呼出しの後にオブジェクトの他のメソッドの呼出すことは、*ValueError* 例外の送出を引き起こします。このメソッドは開いたファイルのクローズはしません。

closed

ファイルが閉じている場合 True となります。

バージョン 3.2 で追加.

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

オブジェクト内の [*start*, *end*] の範囲に含まれている部分シーケンス *sub* が見つかった場所の最も小さいインデックスを返します。オプションの引数 *start* と *end* はスライスに使われるときのように解釈されます。失敗したときには -1 を返します。

バージョン 3.5 で変更: 書き込み可能な *bytes-like object* を使用できるようになりました。

flush([*offset*[, *size*]])

Flushes changes made to the in-memory copy of a file back to disk. Without use of this call there is no guarantee that changes are written back before the object is destroyed. If *offset* and *size* are specified, only changes to the given range of bytes will be flushed to disk; otherwise, the whole extent of the mapping is flushed. *offset* must be a multiple of the PAGESIZE or ALLOCATIONGRANULARITY.

None is returned to indicate success. An exception is raised when the call failed.

バージョン 3.8 で変更: Previously, a nonzero value was returned on success; zero was returned on error under Windows. A zero value was returned on success; an exception was raised on error under Unix.

madvise(*option*[, *start*[, *length*]])

Send advice *option* to the kernel about the memory region beginning at *start* and extending *length* bytes. *option* must be one of the *MADV_* constants* available on the system. If *start* and *length* are omitted, the entire mapping is spanned. On some systems (including Linux), *start* must be a multiple of the PAGESIZE.

Availability: Systems with the madvise() system call.

バージョン 3.8 で追加.

move(*dest*, *src*, *count*)

オフセット *src* から始まる *count* バイトをインデックス *dest* の位置へコピーします。もし mmap が ACCESS_READ で作成されていた場合、*TypeError* 例外を発生させます。

read([*n*])

現在のファイル位置からの最大 *n* バイトを含む *bytes* を返します。引数が省略されるか、None もしくは負の値が指定された場合、現在のファイル位置からマップ終端までの全てのバイト列を返します。ファイル位置は返されたバイト列の直後を指すように更新されます。

バージョン 3.3 で変更: 引数が省略可能になり、None も受け付けるようになりました。

read_byte()
現在のファイル位置のバイトを整数値として返し、ファイル位置を 1 進めます。

readline()
現在のファイル位置から次の改行までの、1 行を返します。

resize(*newsize*)
マップと元ファイル (がもしあれば) のサイズを変更します。もし mmap が ACCESS_READ または ACCESS_COPY で作成されたならば、マップサイズの変更は *TypeError* 例外を発生させます。

rfind(*sub*[, *start*[, *end*]])
オブジェクト内の [*start*, *end*] の範囲に含まれている部分シーケンス *sub* が見つかった場所の最も大きいインデックスを返します。オプションの引数 *start* と *end* はスライスに使われるときのように解釈されます。失敗したときには -1 を返します。

バージョン 3.5 で変更: 書き込み可能な *bytes-like object* を使用できるようになりました。

seek(*pos*[, *whence*])
ファイルの現在位置をセットします。*whence* 引数はオプションであり、デフォルトは os.SEEK_SET つまり 0 (絶対位置) です。その他の値として、os.SEEK_CUR つまり 1 (現在位置からの相対位置) と os.SEEK_END つまり 2 (ファイルの終わりからの相対位置) があります。

size()
ファイルの長さを返します。メモリマップ領域のサイズより大きいかもしれません。

tell()
ファイルポインタの現在位置を返します。

write(*bytes*)
メモリ内のファイルポイントの現在位置に *bytes* のバイト列を書き込み、書き込まれたバイト数を返します (もし書き込みが失敗したら *ValueError* が送出されるため、len(bytes) より少なくなりません)。ファイル位置はバイト列が書き込まれた位置に更新されます。もし mmap が:const:*ACCESS_READ* とともに作成されていた場合は、書き込みは *TypeError* 例外を送出するでしょう。

バージョン 3.5 で変更: 書き込み可能な *bytes-like object* を使用できるようになりました。

バージョン 3.6 で変更: 書きこまれたバイト数を返すようになりました。

write_byte(*byte*)
メモリ内のファイル・ポインタの現在位置に整数 *byte* を書き込みます。ファイル位置は 1 だけ進みます。もし mmap が ACCESS_READ で作成されていた場合、書き込み時に *TypeError* 例外を発生させるでしょう。

18.9.1 MADV_* Constants

mmap.MADV_NORMAL
mmap.MADV_RANDOM
mmap.MADV_SEQUENTIAL
mmap.MADV_WILLNEED
mmap.MADV_DONTNEED
mmap.MADV_REMOVE
mmap.MADV_DONTFORK
mmap.MADV_DOFORK
mmap.MADV_HWPOISON
mmap.MADV_MERGEABLE
mmap.MADV_UNMERGEABLE
mmap.MADV_SOFT_OFFLINE
mmap.MADV_HUGEPAGE
mmap.MADV_NOHUGEPAGE
mmap.MADV_DONTDUMP
mmap.MADV_DODUMP
mmap.MADV_FREE
mmap.MADV_NOSYNC
mmap.MADV_AUTOSYNC
mmap.MADV_NOCORE
mmap.MADV_CORE
mmap.MADV_PROTECT

These options can be passed to *mmap.madvise()*. Not every option will be present on every system.

Availability: Systems with the madvise() system call.

バージョン 3.8 で追加.

第

NINETEEN

インターネット上のデータの操作

この章ではインターネット上で一般的に利用されているデータ形式の操作をサポートするモジュール群について記述します。

19.1 email --- 電子メールと MIME 処理のためのパッケージ

ソースコード: Lib/email/__init__.py

---

email パッケージは、電子メールメッセージを管理するライブラリです。特に、SMTP（**:rfc:‘2821)、NNTP、またはその他のサーバーに電子メールメッセージを送信するようには設計されていません。これらは、:mod:‘smtplib‘** や、*nntplib* などのモジュールの関数群です。*email* パッケージは、可能な限り RFC に準拠するよう試みています。**:rfc:‘5322‘や :rfc:‘6532‘** のほか、**RFC 2045**、**RFC 2046**、**RFC 2047**、**RFC 2183**、**RFC 2231** などの MIME 関連の RFC に対応しています。

The overall structure of the email package can be divided into three major components, plus a fourth component that controls the behavior of the other components.

The central component of the package is an ”object model” that represents email messages. An application interacts with the package primarily through the object model interface defined in the *message* sub-module. The application can use this API to ask questions about an existing email, to construct a new email, or to add or remove email subcomponents that themselves use the same object model interface. That is, following the nature of email messages and their MIME subcomponents, the email object model is a tree structure of objects that all provide the *EmailMessage* API.

The other two major components of the package are the *parser* and the *generator*. The parser takes the serialized version of an email message (a stream of bytes) and converts it into a tree of *EmailMessage* objects. The generator takes an *EmailMessage* and turns it back into a serialized byte stream. (The parser and generator also handle streams of text characters, but this usage is discouraged as it is too easy to end up with messages that are not valid in one way or another.)

The control component is the *policy* module. Every *EmailMessage*, every *generator*, and every *parser* has an associated *policy* object that controls its behavior. Usually an application only needs to specify the policy when an *EmailMessage* is created, either by directly instantiating an *EmailMessage* to create a new email, or by parsing an input stream using a *parser*. But the policy can be changed when the

message is serialized using a *generator*. This allows, for example, a generic email message to be parsed from disk, but to serialize it using standard SMTP settings when sending it to an email server.

The email package does its best to hide the details of the various governing RFCs from the application. Conceptually the application should be able to treat the email message as a structured tree of unicode text and binary attachments, without having to worry about how these are represented when serialized. In practice, however, it is often necessary to be aware of at least some of the rules governing MIME messages and their structure, specifically the names and nature of the MIME "content types" and how they identify multipart documents. For the most part this knowledge should only be required for more complex applications, and even then it should only be the high level structure in question, and not the details of how those structures are represented. Since MIME content types are used widely in modern internet software (not just email), this will be a familiar concept to many programmers.

The following sections describe the functionality of the *email* package. We start with the *message* object model, which is the primary interface an application will use, and follow that with the *parser* and *generator* components. Then we cover the *policy* controls, which completes the treatment of the main components of the library.

The next three sections cover the exceptions the package may raise and the defects (non-compliance with the RFCs) that the *parser* may detect. Then we cover the *headerregistry* and the *contentmanager* sub-components, which provide tools for doing more detailed manipulation of headers and payloads, respectively. Both of these components contain features relevant to consuming and producing non-trivial messages, but also document their extensibility APIs, which will be of interest to advanced applications.

Following those is a set of examples of using the fundamental parts of the APIs covered in the preceding sections.

The foregoing represent the modern (unicode friendly) API of the email package. The remaining sections, starting with the *Message* class, cover the legacy *compat32* API that deals much more directly with the details of how email messages are represented. The *compat32* API does *not* hide the details of the RFCs from the application, but for applications that need to operate at that level, they can be useful tools. This documentation is also relevant for applications that are still using the *compat32* API for backward compatibility reasons.

バージョン 3.6 で変更: Docs reorganized and rewritten to promote the new *EmailMessage*/*EmailPolicy* API.

email パッケージ文書の内容

19.1.1 email.message: 電子メールメッセージの表現

ソースコード: Lib/email/message.py

バージョン 3.6 で追加:[*1]

The central class in the *email* package is the *EmailMessage* class, imported from the *email.message* module. It is the base class for the *email* object model. *EmailMessage* provides the core functionality for setting and querying header fields, for accessing message bodies, and for creating or modifying structured messages.

An email message consists of *headers* and a *payload* (which is also referred to as the *content*). Headers are **RFC 5322** or **RFC 6532** style field names and values, where the field name and value are separated by a colon. The colon is not part of either the field name or the field value. The payload may be a simple text message, or a binary object, or a structured sequence of sub-messages each with their own set of headers and their own payload. The latter type of payload is indicated by the message having a MIME type such as *multipart/** or *message/rfc822*.

The conceptual model provided by an *EmailMessage* object is that of an ordered dictionary of headers coupled with a *payload* that represents the **RFC 5322** body of the message, which might be a list of sub-`EmailMessage` objects. In addition to the normal dictionary methods for accessing the header names and values, there are methods for accessing specialized information from the headers (for example the MIME content type), for operating on the payload, for generating a serialized version of the message, and for recursively walking over the object tree.

The *EmailMessage* dictionary-like interface is indexed by the header names, which must be ASCII values. The values of the dictionary are strings with some extra methods. Headers are stored and returned in case-preserving form, but field names are matched case-insensitively. Unlike a real dict, there is an ordering to the keys, and there can be duplicate keys. Additional methods are provided for working with headers that have duplicate keys.

The *payload* is either a string or bytes object, in the case of simple message objects, or a list of *EmailMessage* objects, for MIME container documents such as *multipart/** and *message/rfc822* message objects.

`class email.message.EmailMessage(`*policy=default*`)`

If *policy* is specified use the rules it specifies to update and serialize the representation of the message. If *policy* is not set, use the *default* policy, which follows the rules of the email RFCs except for line endings (instead of the RFC mandated `\r\n`, it uses the Python standard `\n` line endings). For more information see the *policy* documentation.

`as_string(`*unixfrom=False*, *maxheaderlen=None*, *policy=None*`)`

Return the entire message flattened as a string. When optional *unixfrom* is true, the envelope

[*1] Originally added in 3.4 as a *provisional module*. Docs for legacy message class moved to *email.message.Message: compat32 API を使用した電子メールメッセージの表現*.

header is included in the returned string. *unixfrom* defaults to `False`. For backward compatibility with the base *Message* class *maxheaderlen* is accepted, but defaults to `None`, which means that by default the line length is controlled by the `max_line_length` of the policy. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the *Generator*.

もし、文字列への変換を完全に行うためにデフォルト値を埋める必要がある場合、メッセージのフラット化は *EmailMessage* の変更を引き起こす可能性があります (例えば、MIME の境界が生成される、変更される等)。

Note that this method is provided as a convenience and may not be the most useful way to serialize messages in your application, especially if you are dealing with multiple messages. See *email.generator.Generator* for a more flexible API for serializing messages. Note also that this method is restricted to producing messages serialized as "7 bit clean" when *utf8* is `False`, which is the default.

バージョン 3.6 で変更: the default behavior when *maxheaderlen* is not specified was changed from defaulting to 0 to defaulting to the value of *max_line_length* from the policy.

`__str__()`

Equivalent to `as_string(policy=self.policy.clone(utf8=True))`. Allows `str(msg)` to produce a string containing the serialized message in a readable format.

バージョン 3.4 で変更: the method was changed to use `utf8=True`, thus producing an **RFC 6531**-like message representation, instead of being a direct alias for *as_string()*.

`as_bytes(`*unixfrom=False*, *policy=None*`)`

Return the entire message flattened as a bytes object. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the *BytesGenerator*.

もし、文字列への変換を完全に行うためにデフォルト値を埋める必要がある場合、メッセージのフラット化は *EmailMessage* の変更を引き起こす可能性があります (例えば、MIME の境界が生成される、変更される等)。

Note that this method is provided as a convenience and may not be the most useful way to serialize messages in your application, especially if you are dealing with multiple messages. See *email.generator.BytesGenerator* for a more flexible API for serializing messages.

`__bytes__()`

Equivalent to *as_bytes()*. Allows `bytes(msg)` to produce a bytes object containing the serialized message.

`is_multipart()`

Return `True` if the message's payload is a list of sub-*EmailMessage* objects, otherwise return

False. When *is_multipart()* returns False, the payload should be a string object (which might be a CTE encoded binary payload). Note that *is_multipart()* returning True does not necessarily mean that "msg.get_content_maintype() == 'multipart'" will return the True. For example, is_multipart will return True when the *EmailMessage* is of type message/rfc822.

set_unixfrom(*unixfrom*)

Set the message's envelope header to *unixfrom*, which should be a string. (See *mboxMessage* for a brief description of this header.)

get_unixfrom()

メッセージのエンベロープヘッダを返します。エンベロープヘッダが設定されていない場合は None が返されます。

The following methods implement the mapping-like interface for accessing the message's headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by *keys()*, but in an *EmailMessage* object, headers are always returned in the order they appeared in the original message, or in which they were added to the message later. Any header deleted and then re-added is always appended to the end of the header list.

These semantic differences are intentional and are biased toward convenience in the most common use cases.

注意: どんな場合も、メッセージ中のエンベロープヘッダはこのマップ形式のインタフェイスには含まれません。

__len__()

複製されたものもふくめてヘッダ数の合計を返します。

__contains__(*name*)

Return True if the message object has a field named *name*. Matching is done without regard to case and *name* does not include the trailing colon. Used for the in operator. For example:

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)

Return the value of the named header field. *name* does not include the colon field separator. If the header is missing, None is returned; a *KeyError* is never raised.

Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the *get_all()* method to get the values of all the extant headers named *name*.

Using the standard (non-compat32) policies, the returned value is an instance of a subclass of *email.headerregistry.BaseHeader*.

__setitem__(*name*, *val*)
メッセージヘッダに *name* という名前の *val* という値をもつフィールドをあらたに追加します。このフィールドは現在メッセージに存在するヘッダーのいちばん後に追加されます。

注意: このメソッドでは、すでに同一の名前で存在するフィールドは上書き **されません**。もしメッセージが名前 *name* をもつフィールドをひとつしか持たないようにしたければ、最初にそれを除去してください。たとえば:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

If the `policy` defines certain headers to be unique (as the standard policies do), this method may raise a *ValueError* when an attempt is made to assign a value to such a header when one already exists. This behavior is intentional for consistency's sake, but do not depend on it as we may choose to make such assignments do an automatic deletion of the existing header in the future.

__delitem__(*name*)
メッセージのヘッダから、*name* という名前をもつフィールドをすべて除去します。たとえこの名前をもつヘッダが存在していなくても例外は発生しません。

keys()
メッセージ中にあるすべてのヘッダのフィールド名のリストを返します。

values()
メッセージ中にあるすべてのフィールドの値のリストを返します。

items()
メッセージ中にあるすべてのヘッダのフィールド名とその値を 2-タプルのリストとして返します。

get(*name*, *failobj=None*)
指定された名前をもつフィールドの値を返します。これは指定された名前がないときにオプション引数の *failobj* (*failobj* のデフォルトは `None`) を返すことをのぞけば、*__getitem__()* と同じです。

Here are some additional useful header related methods:

get_all(*name*, *failobj=None*)
name の名前をもつフィールドのすべての値からなるリストを返します。該当する名前のヘッダがメッセージ中に含まれていない場合は *failobj* (デフォルトでは `None`) が返されます。

add_header(*_name*, *_value*, ***_params*)
拡張ヘッダ設定。このメソッドは *__setitem__()* と似ていますが、追加のヘッダ・パラメータをキーワード引数で指定できるところが違っています。*_name* に追加するヘッダフィールドを、*_value* にそのヘッダの **最初の** 値を渡します。

For each item in the keyword argument dictionary *_params*, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers).

Normally, the parameter will be added as `key="value"` unless the value is `None`, in which case only the key will be added.

If the value contains non-ASCII characters, the charset and language may be explicitly controlled by specifying the value as a three tuple in the format `(CHARSET, LANGUAGE, VALUE)`, where `CHARSET` is a string naming the charset to be used to encode the value, `LANGUAGE` can usually be set to `None` or the empty string (see **RFC 2231** for other possibilities), and `VALUE` is the string value containing non-ASCII code points. If a three tuple is not passed and the value contains non-ASCII characters, it is automatically encoded in **RFC 2231** format using a `CHARSET` of `utf-8` and a `LANGUAGE` of `None`.

以下に例を示します。:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

こうするとヘッダには以下のように追加されます

```
Content-Disposition: attachment; filename="bud.gif"
```

An example of the extended interface with non-ASCII characters:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

replace_header(*_name*, *_value*)

Replace a header. Replace the first header found in the message that matches *_name*, retaining header order and field name case of the original header. If no matching header is found, raise a *KeyError*.

get_content_type()

Return the message's content type, coerced to lower case of the form *maintype/subtype*. If there is no *Content-Type* header in the message return the value returned by *get_default_type()*. If the *Content-Type* header is invalid, return `text/plain`.

(According to **RFC 2045**, messages always have a default type, *get_content_type()* will always return a value. **RFC 2045** defines a message's default type to be *text/plain* unless it appears inside a *multipart/digest* container, in which case it would be *message/rfc822*. If the *Content-Type* header has an invalid type specification, **RFC 2045** mandates that the default type be *text/plain*.)

get_content_maintype()

そのメッセージの主 content-type を返します。これは *get_content_type()* によって返される文字列の *maintype* 部分です。

get_content_subtype()

そのメッセージの副 content-type (sub content-type、subtype) を返します。これは *get_content_type()* によって返される文字列の *subtype* 部分です。

get_default_type()
デフォルトの content-type を返します。ほどんどのメッセージではデフォルトの content-type は *text/plain* ですが、メッセージが *multipart/digest* コンテナに含まれているときだけ例外的に *message/rfc822* になります。

set_default_type(*ctype*)
Set the default content type. *ctype* should either be *text/plain* or *message/rfc822*, although this is not enforced. The default content type is not stored in the *Content-Type* header, so it only affects the return value of the get_content_type methods when no *Content-Type* header is present in the message.

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)
Set a parameter in the *Content-Type* header. If the parameter already exists in the header, replace its value with *value*. When *header* is Content-Type (the default) and the header does not yet exist in the message, add it, set its value to *text/plain*, and append the new parameter value. Optional *header* specifies an alternative header to *Content-Type*.

If the value contains non-ASCII characters, the charset and language may be explicitly specified using the optional *charset* and *language* parameters. Optional *language* specifies the **RFC 2231** language, defaulting to the empty string. Both *charset* and *language* should be strings. The default is to use the utf8 *charset* and None for the *language*.

If *replace* is False (the default) the header is moved to the end of the list of headers. If *replace* is True, the header will be updated in place.

Use of the *requote* parameter with *EmailMessage* objects is deprecated.

Note that existing parameter values of headers may be accessed through the params attribute of the header value (for example, msg['Content-Type'].params['charset']).

バージョン 3.4 で変更: replace キーワードが追加されました。

del_param(*param*, *header='content-type'*, *requote=True*)
指定されたパラメータを *Content-Type* ヘッダ中から完全にとりのぞきます。ヘッダはそのパラメータと値がない状態に書き換えられます。オプション変数 *header* が与えられた場合、*Content-Type* のかわりにそのヘッダが使用されます。

Use of the *requote* parameter with *EmailMessage* objects is deprecated.

get_filename(*failobj=None*)
そのメッセージ中の *Content-Disposition* ヘッダにある、filename パラメータの値を返します。目的のヘッダに filename パラメータがない場合には *Content-Type* ヘッダにある name パラメータを探します。それも無い場合またはヘッダが無い場合には *failobj* が返されます。返される文字列はつねに *email.utils.unquote()* によって unquote されます。

get_boundary(*failobj=None*)
そのメッセージ中の *Content-Type* ヘッダにある、boundary パラメータの値を返します。目的のヘッダが欠けていたり、boundary パラメータがない場合には *failobj* が返されます。返される

文字列はつねに *email.utils.unquote()* によって unquote されます。

set_boundary(*boundary*)
メッセージ中の *Content-Type* ヘッダにある、boundary パラメータに値を設定します。*set_boundary()* は必要に応じて *boundary* を quote します。そのメッセージが *Content-Type* ヘッダを含んでいない場合、*HeaderParseError* が発生します。

Note that using this method is subtly different from deleting the old *Content-Type* header and adding a new one with the new boundary via *add_header()*, because *set_boundary()* preserves the order of the *Content-Type* header in the list of headers.

get_content_charset(*failobj=None*)
そのメッセージ中の *Content-Type* ヘッダにある、charset パラメータの値を返します。値はすべて小文字に変換されます。メッセージ中に *Content-Type* がなかったり、このヘッダ中に charaset パラメータがない場合には *failobj* が返されます。

get_charsets(*failobj=None*)
メッセージ中に含まれる文字セットの名前をすべてリストにして返します。そのメッセージが *multipart* である場合、返されるリストの各要素がそれぞれの subpart のペイロードに対応します。それ以外の場合、これは長さ 1 のリストを返します。

リスト中の各要素は文字列であり、これは対応する subpart 中のそれぞれの *Content-Type* ヘッダにある charset の値です。その subpart が *Content-Type* をもってないか、charset がないか、あるいは MIME maintype が *text* でないいずれかの場合には、リストの要素として *failobj* が返されます。

is_attachment()
Content-Disposition ヘッダが存在し、その (大文字小文字を区別しない) 値が attachment の場合、True を返します。それ以外の場合は False を返します。

バージョン 3.4.2 で変更: *is_multipart()* との一貫性のために、is_attachment が属性からメソッドになりました。

get_content_disposition()
Return the lowercased value (without parameters) of the message's *Content-Disposition* header if it has one, or None. The possible values for this method are *inline*, *attachment* or None if the message follows **RFC 2183**.

バージョン 3.5 で追加.

The following methods relate to interrogating and manipulating the content (payload) of the message.

walk()
walk() メソッドは多目的のジェネレータで、これはあるメッセージオブジェクトツリー中のすべての part および subpart をわたり歩くのに使えます。順序は深さ優先です。おそらく典型的な用法は、*walk()* を for ループ中でのイテレータとして使うことでしょう。ループを一回まわるごとに、次の subpart が返されるのです。

以下の例は、multipart メッセージのすべての part において、その MIME タイプを表示していくものです。:

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

`walk` iterates over the subparts of any part where *is_multipart()* returns `True`, even though `msg.get_content_maintype() == 'multipart'` may return `False`. We can see this in our example by making use of the `_structure` debug helper function:

```
>>> from email.iterators import _structure
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

Here the `message` parts are not `multiparts`, but they do contain subparts. `is_multipart()` returns `True` and `walk` descends into the subparts.

`get_body`(*preferencelist=('related', 'html', 'plain')*)
メッセージの ” 本体” の最有力候補となる MIME 部を返します。

preferencelist must be a sequence of strings from the set `related`, `html`, and `plain`, and indicates the order of preference for the content type of the part returned.

Start looking for candidate matches with the object on which the `get_body` method is called.

If `related` is not included in *preferencelist*, consider the root part (or subpart of the root part) of any related encountered as a candidate if the (sub-)part matches a preference.

When encountering a `multipart/related`, check the `start` parameter and if a part with a

matching *Content-ID* is found, consider only it when looking for candidate matches. Otherwise consider only the first (default root) part of the `multipart/related`.

If a part has a *Content-Disposition* header, only consider the part a candidate match if the value of the header is `inline`.

If none of the candidates matches any of the preferences in *preferencelist*, return `None`.

Notes: (1) For most applications the only *preferencelist* combinations that really make sense are `('plain',)`, `('html', 'plain')`, and the default `('related', 'html', 'plain')`. (2) Because matching starts with the object on which `get_body` is called, calling `get_body` on a `multipart/related` will return the object itself unless *preferencelist* has a non-default value. (3) Messages (or message parts) that do not specify a *Content-Type* or whose *Content-Type* header is invalid will be treated as if they are of type `text/plain`, which may occasionally cause `get_body` to return unexpected results.

`iter_attachments()`

Return an iterator over all of the immediate sub-parts of the message that are not candidate "body" parts. That is, skip the first occurrence of each of `text/plain`, `text/html`, `multipart/related`, or `multipart/alternative` (unless they are explicitly marked as attachments via *Content-Disposition: attachment*), and return all remaining parts. When applied directly to a `multipart/related`, return an iterator over the all the related parts except the root part (ie: the part pointed to by the `start` parameter, or the first part if there is no `start` parameter or the `start` parameter doesn't match the *Content-ID* of any of the parts). When applied directly to a `multipart/alternative` or a non-`multipart`, return an empty iterator.

`iter_parts()`

Return an iterator over all of the immediate sub-parts of the message, which will be empty for a non-`multipart`. (See also *walk()*.)

`get_content(`**args*, *content_manager=None*, ***kw*`)`

Call the *get_content()* method of the *content_manager*, passing self as the message object, and passing along any other arguments or keywords as additional arguments. If *content_manager* is not specified, use the `content_manager` specified by the current *policy*.

`set_content(`**args*, *content_manager=None*, ***kw*`)`

Call the *set_content()* method of the *content_manager*, passing self as the message object, and passing along any other arguments or keywords as additional arguments. If *content_manager* is not specified, use the `content_manager` specified by the current *policy*.

`make_related(`*boundary=None*`)`

Convert a non-`multipart` message into a `multipart/related` message, moving any existing *Content-* headers and payload into a (new) first part of the `multipart`. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

`make_alternative(`*boundary=None*`)`

Convert a non-multipart or a multipart/related into a multipart/alternative, moving any existing *Content-* headers and payload into a (new) first part of the multipart. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

make_mixed(*boundary=None*)

Convert a non-multipart, a multipart/related, or a multipart-alternative into a multipart/mixed, moving any existing *Content-* headers and payload into a (new) first part of the multipart. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

add_related(**args*, *content_manager=None*, ***kw*)

If the message is a multipart/related, create a new message object, pass all of the arguments to its *set_content()* method, and *attach()* it to the multipart. If the message is a non-multipart, call *make_related()* and then proceed as above. If the message is any other type of multipart, raise a *TypeError*. If *content_manager* is not specified, use the content_manager specified by the current *policy*. If the added part has no *Content-Disposition* header, add one with the value inline.

add_alternative(**args*, *content_manager=None*, ***kw*)

If the message is a multipart/alternative, create a new message object, pass all of the arguments to its *set_content()* method, and *attach()* it to the multipart. If the message is a non-multipart or multipart/related, call *make_alternative()* and then proceed as above. If the message is any other type of multipart, raise a *TypeError*. If *content_manager* is not specified, use the content_manager specified by the current *policy*.

add_attachment(**args*, *content_manager=None*, ***kw*)

If the message is a multipart/mixed, create a new message object, pass all of the arguments to its *set_content()* method, and *attach()* it to the multipart. If the message is a non-multipart, multipart/related, or multipart/alternative, call *make_mixed()* and then proceed as above. If *content_manager* is not specified, use the content_manager specified by the current *policy*. If the added part has no *Content-Disposition* header, add one with the value attachment. This method can be used both for explicit attachments (*Content-Disposition: attachment*) and inline attachments (*Content-Disposition: inline*), by passing appropriate options to the content_manager.

clear()

ペイロードとヘッダの全てを削除します。

clear_content()

Remove the payload and all of the Content- headers, leaving all other headers intact and in their original order.

EmailMessage オブジェクトは次のようなインスタンス属性を持ちます:

preamble

MIME ドキュメントの形式では、ヘッダ直後にくる空行と最初の multipart 境界をあらわす文字列のあいだにいくらかのテキスト (訳注: preamble, 序文) を埋めこむことを許しています。このテキストは標準的な MIME の範疇からはみ出しているので、MIME 形式を認識するメールソフトからこれらは通常まったく見えません。しかしメッセージのテキストを生で見る場合、あるいはメッセージを MIME 対応していないメールソフトで見る場合、このテキストは目に見えることになります。

preamble 属性は MIME ドキュメントに加えるこの最初の MIME 範囲外テキストを含んでいます。*Parser* があるテキストをヘッダ以降に発見したが、それはまだ最初の MIME 境界文字列が現れる前だった場合、パーザはそのテキストをメッセージの *preamble* 属性に格納します。*Generator* がある MIME メッセージからプレーンテキスト形式を生成するときメッセージが *preamble* 属性を持つことが発見されれば、これはそのテキストをヘッダと最初の MIME 境界の間に挿入します。詳細は *email.parser* および *email.generator* を参照してください。

注意: そのメッセージに preamble がない場合、*preamble* 属性には `None` が格納されます。

epilogue

The *epilogue* attribute acts the same way as the *preamble* attribute, except that it contains text that appears between the last boundary and the end of the message. As with the *preamble*, if there is no epilog text this attribute will be `None`.

defects

defects 属性はメッセージを解析する途中で検出されたすべての問題点 (defect、障害) のリストを保持しています。解析中に発見されうる障害についてのより詳細な説明は *email.errors* を参照してください。

`class email.message.MIMEPart`(*policy=default*)

This class represents a subpart of a MIME message. It is identical to *EmailMessage*, except that no *MIME-Version* headers are added when *set_content()* is called, since sub-parts do not need their own *MIME-Version* headers.

脚注

19.1.2 email.parser: 電子メールメッセージのパース

ソースコード: Lib/email/parser.py

Message object structures can be created in one of two ways: they can be created from whole cloth by creating an *EmailMessage* object, adding headers using the dictionary interface, and adding payload(s) using *set_content()* and related methods, or they can be created by parsing a serialized representation of the email message.

The *email* package provides a standard parser that understands most email document structures, including MIME documents. You can pass the parser a bytes, string or file object, and the parser will return to you the root *EmailMessage* instance of the object structure. For simple, non-MIME messages

the payload of this root object will likely be a string containing the text of the message. For MIME messages, the root object will return `True` from its *is_multipart()* method, and the subparts can be accessed via the payload manipulation methods, such as *get_body()*, *iter_parts()*, and *walk()*.

There are actually two parser interfaces available for use, the *Parser* API and the incremental *FeedParser* API. The *Parser* API is most useful if you have the entire text of the message in memory, or if the entire message lives in a file on the file system. *FeedParser* is more appropriate when you are reading the message from a stream which might block waiting for more input (such as reading an email message from a socket). The *FeedParser* can consume and parse the message incrementally, and only returns the root object when you close the parser.

Note that the parser can be extended in limited ways, and of course you can implement your own parser completely from scratch. All of the logic that connects the *email* package's bundled parser and the *EmailMessage* class is embodied in the `policy` class, so a custom parser can create message object trees any way it finds necessary by implementing custom versions of the appropriate `policy` methods.

FeedParser API

The *BytesFeedParser*, imported from the `email.feedparser` module, provides an API that is conducive to incremental parsing of email messages, such as would be necessary when reading the text of an email message from a source that can block (such as a socket). The *BytesFeedParser* can of course be used to parse an email message fully contained in a *bytes-like object*, string, or file, but the *BytesParser* API may be more convenient for such use cases. The semantics and results of the two parser APIs are identical.

BytesFeedParser API は簡単です。まずインスタンスをつくり、それに bytes を (それ以上 bytes が必要なくなるまで) 流しこみます。その後パーザを close すると根っこ (root) のメッセージオブジェクトが返されます。標準に従ったメッセージを解析する場合、*BytesFeedParser* は非常に正確であり、標準に従っていないメッセージでもちゃんと動きます。そのさい、これはメッセージがどのように壊れていると認識されたかについての情報を残します。これはメッセージオブジェクトの *defects* 属性にリストとして現れ、メッセージ中に発見された問題が記録されます。パーザが検出できる障害 (defect) については *email.errors* モジュールを参照してください。

以下は *BytesFeedParser* の API です:

class email.parser.**BytesFeedParser**(*_factory=None*, ***, *policy=policy.compat32*)

Create a *BytesFeedParser* instance. Optional *_factory* is a no-argument callable; if not specified use the *message_factory* from the *policy*. Call *_factory* whenever a new message object is needed.

If *policy* is specified use the rules it specifies to update the representation of the message. If *policy* is not set, use the *compat32* policy, which maintains backward compatibility with the Python 3.2 version of the email package and provides *Message* as the default factory. All other policies provide *EmailMessage* as the default *_factory*. For more information on what else *policy* controls, see the *policy* documentation.

Note: **The policy keyword should always be specified**; The default will change to *email.policy.default* in a future version of Python.

バージョン 3.2 で追加.

バージョン 3.3 で変更: キーワード引数 *policy* が追加されました。

バージョン 3.6 で変更: *_factory* defaults to the policy `message_factory`.

`feed`(*data*)
パーサーにデータを供給します。*data* は 1 行または複数行からなる *bytes-like object* を渡します。渡される行は完結していなくてもよく、その場合パーサーは部分的な行を適切につなぎ合わせます。各行は 3 種類の標準的な行末文字 (復帰 CR、改行 LF、または CR+LF) どれかの組み合わせでよく、これらが混在してもかまいません。

`close`()
Complete the parsing of all previously fed data and return the root message object. It is undefined what happens if *feed()* is called after this method has been called.

`class email.parser.FeedParser`(*_factory=None*, *, *policy=policy.compat32*)
Works like *BytesFeedParser* except that the input to the *feed()* method must be a string. This is of limited utility, since the only way for such a message to be valid is for it to contain only ASCII text or, if `utf8` is `True`, no binary attachments.

バージョン 3.3 で変更: キーワード引数 *policy* が追加されました。

Parser API

The *BytesParser* class, imported from the *email.parser* module, provides an API that can be used to parse a message when the complete contents of the message are available in a *bytes-like object* or file. The *email.parser* module also provides *Parser* for parsing strings, and header-only parsers, *BytesHeaderParser* and *HeaderParser*, which can be used if you're only interested in the headers of the message. *BytesHeaderParser* and *HeaderParser* can be much faster in these situations, since they do not attempt to parse the message body, instead setting the payload to the raw body.

`class email.parser.BytesParser`(*_class=None*, *, *policy=policy.compat32*)
Create a *BytesParser* instance. The *_class* and *policy* arguments have the same meaning and semantics as the *_factory* and *policy* arguments of *BytesFeedParser*.

Note: **The policy keyword should always be specified**; The default will change to *email.policy.default* in a future version of Python.

バージョン 3.3 で変更: 2.4 で非推奨になった *strict* 引数の削除。キーワード引数 *policy* の追加。

バージョン 3.6 で変更: *_class* defaults to the policy `message_factory`.

`parse`(*fp*, *headersonly=False*)
Read all the data from the binary file-like object *fp*, parse the resulting bytes, and return the message object. *fp* must support both the *readline()* and the `read()` methods.

The bytes contained in *fp* must be formatted as a block of **RFC 5322** (or, if `utf8` is `True`, **RFC 6532**) style headers and header continuation lines, optionally preceded by an envelope

header. The header block is terminated either by the end of the data or by a blank line. Following the header block is the body of the message (which may contain MIME-encoded subparts, including subparts with a *Content-Transfer-Encoding* of 8bit).

オプション引数 *headersonly* はヘッダを読み終えた後にパースを止めるかを指定するフラグです。デフォルトは False で、ファイルの内容全体をパースします。

parsebytes(*bytes*, *headersonly=False*)

Similar to the *parse()* method, except it takes a *bytes-like object* instead of a file-like object. Calling this method on a *bytes-like object* is equivalent to wrapping *bytes* in a *BytesIO* instance first and calling *parse()*.

オプション引数 *headersonly* は *parse()* メソッドと同じです。

バージョン 3.2 で追加.

class email.parser.BytesHeaderParser(*_class=None*, ***, *policy=policy.compat32*)

Exactly like *BytesParser*, except that *headersonly* defaults to True.

バージョン 3.3 で追加.

class email.parser.Parser(*_class=None*, ***, *policy=policy.compat32*)

This class is parallel to *BytesParser*, but handles string input.

バージョン 3.3 で変更: *strict* 引数の削除。キーワード引数 *policy* の追加。

バージョン 3.6 で変更: *_class* defaults to the policy message_factory.

parse(*fp*, *headersonly=False*)

ファイルなどテキストモードのストリーム形式 (file-like) のオブジェクト *fp* からすべてのデータを読み込み、得られたテキストを解析して基底 (root) メッセージオブジェクト構造を返します。*fp* はストリーム形式のオブジェクトで *readline()* および *read()* 両方のメソッドをサポートしている必要があります。

Other than the text mode requirement, this method operates like *BytesParser.parse()*.

parsestr(*text*, *headersonly=False*)

parse() メソッドに似ていますが、ファイルなどのストリーム形式のかわりに文字列を引数としてとるところが違います。文字列に対してこのメソッドを呼ぶことは、*text* を *StringIO* インスタンスとして作成して *parse()* を適用するのと同じです。

オプション引数 *headersonly* は *parse()* メソッドと同じです。

class email.parser.HeaderParser(*_class=None*, ***, *policy=policy.compat32*)

Exactly like *Parser*, except that *headersonly* defaults to True.

ファイルや文字列からメッセージオブジェクト構造を作成するのはかなりよくおこなわれる作業なので、便宜上次のような 4 つの関数が提供されています。これらは *email* パッケージのトップレベルの名前空間で使用できます。

email.message_from_bytes(*s*, *_class=None*, ***, *policy=policy.compat32*)

bytes-like **オブジェクト** からメッセージオブジェクト構造を作成して返します。これは `BytesParser().parsebytes(s)` と同じです。オプション引数 *_class* および *policy* は *BytesParser* クラスのコンストラクタと同様に解釈されます。

バージョン 3.2 で追加.

バージョン 3.3 で変更: *strict* 引数の削除。キーワード引数 *policy* の追加。

`email.message_from_binary_file`(*fp*, *_class=None*, ***, *policy=policy.compat32*)
オープンされたバイナリ *file object* からメッセージオブジェクト構造を作成して返します。これは `BytesParser().parse(fp)` と同じです。*_class* および *policy* は *BytesParser* クラスのコンストラクタと同様に解釈されます。

バージョン 3.2 で追加.

バージョン 3.3 で変更: *strict* 引数の削除。キーワード引数 *policy* の追加。

`email.message_from_string`(*s*, *_class=None*, ***, *policy=policy.compat32*)
文字列からメッセージオブジェクト構造を作成して返します。これは `Parser().parsestr(s)` と同じです。*_class* および *policy* は *Parser* クラスのコンストラクタと同様に解釈されます。

バージョン 3.3 で変更: *strict* 引数の削除。キーワード引数 *policy* の追加。

`email.message_from_file`(*fp*, *_class=None*, ***, *policy=policy.compat32*)
オープンされた *file object* からメッセージオブジェクト構造を作成して返します。これは `Parser().parse(fp)` と同じです。*_class* および *policy* は *Parser* クラスのコンストラクタと同様に解釈されます。

バージョン 3.3 で変更: *strict* 引数の削除。キーワード引数 *policy* の追加。

バージョン 3.6 で変更: *_class* defaults to the policy `message_factory`.

対話的な Python プロンプトで *message_from_bytes()* を使用するとすれば、このようになります:

```
>>> import email
>>> msg = email.message_from_bytes(myBytes)
```

追記事項

以下はテキスト解析の際に適用されるいくつかの規約です:

- Most non-*multipart* type messages are parsed as a single message object with a string payload. These objects will return `False` for *is_multipart()*, and *iter_parts()* will yield an empty list.
- All *multipart* type messages will be parsed as a container message object with a list of sub-message objects for their payload. The outer container message will return `True` for *is_multipart()*, and *iter_parts()* will yield a list of subparts.
- Most messages with a content type of *message/** (such as *message/delivery-status* and *message/rfc822*) will also be parsed as container object containing a list payload of length 1.

Their *is_multipart()* method will return `True`. The single element yielded by *iter_parts()* will be a sub-message object.

- いくつかの標準的でないメッセージは、*multipart* の使い方に統一がとれていない場合があります。このようなメッセージは *Content-Type* ヘッダに *multipart* を指定しているものの、その *is_multipart()* メソッドは `False` を返すことがあります。もしこのようなメッセージが *FeedParser* によって解析されると、その *defects* 属性のリスト中には `MultipartInvariantViolationDefect` クラスのインスタンスが現れます。詳しい情報については *email.errors* を参照してください。

19.1.3 email.generator: MIME 文書の生成

ソースコード: Lib/email/generator.py

One of the most common tasks is to generate the flat (serialized) version of the email message represented by a message object structure. You will need to do this if you want to send your message via *smtplib.SMTP.sendmail()* or the *nntplib* module, or print the message on the console. Taking a message object structure and producing a serialized representation is the job of the generator classes.

As with the *email.parser* module, you aren't limited to the functionality of the bundled generator; you could write one from scratch yourself. However the bundled generator knows how to generate most email in a standards-compliant way, should handle MIME and non-MIME email messages just fine, and is designed so that the bytes-oriented parsing and generation operations are inverses, assuming the same non-transforming *policy* is used for both. That is, parsing the serialized byte stream via the *BytesParser* class and then regenerating the serialized byte stream using *BytesGenerator* should produce output identical to the input[*1]. (On the other hand, using the generator on an *EmailMessage* constructed by program may result in changes to the *EmailMessage* object as defaults are filled in.)

The *Generator* class can be used to flatten a message into a text (as opposed to binary) serialized representation, but since Unicode cannot represent binary data directly, the message is of necessity transformed into something that contains only ASCII characters, using the standard email RFC Content Transfer Encoding techniques for encoding email messages for transport over channels that are not "8 bit clean".

To accommodate reproducible processing of SMIME-signed messages *Generator* disables header folding for message parts of type `multipart/signed` and all subparts.

class email.generator.BytesGenerator(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, ***, *policy=None*)

Return a *BytesGenerator* object that will write any message provided to the *flatten()* method, or any surrogateescape encoded text provided to the *write()* method, to the *file-like object* *outfp*. *outfp* must support a `write` method that accepts binary data.

[*1] This statement assumes that you use the appropriate setting for `unixfrom`, and that there are no `policy` settings calling for automatic adjustments (for example, `refold_source` must be `none`, which is *not* the default). It is also not 100% true, since if the message does not conform to the RFC standards occasionally information about the exact original text is lost during parsing error recovery. It is a goal to fix these latter edge cases when possible.

If optional *mangle_from_* is `True`, put a `>` character in front of any line in the body that starts with the exact string `"From "`, that is `From` followed by a space at the beginning of a line. *mangle_from_* defaults to the value of the *mangle_from_* setting of the *policy* (which is `True` for the *compat32* policy and `False` for all others). *mangle_from_* is intended for use when messages are stored in unix mbox format (see *mailbox* and WHY THE CONTENT-LENGTH FORMAT IS BAD).

If *maxheaderlen* is not `None`, refold any header lines that are longer than *maxheaderlen*, or if `0`, do not rewrap any headers. If *manheaderlen* is `None` (the default), wrap headers and other message lines according to the *policy* settings.

If *policy* is specified, use that policy to control message generation. If *policy* is `None` (the default), use the policy associated with the *Message* or *EmailMessage* object passed to `flatten` to control the message generation. See *email.policy* for details on what *policy* controls.

バージョン 3.2 で追加.

バージョン 3.3 で変更: キーワード引数 *policy* が追加されました。

バージョン 3.6 で変更: The default behavior of the *mangle_from_* and *maxheaderlen* parameters is to follow the policy.

flatten(*msg*, *unixfrom=False*, *linesep=None*)

msg を基点とするメッセージオブジェクト構造体の文字表現を出力します。出力先のファイルにはこの *BytesGenerator* インスタンスが作成されたときに指定されたものが使われます。

If the *policy* option *cte_type* is `8bit` (the default), copy any headers in the original parsed message that have not been modified to the output with any bytes with the high bit set reproduced as in the original, and preserve the non-ASCII *Content-Transfer-Encoding* of any body parts that have them. If `cte_type` is `7bit`, convert the bytes with the high bit set as needed using an ASCII-compatible *Content-Transfer-Encoding*. That is, transform parts with non-ASCII *Content-Transfer-Encoding* (*Content-Transfer-Encoding: 8bit*) to an ASCII compatible *Content-Transfer-Encoding*, and encode RFC-invalid non-ASCII bytes in headers using the MIME `unknown-8bit` character set, thus rendering them RFC-compliant.

If *unixfrom* is `True`, print the envelope header delimiter used by the Unix mailbox format (see *mailbox*) before the first of the **RFC 5322** headers of the root message object. If the root object has no envelope header, craft a standard one. The default is `False`. Note that for subparts, no envelope header is ever printed.

If *linesep* is not `None`, use it as the separator character between all the lines of the flattened message. If *linesep* is `None` (the default), use the value specified in the *policy*.

clone(*fp*)

Return an independent clone of this *BytesGenerator* instance with the exact same option settings, and *fp* as the new *outfp*.

write(*s*)

Encode *s* using the `ASCII` codec and the `surrogateescape` error handler, and pass it to the *write* method of the *outfp* passed to the *BytesGenerator*'s constructor.

As a convenience, *EmailMessage* provides the methods *as_bytes()* and `bytes(aMessage)` (a.k.a. *__bytes__()*), which simplify the generation of a serialized binary representation of a message object. For more detail, see *email.message*.

Because strings cannot represent binary data, the *Generator* class must convert any binary data in any message it flattens to an ASCII compatible format, by converting them to an ASCII compatible *Content-Transfer_Encoding*. Using the terminology of the email RFCs, you can think of this as *Generator* serializing to an I/O stream that is not "8 bit clean". In other words, most applications will want to be using *BytesGenerator*, and not *Generator*.

`class email.generator.Generator`(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, ***, *policy=None*)

Return a *Generator* object that will write any message provided to the *flatten()* method, or any text provided to the *write()* method, to the *file-like object* *outfp*. *outfp* must support a `write` method that accepts string data.

If optional *mangle_from_* is `True`, put a `>` character in front of any line in the body that starts with the exact string `"From "`, that is `From` followed by a space at the beginning of a line. *mangle_from_* defaults to the value of the *mangle_from_* setting of the *policy* (which is `True` for the *compat32* policy and `False` for all others). *mangle_from_* is intended for use when messages are stored in unix mbox format (see *mailbox* and WHY THE CONTENT-LENGTH FORMAT IS BAD).

If *maxheaderlen* is not `None`, refold any header lines that are longer than *maxheaderlen*, or if `0`, do not rewrap any headers. If *manheaderlen* is `None` (the default), wrap headers and other message lines according to the *policy* settings.

If *policy* is specified, use that policy to control message generation. If *policy* is `None` (the default), use the policy associated with the *Message* or *EmailMessage* object passed to `flatten` to control the message generation. See *email.policy* for details on what *policy* controls.

バージョン 3.3 で変更: キーワード引数 *policy* が追加されました。

バージョン 3.6 で変更: The default behavior of the *mangle_from_* and *maxheaderlen* parameters is to follow the policy.

`flatten`(*msg*, *unixfrom=False*, *linesep=None*)

msg を基点とするメッセージオブジェクト構造体の文字表現を出力します。出力先のファイルにはこの *Generator* インスタンスが作成されたときに指定されたものが使われます。

If the *policy* option *cte_type* is `8bit`, generate the message as if the option were set to `7bit`. (This is required because strings cannot represent non-ASCII bytes.) Convert any bytes with the high bit set as needed using an ASCII-compatible *Content-Transfer-Encoding*. That is, transform parts with non-ASCII *Content-Transfer-Encoding* (*Content-Transfer-Encoding: 8bit*) to an ASCII compatible *Content-Transfer-Encoding*, and encode RFC-invalid non-ASCII bytes in headers using the MIME `unknown-8bit` character set, thus rendering them RFC-compliant.

If *unixfrom* is `True`, print the envelope header delimiter used by the Unix mailbox format (see *mailbox*) before the first of the **RFC 5322** headers of the root message object. If the root

object has no envelope header, craft a standard one. The default is `False`. Note that for subparts, no envelope header is ever printed.

If *linesep* is not `None`, use it as the separator character between all the lines of the flattened message. If *linesep* is `None` (the default), use the value specified in the *policy*.

バージョン 3.2 で変更: `8bit` メッセージ本体の再エンコードがサポートされました。*linesep* 引数が追加されました。

`clone`(*fp*)

Return an independent clone of this *Generator* instance with the exact same options, and *fp* as the new *outfp*.

`write`(*s*)

Write *s* to the *write* method of the *outfp* passed to the *Generator*'s constructor. This provides just enough file-like API for *Generator* instances to be used in the *print()* function.

As a convenience, *EmailMessage* provides the methods *as_string()* and `str(aMessage)` (a.k.a. *__str__()*), which simplify the generation of a formatted string representation of a message object. For more detail, see *email.message*.

The *email.generator* module also provides a derived class, *DecodedGenerator*, which is like the *Generator* base class, except that non-*text* parts are not serialized, but are instead represented in the output stream by a string derived from a template filled in with information about the part.

`class email.generator.DecodedGenerator`(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *fmt=None*, ***, *policy=None*)

Act like *Generator*, except that for any subpart of the message passed to *Generator.flatten()*, if the subpart is of main type *text*, print the decoded payload of the subpart, and if the main type is not *text*, instead of printing it fill in the string *fmt* using information from the part and print the resulting filled-in string.

To fill in *fmt*, execute `fmt % part_info`, where `part_info` is a dictionary composed of the following keys and values:

- `type` -- 非 *text* 型 subpart の MIME 形式
- `maintype` -- 非 *text* 型 subpart の MIME 主形式 (maintype)
- `subtype` -- 非 *text* 型 subpart の MIME 副形式 (subtype)
- `filename` -- 非 *text* 型 subpart のファイル名
- `description` -- 非 *text* 型 subpart につけられた説明文字列
- `encoding` -- 非 *text* 型 subpart の Content-transfer-encoding

If *fmt* is `None`, use the following default *fmt*:

"[Non-text (%(type)s) part of message omitted, filename %(filename)s]"

Optional *_mangle_from_* and *maxheaderlen* are as with the *Generator* base class.

脚注

19.1.4 email.policy: ポリシーオブジェクト

バージョン 3.3 で追加.

ソースコード: Lib/email/policy.py

---

email パッケージの主要な目的は様々な E メールや MIME の RFC で記述された E メールメッセージを取り扱うことにあります。しかし、E メールメッセージの一般的なフォーマット (名前の後にコロンが続き、コロンの後に値が続くという構成の複数のヘッダーフィールドのブロック、空白行、任意の 'body') は E メールの分野外での用途が見いだされたフォーマットです。これらの用途には主となる E メールの RFC に厳密に従っているものもあれば、そうでないものもあります。E メールを使った working のときでさえも、厳密な RFC 準拠にしないのが望ましいことがあります。たとえば、基準に従わない E メールサーバーと相互運用する E メールを作成するときや、基準に違反する方法で使用したい拡張機能を実装する E メールを作成するときです。

これらのばらばらな用途に対応するため Policy オブジェクトは E メールパッケージに対して柔軟性を提供します。

A *Policy* object encapsulates a set of attributes and methods that control the behavior of various components of the email package during use. *Policy* instances can be passed to various classes and methods in the email package to alter the default behavior. The settable values and their defaults are described below.

There is a default policy used by all classes in the email package. For all of the *parser* classes and the related convenience functions, and for the *Message* class, this is the *Compat32* policy, via its corresponding pre-defined instance *compat32*. This policy provides for complete backward compatibility (in some cases, including bug compatibility) with the pre-Python3.3 version of the email package.

This default value for the *policy* keyword to *EmailMessage* is the *EmailPolicy* policy, via its pre-defined instance *default*.

When a *Message* or *EmailMessage* object is created, it acquires a policy. If the message is created by a *parser*, a policy passed to the parser will be the policy used by the message it creates. If the message is created by the program, then the policy can be specified when it is created. When a message is passed to a *generator*, the generator uses the policy from the message by default, but you can also pass a specific policy to the generator that will override the one stored on the message object.

The default value for the *policy* keyword for the *email.parser* classes and the parser convenience functions **will be changing** in a future version of Python. Therefore you should **always specify explicitly which policy you want to use** when calling any of the classes and functions described in the *parser* module.

The first part of this documentation covers the features of *Policy*, an *abstract base class* that defines the features that are common to all policy objects, including *compat32*. This includes certain hook methods that are called internally by the email package, which a custom policy could override to obtain

different behavior. The second part describes the concrete classes *EmailPolicy* and *Compat32*, which implement the hooks that provide the standard behavior and the backward compatible behavior and features, respectively.

Policy instances are immutable, but they can be cloned, accepting the same keyword arguments as the class constructor and returning a new *Policy* instance that is a copy of the original but with the specified attributes values changed.

As an example, the following code could be used to read an email message from a file on disk and pass it to the system `sendmail` program on a Unix system:

```
>>> from email import message_from_binary_file
>>> from email.generator import BytesGenerator
>>> from email import policy
>>> from subprocess import Popen, PIPE
>>> with open('mymsg.txt', 'rb') as f:
...     msg = message_from_binary_file(f, policy=policy.default)
>>> p = Popen(['sendmail', msg['To'].addresses[0]], stdin=PIPE)
>>> g = BytesGenerator(p.stdin, policy=msg.policy.clone(linesep='\r\n'))
>>> g.flatten(msg)
>>> p.stdin.close()
>>> rc = p.wait()
```

Here we are telling *BytesGenerator* to use the RFC correct line separator characters when creating the binary string to feed into `sendmail's stdin`, where the default policy would use `\n` line separators.

Some email package methods accept a *policy* keyword argument, allowing the policy to be overridden for that method. For example, the following code uses the *as_bytes()* method of the *msg* object from the previous example and writes the message to a file using the native line separators for the platform on which it is running:

```
>>> import os
>>> with open('converted.txt', 'wb') as f:
...     f.write(msg.as_bytes(policy=msg.policy.clone(linesep=os.linesep)))
17
```

Policy オブジェクトは加算オペレータを使用して組み合わせることも可能で、それらのオブジェクトの非デフォルト値を組み合わせた設定を持つ Policy オブジェクトを生成します:

```
>>> compat_SMTP = policy.compat32.clone(linesep='\r\n')
>>> compat_strict = policy.compat32.clone(raise_on_defect=True)
>>> compat_strict_SMTP = compat_SMTP + compat_strict
```

この操作には交換法則が成り立ちません。つまり、オブジェクトを加える順番に結果が依存します。次のように説明できます:

```
>>> policy100 = policy.compat32.clone(max_line_length=100)
>>> policy80 = policy.compat32.clone(max_line_length=80)
>>> apolicy = policy100 + policy80
>>> apolicy.max_line_length
```

(次のページに続く)

(前のページからの続き)

```
80
>>> apolicy = policy80 + policy100
>>> apolicy.max_line_length
100
```

class email.policy.Policy(***kw*)

This is the *abstract base class* for all policy classes. It provides default implementations for a couple of trivial methods, as well as the implementation of the immutability property, the *clone()* method, and the constructor semantics.

The constructor of a policy class can be passed various keyword arguments. The arguments that may be specified are any non-method properties on this class, plus any additional non-method properties on the concrete class. A value specified in the constructor will override the default value for the corresponding attribute.

This class defines the following properties, and thus values for the following may be passed in the constructor of any policy class:

max_line_length

行の終端文字を数に含まない、シリアライズした出力の任意の行の最大長。デフォルトは **RFC 5322** に従い 78 となっています。値が `0` または *None* の場合は行の折り返しを全くしないべきであるということを示します。

linesep

The string to be used to terminate lines in serialized output. The default is `\n` because that's the internal end-of-line discipline used by Python, though `\r\n` is required by the RFCs.

cte_type

Controls the type of Content Transfer Encodings that may be or are required to be used. The possible values are:

`7bit`	all data must be "7 bit clean" (ASCII-only). This means that where necessary data will be encoded using either quoted-printable or base64 encoding.
`8bit`	data is not constrained to be 7 bit clean. Data in headers is still required to be ASCII-only and so will be encoded (see *fold_binary()* and *utf8* below for exceptions), but body parts may use the `8bit` CTE.

A `cte_type` value of `8bit` only works with `BytesGenerator`, not `Generator`, because strings cannot contain binary data. If a `Generator` is operating under a policy that specifies `cte_type=8bit`, it will act as if `cte_type` is `7bit`.

raise_on_defect

If *True*, any defects encountered will be raised as errors. If *False* (the default), defects will be passed to the *register_defect()* method.

mangle_from_

If *True*, lines starting with *"From "* in the body are escaped by putting a > in front of them. This parameter is used when the message is being serialized by a generator. Default: *False*.

バージョン 3.5 で追加: *mangle_from_* 引数。

message_factory

A factory function for constructing a new empty message object. Used by the parser when building messages. Defaults to None, in which case *Message* is used.

バージョン 3.6 で追加.

The following *Policy* method is intended to be called by code using the email library to create policy instances with custom settings:

clone(***kw*)

Return a new *Policy* instance whose attributes have the same values as the current instance, except where those attributes are given new values by the keyword arguments.

The remaining *Policy* methods are called by the email package code, and are not intended to be called by an application using the email package. A custom policy must implement all of these methods.

handle_defect(*obj*, *defect*)

Handle a *defect* found on *obj*. When the email package calls this method, *defect* will always be a subclass of Defect.

The default implementation checks the *raise_on_defect* flag. If it is True, *defect* is raised as an exception. If it is False (the default), *obj* and *defect* are passed to *register_defect()*.

register_defect(*obj*, *defect*)

defect を *obj* に登録します。email パッケージでは、*defect* は常に Defect の派生クラスです。

The default implementation calls the **append** method of the **defects** attribute of *obj*. When the email package calls *handle_defect*, *obj* will normally have a **defects** attribute that has an **append** method. Custom object types used with the email package (for example, custom Message objects) should also provide such an attribute, otherwise defects in parsed messages will raise unexpected errors.

header_max_count(*name*)

name というヘッダに許される最大の数を返します。

Called when a header is added to an *EmailMessage* or *Message* object. If the returned value is not 0 or None, and there are already a number of headers with the name *name* greater than or equal to the value returned, a *ValueError* is raised.

Because the default behavior of Message.__setitem__ is to append the value to the list of headers, it is easy to create duplicate headers without realizing it. This method allows certain headers to be limited in the number of instances of that header that may be added to a Message programmatically. (The limit is not observed by the parser, which will faithfully produce as many headers as exist in the message being parsed.)

デフォルトの実装は全てのヘッダ名に None を返します。

header_source_parse(*sourcelines*)

The email package calls this method with a list of strings, each string ending with the line separation characters found in the source being parsed. The first line includes the field header name and separator. All whitespace in the source is preserved. The method should return the `(name, value)` tuple that is to be stored in the `Message` to represent the parsed header.

If an implementation wishes to retain compatibility with the existing email package policies, *name* should be the case preserved name (all characters up to the ':' separator), while *value* should be the unfolded value (all line separator characters removed, but whitespace kept intact), stripped of leading whitespace.

sourcelines はサロゲートエスケープされたバイナリーデータを持つことがあります。

デフォルトの実装はありません。

header_store_parse(*name*, *value*)

The email package calls this method with the name and value provided by the application program when the application program is modifying a `Message` programmatically (as opposed to a `Message` created by a parser). The method should return the `(name, value)` tuple that is to be stored in the `Message` to represent the header.

If an implementation wishes to retain compatibility with the existing email package policies, the *name* and *value* should be strings or string subclasses that do not change the content of the passed in arguments.

デフォルトの実装はありません。

header_fetch_parse(*name*, *value*)

The email package calls this method with the *name* and *value* currently stored in the `Message` when that header is requested by the application program, and whatever the method returns is what is passed back to the application as the value of the header being retrieved. Note that there may be more than one header with the same name stored in the `Message`; the method is passed the specific name and value of the header destined to be returned to the application.

value はサロゲートエスケープされたバイナリーデータを持つことがあります。このメソッドの返り値にはサロゲートエスケープされたバイナリーデータはありません。

デフォルトの実装はありません。

fold(*name*, *value*)

The email package calls this method with the *name* and *value* currently stored in the `Message` for a given header. The method should return a string that represents that header "folded" correctly (according to the policy settings) by composing the *name* with the *value* and inserting *linesep* characters at the appropriate places. See **RFC 5322** for a discussion of the rules for folding email headers.

value はサロゲートエスケープされたバイナリーデータを持つことがあります。このメソッドが返

す文字列にはサロゲートエスケープされたバイナリーデータはありません。

fold_binary(*name*, *value*)
返り値が文字列でなく bytes オブジェクトである点を除けば、*fold()* と同じです。

*value*はサロゲートエスケープされたバイナリデータを持つことがあります。これらは返された bytes オブジェクト内でバイナリデータに変換されることがあります。

class email.policy.EmailPolicy(***kw*)
This concrete *Policy* provides behavior that is intended to be fully compliant with the current email RFCs. These include (but are not limited to) **RFC 5322**, **RFC 2047**, and the current MIME RFCs.

This policy adds new header parsing and folding algorithms. Instead of simple strings, headers are `str` subclasses with attributes that depend on the type of the field. The parsing and folding algorithm fully implement **RFC 2047** and **RFC 5322**.

The default value for the *message_factory* attribute is *EmailMessage*.

In addition to the settable attributes listed above that apply to all policies, this policy adds the following additional attributes:

バージョン 3.6 で追加:[*1]

utf8
If `False`, follow **RFC 5322**, supporting non-ASCII characters in headers by encoding them as "encoded words". If `True`, follow **RFC 6532** and use `utf-8` encoding for headers. Messages formatted in this way may be passed to SMTP servers that support the `SMTPUTF8` extension (**RFC 6531**).

refold_source
If the value for a header in the `Message` object originated from a *parser* (as opposed to being set by a program), this attribute indicates whether or not a generator should refold that value when transforming the message back into serialized form. The possible values are:

`none`	all source values use original folding
`long`	source values that have any line that is longer than `max_line_length` will be refolded
`all`	all values are refolded.

デフォルトは `long` です。

header_factory
A callable that takes two arguments, `name` and `value`, where `name` is a header field name and `value` is an unfolded header field value, and returns a string subclass that represents that header. A default `header_factory` (see *headerregistry*) is provided that supports custom

[*1] Originally added in 3.3 as a *provisional feature*.

parsing for the various address and date **RFC 5322** header field types, and the major MIME header field stypes. Support for additional custom parsing will be added in the future.

`content_manager`
: An object with at least two methods: get_content and set_content. When the *get_content()* or *set_content()* method of an *EmailMessage* object is called, it calls the corresponding method of this object, passing it the message object as its first argument, and any arguments or keywords that were passed to it as additional arguments. By default `content_manager` is set to *raw_data_manager*.

  バージョン 3.4 で追加.

このクラスは *Policy* の抽象メソッドの具象実装を提供します:

header_max_count(*name*)
: Returns the value of the *max_count* attribute of the specialized class used to represent the header with the given name.

header_source_parse(*sourcelines*)
: The name is parsed as everything up to the ':' and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

header_store_parse(*name*, *value*)
: The name is returned unchanged. If the input value has a `name` attribute and it matches *name* ignoring case, the value is returned unchanged. Otherwise the *name* and *value* are passed to `header_factory`, and the resulting header object is returned as the value. In this case a `ValueError` is raised if the input value contains CR or LF characters.

header_fetch_parse(*name*, *value*)
: If the value has a `name` attribute, it is returned to unmodified. Otherwise the *name*, and the *value* with any CR or LF characters removed, are passed to the `header_factory`, and the resulting header object is returned. Any surrogateescaped bytes get turned into the unicode unknown-character glyph.

fold(*name*, *value*)
: Header folding is controlled by the *refold_source* policy setting. A value is considered to be a 'source value' if and only if it does not have a `name` attribute (having a `name` attribute means it is a header object of some sort). If a source value needs to be refolded according to the policy, it is converted into a header object by passing the *name* and the *value* with any CR and LF characters removed to the `header_factory`. Folding of a header object is done by calling its `fold` method with the current policy.

  Source values are split into lines using *splitlines()*. If the value is not to be refolded, the lines are rejoined using the `linesep` from the policy and returned. The exception is lines containing non-ascii binary data. In that case the value is refolded regardless of the `refold_source` setting, which causes the binary data to be CTE encoded using the `unknown-8bit` charset.

fold_binary(*name*, *value*)
返り値が bytes である点を除いて、*cte_type* が 7bit の場合は *fold()* と同じです。

If *cte_type* is 8bit, non-ASCII binary data is converted back into bytes. Headers with binary data are not refolded, regardless of the **refold_header** setting, since there is no way to know whether the binary data consists of single byte characters or multibyte characters.

The following instances of *EmailPolicy* provide defaults suitable for specific application domains. Note that in the future the behavior of these instances (in particular the HTTP instance) may be adjusted to conform even more closely to the RFCs relevant to their domains.

email.policy.**default**
デフォルト値を変更していない EmailPolicy のインスタンスです。このポリシーの行末は、RFC で正しい \r\n ではなく Python の標準の \n です。

email.policy.**SMTP**
Suitable for serializing messages in conformance with the email RFCs. Like **default**, but with **linesep** set to \r\n, which is RFC compliant.

email.policy.**SMTPUTF8**
The same as SMTP except that *utf8* is True. Useful for serializing messages to a message store without using encoded words in the headers. Should only be used for SMTP transmission if the sender or recipient addresses have non-ASCII characters (the *smtplib.SMTP.send_message()* method handles this automatically).

email.policy.**HTTP**
Suitable for serializing headers with for use in HTTP traffic. Like SMTP except that **max_line_length** is set to None (unlimited).

email.policy.**strict**
Convenience instance. The same as **default** except that **raise_on_defect** is set to True. This allows any policy to be made strict by writing:

```
somepolicy + policy.strict
```

With all of these *EmailPolicies*, the effective API of the email package is changed from the Python 3.2 API in the following ways:

- Setting a header on a *Message* results in that header being parsed and a header object created.
- Fetching a header value from a *Message* results in that header being parsed and a header object created and returned.
- Any header object, or any header that is refolded due to the policy settings, is folded using an algorithm that fully implements the RFC folding algorithms, including knowing where encoded words are required and allowed.

From the application view, this means that any header obtained through the *EmailMessage* is a header object with extra attributes, whose string value is the fully decoded unicode value of the header. Likewise,

a header may be assigned a new value, or a new header created, using a unicode string, and the policy will take care of converting the unicode string into the correct RFC encoded form.

ヘッダオブジェクトとその属性は *headerregistry* で述べられています。

class email.policy.Compat32(***kw*)

This concrete *Policy* is the backward compatibility policy. It replicates the behavior of the email package in Python 3.2. The *policy* module also defines an instance of this class, *compat32*, that is used as the default policy. Thus the default behavior of the email package is to maintain compatibility with Python 3.2.

以下の属性は *Policy* デフォルトとは異なる値を持ちます:

mangle_from_

デフォルトは `True` です。

このクラスは *Policy* の抽象メソッドの具象実装を提供します:

header_source_parse(*sourcelines*)

The name is parsed as everything up to the ':' and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

header_store_parse(*name*, *value*)

名前と値は変更されずに返されます。

header_fetch_parse(*name*, *value*)

If the value contains binary data, it is converted into a *Header* object using the `unknown-8bit` charset. Otherwise it is returned unmodified.

fold(*name*, *value*)

Headers are folded using the *Header* folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the `max_line_length`. Non-ASCII binary data are CTE encoded using the `unknown-8bit` charset.

fold_binary(*name*, *value*)

Headers are folded using the *Header* folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the `max_line_length`. If `cte_type` is `7bit`, non-ascii binary data is CTE encoded using the `unknown-8bit` charset. Otherwise the original source header is used, with its existing line breaks and any (RFC invalid) binary data it may contain.

email.policy.compat32

Compat32 のインスタンスで、Python 3.2 の email パッケージの挙動との後方互換性を提供します。

脚注

19.1.5 email.errors: 例外及び欠陥クラス

ソースコード: Lib/email/errors.py

---

email.errors モジュールでは、以下の例外クラスが定義されています:

exception email.errors.MessageError
これは *email* パッケージが送出しうるすべての例外の基底クラスです。これは標準の *Exception* クラスから派生しており、追加のメソッドは定義されていません。

exception email.errors.MessageParseError
This is the base class for exceptions raised by the *Parser* class. It is derived from *MessageError*. This class is also used internally by the parser used by *headerregistry*.

exception email.errors.HeaderParseError
Raised under some error conditions when parsing the **RFC 5322** headers of a message, this class is derived from *MessageParseError*. The *set_boundary()* method will raise this error if the content type is unknown when the method is called. *Header* may raise this error for certain base64 decoding errors, and when an attempt is made to create a header that appears to contain an embedded header (that is, there is what is supposed to be a continuation line that has no leading whitespace and looks like a header).

exception email.errors.BoundaryError
Deprecated and no longer used.

exception email.errors.MultipartConversionError
この例外は、*Message* オブジェクトに add_payload() メソッドでペイロードを追加したが、そのペイロードがすでにスカラー値で、メッセージの *Content-Type* メインタイプが *multipart* でないか見付からない場合に送出されます。*MultipartConversionError* は *MessageError* と組み込みの *TypeError* を多重継承しています。

Message.add_payload() は非推奨なので、実際のところこの例外が送出されることはほとんどありません。しかしながら、*attach()* メソッドを *MIMENonMultipart* から派生したインスタンス (たとえば *MIMEImage*) に対して呼んだ場合にも送出されることがあります。

以下は *FeedParser* がメッセージの解析中に検出する障害 (defect) の一覧です。これらの障害は、問題が見つかったメッセージに追加されるため、たとえば *multipart/alternative* 内にあるネストしたメッセージが異常なヘッダをもっていた場合には、そのネストしたメッセージが障害を持っているが、その親メッセージには障害はないとみなされることに注意してください。

すべての障害クラスは email.errors.MessageDefect のサブクラスです。

- NoBoundaryInMultipartDefect -- メッセージが multipart だと宣言されているのに、*boundary* パラメータがありません。
- StartBoundaryNotFoundDefect -- *Content-Type* ヘッダで宣言された開始境界がありません。

- CloseBoundaryNotFoundDefect -- 開始境界はあるが対応する終了境界がありません。

  バージョン 3.3 で追加.

- FirstHeaderLineIsContinuationDefect -- メッセージの最初のヘッダ行が継続行です。

- MisplacedEnvelopeHeaderDefect -- ヘッダブロックの途中に "Unix From" ヘッダがあります。

- MissingHeaderBodySeparatorDefect - 先頭に空白はないが ':' がないヘッダの解析中に行が見付かりました。その行を本体の最初の行とみなして解析を続けます。

  バージョン 3.3 で追加.

- MalformedHeaderDefect -- ヘッダにコロンがありません、あるいはそれ以外の不正な形式です。

  バージョン 3.3 で非推奨: この欠陥が使われていない Python バージョンがいくつかあります。

- MultipartInvariantViolationDefect -- メッセージが *multipart* だと宣言されているのに、サブパートが存在しません。メッセージがこの欠陥を持つ場合、内容の型が *multipart* と宣言されていても *is_multipart()* メソッドは False を返すことがあるので注意してください。

- InvalidBase64PaddingDefect -- 一連の base64 でエンコードされた bytes をデコードしているときにパディングが誤っていました。デコードするのに十分なパディングがなされますが、デコードされた bytes は不正かもしれません。

- InvalidBase64CharactersDefect -- 一連の base64 でエンコードされた bytes をデコードしているときに base64 アルファベット外の文字がありました。その文字は無視されますが、デコードされた bytes は不正かもしれません。

- InvalidBase64LengthDefect -- When decoding a block of base64 encoded bytes, the number of non-padding base64 characters was invalid (1 more than a multiple of 4). The encoded block was kept as-is.

19.1.6 email.headerregistry: カスタムヘッダーオブジェクト

ソースコード: Lib/email/headerregistry.py

バージョン 3.6 で追加:[*1]

ヘッダーは:class:*str* のカスタマイズされたサブクラスで表現されます。与えられたヘッダーを表現するために使用される特定のクラスは、ヘッダーが作成されるときに有効な *policy* の *header_factory* で決定されます。このセクションでは、email ライブラリが **RFC 5322** に準拠したメールメッセージを扱うために実装している、特定の header_factory について説明します。このヘッダーは、様々なヘッダータイプに対してカスタマイズされたヘッダーオブジェクトを提供するだけでなく、アプリケーションが独自のカスタムヘッダータイプを追加できるような拡張メカニズムも備えています。

EmailPolicy から派生したポリシーオブジェクトのいずれかを使用する場合、すべてのヘッダーは *HeaderRegistry* によって生成され、最終的な基底クラスとして *BaseHeader* を有します。各ヘッダク

[*1] Originally added in 3.3 as a *provisional module*

ラスには、ヘッダの種類によって決まる追加のベースクラスがあります。例えば、多くのヘッダーは *UnstructuredHeader* というクラスをもうひとつの基底クラスとして持っています。ヘッダーに特化した第二のクラスは、ヘッダーの名前によって決定され、*HeaderRegistry* に格納されたルックアップテーブルを使用します。これらすべては、典型的なアプリケーションプログラムに対しては透過的に管理されますが、より複雑なアプリケーションで使用するためにデフォルトの動作を変更するためのインタフェースが提供されています。

以下のセクションでは、まずヘッダーの基本クラスとその属性を説明し、次に *HeaderRegistry* の動作を変更するための API、そして最後に構造化ヘッダーからパースされたデータを表現するためのサポートクラスについて説明します。

class email.headerregistry.**BaseHeader**(*name*, *value*)

name と *value* は *header_factory* 呼び出しから BaseHeader に渡されます。どのヘッダーオブジェクトの文字列値も、完全にユニコードにデコードされた *value* です。

基底クラスは以下の読み出し専用属性を定義しています:

name

ヘッダーの名前 (フィールドの':' の前の部分)。これは *header_factory* の *name* の呼び出しで渡された値と全く同じです; つまり、大文字・小文字が保持されます。

defects

パース中に見つかった RFC のコンプライアンス問題を報告する HeaderDefect インスタンスのタプルです。email パッケージはコンプライアンスに関する問題を完全に検出するように努めています。報告されるかもしれない欠陥の種類の議論については、*errors* モジュールを参照してください。

max_count

このタイプのヘッダーで、同じ name を持つことができる最大数。値として None を指定すると、無制限になります。この属性の BaseHeader の値は None です; 特殊な header クラスでは、必要に応じてこの値をオーバーライドすることが期待されます。

BaseHeader は以下のメソッドも提供します。このメソッドは email ライブラリのコードから呼び出され、一般にアプリケーションプログラムからは呼び出されません：

fold(***, *policy*)

ヘッダを *policy* に従って正しく折りたたむために必要な *linesep* 文字を含む文字列を返します。ヘッダには任意のバイナリデータを含めることができないため、*cte_type* が 8bit の場合は 7bit と同じように扱われます。もし *utf8* が False であれば、非 ASCII データは **RFC 2047** でエンコードされます。

BaseHeader 自身はヘッダーオブジェクトを生成するために使用することはできません。このクラスは、それぞれの特殊化ヘッダーがヘッダーオブジェクトを生成するために協力するプロトコルを定義しています。具体的には、BaseHeader は parse という名前の *classmethod()* を特殊化されたクラスが提供することを要求しています。このメソッドは以下のように呼び出されます:

```
parse(string, kwds)
```

kwds は、初期化されたキー defects を含む辞書です。defects は空のリストです。解析メソッドは、検出された不具合をこのリストに追加するべきです。return の際には、kwds 辞書に少なくとも decoded と “defects“ のキーを含む必要があります。decoded はヘッダの文字列値 (つまり、ヘッダを Unicode に完全にデコードした値) でなければなりません。parse メソッドは、*string* が content-transfer-encoded な部分を含む可能性があると仮定すべきですが、エンコードされていないヘッダ値をパースできるように、すべての有効な Unicode 文字も正しく処理する必要があります。

その後、BaseHeader の __new__ がヘッダのインスタンスを生成し、init メソッドを呼び出します。特殊クラスは、BaseHeader が提供しない属性を設定したい場合にのみ init メソッドを提供する必要があります。このような “init“ メソッドは以下のようなものです：

```
def init(self, /, *args, **kw):
    self._myattr = kw.pop('myattr')
    super().init(*args, **kw)
```

つまり、特殊クラスが kwds の辞書に追加したものはすべて削除して処理し、kw``の残りの内容（と``args) は “BaseHeader“ の “init“ メソッドに渡す必要があるのです。

class email.headerregistry.UnstructuredHeader

”unstructured” ヘッダは **RFC 5322** におけるデフォルトのヘッダのタイプです。指定された構文を持たないすべてのヘッダは、unstructured として扱われます。構造化されていないヘッダの典型的な例は *Subject* ヘッダです。

RFC 5322 では、非構造化ヘッダは ASCII 文字セットの任意のテキストの集まりです。しかし、**RFC 2047** にはヘッダ値の中で非 ASCII テキストを ASCII 文字としてエンコードするための **RFC 5322** と互換性のあるメカニズムが備わっています。エンコードされた単語を含む *value* がコンストラクタに渡されると、UnstructuredHeader パーサは非構造化テキストに対する **RFC 2047** 規則に従って、エンコードされた単語を unicode に変換します。パーサは特定の非適合なエンコードされた単語のデコードを試みるためにヒューリスティックを使用します。このような場合、欠陥が登録されます。また、エンコードされた単語やエンコードされていないテキスト内の無効な文字などの問題に対しても欠陥が登録されます。

このヘッダ型には追加の属性はありません。

class email.headerregistry.DateHeader

RFC 5322 specifies a very specific format for dates within email headers. The DateHeader parser recognizes that date format, as well as recognizing a number of variant forms that are sometimes found ”in the wild”.

このヘッダ型は以下の属性も提供しています:

datetime

If the header value can be recognized as a valid date of one form or another, this attribute will contain a *datetime* instance representing that date. If the timezone of the input date is specified as -0000 (indicating it is in UTC but contains no information about the source timezone), then *datetime* will be a naive *datetime*. If a specific timezone offset is found (including *+0000*), then *datetime* will contain an aware datetime that uses *datetime.timezone* to record the timezone offset.

ヘッダーの decoded 値は、datetime を **RFC 5322** のルールに従ってフォーマットすることで決定されます。

```
email.utils.format_datetime(self.datetime)
```

DateHeader を作成する際に、*value* は *datetime* インスタンスである可能性があります。これは、例えば、次のようなコードは有効であり、期待通りの動作をすることを意味します:

```
msg['Date'] = datetime(2011, 7, 15, 21)
```

これは単純な datetime であるため、UTC タイムスタンプとして解釈され、結果として -0000 というタイムゾーンを持つ値になります。もっと便利なのは *utils* モジュールの *localtime()* 関数を使用することです:

```
msg['Date'] = utils.localtime()
```

この例では、現在のタイムゾーンオフセットを使用して、日付ヘッダーを現在の時刻と日付に設定します。

class email.headerregistry.**AddressHeader**

アドレスヘッダは最も複雑な構造のヘッダタイプの 1 つです。AddressHeader クラスは、あらゆるアドレスヘッダに対する汎用的なインターフェースを提供します。

このヘッダ型は以下の属性も提供しています:

groups

ヘッダー値に含まれるアドレスとグループをエンコードした *Group* オブジェクトのタプルです。グループに属さないアドレスは、このリストでは *display_name* が None であるシングルアドレスの Groups として表現されます。

addresses

ヘッダー値に含まれる全ての個別アドレスをエンコードした *Address* オブジェクトのタプルです。ヘッダ値にグループが含まれている場合、そのグループに含まれる個々のアドレスは、ヘッダ値でグループが出現した時点でリストに含まれます (つまり、アドレスのリストは一次元のリストに「フラット化」されます)。

The decoded value of the header will have all encoded words decoded to unicode. *idna* encoded domain names are also decoded to unicode. The decoded value is set by *join*ing the *str* value of the elements of the groups attribute with ', '.

A list of *Address* and *Group* objects in any combination may be used to set the value of an address header. Group objects whose display_name is None will be interpreted as single addresses, which allows an address list to be copied with groups intact by using the list obtained from the groups attribute of the source header.

class email.headerregistry.**SingleAddressHeader**

追加の属性を 1 つ持つ、*AddressHeader* の派生クラスです:

address

The single address encoded by the header value. If the header value actually contains more than one address (which would be a violation of the RFC under the default *policy*), accessing this attribute will result in a *ValueError*.

Many of the above classes also have a `Unique` variant (for example, `UniqueUnstructuredHeader`). The only difference is that in the `Unique` variant, *max_count* is set to 1.

class email.headerregistry.MIMEVersionHeader

There is really only one valid value for the *MIME-Version* header, and that is `1.0`. For future proofing, this header class supports other valid version numbers. If a version number has a valid value per **RFC 2045**, then the header object will have non-`None` values for the following attributes:

version

バージョン番号 (文字列)、空白やコメントは除かれます。

major

メジャーバージョン番号 (整数)

minor

マイナーバージョン番号 (整数)

class email.headerregistry.ParameterizedMIMEHeader

MIME headers all start with the prefix 'Content-'. Each specific header has a certain value, described under the class for that header. Some can also take a list of supplemental parameters, which have a common format. This class serves as a base for all the MIME headers that take parameters.

params

A dictionary mapping parameter names to parameter values.

class email.headerregistry.ContentTypeHeader

Content-Type ヘッダを扱う *ParameterizedMIMEHeader* クラスです。

content_type

The content type string, in the form `maintype/subtype`.

maintype

subtype

class email.headerregistry.ContentDispositionHeader

Content-Disposition ヘッダを扱う *ParameterizedMIMEHeader* クラスです。

content_disposition

`inline` and `attachment` are the only valid values in common use.

class email.headerregistry.ContentTransferEncoding

Content-Transfer-Encoding ヘッダを扱います。

`cte`
有効な値は `7bit`、`8bit`、`base64`、`quoted-printable` です。詳細については **RFC 2045** を参照してください。

class email.headerregistry.**HeaderRegistry**(*base_class=BaseHeader*, *default_class=UnstructuredHeader*, *use_default_map=True*)

This is the factory used by *EmailPolicy* by default. `HeaderRegistry` builds the class used to create a header instance dynamically, using *base_class* and a specialized class retrieved from a registry that it holds. When a given header name does not appear in the registry, the class specified by *default_class* is used as the specialized class. When *use_default_map* is `True` (the default), the standard mapping of header names to classes is copied in to the registry during initialization. *base_class* is always the last class in the generated class's `__bases__` list.

デフォルトのマッピング:

subject　UniqueUnstructuredHeader

date　UniqueDateHeader

resent-date　DateHeader

orig-date　UniqueDateHeader

sender　UniqueSingleAddressHeader

resent-sender　SingleAddressHeader

to　UniqueAddressHeader

resent-to　AddressHeader

cc　UniqueAddressHeader

resent-cc　AddressHeader

bcc　UniqueAddressHeader

resent-bcc　AddressHeader

from　UniqueAddressHeader

resent-from　AddressHeader

reply-to　UniqueAddressHeader

mime-version　MIMEVersionHeader

content-type　ContentTypeHeader

content-disposition　ContentDispositionHeader

content-transfer-encoding　ContentTransferEncodingHeader

message-id　MessageIDHeader

HeaderRegistry には以下のメソッドがあります:

map_to_type(*self*, *name*, *cls*)

name is the name of the header to be mapped. It will be converted to lower case in the registry. *cls* is the specialized class to be used, along with *base_class*, to create the class used to instantiate headers that match *name*.

__getitem__(*name*)

Construct and return a class to handle creating a *name* header.

__call__(*name*, *value*)

Retrieves the specialized header associated with *name* from the registry (using *default_class* if *name* does not appear in the registry) and composes it with *base_class* to produce a class, calls the constructed class's constructor, passing it the same argument list, and finally returns the class instance created thereby.

The following classes are the classes used to represent data parsed from structured headers and can, in general, be used by an application program to construct structured values to assign to specific headers.

class email.headerregistry.Address(*display_name=''*, *username=''*, *domain=''*, *addr_spec=None*)

このクラスは電子メールのアドレスを表すのに使われます。アドレスの一般的な形式は:

```
[display_name] <username@domain>
```

もしくは:

```
username@domain
```

where each part must conform to specific syntax rules spelled out in **RFC 5322**.

As a convenience *addr_spec* can be specified instead of *username* and *domain*, in which case *username* and *domain* will be parsed from the *addr_spec*. An *addr_spec* must be a properly RFC quoted string; if it is not Address will raise an error. Unicode characters are allowed and will be property encoded when serialized. However, per the RFCs, unicode is *not* allowed in the username portion of the address.

display_name

The display name portion of the address, if any, with all quoting removed. If the address does not have a display name, this attribute will be an empty string.

username

アドレスの username 部分、クォートは取り除かれます。

domain

アドレスの domain 部分。

addr_spec

The username@domain portion of the address, correctly quoted for use as a bare address (the second form shown above). This attribute is not mutable.

__str__()

The str value of the object is the address quoted according to **RFC 5322** rules, but with no Content Transfer Encoding of any non-ASCII characters.

To support SMTP (**RFC 5321**), Address handles one special case: if username and domain are both the empty string (or None), then the string value of the Address is <>.

class email.headerregistry.Group(*display_name=None*, *addresses=None*)

このクラスはアドレスグループを表すのに使われます。アドレスグループの一般的な形式は:

```
display_name: [address-list];
```

As a convenience for processing lists of addresses that consist of a mixture of groups and single addresses, a Group may also be used to represent single addresses that are not part of a group by setting *display_name* to None and providing a list of the single address as *addresses*.

display_name

The display_name of the group. If it is None and there is exactly one Address in addresses, then the Group represents a single address that is not in a group.

addresses

A possibly empty tuple of *Address* objects representing the addresses in the group.

__str__()

The str value of a Group is formatted according to **RFC 5322**, but with no Content Transfer Encoding of any non-ASCII characters. If display_name is none and there is a single Address in the addresses list, the str value will be the same as the str of that single Address.

脚注

19.1.7 email.contentmanager: MIME 内容の管理

ソースコード: Lib/email/contentmanager.py

バージョン 3.6 で追加:[*1]

class email.contentmanager.ContentManager

Base class for content managers. Provides the standard registry mechanisms to register converters between MIME content and other representations, as well as the get_content and set_content dispatch methods.

get_content(*msg*, **args*, ***kw*)

Look up a handler function based on the mimetype of *msg* (see next paragraph), call it, passing through all arguments, and return the result of the call. The expectation is that the handler

[*1] Originally added in 3.4 as a *provisional module*

will extract the payload from *msg* and return an object that encodes information about the extracted data.

To find the handler, look for the following keys in the registry, stopping with the first one found:

- 完全な MIME 型を表す文字列 (`maintype/subtype`)
- `maintype` を表す文字列
- 空の文字列

If none of these keys produce a handler, raise a *KeyError* for the full MIME type.

set_content(*msg*, *obj*, **args*, ***kw*)

If the `maintype` is `multipart`, raise a *TypeError*; otherwise look up a handler function based on the type of *obj* (see next paragraph), call *clear_content()* on the *msg*, and call the handler function, passing through all arguments. The expectation is that the handler will transform and store *obj* into *msg*, possibly making other changes to *msg* as well, such as adding various MIME headers to encode information needed to interpret the stored data.

To find the handler, obtain the type of *obj* (`typ = type(obj)`), and look for the following keys in the registry, stopping with the first one found:

- 型自身 (`typ`)
- 型の完全修飾名 (`typ.__module__ + '.' + typ.__qualname__`).
- 型の qualname (`typ.__qualname__`)
- 型の name (`typ.__name__`).

If none of the above match, repeat all of the checks above for each of the types in the *MRO* (`typ.__mro__`). Finally, if no other key yields a handler, check for a handler for the key `None`. If there is no handler for `None`, raise a *KeyError* for the fully qualified name of the type.

Also add a ***MIME-Version*** header if one is not present (see also *MIMEPart*).

add_get_handler(*key*, *handler*)

Record the function *handler* as the handler for *key*. For the possible values of *key*, see *get_content()*.

add_set_handler(*typekey*, *handler*)

Record *handler* as the function to call when an object of a type matching *typekey* is passed to *set_content()*. For the possible values of *typekey*, see *set_content()*.

Content Manager Instances

Currently the email package provides only one concrete content manager, *raw_data_manager*, although more may be added in the future. *raw_data_manager* is the *content_manager* provided by *EmailPolicy* and its derivatives.

email.contentmanager.raw_data_manager

This content manager provides only a minimum interface beyond that provided by *Message* itself: it deals only with text, raw byte strings, and *Message* objects. Nevertheless, it provides significant advantages compared to the base API: `get_content` on a text part will return a unicode string without the application needing to manually decode it, `set_content` provides a rich set of options for controlling the headers added to a part and controlling the content transfer encoding, and it enables the use of the various `add_` methods, thereby simplifying the creation of multipart messages.

email.contentmanager.get_content(*msg*, *errors='replace'*)

Return the payload of the part as either a string (for `text` parts), an *EmailMessage* object (for `message/rfc822` parts), or a `bytes` object (for all other non-multipart types). Raise a *KeyError* if called on a `multipart`. If the part is a `text` part and *errors* is specified, use it as the error handler when decoding the payload to unicode. The default error handler is `replace`.

email.contentmanager.set_content(*msg*, *<'str'>*, *subtype="plain"*, *charset='utf-8'*, *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)
email.contentmanager.set_content(*msg*, *<'bytes'>*, *maintype*, *subtype*, *cte="base64"*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)
email.contentmanager.set_content(*msg*, *<'EmailMessage'>*, *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)

Add headers and payload to *msg*:

Add a *Content-Type* header with a `maintype/subtype` value.

- For `str`, set the MIME `maintype` to `text`, and set the subtype to *subtype* if it is specified, or `plain` if it is not.
- For `bytes`, use the specified *maintype* and *subtype*, or raise a *TypeError* if they are not specified.
- For *EmailMessage* objects, set the maintype to `message`, and set the subtype to *subtype* if it is specified or `rfc822` if it is not. If *subtype* is `partial`, raise an error (`bytes` objects must be used to construct `message/partial` parts).

If *charset* is provided (which is valid only for `str`), encode the string to bytes using the specified character set. The default is `utf-8`. If the specified *charset* is a known alias for a standard MIME charset name, use the standard charset instead.

If *cte* is set, encode the payload using the specified content transfer encoding, and

set the *Content-Transfer-Encoding* header to that value. Possible values for *cte* are `quoted-printable`, `base64`, `7bit`, `8bit`, and `binary`. If the input cannot be encoded in the specified encoding (for example, specifying a *cte* of `7bit` for an input that contains non-ASCII values), raise a *ValueError*.

- For `str` objects, if *cte* is not set use heuristics to determine the most compact encoding.
- For *EmailMessage*, per **RFC 2046**, raise an error if a *cte* of `quoted-printable` or `base64` is requested for *subtype* `rfc822`, and for any *cte* other than `7bit` for *subtype* `external-body`. For `message/rfc822`, use `8bit` if *cte* is not specified. For all other values of *subtype*, use `7bit`.

注釈: A *cte* of `binary` does not actually work correctly yet. The `EmailMessage` object as modified by `set_content` is correct, but *BytesGenerator* does not serialize it correctly.

If *disposition* is set, use it as the value of the *Content-Disposition* header. If not specified, and *filename* is specified, add the header with the value `attachment`. If *disposition* is not specified and *filename* is also not specified, do not add the header. The only valid values for *disposition* are `attachment` and `inline`.

If *filename* is specified, use it as the value of the `filename` parameter of the *Content-Disposition* header.

If *cid* is specified, add a *Content-ID* header with *cid* as its value.

If *params* is specified, iterate its `items` method and use the resulting `(key, value)` pairs to set additional parameters on the *Content-Type* header.

If *headers* is specified and is a list of strings of the form `headername: headervalue` or a list of `header` objects (distinguished from strings by having a `name` attribute), add the headers to *msg*.

脚注

19.1.8 email: 使用例

ここでは *email* パッケージを使って電子メールメッセージを読む・書く・送信するいくつかの例を紹介します。より複雑な MIME メッセージについても扱います。

First, let's see how to create and send a simple text message (both the text content and the addresses may contain unicode characters):

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
```

(次のページに続く)

(前のページからの続き)

```
from email.message import EmailMessage

# Open the plain text file whose name is in textfile for reading.
with open(textfile) as fp:
    # Create a text/plain message
    msg = EmailMessage()
    msg.set_content(fp.read())

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = f'The contents of {textfile}'
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server.
s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()
```

Parsing **RFC 822** headers can easily be done by the using the classes from the *parser* module:

```
# Import the email modules we'll need
from email.parser import BytesParser, Parser
from email.policy import default

# If the e-mail headers are in a file, uncomment these two lines:
# with open(messagefile, 'rb') as fp:
#     headers = BytesParser(policy=default).parse(fp)

#  Or for parsing headers in a string (this is an uncommon operation), use:
headers = Parser(policy=default).parsestr(
        'From: Foo Bar <user@example.com>\n'
        'To: <someone_else@example.com>\n'
        'Subject: Test message\n'
        '\n'
        'Body would go here\n')

#  Now the header items can be accessed as a dictionary:
print('To: {}'.format(headers['to']))
print('From: {}'.format(headers['from']))
print('Subject: {}'.format(headers['subject']))

# You can also access the parts of the addresses:
print('Recipient username: {}'.format(headers['to'].addresses[0].username))
print('Sender name: {}'.format(headers['from'].addresses[0].display_name))
```

つぎに、あるディレクトリ内にある何枚かの家族写真をひとつの MIME メッセージに収めて送信する例です:

```
# Import smtplib for the actual sending function
import smtplib

```

(次のページに続く)

(前のページからの続き)

```
# And imghdr to find the types of our images
import imghdr

# Here are the email package modules we'll need
from email.message import EmailMessage

# Create the container email message.
msg = EmailMessage()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = ', '.join(family)
msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

# Open the files in binary mode.  Use imghdr to figure out the
# MIME subtype for each specific image.
for file in pngfiles:
    with open(file, 'rb') as fp:
        img_data = fp.read()
    msg.add_attachment(img_data, maintype='image',
                                 subtype=imghdr.what(None, img_data))

# Send the email via our own SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

つぎはあるディレクトリに含まれている内容全体をひとつの電子メールメッセージとして送信するやり方です:[*1]

```
#!/usr/bin/env python3

"""Send the contents of a directory as a MIME message."""

import os
import smtplib
# For guessing MIME type based on file name extension
import mimetypes

from argparse import ArgumentParser

from email.message import EmailMessage
from email.policy import SMTP


def main():
    parser = ArgumentParser(description="""\
Send the contents of a directory as a MIME message.
Unless the -o option is given, the email is sent by forwarding to your local
```

(次のページに続く)

[*1] 最初の思いつきと用例は Matthew Dixon Cowles のおかげです。

(前のページからの続き)

```
SMTP server, which then does the normal delivery process.  Your local machine
must be running an SMTP server.
""")
    parser.add_argument('-d', '--directory',
                        help="""Mail the contents of the specified directory,
                        otherwise use the current directory.  Only the regular
                        files in the directory are sent, and we don't recurse to
                        subdirectories.""")
    parser.add_argument('-o', '--output',
                        metavar='FILE',
                        help="""Print the composed message to FILE instead of
                        sending the message to the SMTP server.""")
    parser.add_argument('-s', '--sender', required=True,
                        help='The value of the From: header (required)')
    parser.add_argument('-r', '--recipient', required=True,
                        action='append', metavar='RECIPIENT',
                        default=[], dest='recipients',
                        help='A To: header value (at least one required)')
    args = parser.parse_args()
    directory = args.directory
    if not directory:
        directory = '.'
    # Create the message
    msg = EmailMessage()
    msg['Subject'] = f'Contents of directory {os.path.abspath(directory)}'
    msg['To'] = ', '.join(args.recipients)
    msg['From'] = args.sender
    msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

    for filename in os.listdir(directory):
        path = os.path.join(directory, filename)
        if not os.path.isfile(path):
            continue
        # Guess the content type based on the file's extension.  Encoding
        # will be ignored, although we should check for simple things like
        # gzip'd or compressed files.
        ctype, encoding = mimetypes.guess_type(path)
        if ctype is None or encoding is not None:
            # No guess could be made, or the file is encoded (compressed), so
            # use a generic bag-of-bits type.
            ctype = 'application/octet-stream'
        maintype, subtype = ctype.split('/', 1)
        with open(path, 'rb') as fp:
            msg.add_attachment(fp.read(),
                               maintype=maintype,
                               subtype=subtype,
                               filename=filename)
    # Now send or store the message
    if args.output:
        with open(args.output, 'wb') as fp:
            fp.write(msg.as_bytes(policy=SMTP))
```

(次のページに続く)

(前のページからの続き)

```
    else:
        with smtplib.SMTP('localhost') as s:
            s.send_message(msg)


if __name__ == '__main__':
    main()
```

つぎに、上のような MIME メッセージをどうやって展開してひとつのディレクトリ上の複数ファイルにするかを示します:

```
#!/usr/bin/env python3

"""Unpack a MIME message into a directory of files."""

import os
import email
import mimetypes

from email.policy import default

from argparse import ArgumentParser


def main():
    parser = ArgumentParser(description="""\
Unpack a MIME message into a directory of files.
""")
    parser.add_argument('-d', '--directory', required=True,
                        help="""Unpack the MIME message into the named
                        directory, which will be created if it doesn't already
                        exist.""")
    parser.add_argument('msgfile')
    args = parser.parse_args()

    with open(args.msgfile, 'rb') as fp:
        msg = email.message_from_binary_file(fp, policy=default)

    try:
        os.mkdir(args.directory)
    except FileExistsError:
        pass

    counter = 1
    for part in msg.walk():
        # multipart/* are just containers
        if part.get_content_maintype() == 'multipart':
            continue
        # Applications should really sanitize the given filename so that an
        # email message can't be used to overwrite important files
        filename = part.get_filename()
```

(次のページに続く)

(前のページからの続き)

```
        if not filename:
            ext = mimetypes.guess_extension(part.get_content_type())
            if not ext:
                # Use a generic bag-of-bits extension
                ext = '.bin'
            filename = f'part-{counter:03d}{ext}'
        counter += 1
        with open(os.path.join(args.directory, filename), 'wb') as fp:
            fp.write(part.get_payload(decode=True))


if __name__ == '__main__':
    main()
```

Here's an example of how to create an HTML message with an alternative plain text version. To make things a bit more interesting, we include a related image in the html part, and we save a copy of what we are going to send to disk, as well as sending it.

```
#!/usr/bin/env python3

import smtplib

from email.message import EmailMessage
from email.headerregistry import Address
from email.utils import make_msgid

# Create the base text message.
msg = EmailMessage()
msg['Subject'] = "Ayons asperges pour le déjeuner"
msg['From'] = Address("Pepé Le Pew", "pepe", "example.com")
msg['To'] = (Address("Penelope Pussycat", "penelope", "example.com"),
             Address("Fabrette Pussycat", "fabrette", "example.com"))
msg.set_content("""\
Salut!

Cela ressemble à un excellent recipie[1] déjeuner.

[1] http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718

--Pepé
""")

# Add the html version.  This converts the message into a multipart/alternative
# container, with the original text message as the first part and the new html
# message as the second part.
asparagus_cid = make_msgid()
msg.add_alternative("""\
<html>
  <head></head>
  <body>
```

(次のページに続く)

(前のページからの続き)

```
    <p>Salut!</p>
    <p>Cela ressemble à un excellent
        <a href="http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718">
            recipie
        </a> déjeuner.
    </p>
    <img src="cid:{asparagus_cid}" />
  </body>
</html>
""".format(asparagus_cid=asparagus_cid[1:-1]), subtype='html')
# note that we needed to peel the <> off the msgid for use in the html.

# Now add the related image to the html part.
with open("roasted-asparagus.jpg", 'rb') as img:
    msg.get_payload()[1].add_related(img.read(), 'image', 'jpeg',
                                     cid=asparagus_cid)

# Make a local copy of what we are going to send.
with open('outgoing.msg', 'wb') as f:
    f.write(bytes(msg))

# Send the message via local SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

If we were sent the message from the last example, here is one way we could process it:

```
import os
import sys
import tempfile
import mimetypes
import webbrowser

# Import the email modules we'll need
from email import policy
from email.parser import BytesParser

# An imaginary module that would make this work and be safe.
from imaginary import magic_html_parser

# In a real program you'd get the filename from the arguments.
with open('outgoing.msg', 'rb') as fp:
    msg = BytesParser(policy=policy.default).parse(fp)

# Now the header items can be accessed as a dictionary, and any non-ASCII will
# be converted to unicode:
print('To:', msg['to'])
print('From:', msg['from'])
print('Subject:', msg['subject'])

# If we want to print a preview of the message content, we can extract whatever
```

(次のページに続く)

(前のページからの続き)

```
# the least formatted payload is and print the first three lines.  Of course,
# if the message has no plain text part printing the first three lines of html
# is probably useless, but this is just a conceptual example.
simplest = msg.get_body(preferencelist=('plain', 'html'))
print()
print(''.join(simplest.get_content().splitlines(keepends=True)[:3]))

ans = input("View full message?")
if ans.lower()[0] == 'n':
    sys.exit()

# We can extract the richest alternative in order to display it:
richest = msg.get_body()
partfiles = {}
if richest['content-type'].maintype == 'text':
    if richest['content-type'].subtype == 'plain':
        for line in richest.get_content().splitlines():
            print(line)
        sys.exit()
    elif richest['content-type'].subtype == 'html':
        body = richest
    else:
        print("Don't know how to display {}".format(richest.get_content_type()))
        sys.exit()
elif richest['content-type'].content_type == 'multipart/related':
    body = richest.get_body(preferencelist=('html'))
    for part in richest.iter_attachments():
        fn = part.get_filename()
        if fn:
            extension = os.path.splitext(part.get_filename())[1]
        else:
            extension = mimetypes.guess_extension(part.get_content_type())
        with tempfile.NamedTemporaryFile(suffix=extension, delete=False) as f:
            f.write(part.get_content())
            # again strip the <> to go from email form of cid to html form.
            partfiles[part['content-id'][1:-1]] = f.name
else:
    print("Don't know how to display {}".format(richest.get_content_type()))
    sys.exit()
with tempfile.NamedTemporaryFile(mode='w', delete=False) as f:
    # The magic_html_parser has to rewrite the href="cid:...." attributes to
    # point to the filenames in partfiles.  It also has to do a safety-sanitize
    # of the html.  It could be written using html.parser.
    f.write(magic_html_parser(body.get_content(), partfiles))
webbrowser.open(f.name)
os.remove(f.name)
for fn in partfiles.values():
    os.remove(fn)

# Of course, there are lots of email messages that could break this simple
# minded program, but it will handle the most common ones.
```

Up to the prompt, the output from the above is:

```
To: Penelope Pussycat <penelope@example.com>, Fabrette Pussycat <fabrette@example.com>
From: Pepé Le Pew <pepe@example.com>
Subject: Ayons asperges pour le déjeuner

Salut!

Cela ressemble à un excellent recipie[1] déjeuner.
```

脚注

レガシー API:

19.1.9 email.message.Message: compat32 API を使用した電子メールメッセージの表現

The *Message* class is very similar to the *EmailMessage* class, without the methods added by that class, and with the default behavior of certain other methods being slightly different. We also document here some methods that, while supported by the *EmailMessage* class, are not recommended unless you are dealing with legacy code.

The philosophy and structure of the two classes is otherwise the same.

This document describes the behavior under the default (for *Message*) policy *Compat32*. If you are going to use another policy, you should be using the *EmailMessage* class instead.

An email message consists of *headers* and a *payload*. Headers must be **RFC 5322** style names and values, where the field name and value are separated by a colon. The colon is not part of either the field name or the field value. The payload may be a simple text message, or a binary object, or a structured sequence of sub-messages each with their own set of headers and their own payload. The latter type of payload is indicated by the message having a MIME type such as *multipart/** or *message/rfc822*.

The conceptual model provided by a *Message* object is that of an ordered dictionary of headers with additional methods for accessing both specialized information from the headers, for accessing the payload, for generating a serialized version of the message, and for recursively walking over the object tree. Note that duplicate headers are supported but special methods must be used to access them.

The *Message* pseudo-dictionary is indexed by the header names, which must be ASCII values. The values of the dictionary are strings that are supposed to contain only ASCII characters; there is some special handling for non-ASCII input, but it doesn't always produce the correct results. Headers are stored and returned in case-preserving form, but field names are matched case-insensitively. There may also be a single envelope header, also known as the *Unix-From* header or the `From_` header. The *payload* is either a string or bytes, in the case of simple message objects, or a list of *Message* objects, for MIME container documents (e.g. *multipart/** and *message/rfc822*).

Message クラスのメソッドは以下のとおりです:

class email.message.Message(*policy=compat32*)

If *policy* is specified (it must be an instance of a *policy* class) use the rules it specifies to update and serialize the representation of the message. If *policy* is not set, use the *compat32* policy, which maintains backward compatibility with the Python 3.2 version of the email package. For more information see the *policy* documentation.

バージョン 3.3 で変更: キーワード引数 *policy* が追加されました。

as_string(*unixfrom=False*, *maxheaderlen=0*, *policy=None*)

Return the entire message flattened as a string. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. For backward compatibility reasons, *maxheaderlen* defaults to `0`, so if you want a different value you must override it explicitly (the value specified for *max_line_length* in the policy will be ignored by this method). The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the `Generator`.

もし、文字列への変換を完全に行うためにデフォルト値を埋める必要がある場合、メッセージのフラット化は *Message* の変更を引き起こす可能性があります (例えば、MIME の境界が生成される、変更される等)。

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with `From` that is required by the unix mbox format. For more flexibility, instantiate a *Generator* instance and use its *flatten()* method directly. For example:

```
from io import StringIO
from email.generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

If the message object contains binary data that is not encoded according to RFC standards, the non-compliant data will be replaced by unicode "unknown character" code points. (See also *as_bytes()* and *BytesGenerator*.)

バージョン 3.4 で変更: *policy* キーワード引数が追加されました。

__str__()

as_string() と等価です。これにより `str(msg)` は書式化されたメッセージの文字列を作ります。

as_bytes(*unixfrom=False*, *policy=None*)

Return the entire message flattened as a bytes object. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the `BytesGenerator`.

もし、文字列への変換を完全に行うためにデフォルト値を埋める必要がある場合、メッセージのフラット化は *Message* の変更を引き起こす可能性があります (例えば、MIME の境界が生成される、変更される等)。

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with `From` that is required by the unix mbox format. For more flexibility, instantiate a *BytesGenerator* instance and use its *flatten()* method directly. For example:

```
from io import BytesIO
from email.generator import BytesGenerator
fp = BytesIO()
g = BytesGenerator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

バージョン 3.4 で追加.

__bytes__()

as_bytes() と等価です。これにより `bytes(msg)` は書式化されたメッセージの bytes オブジェクトを作ります。

バージョン 3.4 で追加.

is_multipart()

Return `True` if the message's payload is a list of sub-*Message* objects, otherwise return `False`. When *is_multipart()* returns `False`, the payload should be a string object (which might be a CTE encoded binary payload). (Note that *is_multipart()* returning `True` does not necessarily mean that "msg.get_content_maintype() == 'multipart'" will return the `True`. For example, `is_multipart` will return `True` when the *Message* is of type `message/rfc822`.)

set_unixfrom(*unixfrom*)

メッセージのエンベロープヘッダを *unixfrom* に設定します。これは文字列でなければなりません。

get_unixfrom()

メッセージのエンベロープヘッダを返します。エンベロープヘッダが設定されていない場合は `None` が返されます。

attach(*payload*)

与えられた *payload* を現在のペイロードに追加します。この時点でのペイロードは `None` か、あるいは *Message* オブジェクトのリストである必要があります。このメソッドの実行後、ペイロードは必ず *Message* オブジェクトのリストになります。ペイロードにスカラーオブジェクト (文字列など) を格納したい場合は、かわりに *set_payload()* を使ってください。

This is a legacy method. On the `EmailMessage` class its functionality is replaced by *set_content()* and the related `make` and `add` methods.

get_payload(*i=None*, *decode=False*)

現在のペイロードへの参照を返します。これは *is_multipart()* が `True` の場合 *Message* オブ

ジェクトのリストになり、*is_multipart()* が False の場合は文字列になります。ペイロードがリストの場合、リストを変更することはそのメッセージのペイロードを変更することになります。

オプション引数の *i* がある場合、*is_multipart()* が True ならば *get_payload()* はペイロード中で 0 から数えて *i* 番目の要素を返します。*i* が 0 より小さい場合、あるいはペイロードの個数以上の場合は *IndexError* が発生します。ペイロードが文字列 (つまり *is_multipart()* が False) にもかかわらず *i* が与えられたときは *TypeError* が発生します。

Optional *decode* is a flag indicating whether the payload should be decoded or not, according to the *Content-Transfer-Encoding* header. When True and the message is not a multipart, the payload will be decoded if this header's value is quoted-printable or base64. If some other encoding is used, or *Content-Transfer-Encoding* header is missing, the payload is returned as-is (undecoded). In all cases the returned value is binary data. If the message is a multipart and the *decode* flag is True, then None is returned. If the payload is base64 and it was not perfectly formed (missing padding, characters outside the base64 alphabet), then an appropriate defect will be added to the message's defect property (InvalidBase64PaddingDefect or InvalidBase64CharactersDefect, respectively).

When *decode* is False (the default) the body is returned as a string without decoding the *Content-Transfer-Encoding*. However, for a *Content-Transfer-Encoding* of 8bit, an attempt is made to decode the original bytes using the charset specified by the *Content-Type* header, using the replace error handler. If no charset is specified, or if the charset given is not recognized by the email package, the body is decoded using the default ASCII charset.

This is a legacy method. On the EmailMessage class its functionality is replaced by *get_content()* and *iter_parts()*.

set_payload(*payload*, *charset=None*)

メッセージオブジェクト全体のペイロードを *payload* に設定します。クライアントはペイロードを変更してはいけません。オプションの *charset* はメッセージのデフォルト文字セットを設定します。詳しくは *set_charset()* を参照してください。

This is a legacy method. On the EmailMessage class its functionality is replaced by *set_content()*.

set_charset(*charset*)

ペイロードの文字セットを *charset* に変更します。これは *Charset* インスタンス (*email.charset* 参照)、文字セット名を表す文字列、あるいは None のいずれかです。文字列を指定した場合は *Charset* インスタンスに変換されます。*charset* が None の場合、charset 引数は *Content-Type* ヘッダから除去されます (それ以外にメッセージは変更されません)。これら以外のものを文字セットとして指定した場合、*TypeError* を送出します。

MIME-Version ヘッダが存在しなければ、追加されます。*Content-Type* ヘッダが存在しなければ、*text/plain* を値として追加されます。*Content-Type* が存在していてもいなくても、charset パラメタは *charset.output_charset* に設定されます。*charset.input_charset* と *charset.output_charset* が異なるなら、ペイロードは *output_charset* に再エンコードされます。*Content-Transfer-Encoding* ヘッダが存在しなければ、ペイロードは、必要なら

指定された *Charset* を使って transfer エンコードされ、適切な値のヘッダが追加されます。*Content-Transfer-Encoding* ヘッダがすでに存在すれば、ペイロードはすでにその *Content-Transfer-Encoding* によって正しくエンコードされたものと見なされ、変形されません。

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the *charset* parameter of the `email.emailmessage.EmailMessage.set_content()` method.

get_charset()
そのメッセージ中のペイロードの *Charset* インスタンスを返します。

This is a legacy method. On the `EmailMessage` class it always returns `None`.

以下のメソッドは、メッセージの **RFC 2822** ヘッダにアクセスするためのマップ (辞書) 形式のインタフェイスを実装したものです。これらのメソッドと、通常のマップ (辞書) 型はまったく同じ意味をもつわけではないことに注意してください。たとえば辞書型では、同じキーが複数あることは許されていませんが、ここでは同じメッセージヘッダが複数ある場合があります。また、辞書型では *keys()* で返されるキーの順序は保証されていませんが、*Message* オブジェクト内のヘッダはつねに元のメッセージ中に現れた順序、あるいはそのあとに追加された順序で返されます。削除され、その後ふたたび追加されたヘッダはリストの一番最後に現れます。

こういった意味のちがいは意図的なもので、最大の利便性をもつようにつくられています。

注意: どんな場合も、メッセージ中のエンベロープヘッダはこのマップ形式のインタフェイスには含まれません。

In a model generated from bytes, any header values that (in contravention of the RFCs) contain non-ASCII bytes will, when retrieved through this interface, be represented as *Header* objects with a charset of *unknown-8bit*.

__len__()
複製されたものもふくめてヘッダ数の合計を返します。

__contains__(*name*)
メッセージオブジェクトが *name* という名前のフィールドを持っていれば `True` を返します。この検査では名前の大文字小文字は区別されません。*name* は最後にコロンをふくんでいてはいけません。このメソッドは以下のように `in` 演算子で使われます:

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)
指定された名前のヘッダフィールドの値を返します。*name* は最後にコロンをふくんでいてはいけません。そのヘッダがない場合は `None` が返され、*KeyError* 例外は発生しません。

注意: 指定された名前のフィールドがメッセージのヘッダに 2 回以上現れている場合、どちらの値が返されるかは未定義です。ヘッダに存在するフィールドの値をすべて取り出したい場合は *get_all()* メソッドを使ってください。

__setitem__(*name*, *val*)
メッセージヘッダに *name* という名前の *val* という値をもつフィールドをあらたに追加します。このフィールドは現在メッセージに存在するフィールドのいちばん後に追加されます。

注意: このメソッドでは、すでに同一の名前で存在するフィールドは上書き **されません**。もしメッセージが名前 *name* をもつフィールドをひとつしか持たないようにしたければ、最初にそれを除去してください。たとえば:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

__delitem__(*name*)
メッセージのヘッダから、*name* という名前をもつフィールドをすべて除去します。たとえこの名前をもつヘッダが存在していなくても例外は発生しません。

keys()
メッセージ中にあるすべてのヘッダのフィールド名のリストを返します。

values()
メッセージ中にあるすべてのフィールドの値のリストを返します。

items()
メッセージ中にあるすべてのヘッダのフィールド名とその値を 2-タプルのリストとして返します。

get(*name*, *failobj=None*)
指定された名前をもつフィールドの値を返します。これは指定された名前がないときにオプション引数の *failobj* (デフォルトでは None) を返すことをのぞけば、*__getitem__()* と同じです。

さらに、役に立つメソッドをいくつか紹介します:

get_all(*name*, *failobj=None*)
name の名前をもつフィールドのすべての値からなるリストを返します。該当する名前のヘッダがメッセージ中に含まれていない場合は *failobj* (デフォルトでは None) が返されます。

add_header(*_name*, *_value*, ***_params*)
拡張ヘッダ設定。このメソッドは *__setitem__()* と似ていますが、追加のヘッダ・パラメータをキーワード引数で指定できるところが違っています。*_name* に追加するヘッダフィールドを、*_value* にそのヘッダの **最初の** 値を渡します。

キーワード引数辞書 *_params* の各項目ごとに、そのキーがパラメータ名として扱われ、キー名にふくまれるアンダースコアはハイフンに置換されます (アンダースコアは、Python 識別子としてハイフンを使えないための代替です)。ふつう、パラメータの値が None 以外のときは、`key="value"` の形で追加されます。パラメータの値が None のときはキーのみが追加されます。値が非 ASCII 文字を含むなら、それは `(CHARSET, LANGUAGE, VALUE)` の形式の 3 タプルにすることが出来ます。ここで CHARSET はその値をエンコードするのに使われる文字セットを指示する文字列で、LANGUAGE は通常 None か空文字列にでき (これら以外で指定出来るものについては **RFC 2231** を参照してください)、VALUE は非 ASCII コードポイントを含む文字列値です。この 3 タプルではなく非 ASCII 文字を含む値が渡された場合は、CHARSET に `utf-8` 、LANGUAGE に None を使って **RFC 2231** 形式に自動的にエンコードされます。

以下はこの使用例です:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

こうするとヘッダには以下のように追加されます

```
Content-Disposition: attachment; filename="bud.gif"
```

非 ASCII 文字を使った例:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

は、以下のようになります

```
Content-Disposition: attachment; filename*="iso-8859-1''Fu%DFballer.ppt"
```

replace_header(*_name*, *_value*)

ヘッダの置換。*_name* と一致するヘッダで最初に見つかったものを置き換えます。このときヘッダの順序とフィールド名の大文字小文字は保存されます。一致するヘッダがない場合、*KeyError* が発生します。

get_content_type()

そのメッセージの content-type を返します。返された文字列は強制的に小文字で *maintype/subtype* の形式に変換されます。メッセージ中に *Content-Type* ヘッダがない場合、デフォルトの content-type は *get_default_type()* が返す値によって与えられます。**RFC 2045** によればメッセージはつねにデフォルトの content-type をもっているので、*get_content_type()* はつねになんらかの値を返すはずです。

RFC 2045 はメッセージのデフォルト content-type を、それが *multipart/digest* コンテナに現れているとき以外は *text/plain* に規定しています。あるメッセージが *multipart/digest* コンテナ中にある場合、その content-type は *message/rfc822* になります。もし *Content-Type* ヘッダが適切でない content-type 書式だった場合、**RFC 2045** はそれのデフォルトを *text/plain* として扱うよう定めています。

get_content_maintype()

そのメッセージの主 content-type を返します。これは *get_content_type()* によって返される文字列の *maintype* 部分です。

get_content_subtype()

そのメッセージの副 content-type (sub content-type、subtype) を返します。これは *get_content_type()* によって返される文字列の *subtype* 部分です。

get_default_type()

デフォルトの content-type を返します。ほどんどのメッセージではデフォルトの content-type は *text/plain* ですが、メッセージが *multipart/digest* コンテナに含まれているときだけ例外的に *message/rfc822* になります。

set_default_type(*ctype*)
デフォルトの content-type を設定します。*ctype* は *text/plain* あるいは *message/rfc822* である必要がありますが、強制ではありません。デフォルトの content-type はヘッダの *Content-Type* には格納されません。

get_params(*failobj=None*, *header='content-type'*, *unquote=True*)
メッセージの *Content-Type* パラメータをリストとして返します。返されるリストはキー/値の組からなる 2 要素タプルが連なったものであり、これらは `'='` 記号で分離されています。`'='` の左側はキーになり、右側は値になります。パラメータ中に `'='` がなかった場合、値の部分は空文字列になり、そうでなければその値は *get_param()* で説明されている形式になります。また、オプション引数 *unquote* が `True` (デフォルト) である場合、この値は unquote されます。

オプション引数 *failobj* は、*Content-Type* ヘッダが存在しなかった場合に返すオブジェクトです。オプション引数 *header* には *Content-Type* のかわりに検索すべきヘッダを指定します。

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the *params* property of the individual header objects returned by the header access methods.

get_param(*param*, *failobj=None*, *header='content-type'*, *unquote=True*)
メッセージの *Content-Type* ヘッダ中のパラメータ *param* を文字列として返します。そのメッセージ中に *Content-Type* ヘッダが存在しなかった場合、*failobj* (デフォルトは `None`) が返されます。

オプション引数 *header* が与えられた場合、*Content-Type* のかわりにそのヘッダが使用されます。

パラメータのキー比較は常に大文字小文字を区別しません。返り値は文字列か 3 要素のタプルで、タプルになるのはパラメータが **RFC 2231** エンコードされている場合です。3 要素タプルの場合、各要素の値は (`CHARSET, LANGUAGE, VALUE`) の形式になっています。`CHARSET` と `LAGUAGE` は `None` になることがあり、その場合 `VALUE` は `us-ascii` 文字セットでエンコードされているとみなさなければならないので注意してください。普段は `LANGUAGE` を無視できます。

この関数を使うアプリケーションが、パラメータが **RFC 2231** 形式でエンコードされているかどうかを気にしないのであれば、*email.utils.collapse_rfc2231_value()* に *get_param()* の返り値を渡して呼び出すことで、このパラメータをひとつにまとめることができます。この値がタプルならばこの関数は適切にデコードされた Unicode 文字列を返し、そうでない場合は unquote された元の文字列を返します。たとえば:

```
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

いずれの場合もパラメータの値は (文字列であれ 3 要素タプルの `VALUE` 項目であれ) つねに unquote されます。ただし、*unquote* が `False` に指定されている場合は unquote されません。

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the *params* property of the individual header objects returned by the header access methods.

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)
Content-Type ヘッダ中のパラメータを設定します。指定されたパラメータがヘッダ中にすでに存

在する場合、その値は *value* に置き換えられます。*Content-Type* ヘッダがまだこのメッセージ中に存在していない場合、**RFC 2045** にしたがいこの値には *text/plain* が設定され、新しいパラメータ値が末尾に追加されます。

オプション引数 *header* が与えられた場合、*Content-Type* のかわりにそのヘッダが使用されます。オプション引数 *requote* が `False` でない限り、この値は quote されます (デフォルトは `True`)。

オプション引数 *charset* が与えられると、そのパラメータは **RFC 2231** に従ってエンコードされます。オプション引数 *language* は RFC 2231 の言語を指定しますが、デフォルトではこれは空文字列となります。*charset* と *language* はどちらも文字列である必要があります。

If *replace* is `False` (the default) the header is moved to the end of the list of headers. If *replace* is `True`, the header will be updated in place.

バージョン 3.4 で変更: `replace` キーワードが追加されました。

del_param(*param*, *header='content-type'*, *requote=True*)

指定されたパラメータを *Content-Type* ヘッダ中から完全にとりのぞきます。ヘッダはそのパラメータと値がない状態に書き換えられます。*requote* が `False` でない限り (デフォルトでは `True` です)、すべての値は必要に応じて quote されます。オプション変数 *header* が与えられた場合、*Content-Type* のかわりにそのヘッダが使用されます。

set_type(*type*, *header='Content-Type'*, *requote=True*)

Content-Type ヘッダの maintype と subtype を設定します。*type* は *maintype/subtype* という形の文字列でなければなりません。それ以外の場合は *ValueError* が発生します。

このメソッドは *Content-Type* ヘッダを置き換えますが、すべてのパラメータはそのままにします。*requote* が `False` の場合、これはすでに存在するヘッダを quote せず放置しますが、そうでない場合は自動的に quote します (デフォルト動作)。

オプション変数 *header* が与えられた場合、*Content-Type* のかわりにそのヘッダが使用されます。*Content-Type* ヘッダが設定される場合には、*MIME-Version* ヘッダも同時に付加されます。

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the `make_` and `add_` methods.

get_filename(*failobj=None*)

そのメッセージ中の *Content-Disposition* ヘッダにある、`filename` パラメータの値を返します。目的のヘッダに `filename` パラメータがない場合には *Content-Type* ヘッダにある `name` パラメータを探します。それも無い場合またはヘッダが無い場合には *failobj* が返されます。返される文字列はつねに *email.utils.unquote()* によって unquote されます。

get_boundary(*failobj=None*)

そのメッセージ中の *Content-Type* ヘッダにある、`boundary` パラメータの値を返します。目的のヘッダが欠けていたり、`boundary` パラメータがない場合には *failobj* が返されます。返される文字列はつねに *email.utils.unquote()* によって unquote されます。

set_boundary(*boundary*)

メッセージ中の *Content-Type* ヘッダにある、`boundary` パラメータに値を設定します。

set_boundary() は必要に応じて *boundary* を quote します。そのメッセージが *Content-Type* ヘッダを含んでいない場合、*HeaderParseError* が発生します。

注意: このメソッドを使うのは、古い *Content-Type* ヘッダを削除して新しい boundary をもったヘッダを *add_header()* で足すのとは少し違います。*set_boundary()* は一連のヘッダ中での *Content-Type* ヘッダの位置を保つからです。しかし、これは元の *Content-Type* ヘッダ中に存在していた連続する行の順番までは **保ちません** 。

get_content_charset(*failobj=None*)

そのメッセージ中の *Content-Type* ヘッダにある、charset パラメータの値を返します。値はすべて小文字に変換されます。メッセージ中に *Content-Type* がなかったり、このヘッダ中に charaset パラメータがない場合には *failobj* が返されます。

注意: これは *get_charset()* メソッドとは異なります。こちらのほうは文字列のかわりに、そのメッセージボディのデフォルトエンコーディングの *Charset* インスタンスを返します。

get_charsets(*failobj=None*)

メッセージ中に含まれる文字セットの名前をすべてリストにして返します。そのメッセージが *multipart* である場合、返されるリストの各要素がそれぞれの subpart のペイロードに対応します。それ以外の場合、これは長さ 1 のリストを返します。

リスト中の各要素は文字列であり、これは対応する subpart 中のそれぞれの *Content-Type* ヘッダにある charset の値です。しかし、その subpart が *Content-Type* をもってないか、charset がないか、あるいは MIME maintype が *text* でないいずれかの場合には、リストの要素として *failobj* が返されます。

get_content_disposition()

Return the lowercased value (without parameters) of the message's *Content-Disposition* header if it has one, or None. The possible values for this method are *inline*, *attachment* or None if the message follows **RFC 2183**.

バージョン 3.5 で追加.

walk()

walk() メソッドは多目的のジェネレータで、これはあるメッセージオブジェクトツリー中のすべての part および subpart をわたり歩くのに使えます。順序は深さ優先です。おそらく典型的な用法は、*walk()* を for ループ中でのイテレータとして使うことでしょう。ループを一回まわるごとに、次の subpart が返されるのです。

以下の例は、multipart メッセージのすべての part において、その MIME タイプを表示していくものです。:

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
```

(次のページに続く)

(前のページからの続き)

```
message/rfc822
text/plain
```

`walk` iterates over the subparts of any part where *is_multipart()* returns `True`, even though `msg.get_content_maintype() == 'multipart'` may return `False`. We can see this in our example by making use of the `_structure` debug helper function:

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

Here the `message` parts are not `multiparts`, but they do contain subparts. `is_multipart()` returns `True` and `walk` descends into the subparts.

Message オブジェクトはオプションとして 2 つのインスタンス属性をとることができます。これはある MIME メッセージからプレーンテキストを生成するのに使うことができます。

preamble

MIME ドキュメントの形式では、ヘッダ直後にくる空行と最初の multipart 境界をあらわす文字列のあいだにいくらかのテキスト (訳注: preamble, 序文) を埋めこむことを許しています。このテキストは標準的な MIME の範疇からはみ出しているので、MIME 形式を認識するメールソフトからこれらは通常まったく見えません。しかしメッセージのテキストを生で見る場合、あるいはメッセージを MIME 対応していないメールソフトで見る場合、このテキストは目に見えることになります。

preamble 属性は MIME ドキュメントに加えるこの最初の MIME 範囲外テキストを含んでいます。*Parser* があるテキストをヘッダ以降に発見したが、それはまだ最初の MIME 境界文字列が現れる前だった場合、パーザはそのテキストをメッセージの *preamble* 属性に格納します。*Generator* がある MIME メッセージからプレーンテキスト形式を生成するときメッセージが *preamble* 属性を持つことが発見されれば、これはそのテキストをヘッダと最初の MIME 境界の間に挿入します。詳細は *email.parser* および *email.generator* を参照してください。

注意: そのメッセージに preamble がない場合、*preamble* 属性には `None` が格納されます。

epilogue

epilogue 属性はメッセージの最後の MIME 境界文字列からメッセージ末尾までのテキストを含むもので、それ以外は *preamble* 属性と同じです。

Generator でファイル終端に改行を出力するため、epilogue に空文字列を設定する必要はなくなりました。

defects

defects 属性はメッセージを解析する途中で検出されたすべての問題点 (defect、障害) のリストを保持しています。解析中に発見されうる障害についてのより詳細な説明は *email.errors* を参照してください。

19.1.10 email.mime: メールと MIME オブジェクトを一から作成

ソースコード: Lib/email/mime/

This module is part of the legacy (Compat32) email API. Its functionality is partially replaced by the *contentmanager* in the new API, but in certain applications these classes may still be useful, even in non-legacy code.

ふつう、メッセージオブジェクト構造はファイルまたは何がしかのテキストをパーザに通すことで得られます。パーザは与えられたテキストを解析し、基底となる root のメッセージオブジェクトを返します。しかし、完全なメッセージオブジェクト構造を何もないところから作成することもまた可能です。個別の *Message* を手で作成することさえできます。実際には、すでに存在するメッセージオブジェクト構造をとってきて、そこに新たな *Message* オブジェクトを追加したり、あるものを別のところへ移動させたりできます。これは MIME メッセージを切ったりおろしたりするために非常に便利なインターフェイスを提供します。

新しいメッセージオブジェクト構造は *Message* インスタンスを作成することにより作れます。ここに添付ファイルやその他適切なものをすべて手で加えてやればよいのです。MIME メッセージの場合、*email* パッケージはこれらを簡単におこなえるようにするためにいくつかの便利なサブクラスを提供しています。

以下がそのサブクラスです:

class email.mime.base.**MIMEBase**(*_maintype*, *_subtype*, ***, *policy=compat32*, ***_params*)

モジュール: email.mime.base

これはすべての *Message* の MIME 用サブクラスの基底となるクラスです。とくに *MIMEBase* のインスタンスを直接作成することは (可能ではありますが) ふつうはしないでしょう。*MIMEBase* は単により特化された MIME 用サブクラスのための便宜的な基底クラスとして提供されています。

_maintype は ***Content-Type*** の主形式 (maintype) であり (***text*** や ***image*** など)、*_subtype* は ***Content-Type*** の副形式 (subtype) です (***plain*** や ***gif*** など)。*_params* は各パラメータのキーと値を格納した辞書であり、これは直接 *Message.add_header* に渡されます。

If *policy* is specified, (defaults to the *compat32* policy) it will be passed to *Message*.

MIMEBase クラスはつねに (*_maintype* 、*_subtype* 、および *_params* にもとづいた) ***Content-Type*** ヘッダと、***MIME-Version*** ヘッダ (必ず 1.0 に設定される) を追加します。

バージョン 3.6 で変更: キーワード専用引数 *policy* が追加されました。

class email.mime.nonmultipart.**MIMENonMultipart**

モジュール: email.mime.nonmultipart

MIMEBase のサブクラスで、これは ***multipart*** 形式でない MIME メッセージのための中間的な基底クラスです。このクラスのおもな目的は、通常 ***multipart*** 形式のメッセージに対してのみ意味をなす *attach()* メソッドの使用をふせぐことです。もし *attach()* メソッドが呼ばれた場合、これは *MultipartConversionError* 例外を発生します。

class email.mime.multipart.**MIMEMultipart**(*_subtype='mixed'*, *boundary=None*, *_subparts=None*, ***, *policy=compat32*, ***_params*)

モジュール: email.mime.multipart

MIMEBase のサブクラスで、これは ***multipart*** 形式の MIME メッセージのための中間的な基底クラスです。オプション引数 *_subtype* はデフォルトでは ***mixed*** になっていますが、そのメッセージの副形式 (subtype) を指定するのに使うことができます。メッセージオブジェクトには ***multipart/_subtype*** という値をもつ *Content-Type* ヘッダとともに、*MIME-Version* ヘッダが追加されるでしょう。

オプション引数 *boundary* は multipart の境界文字列です。これが None の場合 (デフォルト)、境界は必要に応じて計算されます（例えばメッセージがシリアライズされるときなど)。

_subparts はそのペイロードの subpart の初期値からなるシーケンスです。このシーケンスはリストに変換できるようになっている必要があります。新しい subpart はつねに *Message.attach* メソッドを使ってそのメッセージに追加できるようになっています。

Optional *policy* argument defaults to *compat32*.

Content-Type ヘッダに対する追加のパラメータはキーワード引数 *_params* を介して取得あるいは設定されます。これはキーワード辞書になっています。

バージョン 3.6 で変更: キーワード専用引数 *policy* が追加されました。

class email.mime.application.**MIMEApplication**(*_data*, *_subtype='octet-stream'*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

モジュール: email.mime.application

MIMENonMultipart のサブクラスである *MIMEApplication* クラスは MIME メッセージオブジェクトのメジャータイプ ***application*** を表します。*_data* は生のバイト列が入った文字列です。オプション引数 *_subtype* は MIME のサブタイプを設定します。サブタイプのデフォルトは ***octet-stream*** です。

オプション引数の *_encoder* は呼び出し可能なオブジェクト (関数など) で、データの転送に使う実際のエンコード処理を行います。この呼び出し可能なオブジェクトは引数を 1 つ取り、それは *MIMEApplication* のインスタンスです。ペイロードをエンコードされた形式に変更するために *get_payload()* と *set_payload()* を使い、必要に応じて *Content-Transfer-Encoding* やその他のヘッダをメッセージオブジェクトに追加するべきです。デフォルトのエンコードは base64 です。組み込みのエンコーダの一覧は *email.encoders* モジュールを見てください。

Optional *policy* argument defaults to *compat32*.

_params は基底クラスのコンストラクタにそのまま渡されます。

バージョン 3.6 で変更: キーワード専用引数 *policy* が追加されました。

class email.mime.audio.**MIMEAudio**(*_audiodata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

モジュール: email.mime.audio

MIMEAudio クラスは *MIMENonMultipart* のサブクラスで、主形式 (maintype) が ***audio*** の MIME オブジェクトを作成するのに使われます。*_audiodata* は実際の音声データを格納した文字列です。もしこのデータが標準の Python モジュール *sndhdr* によって認識できるものであれば、***Content-Type*** ヘッダの副形式 (subtype) は自動的に決定されます。そうでない場合はその画像の形式 (subtype) を *_subtype* で明示的に指定する必要があります。副形式が自動的に決定できず、*_subtype* の指定もない場合は、*TypeError* が発生します。

オプション引数の *_encoder* は呼び出し可能なオブジェクト (関数など) で、オーディオデータの転送に使う実際のエンコード処理を行います。この呼び出し可能なオブジェクトは引数を 1 つ取り、それは *MIMEAudio* のインスタンスです。ペイロードをエンコードされた形式に変更するために *get_payload()* と *set_payload()* を使い、必要に応じて ***Content-Transfer-Encoding*** やその他のヘッダをメッセージオブジェクトに追加するべきです。デフォルトのエンコードは base64 です。組み込みのエンコーダの一覧は *email.encoders* モジュールを見てください。

Optional *policy* argument defaults to *compat32*.

_params は基底クラスのコンストラクタにそのまま渡されます。

バージョン 3.6 で変更: キーワード専用引数 *policy* が追加されました。

class email.mime.image.**MIMEImage**(*_imagedata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

モジュール: email.mime.image

MIMEImage クラスは *MIMENonMultipart* のサブクラスで、主形式 (maintype) が ***image*** の MIME オブジェクトを作成するのに使われます。*_imagedata* は実際の画像データを格納した文字列です。もしこのデータが標準の Python モジュール *imghdr* によって認識できるものであれば、***Content-Type*** ヘッダの副形式 (subtype) は自動的に決定されます。そうでない場合はその画像の形式 (subtype) を *_subtype* で明示的に指定する必要があります。副形式が自動的に決定できず、*_subtype* の指定もない場合は、*TypeError* が発生します。

オプション引数の *_encoder* は呼び出し可能なオブジェクト (関数など) で、画像データの転送に使う実際のエンコード処理を行います。この呼び出し可能なオブジェクトは引数を 1 つ取り、それは *MIMEImage* のインスタンスです。ペイロードをエンコードされた形式に変更するために *get_payload()* と *set_payload()* を使い、必要に応じて ***Content-Transfer-Encoding*** やその他のヘッダをメッセージオブジェクトに追加するべきです。デフォルトのエンコードは base64 です。組み込みのエンコーダの一覧は *email.encoders* モジュールを見てください。

Optional *policy* argument defaults to *compat32*.

_params は *MIMEBase* コンストラクタに直接渡されます。

バージョン 3.6 で変更: キーワード専用引数 *policy* が追加されました。

class email.mime.message.MIMEMessage(*_msg*, *_subtype='rfc822'*, ***, *policy=compat32*)

モジュール: email.mime.message

MIMEMessage クラスは *MIMENonMultipart* のサブクラスで、主形式 (maintype) が ***message*** の MIME オブジェクトを作成するのに使われます。ペイロードとして使われるメッセージは *_msg* になります。これは *Message* クラス (あるいはそのサブクラス) のインスタンスでなければいけません。そうでない場合、この関数は *TypeError* を発生します。

オプション引数 *_subtype* はそのメッセージの副形式 (subtype) を設定します。デフォルトではこれは ***rfc822*** になっています。

Optional *policy* argument defaults to *compat32*.

バージョン 3.6 で変更: キーワード専用引数 *policy* が追加されました。

class email.mime.text.MIMEText(*_text*, *_subtype='plain'*, *_charset=None*, ***, *policy=compat32*)

モジュール: email.mime.text

MIMEText クラスは *MIMENonMultipart* のサブクラスで、主形式 (maintype) が ***text*** の MIME オブジェクトを作成するのに使われます。ペイロードの文字列は *_text* になります。*_subtype* には副形式 (subtype) を指定し、デフォルトは ***plain*** です。*_charset* はテキストの文字セットで、*MIMENonMultipart* コンストラクタに引数として渡されます。この値は、文字列が ascii コードポイントのみを含む場合 us-ascii 、それ以外は utf-8 がデフォルトになっています。*_charset* パラメータは、文字列と *Charset* インスタンスの両方を受け付けます。

_charset 引数に明示的に None をセットしない限りは、作成される MIMEText オブジェクトは charset の付いた ***Content-Type*** ヘッダと ***Content-Transfer-Endcoding*** ヘッダの両方を持ちます。これは後続の set_payload 呼び出しが、set_payload コマンドに charset が渡したとしてもエンコードされたペイロードにはならないことを意味します。Content-Transfer-Encoding ヘッダを削除することでこの振る舞いを「リセット」出来ます。これにより set_payload 呼び出しが新たなペイロードを自動的にエンコード (そして新たな ***Content-Transfer-Encoding*** ヘッダを追加) します。

Optional *policy* argument defaults to *compat32*.

バージョン 3.5 で変更: *_charset* は *Charset* インスタンスも受け取ります。

バージョン 3.6 で変更: キーワード専用引数 *policy* が追加されました。

19.1.11 email.header: 国際化されたヘッダ

ソースコード: Lib/email/header.py

このモジュールはレガシー (Compat32) な電子メール API の一部です。現在の API ではヘッダのエンコードとデコードは *EmailMessage* クラスの辞書的な API によって透過的に処理されます。レガシーコードでの使用に加えて、このモジュールは、ヘッダーをエンコードする際に使用される文字セットを完全に制御する必要があるアプリケーションで役立ちます

この節の以降のテキストはモジュールの元々のドキュメントです。

RFC 2822 は電子メールメッセージの形式を規定する基本規格です。これはほとんどの電子メールが ASCII 文字のみで構成されていたころ普及した **RFC 822** 標準から発展したものです。**RFC 2822** は電子メールがすべて 7-bit ASCII 文字のみから構成されていると仮定して作られた仕様です。

もちろん、電子メールが世界的に普及するにつれ、この仕様は国際化されてきました。今では電子メールに言語依存の文字集合を使うことができます。基本規格では、まだ電子メールメッセージを 7-bit ASCII 文字のみを使って転送するよう要求していますので、多くの RFC でどうやって非 ASCII の電子メールを **RFC 2822** 準拠な形式にエンコードするかが記述されています。これらの RFC は以下のものを含みます: **RFC 2045** 、**RFC 2046** 、**RFC 2047** 、および **RFC 2231** 。*email* パッケージは、*email.header* および *email.charset* モジュールでこれらの規格をサポートしています。

ご自分の電子メールヘッダ、たとえば *Subject* や *To* などのフィールドに非 ASCII 文字を入れたい場合、*Header* クラスを使う必要があります。*Message* オブジェクトの該当フィールドに文字列ではなく、*Header* インスタンスを使うのです。*Header* クラスは *email.header* モジュールからインポートしてください。たとえば:

```
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('p\xf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> msg.as_string()
'Subject: =?iso-8859-1?q?p=F6stal?=\n\n'
```

Subject フィールドに非 ASCII 文字を含めていることに注目してください。ここでは、含めたいバイト列がエンコードされている文字集合を指定して *Header* インスタンスを作成することによって実現しています。のちにこの *Message* インスタンスからフラットなテキストを生成する際に、この *Subject* フィールドは **RFC 2047** 準拠の適切な形式にエンコードされます。MIME 機能のあるメーラなら、このヘッダに埋めこまれた ISO-8859-1 文字を正しく表示するでしょう。

以下は *Header* クラスの説明です:

class email.header.**Header**(*s=None*, *charset=None*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*, *errors='strict'*)

別の文字集合の文字列を含む MIME 準拠なヘッダを作成します。

オプション引数 *s* はヘッダの値の初期値です。これが None の場合 (デフォルト)、ヘッダの初期値は設定されません。この値はあとから *append()* メソッドを呼びだすことによって追加することができます。*s* は *bytes* または *str* のインスタンスにできます。このセマンティクスについては *append()* の項を参照してください。

オプション引数 *charset* には 2 つの目的があります。ひとつは *append()* メソッドにおける *charset* 引数と同じものです。もうひとつは、これ以降 *charset* 引数を省略した *append()* メソッド呼び出しすべてにおける、デフォルト文字集合を決定するものです。コンストラクタに *charset* が与えられない場合 (デフォルト)、初期値の *s* および以後の *append()* 呼び出しにおける文字集合として us-ascii が使われます。

行の最大長は *maxlinelen* によって明示的に指定できます。最初の行を (*Subject* などの *s* に含まれないフィールドヘッダの責任をとるため) 短く切りとる場合、*header_name* にそのフィールド名を指定してください。*maxlinelen* のデフォルト値は 76 であり、*header_name* のデフォルト値は `None` です。これはその最初の行を長い、切りとられたヘッダとして扱わないことを意味します。

オプション引数 *continuation_ws* は **RFC 2822** 準拠の折り返し用余白文字で、ふつうこれは空白か、ハードタブ文字 (hard tab) である必要があります。ここで指定された文字は複数にわたる行の行頭に挿入されます。*continuation_ws* のデフォルト値は 1 つのスペース文字です。

オプション引数 *errors* は、*append()* メソッドにそのまま渡されます。

append(*s*, *charset=None*, *errors='strict'*)

この MIME ヘッダに文字列 *s* を追加します。

オプション引数 *charset* がもし与えられた場合、これは *Charset* インスタンス (*email.charset* を参照) か、あるいは文字集合の名前でなければなりません。この場合は *Charset* インスタンスに変換されます。この値が `None` の場合 (デフォルト)、コンストラクタで与えられた *charset* が使われます。

s は *bytes* または *str* のインスタンスです。*bytes* のインスタンスの場合、*charset* はその文字列のエンコーディングであり、この文字セットでデコードできないときは *UnicodeError* が発生します。

s が *str* のインスタンスの場合、*charset* はその文字列の文字セットを決定するためのヒントとして使われます。

いずれの場合でも、**RFC 2822** 準拠のヘッダを **RFC 2047** の規則を用いて生成する際、文字列は指定された文字セットの出力コーデックを用いてエンコードされます。出力コーデックを用いて文字列がエンコードできないときは *UnicodeError* が発生します。

オプションの *errors* は、*s* がバイト文字列だった場合のデコード呼び出しに errors 引数として渡されます。

encode(*splitchars=';, \t'*, *maxlinelen=None*, *linesep='\n'*)

メッセージヘッダを RFC に沿ったやり方でエンコードします。おそらく長い行は折り返され、非 ASCII 部分は base64 または quoted-printable エンコーディングで包含されるでしょう。

オプション引数 *splitchars* は、通常のヘッダーの折り返し処理の間に分割アルゴリズムによって特別な重みが与えられるべき文字を含む文字列です。これは、**RFC 2822** の 'higher level syntactic breaks' の非常に荒いサポートです: splitchar の後の分割点は、行分割において優先されます。分割文字は文字列中での出現順に優先されます。スペースとタブは、分割しようとする行に他の分割文字が出現しない時に、分割点として他の文字と比べてどのような優先順位が与えられるべきかを示すために、文字列に含めることができます。Splitchars は **RFC 2047** エンコードされた行には影響しません。

与えられた場合、*maxlinelen* はインスタンスの最大行長の値を上書きします。

linesep は、折り返しヘッダの行を区切る文字を指定します。デフォルトでは Python アプリケーションコードに最も有用な値 (`\n`) になりますが、RFC 準拠の行区切り文字でヘッダを生成するた

めに “rn“を指定することができます。

バージョン 3.2 で変更: *linesep* 引数が追加されました。

Header クラスは、標準の演算子や組み込み関数をサポートするためのメソッドもいくつか提供しています。

__str__()
Header の概要を文字列として返します。無制限の行長を使用します。すべての箇所は、指定されたエンコーディングを使用して Unicode に変換され、適切に結合されます。文字セット 'unknown-8bit' を持つ箇所は、'replace' エラーハンドラを使って ASCII としてデコードされます。

バージョン 3.2 で変更: 'unknown-8bit' 文字集合の処理が追加されました。

__eq__(*other*)
このメソッドは、ふたつの *Header* インスタンスどうしが等しいかどうか判定するのに使えます。

__ne__(*other*)
このメソッドは、ふたつの *Header* インスタンスどうしが異なっているかどうかを判定するのに使えます。

さらに、*email.header* モジュールは以下のような簡易関数も提供しています。

email.header.**decode_header**(*header*)
文字集合を変換せずにメッセージのヘッダをデコードします。ヘッダの値は *header* にあります。

この関数はヘッダのそれぞれのデコードされた部分ごとに、(decoded_string, charset) という形式の 2 要素タプルからなるリストを返します。*charset* はヘッダのエンコードされていない部分に対しては None を、それ以外の場合はエンコードされた文字列が指定している文字集合の名前を小文字からなる文字列で返します。

以下はこの使用例です:

```
>>> from email.header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[(b'p\xf6stal', 'iso-8859-1')]
```

email.header.**make_header**(*decoded_seq*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*)
decode_header() によって返される 2 要素タプルのリストから *Header* インスタンスを作成します。

decode_header() はヘッダの値をとってきて、(decoded_string, charset) という形式の 2 要素タプルからなるリストを返します。ここで *decoded_string* はデコードされた文字列、*charset* はその文字集合です。

この関数はこれらのリストの項目から、*Header* インスタンスを返します。オプション引数 *maxlinelen*、*header_name* および *continuation_ws* は *Header* コンストラクタに与えるものと同じです。

19.1.12 email.charset: 文字集合の表現

ソースコード: Lib/email/charset.py

このモジュールは、レガシーな (Compat32) email API の一部分です。新しい API では別名の表のみ使われています。

この節の以降のテキストはモジュールの元々のドキュメントです。

このモジュールは文字集合の表現および電子メールメッセージの文字集合の変換を行う *Charset* クラスに加え、文字集合のレジストリとそれを操作する簡易メソッドを提供しています。*Charset* インスタンスは *email* パッケージ中にある他のいくつかのモジュールで使用されます。

このクラスは *email.charset* モジュールからインポートしてください。

class email.charset.**Charset**(*input_charset=DEFAULT_CHARSET*)

文字集合を電子メールのプロパティに写像します。

このクラスは、ある特定の文字集合に対し電子メールに課される条件についての情報を提供します。また、適用可能なコーデックスが利用出来れば、文字集合間の変換を行う簡易ルーチンを提供します。文字集合について、この関数は電子メールメッセージ内での RFC に準拠した文字集合の使い方に関する情報を提供するのに最善を尽くします。

文字集合によっては、電子メールのヘッダや本体で使う場合に quoted-printable や base64 形式でエンコードされなければなりません。また、文字集合によっては完全に変換する必要があり、電子メールの中では使用できません。

以下ではオプション引数 *input_charset* について説明します。この値は常に小文字に強制的に変換されます。そして文字集合の別名が正規化されたあと、この値は文字集合のレジストリ内を検索し、ヘッダのエンコーディングとメッセージ本体のエンコーディング、および出力時の変換に使われるコーデックを見付けるのに使われます。たとえば *input_charset* が `iso-8859-1` の場合、ヘッダおよびメッセージ本体は quoted-printable でエンコードされ、出力時の変換用コーデックは必要ありません。もし *input_charset* が `euc-jp` ならば、ヘッダは base64 でエンコードされ、メッセージ本体はエンコードされませんが、出力されるテキストは `euc-jp` 文字集合から `iso-2022-jp` 文字集合に変換されます。

Charset インスタンスは以下のようなデータ属性をもっています:

input_charset

最初に指定される文字集合です。一般的な別名は、**正式な** 電子メール用の名前に変換されます (たとえば、`latin_1` は `iso-8859-1` に変換されます)。デフォルトは 7-bit の `us-ascii` です。

header_encoding

この文字集合が電子メールヘッダに使われる前にエンコードされなければならない場合、この属性は `Charset.QP` (quoted-printable エンコーディング)、`Charset.BASE64` (base64 エンコーディング)、あるいは最短の QP または BASE64 エンコーディングである `Charset.SHORTEST` に設定されます。そうでない場合、この値は `None` になります。

body_encoding

header_encoding と同じですが、この値はメッセージ本体のためのエンコーディングを記述

します。これはヘッダ用のエンコーディングとは違うかもしれません。*body_encoding* では、`Charset.SHORTEST` を使うことはできません。

output_charset
文字集合によっては、電子メールのヘッダあるいはメッセージ本体に使う前に変換しなければなりません。もし *input_charset* がそれらの文字集合のどれかをさしていたら、この *output_charset* 属性はそれが出力時に変換される文字集合の名前を表しています。それ以外の場合、この値は `None` になります。

input_codec
input_charset を Unicode に変換するための Python 用コーデック名です。変換用のコーデックが必要ないときは、この値は `None` になります。

output_codec
Unicode を *output_charset* に変換するための Python 用コーデック名です。変換用のコーデックが必要ないときは、この値は `None` になります。この属性は *input_codec* と同じ値を持つことになるでしょう。

Charset インスタンスは、以下のメソッドも持っています:

get_body_encoding()
メッセージ本体のエンコードに使われる content-transfer-encoding の値を返します。

この値は使用しているエンコーディングの文字列 `quoted-printable` または `base64` か、あるいは関数のいずれかです。後者の場合、これはエンコードされる Message オブジェクトを単一の引数として取るような関数である必要があります。この関数は変換後 *Content-Transfer-Encoding* ヘッダ自体を、なんであれ適切な値に設定する必要があります。

このメソッドは *body_encoding* が `QP` の場合 `quoted-printable` を返し、*body_encoding* が `BASE64` の場合 `base64` を返します。それ以外の場合は文字列 `7bit` を返します。

get_output_charset()
出力用の文字集合を返します。

これは *output_charset* 属性が `None` でなければその値になります。それ以外の場合、この値は *input_charset* と同じです。

header_encode(*string*)
文字列 *string* をヘッダ用にエンコードします。

エンコーディングの形式 (base64 または quoted-printable) は、*header_encoding* 属性に基づきます。

header_encode_lines(*string*, *maxlengths*)
string を最初にバイト列に変換し、ヘッダ用にエンコードします。

これは *header_encode()* と似ていますが、与えられた引数 *maxlengths* に従って、行の最大長に合うように文字列が調整されるところが異なります。*maxlengths* はイテレータでなければなりません: このイテレータが返す要素は次の行の最大長を表します。

body_encode(*string*)
文字列 *string* をメッセージ本体用にエンコードします。

エンコーディングの形式 (base64 または quoted-printable) は、*body_encoding* 属性に基づきます。

Charset クラスには、標準的な演算と組み込み関数をサポートするいくつかのメソッドがあります。

__str__()
input_charset を小文字に変換された文字列型として返します。**__repr__**() は、*__str__()* の別名となっています。

__eq__(*other*)
このメソッドは、2 つの *Charset* インスタンスが同じかどうかをチェックするのに使います。

__ne__(*other*)
このメソッドは、2 つの *Charset* インスタンスが異なるかどうかをチェックするのに使います。

また、*email.charset* モジュールには、グローバルな文字集合、文字集合の別名およびコーデック用のレジストリに新しいエントリを追加する以下の関数も含まれています:

email.charset.**add_charset**(*charset*, *header_enc=None*, *body_enc=None*, *output_charset=None*)
文字の属性をグローバルなレジストリに追加します。

charset は入力用の文字集合で、その文字集合の正式名称を指定する必要があります。

オプション引数 *header_enc* および *body_enc* は quoted-printable エンコーディングを表す Charset.QP か、base64 エンコーディングを表す Charset.BASE64、最短の quoted-printable または base64 エンコーディングを表す Charset.SHORTEST、あるいはエンコーディングなしの None のいずれかになります。SHORTEST が使えるのは *header_enc* だけです。デフォルトの値はエンコーディングなしの None になっています。

オプション引数 *output_charset* には出力用の文字集合が入ります。Charset.convert() が呼ばれたときの変換はまず入力用の文字集合を Unicode に変換し、それから出力用の文字集合に変換されます。デフォルトでは、出力は入力と同じ文字集合になっています。

input_charset および *output_charset* はこのモジュール中の文字集合-コーデック対応表にある Unicode コーデックエントリである必要があります。モジュールがまだ対応していないコーデックを追加するには、*add_codec()* を使ってください。より詳しい情報については *codecs* モジュールの文書を参照してください。

グローバルな文字集合用のレジストリは、モジュールのグローバル辞書 CHARSETS 内に保持されています。

email.charset.**add_alias**(*alias*, *canonical*)
文字集合の別名を追加します。*alias* はその別名で、たとえば latin-1 のように指定します。*canonical* はその文字集合の正式名称で、たとえば iso-8859-1 のように指定します。

文字集合のグローバルな別名用レジストリは、モジュールのグローバル辞書 ALIASES 内に保持されています。

email.charset.add_codec(*charset*, *codecname*)
: 与えられた文字集合の文字と Unicode との変換を行うコーデックを追加します。

  charset は文字集合の正式な名前です。*codecname* は、*str* の *encode()* メソッドの第 2 引数で使える Python コーデックの名前です。

19.1.13 email.encoders: エンコーダ

ソースコード: Lib/email/encoders.py

このモジュールは、レガシーな (Compat32) email API の一部です。新しい API では、この機能は *set_content()* メソッドの*cte*パラメータによって提供されます。

このモジュールは Python3 で非推奨になりました。*MIMEText* クラスはクラスのインスタンス化中に渡された *_subtype* と *_charset* の値を使って content type と CTE ヘッダを設定するので、ここで提供されている関数は明示的に呼び出すべきではありません。

この節の以降のテキストはモジュールの元々のドキュメントです。

何もないところから *Message* を作成するときしばしば必要になるのが、ペイロードをメールサーバに通すためにエンコードすることです。これはとくにバイナリデータを含んだ *image/** や *text/** タイプのメッセージで必要です。

email パッケージでは、encoders モジュールにおいていくかの便宜的なエンコーダをサポートしています。実際にはこれらのエンコーダは *MIMEAudio* および *MIMEImage* クラスのコンストラクタでデフォルトエンコーダとして使われています。すべてのエンコーディング関数は、エンコードするメッセージオブジェクトひとつだけを引数にとります。これらはふつうペイロードを取りだし、それをエンコードして、ペイロードをエンコードされたものにセットしなおします。これらはまた *Content-Transfer-Encoding* ヘッダを適切な値に設定します。

マルチパートメッセージにこれら関数を使うことは全く無意味です。それらは各々のサブパートごとに適用されるべきものです。メッセージがマルチパートのものを渡すと *TypeError* が発生します。

提供されているエンコーディング関数は以下のとおりです:

email.encoders.encode_quopri(*msg*)
: ペイロードを quoted-printable 形式にエンコードし、*Content-Transfer-Encoding* ヘッダを quoted-printable[*1] に設定します。これはそのペイロードのほとんどが通常の印刷可能な文字からなっているが、印刷不可能な文字がすこしだけあるときのエンコード方法として適しています。

email.encoders.encode_base64(*msg*)
: ペイロードを base64 形式でエンコードし、*Content-Transfer-Encoding* ヘッダを base64 に変更します。これはペイロード中のデータのほとんどが印刷不可能な文字である場合に適しています。quoted-printable 形式よりも結果としてはコンパクトなサイズになるからです。base64 形式の欠点は、これが人間にはまったく読めないテキストになってしまうことです。

[*1] 注意: *encode_quopri()* を使ってエンコードすると、データ中のタブ文字や空白文字もエンコードされます。

email.encoders.**encode_7or8bit**(*msg*)
これは実際にはペイロードを変更はしませんが、ペイロードの形式に応じて *Content-Transfer-Encoding* ヘッダを 7bit あるいは 8bit に適した形に設定します。

email.encoders.**encode_noop**(*msg*)
これは何もしないエンコーダです。*Content-Transfer-Encoding* ヘッダを設定さえしません。

脚注

19.1.14 email.utils: 多方面のユーティリティ

ソースコード: Lib/email/utils.py

---

email.utils モジュールではいくつかの便利なユーティリティを提供しています:

email.utils.**localtime**(*dt=None*)
Return local time as an aware datetime object. If called without arguments, return current time. Otherwise *dt* argument should be a *datetime* instance, and it is converted to the local time zone according to the system time zone database. If *dt* is naive (that is, `dt.tzinfo` is `None`), it is assumed to be in local time. In this case, a positive or zero value for *isdst* causes `localtime` to presume initially that summer time (for example, Daylight Saving Time) is or is not (respectively) in effect for the specified time. A negative value for *isdst* causes the `localtime` to attempt to divine whether summer time is in effect for the specified time.

バージョン 3.3 で追加.

email.utils.**make_msgid**(*idstring=None*, *domain=None*)
Returns a string suitable for an **RFC 2822**-compliant *Message-ID* header. Optional *idstring* if given, is a string used to strengthen the uniqueness of the message id. Optional *domain* if given provides the portion of the msgid after the '@'. The default is the local hostname. It is not normally necessary to override this default, but may be useful certain cases, such as a constructing distributed system that uses a consistent domain name across multiple hosts.

バージョン 3.2 で変更: *domain* キーワードが追加されました。

The remaining functions are part of the legacy (`Compat32`) email API. There is no need to directly use these with the new API, since the parsing and formatting they provide is done automatically by the header parsing machinery of the new API.

email.utils.**quote**(*str*)
文字列 *str* 内のバックスラッシュをバックスラッシュ 2 つに置換した新しい文字列を返します。また、ダブルクォートはバックスラッシュ + ダブルクォートに置換されます。

email.utils.**unquote**(*str*)
文字列 *str* の **クォートを外した** 新しい文字列を返します。*str* の先頭と末尾がダブルクォートだった場合、それらは取り除かれます。同様に *str* の先頭と末尾が角ブラケット (<、>) だった場合もそれらは取り除かれます。

email.utils.**parseaddr**(*address*)

To や *Cc* のようなフィールドを持つアドレスをパースして、構成要素の **実名** と **電子メールアドレス** を取り出します。パースに成功した場合、これらの情報持つタプルを返します。失敗した場合は 2 要素のタプル `('', '')` を返します。

email.utils.**formataddr**(*pair*, *charset='utf-8'*)

parseaddr() の逆で、2 要素のタプル `(realname, email_address)` を取って *To* や *Cc* ヘッダに適した文字列を返します。タプル *pair* の第 1 要素が偽である場合、第 2 要素の値をそのまま返します。

任意の *charset* は、`realname` が非 ASCII 文字を含んでいる場合にその **RFC 2047** エンコーディングに使われる文字集合です。*str* か *Charset* のインスタンスで、デフォルトは `utf-8` です。

バージョン 3.3 で変更: *charset* オプションが追加されました。

email.utils.**getaddresses**(*fieldvalues*)

このメソッドは `parseaddr()` が返す形式の 2 要素タプルのリストを返します。*fieldvalues* は *Message.get_all* が返すような一連のヘッダフィールドです。以下はメッセージの全ての受信者を得る簡単な例です:

```
from email.utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

email.utils.**parsedate**(*date*)

RFC 2822 の規則に基づいて日付の解析を試みます。しかしながらメイラーによっては指定された形式に従っていないものがあるので、その場合 *parsedate()* は正しく推測しようとします。*date* は `"Mon, 20 Nov 1995 19:12:08 -0500"` のような **RFC 2822** 形式の日付を含む文字列です。日付の解析に成功した場合、*parsedate()* は関数 *time.mktime()* に直接渡せる形式の 9 要素からなるタプルを返します。失敗した場合は `None` を返します。返されるタプルの 6、7、8 番目の添字は使用不可なので注意してください。

email.utils.**parsedate_tz**(*date*)

Performs the same function as *parsedate()*, but returns either `None` or a 10-tuple; the first 9 elements make up a tuple that can be passed directly to *time.mktime()*, and the tenth is the offset of the date's timezone from UTC (which is the official term for Greenwich Mean Time)[*1]. If the input string has no timezone, the last element of the tuple returned is `0`, which represents UTC. Note that indexes 6, 7, and 8 of the result tuple are not usable.

email.utils.**parsedate_to_datetime**(*date*)

format_datetime() の逆です。*parsedate()* と同様に機能しますが、成功時に *datetime* を返します。入力文字列がタイムゾーン `-0000` を持つ場合、`datetime` はナイーブな `datetime` です。日付が RFC に従っている場合、`datetime` は UTC での時間を表しますが、日付の元になったメッセージの

[*1] 注意: この時間帯のオフセット値は `time.timezone` の値と符号が逆です。これは `time.timezone` が POSIX 標準に準拠しているのに対して、こちらは **RFC 2822** に準拠しているからです。

実際のタイムゾーンについての情報を持ちません。入力データにその他の有効なタイムゾーンの差がある場合、datetime は対応する *timezone tzinfo* のあるそつのない datetime です。

バージョン 3.3 で追加.

email.utils.**mktime_tz**(*tuple*)

parsedate_tz() が返す 10 要素のタプルを UTC のタイムスタンプ (エポックからの秒数) に変換します。与えられた時間帯が None である場合、時間帯として現地時間 (localtime) が仮定されます。

email.utils.**formatdate**(*timeval=None*, *localtime=False*, *usegmt=False*)

日付を **RFC 2822** 形式の文字列で返します。例:

```
Fri, 09 Nov 2001 01:08:47 -0000
```

与えられた場合、オプションの *timeval* は *time.gmtime()* や *time.localtime()* に渡すことの出来る浮動小数の時刻です。それ以外の場合、現在時刻が使われます。

オプション引数 *localtime* はフラグです。True の場合、この関数は *timeval* を解析して UTC の代わりに現地のタイムゾーンに対する日付を返します。おそらく夏時間も考慮するでしょう。デフォルトは False で、UTC が使われます。

オプション引数 *usegmt* はフラグです。True の場合、この関数はタイムゾーンを数値の -0000 ではなく ascii 文字列 GMT として日付を出力します。これはプロトコルによっては (例えば HTTP) 必要です。これは *localtime* が False のときのみ適用されます。デフォルトは False です。

email.utils.**format_datetime**(*dt*, *usegmt=False*)

Like formatdate, but the input is a *datetime* instance. If it is a naive datetime, it is assumed to be "UTC with no information about the source timezone", and the conventional -0000 is used for the timezone. If it is an aware datetime, then the numeric timezone offset is used. If it is an aware timezone with offset zero, then *usegmt* may be set to True, in which case the string GMT is used instead of the numeric timezone offset. This provides a way to generate standards conformant HTTP date headers.

バージョン 3.3 で追加.

email.utils.**decode_rfc2231**(*s*)

RFC 2231 に従って文字列 *s* をデコードします。

email.utils.**encode_rfc2231**(*s*, *charset=None*, *language=None*)

RFC 2231 に従って *s* をエンコードします。オプション引数 *charset* および *language* が与えられた場合、これらは文字セット名と言語名として使われます。もしこれらのどちらも与えられていない場合、*s* はそのまま返されます。*charset* は与えられているが *language* が与えられていない場合、文字列 *s* は *language* の空文字列を使ってエンコードされます。

email.utils.**collapse_rfc2231_value**(*value*, *errors='replace'*, *fallback_charset='us-ascii'*)

ヘッダのパラメータが **RFC 2231** 形式でエンコードされている場合、*Message.get_param* は 3 要素からなるタプルを返すことがあります。ここには、そのパラメータの文字セット、言語、および値の順に格納されています。*collapse_rfc2231_value()* はこのパラメータをひとつの Unicode 文字列にまとめます。オプション引数 *errors* は *str* の *encode()* メソッドの引数 *errors* に渡されます。

このデフォルト値は 'replace' となっています。オプション引数 *fallback_charset* は、もし **RFC 2231** ヘッダの使用している文字セットが Python の知っているものではなかった場合の非常用文字セットとして使われます。デフォルトでは、この値は 'us-ascii' です。

便宜上、*collapse_rfc2231_value()* に渡された引数 *value* がタプルでない場合には、これは文字列でなければなりません。その場合にはクォートを除いた文字列を返します。

email.utils.**decode_params**(*params*)

RFC 2231 に従って引数のリストをデコードします。*params* は (content-type, string-value) のような形式の 2 要素タプルです。

脚注

19.1.15 email.iterators: イテレータ

ソースコード: Lib/email/iterators.py

Message.walk メソッドを使うと、簡単にメッセージオブジェクトツリー内を次から次へとたどる (iteration) ことができます。*email.iterators* モジュールはこのための高水準イテレータをいくつか提供します。

email.iterators.**body_line_iterator**(*msg*, *decode=False*)

このイテレータは *msg* 中のすべてのサブパートに含まれるペイロードをすべて順にたどっていき、ペイロード内の文字列を 1 行ずつ返します。サブパートのヘッダはすべて無視され、Python 文字列でないペイロードからなるサブパートも無視されます。これは *readline()* を使って、ファイルからメッセージを (ヘッダだけとばして) フラットなテキストとして読むのにいくぶん似ているかもしれません。

オプション引数 *decode* は、*Message.get_payload* にそのまま渡されます。

email.iterators.**typed_subpart_iterator**(*msg*, *maintype='text'*, *subtype=None*)

このイテレータは *msg* 中のすべてのサブパートをたどり、それらの中で指定された MIME 形式 *maintype* と *subtype* をもつようなパートのみを返します。

subtype は省略可能であることに注意してください。これが省略された場合、サブパートの MIME 形式は maintype のみがチェックされます。じつは *maintype* も省略可能で、その場合にはデフォルトは *text* です。

つまり、デフォルトでは *typed_subpart_iterator()* は MIME 形式 *text/** をもつサブパートを順に返していくというわけです。

以下の関数は役に立つデバッグ用ツールとして追加されたもので、パッケージとして公式なサポートのあるインターフェイスでは **ありません**。

email.iterators.**_structure**(*msg*, *fp=None*, *level=0*, *include_default=False*)

そのメッセージオブジェクト構造の content-type をインデントつきで表示します。例えば:

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
```

(次のページに続く)

(前のページからの続き)

```
multipart/mixed
    text/plain
    text/plain
    multipart/digest
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
    text/plain
```

オプション引数 *fp* は出力を渡すための file-like オブジェクトです。これは Python の *print()* 関数が対応できるようになっている必要があります。*level* は内部的に使用されます。*include_default* が真の場合、デフォルトの型も出力します。

参考:

smtplib **モジュール** SMTP（簡易メール転送プロトコル）クライアント

poplib **モジュール** POP (Post Office Protocol) クライアント

モジュール *imaplib* IMAP (Internet Message Access Protocol) クライアント

nntplib **モジュール** NNTP (Net News Transport Protocol) クライアント

mailbox **モジュール** Tools for creating, reading, and managing collections of messages on disk using a variety standard formats.

smtpd **モジュール** SMTP server framework (primarily useful for testing)

19.2 json --- JSON エンコーダおよびデコーダ

ソースコード: Lib/json/___init___.py

JSON (JavaScript Object Notation) は、**RFC 7159** (**RFC 4627** を obsolete) と ECMA-404 によって定義された軽量のデータ交換用のフォーマットです。JavaScript のオブジェクトリテラル記法に由来していますが (JavaScript の厳密なサブセットではありませんが[*1])。

[*1] RFC 7159 正誤表 で述べられている通り、JSON は (ECMAScript Edition 5.1 の) JavaScript とは逆に、U+2028 (LINE SEPARATOR) と U+2029 (PARAGRAPH SEPARATOR) が文字列内に含まれることを許容しています。

警告: Be cautious when parsing JSON data from untrusted sources. A malicious JSON string may cause the decoder to consume considerable CPU and memory resources. Limiting the size of data to be parsed is recommended.

json の API は標準ライブラリの *marshal* や *pickle* のユーザに馴染み深いものです。

基本的な Python オブジェクト階層のエンコーディング:

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps('\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

コンパクトなエンコーディング:

```
>>> import json
>>> json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
```

見やすい表示:

```
>>> import json
>>> print(json.dumps({'4': 5, '6': 7}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

JSON のデコーディング:

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
'"foo\x08ar'
>>> from io import StringIO
>>> io = StringIO('["streaming API"]')
```

(次のページに続く)

(前のページからの続き)

```
>>> json.load(io)
['streaming API']
```

JSON オブジェクトのデコーディング方法を誂える:

```
>>> import json
>>> def as_complex(dct):
...     if '__complex__' in dct:
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...     object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads('1.1', parse_float=decimal.Decimal)
Decimal('1.1')
```

JSONEncoder の拡張:

```
>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         # Let the base class default method raise the TypeError
...         return json.JSONEncoder.default(self, obj)
...
>>> json.dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[2.0', ', 1.0', ']']
```

シェルから *json.tool* を使って妥当性チェックをして見やすく表示:

```
$ echo '{"json":"obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

詳細については **コマンドラインインターフェイス** を参照してください。

注釈: JSON は YAML 1.2 のサブセットです。このモジュールのデフォルト設定 (特に、デフォルトの **セパレータ** 値) で生成される JSON は YAML 1.0 および 1.1 のサブセットでもあります。このモジュールは

YAML シリアライザとしても使えます。

注釈: このモジュールのエンコーダとデコーダは、デフォルトで入力順と出力順を保つようになっています。根底のコンテナに順序がない場合のみ、順序が失われます。

Python3.7 以前では、*dict* は順序が保証されておらず、*collections.OrderedDict* が指定された場合以外は、出力順や入力順が保たれていませんでした。Python3.7 からは、標準の *dict* の挿入順序が保証されるため、JSON の生成と解析にあたって *collections.OrderedDict* を指定する必要がなくなりました。

19.2.1 基本的な使い方

json.dump(*obj*, *fp*, ***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

この **変換表** を使って、*obj* を JSON 形式の *fp* (`.write()` がサポートされている *file-like object*) へのストリームとして直列化します。

skipkeys が true (デフォルトは `False`) ならば、基本型 (*str*, *int*, *float*, *bool*, `None`) 以外の辞書のキーは *TypeError* を送出せずに読み飛ばされます。

この *json* モジュールは常に、*bytes* オブジェクトではなく、*str* オブジェクトを生成します。従って、`fp.write()` は *str* の入力をサポートしていなければなりません。

ensure_ascii が (デフォルト値の) true の場合、出力では入力された全ての非 ASCII 文字はエスケープされていることが保証されています。*ensure_ascii* が false の場合、これらの文字はそのまま出力されます。

check_circular が false (デフォルトは `True`) ならば、コンテナ型の循環参照チェックが省かれ、循環参照があれば *OverflowError* (またはもっと悪い結果) に終わります。

allow_nan が偽 (デフォルトは `True`) の場合、許容範囲外の *float* 値 (`nan`, `inf`, `-inf`) を JSON 仕様を厳格に守って 直列化すると、*ValueError* になります。*allow_non* が真の場合は、JavaScript での等価なもの (`NaN`, `Infinity`, `-Infinity`) が使われます。

indent が非負の整数または文字列であれば、JSON の配列要素とオブジェクトメンバはそのインデントレベルで見やすく表示されます。インデントレベルが 0 か負数または `""` であれば 改行だけが挿入されます。`None` (デフォルト) では最もコンパクトな表現が選択されます。正の数の indent はレベル毎に、指定した数のスペースでインデントします。もし *indent* が文字列 (`"\t"` のような) であれば、その文字列が個々のレベルのインデントに使用されます。

バージョン 3.2 で変更: 整数に加えて、文字列が *indent* に使用できるようになりました。

separators はもし指定するなら `(item_separator, key_separator)` というタプルでなければなりません。デフォルトは *indent* が `None` のとき `(', ', ': ')` で、そうでなければ `(',', ': ')` です。最もコンパクトな JSON の表現を得たければ空白を削った `(',', ':')` を指定すればいいでしょう。

バージョン 3.4 で変更: *indent* が None でなければ (',', ': ') がデフォルトで使われます。

default を指定する場合は関数を指定して、この関数はそれ以外では直列化できないオブジェクトに対して呼び出されます。その関数は、オブジェクトを JSON でエンコードできるバージョンにして返すか、さもなければ *TypeError* を送出しなければなりません。指定しない場合は、*TypeError* が送出されます。

sort_keys が true (デフォルトでは False です) であれば、辞書の出力がキーでソートされます。

カスタマイズされた *JSONEncoder* のサブクラス (たとえば追加の型を直列化するように default() メソッドをオーバーライドしたもの) を使うには、*cls* キーワード引数に指定します; 指定しなければ *JSONEncoder* が使われます。

バージョン 3.6 で変更: すべてのオプション引数は、**キーワード専用** になりました。

注釈: *pickle* や *marshal* とは異なり JSON はフレーム付きのプロトコルではないので、同じ *fp* に対し繰り返し *dump()* を呼び、複数のオブジェクトを直列化しようとすると、不正な JSON ファイルが作られてしまいます。

---

json.**dumps**(*obj*, ***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

この **変換表** を使って、*obj* を JSON 形式の *str* オブジェクトに直列化します。引数は *dump()* と同じ意味です。

注釈: JSON のキー値ペアのキーは、常に *str* 型です。辞書が JSON に変換されるとき、辞書の全てのキーは文字列へ強制的に変換が行われます。この結果として、辞書が JSON に変換され、それから辞書に戻された場合、辞書は元のものと同じではありません。つまり文字列ではないキーを持っている場合、loads(dumps(x)) != x となるということです。

---

json.**load**(*fp*, ***, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

この **変換表** を使い、*fp* (.read() をサポートし JSON ドキュメントを含んでいる *text file* もしくは *binary file*) を Python オブジェクトへ脱直列化します。

object_hook はオプションの関数で、任意のオブジェクトリテラルがデコードされた結果 (*dict*) に対し呼び出されます。*object_hook* の返り値は *dict* の代わりに使われます。この機能は独自のデコーダ (例えば JSON-RPC クラスヒンティング) を実装するのに使えます。

object_pairs_hook はオプションで渡す関数で、ペアの順序付きリストのデコード結果に対して呼ばれます。*object_pairs_hook* の返り値は *dict* の代わりに使われます。この機能は独自のデコーダを実装するのに使えます。*object_hook* も定義されている場合は、*object_pairs_hook* が優先して使用されます。

バージョン 3.1 で変更: *object_pairs_hook* のサポートが追加されました。

parse_float は、もし指定されれば、全てのデコードされる JSON の浮動小数点数文字列に対して呼

ばれます。デフォルトでは、`float(num_str)` と等価です。これは JSON 浮動小数点数に対して他のデータ型やパーサ (たとえば *decimal.Decimal*) を使うのに使えます。

parse_int は、もし指定されれば、全てのデコードされる JSON の整数文字列に対して呼ばれます。デフォルトでは、`int(num_str)` と等価です。これは JSON 整数に対して他のデータ型やパーサ (たとえば *float*) を使うのに使えます。

バージョン 3.8.14 で変更: The default *parse_int* of *int()* now limits the maximum length of the integer string via the interpreter's *integer string conversion length limitation* to help avoid denial of service attacks.

parse_constant は、もし指定されれば、次の文字列に対して呼ばれます: `'-Infinity'`, `'Infinity'`, `'NaN'`, `'null'`, `'true'`, `'false'`。これは不正な JSON 数値に遭遇したときに例外を送出するのに使えます。

バージョン 3.1 で変更: 'null', 'true', 'false' に対して *parse_constant* は呼びされません。

カスタマイズされた *JSONDecoder* のサブクラスを使うには、*cls* キーワード引数に指定します; 指定しなかった場合は *JSONDecoder* が使われます。追加のキーワード引数はこのクラスのコンストラクタに引き渡されます。

脱直列化しようとしているデータが不正な JSON ドキュメントだった場合、*JSONDecodeError* が送出されます。

バージョン 3.6 で変更: すべてのオプション引数は、**キーワード専用** になりました。

バージョン 3.6 で変更: *fp* には term:*binary file* 型も使えるようになりました。入力のエンコーディングは UTF-8, UTF-16, UTF-32 のいずれかでなければなりません。

`json.loads`(*s*, ***, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

この **変換表** を使い、*s* (JSON ドキュメントを含んでいる *str*, *bytes*, *bytearray* のいずれかのインスタンス) を Python オブジェクトへ脱直列化します。

Python3.1 から無視される非推奨の引数 *encoding* を除いて、その他の引数は *load()* のものと同じ意味です。

脱直列化しようとしているデータが不正な JSON ドキュメントだった場合、*JSONDecodeError* が送出されます。

Deprecated since version 3.1, will be removed in version 3.9: *encoding* キーワード引数

バージョン 3.6 で変更: *s* には *bytes* 型と *bytearray* 型も使えるようになりました。入力エンコーディングは UTF-8, UTF-16, UTF-32 のいずれかでなければなりません。

19.2.2 エンコーダとデコーダ

class json.JSONDecoder(*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)

単純な JSON デコーダ。

デフォルトではデコーディングの際、以下の変換を行います:

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	浮動小数点数
true	True
false	False
null	None

また、このデコーダは `NaN`, `Infinity`, `-Infinity` を対応する `float` の値として、JSON の仕様からは外れますが、理解します。

object_hook は、もし指定されれば、全てのデコードされた JSON オブジェクトに対して呼ばれその返値は与えられた *dict* の代わりに使われます。この機能は独自の脱直列化 (たとえば JSON-RPC クラスヒンティングをサポートするような) を提供するのに使えます。

object_pairs_hook が指定された場合、ペアの順序付きリストのデコード結果に対して毎回呼ばれます。*object_pairs_hook* の返り値は *dict* の代わりに使われます。この機能は独自のデコーダを実装するのに使えます。*object_hook* も定義されている場合は、*object_pairs_hook* が優先して使用されます。

バージョン 3.1 で変更: *object_pairs_hook* のサポートが追加されました。

parse_float は、もし指定されれば、全てのデコードされる JSON の浮動小数点数文字列に対して呼ばれます。デフォルトでは、`float(num_str)` と等価です。これは JSON 浮動小数点数に対して他のデータ型やパーサ (たとえば *decimal.Decimal*) を使うのに使えます。

parse_int は、もし指定されれば、全てのデコードされる JSON の整数文字列に対して呼ばれます。デフォルトでは、`int(num_str)` と等価です。これは JSON 整数に対して他のデータ型やパーサ (たとえば *float*) を使うのに使えます。

parse_constant は、もし指定されれば、次の文字列に対して呼ばれます: `'-Infinity'`, `'Infinity'`, `'NaN'`, `'null'`, `'true'`, `'false'`。これは不正な JSON 数値に遭遇したときに例外を送出するのに使えます。

strict が false (デフォルトは `True`) の場合、制御文字を文字列に含めることができます。ここで言う制御文字とは、`'\t'` (タブ)、`'\n'`、`'\r'`、`'\0'` を含む 0-31 の範囲のコードを持つ文字のことです。

脱直列化しようとしているデータが不正な JSON ドキュメントだった場合、*JSONDecodeError* が送出されます。

バージョン 3.6 で変更: すべての引数は、キーワード専用 になりました。

decode(*s*)

s (*str* インスタンスで JSON 文書を含むもの) の Python 表現を返します。

不正な JSON ドキュメントが与えられた場合、*JSONDecodeError* が送出されます。

raw_decode(*s*)

s (*str* インスタンスで JSON 文書で始まるもの) から JSON 文書をデコードし、Python 表現と *s* の文書の終わるところのインデックスからなる 2 要素のタプルを返します。

このメソッドは後ろに余分なデータを従えた文字列から JSON 文書をデコードするのに使えます。

class json.JSONEncoder(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

Python データ構造に対する拡張可能な JSON エンコーダ。

デフォルトでは以下のオブジェクトと型をサポートします:

Python	JSON
dict	object
list, tuple	array
str	string
int、float と int や float の派生列挙型	number
True	true
False	false
None	null

バージョン 3.4 で変更: int と float の派生列挙型クラスの対応が追加されました。

このクラスを拡張して他のオブジェクトも認識するようにするには、サブクラスを作って *default()* メソッドを次のように実装します。もう一つ別のメソッドでオブジェクト `o` に対する直列化可能なオブジェクトを返すものを呼び出すようにします。変換できない時はスーパークラスの実装を (*TypeError* を送出させるために) 呼ばなければなりません。

skipkeys が false (デフォルト) ならば、*str*, *int*, *float*, `None` 以外のキーをエンコードする試みは *TypeError* に終わります。*skipkeys* が true の場合は、それらのアイテムは単に読み飛ばされます。

ensure_ascii が (デフォルト値の) true の場合、出力では入力された全ての非 ASCII 文字はエスケープされていることが保証されています。*ensure_ascii* が false の場合、これらの文字はそのまま出力されます。

check_circular が true (デフォルト) ならば、リスト、辞書および自作でエンコードしたオブジェクトは循環参照がないかエンコード中にチェックされ、無限再帰 (これは *OverflowError* を引き起こします) を防止します。`True` でない場合は、そういったチェックは施されません。

allow_nan が true (デフォルト) ならば、`NaN`, `Infinity`, `-Infinity` はそのままエンコードされます。この振る舞いは JSON 仕様に従っていませんが、大半の JavaScript ベースのエンコーダ、デコー

ダと矛盾しません。True でない場合は、そのような浮動小数点数をエンコードすると *ValueError* が送出されます。

sort_keys が true (デフォルトは False) ならば、辞書の出力がキーでソートされます。これは JSON の直列化がいつでも比較できるようになるので回帰試験の際に便利です。

indent が非負の整数または文字列であれば、JSON の配列要素とオブジェクトメンバはそのインデントレベルで見やすく表示されます。インデントレベルが 0 か負数または "" であれば 改行だけが挿入されます。None (デフォルト) では最もコンパクトな表現が選択されます。正の数の indent はレベル毎に、指定した数のスペースでインデントします。もし *indent* が文字列 ("\t" のような) であれば、その文字列が個々のレベルのインデントに使用されます。

バージョン 3.2 で変更: 整数に加えて、文字列が *indent* に使用できるようになりました。

separators はもし指定するなら (item_separator, key_separator) というタプルでなければなりません。デフォルトは *indent* が None のとき (', ', ': ') で、そうでなければ (',', ': ') です。最もコンパクトな JSON の表現を得たければ空白を削った (',', ':') を指定すればいいでしょう。

バージョン 3.4 で変更: *indent* が None でなければ (',', ': ') がデフォルトで使われます。

default を指定する場合は関数を指定して、この関数はそれ以外では直列化できないオブジェクトに対して呼び出されます。その関数は、オブジェクトを JSON でエンコードできるバージョンにして返すか、さもなければ *TypeError* を送出しなければなりません。指定しない場合は、*TypeError* が送出されます。

バージョン 3.6 で変更: すべての引数は、**キーワード専用** になりました。

default(*o*)

このメソッドをサブクラスで実装する際には *o* に対して直列化可能なオブジェクトを返すか、基底クラスの実装を (*TypeError* を送出するために) 呼び出すかします。

たとえば、任意のイテレータをサポートするために、次のように実装します:

```
def default(self, o):
   try:
       iterable = iter(o)
   except TypeError:
       pass
   else:
       return list(iterable)
   # Let the base class default method raise the TypeError
   return json.JSONEncoder.default(self, o)
```

encode(*o*)

Python データ構造 *o* の JSON 文字列表現を返します。たとえば:

```
>>> json.JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(*o*)

与えられたオブジェクト *o* をエンコードし、得られた文字列表現ごとに yield します。たとえば:

```
for chunk in json.JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

19.2.3 例外

exception json.**JSONDecodeError**(*msg*, *doc*, *pos*)

ValueError のサブクラスで、以下の追加の属性を持ちます:

msg

フォーマットされていないエラーメッセージです。

doc

パース対象 JSON ドキュメントです。

pos

doc の、解析に失敗した開始インデクスです。

lineno

pos に対応する行です。

colno

pos に対応する列です。

バージョン 3.5 で追加.

19.2.4 標準への準拠と互換性

JSON 形式の仕様は **RFC 7159** と ECMA-404 で規定されています。この節では、このモジュールの RFC への準拠水準について詳しく述べます。簡単のために、*JSONEncoder* および *JSONDecoder* の子クラスと、明示的に触れられていないパラメータについては考慮しないことにします。

このモジュールは、JavaScript では正しいが JSON では不正ないくつかの拡張が実装されているため、厳密な意味では RFC に準拠していません。特に:

- 無限および NaN の数値を受け付け、また出力します;
- あるオブジェクト内での同じ名前の繰り返しを受け付け、最後の名前と値のペアの値のみを使用します。

この RFC は、RFC 準拠のパーサが RFC 準拠でない入力テキストを受け付けることを許容しているので、このモジュールの脱直列化は技術的に言えば、デフォルトの設定では RFC に準拠しています。

文字エンコーディング

RFC は、UTF-8、UTF-16、UTF-32 のいずれかで JSON を表現するように要求しており、UTF-8 が最大の互換性を確保するために推奨されるデフォルトです。

RFC で要求ではなく許可されている通り、このモジュールのシリアライザはデフォルトで *ensure_ascii=True* という設定を用い、従って、結果の文字列が ASCII 文字しか含まないように出力をエスケープします。

ensure_ascii パラメータ以外は、このモジュールは Python オブジェクトと *Unicode* **文字列** の間の変換において厳密に定義されていて、それ以外のパラメータで文字エンコーディングに直接的に関わるものはありません。

RFC は JSON テキストの最初にバイトオーダマーク (BOM) を追加することを禁止していますので、このモジュールはその出力に BOM を追加しません。RFC は JSON デシリアライザが入力の一番最初の BOM を無視することを、許容はしますが求めてはいません。このモジュールのデシリアライザは一番最初の BOM を見つけると *ValueError* を送出します。

RFC は JSON 文字列に正当な Unicode 文字に対応付かないバイト列 (例えばペアにならない UTF-16 サロゲートのかたわれ) が含まれることを明示的に禁止してはおらず、もちろんこれは相互運用性の問題を引き起こします。デフォルトでは、このモジュールは (オリジナルの *str* にある場合) そのようなシーケンスのコードポイントを受け取り、出力します。

無限および NaN の数値

RFC は、無限もしくは NaN の数値の表現は許可していません。それにも関わらずデフォルトでは、このモジュールは `Infinity`、`-Infinity`、`NaN` を正しい JSON の数値リテラルの値であるかのように受け付け、出力します:

```
>>> # Neither of these calls raises an exception, but the results are not valid JSON
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # Same when deserializing
>>> json.loads('-Infinity')
-inf
>>> json.loads('NaN')
nan
```

シリアライザでは、この振る舞いを変更するのに *allow_nan* パラメータが使えます。デシリアライザでは、この振る舞いを変更するのに *parse_constant* パラメータが使えます。

オブジェクト中に重複した名前の扱い

RFC は JSON オブジェクト中の名前はユニークでなければならないと規定していますが、JSON オブジェクトで名前が繰り返された場合の扱いについて指定していません。デフォルトでは、このモジュールは例外を送出せず、かわりに重複した名前のうち、最後に出現した名前と値のペア以外を無視します。

```
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{'x': 3}
```

object_pairs_hook パラメータでこの動作を変更できます。

トップレベルの非オブジェクト、非配列の値の扱い

廃止された **RFC 4627** によって規定された古いバージョンの JSON では、JSON テキストのトップレベルの値は JSON オブジェクトか配列 (Python での *dict* か *list*) であることを要求していて、JSON の null, boolean, number, string であることは許されていませんでしたが、この制限は **RFC 7159** により取り払われました。このモジュールはこの制限を持っていませんし、シリアライザでもデシリアライズでも、一度としてこの制限で実装されたことはありません。

それにも関わらず、相互運用可能性を最大化したいならば、あなた自身の手で自発的にその制約に忠実に従いたいと思うでしょう。

実装の制限

いくつかの JSON デシリアライザの実装は、以下の制限を設定することがあります。

- 受け入れられる JSON テキストのサイズ
- JSON オブジェクトと配列のネストの最大の深さ
- JSON 数値の範囲と精度
- JSON 文字列の内容と最大の長さ

このモジュールは関連する Python データ型や Python インタプリタ自身の制約の世界を超えたそのような制約を強要はしません。

JSON にシリアライズする際には、あなたの JSON を消費する側のアプリケーションが持つ当該制約に思いを馳せてください。とりわけ JSON 数値を IEEE 754 倍精度浮動小数にデシリアライズする際の問題はありがちで、すなわちその有効桁数と精度の制限の影響を受けます。これは、極端に大きな値を持った Python *int* をシリアライズするとき、あるいは *decimal.Decimal* のような ”風変わりな” 数値型をシリアライズするとき、に特に関係があります。

19.2.5 コマンドラインインターフェイス

ソースコード: Lib/json/tool.py

json.tool モジュールは JSON オブジェクトの検証と整形出力のための、単純なコマンドラインインターフェイスを提供します。

オプショナルな `infile` 引数、`outfile` 引数が指定されない場合、それぞれ *sys.stdin* と *sys.stdout* が使われます:

```
$ echo '{"json": "obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

バージョン 3.5 で変更: 出力が、入力と同じ順序になりました。辞書をキーでアルファベット順に並べ替えた出力が欲しければ、*--sort-keys* オプションを使ってください。

コマンドラインオプション

infile

検証を行う、あるいは整形出力を行う JSON ファイルを指定します:

```
$ python -m json.tool mp_films.json
[
    {
        "title": "And Now for Something Completely Different",
        "year": 1971
    },
    {
        "title": "Monty Python and the Holy Grail",
        "year": 1975
    }
]
```

infile を指定しない場合、*sys.stdin* から読み込みます。

outfile

infile に対する出力を、この *outfile* ファイルに出力します。指定しない場合 *sys.stdout* に出力します。

--sort-keys

辞書の出力を、キーのアルファベット順にソートします。

バージョン 3.5 で追加.

--json-lines

すべての入力行を個別の JSON オブジェクトとしてパースします。

バージョン 3.8 で追加.

-h, --help
ヘルプメッセージを出力します

脚注

19.3 mailcap --- mailcap ファイルの操作

ソースコード: Lib/mailcap.py

---

mailcap ファイルは、メールリーダや Web ブラウザのような MIME 対応のアプリケーションが、異なる MIME タイプのファイルにどのように反応するかを設定するために使われます (”mailcap” の名前は ”mail capability” から取られました)。例えば、ある mailcap ファイルに `video/mpeg; xmpeg %s` のような行が入っていたとします。ユーザが email メッセージや Web ドキュメント上でその MIME タイプ *video/mpeg* に遭遇すると、`%s` はファイル名 (通常テンポラリファイルに属するものになります) に置き換えられ、ファイルを閲覧するために `xmpeg` プログラムが自動的に起動されます。

mailcap の形式は **RFC 1524**, ”A User Agent Configuration Mechanism For Multimedia Mail Format Information” で文書化されていますが、この文書はインターネット標準ではありません。しかしながら、mailcap ファイルはほとんどの Unix システムでサポートされています。

mailcap.findmatch(*caps*, *MIMEtype*, *key='view'*, *filename='/dev/null'*, *plist=[]*)
2 要素のタプルを返します; 最初の要素は文字列で、実行すべきコマンド (*os.system()* に渡されます) が入っています。二つめの要素は与えられた MIME タイプに対する mailcap エントリです。一致する MIME タイプが見つからなかった場合、`(None, None)` が返されます。

key は desired フィールドの値で、実行すべき動作のタイプを表現します; ほとんどの場合、単に MIME 形式のデータ本体を見たいと思うので、標準の値は 'view' になっています。与えられた MIME 型をもつ新たなデータ本体を作成した場合や、既存のデータ本体を置き換えたい場合には、'view' の他に 'compose' および 'edit' を取ることもできます。これらフィールドの完全なリストについては **RFC 1524** を参照してください。

filename はコマンドライン中で `%s` に代入されるファイル名です; 標準の値は `'/dev/null'` で、たいていこの値を使いたいわけではないはずです。従って、ファイル名を指定してこのフィールドを上書きする必要があるでしょう。

plist は名前付けされたパラメタのリストです; 標準の値は単なる空のリストです。リスト中の各エントリはパラメタ名を含む文字列、等号 (`'='`)、およびパラメタの値でなければなりません。mailcap エントリには `%{foo}` といったような名前つきのパラメタを含めることができ、'foo' と名づけられたパラメタの値に置き換えられます。例えば、コマンドライン `showpartial %{id} %{number} %{total}` が mailcap ファイルにあり、*plist* が `['id=1', 'number=2', 'total=3']` に設定されていれば、コマンドラインは `'showpartial 1 2 3'` になります。

mailcap ファイル中では、オプションの ”test” フィールドを使って、(計算機アーキテクチャや、利用しているウィンドウシステムといった) 何らかの外部条件をテストするよう指定することができます。

findmatch() はこれらの条件を自動的にチェックし、チェックが失敗したエントリを読み飛ばします。

バージョン 3.8.16 で変更: To prevent security issues with shell metacharacters (symbols that have special effects in a shell command line), `findmatch` will refuse to inject ASCII characters other than alphanumerics and `@+=:,./-_` into the returned command line.

If a disallowed character appears in *filename*, `findmatch` will always return `(None, None)` as if no entry was found. If such a character appears elsewhere (a value in *plist* or in *MIMEtype*), `findmatch` will ignore all mailcap entries which use that value. A *warning* will be raised in either case.

`mailcap.getcaps()`

MIME タイプを mailcap ファイルのエントリに対応付ける辞書を返します。この辞書は *findmatch()* 関数に渡されるべきものです。エントリは辞書のリストとして記憶されますが、この表現形式の詳細について知っておく必要はないでしょう。

mailcap 情報はシステム上で見つかった全ての mailcap ファイルから導出されます。ユーザ設定の mailcap ファイル `$HOME/.mailcap` はシステムの mailcap ファイル `/etc/mailcap` 、`/usr/etc/mailcap` 、および `/usr/local/etc/mailcap` の内容を上書きします。

以下に使用例を示します:

```
>>> import mailcap
>>> d = mailcap.getcaps()
>>> mailcap.findmatch(d, 'video/mpeg', filename='tmp1223')
('xmpeg tmp1223', {'view': 'xmpeg %s'})
```

19.4 mailbox --- 様々な形式のメールボックス操作

ソースコード: Lib/mailbox.py

このモジュールでは二つのクラス *Mailbox* および *Message* をディスク上のメールボックスとそこに収められたメッセージへのアクセスと操作のために定義しています。*Mailbox* は辞書のようなキーからメッセージへの対応付けを提供しています。*Message* は *email.message* モジュールの *Message* を拡張して形式ごとの状態と振る舞いを追加しています。サポートされるメールボックスの形式は Maildir, mbox, MH, Babyl, MMDF です。

参考:

email **モジュール**　メッセージの表現と操作。

19.4.1 Mailbox オブジェクト

class mailbox.**Mailbox**

メールボックス。内容を確認したり変更したりできます。

Mailbox 自体はインタフェースを定義し形式ごとのサブクラスに継承されるように意図されたもので、インスタンス化されることは想定されていません。インスタンス化したいならばサブクラスを代わりに使うべきです。

Mailbox のインタフェースは辞書風で、小さなキーがメッセージに対応します。キーは対象となる *Mailbox* インスタンスが発行するもので、そのインスタンスに対してのみ意味を持ちます。一つのキーは一つのメッセージにひも付けられ、その対応はメッセージが他のメッセージで置き換えられるような更新をされたあとも続きます。

メッセージを *Mailbox* インスタンスに追加するには集合風のメソッド *add()* を使います。また削除は `del` 文または集合風の *remove()* や *discard()* を使って行ないます。

Mailbox インタフェースのセマンティクスと辞書のそれとは注意すべき違いがあります。メッセージは、要求されるたびに新しい表現 (典型的には *Message* インスタンス) が現在のメールボックスの状態に基づいて生成されます。同様に、メッセージが *Mailbox* インスタンスに追加される時も、渡されたメッセージ表現の内容がコピーされます。どちらの場合も *Mailbox* インスタンスにメッセージ表現への参照は保たれません。

デフォルトの *Mailbox* イテレータはメッセージ表現ごとに繰り返すもので、辞書のイテレータのようにキーごとの繰り返しではありません。さらに、繰り返し中のメールボックスを変更することは安全であり整合的に定義されています。イテレータが作られた後にメールボックスに追加されたメッセージはそのイテレータからは見えません。そのイテレータが yield するまえにメールボックスから削除されたメッセージは黙ってスキップされますが、イテレータからのキーを使ったときにはそのキーに対応するメッセージが削除されているならば *KeyError* を受け取ることになります。

> **警告:** 十分な注意を、何か他のプロセスによっても同時に変更される可能性のあるメールボックスを更新する時は、払わなければなりません。そのようなタスクをこなすのに最も安全なメールボックス形式は Maildir で、mbox のような単一ファイルの形式を並行した書き込みに利用するのは避けるように努力しましょう。メールボックスを更新する場面では、**必ず** *lock()* と *unlock()* メソッドを、ファイル内のメッセージを読んだり書き込んだり削除したりといった操作をする **前** に、呼び出してロックします。メールボックスをロックし損なうと、メッセージを失ったりメールボックス全体をぐちゃぐちゃにしたりする羽目に陥ります。

Mailbox インスタンスには次のメソッドがあります:

add(*message*)

メールボックスに *message* を追加し、それに割り当てられたキーを返します。

引数 *message* は *Message* インスタンス、*email.message.Message* インスタンス、文字列、バイト文字列、ファイル風オブジェクト (バイナリモードで開かれていなければなりません) を使えます。*message* が適切な形式に特化した *Message* サブクラスのインスタンス (例えばメールボック

スが *mbox* インスタンスのときの *mboxMessage* インスタンス) であれば、形式ごとの情報が利用されます。そうでなければ、形式ごとに必要な情報は適当なデフォルトが使われます。

バージョン 3.2 で変更: バイナリ入力のサポートが追加されました。

remove(*key*)
__delitem__(*key*)
discard(*key*)
メールボックスから *key* に対応するメッセージを削除します。

対応するメッセージが無い場合、メソッドが *remove()* または *__delitem__()* として呼び出されている時は *KeyError* 例外が送出されます。しかし、*discard()* として呼び出されている場合は例外は発生しません。基づいているメールボックス形式が別のプロセスからの平行した変更をサポートしているならば、この *discard()* の振る舞いの方が好まれるかもしれません。

__setitem__(*key*, *message*)
key に対応するメッセージを *message* で置き換えます。*key* に対応しているメッセージが既に無くなっている場合 *KeyError* 例外が送出されます。

add() と同様に、引数の *message* には *Message* インスタンス、*email.message.Message* インスタンス、文字列、バイト文字列、ファイル風オブジェクト (バイナリモードで開かれていなければなりません) を使えます。*message* が適切な形式に特化した *Message* サブクラスのインスタンス (例えばメールボックスが *mbox* インスタンスのときの *mboxMessage* インスタンス) であれば、形式ごとの情報が利用されます。そうでなければ、現在 *key* に対応するメッセージの形式ごとの情報が変更されずに残ります。

iterkeys()
keys()
iterkeys() として呼び出されると全てのキーについてのイテレータを返しますが、*keys()* として呼び出されるとキーのリストを返します。

itervalues()
__iter__()
values()
itervalues() または *__iter__()* として呼び出されると全てのメッセージの表現についてのイテレータを返しますが、*values()* として呼び出されるとその表現のリストを返します。メッセージは適切な形式ごとの *Message* サブクラスのインスタンスとして表現されるのが普通ですが、*Mailbox* インスタンスが初期化されるときに指定すればお好みのメッセージファクトリを使うこともできます。

注釈: *__iter__()* は辞書のそれのようにキーについてのイテレータではありません。

---

iteritems()
items()
(*key*, *message*) ペア、ただし *key* はキーで *message* はメッセージ表現、のイテレータ (*iteritems()* として呼び出された場合)、またはリスト (*items()* として呼び出された場合)

を返します。メッセージは適切な形式ごとの *Message* サブクラスのインスタンスとして表現されるのが普通ですが、*Mailbox* インスタンスが初期化されるときに指定すればお好みのメッセージファクトリを使うこともできます。

get(*key*, *default=None*)

__getitem__(*key*)

key に対応するメッセージの表現を返します。対応するメッセージが存在しない場合、*get()* として呼び出されたなら *default* を返しますが、*__getitem__()* として呼び出されたなら *KeyError* 例外が送出されます。メッセージは適切な形式ごとの *Message* サブクラスのインスタンスとして表現されるのが普通ですが、*Mailbox* スタンスが初期化されるときに指定すればお好みのメッセージファクトリを使うこともできます。

get_message(*key*)

key に対応するメッセージの表現を形式ごとの *Message* サブクラスのインスタンスとして返します。もし対応するメッセージが存在しなければ *KeyError* 例外が送出されます。

get_bytes(*key*)

key に対応するメッセージのバイト列を返すか、そのようなメッセージが存在しない場合は *KeyError* 例外を送出します。

バージョン 3.2 で追加.

get_string(*key*)

key に対応するメッセージの文字列表現を返すか、そのようなメッセージが存在しない場合は *KeyError* 例外を送出します。このメッセージは *email.message.Message* を通して処理されて 7 ビットクリーンな表現へ変換されます。

get_file(*key*)

key に対応するメッセージの表現をファイル風表現として返します。もし対応するメッセージが存在しなければ *KeyError* 例外が送出されます。ファイル風オブジェクトはバイナリモードで開かれているように振る舞います。このファイルは必要がなくなったら閉じなければなりません。

バージョン 3.2 で変更: ファイルオブジェクトは実際はバイナリファイルです; 以前は誤ってテキストモードで返されていました。また、現在ファイル風オブジェクトはコンテキストマネージャプロトコルをサポートしています: `with` 文を用いることで自動的にファイルを閉じることができます。

注釈: 他の表現方法とは違い、ファイル風オブジェクトはそれを作り出した *Mailbox* インスタンスやそれが基づいているメールボックスと独立である必要がありません。より詳細な説明は各サブクラスごとにあります。

__contains__(*key*)

key がメッセージに対応していれば `True` を、そうでなければ `False` を返します。

__len__()

メールボックス中のメッセージ数を返します。

clear()
メールボックスから全てのメッセージを削除します。

pop(*key*, *default=None*)
key に対応するメッセージの表現を返します。メッセージは適切な形式ごとの *Message* サブクラスのインスタンスとして表現されるのが普通ですが、*Mailbox* インスタンスが初期化されるときに指定すればお好みのメッセージファクトリを使うこともできます。

popitem()
任意に選んだ (*key*, *message*) ペアを返します。ただしここで *key* はキーで *message* はメッセージ表現です。もしメールボックスが空ならば、*KeyError* 例外を送出します。メッセージは適切な形式ごとの *Message* サブクラスのインスタンスとして表現されるのが普通ですが、*Mailbox* インスタンスが初期化されるときに指定すればお好みのメッセージファクトリを使うこともできます。

update(*arg*)
引数 *arg* は *key* から *message* へのマッピングまたは (*key*, *message*) ペアのイテレート可能オブジェクトでなければなりません。メールボックスは、各 *key* と *message* のペアについて *__setitem__()* を使ったかのように *key* に対応するメッセージが *message* になるように更新されます。*__setitem__()* と同様に、*key* は既存のメールボックス中のメッセージに対応しているものでなければならず、そうでなければ *KeyError* が送出されます。ですから、一般的には *arg* に *Mailbox* インスタンスを渡すのは間違いです。

注釈: 辞書と違い、キーワード引数はサポートされていません。

---

flush()
保留されている変更をファイルシステムに書き込みます。*Mailbox* のサブクラスによっては変更はいつも直ちにファイルに書き込まれ *flush()* は何もしないということもありますが、それでもこのメソッドを呼ぶように習慣付けておきましょう。

lock()
メールボックスの排他的アドバイザリロックを取得し、他のプロセスが変更しないようにします。ロックが取得できない場合 *ExternalClashError* が送出されます。ロック機構はメールボックス形式によって変わります。メールボックスの内容に変更を加えるときは **いつも** ロックを掛けるべきです。

unlock()
メールボックスのロックが存在する場合は解放します。

close()
メールボックスをフラッシュし、必要ならばアンロックし、開いているファイルを閉じます。*Mailbox* サブクラスによっては何もしないこともあります。

Maildir

class mailbox.**Maildir**(*dirname*, *factory=None*, *create=True*)

Maildir 形式のメールボックスのための *Mailbox* のサブクラス。パラメータ *factory* は呼び出し可能オブジェクトで (バイナリモードで開かれているかのように振る舞う) ファイル風メッセージ表現を受け付けて好みの表現を返すものです。*factory* が None ならば、*MaildirMessage* がデフォルトのメッセージ表現として使われます。*create* が True ならばメールボックスが存在しないときには作成します。

create が True で、パス *dirname* が存在する場合、ディレクトリレイアウトを検証せずに既存の maildir として扱います。

path ではなく *dirname* と命名される歴史的な理由のためです。

Maildir はディレクトリ型のメールボックス形式でメール転送エージェント qmail 用に発明され、現在では多くの他のプログラムでもサポートされているものです。Maildir メールボックス中のメッセージは共通のディレクトリ構造の下で個別のファイルに保存されます。このデザインにより、Maildir メールボックスは複数の無関係のプログラムからデータを失うことなくアクセスしたり変更したりできます。そのためロックは不要です。

Maildir メールボックスには三つのサブディレクトリ tmp, new, cur があります。メッセージはまず tmp サブディレクトリに瞬間的に作られた後、new サブディレクトリに移動されて配送を完了します。メールユーザエージェントが引き続いて cur サブディレクトリにメッセージを移動しメッセージの状態についての情報をファイル名に追加される特別な "info" セクションに保存することができます。

Courier メール転送エージェントによって導入されたスタイルのフォルダもサポートされます。主たるメールボックスのサブディレクトリは '.' がファイル名の先頭であればフォルダと見なされます。フォルダ名は *Maildir* によって先頭の '.' を除いて表現されます。各フォルダはまた Maildir メールボックスですがさらにフォルダを含むことはできません。その代わり、論理的包含関係は例えば "Archived.2005.07" のような '.' を使ったレベル分けで表わされます。

注釈: 本来の Maildir 仕様ではある種のメッセージのファイル名にコロン (':') を使う必要があります。しかしながら、オペレーティングシステムによってはこの文字をファイル名に含めることができないことがあります。そういった環境で Maildir のような形式を使いたい場合、代わりに使われる文字を指定する必要があります。感嘆符 ('!') を使うのが一般的な選択です。以下の例を見てください:

```
import mailbox
mailbox.Maildir.colon = '!'
```

colon 属性はインスタンスごとにセットしても構いません。

Maildir インスタンスには *Mailbox* の全てのメソッドに加え以下のメソッドもあります:

list_folders()

全てのフォルダ名のリストを返します。

get_folder(*folder*)

名前が *folder* であるフォルダを表わす *Maildir* インスタンスを返します。そのようなフォルダが存在しなければ *NoSuchMailboxError* 例外が送出されます。

add_folder(*folder*)
名前が *folder* であるフォルダを作り、それを表わす *Maildir* インスタンスを返します。

remove_folder(*folder*)
名前が *folder* であるフォルダを削除します。もしフォルダに一つでもメッセージが含まれていれば *NotEmptyError* 例外が送出されフォルダは削除されません。

clean()
過去 36 時間以内にアクセスされなかったメールボックス内の一時ファイルを削除します。Maildir 仕様はメールを読むプログラムはときどきこの作業をすべきだとしています。

Maildir で実装された *Mailbox* のいくつかのメソッドには特別な注意が必要です:

add(*message*)
__setitem__(*key*, *message*)
update(*arg*)

> **警告:** これらのメソッドは一意的なファイル名をプロセス ID に基づいて生成します。複数のスレッドを使う場合は、同じメールボックスを同時に操作しないようにスレッド間で調整しておかないと検知されない名前の衝突が起こりメールボックスを壊すかもしれません。

flush()
Maildir メールボックスへの変更は即時に適用されるので、このメソッドは何もしません。

lock()
unlock()
Maildir メールボックスはロックをサポート (または要求) しないので、このメソッドは何もしません。

close()
Maildir インスタンスは開いたファイルを保持しませんしメールボックスはロックをサポートしませんので、このメソッドは何もしません。

get_file(*key*)
ホストのプラットフォームによっては、返されたファイルが開いている間、元になったメッセージを変更したり削除したりできない場合があります。

参考:

Courier の maildir マニュアルページ　Maildir 形式の仕様。フォルダをサポートする一般的な拡張について記述されています。

Using maildir format　Maildir 形式の発明者による注意書き。更新された名前生成規則と "info" の解釈についても含まれます。

mbox

class mailbox.**mbox**(*path*, *factory=None*, *create=True*)

mbox 形式のメールボックスのための *Mailbox* のサブクラス。パラメータ *factory* は呼び出し可能オブジェクトで (バイナリモードで開かれているかのように振る舞う) ファイル風メッセージ表現を受け付けて好みの表現を返すものです。*factory* が None ならば、*mboxMessage* がデフォルトのメッセージ表現として使われます。*create* が True ならばメールボックスが存在しないときには作成します。

mbox 形式は Unix システム上でメールを保存する古くからある形式です。mbox メールボックスでは全てのメッセージが一つのファイルに保存されておりそれぞれのメッセージは "From " という 5 文字で始まる行を先頭に付けられています。

mbox 形式には幾つかのバリエーションがあり、それぞれオリジナルの形式にあった欠点を克服すると主張しています。互換性のために、*mbox* はオリジナルの (時に *mboxo* と呼ばれる) 形式を実装しています。すなわち、*Content-Length* ヘッダはもしあっても無視され、メッセージのボディにある行頭の "From " はメッセージを保存する際に ">From " に変換されますが、この ">From " は読み出し時にも "From " に変換されません。

mbox で実装された *Mailbox* のいくつかのメソッドには特別な注意が必要です:

get_file(*key*)

mbox インスタンスに対し flush() や close() を呼び出した後でファイルを使用すると予期しない結果を引き起こしたり例外が送出されたりすることがあります。

lock()

unlock()

3 種類のロック機構が使われます --- ドットロッキングと、もし使用可能ならば flock() と lockf() システムコールです。

参考:

tin の mbox マニュアルページ　mbox 形式の仕様でロックについての詳細を含む。

Configuring Netscape Mail on Unix: Why The Content-Length Format is Bad　バリエーションの一つではなくオリジナルの mbox を使う理由。

"mbox" は相互に互換性を持たないいくつかのメールボックスフォーマットの集まりです　mbox バリエーションの歴史。

MH

class mailbox.**MH**(*path*, *factory=None*, *create=True*)

MH 形式のメールボックスのための *Mailbox* のサブクラス。パラメータ *factory* は呼び出し可能オブジェクトで (バイナリモードで開かれているかのように振る舞う) ファイル風メッセージ表現を受け付けて好みの表現を返すものです。*factory* が None ならば、*MHMessage* がデフォルトのメッセージ表現として使われます。*create* が True ならばメールボックスが存在しないときには作成します。

MH はディレクトリに基づいたメールボックス形式で MH Message Handling System というメールユーザエージェントのために発明されました。MH メールボックス中のそれぞれのメッセージは一つの

ファイルとして収められています。MH メールボックスにはメッセージの他に別の MH メールボックス (**フォルダ** と呼ばれます) を含んでもかまいません。フォルダは無限にネストできます。MH メールボックスにはもう一つ **シーケンス** という名前付きのリストでメッセージをサブフォルダに移動することなく論理的に分類するものがサポートされています。シーケンスは各フォルダの `.mh_sequences` というファイルで定義されます。

MH クラスは MH メールボックスを操作しますが、**mh** の動作の全てを模倣しようとはしていません。特に、**mh** が状態と設定を保存する `context` や `.mh_profile` といったファイルは書き換えませんし影響も受けません。

MH インスタンスには *Mailbox* の全てのメソッドの他に次のメソッドがあります:

list_folders()
全てのフォルダ名のリストを返します。

get_folder(*folder*)
folder という名前のフォルダを表わす *MH* インスタンスを返します。もしフォルダが存在しなければ *NoSuchMailboxError* 例外が送出されます。

add_folder(*folder*)
folder という名前のフォルダを作成し、それを表わす *MH* インスタンスを返します。

remove_folder(*folder*)
名前が *folder* であるフォルダを削除します。もしフォルダに一つでもメッセージが含まれていれば *NotEmptyError* 例外が送出されフォルダは削除されません。

get_sequences()
folder という名前のフォルダを削除します。フォルダにメッセージが一つでも残っていれば、*NotEmptyError* 例外が送出されフォルダは削除されません。

set_sequences(*sequences*)
シーケンス名をキーのリストに対応付ける辞書を返します。シーケンスが一つもなければ空の辞書を返します。

pack()
メールボックス中のシーケンスを `get_sequences()` で返されるような名前とキーのリストを対応付ける辞書 *sequences* に基づいて再定義します。

注釈: 番号付けの間隔を詰める必要に応じてメールボックス中のメッセージの名前を付け替えます。シーケンスのリストのエントリもそれに応じて更新されます。

既に発行されたキーはこの操作によって無効になるのでそれ以降使ってはなりません:

remove(*key*)
__delitem__(*key*)
discard(*key*)
これらのメソッドはメッセージを直ちに削除します。名前の前にコンマを付加してメッセージに削

除の印を付けるという MH の規約は使いません。

lock()
unlock()
3 種類のロック機構が使われます --- ドットロッキングと、もし使用可能ならば `flock()` と `lockf()` システムコールです。MH メールボックスに対するロックとは `.mh_sequences` のロックと、それが影響を与える操作中だけの個々のメッセージファイルに対するロックを意味します。

get_file(*key*)
ホストプラットフォームにより、ファイルが開かれたままの場合はメッセージを削除することができない場合があります。

flush()
MH メールボックスへの変更は即時に適用されますのでこのメソッドは何もしません。

close()
MH インスタンスは開いたファイルを保持しませんのでこのメソッドは *unlock()* と同じです。

参考:

nmh - Message Handling System　**mh** の改良版である **nmh** のホームページ。

MH & nmh: Email for Users & Programmers　GPL ライセンスの **mh** および **nmh** の本で、このメールボックス形式についての情報があります。

Babyl

class mailbox.Babyl(*path*, *factory=None*, *create=True*)
Babyl 形式のメールボックスのための *Mailbox* のサブクラス。パラメータ *factory* は呼び出し可能オブジェクトで (バイナリモードで開かれているかのように振る舞う) ファイル風メッセージ表現を受け付けて好みの表現を返すものです。*factory* が `None` ならば、*BabylMessage* がデフォルトのメッセージ表現として使われます。*create* が `True` ならばメールボックスが存在しないときには作成します。

Babyl は単一ファイルのメールボックス形式で Emacs に付属している Rmail メールユーザエージェントで使われているものです。メッセージの開始は Control-Underscore (`'\037'`) および Control-L (`'\014'`) の二文字を含む行で示されます。メッセージの終了は次のメッセージの開始または最後のメッセージの場合には Control-Underscore を含む行で示されます。

Babyl メールボックス中のメッセージには二つのヘッダのセット、オリジナルヘッダといわゆる可視ヘッダ、があります。可視ヘッダは典型的にはオリジナルヘッダの一部を分り易いように再整形したり短くしたりしたものです。Babyl メールボックス中のそれぞれのメッセージには **ラベル** というそのメッセージについての追加情報を記録する短い文字列のリストを伴い、メールボックス中に見出されるユーザが定義した全てのラベルのリストは Babyl オプションセクションに保持されます。

Babyl インスタンスには *Mailbox* の全てのメソッドの他に次のメソッドがあります:

get_labels()
メールボックスで使われているユーザが定義した全てのラベルのリストを返します。

注釈: メールボックスにどのようなラベルが存在するかを決めるのに、Babyl オプションセクションのリストを参考にせず、実際のメッセージを捜索しますが、Babyl セクションもメールボックスが変更されたときにはいつでも更新されます。

Babyl で実装された *Mailbox* のいくつかのメソッドには特別な注意が必要です:

get_file(*key*)
Babyl メールボックスにおいて、メッセージのヘッダはボディと繋がって格納されていません。ファイル風の表現を生成するために、ヘッダとボディがファイルと同じ API を持つ *io.BytesIO* インスタンスに一緒にコピーされます。その結果、ファイル風オブジェクトは元にしているメールボックスとは真に独立していますが、文字列表現と比べてメモリーを節約することにはなりません。

lock()
unlock()
3 種類のロック機構が使われます --- ドットロッキングと、もし使用可能ならば `flock()` と `lockf()` システムコールです。

参考:

Format of Version 5 Babyl Files　Babyl 形式の仕様。

Reading Mail with Rmail　Rmail のマニュアルで Babyl のセマンティクスについての情報も少しある。

MMDF

class mailbox.MMDF(*path*, *factory=None*, *create=True*)
MMDF 形式のメールボックスのための *Mailbox* のサブクラス。パラメータ *factory* は呼び出し可能オブジェクトで (バイナリモードで開かれているかのように振る舞う) ファイル風メッセージ表現を受け付けて好みの表現を返すものです。*factory* が `None` ならば、*MMDFMessage* がデフォルトのメッセージ表現として使われます。*create* が `True` ならばメールボックスが存在しないときには作成します。

MMDF は単一ファイルのメールボックス形式で Multichannel Memorandum Distribution Facility というメール転送エージェント用に発明されたものです。各メッセージは mbox と同様の形式で収められますが、前後を 4 つの Control-A (`'\001'`) を含む行で挟んであります。mbox 形式と同じようにそれぞれのメッセージの開始は "From " の 5 文字を含む行で示されますが、それ以外の場所での "From " は格納の際 ">From " には変えられません。それは追加されたメッセージ区切りによって新たなメッセージの開始と見間違うことが避けられるからです。

MMDF で実装された *Mailbox* のいくつかのメソッドには特別な注意が必要です:

get_file(*key*)
MMDF インスタンスに対し `flush()` や `close()` を呼び出した後でファイルを使用すると予期しない結果を引き起こしたり例外が送出されたりすることがあります。

lock()

unlock()
3 種類のロック機構が使われます --- ドットロッキングと、もし使用可能ならば flock() と lockf() システムコールです。

参考:

mmdf man page from tin　ニュースリーダ tin のドキュメント中の MMDF 形式仕様。

MMDF　Multichannel Memorandum Distribution Facility についてのウィキペディアの記事。

19.4.2 Message objects

class mailbox.Message(*message=None*)
email.message モジュールの *Message* のサブクラス。*mailbox.Message* のサブクラスはメールボックス形式ごとの状態と動作を追加します。

message が省略された場合、新しいインスタンスはデフォルトの空の状態で生成されます。*message* が *email.message.Message* インスタンスならばその内容がコピーされます。さらに、*message* が *Message* インスタンスならば、形式固有の情報も可能な限り変換されます。*message* が文字列かバイト列またはファイルならば、読まれ解析されるべき **RFC 2822** 準拠のメッセージを含んでいなければなりません。ファイルはバイナリモードで開かれているべきですが、後方互換性のためテキストモードファイルも受け付けます。

サブクラスにより提供される形式ごとの状態と動作は様々ですが、一般に或るメールボックスに固有のものでないプロパティだけがサポートされます (おそらくプロパティのセットはメールボックス形式ごとに固有でしょうが)。例えば、単一ファイルメールボックス形式におけるファイルオフセットやディレクトリ式メールボックス形式におけるファイル名は保持されません、というのもそれらは元々のメールボックスにしか適用できないからです。しかし、メッセージがユーザに読まれたかどうかあるいは重要だとマークされたかどうかという状態は保持されます、というのはそれらはメッセージ自体に適用されるからです。

Mailbox インスタンスを使って取得したメッセージを表現するのに *Message* インスタンスが使われなければいけないとは要求していません。ある種の状況では *Message* による表現を生成するのに必要な時間やメモリーが受け入れられないこともあります。そういった状況では *Mailbox* インスタンスは文字列やファイル風オブジェクトの表現も提供できますし、*Mailbox* インスタンスを初期化する際にメッセージファクトリーを指定することもできます。

MaildirMessage

class mailbox.MaildirMessage(*message=None*)
Maildir 固有の動作をするメッセージ。引数 *message* は *Message* のコンストラクタと同じ意味を持ちます。

通常、メールユーザエージェントは new サブディレクトリにある全てのメッセージをユーザが最初にメールボックスを開くか閉じるかした後で cur サブディレクトリに移動し、メッセージが実際に読まれたかどうかを記録します。cur にある各メッセージには状態情報を保存するファイル名に付け加えら

れた ”info” セクションがあります。(メールリーダの中には ”info” セクションを new にあるメッセージに付けることもあります。) ”info” セクションには二つの形式があります。一つは ”2,” の後に標準化されたフラグのリストを付けたもの (たとえば ”2,FR”)、もう一つは ”1,” の後にいわゆる実験的情報を付け加えるものです。Maildir の標準的なフラグは以下の通りです:

Flag	意味	説明
D	ドラフト (Draft)	作成中
F	フラグ付き (Flagged)	重要とされたもの
P	通過 (Passed)	転送、再送またはバウンス
R	返答済み (Replied)	返答されたもの
S	既読 (Seen)	読んだもの
T	ごみ (Trashed)	削除予定とされたもの

MaildirMessage インスタンスは以下のメソッドを提供します:

get_subdir()
”new” (メッセージが new サブディレクトリに保存されるべき場合) または ”cur” (メッセージが cur サブディレクトリに保存されるべき場合) のどちらかを返します。

注釈: メッセージは通常メールボックスがアクセスされた後、メッセージが読まれたかどうかに関わらず new から cur に移動されます。メッセージ msg は "S" in msg.get_flags() が True ならば読まれています。

---

set_subdir(*subdir*)
メッセージが保存されるべきサブディレクトリをセットします。パラメータ *subdir* は ”new” または ”cur” のいずれかでなければなりません。

get_flags()
現在セットされているフラグを特定する文字列を返します。メッセージが標準 Maildir 形式に準拠しているならば、結果はアルファベット順に並べられたゼロまたは 1 回の 'D'、'F'、'P'、'R'、'S'、'T' をつなげたものです。空文字列が返されるのはフラグが一つもない場合、または ”info” が実験的セマンティクスを使っている場合です。

set_flags(*flags*)
flags で指定されたフラグをセットし、他のフラグは下ろします。

add_flag(*flag*)
flag で指定されたフラグをセットしますが他のフラグは変えません。一度に二つ以上のフラグをセットすることは、*flag* に 2 文字以上の文字列を指定すればできます。現在の ”info” はフラグの代わりに実験的情報を使っていても上書きされます。

remove_flag(*flag*)
flag で指定されたフラグを下ろしますが他のフラグは変えません。一度に二つ以上のフラグを取り除くことは、*flag* に 2 文字以上の文字列を指定すればできます。”info” がフラグの代わりに実験的情報を使っている場合は現在の ”info” は書き換えられません。

get_date()
メッセージの配送日時をエポックからの秒数を表わす浮動小数点数で返します。

set_date(*date*)
メッセージの配送日時を *date* にセットします。*date* はエポックからの秒数を表わす浮動小数点数です。

get_info()
メッセージの ”info” を含む文字列を返します。このメソッドは実験的 (即ちフラグのリストでない) ”info” にアクセスし、また変更するのに役立ちます。

set_info(*info*)
”info” に文字列 *info* をセットします。

MaildirMessage インスタンスが *mboxMessage* や *MMDFMessage* のインスタンスに基づいて生成されるとき、*Status* および *X-Status* ヘッダは省かれ以下の変換が行われます:

結果の状態	*mboxMessage* または *MMDFMessage* の状態
”cur” サブディレクトリ	O フラグ
F フラグ	F フラグ
R フラグ	A フラグ
S フラグ	R フラグ
T フラグ	D フラグ

MaildirMessage インスタンスが *MHMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	*MHMessage* の状態
”cur” サブディレクトリ	”unseen” シーケンス
”cur” サブディレクトリおよび S フラグ	”unseen” シーケンス無し
F フラグ	”flagged” シーケンス
R フラグ	”replied” シーケンス

MaildirMessage インスタンスが *BabylMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	*BabylMessage* の状態
”cur” サブディレクトリ	”unseen” ラベル
”cur” サブディレクトリおよび S フラグ	”unseen” ラベル無し
P フラグ	”forwarded” または ”resent” ラベル
R フラグ	”answered” ラベル
T フラグ	”deleted” ラベル

mboxMessage

class mailbox.**mboxMessage**(*message=None*)

mbox 固有の動作をするメッセージ。引数 *message* は *Message* のコンストラクタと同じ意味を持ちます。

mbox メールボックス中のメッセージは単一ファイルにまとめて格納されています。送り主のエンベロープアドレスおよび配送日時は通常メッセージの開始を示す "From " から始まる行に記録されますが、正確なフォーマットに関しては mbox の実装ごとに大きな違いがあります。メッセージの状態を示すフラグ、たとえば読んだかどうかあるいは重要だとマークを付けられているかどうかといったようなもの、は典型的には ***Status*** および ***X-Status*** に収められます。

規定されている mbox メッセージのフラグは以下の通りです:

Flag	意味	説明
R	読んだもの	読んだもの
O	古い (Old)	以前に MUA に発見された
D	削除 (Deleted)	削除予定とされたもの
F	フラグ付き (Flagged)	重要とされたもの
A	返答済み (Answered)	返答されたもの

"R" および "O" フラグは ***Status*** ヘッダに記録され、"D"、"F"、"A" フラグは ***X-Status*** ヘッダに記録されます。フラグとヘッダは通常記述された順番に出現します。

mboxMessage インスタンスは以下のメソッドを提供します:

get_from()

mbox メールボックスのメッセージの開始を示す "From " 行を表わす文字列を返します。先頭の "From " および末尾の改行は含まれません。

set_from(*from_*, *time_=None*)

"From " 行を *from_* にセットします。*from_* は先頭の "From " や末尾の改行を含まない形で指定しなければなりません。利便性のために、*time_* を指定して適切に整形して *from_* に追加させることができます。*time_* を指定する場合、それは *time.struct_time* インスタンス、*time.strftime()* に渡すのに適したタプル、または `True` (この場合 *time.gmtime()* を使います) のいずれかでなければなりません。

get_flags()

現在セットされているフラグを特定する文字列を返します。メッセージが規定された形式に準拠しているならば、結果は次の順に並べられた 0 回か 1 回の `'R'`、`'O'`、`'D'`、`'F'`、`'A'` です。

set_flags(*flags*)

flags で指定されたフラグをセットして、他のフラグは下ろします。*flags* は並べられたゼロまたは 1 回の `'R'`、`'O'`、`'D'`、`'F'`、`'A'` です。

add_flag(*flag*)

flag で指定されたフラグをセットしますが他のフラグは変えません。一度に二つ以上のフラグをセットすることは、*flag* に 2 文字以上の文字列を指定すればできます。

remove_flag(*flag*)

flag で指定されたフラグを下ろしますが他のフラグは変えません。一二つ以上のフラグを取り除くことは、*flag* に 2 文字以上の文字列を指定すればできます。

mboxMessage インスタンスが *MaildirMessage* インスタンスに基づいて生成されるとき、*MaildirMessage* インスタンスの配送日時に基づいて "From " 行が作り出され、次の変換が行われます:

結果の状態	*MaildirMessage* の状態
R フラグ	S フラグ
O フラグ	"cur" サブディレクトリ
D フラグ	T フラグ
F フラグ	F フラグ
A フラグ	R フラグ

mboxMessage インスタンスが *MHMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	*MHMessage* の状態
R フラグおよび O フラグ	"unseen" シーケンス無し
O フラグ	"unseen" シーケンス
F フラグ	"flagged" シーケンス
A フラグ	"replied" シーケンス

mboxMessage インスタンスが *BabylMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	*BabylMessage* の状態
R フラグおよび O フラグ	"unseen" ラベル無し
O フラグ	"unseen" ラベル
D フラグ	"deleted" ラベル
A フラグ	"answered" ラベル

mboxMessage インスタンスが *MMDFMessage* インスタンスに基づいて生成されるとき、"From " 行はコピーされ全てのフラグは直接対応します:

結果の状態	*MMDFMessage* の状態
R フラグ	R フラグ
O フラグ	O フラグ
D フラグ	D フラグ
F フラグ	F フラグ
A フラグ	A フラグ

MHMessage

class mailbox.**MHMessage**(*message=None*)

MH 固有の動作をするメッセージ。引数 *message* は *Message* のコンストラクタと同じ意味を持ちます。

MH メッセージは伝統的な意味あいにおいてマークやフラグをサポートしません。しかし、MH メッセージにはシーケンスがあり任意のメッセージを論理的にグループ分けできます。いくつかのメールソフト (標準の **mh** や **nmh** はそうではありませんが) は他の形式におけるフラグとほぼ同じようにシーケンスを使います:

シーケンス	説明
unseen	読んではいないが既に MUA に見つけられている
replied	返答されたもの
flagged	重要とされたもの

MHMessage インスタンスは以下のメソッドを提供します:

get_sequences()

このメッセージを含むシーケンスの名前のリストを返す。

set_sequences(*sequences*)

このメッセージを含むシーケンスのリストをセットする。

add_sequence(*sequence*)

sequence をこのメッセージを含むシーケンスのリストに追加する。

remove_sequence(*sequence*)

sequence をこのメッセージを含むシーケンスのリストから除く。

MHMessage インスタンスが *MaildirMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	*MaildirMessage* の状態
"unseen" シーケンス	S フラグ無し
"replied" シーケンス	R フラグ
"flagged" シーケンス	F フラグ

MHMessage インスタンスが *mboxMessage* や *MMDFMessage* のインスタンスに基づいて生成されるとき、*Status* および *X-Status* ヘッダは省かれ以下の変換が行われます:

結果の状態	*mboxMessage* または *MMDFMessage* の状態
"unseen" シーケンス	R フラグ無し
"replied" シーケンス	A フラグ
"flagged" シーケンス	F フラグ

MHMessage インスタンスが *BabylMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	*BabylMessage* の状態
"unseen" シーケンス	"unseen" ラベル
"replied" シーケンス	"answered" ラベル

BabylMessage

class mailbox.**BabylMessage**(*message=None*)

Babyl 固有の動作をするメッセージ。引数 *message* は *Message* のコンストラクタと同じ意味を持ちます。

ある種のメッセージラベルは **アトリビュート** と呼ばれ、規約により特別な意味が与えられています。アトリビュートは以下の通りです:

ラベル	説明
unseen	読んではいないが既に MUA に見つけられている
deleted	削除予定とされたもの
filed	他のファイルまたはメールボックスにコピーされた
answered	返答されたもの
forwarded	転送された
edited	ユーザによって変更された
resent	再送された

デフォルトでは Rmail は可視ヘッダのみ表示します。*BabylMessage* クラスはしかし、オリジナルヘッダをより完全だという理由で使います。可視ヘッダは望むならそのように指示してアクセスすることができます。

BabylMessage インスタンスは以下のメソッドを提供します:

get_labels()

メッセージに付いているラベルのリストを返します。

set_labels(*labels*)

メッセージに付いているラベルのリストを *labels* にセットします。

add_label(*label*)

メッセージに付いているラベルのリストに *label* を追加します。

remove_label(*label*)

メッセージに付いているラベルのリストから *label* を削除します。

get_visible()

ヘッダがメッセージの可視ヘッダでありボディが空であるような *Message* インスタンスを返します。

set_visible(*visible*)

メッセージの可視ヘッダを *visible* のヘッダと同じにセットします。引数 *visible* は *Message* インスタンスまたは *email.message.Message* インスタンス、文字列、ファイル風オブジェクト (テキストモードで開かれてなければなりません) のいずれかです。

update_visible()

BabylMessage インスタンスのオリジナルヘッダが変更されたとき、可視ヘッダは自動的に対応して変更されるわけではありません。このメソッドは可視ヘッダを以下のように更新します。対応するオリジナルヘッダのある可視ヘッダはオリジナルヘッダの値がセットされます。対応するオリジナルヘッダの無い可視ヘッダは除去されます。そして、オリジナルヘッダにあって可視ヘッダに無い *Date* 、*From* 、*Reply-To* 、*To* 、*CC* 、*Subject* は可視ヘッダに追加されます。

BabylMessage インスタンスが *MaildirMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	*MaildirMessage* の状態
"unseen" ラベル	S フラグ無し
"deleted" ラベル	T フラグ
"answered" ラベル	R フラグ
"forwarded" ラベル	P フラグ

BabylMessage インスタンスが *mboxMessage* や *MMDFMessage* のインスタンスに基づいて生成されるとき、*Status* および *X-Status* ヘッダは省かれ以下の変換が行われます:

結果の状態	*mboxMessage* または *MMDFMessage* の状態
"unseen" ラベル	R フラグ無し
"deleted" ラベル	D フラグ
"answered" ラベル	A フラグ

BabylMessage インスタンスが *MHMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	*MHMessage* の状態
"unseen" ラベル	"unseen" シーケンス
"answered" ラベル	"replied" シーケンス

MMDFMessage

class mailbox.**MMDFMessage**(*message=None*)

MMDF 固有の動作をするメッセージ。引数 *message* は *Message* のコンストラクタと同じ意味を持ちます。

mbox メールボックスのメッセージと同様に、MMDF メッセージは送り主のアドレスと配送日時が最初の "From " で始まる行に記録されています。同様に、メッセージの状態を示すフラグは通常 *Status* および *X-Status* ヘッダに収められています。

よく使われる MMDF メッセージのフラグは mbox メッセージのものと同一で以下の通りです:

Flag	意味	説明
R	読んだもの	読んだもの
O	古い (Old)	以前に MUA に発見された
D	削除 (Deleted)	削除予定とされたもの
F	フラグ付き (Flagged)	重要とされたもの
A	返答済み (Answered)	返答されたもの

"R" および "O" フラグは *Status* ヘッダに記録され、"D"、"F"、"A" フラグは *X-Status* ヘッダに記録されます。フラグとヘッダは通常記述された順番に出現します。

MMDFMessage インスタンスは *mboxMessage* インスタンスと同一の以下のメソッドを提供します:

get_from()

mbox メールボックスのメッセージの開始を示す "From " 行を表わす文字列を返します。先頭の "From " および末尾の改行は含まれません。

set_from(*from_*, *time_=None*)

"From " 行を *from_* にセットします。*from_* は先頭の "From " や末尾の改行を含まない形で指定しなければなりません。利便性のために、*time_* を指定して適切に整形して *from_* に追加させることができます。*time_* を指定する場合、それは *time.struct_time* インスタンス、*time.strftime()* に渡すのに適したタプル、または `True` (この場合 *time.gmtime()* を使います) のいずれかでなければなりません。

get_flags()

現在セットされているフラグを特定する文字列を返します。メッセージが規定された形式に準拠しているならば、結果は次の順に並べられた 0 回か 1 回の `'R'`、`'O'`、`'D'`、`'F'`、`'A'` です。

set_flags(*flags*)

flags で指定されたフラグをセットして、他のフラグは下ろします。*flags* は並べられたゼロまたは 1 回の `'R'`、`'O'`、`'D'`、`'F'`、`'A'` です。

add_flag(*flag*)

flag で指定されたフラグをセットしますが他のフラグは変えません。一度に二つ以上のフラグをセットすることは、*flag* に 2 文字以上の文字列を指定すればできます。

`remove_flag`(*flag*)

flag で指定されたフラグを下ろしますが他のフラグは変えません。一二つ以上のフラグを取り除くことは、*flag* に 2 文字以上の文字列を指定すればできます。

MMDFMessage インスタンスが *MaildirMessage* インスタンスに基づいて生成されるとき、”From” 行が *MaildirMessage* インスタンスの配信日をもとに生成され、以下の変換が行われます:

結果の状態	*MaildirMessage* の状態
R フラグ	S フラグ
O フラグ	”cur” サブディレクトリ
D フラグ	T フラグ
F フラグ	F フラグ
A フラグ	R フラグ

MMDFMessage インスタンスが *MHMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	*MHMessage* の状態
R フラグおよび O フラグ	”unseen” シーケンス無し
O フラグ	”unseen” シーケンス
F フラグ	”flagged” シーケンス
A フラグ	”replied” シーケンス

MMDFMessage インスタンスが *BabylMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	*BabylMessage* の状態
R フラグおよび O フラグ	”unseen” ラベル無し
O フラグ	”unseen” ラベル
D フラグ	”deleted” ラベル
A フラグ	”answered” ラベル

MMDFMessage インスタンスが *mboxMessage* インスタンスに基づいて生成されるとき、”From” 行がコピーされ、全てのフラグが直接対応します:

結果の状態	*mboxMessage* の状態
R フラグ	R フラグ
O フラグ	O フラグ
D フラグ	D フラグ
F フラグ	F フラグ
A フラグ	A フラグ

19.4.3 例外

mailbox モジュールでは以下の例外クラスが定義されています:

exception mailbox.**Error**
他の全てのモジュール固有の例外の基底クラス。

exception mailbox.**NoSuchMailboxError**
メールボックスがあると思っていたが見つからなかった場合に送出されます。これはたとえば *Mailbox* のサブクラスを存在しないパスでインスタンス化しようとしたとき (かつ *create* パラメータは `False` であった場合)、あるいは存在しないフォルダを開こうとした時などに発生します。

exception mailbox.**NotEmptyError**
メールボックスが空であることを期待されているときに空でない場合、たとえばメッセージの残っているフォルダを削除しようとした時などに送出されます。

exception mailbox.**ExternalClashError**
メールボックスに関係したある条件がプログラムの制御を外れてそれ以上作業を続けられなくなった場合、たとえば他のプログラムが既に保持しているロックを取得しようとして失敗したとき、あるいは一意的に生成されたファイル名が既に存在していた場合などに送出されます。

exception mailbox.**FormatError**
ファイル中のデータが解析できない場合、たとえば *MH* インスタンスが壊れた `.mh_sequences` ファイルを読もうと試みた場合などに送出されます。

19.4.4 使用例

メールボックス中の面白そうなメッセージのサブジェクトを全て印字する簡単な例:

```
import mailbox
for message in mailbox.mbox('~/mbox'):
    subject = message['subject']       # Could possibly be None.
    if subject and 'python' in subject.lower():
        print(subject)
```

Babyl メールボックスから MH メールボックスへ全てのメールをコピーし、変換可能な全ての形式固有の情報を変換する:

```
import mailbox
destination = mailbox.MH('~/Mail')
destination.lock()
for message in mailbox.Babyl('~/RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

この例は幾つかのメーリングリストのメールをソートするものです。他のプログラムと平行して変更を加えることでメールが破損したり、プログラムを中断することでメールを失ったり、はたまた半端なメッセージが

メールボックス中にあることで途中で終了してしまう、といったことを避けるように注意深く扱っています:

```
import mailbox
import email.errors

list_names = ('python-list', 'python-dev', 'python-bugs')

boxes = {name: mailbox.mbox('~/email/%s' % name) for name in list_names}
inbox = mailbox.Maildir('~/Maildir', factory=None)

for key in inbox.iterkeys():
    try:
        message = inbox[key]
    except email.errors.MessageParseError:
        continue                # The message is malformed. Just leave it.

    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            # Get mailbox to use
            box = boxes[name]

            # Write copy to disk before removing original.
            # If there's a crash, you might duplicate a message, but
            # that's better than losing a message completely.
            box.lock()
            box.add(message)
            box.flush()
            box.unlock()

            # Remove original message
            inbox.lock()
            inbox.discard(key)
            inbox.flush()
            inbox.unlock()
            break               # Found destination, so stop looking.

for box in boxes.itervalues():
    box.close()
```

19.5 mimetypes --- ファイル名を MIME 型へマップする

ソースコード: Lib/mimetypes.py

mimetypes モジュールは、ファイル名あるいは URL と、ファイル名拡張子に関連付けられた MIME 型とを変換します。ファイル名から MIME 型へと、MIME 型からファイル名拡張子への変換が提供されます; 後者の変換では符号化方式はサポートされていません。

このモジュールは、一つのクラスと多くの便利な関数を提供します。これらの関数がこのモジュールへの標準のインターフェースですが、アプリケーションによっては、そのクラスにも関係するかもしれません。

以下で説明されている関数は、このモジュールへの主要なインターフェースを提供します。たとえモジュールが初期化されていなくても、もしこれらの関数が、*init()* がセットアップする情報に依存していれば、これらの関数は、*init()* を呼びます。

mimetypes.**guess_type**(*url*, *strict=True*)

url で与えられるファイル名あるいは URL に基づいて、ファイルの型を推定します。URL は文字列または *path-like object* です。

戻り値は、タプル (type, encoding) です、ここで *type* は、もし型が (拡張子がないあるいは未定義のため) 推定できない場合は、None を、あるいは、MIME *content-type* ヘッダ に利用できる、'type/subtype' の形の文字列です。

encoding は、符合化方式がない場合は None を、あるいは、符号化に使われるプログラムの名前 (たとえば、**compress** あるいは **gzip**) です。符号化方式は *Content-Encoding* ヘッダとして使うのに適しており、*Content-Transfer-Encoding* ヘッダには適して **いません** 。マッピングはテーブル駆動です。符号化方式のサフィックスは大/小文字を区別します; データ型サフィックスは、最初大/小文字を区別して試し、それから大/小文字を区別せずに試します。

省略可能な *strict* 引数は、既知の MIME 型のリストとして認識されるものが、IANA に登録された 正式な型のみに限定されるかどうかを指定するフラグです。*strict* が True (デフォルト) の時は、IANA 型のみがサポートされます; *strict* が False のときは、いくつかの追加の、非標準ではあるが、一般的に使用される MIME 型も認識されます。

バージョン 3.8 で変更: url に *path-like object* のサポートが追加されました。

mimetypes.**guess_all_extensions**(*type*, *strict=True*)

type で与えられる MIME 型に基づいてファイルの拡張子を推定します。戻り値は、先頭のドット ('.') を含む、可能なファイル拡張子すべてを与える文字列のリストです。拡張子と特別なデータストリームとの関連付けは保証されませんが、*guess_type()* によって MIME 型 *type* とマップされます。

省略可能な *strict* 引数は *guess_type()* 関数のものと同じ意味を持ちます。

mimetypes.**guess_extension**(*type*, *strict=True*)

type で与えられる MIME 型に基づいてファイルの拡張子を推定します。戻り値は、先頭のドット ('.') を含む、ファイル拡張子を与える文字列のリストです。拡張子と特別なデータストリームとの関連付けは保証されませんが、*guess_type()* によって MIME 型 *type* とマップされます。もし *type* に対して拡張子が推定できない場合は、None が返されます。

省略可能な *strict* 引数は *guess_type()* 関数のものと同じ意味を持ちます。

モジュールの動作を制御するために、いくつかの追加の関数とデータ項目が利用できます。

mimetypes.**init**(*files=None*)

内部のデータ構造を初期化します。もし *files* が与えられていれば、これはデフォルトの type map を増やすために使われる、一連のファイル名でなければなりません。もし省略されていれば、使われるファイル名は *knownfiles* から取られます。Windows であれば、現在のレジストリの設定が読み込まれます。*files* あるいは *knownfiles* 内の各ファイル名は、それ以前に現れる名前より優先されます。繰り返し *init()* を呼び出すことは許されています。

files に空リストを与えることで、システムのデフォルトが適用されるのを避けることが出来ます; 組み込みのリストから well-known な値だけが取り込まれます。

files が None の場合、内部のデータ構造は初期のデフォルト値に完全に再構築されます。これは安定な操作であり、複数回呼び出されたときは同じ結果になります。

バージョン 3.2 で変更: 前のバージョンでは、Windows のレジストリの設定は無視されていました。

mimetypes.**read_mime_types**(*filename*)

ファイル *filename* で与えられた型のマップが、もしあればロードします。型のマップは、先頭の dot ('.') を含むファイル名拡張子を、'type/subtype' の形の文字列にマッピングする辞書として返されます。もしファイル *filename* が存在しないか、読み込めなければ、None が返されます。

mimetypes.**add_type**(*type*, *ext*, *strict=True*)

MIME 型 *type* からのマッピングを拡張子 *ext* に追加します。拡張子がすでに既知であれば、新しい型が古いものに置き替わります。その型がすでに既知であれば、その拡張子が、既知の拡張子のリストに追加されます。

strict が True の時 (デフォルト) は、そのマッピングは正式な MIME 型に、そうでなければ、非標準の MIME 型に追加されます。

mimetypes.**inited**

グローバルなデータ構造が初期化されているかどうかを示すフラグ。これは *init()* により True に設定されます。

mimetypes.**knownfiles**

共通にインストールされた型マップファイル名のリスト。これらのファイルは、普通 mime.types という名前であり、パッケージごとに異なる場所にインストールされます。

mimetypes.**suffix_map**

サフィックスをサフィックスにマップする辞書。これは、符号化方式と型が同一拡張子で示される符号化ファイルが認識できるように使用されます。例えば、.tgz 拡張子は、符号化と型が別個に認識できるように .tar.gz にマップされます。

mimetypes.**encodings_map**

ファイル名拡張子を符号化方式型にマッピングする辞書。

mimetypes.**types_map**

ファイル名拡張子を MIME 型にマップする辞書。

mimetypes.**common_types**

ファイル名拡張子を非標準ではあるが、一般に使われている MIME 型にマップする辞書。

モジュールの使用例:

```
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
```

(次のページに続く)

(前のページからの続き)

```
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

19.5.1 Mime 型オブジェクト

MimeTypes クラスは一つ以上の MIME 型データベースが欲しいアプリケーションにとって有用でしょう。これは *mimetypes* モジュールのそれと似たインターフェースを提供します。

class mimetypes.**MimeTypes**(*filenames=()*, *strict=True*)

このクラスは、MIME 型データベースを表現します。デフォルトでは、このモジュールの他のものと同じデータベースへのアクセスを提供します。初期データベースは、このモジュールによって提供されるもののコピーで、追加の `mime.types` 形式のファイルを、*read()* あるいは *readfp()* メソッドを使って、データベースにロードすることで拡張されます。マッピング辞書も、もしデフォルトのデータが望むものでなければ、追加のデータをロードする前にクリアされます。

省略可能な *filenames* パラメータは、追加のファイルを、デフォルトデータベースの" トップに" ロードさせるのに使うことができます。

suffix_map

サフィックスをサフィックスにマップする辞書。これは、符号化方式と型が同一拡張子で示されるような符号化ファイルが認識できるように使用されます。例えば、`.tgz` 拡張子は、符号化方式と型が別個に認識できるように `.tar.gz` に対応づけられます。これは、最初はモジュールで定義されたグローバルな *suffix_map* のコピーです。

encodings_map

ファイル名拡張子を符号化型にマッピングする辞書。これは、最初はモジュールで定義されたグローバルな *encodings_map* のコピーです。

types_map

ファイル名拡張子を MIME 型にマッピングする 2 種類の辞書のタプル; 最初の辞書は非標準型、二つ目は標準型の辞書です。初期状態ではそれぞれ *common_types* と *types_map* です。

types_map_inv

MIME 型をファイル名拡張子のリストにマッピングする 2 種類の辞書のタプル; 最初の辞書は非標準型、二つ目は標準型の辞書です。初期状態ではそれぞれ *common_types* と *types_map* です。

guess_extension(*type*, *strict=True*)

guess_extension() 関数と同様ですが、オブジェクトに保存されたテーブルを使用します。

guess_type(*url*, *strict=True*)

guess_type() 関数と同様ですが、オブジェクトに保存されたテーブルを使用します。

guess_all_extensions(*type*, *strict=True*)

guess_all_extensions() と同様ですが、オブジェクトに保存されたテーブルを参照します。

read(*filename*, *strict=True*)

MIME 情報を、*filename* という名のファイルからロードします。これはファイルを解析するのに *readfp()* を使用します。

strict が True の時 (デフォルト) は、そのマッピングは標準 MIME 型のリストに、そうでなければ、非標準 MIME 型のリストに追加されます。

readfp(*fp*, *strict=True*)

MIME 型情報を、オープンしたファイル *fp* からロードします。ファイルは、標準の mime.types ファイルの形式でなければなりません。

strict が True の時 (デフォルト) は、そのマッピングは標準 MIME 型のリストに、そうでなければ、非標準 MIME 型のリストに追加されます。

read_windows_registry(*strict=True*)

MIME type 情報を Windows のレジストリから読み込みます。

利用可能な環境: Windows 。

strict が True の時 (デフォルト) は、そのマッピングは標準 MIME 型のリストに、そうでなければ、非標準 MIME 型のリストに追加されます。

バージョン 3.2 で追加.

19.6 base64 --- Base16, Base32, Base64, Base85 データの符号化

ソースコード: Lib/base64.py

このモジュールはバイナリデータを印字可能な ASCII にエンコード関数、およびそのようなエンコードデータをバイナリにデコードする関数を提供します。それらは、**RFC 3548** が定義する Base16, Base32, Base64 のエンコーディング、デファクトスタンダードになっている Ascii85, Base85 のエンコーディングについてのエンコード、デコード関数です。

RFC 3548 エンコーディングは、email で安全に送信したり、URL の一部として使ったり、あるいは HTTP POST リクエストの一部に含めるために用いるのに適しています。このエンコーディングで使われているアルゴリズムは **uuencode** プログラムで用いられているものとは同じではありません。

このモジュールは、2 つのインターフェースを提供します。このモダンなインターフェースは、*bytes-like object* を ASCII *bytes* にエンコードし、*bytes-like object* か ASCII 文字列を、*bytes* にデコードすることができます。**RFC 3548** に定義されている base-64 アルファベット (一般の、URL あるいはファイルシステムセーフなもの) の両方が使用できます。

従来のインターフェースは文字列からのデコードができませんが、*file object* との間のエンコードとデコードが可能な関数を提供します。これは標準の base64 アルファベットのみをサポートし、**RFC 2045** の規定にあるように、76 文字ごとに改行されます。**RFC 2045** のサポートのためには、代わりに *email* パッケージを参照する必要があるかもしれません。

バージョン 3.3 で変更: モダンなインターフェイスのデコード関数が ASCII のみの Unicode 文字列を受け付けるようになりました。

バージョン 3.4 で変更: このモジュールのすべてのエンコード・デコード関数が任意の *bytes-like* **オブジェクト** を受け取るようになりました。Ascii85/Base85 のサポートが追加されました。

モダンなインターフェイスは以下のものを提供します:

`base64.b64encode(s, altchars=None)`

Base64 を使って *bytes-like object* の *s* をエンコードし、エンコードされた *bytes* を返します。

オプション引数 *altchars* は最低でも 2 の長さをもつ *bytes-like object* で (これ以降の文字は無視されます)、これは `+` と `/` の代わりに使われる代替アルファベットを指定します。これにより、アプリケーションはたとえば URL やファイルシステムの影響をうけない Base64 文字列を生成することができます。デフォルトの値は `None` で、これは標準の Base64 アルファベット集合が使われることを意味します。

`base64.b64decode(s, altchars=None, validate=False)`

Base64 エンコードされた *bytes-like object* または ASCII 文字列 *s* をデコードし、デコードされた *bytes* を返します。

オプション引数の *altchars* は最低でも 2 の長さをもつ *bytes-like object* または ASCII 文字列で (これ以降の文字は無視されます)、これは `+` と `/` の代わりに使われる代替アルファベットを指定します。

s が正しくパディングされていない場合は *binascii.Error* 例外を発生させます。

validate が `False` (デフォルト) の場合、標準の base64 アルファベットでも代替文字でもない文字はパディングチェックの前に無視されます。*validate* が `True` の場合、入力に base64 アルファベット以外の文字があると *binascii.Error* を発生させます。

`base64.standard_b64encode(s)`

標準の base64 アルファベットを使用して *bytes-like object* の *s* をエンコードし、エンコードされた *bytes* を返します。

`base64.standard_b64decode(s)`

標準の base64 アルファベットを使用した *bytes-like object* または ASCII 文字列 *s* をデコードし、デコードされた *bytes* を返します。

`base64.urlsafe_b64encode(s)`

bytes-like object *s* を URL とファイルシステムセーフなアルファベットを利用してエンコードし、エンコードされた *bytes* を返します。標準 base64 アルファベットに比べて、`+` の替わりに `-` を、`/` の替わりに `_` を利用します。結果は `=` を含みます。

`base64.urlsafe_b64decode(s)`

bytes-like object または ASCII 文字列 *s* を URL とファイルシステムセーフなアルファベットを利用してデコードし、デコードされた *bytes* を返します。標準 base64 アルファベットに比べて、`+` の替わりに `-` を、`/` の替わりに `_` を置換します。

`base64.b32encode(s)`

Base32 を使って *bytes-like object* の *s* をエンコードし、エンコードされた *bytes* を返します。

base64.b32decode(*s*, *casefold=False*, *map01=None*)

Base32 エンコードされた *bytes-like object* または ASCII 文字列 *s* をデコードし、デコードされた *bytes* を返します。

オプション引数 *casefold* は小文字のアルファベットを受けつけるかどうかを指定します。セキュリティ上の理由により、デフォルトではこれは `False` になっています。

RFC 3548 は付加的なマッピングとして、数字の 0 (零) をアルファベットの O (オー) に、数字の 1 (壱) をアルファベットの I (アイ) または L (エル) に対応させることを許しています。オプション引数は *map01* は、`None` でないときは、数字の 1 をどの文字に対応づけるかを指定します (*map01* が `None` でないとき、数字の 0 はつねにアルファベットの O (オー) に対応づけられます)。セキュリティ上の理由により、これはデフォルトでは `None` になっているため、0 および 1 は入力として許可されていません。

s が正しくパディングされていない場合や、入力にアルファベットでない文字が含まれていた場合に、*binascii.Error* 例外を発生させます。

base64.b16encode(*s*)

Base16 を使って *bytes-like object* の *s* をエンコードし、エンコードされた *bytes* を返します。

base64.b16decode(*s*, *casefold=False*)

Base16 エンコードされた *bytes-like object* または ASCII 文字列 *s* をデコードし、デコードされた *bytes* を返します。

オプション引数 *casefold* は小文字のアルファベットを受けつけるかどうかを指定します。セキュリティ上の理由により、デフォルトではこれは `False` になっています。

s が正しくパディングされていない場合や、入力にアルファベットでない文字が含まれていた場合に、*binascii.Error* 例外を発生させます。

base64.a85encode(*b*, ***, *foldspaces=False*, *wrapcol=0*, *pad=False*, *adobe=False*)

Ascii85 を使って *bytes-like object* の *b* をエンコードし、エンコードされた *bytes* を返します。

foldspaces を使うと、4 つの連続した空白文字 (ASCII 0x20) を 'btoa' によってサポートされている短い特殊文字 'y' に置き換えます。この機能は " 標準" Ascii85 ではサポートされていません。

wrapcol は何文字ごとに改行文字 (`b'\n'`) を挿入するかを制御します。ゼロでない場合、出力の各行はこの与えられた文字数を超えません。

pad を指定すると、エンコード前に入力が 4 の倍数になるようにパディングされます。なお、`btoa` の実装は常にパディングします。

adobe を指定すると、エンコードしたバイトシーケンスを `<~` と `~>` で囲みます。これは Adobe の実装で使われています。

バージョン 3.4 で追加.

base64.a85decode(*b*, ***, *foldspaces=False*, *adobe=False*, *ignorechars=b' \t\n\r\v'*)

Ascii85 エンコードされた *bytes-like object* または ASCII 文字列 *b* をデコードし、デコードされた *bytes* を返します。

foldspaces は、短い特殊文字 'y' を受け取って 4 つの連続した空白文字 (ASCII 0x20) と解釈するかどうかを制御します。この機能は " 標準" Ascii85 ではサポートされていません。

adobe で、入力シーケンスが Adobe Ascii85 (つまり `<~` と `~>` で囲まれている) かどうかを伝えます。

ignorechars には、入力に含まれていれば無視する文字で構成された *bytes-like object* または ASCII 文字列を指定してください。これは空白文字だけで構成されているべきです。デフォルトは ASCII における空白文字全てです。

バージョン 3.4 で追加.

`base64.b85encode`(*b*, *pad=False*)

base85 (これは例えば git スタイルのバイナリ diff で用いられています) を使って *bytes-like object* の *b* をエンコードし、エンコードされた *bytes* を返します。

pad が真ならば、エンコードに先立って、バイト数が 4 の倍数となるように入力が `b'\0'` でパディングされます。

バージョン 3.4 で追加.

`base64.b85decode`(*b*)

base85 でエンコードされた *bytes-like object* または ASCII 文字列の *b* をデコードし、デコードされた *bytes* を返します。パディングは、もしあれば、暗黙に削除されます。

バージョン 3.4 で追加.

レガシーなインターフェイスは以下のものを提供します:

`base64.decode`(*input*, *output*)

input ファイルの中身をデコードし、結果のバイナリデータを *output* ファイルに出力します。*input* 、*output* ともに *file objects* でなければなりません。*input* は `input.readline()` が空バイト列を返すまで読まれます。

`base64.decodebytes`(*s*)

bytes-like object *s* をデコードし、デコードされた *bytes* を返します。*s* には一行以上の base64 形式でエンコードされたデータが含まれている必要があります。

バージョン 3.1 で追加.

`base64.decodestring`(*s*)

`decodestring` は廃止されたエイリアスです。

バージョン 3.1 で非推奨.

`base64.encode`(*input*, *output*)

バイナリの *input* ファイルの中身を base64 形式でエンコードした結果を *output* ファイルに出力します。*input* 、*output* ともに *file objects* でなければなりません。*input* は `input.read()` が空バイト列を返すまで読まれます。*encode()* は 76 バイトの出力ごとに改行文字 (`b'\n'`) を挿入し、**RFC 2045** (MIME) の規定にあるように常に出力が新しい行で終わることを保証します。

base64.**encodebytes**(*s*)

bytes-like object *s* (任意のバイナリデータを含むことができます) を、**RFC 2045** (MIME) に規定されるように末尾に新しい行のある、76 バイトの出力ごとに新しい行 (`b'\n'`) が挿入された、base64 形式でエンコードしたデータを含む *bytes* を返します。

バージョン 3.1 で追加.

base64.**encodestring**(*s*)

encodebytes() の廃止されたエイリアスです。

バージョン 3.1 で非推奨.

モジュールの使用例:

```
>>> import base64
>>> encoded = base64.b64encode(b'data to be encoded')
>>> encoded
b'ZGF0YSBOobyBiZSBlbmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
b'data to be encoded'
```

参考:

モジュール *binascii* ASCII からバイナリへ、バイナリから ASCII への変換をサポートするモジュール。

RFC 1521 - MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format
Section 5.2, "Base64 Content-Transfer-Encoding," provides the definition of the base64 encoding.

19.7 binhex --- binhex4 形式ファイルのエンコードおよびデコード

ソースコード: Lib/binhex.py

このモジュールは binhex4 形式のファイルに対するエンコードやデコードを行います。binhex4 は Macintosh のファイルを ASCII で表現できるようにしたものです。データフォークだけが処理されます。

binhex モジュールでは以下の関数を定義しています:

binhex.**binhex**(*input*, *output*)

ファイル名 *input* のバイナリファイルをファイル名 *output* の binhex 形式ファイルに変換します。*output* パラメタはファイル名でも (`write()` および `close()` メソッドをサポートするような) ファイル様オブジェクトでもかまいません。

binhex.**hexbin**(*input*, *output*)

binhex 形式のファイル *input* をデコードします。*input* はファイル名でも、`read()` および `close()` メソッドをサポートするようなファイル様オブジェクトでもかまいません。変換結果のファイルはファイル名 *output* になります。この引数が `None` なら、出力ファイルは binhex ファイルの中から復元されます。

以下の例外も定義されています:

exception binhex.Error

binhex 形式を使ってエンコードできなかった場合 (例えば、ファイル名が filename フィールドに収まらないくらい長かった場合など) や、入力が正しくエンコードされた binhex 形式のデータでなかった場合に送出される例外です。

参考:

モジュール *binascii* ASCII からバイナリへ、バイナリから ASCII への変換をサポートするモジュール。

19.7.1 注釈

別のより強力なエンコーダおよびデコーダへのインタフェースが存在します。詳しくはソースを参照してください。

非 Macintosh プラットフォームでテキストファイルをエンコードしたりデコードしたりする場合でも、古い Macintosh の改行文字変換 (行末をキャリッジリターンとする) が行われます。

19.8 binascii --- バイナリデータと ASCII データとの間での変換

---

binascii モジュールにはバイナリと ASCII コード化されたバイナリ表現との間の変換を行うための多数のメソッドが含まれています。通常、これらの関数を直接使う必要はなく、*uu* 、*base64* や *binhex* といった、ラッパ (wrapper) モジュールを使うことになるでしょう。*binascii* モジュールは C で書かれた高速な低水準関数を提供していて、それらは上記の高水準なモジュールで利用されます。

注釈: a2b_* 関数は ASCII 文字だけを含むユニコード文字列を受け取ります。他の関数は (*bytes* や *bytearray* またはバッファープロトコルをサポートするその他のオブジェクトのような) *bytes-like* **オブジェクト** だけを受け取ります。

バージョン 3.3 で変更: a2b_* 関数は ASCII のみのユニコード文字列を受け取るようになりました。

---

binascii モジュールでは以下の関数を定義します:

binascii.a2b_uu(*string*)

uuencode された 1 行のデータをバイナリに変換し、変換後のバイナリデータを返します。最後の行を除いて、通常 1 行には (バイナリデータで) 45 バイトが含まれます。入力データの先頭には空白文字が連続していてもかまいません。

binascii.b2a_uu(*data*, ***, *backtick=False*)

Convert binary data to a line of ASCII characters, the return value is the converted line, including a newline char. The length of *data* should be at most 45. If *backtick* is true, zeros are represented by '`' instead of spaces.

バージョン 3.7 で変更: *backtick* パラメータを追加しました.

binascii.a2b_base64(*string*)

base64 でエンコードされたデータのブロックをバイナリに変換し、変換後のバイナリデータを返します。一度に 1 行以上のデータを与えてもかまいません。

binascii.b2a_base64(*data*, ***, *newline=True*)

バイナリデータを base64 でエンコードされた 1 行の ASCII 文字列に変換します。戻り値は変換後の 1 行の文字列で、*newline* が真の場合改行文字を含みます。この関数の出力は **RFC 3548** を遵守します。

バージョン 3.6 で変更: パラメータに *newline* を追加しました。

binascii.a2b_qp(*data*, *header=False*)

quoted-printable 形式のデータをバイナリに変換し、バイナリデータを返します。一度に 1 行以上のデータを渡すことができます。オプション引数 *header* が与えられており、かつその値が真であれば、アンダースコアは空白文字にデコードされます。

binascii.b2a_qp(*data*, *quotetabs=False*, *istext=True*, *header=False*)

バイナリデータを quoted-printable 形式でエンコードして 1 行から複数行の ASCII 文字列に変換します。変換後の文字列を返します。オプション引数 *quptetabs* が存在し、かつその値が真であれば、全てのタブおよび空白文字もエンコードされます。オプション引数 *istext* が存在し、かつその値が真であれば、改行はエンコードされませんが、行末の空白文字はエンコードされます。オプション引数 *header* が存在し、かつその値が真である場合、空白文字は **RFC 1522** にしたがってアンダースコアにエンコードされます。オプション引数 *header* が存在し、かつその値が偽である場合、改行文字も同様にエンコードされます。そうでない場合、復帰 (linefeed) 文字の変換によってバイナリデータストリームが破損してしまうかもしれません。

binascii.a2b_hqx(*string*)

binhex4 形式の ASCII 文字列データを RLE 展開を行わないでバイナリに変換します。文字列はバイナリのバイトデータを完全に含むような長さか、または (binhex4 データの最後の部分の場合) 余白のビットがゼロになっていなければなりません。

binascii.rledecode_hqx(*data*)

data に対し、binhex4 標準に従って RLE 展開を行います。このアルゴリズムでは、あるバイトの後ろに `0x90` がきた場合、そのバイトの反復を指示しており、さらにその後ろに反復カウントが続きます。カウントが `0` の場合 `0x90` 自体を示します。このルーチンは入力データの末端における反復指定が不完全でないかぎり解凍されたデータを返しますが、不完全な場合、例外 *Incomplete* が送出されます。

バージョン 3.2 で変更: 入力として bytestring または bytearray オブジェクトのみを受け取ります。

binascii.rlecode_hqx(*data*)

binhex4 方式の RLE 圧縮を *data* に対して行い、その結果を返します。

binascii.b2a_hqx(*data*)

バイナリを hexbin4 エンコードして ASCII 文字列に変換し、変換後の文字列を返します。引数の *data* はすでに RLE エンコードされていなければならず、その長さは (最後のフラグメントを除いて) 3 で割り切れなければなりません。

binascii.**crc_hqx**(*data*, *value*)

value を CRC の初期値として *data* の 16 ビット CRC 値を計算し、その結果を返します。この関数は、よく 0x1021 と表現される CRC-CCITT 多項式 $x^{16} + x^{12} + x^5 + 1$ を使います。この CRC は binhex4 形式で使われています。

binascii.**crc32**(*data*[, *value*])

32 ビットチェックサムである CRC-32 を *data* に対して計算します。crc の初期値は *value* です。デフォルトの CRC の初期値はゼロです。このアルゴリズムは ZIP ファイルのチェックサムと同じです。このアルゴリズムはチェックサムアルゴリズムとして設計されたもので、一般的なハッシュアルゴリズムには向きません。以下のようにして使います:

```
print(binascii.crc32(b"hello world"))
# Or, in two pieces:
crc = binascii.crc32(b"hello")
crc = binascii.crc32(b" world", crc)
print('crc32 = {:#010x}'.format(crc))
```

バージョン 3.0 で変更: 結果は常に unsigned です。すべてのバージョンとプラットフォームの Python に渡って同一の数値を生成するには、`crc32(data) & 0xffffffff` を使用します。

binascii.**b2a_hex**(*data*[, *sep*[, *bytes_per_sep=1*]])
binascii.**hexlify**(*data*[, *sep*[, *bytes_per_sep=1*]])

バイナリ *data* の 16 進表現を返します。*data* の各バイトは、対応する 2 桁の 16 進表現に変換されます。したがって、返されるバイトオブジェクトは *data* の 2 倍の長さになります。

Similar functionality (but returning a text string) is also conveniently accessible using the *bytes.hex()* method.

If *sep* is specified, it must be a single character str or bytes object. It will be inserted in the output after every *bytes_per_sep* input bytes. Separator placement is counted from the right end of the output by default, if you wish to count from the left, supply a negative *bytes_per_sep* value.

```
>>> import binascii
>>> binascii.b2a_hex(b'\xb9\x01\xef')
b'b901ef'
>>> binascii.hexlify(b'\xb9\x01\xef', '-')
b'b9-01-ef'
>>> binascii.b2a_hex(b'\xb9\x01\xef', b'_', 2)
b'b9_01ef'
>>> binascii.b2a_hex(b'\xb9\x01\xef', b' ', -2)
b'b901 ef'
```

バージョン 3.8 で変更: 引数 *sep* と *bytes_per_sep* が追加されました.

binascii.**a2b_hex**(*hexstr*)
binascii.**unhexlify**(*hexstr*)

16 進数表記の文字列 *hexstr* の表すバイナリデータを返します。この関数は *b2a_hex()* の逆です。*hexstr* は 16 進数字 (大文字でも小文字でも構いません) を偶数個含んでいなければなりません。そうでない場合、例外 *Error* が送出されます。

Similar functionality (accepting only text string arguments, but more liberal towards whitespace) is also accessible using the *bytes.fromhex()* class method.

exception binascii.Error

エラーが発生した際に送出される例外です。通常はプログラムのエラーです。

exception binascii.Incomplete

変換するデータが不完全な場合に送出される例外です。通常はプログラムのエラーではなく、多少追加読み込みを行って再度変換を試みることで対処できます。

参考:

base64 **モジュール** RFC 準拠の base64 形式の、底が 16、32、64、85 のエンコーディング。

binhex **モジュール** Macintosh で使われる binhex フォーマットのサポート。

uu **モジュール** Unix で使われる UU エンコードのサポート。

quopri **モジュール** MIME 電子メールメッセージで使われる quoted-printable エンコードのサポート。

19.9 quopri --- MIME quoted-printable 形式データのエンコードおよびデコード

ソースコード: Lib/quopri.py

このモジュールは **RFC 1521**: "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies" で定義されている quoted-printable による伝送のエンコードおよびデコードを行います。quoted-printable エンコーディングは比較的印字不可能な文字の少ないデータのために設計されています; 画像ファイルを送るときのように印字不可能な文字がたくさんある場合には、*base64* モジュールで利用できる base64 エンコーディングのほうがよりコンパクトになります。

quopri.decode(*input*, *output*, *header=False*)

ファイル *input* の内容をデコードして、デコードされたバイナリデータをファイル *output* に書き出します。*input* および *output* は **バイナリファイルオブジェクト** でなければなりません。オプション引数 *header* が存在し、かつその値が真である場合、アンダースコアは空白文字にデコードされます。これは **RFC 1522**: "MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text" で記述されているところの "Q"-エンコードされたヘッダをデコードするのに使われます。

quopri.encode(*input*, *output*, *quotetabs*, *header=False*)

ファイル *input* の内容をエンコードして、その結果の quoted-printable データをファイル *output* に書き出します。*input* および *output* は **バイナリファイルオブジェクト** でなければなりません。*quotetabs* は、内容に含まれている空白とタブ文字をエンコードするかどうかを制御する必須のフラグで、真のときは空白文字をエンコードし、偽のときはエンコードしないままにしておきます。**RFC 1521** に従って、行末にある空白とタブ文字は常にエンコードされることに注意してください。*header* は、空白を **RFC 1522** に従ってアンダースコアにエンコードするかどうかを制御するフラグです。

quopri.**decodestring**(*s*, *header=False*)

decode() に似ていますが、ソース *bytes* を受け取り、対応するデコードされた *bytes* を返します。

quopri.**encodestring**(*s*, *quotetabs=False*, *header=False*)

encode() に似ていますが、ソース *bytes* を受け取り、対応するエンコードされた *bytes* を返します。デフォルトでは、*encode()* 関数の *quotetabs* パラメータに `False` を渡します。

参考:

base64 **モジュール** MIME base64 形式データのエンコードおよびデコード

19.10 uu --- uuencode 形式のエンコードとデコード

ソースコード: Lib/uu.py

このモジュールではファイルを uuencode 形式 (任意のバイナリデータを ASCII 文字列に変換したもの) にエンコード、デコードする機能を提供します。引数としてファイルが仮定されている所では、ファイルのようなオブジェクトが利用できます。後方互換性のために、パス名を含む文字列も利用できるようにしていて、対応するファイルを開いて読み書きします。しかし、このインタフェースは利用しないでください。呼び出し側でファイルを開いて (Windows では `'rb'` か `'wb'` のモードで) 利用する方法が推奨されます。

このコードは Lance Ellinghouse によって提供され、Jack Jansen によって更新されました。

uu モジュールでは以下の関数を定義しています:

uu.**encode**(*in_file*, *out_file*, *name=None*, *mode=None*, ***, *backtick=False*)

in_file を *out_file* にエンコードします。エンコードされたファイルには、デフォルトでデコード時に利用される *name* と *mode* を含んだヘッダがつきます。省略された場合には、*in_file* から取得された名前か `'-'` という文字と、`0o666` がそれぞれデフォルト値として与えられます。*backtick* が真だった場合は、ゼロは空白ではなく `'`'` で表現されます。

バージョン 3.7 で変更: *backtick* パラメータを追加しました.

uu.**decode**(*in_file*, *out_file=None*, *mode=None*, *quiet=False*)

uuencode 形式でエンコードされた *in_file* をデコードして varout_file に書き出します。もし *out_file* がパス名でかつファイルを作る必要があるときには、*mode* がパーミッションの設定に使われます。*out_file* と *mode* のデフォルト値は *in_file* のヘッダから取得されます。しかし、ヘッダで指定されたファイルが既に存在していた場合は、*uu.Error* が送出されます。

誤った実装の uuencoder による入力で、エラーから復旧できた場合、*decode()* は標準エラー出力に警告を表示するかもしれません。*quiet* を真にすることでこの警告を抑制することができます。

exception uu.**Error**

Exception のサブクラスで、*uu.decode()* によって、さまざまな状況で送出される可能性があります。上で紹介された場合以外にも、ヘッダのフォーマットが間違っている場合や、入力ファイルが途中で区切れた場合にも送出されます。

参考:

モジュール *binascii*　ASCII からバイナリへ、バイナリから ASCII への変換をサポートするモジュール。

第

TWENTY

構造化マークアップツール

Python は様々な構造化データマークアップ形式を扱うための、様々なモジュールをサポートしています。これらは標準化一般マークアップ言語 (SGML) およびハイパーテキストマークアップ言語 (HTML)、そして可拡張性マークアップ言語 (XML) を扱うためのいくつかのインタフェースからなります。

20.1 html --- HyperText Markup Language のサポート

ソースコード: Lib/html/__init__.py

このモジュールは HTML を操作するユーティリティを定義しています。

html.escape(*s*, *quote=True*)

文字列 *s* 内の `&`、`<`、および `>` を HTML セーフなシーケンスに変換します。これらの文字を含む HTML を表示する必要がある場合に使用します。オプションフラグ *quote* が真の場合、文字 (`"`) および (`'`) も変換します。これは例えば `<a href="...">` など、引用符で括られている HTML 属性値を包含する時に役立ちます。

バージョン 3.2 で追加.

html.unescape(*s*)

文字列 *s* 中の名前や数字による参照 (例えば `>`, `>`, `>`) を全て対応するユニコード文字に変換します。この関数は、HTML 5 標準規格で定められた有効な文字参照および無効な文字参照、*list of HTML 5 named character references* を対象とします。

バージョン 3.4 で追加.

---

html パッケージのサブモジュールは以下のとおりです:

- *html.parser* -- 許容性のあるモードを持つ HTML/XHTML パーサー
- *html.entities* -- HTML 実体の定義

20.2 html.parser--- HTML および XHTML のシンプルなパーサー

ソースコード: Lib/html/parser.py

このモジュールでは *HTMLParser* クラスを定義します。このクラスは HTML (ハイパーテキスト記述言語、HyperText Mark-up Language) および XHTML で書式化されているテキストファイルを解釈するための基礎となります。

class html.parser.**HTMLParser**(*, *convert_charrefs=True*)

不正なマークアップをパースできるパーサーインスタンスを作成します。

convert_charrefs が (デフォルトの) True である場合、全ての文字参照 (script/style 要素にあるものは除く) は自動的に対応する Unicode 文字に変換されます。

HTMLParser インスタンスは、HTML データが入力されると、開始タグ、終了タグ、およびその他の要素が見つかる度にハンドラーメソッドを呼び出します。各メソッドの挙動を実装するには *HTMLParser* サブクラスを使ってそれぞれを上書きして行います。

このパーサーは終了タグが開始タグと一致しているか調べたり、外側のタグ要素が閉じるときに内側で明示的に閉じられていないタグ要素のタグ終了ハンドラーを呼び出したりはしません。

バージョン 3.4 で変更: キーワード引数 *convert_charrefs* を追加。

バージョン 3.5 で変更: *convert_charrefs* のデフォルト値は True になりました。

20.2.1 HTML パーサーアプリケーションの例

基礎的な例として、*HTMLParser* クラスを使い、発見した開始タグ、終了タグ、およびデータを出力する、シンプルな HTML パーサーを以下に示します:

```
from html.parser import HTMLParser

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Encountered a start tag:", tag)

    def handle_endtag(self, tag):
        print("Encountered an end tag :", tag)

    def handle_data(self, data):
        print("Encountered some data  :", data)

parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>'
            '<body><h1>Parse me!</h1></body></html>')
```

出力は以下のようになります:

```
Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data  : Test
Encountered an end tag : title
Encountered an end tag : head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data  : Parse me!
Encountered an end tag : h1
Encountered an end tag : body
Encountered an end tag : html
```

20.2.2 HTMLParser メソッド

HTMLParser インスタンスは以下のメソッドを提供します:

HTMLParser.**feed**(*data*)
: パーサーにテキストを入力します。入力が完全なタグ要素で構成されている場合に限り処理が行われます; 不完全なデータであった場合、新たにデータが入力されるか、*close()* が呼び出されるまでバッファーされます。*data* は *str* でなければなりません。

HTMLParser.**close**()
: 全てのバッファーされているデータについて、その後にファイル終端マークが続いているとみなして強制的に処理を行います。このメソッドは入力データの終端で行う追加処理を定義するために、派生クラスで再定義することができます。しかし、再定義されたバージョンでは、常に *HTMLParser* 基底クラスのメソッド *close()* を呼び出さなくてはなりません。

HTMLParser.**reset**()
: インスタンスをリセットします。未処理のデータはすべて失われます。インスタンス化の際に暗黙的に呼び出されます。

HTMLParser.**getpos**()
: 現在の行番号およびオフセット値を返します。

HTMLParser.**get_starttag_text**()
: 最も最近開かれた開始タグのテキスト部分を返します。このテキストは必ずしも元データを構造化する上で必須ではありませんが、" 広く知られている (as deployed)" HTML を扱ったり、入力を最小限の変更で再生成 (属性間の空白をそのままにする、など) したりする場合に便利なことがあります。

以下のメソッドはデータまたはマークアップ要素が見つかる度に呼び出されます。これらはサブクラスで上書きされることを想定されています。基底クラスの実装は (*handle_startendtag()* を除き) 何もしません:

HTMLParser.**handle_starttag**(*tag*, *attrs*)
: このメソッドは開始タグを扱うために呼び出されます (例: `<div id="main">`)。

  引数 *tag* はタグの名前で、小文字に変換されます。引数 *attrs* は `(name, value)` のペアからなるリストで、タグの <> 括弧内にある属性が収められています。*name* は小文字に変換され、*value* 内の引用

符は取り除かれ、文字参照と実態参照は置き換えられます。

例えば、タグ <A HREF="https://www.cwi.nl/"> を処理する場合、このメソッドは handle_starttag('a', [('href', 'https://www.cwi.nl/')]) として呼び出されます。

html.entities からのすべての実態参照は、属性値に置き換えられます。

HTMLParser.**handle_endtag**(*tag*)
このメソッドは要素の終了タグを扱うために呼び出されます (例:)。

引数 *tag* はタグの名前で、小文字に変換されます。

HTMLParser.**handle_startendtag**(*tag*, *attrs*)
handle_starttag() と似ていますが、パーサーが XHTML 形式の空タグ () を見つける度に呼び出されます。この特定の字句情報が必要な場合にこのメソッドをサブクラスで上書きすることができます; 既定の実装では、単に *handle_starttag()* および *handle_endtag()* を呼び出します。

HTMLParser.**handle_data**(*data*)
このメソッドは任意のデータを処理するために呼び出されます (例: テキストノードおよび <script>...</script> a や <style>...</style> の内容)。

HTMLParser.**handle_entityref**(*name*)
このメソッドは &name; 形式の名前指定文字参照 (例: >) を処理するために呼び出されます。*name* は一般実体参照になります (例: 'gt')。このメソッドは *convert_charrefs* が True なら呼び出されることはありません。

HTMLParser.**handle_charref**(*name*)
このメソッドは &#NNN; あるいは &#xNNN; 形式の 10 進および 16 進数値文字参照を処理するために呼び出されます。例えば、> と等価な 10 進数は > で、16 進数は > になります。この場合、メソッドは '62' あるいは 'x3E' を受け取ります。このメソッドは *convert_charrefs* が True なら呼び出されることはありません。

HTMLParser.**handle_comment**(*data*)
このメソッドはコメントが見つかった場合に呼び出されます (例: <!--comment-->)。

例えば、コメント <!-- comment --> があると。このメソッドを引数 ' comment ' で呼び出されます。

Internet Explorer 独自拡張の条件付きコメント (condcoms) はこのメソッドに送ることができます。<!--[if IE 9]>IE9-specific content<![endif]--> の場合、このメソッドは '[if IE 9]>IE9-specific content<![endif]' を受け取ります。

HTMLParser.**handle_decl**(*decl*)
このメソッドは HTML doctype 宣言を扱うために呼び出されます (例: <!DOCTYPE html>)。

引数 *decl* は <!...> マークアップ内の宣言の内容全体になります (例: 'DOCTYPE html')。

HTMLParser.**handle_pi**(*data*)
処理指令が見つかった場合に呼び出されます。*data* には、処理指令全体が含まれ、例えば <?proc

color='red'> という処理指令の場合、handle_pi("proc color='red'") のように呼び出されます。このメソッドは派生クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。

注釈: *HTMLParser* クラスでは、処理指令に SGML の構文を使用します。末尾に '?' がある XHTML の処理指令では、'?' が *data* に含まれることになります。

HTMLParser.**unknown_decl**(*data*)

このメソッドはパーサーが未知の宣言を読み込んだ時に呼び出されます。

パラメータ *data* は <![...]> マークアップ内の宣言の内容全体になります。これは派生クラスで上書きする時に役立つことがあります。基底クラスの実装では何もしません。

20.2.3 使用例

以下のクラスは、より多くの例を示すのに用いられるパーサーの実装です:

```
from html.parser import HTMLParser
from html.entities import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Start tag:", tag)
        for attr in attrs:
            print("     attr:", attr)

    def handle_endtag(self, tag):
        print("End tag  :", tag)

    def handle_data(self, data):
        print("Data     :", data)

    def handle_comment(self, data):
        print("Comment  :", data)

    def handle_entityref(self, name):
        c = chr(name2codepoint[name])
        print("Named ent:", c)

    def handle_charref(self, name):
        if name.startswith('x'):
            c = chr(int(name[1:], 16))
        else:
            c = chr(int(name))
        print("Num ent  :", c)

    def handle_decl(self, data):
        print("Decl     :", data)
```

(次のページに続く)

(前のページからの続き)

```
parser = MyHTMLParser()
```

doctype をパースします:

```
>>> parser.feed('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" '
...             '"http://www.w3.org/TR/html4/strict.dtd">')
Decl     : DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.
↪dtd"
```

要素のタイトルと一部属性をパースします:

```
>>> parser.feed('<img src="python-logo.png" alt="The Python logo">')
Start tag: img
     attr: ('src', 'python-logo.png')
     attr: ('alt', 'The Python logo')
>>>
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
Data     : Python
End tag  : h1
```

それ以上のパースを行わずに、script と style 要素の内容をそのまま返します:

```
>>> parser.feed('<style type="text/css">#python { color: green }</style>')
Start tag: style
     attr: ('type', 'text/css')
Data     : #python { color: green }
End tag  : style

>>> parser.feed('<script type="text/javascript">'
...             'alert("<strong>hello!</strong>");</script>')
Start tag: script
     attr: ('type', 'text/javascript')
Data     : alert("<strong>hello!</strong>");
End tag  : script
```

コメントをパースします:

```
>>> parser.feed('<!-- a comment -->'
...             '<!--[if IE 9]>IE-specific content<![endif]-->')
Comment  :  a comment
Comment  : [if IE 9]>IE-specific content<![endif]
```

名前指定および数値文字参照をパースし、正しい文字に変換します (注: これら 3 個の参照はすべて '>' と等価です):

```
>>> parser.feed('&gt;&#62;&#x3E;')
Named ent: >
```

(次のページに続く)

(前のページからの続き)

```
Num ent  : >
Num ent  : >
```

不完全なチャンクを *feed()* に入力しても、(*convert_charrefs* が True に設定されていない限り) *handle_data()* は 1 回以上呼び出される場合があります:

```
>>> for chunk in ['<sp', 'an>buff', 'ered ', 'text</s', 'pan>']:
...     parser.feed(chunk)
...
Start tag: span
Data     : buff
Data     : ered
Data     : text
End tag  : span
```

不正な HTML (例えば属性が引用符で括られていない) のパースも動作します:

```
>>> parser.feed('<p><a class=link href=#main>tag soup</p ></a>')
Start tag: p
Start tag: a
     attr: ('class', 'link')
     attr: ('href', '#main')
Data     : tag soup
End tag  : p
End tag  : a
```

20.3 html.entities --- HTML 一般実体の定義

ソースコード: Lib/html/entities.py

このモジュールは 4 つの辞書、*html5*、*name2codepoint*、*codepoint2name*、および *entitydefs* を定義しています。

html.entities.html5

HTML5 名前付き文字参照[*1] と Unicode 文字とを対応付ける辞書です (例: html5['gt;'] == '>')。末尾のセミコロンは名前に含まれますが (例: 'gt;')、一部の名前はセミコロンなしでも標準で受け付けます。この場合、';' ありの名前と ';' なしの名前が両方存在します。*html.unescape()* も参照してください。

バージョン 3.3 で追加.

html.entities.entitydefs

各 XHTML 1.0 実体定義と ISO Latin-1 における置換テキストとを対応付ける辞書です。

[*1] https://www.w3.org/TR/html5/syntax.html#named-character-references を参照してください

`html.entities.name2codepoint`
: HTML 実体名と Unicode コードポイントとを対応付ける辞書です。

`html.entities.codepoint2name`
: Unicode コードポイントと HTML 実体名とを対応付ける辞書です。

脚注

20.4 XML を扱うモジュール群

ソースコード: Lib/xml/

Python の XML を扱うインタフェースは `xml` パッケージにまとめられています。

> **警告:** XML モジュール群は不正なデータや悪意を持って作成されたデータに対して安全ではありません。信頼できないデータをパースする必要がある場合は *XML* **の脆弱性** と *defusedxml* **パッケージ** を参照してください。

注意すべき重要な点として、*xml* パッケージのモジュールは SAX に対応した XML パーザが少なくとも一つ利用可能でなければなりません。Expat パーザが Python に取り込まれているので、*xml.parsers.expat* モジュールは常に利用できます。

xml.dom および *xml.sax* パッケージのドキュメントは Python による DOM および SAX インタフェースへのバインディングに関する定義です。

XML に関連するサブモジュール:

- *xml.etree.ElementTree*: ElementTree API、シンプルで軽量な XML プロセッサ
- *xml.dom*: DOM API の定義
- *xml.dom.minidom*: 最小限の DOM の実装
- *xml.dom.pulldom*: 部分的な DOM ツリー構築のサポート
- *xml.sax*: SAX2 基底クラスと便利関数群
- *xml.parsers.expat*: Expat parser バインディング

20.4.1 XML の脆弱性

XML 処理モジュールは悪意を持って生成されたデータに対して安全ではありません。攻撃者は XML の機能を悪用して DoS 攻撃、ローカルファイルへのアクセス、他マシンへのネットワーク接続、ファイアーウォールの迂回などを行うことが出来ます。

以下の表は既知の攻撃と各モジュールがそれに対し脆弱かどうかの概要を示しています。

種類	sax	etree	minidom	pulldom	xmlrpc
billion laughs	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)
quadratic blowup	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)
external entity expansion	Safe (5)	Safe (2)	Safe (3)	Safe (5)	安全 (4)
DTD retrieval	Safe (5)	安全	安全	Safe (5)	安全
decompression bomb	安全	安全	安全	安全	**脆弱**
large tokens	**Vulnerable** (6)	**Vulnerable** (6)	**Vulnerable** (6)	**Vulnerable** (6)	**Vulnerable** (6)

1. Expat 2.4.1 and newer is not vulnerable to the ”billion laughs” and ”quadratic blowup” vulnerabilities. Items still listed as vulnerable due to potential reliance on system-provided libraries. Check `pyexpat.EXPAT_VERSION`.

2. *xml.etree.ElementTree* は外部エンティティを展開せず、エンティティが現れた場合は `ParserError` を送出します。

3. *xml.dom.minidom* は外部エンティティを展開せず、展開前のエンティティをそのまま返します。

4. `xmlrpclib` は外部エンティティを展開せず、除外します。

5. Python 3.7.1 からは、一般の外部エンティティはデフォルトで処理されなくなりました。

6. Expat 2.6.0 and newer is not vulnerable to denial of service through quadratic runtime caused by parsing large tokens. Items still listed as vulnerable due to potential reliance on system-provided libraries. Check `pyexpat.EXPAT_VERSION`.

billion laughs / exponential entity expansion　Billion Laughs 攻撃 -- または指数関数的エンティティ展開 (exponential entity expansion) -- は複数階層の入れ子になったエンティティを使用します。各エンティティは別のエンティティを複数回参照し、最終的なエンティティの定義は短い文字列です。指数関数的に展開されることで数 GB のテキストができ、多くのメモリと CPU 時間を消費します。

quadratic blowup entity expansion　二次爆発攻撃 (quadratic blowup attack) はエンティティ展開を悪用する点で Billion Laughs 攻撃に似ています。入れ子になったエンティティの代わりに、この攻撃は数千字の大きなエンティティを何度も繰り返します。この攻撃は指数関数的なものほど効率的ではありませんが、パーザの深い入れ子になったエンティティを禁止する対抗手段をすり抜けます。

external entity expansion　(外部エンティティ展開) エンティティの定義はただのテキスト置換以上のことが

出来ます。外部のリソースやローカルファイルを参照することも出来ます。XML パーザはリソースにアクセスしてその内容を XML 文書に埋め込みます。

DTD retrieval　Python の *xml.dom.pulldom* のような XML ライブラリは DTD をリモートやローカルの場所から読み込みます。この機能には外部エンティティ展開の問題と同じことが予想されます。

decompression bomb　解凍爆弾 (あるいは ZIP 爆弾) は、gzip 圧縮 HTTP ストリームや LZMA 圧縮ファイルなどの圧縮された XML ストリームをパースできる全ての XML ライブラリに対し行われます。攻撃者は送信データ量を 1/3 以下に減らすことができます。

large tokens　Expat needs to re-parse unfinished tokens; without the protection introduced in Expat 2.6.0, this can lead to quadratic runtime that can be used to cause denial of service in the application parsing XML. The issue is known as CVE-2023-52425.

PyPI 上の defusedxml のドキュメントには既知の攻撃手法の詳細が例と文献付きであります。

20.4.2 `defusedxml` パッケージ

defusedxml は潜在的に悪意のある操作を防ぐ、修正された stdlib XML parsers のサブクラスが付属している純 Python パッケージです。信頼出来ない XML データをパースするサーバーコードではこのパッケージの使用が推奨されます。パッケージには悪用の例に加え、XPath injection 等のさらなる XML の悪用の例があります。

20.5 `xml.etree.ElementTree` --- ElementTree XML API

Source code: Lib/xml/etree/ElementTree.py

---

xml.etree.ElementTree モジュールは、XML データを解析および作成するシンプルかつ効率的な API を実装しています。

バージョン 3.3 で変更: このモジュールは利用出来る場合は常に高速な実装を使用します。`xml.etree.cElementTree` は非推奨です。

警告: *xml.etree.ElementTree* モジュールは悪意を持って作成されたデータに対して安全ではありません。信頼できないデータや認証されていないデータををパースする必要がある場合は *XML* の脆弱性 を参照してください。

20.5.1 チュートリアル

これは *xml.etree.ElementTree* (略して ET) を使用するための短いチュートリアルで、ブロックの構築およびモジュールの基本コンセプトを紹介することを目的としています。

XML 木構造と要素

XML は本質的に階層データ形式で、木構造で表すのが最も自然な方法です。ET はこの目的のために 2 つのクラス - XML 文書全体を木で表す *ElementTree* および木構造内の単一ノードを表す *Element* - を持っています。文書全体とのやりとり (ファイルの読み書き) は通常 *ElementTree* レベルで行います。単一 XML 要素およびその子要素とのやりとりは *Element* レベルで行います。

XML の解析

このセクションでは例として以下の XML 文書を使います:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank>68</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

ファイルを読み込むことでこのデータをインポートすることが出来ます:

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

文字列から直接インポートすることも出来ます:

```
root = ET.fromstring(country_data_as_string)
```

fromstring() は XML を文字列から *Element* に直接パースします。*Element* はパースされた木のルート要素です。他のパース関数は *ElementTree* を作成するかもしれません。ドキュメントをきちんと確認してください。

Element として、root はタグと属性の辞書を持ちます:

```
>>> root.tag
'data'
>>> root.attrib
{}
```

さらにイテレート可能な子ノードも持ちます:

```
>>> for child in root:
...     print(child.tag, child.attrib)
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

子ノードは入れ子になっており、インデックスで子ノードを指定してアクセスできます:

```
>>> root[0][1].text
'2008'
```

注釈: XML 入力の全ての要素が、パース後の木に要素として含まれる訳ではありません。現在、このモジュールは入力中のいかなる XML コメント、処理命令、ドキュメントタイプ宣言も読み飛ばします。しかし、XML テキストからのパースではなく、このモジュールの API を使用して構築された木には、コメントや処理命令を含むことができ、それらは XML 出力の生成時に含まれます。ドキュメントタイプ宣言は、*XMLParser* コンストラクタにカスタムの *TreeBuilder* インスタンスを渡すことで、アクセスすることができます。

非ブロックパースのためのプル API

このモジュールが提供するパース関数のほとんどは、結果を返す前に、ドキュメント全体を読む必要があります。*XMLParser* を使用して、インクリメンタルにデータを渡すことは可能ではありますが、それはコールバック対象のメソッドを呼ぶプッシュ API であり、多くの場合、低水準すぎて不便です。ユーザーが望むのは、完全に出来上がった *Element* オブジェクトを便利に使いながら、操作をブロックすることなく XML のパースをインクリメンタルに行えることです。

これを行うための最も強力なツールは、*XMLPullParser* です。XML データを取得するためにブロックするような読み込みは必要なく、*XMLPullParser.feed()* を呼び出して、インクリメンタルにデータを読みます。パースされた XML 要素を取得するには、*XMLPullParser.read_events()* を呼び出します。以下に、例を示します。

```
>>> parser = ET.XMLPullParser(['start', 'end'])
>>> parser.feed('<mytag>sometext')
>>> list(parser.read_events())
[('start', <Element 'mytag' at 0x7fa66db2be58>)]
>>> parser.feed(' more text</mytag>')
>>> for event, elem in parser.read_events():
...     print(event)
...     print(elem.tag, 'text=', elem.text)
...
end
```

これの分かりやすい用途は、XML データをソケットから受信したり、ストレージデバイスからインクリメンタルに読み出したりするような、非ブロック式に動作するアプリケーションです。このような場合、ブロッキング読み出しは使用できません。

XMLPullParser は柔軟性が非常に高いため、単純に使用したいユーザーにとっては不便かもしれません。アプリケーションにおいて、XML データの読み取り時にブロックすることに支障がないが、インクリメンタルにパースする能力が欲しい場合、*iterparse()* を参照してください。大きな XML ドキュメントを読んでいて、全てメモリ上にあるという状態にしたくない場合に有用です。

Where *immediate* feedback through events is wanted, calling method *XMLPullParser.flush()* can help reduce delay; please make sure to study the related security notes.

関心ある要素の検索

Element は、例えば、*Element.iter()* などの、配下 (その子ノードや孫ノードなど) の部分木全体を再帰的にイテレートするいくつかの役立つメソッドを持っています:

```
>>> for neighbor in root.iter('neighbor'):
...     print(neighbor.attrib)
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

Element.findall() はタグで現在の要素の直接の子要素のみ検索します。*Element.find()* は特定のタグで **最初の** 子要素を検索し、*Element.text* は要素のテキストコンテンツにアクセスします。*Element.get()* は要素の属性にアクセスします:

```
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print(name, rank)
...
Liechtenstein 1
Singapore 4
Panama 68
```

XPath を使用すると、より洗練された方法で、検索したい要素を指定することができます。

XML ファイルの編集

ElementTree は XML 文書を構築してファイルに出力する簡単な方法を提供しています。*ElementTree.write()* メソッドはこの目的に適います。

Element オブジェクトを作成すると、そのフィールドの直接変更 (*Element.text* など) や、属性の追加および変更 (*Element.set()* メソッド)、あるいは新しい子ノードの追加 (例えば *Element.append()* など) によってそれを操作できます。

例えば各 country の rank に 1 を足して、rank 要素に `updated` 属性を追加したい場合:

```
>>> for rank in root.iter('rank'):
...     new_rank = int(rank.text) + 1
...     rank.text = str(new_rank)
...     rank.set('updated', 'yes')
...
>>> tree.write('output.xml')
```

XML はこのようになります:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank updated="yes">69</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

Element.remove() を使って要素を削除することが出来ます。例えば rank が 50 より大きい全ての country を削除したい場合:

```
>>> for country in root.findall('country'):
...     # using root.findall() to avoid removal during traversal
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...
>>> tree.write('output.xml')
```

Note that concurrent modification while iterating can lead to problems, just like when iterating and modifying Python lists or dicts. Therefore, the example first collects all matching elements with `root.findall()`, and only then iterates over the list of matches.

XML はこのようになります:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
</data>
```

XML 文書の構築

SubElement() 関数は、与えられた要素に新しい子要素を作成する便利な手段も提供しています:

```
>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
>>> c = ET.SubElement(a, 'c')
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /><c><d /></c></a>
```

名前空間のある XML の解析

XML 入力が 名前空間 を持っている場合、prefix:sometag の形式で修飾されたタグと属性が、その *prefix* が完全な *URI* で置換された {uri}sometag の形に展開されます。さらに、デフォルトの XML 名前空間 があると、修飾されていない全てのタグにその完全 URI が前置されます。

ひとつは接頭辞 "fictional" でもうひとつがデフォルト名前空間で提供された、2 つの名前空間を組み込んだ XML の例をここにお見せします:

```
<?xml version="1.0"?>
<actors xmlns:fictional="http://characters.example.com"
        xmlns="http://people.example.com">
    <actor>
        <name>John Cleese</name>
        <fictional:character>Lancelot</fictional:character>
        <fictional:character>Archie Leach</fictional:character>
    </actor>
    <actor>
        <name>Eric Idle</name>
        <fictional:character>Sir Robin</fictional:character>
        <fictional:character>Gunther</fictional:character>
        <fictional:character>Commander Clement</fictional:character>
    </actor>
</actors>
```

この XML の例を、検索し、渡り歩くためのひとつの方法としては、*find()* や *findall()* に渡す xpath で全てのタグや属性に手作業で URI を付けてまわる手があります:

```
root = fromstring(xml_text)
for actor in root.findall('{http://people.example.com}actor'):
    name = actor.find('{http://people.example.com}name')
    print(name.text)
    for char in actor.findall('{http://characters.example.com}character'):
        print(' |-->', char.text)
```

もっと良い方法があります。接頭辞の辞書を作り、これを検索関数で使うことです:

```
ns = {'real_person': 'http://people.example.com',
      'role': 'http://characters.example.com'}

for actor in root.findall('real_person:actor', ns):
    name = actor.find('real_person:name', ns)
    print(name.text)
    for char in actor.findall('role:character', ns):
        print(' |-->', char.text)
```

どちらのアプローチでも同じ結果です:

```
John Cleese
 |--> Lancelot
 |--> Archie Leach
```

(次のページに続く)

(前のページからの続き)

```
Eric Idle
 |--> Sir Robin
 |--> Gunther
 |--> Commander Clement
```

その他の情報

http://effbot.org/zone/element-index.htm にはチュートリアルと他のドキュメントへのリンクがあります。

20.5.2 XPath サポート

このモジュールは木構造内の要素の位置決めのための XPath 式 を限定的にサポートしています。その目指すところは短縮構文のほんの一部だけのサポートであり、XPath エンジンのフルセットは想定していません。

使用例

以下はこのモジュールの XPath 機能の一部を紹介する例です。*XML* **の解析** 節から XML 文書 `countrydata` を使用します:

```
import xml.etree.ElementTree as ET

root = ET.fromstring(countrydata)

# Top-level elements
root.findall(".")

# All 'neighbor' grand-children of 'country' children of the top-level
# elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall(".//year/..[@name='Singapore']")

# 'year' nodes that are children of nodes with name='Singapore'
root.findall(".//*[@name='Singapore']/year")

# All 'neighbor' nodes that are the second child of their parent
root.findall(".//neighbor[2]")
```

For XML with namespaces, use the usual qualified `{namespace}tag` notation:

```
# All dublin-core "title" tags in the document
root.findall(".//{http://purl.org/dc/elements/1.1/}title")
```

サポートされている XPath 構文

操作	意味
`tag`	Selects all child elements with the given tag. For example, `spam` selects all child elements named `spam`, and `spam/egg` selects all grandchildren named `egg` in all children named `spam`. `{namespace}*` selects all tags in the given namespace, `{*}spam` selects tags named `spam` in any (or no) namespace, and `{}*` only selects tags that are not in a namespace. バージョン 3.8 で変更: Support for star-wildcards was added.
`*`	Selects all child elements, including comments and processing instructions. For example, `*/egg` selects all grandchildren named `egg`.
`.`	現在のノードを選択します。これはパスの先頭に置くことで相対パスであることを示すのに役立ちます。
`//`	現在の要素の下にある全てのレベルの全ての子要素を選択します。例えば、`.//egg` は木全体から `egg` 要素を選択します。
`..`	親ノードを選択します。パスが開始要素 (`find` が呼ばれた要素) の上の要素に進もうとした場合 `None` を返します。
`[@attrib]`	与えられた属性を持つ全ての要素を選択します。
`[@attrib='value']`	与えられた属性が与えられた値を持つ全ての要素を選択します。値に引用符は含められません。
`[tag]`	`tag` という名前の子要素を持つ全ての要素を選択します。直下の子要素のみサポートしています。
`[.='text']`	子孫のうち、与えられた `text` とテキスト全体が等しい全ての要素を選択します。 バージョン 3.7 で追加.
`[tag='text']`	子孫を含む完全なテキストコンテンツと与えられた `text` が一致する、`tag` と名付けられた子要素を持つすべての要素を選択します。
`[position]`	与えられた位置にあるすべての要素を選択します。位置は整数 (先頭は 1)、式 `last()` (末尾)、あるいは末尾からの相対位置 (例えば `last()-1`) のいずれかで指定できます。

述語 (角括弧内の式) の前にはタグ名、アスタリスク、あるいはその他の述語がなければなりません。`position` 述語の前にはタグ名がなければなりません。

20.5.3 リファレンス

関数

`xml.etree.ElementTree.canonicalize`(*xml_data=None*, ***, *out=None*, *from_file=None*, ***options*)

C14N 2.0 transformation function.

Canonicalization is a way to normalise XML output in a way that allows byte-by-byte comparisons and digital signatures. It reduced the freedom that XML serializers have and instead generates a more constrained XML representation. The main restrictions regard the placement of namespace

declarations, the ordering of attributes, and ignorable whitespace.

This function takes an XML data string (*xml_data*) or a file path or file-like object (*from_file*) as input, converts it to the canonical form, and writes it out using the *out* file(-like) object, if provided, or returns it as a text string if not. The output file receives text, not bytes. It should therefore be opened in text mode with `utf-8` encoding.

Typical uses:

```
xml_data = "<root>...</root>"
print(canonicalize(xml_data))

with open("c14n_output.xml", mode='w', encoding='utf-8') as out_file:
    canonicalize(xml_data, out=out_file)

with open("c14n_output.xml", mode='w', encoding='utf-8') as out_file:
    canonicalize(from_file="inputfile.xml", out=out_file)
```

The configuration *options* are as follows:

- *with_comments*: set to true to include comments (default: false)
- *strip_text*: set to true to strip whitespace before and after text content (default: false)
- *rewrite_prefixes*: set to true to replace namespace prefixes by "n{number}" (default: false)
- *qname_aware_tags*: a set of qname aware tag names in which prefixes should be replaced in text content (default: empty)
- *qname_aware_attrs*: a set of qname aware attribute names in which prefixes should be replaced in text content (default: empty)
- *exclude_attrs*: a set of attribute names that should not be serialised
- *exclude_tags*: a set of tag names that should not be serialised

In the option list above, "a set" refers to any collection or iterable of strings, no ordering is expected.

バージョン 3.8 で追加.

`xml.etree.ElementTree.Comment`(*text=None*)

コメント要素のファクトリです。このファクトリ関数は、標準のシリアライザでは XML コメントにシリアライズされる特別な要素を作ります。コメント文字列はバイト文字列でも Unicode 文字列でも構いません。*text* はそのコメント文字列を含んだ文字列です。コメントを表わす要素のインスタンスを返します。

XMLParser は、入力に含まれるコメントを読み飛ばし、コメントオブジェクトは作成しません。*ElementTree* は、*Element* メソッドの 1 つを使用して木内に挿入されたコメントノードのみを含みます。

xml.etree.ElementTree.**dump**(*elem*)

要素の木もしくは要素の構造を sys.stdout に出力します。この関数はデバッグ目的のみに使用してください。

出力される形式の正確なところは実装依存です。このバージョンでは、通常の XML ファイルとして出力されます。

elem は要素の木もしくは個別の要素です。

バージョン 3.8 で変更: The *dump()* function now preserves the attribute order specified by the user.

xml.etree.ElementTree.**fromstring**(*text*, *parser=None*)

文字列定数で与えられた XML 断片を解析します。*XML()* と同じです。*text* には XML データを含む文字列を指定します。*parser* はオプションで、パーザのインスタンスを指定します。指定されなかった場合、標準の *XMLParser* パーザを使用します。*Element* インスタンスを返します。

xml.etree.ElementTree.**fromstringlist**(*sequence*, *parser=None*)

文字列フラグメントのシーケンスから XML ドキュメントを解析します。*sequence* は XML データのフラグメントを格納した、リストかその他のシーケンスです。*parser* はオプションのパーザインスタンスです。パーザが指定されない場合、標準の *XMLParser* パーザが使用されます。*Element* インスタンスを返します。

バージョン 3.2 で追加.

xml.etree.ElementTree.**iselement**(*element*)

オブジェクトが正当な要素オブジェクトであるかをチェックします。*element* は要素インスタンスです。引数が要素オブジェクトの場合 `True` を返します。

xml.etree.ElementTree.**iterparse**(*source*, *events=None*, *parser=None*)

XML セクションを構文解析して要素の木を漸増的に作っていき、その間進行状況をユーザーに報告します。*source* は XML データを含むファイル名または **ファイルオブジェクト** です。*events* は報告すべきイベントのシーケンスです。サポートされているイベントは、文字列の `"start"`, `"end"`,“”comment”, ``"pi"`, `"start-ns"`, `"end-ns"` です (”ns” イベントは、名前空間についての詳細情報を取得するために使用)。*events* が省略された場合は `"end"` イベントだけが報告されます。*parser* はオプションの引数で、パーサーのインスタンスです。指定されなかった場合は標準の *XMLParser* が利用されます。*parser* は *XMLParser* のサブクラスでなくてはならず、ターゲットとして既定の *TreeBuilder* のみしか使用できません。`(event, elem)` ペアを提供する **イテレータ** を返します。

iterparse() は木をインクリメンタルに構築しますが、*source* (または指定のファイル) でのブロッキング読みを起こします。したがって、ブロッキング読みが許可されないアプリケーションには適しません。完全に非ブロックのパースのためには、*XMLPullParser* を参照してください。

注釈: *iterparse()* は ”start” イベントを発行した時に開始タグの文字 ”>” が現れたことだけを保証します。そのため、属性は定義されますが、その時点ではテキストの内容も tail 属性も定義されていません。同じことは子要素にも言えて、その時点ではあるともないとも言えません。

全部揃った要素が必要ならば、”end” イベントを探してください。

バージョン 3.4 で非推奨: *parser* 引数。

バージョン 3.8 で変更: イベント `comment`, `pi` が追加されました。

`xml.etree.ElementTree.parse`(*source*, *parser=None*)

XML 断片を解析して要素の木にします。*source* には XML データを含むファイル名またはファイルオブジェクトを指定します。*parser* はオプションでパーザインスタンスを指定します。パーザが指定されない場合、標準の *XMLParser* パーザが使用されます。*ElementTree* インスタンスを返します。

`xml.etree.ElementTree.ProcessingInstruction`(*target*, *text=None*)

PI 要素のファクトリです。このファクトリ関数は XML の処理命令としてシリアライズされた特別な要素を作成します。*target* は PI ターゲットを含んだ文字列です。*text* を指定する場合は PI コンテンツを含む文字列にします。PI を表わす要素インスタンスを返します。

XMLParser は、入力に含まれる処理命令を読み飛ばし、コメントオブジェクトは作成しません。*ElementTree* は、*Element* メソッドの 1 つを使用して木内に挿入された処理命令ノードのみを含みます。

`xml.etree.ElementTree.register_namespace`(*prefix*, *uri*)

名前空間の接頭辞を登録します。レジストリはグローバルで、与えられた接頭辞か名前空間 URI のどちらかの既存のマッピングはすべて削除されます。*prefix* には名前空間の接頭辞を指定します。*uri* には名前空間の URI を指定します。この名前空間のタグや属性は、可能な限り与えられた接頭辞をつけてシリアライズされます。

バージョン 3.2 で追加.

`xml.etree.ElementTree.SubElement`(*parent*, *tag*, *attrib={}*, ***extra*)

子要素のファクトリです。この関数は要素インスタンスを作成し、それを既存の要素に追加します。

要素名、属性名、および属性値はバイト文字列でも Unicode 文字列でも構いません。*parent* には親要素を指定します。*tag* には要素名を指定します。*attrib* はオプションで要素の属性を含む辞書を指定します。*extra* は追加の属性で、キーワード引数として与えます。要素インスタンスを返します。

`xml.etree.ElementTree.tostring`(*element*, *encoding=”us-ascii”*, *method=”xml”*, ***, *xml_declaration=None*, *default_namespace=None*, *short_empty_elements=True*)

XML 要素を全ての子要素を含めて表現する文字列を生成します。*element* は *Element* のインスタンスです。*encoding*[*1] は出力エンコーディング (デフォルトは US-ASCII) です。Unicode 文字列を生成するには、`encoding="unicode"` を使用してください。*method* は `"xml"`, `"html"`, `"text"` のいずれか (デフォルトは `"xml"`) です。*xml_declaration*, *default_namespace*, *short_empty_elements* は、*ElementTree.write()* での意味と同じ意味を持ちます。XML データを含んだ (オプションで) エンコードされた文字列を返します。

[*1] XML 出力に含まれるエンコーディング文字列は適切な規格に従っていなければなりません。例えば、”UTF-8” は有効ですが、”UTF8” はそうではありません。https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl と https://www.iana.org/assignments/character-sets/character-sets.xhtml を参照してください。

バージョン 3.4 で追加: *short_empty_elements* 引数。

バージョン 3.8 で追加: *xml_declaration* と *default_namespace* 引数。

バージョン 3.8 で変更: The *tostring()* function now preserves the attribute order specified by the user.

xml.etree.ElementTree.**tostringlist**(*element*, *encoding="us-ascii"*, *method="xml"*, ***, *xml_declaration=None*, *default_namespace=None*, *short_empty_elements=True*)

XML 要素を全ての子要素を含めて表現する文字列を生成します。*element* は *Element* のインスタンスです。*encoding*[*1] は出力エンコーディング (デフォルトは US-ASCII) です。Unicode 文字列を生成するには、`encoding="unicode"` を使用してください。*method* は `"xml"`, `"html"`, `"text"` のいずれか (デフォルトは `"xml"`) です。xml_declaration*, *default_namespace*, *short_empty_elements* は、*ElementTree.write()* での意味と同じ意味を持ちます。XML データを含んだ (オプションで) エンコードされた文字列のリストを返します。`b"".join(tostringlist(element)) == tostring(element)` となること以外、特定の順序になる保証はありません。

バージョン 3.2 で追加.

バージョン 3.4 で追加: *short_empty_elements* 引数。

バージョン 3.8 で追加: *xml_declaration* と *default_namespace* 引数。

バージョン 3.8 で変更: *tostringlist()* 関数はユーザーが指定した属性の順序を保持するようになりました。

xml.etree.ElementTree.**XML**(*text*, *parser=None*)

文字列定数で与えられた XML 断片を解析します。この関数は Python コードに "XML リテラル" を埋め込むのに使えます。*text* には XML データを含む文字列を指定します。*parser* はオプションで、パーザのインスタンスを指定します。指定されなかった場合、標準の *XMLParser* パーザを使用します。*Element* インスタンスを返します。

xml.etree.ElementTree.**XMLID**(*text*, *parser=None*)

文字列定数で与えられた XML 断片を解析し、要素 ID と要素を対応付ける辞書を返します。*text* には XML データを含んだ文字列を指定します。*parser* はオプションで、パーザのインスタンスを指定します。指定されなかった場合、標準の *XMLParser* パーザを使用します。*Element* のインスタンスと辞書のタプルを返します。

20.5.4 XInclude サポート

This module provides limited support for XInclude directives, via the `xml.etree.ElementInclude` helper module. This module can be used to insert subtrees and text strings into element trees, based on information in the tree.

使用例

Here's an example that demonstrates use of the XInclude module. To include an XML document in the current document, use the `{http://www.w3.org/2001/XInclude}include` element and set the **parse** attribute to `"xml"`, and use the **href** attribute to specify the document to include.

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="source.xml" parse="xml" />
</document>
```

By default, the **href** attribute is treated as a file name. You can use custom loaders to override this behaviour. Also note that the standard helper does not support XPointer syntax.

To process this file, load it as usual, and pass the root element to the *xml.etree.ElementTree* module:

```
from xml.etree import ElementTree, ElementInclude

tree = ElementTree.parse("document.xml")
root = tree.getroot()

ElementInclude.include(root)
```

The ElementInclude module replaces the `{http://www.w3.org/2001/XInclude}include` element with the root element from the **source.xml** document. The result might look something like this:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <para>This is a paragraph.</para>
</document>
```

If the **parse** attribute is omitted, it defaults to "xml". The href attribute is required.

To include a text document, use the `{http://www.w3.org/2001/XInclude}include` element, and set the **parse** attribute to "text":

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) <xi:include href="year.txt" parse="text" />.
</document>
```

The result might look something like:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) 2003.
</document>
```

20.5.5 リファレンス

関数

xml.etree.ElementInclude.**default_loader**(*href*, *parse*, *encoding=None*)

Default loader. This default loader reads an included resource from disk. *href* is a URL. *parse* is for parse mode either "xml" or "text". *encoding* is an optional text encoding. If not given, encoding is `utf-8`. Returns the expanded resource. If the parse mode is `"xml"`, this is an ElementTree instance. If the parse mode is "text", this is a Unicode string. If the loader fails, it can return None or raise an exception.

xml.etree.ElementInclude.**include**(*elem*, *loader=None*)

This function expands XInclude directives. *elem* is the root element. *loader* is an optional resource loader. If omitted, it defaults to *default_loader()*. If given, it should be a callable that implements the same interface as *default_loader()*. Returns the expanded resource. If the parse mode is `"xml"`, this is an ElementTree instance. If the parse mode is "text", this is a Unicode string. If the loader fails, it can return None or raise an exception.

Element オブジェクト

class xml.etree.ElementTree.**Element**(*tag*, *attrib={}*, ***extra*)

要素クラスです。この関数は Element インタフェースを定義すると同時に、そのリファレンス実装を提供します。

要素名、属性名、および属性値はバイト文字列でも Unicode 文字列でも構いません。*tag* には要素名を指定します。*attrib* はオプションで、要素と属性を含む辞書を指定します。*extra* は追加の属性で、キーワード引数として与えます。要素インスタンスを返します。

tag

この要素が表すデータの種類を示す文字列です (言い替えると、要素の型です)。

text

tail

これらの属性は要素に結びつけられた付加的なデータを保持するのに使われます。これらの属性値はたいてい文字列ですが、アプリケーション固有のオブジェクトであって構いません。要素が XML ファイルから作られる場合、*text* 属性は要素の開始タグとその最初の子要素または終了タグまでのテキストか、あるいは `None` を保持し、*tail* 属性は要素の終了タグと次のタグまでのテキストか、あるいは `None` を保持します。このような XML データ

```
<a><b>1<c>2<d/>3</c></b>4</a>
```

の場合、*a* 要素は *text*, *tail* 属性ともに `None`, *b* 要素は *text* に `"1"` で *tail* に `"4"`, *c* 要素は *text* に `"2"` で *tail* は `None`, *d* 要素 は *text* が `None` で *tail* に `"3"` をそれぞれ保持します。

要素の内側のテキストを収集するためには、*itertext()* を参照してください。例えば `"".join(element.itertext())` のようにします。

アプリケーションはこれらの属性に任意のオブジェクトを格納できます。

attrib
要素の属性を保持する辞書です。*attrib* の値は常に書き換え可能な Python 辞書ですが、ElementTree の実装によっては別の内部表現を使用し、要求されたときにだけ辞書を作るようにしているかもしれません。そうした実装の利益を享受するために、可能な限り下記の辞書メソッドを通じて使用してください。

以下の辞書風メソッドが要素の属性に対して動作します。

clear()
要素をリセットします。この関数は全ての子要素を削除し、全属性を消去し、テキストとテール属性を `None` に設定します。

get(*key*, *default=None*)
要素の *key* という名前の属性を取得します。

属性の値、または属性がない場合は *default* を返します。

items()
要素の属性を (名前, 値) ペアのシーケンスとして返します。返される属性の順番は決まっていません。

keys()
要素の属性名をリストとして返します。返される名前の順番は決まっていません。

set(*key*, *value*)
要素の属性 *key* に *value* をセットします。

以下のメソッドは要素の子要素 (副要素) に対して動作します。

append(*subelement*)
要素 *subelement* を、要素の子要素の内部リストの末尾に追加します。*subelement* *Element* でない場合、*TypeError* を送出します。

extend(*subelements*)
0 個以上の要素のシーケンスオブジェクトによって *subelements* を拡張します。*subelements* が *Element* でない場合、*TypeError* を送出します。

バージョン 3.2 で追加.

find(*match*, *namespaces=None*)
Finds the first subelement matching *match*. *match* may be a tag name or a *path*. Returns an element instance or `None`. *namespaces* is an optional mapping from namespace prefix to full name. Pass `''` as prefix to move all unprefixed tag names in the expression into the given namespace.

findall(*match*, *namespaces=None*)
Finds all matching subelements, by tag name or *path*. Returns a list containing all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to

full name. Pass `''` as prefix to move all unprefixed tag names in the expression into the given namespace.

findtext(*match*, *default=None*, *namespaces=None*)

Finds text for the first subelement matching *match*. *match* may be a tag name or a *path*. Returns the text content of the first matching element, or *default* if no element was found. Note that if the matching element has no text content an empty string is returned. *namespaces* is an optional mapping from namespace prefix to full name. Pass `''` as prefix to move all unprefixed tag names in the expression into the given namespace.

getchildren()

Deprecated since version 3.2, will be removed in version 3.9: `list(elem)` かイテレーションを使用してください。

getiterator(*tag=None*)

Deprecated since version 3.2, will be removed in version 3.9: 代わりに *Element.iter()* メソッドを使用してください。

insert(*index*, *subelement*)

要素内の指定された位置に *subelement* を挿入します。*subelement* が *Element* でない場合、*TypeError* を送出します。

iter(*tag=None*)

現在の要素を根とする木の **イテレータ** を作成します。イテレータは現在の要素とそれ以下のすべての要素を、文書内での出現順 (深さ優先順) でイテレートします。*tag* が `None` または `'*'` でない場合、与えられたタグに等しいものについてのみイテレータから返されます。イテレート中に木構造が変更された場合の結果は未定義です。

バージョン 3.2 で追加.

iterfind(*match*, *namespaces=None*)

タグ名または **パス** にマッチするすべての子要素を検索します。マッチしたすべての要素を文書内での出現順で yield するイテレータを返します。*namespaces* はオプションで、名前空間接頭辞と完全名を対応付けるマップオブジェクトを指定します。

バージョン 3.2 で追加.

itertext()

テキストのイテレータを作成します。イテレータは、この要素とすべての子要素を文書上の順序で巡回し、すべての内部のテキストを返します。

バージョン 3.2 で追加.

makeelement(*tag*, *attrib*)

現在の要素と同じ型の新しい要素オブジェクトを作成します。このメソッドは呼び出さずに、*SubElement()* ファクトリ関数を使って下さい。

remove(*subelement*)

要素から *subelement* を削除します。find* メソッド群と異なり、このメソッドは要素をインスタ

ンスの同一性で比較します。タグや内容では比較しません。

Element オブジェクトは以下のシーケンス型のメソッドを、サブ要素を操作するためにサポートします: `__delitem__()`, `__getitem__()`, `__setitem__()`, `__len__()`.

注意: 子要素を持たない要素の真偽値は `False` になります。この挙動は将来のバージョンで変更されるかもしれません。直接真偽値をテストするのでなく、`len(elem)` か `elem is None` を利用してください。

```
element = root.find('foo')

if not element:  # careful!
    print("element not found, or element has no subelements")

if element is None:
    print("element not found")
```

Prior to Python 3.8, the serialisation order of the XML attributes of elements was artificially made predictable by sorting the attributes by their name. Based on the now guaranteed ordering of dicts, this arbitrary reordering was removed in Python 3.8 to preserve the order in which attributes were originally parsed or created by user code.

In general, user code should try not to depend on a specific ordering of attributes, given that the XML Information Set explicitly excludes the attribute order from conveying information. Code should be prepared to deal with any ordering on input. In cases where deterministic XML output is required, e.g. for cryptographic signing or test data sets, canonical serialisation is available with the *canonicalize()* function.

In cases where canonical output is not applicable but a specific attribute order is still desirable on output, code should aim for creating the attributes directly in the desired order, to avoid perceptual mismatches for readers of the code. In cases where this is difficult to achieve, a recipe like the following can be applied prior to serialisation to enforce an order independently from the Element creation:

```
def reorder_attributes(root):
    for el in root.iter():
        attrib = el.attrib
        if len(attrib) > 1:
            # adjust attribute order, e.g. by sorting
            attribs = sorted(attrib.items())
            attrib.clear()
            attrib.update(attribs)
```

ElementTree オブジェクト

class xml.etree.ElementTree.ElementTree(*element=None*, *file=None*)

ElementTree ラッパークラスです。このクラスは要素の全階層を表現し、さらに標準 XML との相互変換を追加しています。

element は根要素です。*file* が指定されている場合、その XML ファイルの内容により木は初期化されます。

_setroot(*element*)

この木の根要素を置き換えます。従って現在の木の内容は破棄され、与えられた要素が代わりに使われます。注意して使ってください。*element* は要素インスタンスです。

find(*match*, *namespaces=None*)

Element.find() と同じで、木の根要素を起点とします。

findall(*match*, *namespaces=None*)

Element.findall() と同じで、木の根要素を起点とします。

findtext(*match*, *default=None*, *namespaces=None*)

Element.findtext() と同じで、木の根要素を起点とします。

getiterator(*tag=None*)

Deprecated since version 3.2, will be removed in version 3.9: 代わりに *ElementTree.iter()* メソッドを使用してください。

getroot()

この木のルート要素を返します。

iter(*tag=None*)

根要素に対する、木を巡回するイテレータを返します。イテレータは木のすべての要素に渡ってセクション順にループします。*tag* は探したいタグです (デフォルトではすべての要素を返します)。

iterfind(*match*, *namespaces=None*)

Element.iterfind() と同じで、木の根要素を起点とします。

バージョン 3.2 で追加.

parse(*source*, *parser=None*)

外部の XML 断片をこの要素木に入れます。*source* にはファイル名か **ファイルオブジェクト** を指定します。*parser* はオプションで、パーザインスタンスを指定します。パーザが指定されない場合、標準の *XMLParser* パーザが使用されます。断片の根要素を返します。

write(*file*, *encoding="us-ascii"*, *xml_declaration=None*, *default_namespace=None*, *method="xml"*, ***, *short_empty_elements=True*)

要素の木をファイルに XML として書き込みます。*file* は、書き込み用に開かれたファイル名または **ファイルオブジェクト** です。*encoding*[*1] は出力エンコーディング (デフォルトは US-ASCII) です。*xml_declaration* は、XML 宣言がファイルに書かれるかどうかを制御します。`False` の場合は常に書かれず、`True` の場合は常に書かれ、`None` の場合は US-ASCII 、UTF-8 、Unicode 以外の場合に書かれます (デフォルトは `None` です)。*default_namespace* でデフォルトの XML 名

前空間 (”xmlns” 用) を指定します。*method* は "xml", "html", "text" のいずれかです (デフォルトは "xml" です)。キーワード専用の *short_empty_elements* 引数は、内容がない属性のフォーマットを制御します。True (既定) の場合、単一の空要素タグとして書かれ、False の場合、開始タグと終了タグのペアとしてかかれます。

出力は引数 *encoding* によって、文字列 (*str*) かバイト列 (*bytes*) になります。*encoding* が "unicode" の場合、出力は文字列になり、それ以外ではバイト列になります。*file* が **ファイルオブジェクト** の場合、型が衝突する場合があります。文字列をバイト列ファイルへ書き込んだり、その逆を行わないよう注意してください。

バージョン 3.4 で追加: *short_empty_elements* 引数。

バージョン 3.8 で変更: *write()* メソッドはユーザーが指定した属性の順序を保持するようになりました。

以下はこれから操作する XML ファイルです:

```
<html>
    <head>
        <title>Example page</title>
    </head>
    <body>
        <p>Moved to <a href="http://example.org/">example.org</a>
        or <a href="http://example.com/">example.com</a>.</p>
    </body>
</html>
```

第 1 段落のすべてのリンクの ”target” 属性を変更する例:

```
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")     # Finds first occurrence of tag p in body
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))   # Returns list of all links
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77ec1cc>]
>>> for i in links:             # Iterates through all found links
...     i.attrib["target"] = "blank"
>>> tree.write("output.xhtml")
```

QName オブジェクト

class xml.etree.ElementTree.**QName**(*text_or_uri*, *tag=None*)

QName ラッパーです。このクラスは QName 属性値をラップし、出力時に適切な名前空間の扱いを得るために使われます。*text_or_uri* は {uri}local という形式の QName 値を含む文字列、または tag 引数が与えられた場合には QName の URI 部分の文字列です。*tag* が与えられた場合、一つめの引数は URI と解釈され、この引数はローカル名と解釈されます。*QName* インスタンスは不透明です。

TreeBuilder オブジェクト

class xml.etree.ElementTree.**TreeBuilder**(*element_factory=None*, ***, *comment_factory=None*, *pi_factory=None*, *insert_comments=False*, *insert_pis=False*)

汎用の要素構造ビルダー。これは start, data, end, comment, pi のメソッド呼び出しの列を整形式の要素構造に変換します。このクラスを使うと、好みの XML 構文解析器、または他の XML に似た形式の構文解析器を使って、要素構造を作り出すことができます。

element_factory, when given, must be a callable accepting two positional arguments: a tag and a dict of attributes. It is expected to return a new element instance.

The *comment_factory* and *pi_factory* functions, when given, should behave like the *Comment()* and *ProcessingInstruction()* functions to create comments and processing instructions. When not given, the default factories will be used. When *insert_comments* and/or *insert_pis* is true, comments/pis will be inserted into the tree if they appear within the root element (but not outside of it).

close()

ビルダのバッファをフラッシュし、最上位の文書要素を返します。戻り値は *Element* インスタンスになります。

data(*data*)

現在の要素にテキストを追加します。*data* は文字列です。バイト文字列もしくは Unicode 文字列でなければなりません。

end(*tag*)

現在の要素を閉じます。*tag* は要素の名前です。閉じられた要素を返します。

start(*tag*, *attrs*)

新しい要素を開きます。*tag* は要素の名前です。*attrs* は要素の属性を保持した辞書です。開かれた要素を返します。

comment(*text*)

Creates a comment with the given *text*. If `insert_comments` is true, this will also add it to the tree.

バージョン 3.8 で追加.

pi(*target*, *text*)

Creates a comment with the given *target* name and *text*. If `insert_pis` is true, this will also

add it to the tree.

バージョン 3.8 で追加.

加えて、カスタムの *TreeBuilder* オブジェクトは以下のメソッドを提供できます:

doctype(*name*, *pubid*, *system*)
doctype 宣言を処理します。*name* は doctype 名です。*pubid* は公式の識別子です。*system* はシステム識別子です。このメソッドはデフォルトの *TreeBuilder* クラスには存在しません。

バージョン 3.2 で追加.

start_ns(*prefix*, *uri*)
Is called whenever the parser encounters a new namespace declaration, before the `start()` callback for the opening element that defines it. *prefix* is `''` for the default namespace and the declared namespace prefix name otherwise. *uri* is the namespace URI.

バージョン 3.8 で追加.

end_ns(*prefix*)
Is called after the `end()` callback of an element that declared a namespace prefix mapping, with the name of the *prefix* that went out of scope.

バージョン 3.8 で追加.

class xml.etree.ElementTree.**C14NWriterTarget**(*write*, ***, *with_comments=False*, *strip_text=False*, *rewrite_prefixes=False*, *qname_aware_tags=None*, *qname_aware_attrs=None*, *exclude_attrs=None*, *exclude_tags=None*)
A C14N 2.0 writer. Arguments are the same as for the *canonicalize()* function. This class does not build a tree but translates the callback events directly into a serialised form using the *write* function.

バージョン 3.8 で追加.

XMLParser オブジェクト

class xml.etree.ElementTree.**XMLParser**(***, *target=None*, *encoding=None*)
このクラスは、このモジュールの構成要素のうち、低水準のものです。効率的でイベントベースの XML パースのため、*xml.parsers.expat* を使用します。*feed()* メソッドで XML データをインクリメンタルに受け取り、*target* オブジェクトのコールバックを呼び出すことで、パースイベントをプッシュ API に変換します。*target* が省略された場合、標準の *TreeBuilder* が使用されます。*encoding*[1] が指定された場合、このあたいは XML ファイル内で指定されたエンコーディングを上書きします。

バージョン 3.8 で変更: Parameters are now *keyword-only*. The *html* argument no longer supported.

close()
パーザへのデータの提供を完了します。構築中に渡される *target* の `close()` メソッドを呼び出す

結果を返します。既定では、これがトップレベルのドキュメント要素になります。

feed(*data*)

パーザへデータを入力します。*data* はエンコードされたデータです。

flush()

Triggers parsing of any previously fed unparsed data, which can be used to ensure more immediate feedback, in particular with Expat >=2.6.0. The implementation of *flush()* temporarily disables reparse deferral with Expat (if currently enabled) and triggers a reparse. Disabling reparse deferral has security consequences; please see *xml.parsers.expat.xmlparser.SetReparseDeferralEnabled()* for details.

Note that *flush()* has been backported to some prior releases of CPython as a security fix. Check for availability of *flush()* using *hasattr()* if used in code running across a variety of Python versions.

バージョン 3.8.19 で追加.

XMLParser.feed() は *target* の `start(tag, attrs_dict)` メソッドをそれぞれの開始タグに対して呼び、また `end(tag)` メソッドを終了タグに対して呼び、そしてデータを `data(data)` メソッドで処理します。サポートされているその他のコールバックメソッドについては、*TreeBuilder* クラスを参照してください。*XMLParser.close()* は *target* の `close()` メソッドを呼びます。*XMLParser* は木構造を構築する以外にも使えます。以下の例では、XML ファイルの最高の深さを数えます。

```
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:                     # The target object of the parser
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):   # Called for each opening tag.
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):             # Called for each closing tag.
...         self.depth -= 1
...     def data(self, data):
...         pass            # We do not need to do anything with data.
...     def close(self):    # Called when all data has been parsed.
...         return self.maxDepth
...
>>> target = MaxDepth()
>>> parser = XMLParser(target=target)
>>> exampleXml = """
... <a>
...   <b>
...   </b>
...   <b>
...     <c>
...       <d>
...       </d>
...     </c>
...   </b>
```

(次のページに続く)

(前のページからの続き)

```
... </a>"""
>>> parser.feed(exampleXml)
>>> parser.close()
4
```

XMLPullParser オブジェクト

class xml.etree.ElementTree.**XMLPullParser**(*events=None*)

非ブロックアプリケーションに適したプルパーザです。入力側の API は *XMLParser* のものと似ていますが、コールバックターゲットに呼び出しをプッシュするのではなく、*XMLPullParser* はパースイベントの内部リストを収集し、ユーザーがそこから読み出すことができます。*events* は、呼び出し元に報告するイベントのシーケンスです。サポートされているイベントは、文字列の `"start"`, `"end"`, `"comment"`, `"pi"`, `"start-ns"`, `"end-ns"` (”ns” イベントは、名前空間の詳細情報の取得に使用) です。*events* が省略された場合、`"end"` イベントのみが報告されます。

feed(*data*)

指定したバイトデータをパーザに与えます。

flush()

Triggers parsing of any previously fed unparsed data, which can be used to ensure more immediate feedback, in particular with Expat >=2.6.0. The implementation of *flush()* temporarily disables reparse deferral with Expat (if currently enabled) and triggers a reparse. Disabling reparse deferral has security consequences; please see *xml.parsers.expat.xmlparser.SetReparseDeferralEnabled()* for details.

Note that *flush()* has been backported to some prior releases of CPython as a security fix. Check for availability of *flush()* using *hasattr()* if used in code running across a variety of Python versions.

バージョン 3.8.19 で追加.

close()

パーザに、データストリームが終了したことを伝えます。*XMLParser.close()* とは異なり、このメソッドは常に *None* を返します。パーザがクローズした時にまだ帰って来ていないイベントは、まだ *read_events()* で読むことができます。

read_events()

Return an iterator over the events which have been encountered in the data fed to the parser. The iterator yields `(event, elem)` pairs, where *event* is a string representing the type of event (e.g. `"end"`) and *elem* is the encountered *Element* object, or other context value as follows.

- `start`, `end`: the current Element.
- `comment`, `pi`: the current comment / processing instruction
- `start-ns`: a tuple `(prefix, uri)` naming the declared namespace mapping.
- `end-ns`: *None* (this may change in a future version)

read_events() の前の呼び出しで提供されたイベントは、再度 yield されることはありません。イベントは、イテレータから取得された場合にのみ内部キューから消費されるため、*read_events()* から取得されたイテレータに対して複数の読み出しを並行して反復的に行うと、予期せぬ結果が引き起こされます。

注釈: *XMLPullParser* は "start" イベントを発行した時に開始タグの文字 ">" が現れたことだけを保証します。そのため、属性は定義されますが、その時点ではテキストの内容も tail 属性も定義されていません。子要素にもそれが存在する、しないにかかわらず同じ物が適用されます。

全部揃った要素が必要ならば、"end" イベントを探してください。

バージョン 3.4 で追加.

バージョン 3.8 で変更: イベント `comment`, `pi` が追加されました。

例外

`class xml.etree.ElementTree.ParseError`

解析に失敗した時、このモジュールの様々なメソッドから送出される XML 解析エラーです。この例外のインスタンスが表す文字列は、ユーザフレンドリなメッセージを含んでいます。その他に、以下の属性も利用できます:

`code`

expat パーザからの数値エラーコードです。エラーコードの一覧とそれらの意味については、*xml.parsers.expat* のドキュメントを参照してください。

`position`

エラーが発生した場所を示す *line* と *column* 番号のタプルです。

脚注

20.6 xml.dom --- 文書オブジェクトモデル (DOM) API

ソースコード: Lib/xml/dom/__init__.py

文書オブジェクトモデル (Document Object Model)、すなわち "DOM" は、ワールドワイドウェブコンソーシアム (World Wide Web Consortium, W3C) による、XML ドキュメントにアクセスしたり変更を加えたりするための、プログラミング言語間共通の API です。DOM 実装によって、XML ドキュメントはツリー構造として表現されます。また、クライアントコード側でツリー構造をゼロから構築できるようになります。さらに、前述の構造に対して、よく知られたインタフェースをもつ一連のオブジェクトを通したアクセス手段も提供します。

DOM はランダムアクセスを行うアプリケーションで非常に有用です。SAX では、一度に閲覧することができるのはドキュメントのほんの一部分です。ある SAX 要素に注目している際には、別の要素をアクセスする

ことはできません。またテキストノードに注目しているときには、その中に入っている要素をアクセスすることができません。SAX によるアプリケーションを書くときには、プログラムがドキュメント内のどこを処理しているのかを追跡するよう、コードのどこかに記述する必要があります。SAX 自体がその作業を行ってくれることはありません。さらに、XML ドキュメントに対する先読み (look ahead) が必要だとすると不運なことになります。

アプリケーションによっては、ツリーにアクセスできなければイベント駆動モデルを実現できません。もちろん、何らかのツリーを SAX イベントに応じて自分で構築することもできるでしょうが、DOM ではそのようなコードを書かなくてもよくなります。DOM は XML データに対する標準的なツリー表現なのです。

文書オブジェクトモデルは、W3C によっていくつかの段階、W3C の用語で言えば " レベル (level)" で定義されています。Python においては、DOM API への対応付けは実質的には DOM レベル 2 勧告に基づいています。

DOM アプリケーションは、普通は XML を DOM に解析するところから始まります。どのようにして解析を行うかについては DOM レベル 1 では全くカバーしておらず、レベル 2 では限定的な改良だけが行われました: レベル 2 では `Document` を生成するメソッドを提供する `DOMImplementation` オブジェクトクラスがありますが、実装に依存しない方法で XML リーダ (reader)/パーザ (parser)/文書ビルダ (Document builder) にアクセスする方法はありません。また、既存の `Document` オブジェクトなしにこれらのメソッドにアクセスするような、よく定義された方法もありません。Python では、各々の DOM 実装で *`getDOMImplementation()`* が定義されているはずです。DOM レベル 3 ではロード (Load)/ストア (Store) 仕様が追加され、リーダのインタフェースにを定義していますが、Python 標準ライブラリではまだ利用することができません。

DOM 文書オブジェクトを生成したら、そのプロパティとメソッドを使って XML 文書の一部にアクセスできます。これらのプロパティは DOM 仕様で定義されています; 本リファレンスマニュアルでは、Python において DOM 仕様がどのように解釈されているかを記述しています。

W3C から提供されている仕様は、DOM API を Java、ECMAScript、および OMG IDL で定義しています。ここで定義されている Python での対応づけは、大部分がこの仕様の IDL 版に基づいていますが、厳密な準拠は必要とされていません (実装で IDL の厳密な対応付けをサポートするのは自由ですが)。API への対応付けに関する詳細な議論は **適合性** を参照してください。

参考:

Document Object Model (DOM) Level 2 Specification　Python DOM API が準拠している W3C 勧告。

Document Object Model (DOM) Level 1 Specification　*`xml.dom.minidom`* でサポートされている W3C の DOM に関する勧告。

Python Language Mapping Specification　このドキュメントでは OMG IDL から Python への対応付けを記述しています。

20.6.1 モジュールコンテンツ

xml.dom には以下の関数があります:

xml.dom.**registerDOMImplementation**(*name*, *factory*)

ファクトリ関数 (factory function) *factory* を名前 *name* で登録します。ファクトリ関数は DOMImplementation インタフェースを実装するオブジェクトを返さなければなりません。特定の実装 (例えば実装が何らかのカスタマイズをサポートしている場合) に適切となるように、ファクトリ関数は毎回同じオブジェクトを返したり、呼び出しごとに新しいオブジェクトを返したりすることが出来ます。

xml.dom.**getDOMImplementation**(*name=None*, *features=()*)

適切な DOM 実装を返します。*name* は、よく知られた DOM 実装のモジュール名か、None になります。None でない場合、対応するモジュールを import して、import が成功した場合 DOMImplementation オブジェクトを返します。*name* が与えられておらず、環境変数 PYTHON_DOM が設定されていた場合、DOM 実装を見つけるのに環境変数が使われます。

name が与えられない場合、利用可能な実装を調べて、指定された機能 (feature) セットを持つものを探します。実装が見つからなければ *ImportError* を送出します。*features* のリストは (feature, version) のペアからなるシーケンスで、利用可能な DOMImplementation オブジェクトの hasFeature() メソッドに渡されます。

いくつかの便利な定数も提供されています:

xml.dom.**EMPTY_NAMESPACE**

DOM 内のノードに名前空間が何も関連づけられていないことを示すために使われる値です。この値は通常、ノードの namespaceURI の値として見つかったり、名前空間特有のメソッドに対する *namespaceURI* パラメタとして使われます。

xml.dom.**XML_NAMESPACE**

Namespaces in XML (4 節) で定義されている、予約済みプレフィクス (reserved prefix) xml に関連付けられた名前空間 URI です。

xml.dom.**XMLNS_NAMESPACE**

Document Object Model (DOM) Level 2 Core Specification (1.1.8 節) で定義されている、名前空間宣言への名前空間 URI です。

xml.dom.**XHTML_NAMESPACE**

XHTML 1.0: The Extensible HyperText Markup Language (3.1.1 節) で定義されている、XHTML 名前空間 URI です。

加えて、*xml.dom* には基底となる Node クラスと DOM 例外クラスが収められています。このモジュールで提供されている Node クラスは DOM 仕様で定義されているメソッドや属性は何ら実装していません; これらは具体的な DOM 実装において提供しなければなりません。このモジュールの一部として提供されている Node クラスでは、具体的な Node オブジェクトの nodeType 属性として使う定数を提供しています; これらの定数は、DOM 仕様に適合するため、クラスではなくモジュールのレベルに配置されています。

20.6.2 DOM 内のオブジェクト

DOM について最も明確に限定しているドキュメントは W3C による DOM 仕様です。

DOM 属性は単純な文字列としてだけではなく、ノードとして操作されるかもしれないので注意してください。とはいえ、そうしなければならない場合はかなり稀なので、今のところ記述されていません。

インタフェース	節	目的
DOMImplementation	*DOMImplementation* オブジェクト	根底にある実装へのインタフェース。
Node	*Node* オブジェクト	ドキュメント内の大部分のオブジェクトに対する基底インタフェース。
NodeList	*NodeList* オブジェクト	ノード列に対するインタフェース。
DocumentType	*DocumentType* オブジェクト	ドキュメントの処理に必要な宣言についての情報。
Document	*Document* オブジェクト	ドキュメント全体を表現するオブジェクト。
Element	*Element* オブジェクト	ドキュメント階層内の要素ノード。
Attr	*Attr* オブジェクト	要素ノード上の属性値ノード。
Comment	*Comment* オブジェクト	ソースドキュメント内のコメント表現。
Text	*Text* オブジェクトおよび *CDATASection* オブジェクト	ドキュメント内のテキスト記述を含むノード。
ProcessingInstruction	*ProcessingInstruction* オブジェクト	処理命令 (processing instruction) 表現。

さらに追加の節として、Python で DOM を利用するために定義されている例外について記述しています。

DOMImplementation オブジェクト

DOMImplementation インタフェースは、利用している DOM 実装において特定の機能が利用可能かどうかを決定するための方法をアプリケーションに提供します。DOM レベル 2 では、DOMImplementation を使って新たな Document オブジェクトや DocumentType オブジェクトを生成する機能も追加しています。

DOMImplementation.**hasFeature**(*feature*, *version*)
機能名 *feature* とバージョン番号 *version* で識別される機能（feature）が実装されていれば True を返します。

DOMImplementation.**createDocument**(*namespaceUri*, *qualifiedName*, *doctype*)
新たな (DOM のスーパークラスである) Document クラスのオブジェクトを返します。このクラスは *namespaceUri* と *qualifiedName* が設定された子クラス Element のオブジェクトを所有しています。*doctype* は *createDocumentType()* によって生成された DocumentType クラスのオブジェクト、または None である必要があります。Python DOM API では、子クラスである Element を作成しないことを示すために、はじめの 2 つの引数を None に設定することができます。

DOMImplementation.**createDocumentType**(*qualifiedName*, *publicId*, *systemId*)
新たな DocumentType クラスのオブジェクトを返します。このオブジェクトは *qualifiedName* 、*publicId* 、そして *systemId* 文字列をふくんでおり、XML 文書の形式情報を表現しています。

Node オブジェクト

XML 文書の全ての構成要素は Node のサブクラスです。

Node.**nodeType**
: ノード (node) の型を表現する整数値です。型に対応する以下のシンボル定数: ELEMENT_NODE 、 ATTRIBUTE_NODE 、 TEXT_NODE 、 CDATA_SECTION_NODE 、 ENTITY_NODE 、 PROCESSING_INSTRUCTION_NODE 、 COMMENT_NODE 、 DOCUMENT_NODE 、 DOCUMENT_TYPE_NODE 、 NOTATION_NODE 、が Node オブジェクトで定義されています。読み出し専用の属性です。

Node.**parentNode**
: 現在のノードの親ノードか、文書ノードの場合には None になります。この値は常に Node オブジェクトか None になります。Element ノードの場合、この値はルート要素 (root element) の場合を除き親要素 (parent element) となり、ルート要素の場合には Document オブジェクトとなります。Attr ノードの場合、この値は常に None となります。読み出し専用の属性です。

Node.**attributes**
: 属性オブジェクトの NamedNodeMap です。要素だけがこの属性に実際の値を持ちます; その他のオブジェクトでは、この属性を None にします。読み出し専用の属性です。

Node.**previousSibling**
: このノードと同じ親ノードを持ち、直前にくるノードです。例えば、*self* 要素の開始タグの直前にくる終了タグを持つ要素です。もちろん、XML 文書は要素だけで構成されているだけではないので、直前にくる兄弟関係にある要素 (sibling) はテキストやコメント、その他になる可能性があります。このノードが親ノードにおける先頭の子ノードである場合、属性値は None になります。読み出し専用の属性です。

Node.**nextSibling**
: このノードと同じ親ノードを持ち、直後にくるノードです。例えば、*previousSibling* も参照してください。このノードが親ノードにおける末尾の子ノードである場合、属性値は None になります。読み出し専用の属性です。

Node.**childNodes**
: このノード内に収められているノードからなるリストです。読み出し専用の属性です。

Node.**firstChild**
: このノードに子ノードがある場合、その先頭のノードです。そうでない場合 None になります。読み出し専用の属性です。

Node.**lastChild**
: このノードに子ノードがある場合、その末尾のノードです。そうでない場合 None になります。読み出し専用の属性です。

Node.**localName**
: tagName にコロンがあれば、コロン以降の部分に、なければ tagName 全体になります。値は文字列です。

Node.**prefix**
: tagName のコロンがあれば、コロン以前の部分に、なければ空文字列になります。値は文字列か、None

になります。

Node.**namespaceURI**
要素名に関連付けられた名前空間です。文字列か None になります。読み出し専用の属性です。

Node.**nodeName**
この属性はノード型ごとに異なる意味を持ちます。その詳細は DOM 仕様を参照してください。この属性で得られることになる情報は、全てのノード型では tagName 、属性では name プロパティといったように、常に他のプロパティで得ることができます。全てのノード型で、この属性の値は文字列か None になります。読み出し専用の属性です。

Node.**nodeValue**
この属性はノード型ごとに異なる意味を持ちます。その詳細は DOM 仕様を参照してください。その状況は *nodeName* と似ています。この属性の値は文字列または None になります。

Node.**hasAttributes**()
ノードが何らかの属性を持っている場合に True を返します。

Node.**hasChildNodes**()
ノードが何らかの子ノードを持っている場合に True を返します。

Node.**isSameNode**(*other*)
other がこのノードと同じノードを参照している場合に True を返します。このメソッドは、何らかのプロキシ (proxy) 機構を利用するような DOM 実装で特に便利です (一つ以上のオブジェクトが同じノードを参照するかもしれないからです)。

注釈: このメソッドは DOM レベル 3 API の提案に基づいたもので、まだ ” ワーキングドラフト (working draft)” の段階です。しかし、このインタフェースには異論は出ないと考えられます。W3C による変更があっても、必ずしも Python DOM インタフェースにおけるこのメソッドに影響するとは限りません (ただしこのメソッドに対する何らかの新しい W3C API もサポートされるかもしれません)。

Node.**appendChild**(*newChild*)
現在のノードの子ノードリストの末尾に新たな子ノードを追加し、*newChild* を返します。もしノードが既にツリーにあれば、最初に削除されます。

Node.**insertBefore**(*newChild*, *refChild*)
新たな子ノードを既存の子ノードの前に挿入します。*refChild* は現在のノードの子ノードである場合に限られます; そうでない場合、*ValueError* が送出されます。*newChild* が返されます。もし *refChild* が None なら、*newChild* を子ノードリストの最後に挿入します。

Node.**removeChild**(*oldChild*)
子ノードを削除します。*oldChild* はこのノードの子ノードでなければなりません。そうでない場合、*ValueError* が送出されます。成功した場合 *oldChild* が返されます。*oldChild* をそれ以降使わない場合、unlink() メソッドを呼び出さなければなりません。

Node.**replaceChild**(*newChild*, *oldChild*)

既存のノードと新たなノードを置き換えます。この操作は *oldChild* が現在のノードの子ノードである場合に限られます; そうでない場合、*ValueError* が送出されます。

Node.**normalize**()
一続きのテキスト全体を一個の Text インスタンスとして保存するために隣接するテキストノードを結合します。これにより、多くのアプリケーションで DOM ツリーからのテキスト処理が簡単になります。

Node.**cloneNode**(*deep*)
このノードを複製 (clone) します。*deep* を設定すると、子ノードも同様に複製することを意味します。複製されたノードを返します。

NodeList オブジェクト

NodeList はノードのシーケンスを表現します。これらのオブジェクトは DOM コア勧告 (DOM Core recommendation) において、二通りに使われています: Element オブジェクトでは、子ノードのリストを提供するのに NodeList を利用します。また、このインタフェースにおける Node の getElementsByTagName() および getElementsByTagNameNS() メソッドは、クエリに対する結果を表現するのに NodeList を利用します。

DOM レベル 2 勧告では、これらのオブジェクトに対し、以下のようにメソッドを一つ、属性を一つ定義しています。

NodeList.**item**(*i*)
存在する場合シークエンスの i 番目の要素を、そうでない場合 None を返します。i は 0 未満やシークエンスの長さ以上であってはなりません。

NodeList.**length**
シーケンス中のノードの数です。

この他に、Python の DOM インタフェースでは、NodeList オブジェクトを Python のシーケンスとして使えるようにするサポートが追加されていることが必要です。NodeList の実装では、全て __len__() と __getitem__() をサポートしなければなりません; このサポートにより、for 文内で NodeList にわたる繰り返しと、組み込み関数 *len()* の適切なサポートができるようになります。

DOM 実装が文書の変更をサポートしている場合、NodeList の実装でも __setitem__() および __delitem__() メソッドをサポートしなければなりません。

DocumentType オブジェクト

文書で宣言されている記法 (notation) やエンティティ (entity) に関する (外部サブセット (external subset) がパーザから利用でき、情報を提供できる場合にはそれも含めた) 情報は、DocumentType オブジェクトから手に入れることができます。文書の DocumentType は、Document オブジェクトの doctype 属性で入手することができます; 文書の DOCTYPE 宣言がない場合、文書の doctype 属性は、このインタフェースを持つインスタンスの代わりに None に設定されます。

DocumentType は Node を特殊化したもので、以下の属性を加えています:

DocumentType.**publicId**
: 文書型定義 (document type definition) の外部サブセットに対する公開識別子 (public identifier) です。文字列または None になります。

DocumentType.**systemId**
: 文書型定義 (document type definition) の外部サブセットに対するシステム識別子 (system identifier) です。文字列の URI または None になります。

DocumentType.**internalSubset**
: ドキュメントの完全な内部サブセットを与える文字列です。サブセットを囲むブラケットは含みません。ドキュメントが内部サブセットを持たない場合、この値は None です。

DocumentType.**name**
: DOCTYPE 宣言でルート要素の名前が与えられている場合、その値になります。

DocumentType.**entities**
: 外部エンティティの定義を与える NamedNodeMap です。複数回定義されているエンティティに対しては、最初の定義だけが提供されます (その他は XML 勧告での要求仕様によって無視されます)。パーザによって情報が提供されないか、エンティティが定義されていない場合には、この値は None になることがあります。

DocumentType.**notations**
: 記法の定義を与える NamedNodeMap です。複数回定義されている記法名に対しては、最初の定義だけが提供されます (その他は XML 勧告での要求仕様によって無視されます)。パーザによって情報が提供されないか、エンティティが定義されていない場合には、この値は None になることがあります。

Document オブジェクト

Document は XML ドキュメント全体を表現し、その構成要素である要素、属性、処理命令、コメント等を持ちます。Document は Node からプロパティを継承していることを思い出してください。

Document.**documentElement**
: ドキュメントの唯一無二のルート要素です。

Document.**createElement**(*tagName*)
: 新たな要素ノードを生成して返します。要素は、生成された時点ではドキュメント内に挿入されません。insertBefore() や appendChild() のような他のメソッドの一つを使って明示的に挿入を行う必要があります。

Document.**createElementNS**(*namespaceURI*, *tagName*)
: 名前空間を伴う新たな要素ノードを生成して返します。*tagName* には接頭辞 (prefix) があってもかまいません。要素は、生成された時点では文書内に挿入されません。insertBefore() や appendChild() のような他のメソッドの一つを使って明示的に挿入を行う必要があります。

Document.**createTextNode**(*data*)
: 引数として渡されたデータの入ったテキストノードを生成して返します。他の生成 (create) メソッドと同じく、このメソッドは生成されたノードをツリーに挿入しません。

Document.**createComment**(*data*)
引数として渡されたデータの入ったコメントノードを生成して返します。他の生成 (create) メソッドと同じく、このメソッドは生成されたノードをツリーに挿入しません。

Document.**createProcessingInstruction**(*target*, *data*)
引数として渡された *target* および *data* の入った処理命令ノードを生成して返します。他の生成 (create) メソッドと同じく、このメソッドは生成されたノードをツリーに挿入しません。

Document.**createAttribute**(*name*)
属性ノードを生成して返します。このメソッドは属性ノードを特定の要素に関連づけることはしません。新たに生成された属性インスタンスを使うには、適切な Element オブジェクトの setAttributeNode() を使わなければなりません。

Document.**createAttributeNS**(*namespaceURI*, *qualifiedName*)
名前空間を伴う新たな属性ノードを生成して返します。*tagName* には接頭辞 (prefix) があってもかまいません。このメソッドは属性ノードを特定の要素に関連づけることはしません。新たに生成された属性インスタンスを使うには、適切な Element オブジェクトの setAttributeNode() を使わなければなりません。

Document.**getElementsByTagName**(*tagName*)
全ての下位要素 (直接の子要素、子要素の子要素等) から特定の要素型名を持つものを検索します。

Document.**getElementsByTagNameNS**(*namespaceURI*, *localName*)
全ての下位要素 (直接の子要素、子要素の子要素等) から特定の名前空間 URI とローカル名 (local name) を持つものを検索します。ローカル名は名前空間における接頭辞以降の部分です。

Element オブジェクト

Element は Node のサブクラスです。このため Node クラスの全ての属性を継承します。

Element.**tagName**
要素型名です。名前空間使用の文書では、要素型名中にコロンがあるかもしれません。値は文字列です。

Element.**getElementsByTagName**(*tagName*)
Document クラス内における同名のメソッドと同じです。

Element.**getElementsByTagNameNS**(*namespaceURI*, *localName*)
Document クラス内における同名のメソッドと同じです。

Element.**hasAttribute**(*name*)
指定要素に *name* で渡した名前の属性が存在していれば True を返します。

Element.**hasAttributeNS**(*namespaceURI*, *localName*)
指定要素に *namespaceURI* と *localName* で指定した名前の属性が存在していれば True を返します。

Element.**getAttribute**(*name*)
name で指定した属性の値を文字列として返します。もし、属性が存在しない、もしくは属性に値が設定されていない場合、空の文字列が返されます。

Element.**getAttributeNode**(*attrname*)

attrname で指定された属性の Attr ノードを返します。

Element.**getAttributeNS**(*namespaceURI*, *localName*)

namespaceURI と *localName* によって指定した属性の値を文字列として返します。もし、属性が存在しない、もしくは属性に値が設定されていない場合、空の文字列が返されます。

Element.**getAttributeNodeNS**(*namespaceURI*, *localName*)

指定した *namespaceURI* および *localName* を持つ属性値をノードとして返します。

Element.**removeAttribute**(*name*)

名前で指定された属性を削除します。該当する属性がない場合、*NotFoundErr* が送出されます。

Element.**removeAttributeNode**(*oldAttr*)

oldAttr が属性リストにある場合、削除して返します。*oldAttr* が存在しない場合、*NotFoundErr* が送出されます。

Element.**removeAttributeNS**(*namespaceURI*, *localName*)

名前で指定された属性を削除します。このメソッドは *qname* ではなく *localName* を使うので注意してください。該当する属性がなくても例外は送出されません。

Element.**setAttribute**(*name*, *value*)

文字列を使って属性値を設定します。

Element.**setAttributeNode**(*newAttr*)

新たな属性ノードを要素に追加します。name 属性が既存の属性に一致した場合、必要に応じて属性を置換します。置換が行われると古い属性ノードが返されます。*newAttr* がすでに使われていれば、*InuseAttributeErr* が送出されます。

Element.**setAttributeNodeNS**(*newAttr*)

新たな属性ノードを要素に追加します。namespaceURI および localName 属性が既存の属性に一致した場合、必要に応じて属性を置き換えます。置換が生じると、古い属性ノードが返されます。*newAttr* がすでに使われていれば、*InuseAttributeErr* が送出されます。

Element.**setAttributeNS**(*namespaceURI*, *qname*, *value*)

指定された *namespaceURI* および *qname* で与えられた属性の値を文字列で設定します。qname は属性の完全な名前であり、この点が上記のメソッドと違うので注意してください。

Attr オブジェクト

Attr は Node を継承しており、全ての属性を継承しています。

Attr.**name**

要素型名です。名前空間使用の文書では、要素型名中にコロンが含まれるかもしれません。

Attr.**localName**

名前にコロンがあればコロン以降の部分に、なければ名前全体になります。

Attr.prefix
: 名前にコロンがあればコロン以前の部分に、なければ空文字列になります。

Attr.value
: その要素の text value. これは nodeValue 属性の別名です。

NamedNodeMap Objects

NamedNodeMap は Node を継承して **いません** 。

NamedNodeMap.length
: 属性リストの長さです。

NamedNodeMap.item(*index*)
: 特定のインデクスを持つ属性を返します。属性の並び方は任意ですが、DOM 文書が生成されている間は一定になります。各要素は属性ノードです。属性値はノードの value 属性で取得してください。

このクラスをよりマップ型的な動作ができるようにする実験的なメソッドもあります。そうしたメソッドを使うこともできますし、Element オブジェクトに対して、標準化された getAttribute*() ファミリのメソッドを使うこともできます。

Comment オブジェクト

Comment は XML 文書中のコメントを表現します。Comment は Node のサブクラスですが、子ノードを持つことはありません。

Comment.data
: 文字列によるコメントの内容です。この属性には、コメントの先頭にある <!- - と末尾にある - -> 間の全ての文字が入っていますが、<!- - と - -> 自体は含みません。

Text オブジェクトおよび CDATASection オブジェクト

Text インタフェースは XML 文書内のテキストを表現します。パーザおよび DOM 実装が DOM の XML 拡張をサポートしている場合、CDATA でマークされた区域 (section) に入れられている部分テキストは CDATASection オブジェクトに記憶されます。これら二つのインタフェースは同一のものですが、nodeType 属性が異なります。

これらのインタフェースは Node インタフェースを拡張したものです。しかし子ノードを持つことはできません。

Text.data
: 文字列によるテキストノードの内容です。

注釈: CDATASection ノードの利用は、ノードが完全な CDATA マーク区域を表現するという意味ではなく、ノードの内容が CDATA 区域の一部であるということを意味するだけです。単一の CDATA セクション

は文書ツリー内で複数のノードとして表現されることがあります。二つの隣接する CDATASection ノードが、異なる CDATA マーク区域かどうかを決定する方法はありません。

ProcessingInstruction オブジェクト

XML 文書内の処理命令を表現します; Node インタフェースを継承していますが、子ノードを持つことはできません。

ProcessingInstruction.**target**
: 最初の空白文字までの処理命令の内容です。読み出し専用の属性です。

ProcessingInstruction.**data**
: 最初の空白文字以降の処理命令の内容です。

例外

DOM レベル 2 勧告では、単一の例外 *DOMException* と、どの種のエラーが発生したかをアプリケーションが決定できるようにする多くの定数を定義しています。*DOMException* インスタンスは、特定の例外に関する適切な値を提供する *code* 属性を伴っています。

Python DOM インタフェースでは、上記の定数を提供していますが、同時に一連の例外を拡張して、DOM で定義されている各例外コードに対して特定の例外が存在するようにしています。DOM の実装では、適切な特定の例外を送出しなければならず、各例外は *code* 属性に対応する適切な値を伴わなければなりません。

exception xml.dom.DOMException
: 全ての特定の DOM 例外で使われている基底例外クラスです。この例外クラスを直接インスタンス化することはできません。

exception xml.dom.DomstringSizeErr
: 指定された範囲のテキストが文字列に収まらない場合に送出されます。この例外は Python の DOM 実装で使われるかどうかは判っていませんが、Python で書かれていない DOM 実装から送出される場合があります。

exception xml.dom.HierarchyRequestErr
: 挿入できない型のノードを挿入しようと試みたときに送出されます。

exception xml.dom.IndexSizeErr
: メソッドに与えたインデクスやサイズパラメタが負の値や許容範囲の値を超えた際に送出されます。

exception xml.dom.InuseAttributeErr
: 文書中にすでに存在する Attr ノードを挿入しようと試みた際に送出されます。

exception xml.dom.InvalidAccessErr
: パラメタまたは操作が根底にあるオブジェクトでサポートされていない場合に送出されます。

exception xml.dom.InvalidCharacterErr
: この例外は、文字列パラメタが、現在使われているコンテキストで XML 1.0 勧告によって許可されて

いない場合に送出されます。例えば、要素型に空白の入った Element ノードを生成しようとすると、このエラーが送出されます。

exception xml.dom.InvalidModificationErr
ノードの型を変更しようと試みた際に送出されます。

exception xml.dom.InvalidStateErr
定義されていないオブジェクトや、もはや利用できなくなったオブジェクトを使おうと試みた際に送出されます。

exception xml.dom.NamespaceErr
Namespaces in XML に照らして許可されていない方法でオブジェクトを変更しようと試みた場合、この例外が送出されます。

exception xml.dom.NotFoundErr
参照しているコンテキスト中に目的のノードが存在しない場合に送出される例外です。例えば、NamedNodeMap.removeNamedItem() は渡されたノードがノードマップ中に存在しない場合にこの例外を送出します。

exception xml.dom.NotSupportedErr
要求された方のオブジェクトや操作が実装でサポートされていない場合に送出されます。

exception xml.dom.NoDataAllowedErr
データ属性をサポートしないノードにデータを指定した際に送出されます。

exception xml.dom.NoModificationAllowedErr
オブジェクトに対して (読み出し専用ノードに対する修正のように) 許可されていない修正を行おうと試みた際に送出されます。

exception xml.dom.SyntaxErr
無効または不正な文字列が指定された際に送出されます。

exception xml.dom.WrongDocumentErr
ノードが現在属している文書と異なる文書に挿入され、かつある文書から別の文書へのノードの移行が実装でサポートされていない場合に送出されます。

DOM 勧告で定義されている例外コードは、以下のテーブルに従って上記の例外と対応付けられます:

定数	例外
DOMSTRING_SIZE_ERR	*DomstringSizeErr*
HIERARCHY_REQUEST_ERR	*HierarchyRequestErr*
INDEX_SIZE_ERR	*IndexSizeErr*
INUSE_ATTRIBUTE_ERR	*InuseAttributeErr*
INVALID_ACCESS_ERR	*InvalidAccessErr*
INVALID_CHARACTER_ERR	*InvalidCharacterErr*
INVALID_MODIFICATION_ERR	*InvalidModificationErr*
INVALID_STATE_ERR	*InvalidStateErr*
NAMESPACE_ERR	*NamespaceErr*
NOT_FOUND_ERR	*NotFoundErr*
NOT_SUPPORTED_ERR	*NotSupportedErr*
NO_DATA_ALLOWED_ERR	*NoDataAllowedErr*
NO_MODIFICATION_ALLOWED_ERR	*NoModificationAllowedErr*
SYNTAX_ERR	*SyntaxErr*
WRONG_DOCUMENT_ERR	*WrongDocumentErr*

20.6.3 適合性

この節では適合性に関する要求と、Python DOM API、W3C DOM 勧告、および OMG IDL の Python API への対応付けとの間の関係について述べます。

型の対応付け

DOM 仕様で使われている IDL 型は、以下のテーブルに従って Python の型に対応付けられています。

IDL 型	Python の型
boolean	bool または int
int	int
long int	int
unsigned int	int
DOMString	str または bytes
null	None

アクセサメソッド

OMG IDL から Python への対応付けは、IDL `attribute` 宣言へのアクセサ関数の定義を、Java による対応付けが行うのとほとんど同じように行います。IDL 宣言の対応付け

```
readonly attribute string someValue;
         attribute string anotherValue;
```

は、三つのアクセサ関数: `someValue` に対する "get" メソッド (`_get_someValue()`)、そして `anotherValue` に対する "get" および "set" メソッド (`_get_anotherValue()` および `_set_anotherValue()`) を生成します。とりわけ、対応付けでは、IDL 属性が通常の Python 属性としてアクセス可能であることは必須ではありません: `object.someValue` が動作することは必須 **ではなく** 、*AttributeError* を送出してもかまいません。

しかしながら、Python DOM API では、通常の属性アクセスが動作することが必須です。これは、Python IDL コンパイラによって生成された典型的な代用物はまず動作することはなく、DOM オブジェクトが CORBA を介してアクセスされる場合には、クライアント上でラッパオブジェクトが必要であることを意味します。CORBA DOM クライアントでは他にもいくつか考慮すべきことがある一方で、Python から CORBA を介して DOM を使った経験を持つ実装者はこのことを問題視していません。`readonly` であると宣言された属性は、全ての DOM 実装で書き込みアクセスを制限しているとは限りません。

Python DOM API では、アクセサ関数は必須ではありません。アクセサ関数が提供された場合、Python IDL 対応付けによって定義された形式をとらなければなりませんが、属性は Python から直接アクセスすることができるので、それらのメソッドは必須ではないと考えられます。`readonly` であると宣言された属性に対しては、"set" アクセサを提供してはなりません。

この IDL での定義は W3C DOM API の全ての要件を実装しているわけではありません。例えば、一部のオブジェクトの概念や `getElementsByTagName()` が "live" であることなどです。Python DOM API はこれらの要件を実装することを強制しません。

20.7 `xml.dom.minidom` --- 最小限の DOM の実装

ソースコード: Lib/xml/dom/minidom.py

---

xml.dom.minidom は、Document Object Model インタフェースの最小の実装です。他言語の実装と似た API を持ちます。このモジュールは、完全な DOM に比べて単純で、非常に小さくなるように意図されています。DOM について既に熟知しているユーザを除き、XML 処理には代わりに *xml.etree.ElementTree* モジュールを使うことを検討すべきです。

警告: *xml.dom.minidom* モジュールは悪意を持って作成されたデータに対して安全ではありません。信頼できないデータや認証されていないデータをパースする必要がある場合は *XML* **の脆弱性** を参照してください。

DOM アプリケーションは通常、XML を DOM に解析 (parse) することで開始します。*xml.dom.minidom* では、以下のような解析用の関数を介して行います:

```
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\\temp\\mydata.xml')  # parse an XML file by name

datasource = open('c:\\temp\\mydata.xml')
dom2 = parse(datasource)  # parse an open file

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')
```

parse() 関数はファイル名か、開かれたファイルオブジェクトを引数にとることができます。

xml.dom.minidom.**parse**(*filename_or_file*, *parser=None*, *bufsize=None*)

与えられた入力から Document を返します。*filename_or_file* はファイル名でもファイルオブジェクトでもかまいません。*parser* を指定する場合、SAX2 パーザオブジェクトでなければなりません。この関数はパーザの文書ハンドラを変更し、名前空間サポートを有効にします; (エンティティリゾルバ (entity resolver) のような) 他のパーザ設定は前もっておこなわなければなりません。

XML データを文字列で持っている場合、*parseString()* を代わりに使うことができます:

xml.dom.minidom.**parseString**(*string*, *parser=None*)

string を表わす Document を返します。このメソッドは、文字列に対する *io.StringIO* オブジェクトを作成し、それを *parse()* に渡します。

これらの関数は両方とも、文書の内容を表現する Document オブジェクトを返します。

parse() や *parseString()* といった関数が行うのは、XML パーザを、何らかの SAX パーザからくる解析イベント (parse event) を受け取って DOM ツリーに変換できるような "DOM ビルダ (DOM builder)" に結合することです。関数は誤解を招くような名前になっているかもしれませんが、インタフェースについて学んでいるときには理解しやすいでしょう。文書の解析はこれらの関数が戻るより前に完結します; 要するに、これらの関数自体はパーザ実装を提供しないということです。

"DOM 実装" オブジェクトのメソッドを呼び出して Document を生成することもできます。このオブジェクトは、*xml.dom* パッケージ、または *xml.dom.minidom* モジュールの getDOMImplementation() 関数を呼び出して取得できます。Document を取得したら、DOM を構成するために子ノードを追加していくことができます:

```
from xml.dom.minidom import getDOMImplementation

impl = getDOMImplementation()

newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement
text = newdoc.createTextNode('Some textual content.')
top_element.appendChild(text)
```

DOM 文書オブジェクトを手にしたら、XML 文書のプロパティやメソッドを使って、文書の一部にアクセスすることができます。これらのプロパティは DOM 仕様で定義されています。文書オブジェクトの主要なプ

ロパティは documentElement プロパティです。このプロパティは XML 文書の主要な要素、つまり他の全ての要素を保持する要素を与えます。以下にプログラム例を示します。

```
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

DOM ツリーを使い終えたとき、unlink() メソッドを呼び出して不要になったオブジェクトが早く片付けられるように働きかけることができます。unlink() は、DOM API に対する *xml.dom.minidom* 特有の拡張です。ノードに対して unlink() を呼び出した後は、ノードとその下位ノードは本質的には無意味なものとなります。このメソッドを呼び出さなくても、Python のガベージコレクタがいつかはツリーのオブジェクトを後片付けします。

参考:

Document Object Model (DOM) Level 1 Specification *xml.dom.minidom* でサポートされている W3C の DOM に関する勧告。

20.7.1 DOM オブジェクト

Python の DOM API 定義は *xml.dom* モジュールドキュメントの一部として与えられています。この節では、*xml.dom* の API と *xml.dom.minidom* との違いについて列挙します。

Node.**unlink**()

DOM との内部的な参照を破壊して、循環参照ガベージコレクションを持たないバージョンの Python でもガベージコレクションされるようにします。循環参照ガベージコレクションが利用できる場合でも、このメソッドを使えば大量のメモリをすぐに使えるようにできるため、不要になったらすぐに DOM オブジェクトに対してこのメソッドを呼ぶのが良い習慣です。このメソッドは Document オブジェクトに対して呼び出すだけでよいのですが、あるノードの子ノードを破棄するために子ノードに対して呼び出してもかまいません。

with ステートメントを使用することで、このメソッドを明示的に呼ばないようにできます。with ブロックから出る時に自動的に次のコードが *dom* を unlink します:

```
with xml.dom.minidom.parse(datasource) as dom:
    ... # Work with dom.
```

Node.**writexml**(*writer*, *indent=""*, *addindent=""*, *newl=""*)

XML を *writer* オブジェクトに書き込みます。*writer* は入力としてテキストは受け付けますが、バイト列は受け付けません。*writer* はファイルオブジェクトインタフェースの write() に該当するメソッドを持たなければなりません。*indent* 引数には現在のノードのインデントを指定します。*addindent* 引数には現在のノードの下にサブノードを追加する際のインデント増分を指定します。*newl* には、改行時に行末を終端する文字列を指定します。

Document ノードでは、追加のキーワード引数 *encoding* を使って XML ヘッダの encoding フィールドを指定することができます。

バージョン 3.8 で変更: meth:*writexml* メソッドはユーザーが指定した属性の順序を保持するようにな

りました。

Node.**toxml**(*encoding=None*)

DOM ノードによって表わされる XML を含んだ文字列またはバイト文字列を返します。

明示的に *encoding*[*1] 引数を渡すと、結果は指定されたエンコードのバイト文字列になります。*encoding* 引数なしだと、結果は unicode 文字列です。また、結果として生じる文字列の中の XML 宣言はエンコーディングを指定しません。XML のデフォルトエンコーディングは UTF-8 なので、この文字列を UTF-8 以外でエンコードすることはおそらく正しくありません。

バージョン 3.8 で変更: *toxml()* メソッドはユーザーが指定した属性の順序を保持するようになりました。

Node.**toprettyxml**(*indent="\t", newl="\n", encoding=None*)

文書の整形されたバージョンを返します。*indent* はインデントを行うための文字で、デフォルトはタブです; *newl* には行末で出力される文字列を指定し、デフォルトは `\n` です。

encoding 引数は *toxml()* の対応する引数と同様に振る舞います。

バージョン 3.8 で変更: *toprettyxml()* メソッドはユーザーが指定した属性の順序を保持するようになりました。

20.7.2 DOM の例

以下のプログラム例は、単純なプログラムのかなり現実的な例です。特にこの例に関しては、DOM の柔軟性をあまり活用してはいません。

```
import xml.dom.minidom

document = """\
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point>
</slide>

<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>
"""
```

(次のページに続く)

[*1] XML 出力に含まれるエンコード名は適切な規格に従っていなければなりません。例えば "UTF-8" は有効ですが、"UTF8" は XML 文書の宣言では有効ではありません。後者はエンコード名として Python に認められるとしてもです。https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl と https://www.iana.org/assignments/character-sets/character-sets.xhtml を参照してください。

(前のページからの続き)

```
dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = []
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc.append(node.data)
    return ''.join(rc)

def handleSlideshow(slideshow):
    print("<html>")
    handleSlideshowTitle(slideshow.getElementsByTagName("title")[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print("</html>")

def handleSlides(slides):
    for slide in slides:
        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print("<title>%s</title>" % getText(title.childNodes))

def handleSlideTitle(title):
    print("<h2>%s</h2>" % getText(title.childNodes))

def handlePoints(points):
    print("<ul>")
    for point in points:
        handlePoint(point)
    print("</ul>")

def handlePoint(point):
    print("<li>%s</li>" % getText(point.childNodes))

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print("<p>%s</p>" % getText(title.childNodes))

handleSlideshow(dom)
```

20.7.3 minidom と DOM 標準

xml.dom.minidom モジュールは、本質的には DOM 1.0 互換の DOM に、いくつかの DOM 2 機能 (主に名前空間機能) を追加したものです。

Python における DOM インタフェースは率直なものです。以下の対応付け規則が適用されます:

- インタフェースはインスタンスオブジェクトを介してアクセスされます。アプリケーション自身から、クラスをインスタンス化してはなりません; `Document` オブジェクト上で利用可能な生成関数 (creator function) を使わなければなりません。派生インタフェースでは基底インタフェースの全ての演算 (および属性) に加え、新たな演算をサポートします。
- 演算はメソッドとして使われます。DOM では `in` パラメタのみを使うので、引数は通常の順番 (左から右へ) で渡されます。オプション引数はありません。`void` 演算は `None` を返します。
- IDL 属性はインスタンス属性に対応付けられます。OMG IDL 言語における Python への対応付けとの互換性のために、属性 `foo` はアクセサメソッド `_get_foo()` および `_set_foo()` でもアクセスできます。`readonly` 属性は変更してはなりません; とはいえ、これは実行時には強制されません。
- `short int` 、`unsigned int` 、`unsigned long long` 、および `boolean` 型は、全て Python 整数オブジェクトに対応付けられます。
- `DOMString` 型は Python 文字列型に対応付けられます。*xml.dom.minidom* ではバイト列か文字列のどちらかに対応づけられますが、通常文字列を生成します。`DOMString` 型の値は、W3C の DOM 仕様で、IDL `null` 値になってもよいとされている場所では `None` になることもあります。
- `const` 宣言を行うと、(`xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE` のように) 対応するスコープ内の変数に対応付けを行います; これらは変更してはなりません。
- `DOMException` は現状では *xml.dom.minidom* でサポートされていません。その代わり、*xml.dom.minidom* は、*TypeError* や *AttributeError* といった標準の Python 例外を使います。
- `NodeList` オブジェクトは Python の組み込みのリスト型を使って実装されています。これらのオブジェクトは DOM 仕様で定義されたインタフェースを提供していますが、以前のバージョンの Python では、公式の API をサポートしていません。しかしながら、これらの API は W3C 勧告で定義されたインタフェースよりも "Python 的な" ものになっています。

以下のインタフェースは *xml.dom.minidom* では全く実装されていません:

- `DOMTimeStamp`
- `EntityReference`

これらの大部分は、ほとんどの DOM のユーザにとって一般的な用途として有用とはならないような XML 文書内の情報を反映しています。

脚注

20.8 xml.dom.pulldom --- 部分的な DOM ツリー構築のサポート

Source code: Lib/xml/dom/pulldom.py

xml.dom.pulldom モジュールは ” プルパーザ” を提供します。プルパーザは必要に応じて文書の DOM アクセス可能な断片を生成することができます。基本概念は、入力 XML のストリームから ” イベント” を取り出し (pull し) て処理することです。SAX とは、コールバックつきのイベント駆動処理モデルを採用しているという点で同様ですが、SAX とは対照的に、プルパーザの使用者には処理が完了するかエラー状態が発生するまで、明示的にストリームからイベントを取り出し、イベントに対しループを回す責任があります。

> **警告:** *xml.dom.pulldom* モジュールは悪意を持って作成されたデータに対して安全ではありません。信頼できないデータや認証されていないデータをパースする必要がある場合は *XML* **の脆弱性** を参照してください。

バージョン 3.7.1 で変更: SAX パーサーは、デフォルトでセキュリティーを向上させるために、一般的な外部エンティティーをデフォルトでは処理しなくなりました。外部エンティティの処理を有効にするには、次の場所にカスタムパーサーインスタンスを渡します:

```
from xml.dom.pulldom import parse
from xml.sax import make_parser
from xml.sax.handler import feature_external_ges

parser = make_parser()
parser.setFeature(feature_external_ges, True)
parse(filename, parser=parser)
```

以下はプログラム例です:

```
from xml.dom import pulldom

doc = pulldom.parse('sales_items.xml')
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'item':
        if int(node.getAttribute('price')) > 50:
            doc.expandNode(node)
            print(node.toxml())
```

event は定数で以下の内の一つです:

- START_ELEMENT
- END_ELEMENT
- COMMENT
- START_DOCUMENT

- END_DOCUMENT
- CHARACTERS
- PROCESSING_INSTRUCTION
- IGNORABLE_WHITESPACE

node は型 xml.dom.minidom.Document、xml.dom.minidom.Element または xml.dom.minidom.Text のオブジェクトです。

文書はイベントの **フラットな** 流れとして扱われるため、文書の ” 木” は暗黙のうちに全て読み込まれ、目的の要素は木の中の深さに依らずに見つけられます。つまり、文書ノードの再帰的な検索のような階層的な問題を考える必要はありません。しかしながら要素の前後関係が重要な場合は、前後関係の状態を維持する (すなわち文章中の任意の点の場所を記憶する) か、*DOMEventStream.expandNode()* メソッドを使用して DOM 関連の処理に切り替える必要があります。

class xml.dom.pulldom.**PullDom**(*documentFactory=None*)
xml.sax.handler.ContentHandler のサブクラスです。

class xml.dom.pulldom.**SAX2DOM**(*documentFactory=None*)
xml.sax.handler.ContentHandler のサブクラスです。

xml.dom.pulldom.**parse**(*stream_or_string*, *parser=None*, *bufsize=None*)
与えられた入力から *DOMEventStream* を返します。*stream_or_string* はファイル名かファイル様オブジェクトのいずれかです。*parser* は、与えれた場合、*XMLReader* オブジェクトでなければなりません。この関数はパーザの文書ハンドラを変えて名前空間のサポートを有効にします。パーザの他の設定 (例えばエンティティリゾルバ) は前もってしておかなければなりません。

XML データを文字列で持っている場合、*parseString()* を代わりに使うことができます:

xml.dom.pulldom.**parseString**(*string*, *parser=None*)
(ユニコード) *string* を表す *DOMEventStream* を返します。

xml.dom.pulldom.**default_bufsize**
parse() の *bufsize* パラメタのデフォルト値です。

この変数の値は *parse()* を呼び出す前に変更することができます。その場合、その新しい値が有効になります。

20.8.1 DOMEventStream オブジェクト

class xml.dom.pulldom.**DOMEventStream**(*stream*, *parser*, *bufsize*)
バージョン 3.8 で非推奨: **シーケンスプロトコル** のサポートは非推奨になりました。

getEvent()
event が START_DOCUMENT の場合は *event* と xml.dom.minidom.Document としての現在の *node* からなるタプルを、START_ELEMENT か END_ELEMENT の場合は xml.dom.minidom.Element を、

CHARACTERS の場合は xml.dom.minidom.Text を返します。*expandNode()* が呼ばれない限り、現在のノードは子ノードの情報を持ちません。

expandNode(*node*)

node の全子ノードを *node* に展開します。例:

```
from xml.dom import pulldom

xml = '<html><title>Foo</title> <p>Some text <div>and more</div></p> </html>'
doc = pulldom.parseString(xml)
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'p':
        # Following statement only prints '<p/>'
        print(node.toxml())
        doc.expandNode(node)
        # Following statement prints node with all its children '<p>Some text <div>
↪and more</div></p>'
        print(node.toxml())
```

reset()

20.9 xml.sax --- SAX2 パーサのサポート

ソースコード: Lib/xml/sax/__init__.py

---

xml.sax パッケージは Python 用の Simple API for XML (SAX) インターフェースを実装した数多くのモジュールを提供しています。またパッケージには SAX 例外と SAX API 利用者が頻繁に利用するであろう有用な関数群も含まれています。

警告: *xml.sax* モジュールは悪意を持って作成されたデータに対して安全ではありません。信頼できないデータや認証されていないデータをパースする必要がある場合は *XML* **の脆弱性** を参照してください。

バージョン 3.7.1 で変更: SAX パーサーは、セキュリティーを向上させるために、デフォルトで一般的な外部エンティティーを処理しなくなりました。以前は、パーサーは、DTD およびエンティティ用にファイルシステムからリモートファイルまたはロードされたローカルファイルをフェッチするためのネットワーク接続を作成していました。この機能は parser オブジェクトと (実) 引数 *feature_external_ges* の *setFeature()* メソッドで再度有効にすることができます。

その関数群は以下の通りです:

xml.sax.make_parser(*parser_list=[]*)

SAX *XMLReader* オブジェクトを生成し、返します。最初に見つかったパーサーが使用されます。*parser_list* を与える場合、それは create_parser() という名前の関数をもつモジュール名のイテラブルでなければなりません。*parser_list* に列挙されているモジュールは、パーサーのデフォルトリストにあるモジュールよりも先に使われます。

バージョン 3.8 で変更: *parser_list* 引数はリストだけでなく、イテラブルでもよくなりました。

xml.sax.parse(*filename_or_stream*, *handler*, *error_handler=handler.ErrorHandler()*)

SAX パーサを生成し、そのパーサをドキュメントの解析に使用します。*filename_or_stream* として与えられるドキュメントは、ファイル名でもファイルオブジェクトでもかまいません。*handler* 引数は SAX *ContentHandler* のインスタントである必要があります。*error_handler* が与えられる場合は、SAX *ErrorHandler* のインスタンスである必要があります。この引数を省略した場合、全ての例外に対して *SAXParseException* が発生します。戻り値はありません。すべての操作は渡される *handler* によって行われなければなりません。

xml.sax.parseString(*string*, *handler*, *error_handler=handler.ErrorHandler()*)

parse() と同様ですが、こちらは引数で受け取ったバッファ *string* をパースします。*string* は *str* インスタンスか *bytes-like object* でなければなりません。

バージョン 3.5 で変更: *str* インスタンスがサポートされました。

典型的な SAX アプリケーションでは 3 種類のオブジェクト (リーダ、ハンドラ、入力元) が用いられます。ここで言うリーダとはパーサを指しています。つまり、入力元からバイト列または文字列を読み込み、一連のイベントを発生させるコード片のことです。発生したイベントはハンドラ・オブジェクトに割り振られます。言い換えると、リーダがハンドラのメソッドを呼び出すわけです。つまり、SAX アプリケーションは、リーダ・オブジェクトを作成し、入力元のオブジェクトを作成するか開き、ハンドラ・オブジェクトを作成し、これら 3 つのオブジェクトを連携させる必要があります。準備の最終段階では、リーダが呼び出され、入力をパースします。パース中には、入力データからの構造イベントや構文イベントに基づいて、ハンドラ・オブジェクトのメソッドが呼び出されます。

これらのオブジェクトでは、インタフェースだけが関係します。通常、これらはアプリケーション自体によってはインスタンス化されません。Python は明示的なインタフェースの概念を持たないので、インタフェースはクラスとして導入されました。しかし、アプリケーションは、提供されたクラスを継承せずに実装してもかまいません。インタフェース *InputSource*, *Locator*, Attributes, AttributesNS, *XMLReader* はモジュール *xml.sax.xmlreader* で定義されています。ハンドラインタフェースは *xml.sax.handler* で定義されています。利便性のため、*InputSource* (よく直接インスタンス化されるクラス) とハンドラクラスは *xml.sax* からもアクセスできます。これらのインタフェースについて下記で説明します。

このほかに *xml.sax* は次の例外クラスも提供しています。

exception xml.sax.SAXException(*msg*, *exception=None*)

XML エラーと警告をカプセル化します。このクラスには XML パーサとアプリケーションで発生するエラーおよび警告の基本的な情報を持たせることができます。また機能追加や地域化のためにサブクラス化することも可能です。なお *ErrorHandler* で定義されているハンドラがこの例外のインスタンスを受け取ることに注意してください。実際に例外を発生させることは必須でなく、情報のコンテナとして利用されることもあるからです。

インスタンスを作成する際 *msg* はエラー内容を示す可読データにしてください。オプションの *exception* 引数は None にするか、パース用コードで捕捉されて情報として渡される例外にしてください。

このクラスは SAX 例外の基底クラスになります。

exception xml.sax.**SAXParseException**(*msg*, *exception*, *locator*)
パースエラー時に発生する *SAXException* のサブクラスです。パースエラーに関する情報として、このクラスのインスタンスが SAX *ErrorHandler* インターフェースのメソッドに渡されます。このクラスは *SAXException* 同様 SAX *Locator* インターフェースもサポートしています。

exception xml.sax.**SAXNotRecognizedException**(*msg*, *exception=None*)
SAX *XMLReader* が認識できない機能やプロパティに遭遇したとき発生させる *SAXException* のサブクラスです。SAX アプリケーションや拡張モジュールにおいて同様の目的にこのクラスを利用することもできます。

exception xml.sax.**SAXNotSupportedException**(*msg*, *exception=None*)
SAX *XMLReader* が要求された機能をサポートしていないとき発生させる *SAXException* のサブクラスです。SAX アプリケーションや拡張モジュールにおいて同様の目的にこのクラスを利用することもできます。

参考:

SAX: The Simple API for XML　SAX API 定義に関し中心となっているサイトです。Java による実装とオンライン・ドキュメントが提供されています。実装と SAX API の歴史に関する情報のリンクも掲載されています。

xml.sax.handler **モジュール**　アプリケーションが提供するオブジェクトのインターフェース定義。

xml.sax.saxutils **モジュール**　SAX アプリケーション向けの有用な関数群。

xml.sax.xmlreader **モジュール**　パーサが提供するオブジェクトのインターフェース定義。

20.9.1 SAXException オブジェクト

SAXException 例外クラスは以下のメソッドをサポートしています:

SAXException.**getMessage**()
エラー状態を示す可読メッセージを返します。

SAXException.**getException**()
カプセル化した例外オブジェクトまたは None を返します。

20.10 xml.sax.handler --- SAX ハンドラの基底クラス

ソースコード: Lib/xml/sax/handler.py

SAX API はコンテント・ハンドラ、DTD ハンドラ、エラー・ハンドラ、エンティティ・リゾルバという 4 種類のハンドラを定義しています。通常アプリケーション側で実装する必要があるインターフェースは、使用したいイベントを発生させるものだけです。インターフェースは 1 つのオブジェクトにまとめることも、複数のオブジェクトに分けることも可能です。ハンドラの実装は、*xml.sax.handler* で提供される基底クラスを継承して、すべてのメソッドがデフォルトで実装されるようにしてください。

class xml.sax.handler.**ContentHandler**

アプリケーションにとって最も重要なメインの SAX コールバック・インターフェースです。このインターフェースで発生するイベントの順序はドキュメント内の情報の順序を反映しています。

class xml.sax.handler.**DTDHandler**

DTD イベントのハンドラです。

パースされていないエンティティや属性など、基本的なパースに必要な DTD イベントの指定だけを行うインターフェースです。

class xml.sax.handler.**EntityResolver**

エンティティ解決用の基本インターフェースです。このインターフェースを実装したオブジェクトを作成しパーサに登録することで、パーサはすべての外部エンティティを解決するメソッドを呼び出すようになります。

class xml.sax.handler.**ErrorHandler**

エラーや警告メッセージをアプリケーションに通知するためにパーサが使用するインターフェースです。このオブジェクトのメソッドが、エラーをただちに例外に変換するか、あるいは別の方法で処理するかの制御をしています。

これらのクラスに加え、*xml.sax.handler* は機能やプロパティ名のシンボル定数を提供しています。

xml.sax.handler.**feature_namespaces**

値: "http://xml.org/sax/features/namespaces"

真: 名前空間を処理します。

偽: オプションで名前空間を処理しません (暗黙に名前空間接頭辞も無効にします; デフォルト)。

アクセス: (パース時) 読み込み専用; (パース時以外) 読み書き可

xml.sax.handler.**feature_namespace_prefixes**

値: "http://xml.org/sax/features/namespace-prefixes"

真: 名前空間宣言で用いられている元々の接頭辞付きの名前と属性を報告します。

偽: 名前空間宣言で用いられている属性を報告しません。オプションで元々の接頭辞付きの名前も報告しません (デフォルト)。

アクセス: (パース時) 読み込み専用; (パース時以外) 読み書き可

xml.sax.handler.**feature_string_interning**

値: "http://xml.org/sax/features/string-interning"

真: 全ての要素名、接頭辞、属性名、名前空間 URI、ローカル名を組込みの intern 関数を使ってシンボルに登録します。

偽: 名前を必ずしもシンボルに登録しませんが、されるかもしれません (デフォルト)。

アクセス: (パース時) 読み込み専用; (パース時以外) 読み書き可

xml.sax.handler.**feature_validation**

値: "http://xml.org/sax/features/validation"

真: 全ての妥当性検査エラーを報告します (外部一般エンティティと外部変数エンティティが暗示されます)。

偽: 妥当性検査エラーを報告しません。

アクセス: (パース時) 読み込み専用; (パース時以外) 読み書き可

xml.sax.handler.**feature_external_ges**

値: "http://xml.org/sax/features/external-general-entities"

真: 全ての外部一般 (テキスト) エンティティを取り込みます。

偽: 外部一般エンティティを取り込みません。

アクセス: (パース時) 読み込み専用; (パース時以外) 読み書き可

xml.sax.handler.**feature_external_pes**

値: "http://xml.org/sax/features/external-parameter-entities"

真: 外部 DTD サブセットを含む全ての外部変数エンティティを取り込みます。

偽: 外部 DTD サブセットであっても外部変数エンティティを取り込みません。

アクセス: (パース時) 読み込み専用; (パース時以外) 読み書き可

xml.sax.handler.**all_features**

全機能のリスト。

xml.sax.handler.**property_lexical_handler**

値: "http://xml.org/sax/properties/lexical-handler"

データ型: xml.sax.sax2lib.LexicalHandler (Python 2 では未サポート)

説明: コメントなど字句解析イベント用のオプション拡張ハンドラ。

アクセス: 読み書き可

xml.sax.handler.**property_declaration_handler**

値: "http://xml.org/sax/properties/declaration-handler"

データ型: xml.sax.sax2lib.DeclHandler (Python 2 では未サポート)

説明: 表記や未解析エンティティをのぞく DTD 関連イベント用のオプション拡張ハンドラ。

アクセス: 読み書き可

xml.sax.handler.**property_dom_node**

値: "http://xml.org/sax/properties/dom-node"

データ型: org.w3c.dom.Node (Python 2 では未サポート)

説明: パース時は DOM イテレータならば現在の DOM ノードです。非パース時はイテレートのルート DOM ノードです。

アクセス: (パース時) 読み込み専用; (パース時以外) 読み書き可

xml.sax.handler.property_xml_string

値: "http://xml.org/sax/properties/xml-string"

データ型: 文字列

説明: 現在のイベントの元になったリテラル文字列。

アクセス: 読み出し専用

xml.sax.handler.all_properties
既知の全プロパティ名のリスト。

20.10.1 ContentHandler オブジェクト

ContentHandler はアプリケーション側でサブクラス化して利用することが前提になっています。パーサは入力ドキュメントのイベントにより、それぞれに対応する以下のメソッドを呼び出します:

ContentHandler.setDocumentLocator(*locator*)
アプリケーションにドキュメントイベントの発生位置を指すロケータを与えるためにパーサから呼び出されます。

SAX パーサによるロケータの提供は強く推奨されています (必須ではありません)。もし提供する場合は、DocumentHandler インターフェースのどのメソッドよりも先にこのメソッドが呼び出されるようにしなければなりません。

パーサがエラーを報告しない場合でも、ロケータによってアプリケーションは全てのドキュメント関連イベントの終了位置を知ることが出来ます。通常、アプリケーションは自身のエラー (例えば文字コンテンツがアプリケーションの規則に適合しない場合) を報告するためにこれを使用します。ロケータが返す情報は検索エンジンでの利用にはおそらく不充分です。

ロケータが正しい情報を返すのは、このインターフェースからイベントの呼出しが実行されている間だけです。それ以外のときは使用すべきでありません。

ContentHandler.startDocument()
ドキュメントの開始通知を受け取ります。

SAX パーサはこのインターフェースや DTDHandler のどのメソッド (*setDocumentLocator()* を除く) よりも先にこのメソッドを一度だけ呼び出します。

ContentHandler.endDocument()
ドキュメントの終了通知を受け取ります。

SAX パーサはこのメソッドを一度だけ呼び出します。パース中に呼び出す最後のメソッドです。パーサは (回復不能なエラーで) パース処理を中断するか、あるいは入力の最後に到達するまでこのメソッドを呼び出しません。

ContentHandler.**startPrefixMapping**(*prefix*, *uri*)

接頭辞と URI 名前空間の関連付けのスコープを開始します。

このイベントからの情報は名前空間処理に必須ではありません。SAX XML リーダは feature_namespaces 機能が有効な場合 (デフォルト)、要素と属性名の接頭辞を自動的に置換します。

しかしながら、接頭辞の自動展開を安全に行えないために、アプリケーションが文字データや属性値の中で接頭辞を使わなければならない場合があります。必要ならば *startPrefixMapping()* や *endPrefixMapping()* イベントはアプリケーションにコンテキスト自身の中で接頭辞を展開するための情報を提供します。

startPrefixMapping() と *endPrefixMapping()* イベントは相互に正しい入れ子関係になることが保証されていないので注意が必要です。すべての *startPrefixMapping()* は対応する *startElement()* の前に発生し、*endPrefixMapping()* イベントは対応する *endElement()* の後で発生しますが、その順序は保証されていません。

ContentHandler.**endPrefixMapping**(*prefix*)

接頭辞と URI の関連付けのスコープを終了します。

詳しくは *startPrefixMapping()* を参照してください。このイベントは常に対応する *endElement()* の後で発生しますが、複数の *endPrefixMapping()* イベントの順序は特に保証されません。

ContentHandler.**startElement**(*name*, *attrs*)

非名前空間モードでの要素の開始を通知します。

name 引数は要素型の生の XML 1.0 名を文字列として持ち、*attrs* 引数は要素の属性を持つ Attributes インターフェイス (*Attributes* **インタフェース** を参照) のオブジェクトを保持します。*attrs* ととして渡されたオブジェクトはパーサに再利用されるかもしれません。そのため、それへの参照を確保するのは属性のコピーを保持する確実な方法ではありません。属性のコピーを保持するには *attrs* 属性の *copy()* メソッドを使用してください。

ContentHandler.**endElement**(*name*)

非名前空間モードでの要素の終了を通知します。

name 引数は *startElement()* イベントとまったく同じ要素型名を持ちます。

ContentHandler.**startElementNS**(*name*, *qname*, *attrs*)

名前空間モードでの要素の開始を通知します。

name 引数は要素型の名前を (uri, localname) というタプルとして持ち、*qname* 引数は元の文書で使われている生の XML 1.0 名を持ち、要素の属性を持つ AttributesNS インターフェイス (*AttributesNS* **インタフェース** を参照) のインスタンスを保持します。名前空間が要素に関連付けられていない場合、*name* の *uri* 要素は None です。*attrs* ととして渡されたオブジェクトはパーサに再利用されるかもしれません。そのため、それへの参照を確保するのは属性のコピーを保持する確実な方法

ではありません。属性のコピーを保持するには *attrs* 属性の *copy()* メソッドを使用してください。

`feature_namespace_prefixes` 機能が有効でなければ、パーサで *qname* を `None` に設定することも可能です。

`ContentHandler.endElementNS`(*name*, *qname*)
名前空間モードでの要素の終了を通知します。

name 引数は *startElementNS()* イベントとまったく同じ要素型を持ちます。*qname* 引数も同様です。

`ContentHandler.characters`(*content*)
文字データの通知を受け取ります。

パーサはこのメソッドを呼び出して文字データの各チャンクを報告します。SAX パーサは一連の文字データを単一のチャンクとして返す場合と複数のチャンクに分けて返す場合がありますが、ロケータの情報が正しく保たれるように、一つのイベントの文字データは常に同じ外部エンティティのものでなければなりません。

content は文字列、バイト列のどちらでもかまいませんが、`expat` リーダ・モジュールは常に文字列を生成するようになっています。

注釈: Python XML SIG が提供していた初期 SAX 1 では、このメソッドにもっと JAVA 風のインターフェースが用いられています。しかし Python で採用されている大半のパーサでは古いインターフェースを有効に使うことができないため、よりシンプルなものに変更されました。古いコードを新しいインターフェースに変更するには、古い *offset* と *length* パラメータでスライスせずに、*content* を指定するようにしてください。

`ContentHandler.ignorableWhitespace`(*whitespace*)
要素内容中の無視できる空白文字の通知を受け取ります。

妥当性検査を行うパーサはこのメソッドを使って、無視できる空白文字 (W3C XML 1.0 勧告の 2.10 節参照) の各チャンクを報告しなければなりません。妥当性検査をしないパーサもコンテンツモデルの利用とパースが可能な場合、このメソッドを利用することが可能です。

SAX パーサは連続する複数の空白文字を単一のチャンクとして返す場合と複数のチャンクに分けて返す場合があります。しかし、ロケータが有用な情報を提供できるように、単一のイベント中のすべての文字は同じ外部エンティティからのものでなければなりません。

`ContentHandler.processingInstruction`(*target*, *data*)
処理命令の通知を受け取ります。

パーサは処理命令が見つかるたびにこのメソッドを呼び出します。処理命令はメインのドキュメント要素の前や後にも発生することがあるので注意してください。

SAX パーサがこのメソッドを使って XML 宣言 (XML 1.0 のセクション 2.8) やテキスト宣言 (XML 1.0 のセクション 4.3.1) の通知をすることはありません。

ContentHandler.**skippedEntity**(*name*)
スキップしたエンティティの通知を受け取ります。

パーサはエンティティをスキップするたびにこのメソッドを呼び出します。妥当性検査をしないプロセッサは (外部 DTD サブセットで宣言されているなどの理由で) 宣言が見当たらないエンティティをスキップします。すべてのプロセッサは feature_external_ges および feature_external_pes 属性の値によっては外部エンティティをスキップすることがあります。

20.10.2 DTDHandler オブジェクト

DTDHandler インスタンスは以下のメソッドを提供します:

DTDHandler.**notationDecl**(*name*, *publicId*, *systemId*)
表記宣言イベントの通知を扱います。

DTDHandler.**unparsedEntityDecl**(*name*, *publicId*, *systemId*, *ndata*)
未パースのエンティティ宣言イベントの通知を扱います。

20.10.3 EntityResolver オブジェクト

EntityResolver.**resolveEntity**(*publicId*, *systemId*)
エンティティのシステム識別子を解決し、文字列として読み込んだシステム識別子あるいは InputSource オブジェクトのいずれかを返します。デフォルトの実装では *systemId* を返します。

20.10.4 ErrorHandler オブジェクト

このインターフェイスのあるオブジェクトを使って *XMLReader* からエラーと警告情報を受け取ります。このインターフェイスを実装しているオブジェクトを作った場合、それを *XMLReader* に登録します。そうすると、パーサはすべての警告とエラーを報告するためにオブジェクトのメソッドを呼びます。利用可能なエラーには、次の 3 つのレベルがあります: 警告、(おそらく) 回復可能なエラー、回復不能なエラーです。全てのメソッドは唯一の引数として SAXParseException を受け取ります。警告とエラーは、渡された例外オブジェクトを送出することにより、例外に変換される場合があります。

ErrorHandler.**error**(*exception*)
パーサが回復可能なエラーに遭遇すると呼び出されます。このメソッドが例外を送出しない場合パースは継続されますが、アプリケーションは更なるドキュメント情報を期待すべきではありません。パーサの処理を継続を認めることで入力ドキュメント内の他のエラーを見つけることができます。

ErrorHandler.**fatalError**(*exception*)
パーサが回復不能なエラーに遭遇すると呼び出されます。このメソッドが return したとき、パースの停止が求められています。

ErrorHandler.**warning**(*exception*)
パーサが軽微な警告情報をアプリケーションに通知するときに呼び出されます。このメソッドが return

したときはパースの継続が求められ、ドキュメント情報はアプリケーションに送り続けられます。このメソッドでの例外の送出はパースを終了します。

20.11 xml.sax.saxutils --- SAX ユーティリティ

ソースコード: Lib/xml/sax/saxutils.py

モジュール *xml.sax.saxutils* には SAX アプリケーションの作成に役立つ多くの関数やクラスも含まれており、直接利用したり、基底クラスとして使うことができます。

xml.sax.saxutils.**escape**(*data*, *entities={}*)

文字列データ内の '&', '<', '>' をエスケープします。

オプションの *entities* 引数に辞書を渡すことで、そのほかの文字列データをエスケープすることも可能です。辞書のキーと値はすべて文字列で、キーは対応する値に置換されます。*entities* が与えられている場合でも、'&', '<', '>' は常にエスケープされます。

xml.sax.saxutils.**unescape**(*data*, *entities={}*)

エスケープされた文字列 '&', '<', '>' を元の文字に戻します。

オプションの *entities* 引数に辞書を渡すことで、そのほかの文字列データをエスケープ解除することも可能です。辞書のキーと値はすべて文字列で、キーは対応する値に置換されます。*entities* が与えられている場合でも、'&amp', '<', and '>' は常に元の文字に戻されます。

xml.sax.saxutils.**quoteattr**(*data*, *entities={}*)

escape() に似ていますが、*data* は属性値の作成に使われます。戻り値はクォート済みの *data* で、置換する文字の追加も可能です。*quoteattr()* はクォートすべき文字を *data* の文脈から判断し、クォートすべき文字を残さないように文字列をエンコードします。*data* の中にシングル・クォート、ダブル・クォートがあれば、両方ともエンコードし、全体をダブルクォートで囲みます。戻り値の文字列はそのままで属性値として利用できます:

```
>>> print("<element attr=%s>" % quoteattr("ab ' cd \" ef"))
<element attr="ab ' cd &quot; ef">
```

この関数は参照具象構文を使って、HTML や SGML の属性値を生成するのに便利です。

class xml.sax.saxutils.**XMLGenerator**(*out=None*, *encoding='iso-8859-1'*, *short_empty_elements=False*)

このクラスは SAX イベントを XML 文書に書き戻すことで *ContentHandler* インターフェースを実装しています。つまり、*XMLGenerator* をコンテントハンドラとして用いることで、パースしている元々の文書を複製することが出来ます。*out* にはファイル様オブジェクトでなければなりません。デフォルトは *sys.stdout* です。*encoding* は出力ストリームのエンコーディングで、デフォルトは 'iso-8859-1' です。*short_empty_elements* は内容を持たない要素のフォーマットを制御します。False (デフォルト) の場合、開始/終了タグのペアとなり、True の場合、1 つの空タグになります。

バージョン 3.2 で追加: *short_empty_elements* 引数。

class xml.sax.saxutils.XMLFilterBase(*base*)

このクラスは *XMLReader* とクライアント・アプリケーションのイベント・ハンドラとの間に位置するものとして設計されています。デフォルトでは何もせず、ただリクエストをリーダに、イベントをハンドラに、それぞれ加工せず渡すだけです。しかし、サブクラスでメソッドをオーバーライドすると、イベント・ストリームやリクエストを加工してから渡すように変更可能です。

xml.sax.saxutils.prepare_input_source(*source*, *base=''*)

この関数は引数に入力ソース、オプションとして URL を取り、読み取り可能な解決済み *InputSource* オブジェクトを返します。入力ソースは文字列、ファイル風オブジェクト、*InputSource* のいずれでも良く、この関数を使うことで、パーサは様々な *source* パラメータを parse() に渡すことが可能になります。

20.12 xml.sax.xmlreader --- XML パーサのインタフェース

ソースコード: Lib/xml/sax/xmlreader.py

各 SAX パーサは Python モジュールとして *XMLReader* インタフェースを実装しており、関数 create_parser() を提供しています。この関数は新たなパーサ・オブジェクトを生成する際、*xml.sax.make_parser()* から引数なしで呼び出されます。

class xml.sax.xmlreader.XMLReader

SAX パーサが継承可能な基底クラスです。

class xml.sax.xmlreader.IncrementalParser

入力ソースを一度にパースするのではなく、ドキュメントのチャンクが利用可能になるごとに取得したいことがあります。SAX リーダは通常、ファイル全体を一気に読み込まず、チャンク単位で処理するのですが、全体の処理が終わるまで parse() は返りません。そのため、parse() の排他的挙動を望まないときにこれらのインタフェースを使用してください。

パーサのインスタンスが作成されるとすぐに、feed メソッドを通じてデータを受け入れられるようになります。パースが完了して閉じるための呼び出しが行われた後、パーサがフィードからまたはパースメソッドを使用して新しいデータを受け入れられるように、reset メソッドが呼び出される必要があります。

これらのメソッドをパース処理の途中で呼び出すことはできません。つまり、パースが実行された後で、パーサから return する前に呼び出す必要があるのです。

デフォルトでは、SAX 2.0 ドライバを書く人のために、このクラスは IncrementalParser の feed、close、reset メソッドを使って XMLReader インタフェースの parse メソッドを実装しています。

class xml.sax.xmlreader.Locator

SAX イベントと文書の位置を関連付けるインタフェースです。locator オブジェクトは DocumentHandler メソッドを呼び出している間だけ正しい結果を返し、それ以外とのときは、予測できない結果を返します。情報を利用できない場合、メソッドは None を返すこともあります。

class xml.sax.xmlreader.InputSource(*system_id=None*)

XMLReader がエンティティを読み込むために必要な情報をカプセル化します。

このクラスには公開識別子、システム識別子、(場合によっては文字エンコーディング情報を含む) バイト・ストリーム、そしてエンティティの文字ストリームなどの情報が含まれます。

アプリケーションは *XMLReader.parse()* メソッドでの使用や EntityResolver.resolveEntity の戻り値としてこのオブジェトを作成します。

InputSource はアプリケーションに属します。*XMLReader* はアプリケーションから渡された *InputSource* オブジェクトの変更を許可されていませんが、コピーを作ってそれを変更することは可能です。

class xml.sax.xmlreader.AttributesImpl(*attrs*)

Attributes インタフェース (*Attributes* **インタフェース** 参照) の実装です。これは辞書風のオブジェクトで、startElement() 内で要素の属性を表示します。最も有用な辞書操作に加え、インタフェースに記述されているメソッドを多数サポートしています。このクラスのオブジェクトはリーダによってインスタンス化されなければなりません。*attrs* は属性名と属性値の対応付けを含む辞書風オブジェクトでなければなりません。

class xml.sax.xmlreader.AttributesNSImpl(*attrs*, *qnames*)

AttributesImpl を名前空間認識型に改良したクラスで、startElementNS() に渡されます。*AttributesImpl* の派生クラスですが、*namespaceURI* と *localname* の 2 要素のタプルを解釈します。さらに、元の文書に出てくる修飾名を返す多くのメソッドを提供します。このクラスは AttributesNS インタフェース (*AttributesNS* **インタフェース** 参照) の実装です。

20.12.1 XMLReader オブジェクト

XMLReader は次のメソッドをサポートします:

XMLReader.parse(*source*)

入力ソースを処理し、SAX イベントを作成します。*source* オブジェクトはシステム識別子 (入力ソースを特定する文字列 -- 一般にファイル名や URL)、*pathlib.Path* オブジェクトか *path-like* オブジェクトまたは *InputSource* オブジェクトです。*parse()* が return したとき、入力データの処理は完了し、パーサ・オブジェクトは破棄ないしリセットされます。

バージョン 3.5 で変更: 文字ストリームがサポートされました。

バージョン 3.8 で変更: path-like オブジェクトのサポートが追加されました。

XMLReader.getContentHandler()

現在の *ContentHandler* を返します。

XMLReader.setContentHandler(*handler*)

現在の *ContentHandler* を設定します。*ContentHandler* が設定されていない場合、内容イベントは破棄されます。

XMLReader.getDTDHandler()
現在の *DTDHandler* を返します。

XMLReader.setDTDHandler(*handler*)
現在の *DTDHandler* を返します。*DTDHandler* が設定されていな場合、DTD イベントは破棄されます。

XMLReader.getEntityResolver()
現在の *EntityResolver* を返します。

XMLReader.setEntityResolver(*handler*)
現在の *EntityResolver* を返します。*EntityResolver* が設定されていない場合、外部エンティティの解決を試行することでエンティティのシステム識別子が開かれます。利用できない場合は失敗します。

XMLReader.getErrorHandler()
現在の *ErrorHandler* を返します。

XMLReader.setErrorHandler(*handler*)
現在のエラーハンドラを設定します。*ErrorHandler* が設定されていない場合、エラーが例外として送出され、警告が表示されます。

XMLReader.setLocale(*locale*)
アプリケーションにエラーや警告のロケール設定を許可します。

SAX パーサにとって、エラーや警告の地域化は必須ではありません。しかし、パーサが要求されたロケールをサポートしていない場合、SAX 例外を送出しなければなりません。アプリケーションはパースの途中でロケールの変更を要求することができます。

XMLReader.getFeature(*featurename*)
機能 *featurename* の現在の設定を返します。その機能が認識できないときは、SAXNotRecognizedException を送出します。有名な機能名はモジュール *xml.sax.handler* に列挙されています。

XMLReader.setFeature(*featurename*, *value*)
機能名 *featurename* に値 *value* を設定します。その機能が認識できないときは、SAXNotRecognizedException を送出します。また、パーサが指定された機能や設定をサポートしていないときは、SAXNotSupportedException を送出します。

XMLReader.getProperty(*propertyname*)
属性名 *propertyname* の現在の値を返します。その属性が認識できないときは、SAXNotRecognizedException を送出します。有名な属性名はモジュール *xml.sax.handler* に列挙されています。

XMLReader.setProperty(*propertyname*, *value*)
属性名 *propertyname* に値 *value* を設定します。その機能が認識できないときは、SAXNotRecognizedException を送出します。また、パーサが指定された機能や設定をサポートしていないときは、SAXNotSupportedException を送出します。

20.12.2 IncrementalParser オブジェクト

IncrementalParser のインスタンスは次の追加メソッドを提供します:

IncrementalParser.**feed**(*data*)
: *data* のチャンクを処理します。

IncrementalParser.**close**()
: 文書の終端を決定します。文書の適格性を調べ (終端でのみ可能)、ハンドラを起動し、パース時に割り当てた資源を解放します。

IncrementalParser.**reset**()
: このメソッドは close が呼び出された後、新しい文書をパースできるように、パーサをリセットするのに呼び出されます。close 後 reset を呼び出さずに parse や feed を呼び出した場合の戻り値は未定義です。

20.12.3 Locator オブジェクト

Locator のインスタンスは次のメソッドを提供します:

Locator.**getColumnNumber**()
: 現在のイベントが開始する列番号を返します。

Locator.**getLineNumber**()
: 現在のイベントが開始する行番号を返します。

Locator.**getPublicId**()
: 現在の文書イベントの公開識別子を返します。

Locator.**getSystemId**()
: 現在のイベントのシステム識別子を返します。

20.12.4 InputSource オブジェクト

InputSource.**setPublicId**(*id*)
: この *InputSource* の公開識別子を設定します。

InputSource.**getPublicId**()
: この *InputSource* の公開識別子を返します。

InputSource.**setSystemId**(*id*)
: この *InputSource* のシステム識別子を設定します。

InputSource.**getSystemId**()
: この *InputSource* のシステム識別子を返します。

InputSource.**setEncoding**(*encoding*)
: この *InputSource* の文字エンコーディングを設定します。

エンコーディングは XML エンコーディング宣言として受け入れられる文字列でなければなりません (XML 勧告の 4.3.3 節を参照)。

InputSource も文字ストリームを含んでいた場合、*InputSource* のエンコーディング属性は無視されます。

InputSource.**getEncoding**()
この *InputSource* の文字エンコーディングを取得します。

InputSource.**setByteStream**(*bytefile*)
この入力ソースのバイトストリーム (*binary file*) を設定します。

文字ストリームも指定されている場合、SAX パーサはこのバイトストリームを無視しますが、URI 接続自体を開くときには優先してバイトストリームを使います。

アプリケーションがバイトストリームの文字エンコーディングを知っている場合は、setEncoding メソッドで設定する必要があります。

InputSource.**getByteStream**()
この入力ソースのバイトストリームを取得します。

getEncoding メソッドは、このバイトストリームの文字エンコーディングを返します。不明なときは None を返します。

InputSource.**setCharacterStream**(*charfile*)
この入力ソースの文字ストリーム (*text file*) を設定します。

文字ストリームが指定された場合、SAX パーサは全バイトストリームを無視し、システム識別子への URI 接続の開始を試みません。

InputSource.**getCharacterStream**()
この入力ソースの文字ストリームを取得します。

20.12.5 Attributes インタフェース

Attributes オブジェクトは copy()、get()、__contains__()、items()、keys()、values() を含む マッピングプロトコル の一部を実装しています。以下のメソッドも提供されています:

Attributes.**getLength**()
属性の数を返します。

Attributes.**getNames**()
属性の名前を返します。

Attributes.**getType**(*name*)
属性名 *name* のタイプを返します。通常は 'CDATA' です。

Attributes.**getValue**(*name*)
属性 *name* の値を返します。

20.12.6 AttributesNS インタフェース

このインタフェースは Attributes インタフェース (*Attributes インタフェース* 参照) のサブタイプです。Attributes インタフェースがサポートしているすべてのメソッドは AttributesNS オブジェクトでも利用可能です。

次のメソッドもサポートされています:

AttributesNS.getValueByQName(*name*)
修飾名の値を返します。

AttributesNS.getNameByQName(*name*)
修飾名 *name* に対応する (namespace, localname) のペアを返します。

AttributesNS.getQNameByName(*name*)
(namespace, localname) のペアに対応する修飾名を返します。

AttributesNS.getQNames()
すべての属性の修飾名を返します。

20.13 xml.parsers.expat --- Expat を使った高速な XML 解析

警告: pyexpat モジュールは悪意を持って作成されたデータに対して安全ではありません。信頼できないデータや認証されていないデータをパースする必要がある場合は *XML の脆弱性* を参照してください。

xml.parsers.expat モジュールは、検証 (validation) を行わない XML パーザ (parser, 解析器)、Expat への Python インタフェースです。モジュールは一つの拡張型 xmlparser を提供します。これは XML パーザの現在の状況を表します。一旦 xmlparser オブジェクトを生成すると、オブジェクトの様々な属性をハンドラ関数 (handler function) に設定できます。その後、XML 文書をパーザに入力すると、XML 文書の文字列とマークアップに応じてハンドラ関数が呼び出されます。

このモジュールでは、Expat パーザへのアクセスを提供するために pyexpat モジュールを使用します。pyexpat モジュールの直接使用は撤廃されています。

このモジュールは、例外を一つと型オブジェクトを一つ提供しています:

exception xml.parsers.expat.ExpatError
Expat がエラーを報告したときに例外を送出します。Expat のエラーを解釈する上での詳細な情報は、*ExpatError 例外* を参照してください。

exception xml.parsers.expat.error
ExpatError の別名です。

xml.parsers.expat.XMLParserType
ParserCreate() 関数から返された戻り値の型を示します。

xml.parsers.expat モジュールには以下の 2 つの関数が収められています:

xml.parsers.expat.**ErrorString**(*errno*)
　与えられたエラー番号 *errno* を解説する文字列を返します。

xml.parsers.expat.**ParserCreate**(*encoding=None*, *namespace_separator=None*)
　新しい xmlparser オブジェクトを作成し、返します。*encoding* が指定されていた場合、XML データで使われている文字列のエンコード名でなければなりません。Expat は、Python のように多くのエンコードをサポートしておらず、またエンコーディングのレパートリを拡張することはできません; サポートするエンコードは、UTF-8, UTF-16, ISO-8859-1 (Latin1), ASCII です。*encoding*[*1] が指定されると、文書に対する明示的、非明示的なエンコード指定を上書き (override) します。

　Expat はオプションで XML 名前空間の処理を行うことができます。これは引数 *namespace_separator* に値を指定することで有効になります。この値は、1 文字の文字列でなければなりません; 文字列が誤った長さを持つ場合には *ValueError* が送出されます (None は値の省略と見なされます)。名前空間の処理が可能なとき、名前空間に属する要素と属性が展開されます。要素のハンドラである StartElementHandler と EndElementHandler に渡された要素名は、名前空間の URI、名前空間の区切り文字、要素名のローカル部を連結したものになります。名前空間の区切り文字が 0 バイト (chr(0)) の場合、名前空間の URI とローカル部は区切り文字なしで連結されます。

　たとえば、*namespace_separator* に空白文字 (' ') がセットされ、次のような文書が解析されるとします:

```
<?xml version="1.0"?>
<root xmlns    = "http://default-namespace.org/"
      xmlns:py = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

　StartElementHandler は各要素ごとに次のような文字列を受け取ります:

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

　pyexpat が使っている Expat ライブラリの制限により、返される xmlparser インスタンスは単独の XML ドキュメントの解析にしか使えません。それぞれのドキュメントごとに別々のパーサのインスタンスを作るために ParserCreate を呼び出してください。

参考:

The Expat XML Parser　Expat プロジェクトのホームページ。

[*1] XML 出力に含まれるエンコーディング文字列は適切な規格に従っていなければなりません。例えば、"UTF-8" は有効ですが、"UTF8" はそうではありません。https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl と https://www.iana.org/assignments/character-sets/character-sets.xhtml を参照してください。

20.13.1 XMLParser オブジェクト

xmlparser オブジェクトは以下のようなメソッドを持ちます:

xmlparser.**Parse**(*data*[, *isfinal*])

文字列 *data* の内容を解析し、解析されたデータを処理するための適切な関数を呼び出します。このメソッドを最後に呼び出す時は *isfinal* を真にしなければなりません; 単体ファイルを細切れに渡して解析出来ることを意味しますが、複数ファイルは扱えません。*data* にはいつでも空の文字列を渡せます。

xmlparser.**ParseFile**(*file*)

file オブジェクトから読み込んだ XML データを解析します。*file* には *read(nbytes)* メソッドのみが必要です。このメソッドはデータがなくなった場合に空文字列を返さねばなりません。。

xmlparser.**SetBase**(*base*)

(XML) 宣言中のシステム識別子中の相対 URI を解決するための、基底 URI を設定します。相対識別子の解決はアプリケーションに任されます: この値は関数 *ExternalEntityRefHandler()* や *NotationDeclHandler()*, *UnparsedEntityDeclHandler()* に引数 *base* としてそのまま渡されます。

xmlparser.**GetBase**()

以前の *SetBase()* によって設定された基底 URI を文字列の形で返します。*SetBase()* が呼ばれていないときには None を返します。

xmlparser.**GetInputContext**()

現在のイベントを発生させた入力データを文字列として返します。データはテキストの入っているエンティティが持っているエンコードになります。イベントハンドラがアクティブでないときに呼ばれると、戻り値は None となります。

xmlparser.**ExternalEntityParserCreate**(*context*[, *encoding*])

親となるパーザで解析された内容が参照している、外部で解析されるエンティティを解析するために使える ”子の” パーザを作成します。*context* パラメータは、以下に記すように *ExternalEntityRefHandler()* ハンドラ関数に渡される文字列でなければなりません。子のパーザは *ordered_attributes*, *specified_attributes* が現在のパーザの値に設定されて生成されます。

xmlparser.**SetParamEntityParsing**(*flag*)

パラメータエンティティ (外部 DTD サブセットを含む) の解析を制御します。*flag* の有効な値は、XML_PARAM_ENTITY_PARSING_NEVER, XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE, XML_PARAM_ENTITY_PARSING_ALWAYS です。flag の設定をしたら true を返します。

xmlparser.**UseForeignDTD**([*flag*])

flag の値をデフォルトの true にすると、Expat は代わりの DTD をロードするため、すべての引数に *None* を設定して *ExternalEntityRefHandler* を呼び出します。XML 文書が文書型定義を持っていなければ、*ExternalEntityRefHandler* が呼び出しますが、*StartDoctypeDeclHandler* と *EndDoctypeDeclHandler* は呼び出されません。

flag に false を与えると、メソッドが前回呼ばれた時の true の設定が解除されますが、他には何も起こりません。

このメソッドは *Parse()* または *ParseFile()* メソッドが呼び出される前にだけ呼び出されます; これら 2 つのメソッドのどちらかが呼び出されたあとにメソッドが呼ばれると、*code* に定数 errors.codes[errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING] が設定されて例外 *ExpatError* が送出されます。

xmlparser.**SetReparseDeferralEnabled**(*enabled*)

> **警告:** Calling SetReparseDeferralEnabled(False) has security implications, as detailed below; please make sure to understand these consequences prior to using the SetReparseDeferralEnabled method.

Expat 2.6.0 introduced a security mechanism called "reparse deferral" where instead of causing denial of service through quadratic runtime from reparsing large tokens, reparsing of unfinished tokens is now delayed by default until a sufficient amount of input is reached. Due to this delay, registered handlers may — depending of the sizing of input chunks pushed to Expat — no longer be called right after pushing new input to the parser. Where immediate feedback and taking over responsiblity of protecting against denial of service from large tokens are both wanted, calling SetReparseDeferralEnabled(False) disables reparse deferral for the current Expat parser instance, temporarily or altogether. Calling SetReparseDeferralEnabled(True) allows re-enabling reparse deferral.

Note that *SetReparseDeferralEnabled()* has been backported to some prior releases of CPython as a security fix. Check for availability of *SetReparseDeferralEnabled()* using *hasattr()* if used in code running across a variety of Python versions.

バージョン 3.8.19 で追加.

xmlparser.**GetReparseDeferralEnabled**()

Returns whether reparse deferral is currently enabled for the given Expat parser instance.

バージョン 3.8.19 で追加.

xmlparser オブジェクトは次のような属性を持ちます:

xmlparser.**buffer_size**

buffer_text が真の時に使われるバッファのサイズです。この属性に新しい整数値を代入することで違うバッファサイズにできます。サイズが変えられるときにバッファはフラッシュされます。

xmlparser.**buffer_text**

この値を真にすると、xmlparser オブジェクトが Expat から返されたもとの内容をバッファに保持するようになります。これにより可能なときに何度も *CharacterDataHandler()* を呼び出してしまうようなことを避けることができます。Expat は通常、文字列のデータを行末ごと大量に破棄するため、かなりパフォーマンスを改善できるはずです。この属性はデフォルトでは偽で、いつでも変更可能です。

xmlparser.**buffer_used**

buffer_text が利用可能なとき、バッファに保持されたバイト数です。これらのバイトは UTF-8 でエ

ンコードされたテキストを表します。この属性は *buffer_text* が偽の時には意味がありません。

xmlparser.ordered_attributes
: この属性をゼロ以外の整数にすると、報告される (XML ノードの) 属性を辞書型ではなくリスト型にします。属性は文書のテキスト中の出現順で示されます。それぞれの属性は、2 つのリストのエントリ: 属性名とその値、が与えられます。(このモジュールの古いバージョンでも、同じフォーマットが使われています。) デフォルトでは、この属性はデフォルトでは偽となりますが、いつでも変更可能です。

xmlparser.specified_attributes
: ゼロ以外の整数にすると、パーザは文書のインスタンスで特定される属性だけを報告し、属性宣言から導出された属性は報告しないようになります。この属性が指定されたアプリケーションでは、XML プロセッサの振る舞いに関する標準に従うために必要とされる (文書型) 宣言によって、どのような付加情報が利用できるのかということについて特に注意を払わなければなりません。デフォルトで、この属性は偽となりますが、いつでも変更可能です。

以下の属性には、xmlparser オブジェクトで最も最近に起きたエラーに関する値が入っており、また Parse() または ParseFile() メソッドが *xml.parsers.expat.ExpatError* 例外を送出した際にのみ正しい値となります。

xmlparser.ErrorByteIndex
: エラーが発生したバイトのインデクスです。

xmlparser.ErrorCode
: エラーを特定する数値によるコードです。この値は *ErrorString()* に渡したり、errors オブジェクトで定義された内容と比較できます。

xmlparser.ErrorColumnNumber
: エラーの発生したカラム番号です。

xmlparser.ErrorLineNumber
: エラーの発生した行番号です。

以下の属性は xmlparser オブジェクトがその時パースしている位置に関する値を保持しています。コールバックがパースイベントを報告している間、これらの値はイベントの生成した文字列の先頭の位置を指し示します。コールバックの外から参照された時には、(対応するコールバックであるかにかかわらず) 直前のパースイベントの位置を示します。

xmlparser.CurrentByteIndex
: パーサへの入力の、現在のバイトインデックス。

xmlparser.CurrentColumnNumber
: パーサへの入力の、現在のカラム番号。

xmlparser.CurrentLineNumber
: パーサへの入力の、現在の行番号。

以下に指定可能なハンドラのリストを示します。xmlparser オブジェクト *o* にハンドラを指定するには、o.handlername = func を使用します。*handlername* は、以下のリストに挙げた値をとらなければならず、また *func* は正しい数の引数を受理する呼び出し可能なオブジェクトでなければなりません。引数は特に明記

しない限り、すべて文字列となります。

xmlparser.**XmlDeclHandler**(*version*, *encoding*, *standalone*)
XML 宣言が解析された時に呼ばれます。XML 宣言は XML 勧告の適用バージョン、文書テクストのエンコード、ならびに任意の "standalone" 宣言の (任意の) 宣言です。*version* と *encoding* は文字列で、*standalone* は文書がスタンドアローンと宣言された場合は 1、スタンドアローンでないと宣言された場合は 0、スタンドアローン節がない場合は -1 です。Expat のバージョン 1.95.0 以降でのみ使用できます。

xmlparser.**StartDoctypeDeclHandler**(*doctypeName*, *systemId*, *publicId*, *has_internal_subset*)
Expat が文書型宣言 (<!DOCTYPE ...) を解析し始めたときに呼び出されます。*doctypeName* は、与えられた値がそのまま Expat に提供されます。*systemId* と *publicId* パラメタが指定されている場合、それぞれシステムと公開識別子を与えます。省略する時には None にします。文書が内部的な文書宣言のサブセット (internal document declaration subset) を持つか、サブセット自体の場合、*has_internal_subset* は true になります。このハンドラには、Expat version 1.2 以上が必要です。

xmlparser.**EndDoctypeDeclHandler**()
Expat が文書型宣言の解析を終えたときに呼び出されます。このハンドラには、Expat version 1.2 以上が必要です。

xmlparser.**ElementDeclHandler**(*name*, *model*)
それぞれの要素型宣言ごとに呼び出されます。*name* は要素型の名前であり、*model* は内容モデル (content model) の表現です。

xmlparser.**AttlistDeclHandler**(*elname*, *attname*, *type*, *default*, *required*)
ひとつの要素型で宣言される属性ごとに呼び出されます。属性リストの宣言が 3 つの属性を宣言したとすると、このハンドラは各属性に 1 度ずつ、3 度呼び出されます。*elname* は要素名であり、これに対して宣言が適用され、*attname* が宣言された属性名となります。属性型は文字列で、*type* として渡されます。取り得る値は、'CDATA', 'ID', 'IDREF', ... です。*default* は、文書のインスタンスによって属性が指定されていないときに使用されるデフォルト値です。デフォルト値 (#IMPLIED values) が存在しないときには None を与えます。文書のインスタンス中に属性値を与える必要のあるときには *required* が true になります。これには Expat version 1.95.0 以上が必要です。

xmlparser.**StartElementHandler**(*name*, *attributes*)
要素の開始ごとに呼び出されます。*name* は要素名を持つ文字列で、*attributes* は要素の属性です。*ordered_attributes* が真の場合これはリストです (詳細は *ordered_attributes* を参照してください)。そうでなければ名前を値に対応させる辞書です。

xmlparser.**EndElementHandler**(*name*)
要素の終端を処理するごとに呼び出されます。

xmlparser.**ProcessingInstructionHandler**(*target*, *data*)
処理命令を処理するごとに呼び出されます。

xmlparser.**CharacterDataHandler**(*data*)
文字データを処理するときに呼びだされます。このハンドラは通常の文字データ、CDATA セクション、無視できる空白文字列のために呼び出されます。これらを識別しなければならないアプリケーションは、要求された情報を収集するために *StartCdataSectionHandler*, *EndCdataSectionHandler*,

and *ElementDeclHandler* コールバックメソッドを使用できます。

xmlparser.**UnparsedEntityDeclHandler**(*entityName*, *base*, *systemId*, *publicId*, *notationName*)
解析されていない (NDATA) エンティティ宣言を処理するために呼び出されます。このハンドラは Expat ライブラリのバージョン 1.2 のためだけに存在します; より最近のバージョンでは、代わりに *EntityDeclHandler* を使用してください (根底にある Expat ライブラリ内の関数は、撤廃されたものであると宣言されています)。

xmlparser.**EntityDeclHandler**(*entityName*, *is_parameter_entity*, *value*, *base*, *systemId*, *publicId*, *notationName*)
エンティティ宣言ごとに呼び出されます。パラメタと内部エンティティについて、*value* はエンティティ宣言の宣言済みの内容を与える文字列となります; 外部エンティティの時には None となります。解析済みエンティティの場合、*notationName* パラメタは None となり、解析されていないエンティティの時には記法 (notation) 名となります。*is_parameter_entity* は、エンティティがパラメタエンティティの場合真に、一般エンティティ (general entitiy) の場合には偽になります (ほとんどのアプリケーションでは、一般エンティティのことしか気にする必要がありません)。このハンドラは Expat ライブラリのバージョン 1.95.0 以降でのみ使用できます。

xmlparser.**NotationDeclHandler**(*notationName*, *base*, *systemId*, *publicId*)
記法の宣言 (notation declaration) で呼び出されます。*notationName*, *base*, *systemId*, および *publicId* を与える場合、文字列にします。public な識別子が省略された場合、*publicId* は None になります。

xmlparser.**StartNamespaceDeclHandler**(*prefix*, *uri*)
要素が名前空間宣言を含んでいる場合に呼び出されます。名前空間宣言は、宣言が配置されている要素に対して *StartElementHandler* が呼び出される前に処理されます。

xmlparser.**EndNamespaceDeclHandler**(*prefix*)
名前空間宣言を含んでいたエレメントの終了タグに到達したときに呼び出されます。このハンドラは、要素に関する名前空間宣言ごとに、*StartNamespaceDeclHandler* とは逆の順番で一度だけ呼び出され、各名前空間宣言のスコープが開始されたことを示します。このハンドラは、要素が終了する際、対応する *EndElementHandler* が呼ばれた後に呼び出されます。

xmlparser.**CommentHandler**(*data*)
コメントで呼び出されます。*data* はコメントのテキストで、先頭の '<!- -' と末尾の '- ->' を除きます。

xmlparser.**StartCdataSectionHandler**()
CDATA セクションの開始時に呼び出されます。CDATA セクションの構文的な開始と終了位置を識別できるようにするには、このハンドラと *EndCdataSectionHandler* が必要です。

xmlparser.**EndCdataSectionHandler**()
CDATA セクションの終了時に呼び出されます。

xmlparser.**DefaultHandler**(*data*)
XML 文書中で、適用可能なハンドラが指定されていない文字すべてに対して呼び出されます。この文字とは、検出されたことが報告されるが、ハンドラは指定されていないようなコンストラクト (construct) の一部である文字を意味します。

xmlparser.**DefaultHandlerExpand**(*data*)

DefaultHandler() と同じですが、内部エンティティの展開を禁止しません。エンティティ参照はデフォルトハンドラに渡されません。

xmlparser.NotStandaloneHandler()
: XML 文書がスタンドアロンの文書として宣言されていない場合に呼び出されます。外部サブセットやパラメタエンティティへの参照が存在するが、XML 宣言が XML 宣言中で standalone 変数を yes に設定していない場合に起きます。このハンドラが 0 を返すと、パーザは XML_ERROR_NOT_STANDALONE を発生させます。このハンドラが設定されていなければ、パーザは前述の事態で例外を送出しません。

xmlparser.ExternalEntityRefHandler(*context*, *base*, *systemId*, *publicId*)
: 外部エンティティの参照時に呼び出されます。*base* は現在の基底 (base) で、以前の *SetBase()* で設定された値になっています。public、および system の識別子である、*systemId* と *publicId* が指定されている場合、値は文字列です; public 識別子が指定されていない場合、*publicId* は None になります。*context* の値は不明瞭なものであり、以下に記述するようにしか使ってはなりません。

  外部エンティティが解析されるようにするには、このハンドラを実装しなければなりません。このハンドラは、ExternalEntityParserCreate(context) を使って適切なコールバックを指定し、子パーザを生成して、エンティティを解析する役割を担います。このハンドラは整数を返さなければなりません; 0 を返した場合、パーザは XML_ERROR_EXTERNAL_ENTITY_HANDLING エラーを送出します。そうでない場合、解析を継続します。

  このハンドラが与えられておらず、*DefaultHandler* コールバックが指定されていれば、外部エンティティは *DefaultHandler* で報告されます。

20.13.2 ExpatError 例外

ExpatError 例外はいくつかの興味深い属性を備えています:

ExpatError.code
: 特定のエラーに対する Expat の内部エラー番号です。*errors.messages* 辞書はこれらのエラー番号を Expat のエラーメッセージに対応させます。例えば:

```
from xml.parsers.expat import ParserCreate, ExpatError, errors

p = ParserCreate()
try:
    p.Parse(some_xml_document)
except ExpatError as err:
    print("Error:", errors.messages[err.code])
```

  errors モジュールはエラーメッセージ定数と、それらのメッセージをエラーコードに対応させる辞書 *codes* も提供しています。以下を参照してください。

ExpatError.lineno
: エラーが検出された場所の行番号です。最初の行の番号は 1 です。

ExpatError.offset
: エラーが発生した場所の行内でのオフセットです。最初のカラムの番号は 0 です。

20.13.3 使用例

以下のプログラムでは、与えられた引数を出力するだけの三つのハンドラを定義しています。

```
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print('Start element:', name, attrs)
def end_element(name):
    print('End element:', name)
def char_data(data):
    print('Character data:', repr(data))

p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""<?xml version="1.0"?>
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""", 1)
```

このプログラムの出力は以下のようになります:

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

20.13.4 内容モデルの記述

内容モデルは入れ子になったタプルを使って記述されています。各タプルには以下の 4 つの値が収められています: 型、限定詞 (quantifier)、名前、そして子のタプル。子のタプルは単に内容モデルを記述したものです。

最初の二つのフィールドの値は *xml.parsers.expat.model* モジュールで定義されている定数です。これらの定数は二つのグループ: モデル型 (model type) グループと限定子 (quantifier) グループ、に取りまとめられます。

以下にモデル型グループにおける定数を示します:

xml.parsers.expat.model.XML_CTYPE_ANY

モデル名で指定された要素は ANY の内容モデルを持つと宣言されます。

xml.parsers.expat.model.XML_CTYPE_CHOICE
: 指定されたエレメントはいくつかのオプションから選択できるようになっています; (A | B | C) のような内容モデルで用いられます。

xml.parsers.expat.model.XML_CTYPE_EMPTY
: EMPTY であると宣言されている要素はこのモデル型を持ちます。

xml.parsers.expat.model.XML_CTYPE_MIXED

xml.parsers.expat.model.XML_CTYPE_NAME

xml.parsers.expat.model.XML_CTYPE_SEQ
: 順々に続くようなモデルの系列を表すモデルがこのモデル型で表されます。(A, B, C) のようなモデルで用いられます。

限定子グループにおける定数を以下に示します:

xml.parsers.expat.model.XML_CQUANT_NONE
: 修飾子 (modifier) が指定されていません。従って A のように、厳密に一つだけです。

xml.parsers.expat.model.XML_CQUANT_OPT
: このモデルはオプションです: A? のように、一つか全くないかです。

xml.parsers.expat.model.XML_CQUANT_PLUS
: このモデルは (A+ のように) 一つかそれ以上あります。

xml.parsers.expat.model.XML_CQUANT_REP
: このモデルは A* のようにゼロ回以上あります。

20.13.5 Expat エラー定数

以下の定数は *xml.parsers.expat.errors* モジュールで提供されています。これらの定数は、エラーが発生した際に送出される ExpatError 例外オブジェクトのいくつかの属性を解釈する上で便利です。後方互換性の理由で、定数値は数字のエラー **コード** ではなくエラー **メッセージ** です。属性を解釈するには *code* 属性と errors.codes[errors.XML_ERROR_*CONSTANT_NAME*] を比較します。

errors モジュールには以下の属性があります:

xml.parsers.expat.errors.codes
: 文字列の記述をエラーコードに対応させる辞書です。

  バージョン 3.2 で追加.

xml.parsers.expat.errors.messages
: 数値的なエラーコードを文字列の記述に対応させる辞書です。

  バージョン 3.2 で追加.

xml.parsers.expat.errors.XML_ERROR_ASYNC_ENTITY

xml.parsers.expat.errors.**XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF**
属性値中のエンティティ参照が、内部エンティティではなく外部エンティティを参照しました。

xml.parsers.expat.errors.**XML_ERROR_BAD_CHAR_REF**
文字参照が、XML では正しくない (illegal) 文字を参照しました (例えば 0 や '&#0;')。

xml.parsers.expat.errors.**XML_ERROR_BINARY_ENTITY_REF**
エンティティ参照が、記法 (notation) つきで宣言されているエンティティを参照したため、解析できません。

xml.parsers.expat.errors.**XML_ERROR_DUPLICATE_ATTRIBUTE**
一つの属性が一つの開始タグ内に一度より多く使われています。

xml.parsers.expat.errors.**XML_ERROR_INCORRECT_ENCODING**

xml.parsers.expat.errors.**XML_ERROR_INVALID_TOKEN**
入力されたバイトが文字に適切に関連付けできない際に送出されます; 例えば、UTF-8 入力ストリームにおける NUL バイト (値 0) などです。

xml.parsers.expat.errors.**XML_ERROR_JUNK_AFTER_DOC_ELEMENT**
空白以外の何かがドキュメント要素の後にあります。

xml.parsers.expat.errors.**XML_ERROR_MISPLACED_XML_PI**
入力データの先頭以外の場所に XML 定義が見つかりました。

xml.parsers.expat.errors.**XML_ERROR_NO_ELEMENTS**
この文書には要素がありません (XML では全ての文書は確実に最上位の要素を正確に一つ持たなければなりません)。

xml.parsers.expat.errors.**XML_ERROR_NO_MEMORY**
Expat が内部メモリを確保できませんでした。

xml.parsers.expat.errors.**XML_ERROR_PARAM_ENTITY_REF**
パラメータエンティティが許可されていない場所で見つかりました。

xml.parsers.expat.errors.**XML_ERROR_PARTIAL_CHAR**
入力に不完全な文字が見つかりました。

xml.parsers.expat.errors.**XML_ERROR_RECURSIVE_ENTITY_REF**
エンティティ参照中に、同じエンティティへの別の参照が入っていました; おそらく違う名前で参照しているか、間接的に参照しています。

xml.parsers.expat.errors.**XML_ERROR_SYNTAX**
何らかの仕様化されていない構文エラーに遭遇しました。

xml.parsers.expat.errors.**XML_ERROR_TAG_MISMATCH**
終了タグが最も内側で開かれている開始タグに一致しません。

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_TOKEN**
何らかの (開始タグのような) トークンが閉じられないまま、ストリームの終端や次のトークンに遭遇しました。

xml.parsers.expat.errors.**XML_ERROR_UNDEFINED_ENTITY**
定義されていないエンティティへの参照が行われました。

xml.parsers.expat.errors.**XML_ERROR_UNKNOWN_ENCODING**
ドキュメントのエンコードが Expat でサポートされていません。

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_CDATA_SECTION**
CDATA セクションが閉じられていません。

xml.parsers.expat.errors.**XML_ERROR_EXTERNAL_ENTITY_HANDLING**

xml.parsers.expat.errors.**XML_ERROR_NOT_STANDALONE**
XML 文書が "standalone" だと宣言されており `NotStandaloneHandler` が設定され `0` が返されているにもかかわらず、パーサは "standalone" ではないと判別しました。

xml.parsers.expat.errors.**XML_ERROR_UNEXPECTED_STATE**

xml.parsers.expat.errors.**XML_ERROR_ENTITY_DECLARED_IN_PE**

xml.parsers.expat.errors.**XML_ERROR_FEATURE_REQUIRES_XML_DTD**
その操作を完了するには DTD のサポートが必要ですが、Expat が DTD のサポートをしない設定になっています。これは *xml.parsers.expat* モジュールの標準的なビルドでは報告されません。

xml.parsers.expat.errors.**XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING**
パースが始まったあとで動作の変更が要求されました。これはパースが開始される前にのみ変更可能です。(現在のところ) `UseForeignDTD()` によってのみ送出されます。

xml.parsers.expat.errors.**XML_ERROR_UNBOUND_PREFIX**
名前空間の処理を有効すると宣言されていないプレフィックスが見つかります。

xml.parsers.expat.errors.**XML_ERROR_UNDECLARING_PREFIX**
XML 文書はプレフィックスに対応した名前空間宣言を削除しようとしました。

xml.parsers.expat.errors.**XML_ERROR_INCOMPLETE_PE**
パラメータエンティティは不完全なマークアップを含んでいます。

xml.parsers.expat.errors.**XML_ERROR_XML_DECL**
XML 文書中に要素がありません。

xml.parsers.expat.errors.**XML_ERROR_TEXT_DECL**
外部エンティティ中のテキスト宣言にエラーがあります。

xml.parsers.expat.errors.**XML_ERROR_PUBLICID**
パブリック ID 中に許可されていない文字があります。

xml.parsers.expat.errors.**XML_ERROR_SUSPENDED**
要求された操作は一時停止されたパーサで行われていますが、許可されていない操作です。このエラーは追加の入力を行なおうとしている場合、もしくはパーサが停止しようとしている場合にも送出されます。

xml.parsers.expat.errors.**XML_ERROR_NOT_SUSPENDED**
: パーサを一時停止しようとしましたが、停止されませんでした。

xml.parsers.expat.errors.**XML_ERROR_ABORTED**
: Python アプリケーションには通知されません。

xml.parsers.expat.errors.**XML_ERROR_FINISHED**
: 要求された操作で、パース対象となる入力が完了したと判断しましたが、入力は受理されませんでした。このエラーは追加の入力を行なおうとしている場合、もしくはパーサが停止しようとしている場合に送出されます。

xml.parsers.expat.errors.**XML_ERROR_SUSPEND_PE**

脚注

第

TWENTYONE

インターネットプロトコルとサポート

この章で記述されるモジュールは、インターネットプロトコルを実装し、関連技術をサポートします。それらは全て Python で実装されています。これらのモジュールの大部分は、システム依存のモジュール *socket* が存在することが必要ですが、これは現在ではほとんどの一般的なプラットフォーム上でサポートされています。ここに概観を示します:

21.1 webbrowser --- 便利なウェブブラウザコントローラー

ソースコード: Lib/webbrowser.py

---

webbrowser モジュールにはウェブベースのドキュメントを表示するための、とてもハイレベルなインターフェースが定義されています。たいていの環境では、このモジュールの *open()* を呼び出すだけで正しく動作します。

Unix では、X11 上でグラフィカルなブラウザが選択されますが、グラフィカルなブラウザが利用できなかったり、X11 が利用できない場合はテキストモードのブラウザが使われます。もしテキストモードのブラウザが使われたら、ユーザがブラウザから抜け出すまでプロセスの呼び出しはブロックされます。

環境変数 BROWSER が存在する場合、これは *os.pathsep* で区切られたブラウザのリストとして解釈され、プラットフォームのデフォルトのブラウザリストに先立って順に試みられます。リストの中の値に %s が含まれていれば、%s を URL に置換したコマンドライン文字列と解釈されます；もし %s が含まれなければ、起動するブラウザの名前として単純に解釈されます。[*1]

非 Unix プラットフォームあるいは Unix 上でリモートブラウザが利用可能な場合、制御プロセスはユーザがブラウザを終了するのを待ちませんが、ディスプレイにブラウザのウィンドウを表示させたままにします。Unix 上でリモートブラウザが利用可能でない場合、制御プロセスは新しいブラウザを立ち上げ、待ちます。

webbrowser スクリプトをこのモジュールのコマンドライン・インタフェースとして使うことができます。スクリプトは引数に 1 つの URL を受け付けます。また次のオプション引数を受け付けます。-n により可能ならば新しいブラウザウィンドウで指定された URL を開きます。一方、-t では新しいブラウザのページ (「タブ」) で開きます。当然ながらこれらのオプションは排他的です。使用例は次の通りです:

[*1] ここでブラウザの名前が絶対パスで書かれていない場合は PATH 環境変数で与えられたディレクトリから探し出されます。

```
python -m webbrowser -t "http://www.python.org"
```

以下の例外が定義されています:

exception webbrowser.Error

ブラウザのコントロールエラーが起こると発生する例外。

以下の関数が定義されています:

webbrowser.open(*url*, *new=0*, *autoraise=True*)

デフォルトのブラウザで *url* を表示します。*new* が 0 なら、*url* はブラウザの今までと同じウィンドウで開きます。*new* が 1 なら、可能であればブラウザの新しいウィンドウが開きます。*new* が 2 なら、可能であればブラウザの新しいタブが開きます。*autoraise* が True なら、可能であればウィンドウが前面に表示されます（多くのウィンドウマネージャではこの変数の設定に関わらず、前面に表示されます）。

幾つかのプラットフォームにおいて、ファイル名をこの関数で開こうとすると、OS によって関連付けられたプログラムが起動されます。しかし、この動作はポータブルではありませんし、サポートされていません。

引数 url を指定して **監査イベント** webbrowser.open を送出します。

webbrowser.open_new(*url*)

可能であれば、デフォルトブラウザの新しいウィンドウで *url* を開きますが、そうでない場合はブラウザのただ 1 つのウィンドウで *url* を開きます。

webbrowser.open_new_tab(*url*)

可能であれば、デフォルトブラウザの新しいページ (「タブ」) で *url* を開きますが、そうでない場合は *open_new()* と同様に振る舞います。

webbrowser.get(*using=None*)

ブラウザの種類 *using* のコントローラーオブジェクトを返します。もし *using* が None なら、呼び出した環境に適したデフォルトブラウザのコントローラーを返します。

webbrowser.register(*name*, *constructor*, *instance=None*, ***, *preferred=False*)

ブラウザの種類 *name* を登録します。ブラウザの種類が登録されたら、*get()* でそのブラウザのコントローラーを呼び出すことができます。*instance* が指定されなかったり、None なら、インスタンスが必要な時には *constructor* がパラメータなしに呼び出されて作られます。*instance* が指定されたら、*constructor* は呼び出されないので、None でかまいません。

preferred を True に設定すると、*get()* の引数無しの呼び出しの結果が優先的にこのブラウザになります。そうでない場合は、この関数は、変数 BROWSER を設定するか、*get()* を空文字列ではない、宣言したハンドラの名前と一致する引数とともに呼び出すときにだけ、役に立ちます。

バージョン 3.7 で変更: *preferred* キーワード専用引数が追加されました。

いくつかの種類のブラウザがあらかじめ定義されています。このモジュールで定義されている、関数 *get()* に与えるブラウザの名前と、それぞれのコントローラークラスのインスタンスを以下の表に示します。

Type Name	Class Name	注釈
'mozilla'	Mozilla('mozilla')	
'firefox'	Mozilla('mozilla')	
'netscape'	Mozilla('netscape')	
'galeon'	Galeon('galeon')	
'epiphany'	Galeon('epiphany')	
'skipstone'	BackgroundBrowser('skipstone')	
'kfmclient'	Konqueror()	(1)
'konqueror'	Konqueror()	(1)
'kfm'	Konqueror()	(1)
'mosaic'	BackgroundBrowser('mosaic')	
'opera'	Opera()	
'grail'	Grail()	
'links'	GenericBrowser('links')	
'elinks'	Elinks('elinks')	
'lynx'	GenericBrowser('lynx')	
'w3m'	GenericBrowser('w3m')	
'windows-default'	WindowsDefault	(2)
'macosx'	MacOSX('default')	(3)
'safari'	MacOSX('safari')	(3)
'google-chrome'	Chrome('google-chrome')	
'chrome'	Chrome('chrome')	
'chromium'	Chromium('chromium')	
'chromium-browser'	Chromium('chromium-browser')	

注釈:

(1) "Konqueror" は Unix の KDE デスクトップ環境のファイルマネージャで、KDE が動作している時にだけ意味を持ちます。何か信頼できる方法で KDE を検出するのがいいでしょう; 変数 `KDEDIR` では十分ではありません。また、KDE 2 で **konqueror** コマンドを使うときにも、"kfm" が使われます --- Konqueror を動作させるのに最も良い方法が実装によって選択されます。

(2) Windows プラットフォームのみ。

(3) Mac OS X プラットフォームのみ。

バージョン 3.3 で追加: Chrome/Chromium のサポートが追加されました。

簡単な例を示します:

```
url = 'http://docs.python.org/'

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url)
```

(次のページに続く)

(前のページからの続き)

```
# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```

21.1.1 ブラウザコントローラーオブジェクト

ブラウザコントローラーには以下のメソッドが定義されていて、モジュールレベルの便利な 3 つの関数に相当します:

controller.open(*url*, *new=0*, *autoraise=True*)
このコントローラーでハンドルされたブラウザで *url* を表示します。*new* が 1 なら、可能であればブラウザの新しいウィンドウが開きます。*new* が 2 なら、可能であればブラウザの新しいページ (「タブ」) が開きます。

controller.open_new(*url*)
可能であれば、このコントローラーでハンドルされたブラウザの新しいウィンドウで *url* を開きますが、そうでない場合はブラウザのただ 1 つのウィンドウで *url* を開きます。*open_new()* の別名。

controller.open_new_tab(*url*)
可能であれば、このコントローラーでハンドルされたブラウザの新しいページ (「タブ」) で *url* を開きますが、そうでない場合は *open_new()* と同じです。

脚注

21.2 cgi --- CGI (ゲートウェイインタフェース規格) のサポート

ソースコード: Lib/cgi.py

ゲートウェイインタフェース規格 (CGI) に準拠したスクリプトをサポートするためのモジュールです。

このモジュールでは、Python で CGI スクリプトを書く際に使える様々なユーティリティを定義しています。

21.2.1 はじめに

CGI スクリプトは、HTTP サーバによって起動され、通常は HTML の `<FORM>` または `<ISINDEX>` エレメントを通じてユーザが入力した内容を処理します。

ほとんどの場合、CGI スクリプトはサーバ上の特殊なディレクトリ `cgi-bin` の下に置きます。HTTP サーバは、まずスクリプトを駆動するためのシェルの環境変数に、リクエストの全ての情報 (クライアントのホスト名、リクエストされている URL、クエリ文字列、その他諸々) を設定し、スクリプトを実行した後、スクリプトの出力をクライアントに送信します。

スクリプトの入力端もクライアントに接続されていて、この経路を通じてフォームデータを読み込むこともあります。それ以外の場合には、フォームデータは URL の一部分である「クエリ文字列」を介して渡されます。

このモジュールでは、上記のケースの違いに注意しつつ、Python スクリプトに対しては単純なインタフェースを提供しています。このモジュールではまた、スクリプトをデバッグするためのユーティリティも多数提供しています。また、最近はフォームを経由したファイルのアップロードをサポートしています (ブラウザ側がサポートしていればです)。

CGI スクリプトの出力は 2 つのセクションからなり、空行で分割されています。最初のセクションは複数のヘッダからなり、後続するデータがどのようなものかをクライアントに通知します。最小のヘッダセクションを生成するための Python のコードは以下のようなものです:

```
print("Content-Type: text/html")    # HTML is following
print()                             # blank line, end of headers
```

二つ目のセクションは通常、ヘッダやインラインイメージ等の付属したテキストをうまくフォーマットして表示できるようにした HTML です。以下に単純な HTML を出力する Python コードを示します:

```
print("<TITLE>CGI script output</TITLE>")
print("<H1>This is my first CGI script</H1>")
print("Hello, world!")
```

21.2.2 cgi モジュールを使う

`import cgi` と記述して開始します。

新たにスクリプトを書く際には、以下の行を付加するかどうか検討してください:

```
import cgitb
cgitb.enable()
```

これによって、特別な例外処理が有効にされ、エラーが発生した際にブラウザ上に詳細なレポートを出力するようになります。ユーザにスクリプトの内部を見せたくないのなら、以下のようにしてレポートをファイルに保存できます:

```
import cgitb
cgitb.enable(display=0, logdir="/path/to/logdir")
```

スクリプトを開発する際には、この機能はとても役に立ちます。*cgitb* が生成する報告はバグを追跡するためにかかる時間を大幅に減らせるような情報を提供してくれます。スクリプトをテストし終わり、正確に動作することを確認したら、いつでも `cgitb` の行を削除できます。

入力されたフォームデータを取得するには、`FieldStorage` クラスを使います。フォームが非 ASCII 文字を含んでいる場合は、*encoding* キーワードパラメータを使用してドキュメントに対して定義されたエンコーディングの値を設定してください。それは、通常 HTML ドキュメントの HEAD セクション中の META タグ、あるいは *Content-Type* ヘッダーに含まれています。これは、標準入力または環境変数からフォームの内容を読み出します (どちらから読み出すかは、複数の環境変数の値が CGI 標準に従ってどのように設定されているかで決まります)。インスタンスが標準入力を使うかもしれないので、インスタンス生成を行うのは一度だけにしなければなりません。

FieldStorage のインスタンスは Python の辞書型のように添え字アクセスが可能です。in を使用することによって要素が含まれているかの判定も出来ますし、標準の辞書メソッド *keys()* 及び組み込み関数 *len()* もサポートしています。空の文字列を含むフォーム要素は無視され、辞書には現れません。そのような値を保持するには、FieldStorage のインスタンス作成の際にオプションのキーワードパラメータ *keep_blank_values* に true を指定してください。

例えば、以下のコード (*Content-Type* ヘッダと空行はすでに出力された後とします) は name および addr フィールドが両方とも空の文字列に設定されていないか調べます:

```
form = cgi.FieldStorage()
if "name" not in form or "addr" not in form:
    print("<H1>Error</H1>")
    print("Please fill in the name and addr fields.")
    return
print("<p>name:", form["name"].value)
print("<p>addr:", form["addr"].value)
...further form processing here...
```

ここで、form[key] で参照される各フィールドはそれ自体が FieldStorage (または MiniFieldStorage 。フォームのエンコードによって変わります) のインスタンスです。インスタンスの属性 value の内容は対応するフィールドの値で、文字列になります。getvalue() メソッドはこの文字列値を直接返します。getvalue() の 2 つめの引数にオプションの値を与えると、リクエストされたキーが存在しない場合に返すデフォルトの値になります。

入力されたフォームデータに同じ名前のフィールドが二つ以上あれば、form[key] で得られるオブジェクトは FieldStorage や MiniFieldStorage のインスタンスではなく、そうしたインスタンスのリストになります。この場合、form.getvalue(key) も同様に、文字列からなるリストを返します。もしこうした状況が起きうると思うなら (HTML のフォームに同じ名前をもったフィールドが複数含まれているのなら) 、*getlist()* メソッドを使ってください。これは常に値のリストを返します (単一要素のケースを特別扱いする必要はありません)。例えば、以下のコードは任意の数のユーザ名フィールドを結合し、コンマで分割された文字列にします:

```
value = form.getlist("username")
usernames = ",".join(value)
```

フィールドがアップロードされたファイルを表している場合、value 属性や getvalue() メソッドを使ってフィールドの値にアクセスすると、ファイルの内容をすべてメモリ上にバイト列として読み込みます。これは場合によっては望ましい動作ではないかもしれません。アップロードされたファイルがあるかどうかは filename 属性および file 属性のいずれかで調べられます。そして、FieldStorage インスタンスのガベージコレクションの一部として自動的に閉じられるまでの間に、file 属性から以下のようにデータを読み込むことができます (*read()* および *readline()* メソッドはバイト列を返します):

```
fileitem = form["userfile"]
if fileitem.file:
    # It's an uploaded file; count lines
    linecount = 0
    while True:
        line = fileitem.file.readline()
```

(次のページに続く)

(前のページからの続き)

```
        if not line: break
        linecount = linecount + 1
```

FieldStorage オブジェクトは with 文での使用にも対応しています。with 文を使用した場合、オブジェクトは終了時に自動的に閉じられます。

アップロードされたファイルの内容を取得している間にエラーが発生した場合 (例えば、ユーザーが戻るボタンやキャンセルボタンで submit を中断した場合)、そのフィールドのオブジェクトの done 属性には -1 が設定されます。

現在ドラフトとなっているファイルアップロードの標準仕様では、一つのフィールドから (再帰的な *multipart/** エンコーディングを使って) 複数のファイルがアップロードされる可能性を受け入れています。この場合、アイテムは辞書形式の FieldStorage アイテムとなります。複数ファイルかどうかは type 属性が *multipart/form-data* (または *multipart/** にマッチする他の MIME 型) になっているかどうかを調べれば判別できます。この場合、トップレベルのフォームオブジェクトと同様にして再帰的に個別処理できます。

フォームが「古い」形式で入力された場合 (クエリ文字列または単一の *application/x-www-form-urlencoded* データで入力された場合)、データ要素の実体は MiniFieldStorage クラスのインスタンスになります。この場合、list 、file 、および filename 属性は常に None になります。

フォームが POST によって送信され、クエリー文字列も持っていた場合、FieldStorage と MiniFieldStorage の両方が含まれます。

バージョン 3.4 で変更: file 属性は、それを作成した FieldStorage インスタンスのガベージコレクションによって自動的に閉じられます。

バージョン 3.5 で変更: FieldStorage クラスにコンテキスト管理プロトコルのサポートが追加されました。

21.2.3 高水準インタフェース

前節では CGI フォームデータを FieldStorage クラスを使って読み出す方法について解説しました。この節では、フォームデータを分かりやすく直感的な方法で読み出せるようにするために追加された、より高水準のインタフェースについて記述します。このインタフェースは前節で説明した技術を撤廃するものではありません --- 例えば、前節の技術は依然としてファイルのアップロードを効率的に行う上で便利です。

このインタフェースは 2 つの単純なメソッドからなります。このメソッドを使えば、一般的な方法でフォームデータを処理でき、ある名前のフィールドに入力された値が一つなのかそれ以上なのかを心配する必要がなくなります。

前節では、一つのフィールド名に対して二つ以上の値が入力されるかもしれない場合には、常に以下のようなコードを書くよう学びました:

```
item = form.getvalue("item")
if isinstance(item, list):
    # The user is requesting more than one item.
else:
    # The user is requesting only one item.
```

こういった状況は、例えば以下のように、同じ名前を持った複数のチェックボックスからなるグループがフォームに入っているような場合によく起きます:

```
<input type="checkbox" name="item" value="1" />
<input type="checkbox" name="item" value="2" />
```

しかしながら、ほとんどの場合、あるフォーム中で特定の名前を持ったコントロールはただ一つしかないので、その名前に関連付けられた値はただ一つしかないはずだと考えるでしょう。そこで、スクリプトには例えば以下のようなコードを書くでしょう:

```
user = form.getvalue("user").upper()
```

このコードの問題点は、クライアントがスクリプトにとって常に有効な入力を提供するとは期待できないところにあります。例えば、もし好奇心旺盛なユーザがもう一つの `user=foo` ペアをクエリ文字列に追加したら、`getvalue("user")` メソッドは文字列ではなくリストを返すため、このスクリプトはクラッシュするでしょう。リストに対して *upper()* メソッドを呼び出すと、引数が有効でない (リスト型はその名前のメソッドを持っていない) ため、例外 *AttributeError* を送出します。

従って、フォームデータの値を読み出しには、得られた値が単一の値なのか値のリストなのかを常に調べるコードを使うのが適切でした。これでは煩わしく、より読みにくいスクリプトになってしまいます。

ここで述べる高水準のインタフェースで提供している *getfirst()* や *getlist()* メソッドを使うと、もっと便利にアプローチできます。

FieldStorage.getfirst(*name*, *default=None*)

フォームフィールド *name* に関連付けられた値をつねに一つだけ返す軽量メソッドです。同じ名前で 1 つ以上の値がポストされている場合、このメソッドは最初の値だけを返します。フォームから値を受信する際の値の並び順はブラウザ間で異なる可能性があり、特定の順番であるとは期待できないので注意してください。[*1] 指定したフォームフィールドや値がない場合、このメソッドはオプションの引数 *default* を返します。このパラメタを指定しない場合、標準の値は `None` に設定されます。

FieldStorage.getlist(*name*)

このメソッドはフォームフィールド *name* に関連付けられた値を常にリストにして返します。*name* に指定したフォームフィールドや値が存在しない場合、このメソッドは空のリストを返します。値が一つだけ存在する場合、要素を一つだけ含むリストを返します。

これらのメソッドを使うことで、以下のようにナイスでコンパクトにコードを書けます:

```
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", "").upper()    # This way it's safe.
for item in form.getlist("item"):
    do_something(item)
```

[*1] 最近のバージョンの HTML 仕様ではフィールドの値を供給する順番を取り決めてはいますが、ある HTTP リクエストがその取り決めに準拠したブラウザから受信したものかどうか、そもそもブラウザから送信されたものかどうかの判別は退屈で間違いやすいので注意してください。

21.2.4 関数

より細かく CGI をコントロールしたり、このモジュールで実装されているアルゴリズムを他の状況で利用したい場合には、以下の関数が便利です。

cgi.parse(*fp=None*, *environ=os.environ*, *keep_blank_values=False*, *strict_parsing=False*, *separator="&"*)

環境変数、またはファイルからからクエリを解釈します (ファイルは標準で `sys.stdin` になります) *keep_blank_values*, *strict_parsing*, *separator* 引数はそのまま *urllib.parse.parse_qs()* に渡されます。

バージョン 3.8.8 で変更: Added the *separator* parameter.

cgi.parse_multipart(*fp*, *pdict*, *encoding="utf-8"*, *errors="replace"*, *separator="&"*)

Parse input of type ***multipart/form-data*** (for file uploads). Arguments are *fp* for the input file, *pdict* for a dictionary containing other parameters in the ***Content-Type*** header, and *encoding*, the request encoding.

Returns a dictionary just like *urllib.parse.parse_qs()*: keys are the field names, each value is a list of values for that field. For non-file fields, the value is a list of strings.

この関数は簡単に使えますが、数メガバイトのデータがアップロードされると考えられる場合にはあまり適していません --- その場合、より柔軟性のある `FieldStorage` を代りに使ってください。

バージョン 3.7 で変更: Added the *encoding* and *errors* parameters. For non-file fields, the value is now a list of strings, not bytes.

バージョン 3.8.8 で変更: Added the *separator* parameter.

cgi.parse_header(*string*)

(***Content-Type*** のような) MIME ヘッダを解釈し、ヘッダの主要値と各パラメタからなる辞書にします。

cgi.test()

メインプログラムから利用できる堅牢性テストを行う CGI スクリプトです。最小の HTTP ヘッダと、HTML フォームからスクリプトに供給された全ての情報を書式化して出力します。

cgi.print_environ()

シェル変数を HTML に書式化して出力します。

cgi.print_form(*form*)

フォームを HTML に初期化して出力します。

cgi.print_directory()

現在のディレクトリを HTML に書式化して出力します。

cgi.print_environ_usage()

意味のある (CGI の使う) 環境変数を HTML で出力します。

21.2.5 セキュリティへの配慮

重要なルールが一つあります: (関数 *os.system()* または *os.popen()* 、またはその他の同様の機能によって) 外部プログラムを呼び出すなら、クライアントから受信した任意の文字列をシェルに渡していないことをよく確かめてください。これはよく知られているセキュリティホールであり、これによって Web のどこかにいる悪賢いハッカーが、だまされやすい CGI スクリプトに任意のシェルコマンドを実行させてしまえます。URL の一部やフィールド名でさえも信用してはいけません。CGI へのリクエストはあなたの作ったフォームから送信されるとは限らないからです！

安全な方法をとるために、フォームから入力された文字をシェルに渡す場合、文字列に入っているのが英数文字、ダッシュ、アンダースコア、およびピリオドだけかどうかを確認してください。

21.2.6 CGI スクリプトを Unix システムにインストールする

あなたの使っている HTTP サーバのドキュメントを読んでください。そしてローカルシステムの管理者と一緒にどのディレクトリに CGI スクリプトをインストールすべきかを調べてください; 通常これはサーバのファイルシステムツリー内の `cgi-bin` ディレクトリです。

あなたのスクリプトが "others" によって読み取り可能および実行可能であることを確認してください; Unix ファイルモードは 8 進表記で `0o755` です (`chmod 0755 filename` を使ってください)。スクリプトの最初の行の 1 カラム目が、`#!` で開始し、その後に Python インタプリタへのパス名が続いていることを確認してください。例えば:

```
#!/usr/local/bin/python
```

Python インタプリタが存在し、"others" によって実行可能であることを確かめてください。

あなたのスクリプトが読み書きしなければならないファイルが全て "others" によって読み出しや書き込み可能であることを確かめてください --- 読み出し可能のファイルモードは `0o644` で、書き込み可能のファイルモードは `0o666` になるはずです。これは、セキュリティ上の理由から、HTTP サーバがあなたのスクリプトを特権を全く持たないユーザ "nobody" の権限で実行するからです。この権限下では、誰でもが読める (書ける、実行できる) ファイルしか読み出し (書き込み、実行) できません。スクリプト実行時のディレクトリや環境変数のセットもあなたがログインしたときの設定と異なります。特に、実行ファイルに対するシェルの検索パス (`PATH`) や Python のモジュール検索パス (`PYTHONPATH`) が何らかの値に設定されていると期待してはいけません。

モジュールを Python の標準設定におけるモジュール検索パス上にないディレクトリからロードする必要がある場合、他のモジュールを取り込む前にスクリプト内で検索パスを変更できます。例えば:

```
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
sys.path.insert(0, "/usr/local/lib/python")
```

(この方法では、最後に挿入されたディレクトリが最初に検索されます！)

非 Unix システムにおける説明は変わるでしょう; あなたの使っている HTTP サーバのドキュメントを調べてください (普通は CGI スクリプトに関する節があります)。

21.2.7 CGI スクリプトをテストする

残念ながら、CGI スクリプトは普通、コマンドラインから起動しようとしても動きません。また、コマンドラインから起動した場合には完璧に動作するスクリプトが、不思議なことにサーバからの起動では失敗することがあります。しかし、スクリプトをコマンドラインから実行してみなければならない理由が一つあります: もしスクリプトが文法エラーを含んでいれば、Python インタプリタはそのプログラムを全く実行しないため、HTTP サーバはほとんどの場合クライアントに謎めいたエラーを送信するからです。

スクリプトが構文エラーを含まないのにうまく動作しないなら、次の節に読み進むしかありません。

21.2.8 CGI スクリプトをデバッグする

何よりもまず、些細なインストール関連のエラーでないか確認してください --- 上の CGI スクリプトのインストールに関する節を注意深く読めば時間を大いに節約できます。もしインストールの手続きを正しく理解しているか不安なら、このモジュールのファイル (`cgi.py`) をコピーして、CGI スクリプトとしてインストールしてみてください。このファイルはスクリプトとして呼び出すと、スクリプトの実行環境とフォームの内容を HTML 形式で出力します。ファイルに正しいモードを設定するなどして、リクエストを送ってみてください。標準的な `cgi-bin` ディレクトリにインストールされていれば、以下のような URL をブラウザに入力してリクエストを送信できるはずです:

```
http://yourhostname/cgi-bin/cgi.py?name=Joe+Blow&addr=At+Home
```

もしタイプ 404 のエラーになるなら、サーバはスクリプトを発見できないでいます -- おそらくあなたはスクリプトを別のディレクトリに入れる必要があるのでしょう。他のエラーになるなら、先に進む前に解決しなければならないインストール上の問題があります。もし実行環境の情報とフォーム内容 (この例では、各フィールドはフィールド名 "addr" に対して値 "At Home"、およびフィールド名 "name" に対して "Joe Blow") が綺麗にフォーマットされて表示されるなら、`cgi.py` スクリプトは正しくインストールされています。同じ操作をあなたの自作スクリプトに対して行えば、スクリプトをデバッグできるようになるはずです。

次のステップでは *cgi* モジュールの *test()* 関数を呼び出すことになります: メインプログラムコードを以下の 1 文と置き換えてください

```
cgi.test()
```

この操作で `cgi.py` ファイル自体をインストールした時と同じ結果を出力するはずです。

通常の Python スクリプトが例外を処理しきれずに送出した場合 (様々な理由: モジュール名のタイプミス、ファイルが開けなかった、など)、Python インタプリタはナイスなトレースバックを出力して終了します。Python インタプリタはあなたの CGI スクリプトが例外を送出した場合にも同様に振舞うので、トレースバックは大抵 HTTP サーバのいずれかのログファイルに残るかまったく無視されるかです。

幸運なことに、あなたが自作のスクリプトで **何らかの** コードを実行できるようになったら、*cgitb* モジュールを使って簡単にトレースバックをブラウザに送信できます。まだそうでないなら、以下の 2 行:

```
import cgitb
cgitb.enable()
```

をスクリプトの先頭に追加してください。そしてスクリプトを再度走らせます; 問題が発生すれば、クラッシュの原因を見出せるような詳細な報告を読めます。

cgitb モジュールのインポートに問題がありそうだと思うなら、(組み込みモジュールだけを使った) もっと堅牢なアプローチを取れます:

```
import sys
sys.stderr = sys.stdout
print("Content-Type: text/plain")
print()
...your code here...
```

このコードは Python インタプリタがトレースバックを出力することに依存しています。出力のコンテント型はプレーンテキストに設定されており、全ての HTML 処理を無効にしています。スクリプトがうまく動作する場合、生の HTML コードがクライアントに表示されます。スクリプトが例外を送出する場合、最初の 2 行が出力された後、トレースバックが表示されます。HTML の解釈は行われないので、トレースバックを読めるはずです。

21.2.9 よくある問題と解決法

- ほとんどの HTTP サーバはスクリプトの実行が完了するまで CGI からの出力をバッファします。このことは、スクリプトの実行中にクライアントが進捗状況報告を表示できないことを意味します。
- 上のインストールに関する説明を調べましょう。
- HTTP サーバのログファイルを調べましょう。(別のウィンドウで `tail -f logfile` を実行すると便利かもしれません！)
- 常に `python script.py` などとして、スクリプトが構文エラーでないか調べましょう。
- スクリプトに構文エラーがないなら、`import cgitb; cgitb.enable()` をスクリプトの先頭に追加してみましょう。
- 外部プログラムを起動するときには、スクリプトがそのプログラムを見つけられるようにしましょう。これは通常、絶対パス名を使うことを意味します --- `PATH` は普通、あまり CGI スクリプトにとって便利でない値に設定されています。
- 外部のファイルを読み書きする際には、CGI スクリプトを動作させるときに使われる userid でファイルを読み書きできるようになっているか確認しましょう: userid は通常、Web サーバを動作させている userid か、Web サーバの `suexec` 機能で明示的に指定している userid になります。
- CGI スクリプトを set-uid モードにしてはいけません。これはほとんどのシステムで動作せず、セキュリティ上の信頼性もありません。

脚注

21.3 cgitb --- CGI スクリプトのトレースバック管理機構

ソースコード: Lib/cgitb.py

---

cgitb モジュールでは、Python スクリプトのための特殊な例外処理を提供します。(実はこの説明は少し的外れです。このモジュールはもともと徹底的なトレースバック情報を CGI スクリプトで生成した HTML 内に表示するための設計されました。その後この情報を平文テキストでも表示できるように一般化されています。) このモジュールの有効化後に捕捉されない例外が生じた場合、詳細で書式化された報告が Web ブラウザに送信されます。この報告には各レベルにおけるソースコードの抜粋が示されたトレースバックと、現在動作している関数の引数やローカルな変数が収められており、問題のデバッグを助けます。オプションとして、この情報をブラウザに送信する代わりにファイルに保存することもできます。

この機能を有効化するためには、単に自作の CGI スクリプトの最初に以下の 2 行を追加します:

```
import cgitb
cgitb.enable()
```

enable() 関数のオプションは、報告をブラウザに表示するかどうかと、後で解析するためにファイルに報告をログ記録するかどうかを制御します。

cgitb.**enable**(*display=1*, *logdir=None*, *context=5*, *format="html"*)

この関数は、*sys.excepthook* を設定することで、インタプリタの標準の例外処理を *cgitb* モジュールに肩代わりさせるようにします。

オプションの引数 *display* は標準で `1` になっており、この値は `0` にしてトレースバックをブラウザに送らないように抑制することもできます。引数 *logdir* が存在すれば、トレースバックレポートはそのファイルに書き込まれます。*logdir* の値はログファイルを配置するディレクトリです。オプション引数 *context* は、トレースバックの中で現在の行の周辺の何行を表示するかです; この値は標準で `5` です。オプション引数 *format* が `"html"` の場合、出力は HTML に書式化されます。その他の値を指定すると平文テキストの出力を強制します。デフォルトの値は `"html"` です。

cgitb.**text**(*info*, *context=5*)

この関数は *info* (*sys.exc_info()* の結果を含む 3 タプル) に記述されている例外を取り扱い、テキストとしてトレースバックをフォーマットし、結果を文字列として返します。オプションの引数 *context* は、トレースバックにおいてソースコード行の前後のコンテキストを表示する行数です。デフォルトは `5` です。

cgitb.**html**(*info*, *context=5*)

この関数は *info* (*sys.exc_info()* の結果を含む 3 タプル) に記述されている例外を取り扱い、HTML としてトレースバックをフォーマットし、結果を文字列として返します。オプションの引数 *context* は、トレースバックにおいてソースコード行の前後のコンテキストを表示する行数です。デフォルトは `5` です。

cgitb.**handler**(*info=None*)

この関数は標準の設定 (ブラウザに報告を表示しますがファイルにはログを書き込みません) を使って

例外を処理します。この関数は、例外を捕捉した際に *cgitb* を使って報告したい場合に使うことができます。オプションの *info* 引数は、例外の型、例外の値、トレースバックオブジェクトからなる 3 要素のタプルでなければなりません。これは *sys.exc_info()* によって返される値と全く同じです。*info* 引数が与えられていない場合、現在の例外は *sys.exc_info()* から取得されます。

21.4 wsgiref --- WSGI ユーティリティとリファレンス実装

Web Server Gateway Interface (WSGI) は、Web サーバソフトウェアと Python で記述された Web アプリケーションとの標準インターフェースです。標準インターフェースを持つことで、WSGI をサポートするアプリケーションを幾つもの異なる Web サーバで使うことが容易になります。

Web サーバとプログラミングフレームワークの作者だけが、WSGI デザインのあらゆる細部や特例などを知る必要があります。WSGI アプリケーションをインストールしたり、既存のフレームワークを使ったアプリケーションを記述するだけの皆さんは、すべてについて理解する必要はありません。

wsgiref は WSGI 仕様のリファレンス実装で、これは Web サーバやフレームワークに WSGI サポートを加えるのに利用できます。これは WSGI 環境変数やレスポンスヘッダを操作するユーティリティ、WSGI サーバ実装時のベースクラス、WSGI アプリケーションを提供する デモ用 HTTP サーバ、それと WSGI サーバとアプリケーションの WSGI 仕様 (**PEP 3333**) 準拠のバリデーションツールを提供します。

wsgi.readthedocs.io に、WSGI に関するさらなる情報と、チュートリアルやその他のリソースへのリンクがあります。

21.4.1 wsgiref.util -- WSGI 環境のユーティリティ

このモジュールは WSGI 環境で使う様々なユーティリティ関数を提供します。WSGI 環境は **PEP 3333** で記述されているような HTTP リクエスト変数を含む辞書です。すべての *environ* パラメータを取る関数は WSGI 準拠の辞書を与えられることを期待しています; 細かい仕様については **PEP 3333** を参照してください。

wsgiref.util.guess_scheme(*environ*)

environ 辞書の `HTTPS` 環境変数を調べることで `wsgi.url_scheme` が "http" か "https" のどちらであるべきか推測し、その結果を返します。戻り値は文字列です。

この関数は、CGI や FastCGI のような CGI に似たプロトコルをラップするゲートウェイを作成する場合に便利です。典型的には、それらのプロトコルを提供するサーバが SSL 経由でリクエストを受け取った場合には `HTTPS` 変数に値 "1", "yes" または "on" を持つでしょう。そのため、この関数はそのような値が見つかった場合には "https" を返し、そうでなければ "http" を返します。

wsgiref.util.request_uri(*environ*, *include_query=True*)

リクエスト URI 全体 (オプションでクエリ文字列を含む) を、**PEP 3333** の "URL 再構築 (URL Reconstruction)" にあるアルゴリズムを使って返します。*include_query* が false の場合、クエリ文字列は結果となる文字列には含まれません。

wsgiref.util.**application_uri**(*environ*)

PATH_INFO と QUERY_STRING 変数が無視されることを除けば *request_uri()* に似ています。結果はリクエストによって指定されたアプリケーションオブジェクトのベース URI です。

wsgiref.util.**shift_path_info**(*environ*)

PATH_INFO から SCRIPT_NAME に一つの名前をシフトしてその名前を返します。*environ* 辞書は **変更されます** 。PATH_INFO や SCRIPT_NAME のオリジナルをそのまま残したい場合にはコピーを使ってください。

PATH_INFO にパスセグメントが何も残っていなければ、None が返されます。

典型的なこのルーチンの使い方はリクエスト URI のそれぞれの要素の処理で、例えばパスを一連の辞書のキーとして取り扱う場合です。このルーチンは、渡された環境を、ターゲット URL で示される別の WSGI アプリケーションの呼び出しに合うように調整します。例えば、/foo に WSGI アプリケーションがあったとして、そしてリクエスト URL パスが /foo/bar/baz で、/foo の WSGI アプリケーションが *shift_path_info()* を呼んだ場合、これは "bar" 文字列を受け取り、environ は /foo/bar の WSGI アプリケーションへの受け渡しに適するように更新されます。つまり、SCRIPT_NAME は /foo から /foo/bar に変わって、PATH_INFO は /bar/baz から /baz に変化するのです。

PATH_INFO が単に "/" の場合、このルーチンは空の文字列を返し、SCRIPT_NAME の末尾にスラッシュを加えます、これはたとえ空のパスセグメントが通常は無視され、そして SCRIPT_NAME は通常スラッシュで終わる事が無かったとしてもです。これは意図的な振る舞いで、このルーチンでオブジェクト巡回 (object traversal) をした場合に /x で終わる URI と /x/ で終わるものをアプリケーションが識別できることを保証するためのものです。

wsgiref.util.**setup_testing_defaults**(*environ*)

environ をテスト用に自明なデフォルト値で更新します。

このルーチンは WSGI に必要な様々なパラメータを追加します。そのようなパラメータとして HTTP_HOST 、SERVER_NAME 、SERVER_PORT 、REQUEST_METHOD 、SCRIPT_NAME 、PATH_INFO 、そして **PEP 3333** で定義されている wsgi.* 変数群が含まれます。このルーチンはデフォルト値を提供するだけで、これらの変数の既存設定は一切置きかえません。

このルーチンは、ダミー環境をセットアップすることによって WSGI サーバとアプリケーションのユニットテストを容易にすることを意図しています。これは実際の WSGI サーバやアプリケーションで使うべきではありません。なぜならこのデータは偽物なのです！

使用例:

```
from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# A relatively simple WSGI application. It's going to print out the
# environment dictionary after being updated by setup_testing_defaults
def simple_app(environ, start_response):
    setup_testing_defaults(environ)

    status = '200 OK'
    headers = [('Content-type', 'text/plain; charset=utf-8')]
```

(次のページに続く)

(前のページからの続き)

```
    start_response(status, headers)

    ret = [("%s: %s\n" % (key, value)).encode("utf-8")
           for key, value in environ.items()]
    return ret

with make_server('', 8000, simple_app) as httpd:
    print("Serving on port 8000...")
    httpd.serve_forever()
```

上記の環境用関数に加えて、*wsgiref.util* モジュールも以下のようなその他のユーティリティを提供します:

wsgiref.util.**is_hop_by_hop**(*header_name*)

'header_name' が **RFC 2616** で定義されている HTTP/1.1 の "Hop-by-Hop" ヘッダの場合に `True` を返します。

class wsgiref.util.**FileWrapper**(*filelike*, *blksize=8192*)

ファイル風オブジェクトを **イテレータ** に変換するラッパです。結果のオブジェクトは `__getitem__()` と `__iter__()` 両方をサポートしますが、これは Python 2.1 と Jython の互換性のためです。オブジェクトがイテレートされる間、オプションの *blksize* パラメータがくり返し *filelike* オブジェクトの `read()` メソッドに渡されて受け渡すバイト文字列を取得します。`read()` が空バイト文字列を返した場合、イテレーションは終了して再開されることはありません。

filelike に `close()` メソッドがある場合、返されたオブジェクトも `close()` メソッドを持ち、これが呼ばれた場合には *filelike* オブジェクトの `close()` メソッドを呼び出します。

使用例:

```
from io import StringIO
from wsgiref.util import FileWrapper

# We're using a StringIO-buffer for as the file-like object
filelike = StringIO("This is an example file-like object"*10)
wrapper = FileWrapper(filelike, blksize=5)

for chunk in wrapper:
    print(chunk)
```

バージョン 3.8 で非推奨: **シーケンスプロトコル** のサポートは非推奨になりました。

21.4.2 wsgiref.headers -- WSGI レスポンスヘッダツール群

このモジュールは単一のクラス、*Headers* を提供し、WSGI レスポンスヘッダの操作をマップに似たインターフェースで便利にします。

class wsgiref.headers.Headers([*headers*])

headers をラップするマップ風オブジェクトを生成します。これは **PEP 3333** に定義されるようなヘッダの名前／値のタプルのリストです。*headers* のデフォルト値は空のリストです。

Headers オブジェクトは典型的なマッピング操作をサポートし、これには `__getitem__()` 、`get()` 、`__setitem__()` 、`setdefault()` 、`__delitem__()` 、`__contains__()` を含みます。これらメソッドのそれぞれにおいて、キーはヘッダ名で（大文字小文字は区別しません）、値はそのヘッダ名に関連づけられた最初の値です。ヘッダを設定すると既存のヘッダ値は削除され、ラップされたヘッダのリストの末尾に新しい値が加えられます。既存のヘッダの順番は一般に維持され、ラップされたリストの最後に新しいヘッダが追加されます。

辞書とは違って、*Headers* オブジェクトはラップされたヘッダリストに存在しないキーを取得または削除しようとした場合にもエラーを発生しません。単に、存在しないヘッダの取得は `None` を返し、存在しないヘッダの削除は何もしません。

Headers オブジェクトは `keys()` 、`values()` 、`items()` メソッドもサポートします。複数の値を持つヘッダがある場合には、`keys()` と `items()` で返されるリストは同じキーを一つ以上含むことがあります。*Headers* オブジェクトの `len()` は、その `items()` の長さと同じであり、ラップされたヘッダリストの長さと同じです。実際、`items()` メソッドは単にラップされたヘッダリストのコピーを返しているだけです。

Headers オブジェクトに対して `bytes()` を呼ぶと、HTTP レスポンスヘッダとして送信するのに適した形に整形されたバイト文字列を返します。それぞれのヘッダはコロンとスペースで区切られた値と共に一列に並んでいます。それぞれの行はキャリッジリターンとラインフィードで終了し、バイト文字列は空行で終了しています。

これらのマッピングインターフェースと整形機能に加えて、*Headers* オブジェクトは複数の値を持つヘッダの取得と追加、MIME パラメータでヘッダを追加するための以下のようなメソッド群も持っています:

get_all(*name*)

指定されたヘッダのすべての値のリストを返します。

返されるリストは、元々のヘッダリストに現れる順、またはこのインスタンスに追加された順に並んでいて、重複を含む場合があります。削除されて加えられたフィールドはすべてヘッダリストの末尾に付きます。与えられた name に対するフィールドが何もなければ、空のリストが返ります。

add_header(*name*, *value*, ***_params*)

(複数の値を持つ可能性のある) ヘッダを、キーワード引数を通じて指定するオプションの MIME パラメータと共に追加します。

name は追加するヘッダフィールドです。このヘッダフィールドに MIME パラメータを設定するためにキーワード引数を使うことができます。それぞれのパラメータは文字列か `None` でなければ

いけません。パラメータ中のアンダースコアはダッシュ (-) に変換されます。これは、ダッシュが Python の識別子としては不正なのですが、多くの MIME パラメータはダッシュを含むためです。パラメータ値が文字列の場合、これはヘッダ値のパラメータに `name="value"` の形で追加されます。この値がもし `None` の場合、パラメータ名だけが追加されます。(これは値なしの MIME パラメータの場合に使われます。) 使い方の例は:

```
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

上記はこのようなヘッダを追加します:

```
Content-Disposition: attachment; filename="bud.gif"
```

バージョン 3.5 で変更: *headers* 引数が任意になりました。

21.4.3 wsgiref.simple_server -- シンプルな WSGI HTTP サーバ

このモジュールは WSGI アプリケーションを提供するシンプルな HTTP サーバです (*http.server* がベースです)。個々のサーバインスタンスは単一の WSGI アプリケーションを、特定のホストとポート上で提供します。もし一つのホストとポート上で複数のアプリケーションを提供したいならば、`PATH_INFO` をパースして個々のリクエストでどのアプリケーションを呼び出すか選択するような WSGI アプリケーションを作る必要があります。(例えば、*wsgiref.util* から `shift_path_info()` を利用します。)

wsgiref.simple_server.make_server(*host*, *port*, *app*, *server_class=WSGIServer*, *handler_class=WSGIRequestHandler*)

host と *port* 上で待機し、*app* へのコネクションを受け付ける WSGI サーバを作成します。戻り値は与えられた *server_class* のインスタンスで、指定された *handler_class* を使ってリクエストを処理します。*app* は **PEP 3333** で定義されるところの WSGI アプリケーションでなければいけません。

使用例:

```
from wsgiref.simple_server import make_server, demo_app

with make_server('', 8000, demo_app) as httpd:
    print("Serving HTTP on port 8000...")

    # Respond to requests until process is killed
    httpd.serve_forever()

    # Alternative: serve one request, then exit
    httpd.handle_request()
```

wsgiref.simple_server.demo_app(*environ*, *start_response*)

この関数は小規模ながら完全な WSGI アプリケーションで、"Hello world!" メッセージと、*environ* パラメータに提供されているキー／値のペアを含むテキストページを返します。これは WSGI サーバ (*wsgiref.simple_server* のような) がシンプルな WSGI アプリケーションを正しく実行できるかを確かめるのに便利です。

class wsgiref.simple_server.WSGIServer(*server_address*, *RequestHandlerClass*)

WSGIServer インスタンスを作成します。*server_address* は (host,port) のタプル、そして *RequestHandlerClass* はリクエストの処理に使われる *http.server.BaseHTTPRequestHandler* のサブクラスでなければいけません。

make_server() が細かい調整をやってくれるので、通常はこのコンストラクタを呼ぶ必要はありません。

WSGIServer は *http.server.HTTPServer* のサブクラスなので、そのすべてのメソッド (serve_forever() や handle_request() のような) が利用できます。*WSGIServer* も以下のような WSGI 固有メソッドを提供します:

set_app(*application*)
呼び出し可能 (callable) な *application* をリクエストを受け取る WSGI アプリケーションとして設定します。

get_app()
現在設定されている呼び出し可能 (callable) アプリケーションを返します。

しかしながら、通常はこれらの追加されたメソッドを使う必要はありません。*set_app()* は普通は *make_server()* によって呼ばれ、*get_app()* は主にリクエストハンドラインスタンスの便宜上存在するからです。

class wsgiref.simple_server.**WSGIRequestHandler**(*request*, *client_address*, *server*)
与えられた *request* (すなわちソケット) の HTTP ハンドラ、*client_address* ((host,port) のタプル)、*server* (*WSGIServer* インスタンス) の HTTP ハンドラを作成します。

このクラスのインスタンスを直接生成する必要はありません; これらは必要に応じて *WSGIServer* オブジェクトによって自動的に生成されます。しかしながら、このクラスをサブクラス化し、*make_server()* 関数に *handler_class* として与えることは可能でしょう。サブクラスにおいてオーバーライドする意味のありそうなものは:

get_environ()
リクエストに対する WSGI 環境を含む辞書を返します。デフォルト実装では *WSGIServer* オブジェクトの base_environ 辞書属性のコンテンツをコピーし、それから HTTP リクエスト由来の様々なヘッダを追加しています。このメソッド呼び出し毎に、**PEP 3333** に指定されている関連する CGI 環境変数をすべて含む新規の辞書を返さなければいけません。

get_stderr()
wsgi.errors ストリームとして使われるオブジェクトを返します。デフォルト実装では単に sys.stderr を返します。

handle()
HTTP リクエストを処理します。デフォルト実装では実際の WGI アプリケーションインターフェースを実装するのに *wsgiref.handlers* クラスを使ってハンドラインスタンスを作成します。

21.4.4 wsgiref.validate --- WSGI 準拠チェッカー

WSGI アプリケーションのオブジェクト、フレームワーク、サーバまたはミドルウェアの作成時には、その新規のコードを *wsgiref.validate* を使って準拠の検証をすると便利です。このモジュールは WSGI サーバやゲートウェイと WSGI アプリケーションオブジェクト間の通信を検証する WSGI アプリケーションオブジェクトを作成する関数を提供し、双方のプロトコル準拠をチェックします。

このユーティリティは完全な **PEP 3333** 準拠を保証するものでないことは注意してください; このモジュールでエラーが出ないことは必ずしもエラーが存在しないことを意味しません。しかしこのモジュールがエラーを出したならば、ほぼ確実にサーバかアプリケーションのどちらかが 100% 準拠ではありません。

このモジュールは lan Bicking の "Python Paste" ライブラリの `paste.lint` モジュールをベースにしています。

`wsgiref.validate.validator`(*application*)

application をラップし、新しい WSGI アプリケーションオブジェクトを返します。返されたアプリケーションは全てのリクエストを元々の *application* に転送し、*application* とそれを呼び出すサーバの両方が WSGI 仕様と **RFC 2616** の両方に準拠しているかをチェックします。

何らかの非準拠が検出されると、*AssertionError* 例外が送出されます; しかし、このエラーがどう扱われるかはサーバ依存であることに注意してください。例えば、*wsgiref.simple_server* とその他 *wsgiref.handlers* ベースのサーバ（エラー処理メソッドが他のことをするようにオーバライドしていないもの）は単純にエラーが発生したというメッセージとトレースバックのダンプを `sys.stderr` やその他のエラーストリームに出力します。

このラッパは、疑わしいものの実際には **PEP 3333** で禁止されていないかもしれない挙動を指摘するために *warnings* モジュールを使って出力を生成します。これらは Python のコマンドラインオプションや *warnings* API で抑制されなければ、`sys.stderr` (`wsgi.errors` では **ありません** 。ただし、たまたま同一のオブジェクトだった場合を除く) に書き出されます。

使用例:

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# Our callable object which is intentionally not compliant to the
# standard, so the validator is going to break
def simple_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)

    # This is going to break because we need to return a list, and
    # the validator is going to inform us
    return b"Hello World"

# This is the application wrapped in a validator
validator_app = validator(simple_app)
```

(次のページに続く)

(前のページからの続き)

```
with make_server('', 8000, validator_app) as httpd:
    print("Listening on port 8000....")
    httpd.serve_forever()
```

21.4.5 wsgiref.handlers -- サーバ／ゲートウェイのベースクラス

このモジュールは WSGI サーバとゲートウェイ実装のベースハンドラクラスを提供します。これらのベースクラスは、CGI 風の環境と、それに加えて入力、出力そしてエラーストリームが与えられることで、WSGI アプリケーションとの通信の大部分を処理します。

class wsgiref.handlers.**CGIHandler**

sys.stdin、sys.stdout、sys.stderr そして os.environ 経由での CGI ベースの呼び出しです。これは、もしあなたが WSGI アプリケーションを持っていて、これを CGI スクリプトとして実行したい場合に有用です。単に CGIHandler().run(app) を起動してください。app はあなたが起動したい WSGI アプリケーションオブジェクトです。

このクラスは *BaseCGIHandler* のサブクラスで、これは wsgi.run_once を true、wsgi.multithread を false、そして wsgi.multiprocess を true にセットし、常に *sys* と *os* を、必要な CGI ストリームと環境を取得するために使用します。

class wsgiref.handlers.**IISCGIHandler**

(IIS 7 以降の) 設定オプションの allowPathInfo や (IIS 7 より前の) メタベースの allowPathInfoForScriptMappings を設定せずに Microsoft の IIS Web サーバにデプロイするときに使う、*CGIHandler* クラス以外の専用の選択肢です。

By default, IIS gives a PATH_INFO that duplicates the SCRIPT_NAME at the front, causing problems for WSGI applications that wish to implement routing. This handler strips any such duplicated path.

IIS can be configured to pass the correct PATH_INFO, but this causes another bug where PATH_TRANSLATED is wrong. Luckily this variable is rarely used and is not guaranteed by WSGI. On IIS<7, though, the setting can only be made on a vhost level, affecting all other script mappings, many of which break when exposed to the PATH_TRANSLATED bug. For this reason IIS<7 is almost never deployed with the fix (Even IIS7 rarely uses it because there is still no UI for it.).

There is no way for CGI code to tell whether the option was set, so a separate handler class is provided. It is used in the same way as *CGIHandler*, i.e., by calling IISCGIHandler().run(app), where app is the WSGI application object you wish to invoke.

バージョン 3.2 で追加.

class wsgiref.handlers.**BaseCGIHandler**(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

CGIHandler に似ていますが、*sys* と *os* モジュールを使う代わりに CGI 環境と I/O ストリームを明示的に指定します。*multithread* と *multiprocess* の値は、ハンドラインスタンスにより実行されるアプリケーションの wsgi.multithread と wsgi.multiprocess フラグの設定に使われます。

このクラスは *SimpleHandler* のサブクラスで、HTTP の ” 本サーバ” でないソフトウェアと使うことを意図しています。もしあなたが `Status:` ヘッダを HTTP ステータスを送信するのに使うようなゲートウェイプロトコルの実装（CGI、FastCGI、SCGI など）を書いている場合、おそらく *SimpleHandler* ではなくこのクラスをサブクラス化するとよいでしょう。

class wsgiref.handlers.SimpleHandler(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

BaseCGIHandler と似ていますが、HTTP の本サーバと使うためにデザインされています。もしあなたが HTTP サーバ実装を書いている場合、おそらく *BaseCGIHandler* ではなくこのクラスをサブクラス化するとよいでしょう。

このクラスは *BaseHandler* のサブクラスです。これは `__init__()` 、`get_stdin()` 、`get_stderr()` 、`add_cgi_vars()` 、`_write()` 、`_flush()` をオーバーライドして、コンストラクタから明示的に環境とストリームを設定するようにしています。与えられた環境とストリームは `stdin` 、`stdout` 、`stderr` それに `environ` 属性に保存されています。

The *write()* method of *stdout* should write each chunk in full, like *io.BufferedIOBase*.

class wsgiref.handlers.BaseHandler

これは WSGI アプリケーションを実行するための抽象ベースクラスです。それぞれのインスタンスは一つの HTTP リクエストを処理します。しかし原理上は複数のリクエスト用に再利用可能なサブクラスを作成することができます。

BaseHandler インスタンスは外部から利用されるたった一つのメソッドを持ちます:

run(*app*)

指定された WSGI アプリケーション、*app* を実行します。

その他のすべての *BaseHandler* のメソッドはアプリケーション実行プロセスでこのメソッドから呼ばれます。したがって、それらは主にそのプロセスのカスタマイズのために存在しています。

以下のメソッドはサブクラスでオーバーライドされなければいけません:

_write(*data*)

バイト列の *data* をクライアントへの転送用にバッファします。このメソッドが実際にデータを転送しても OK です: 下部システムが実際にそのような区別をしている場合に効率をより良くするために、*BaseHandler* は書き出しとフラッシュ操作を分けているからです。

_flush()

バッファされたデータをクライアントに強制的に転送します。このメソッドは何もしなくても OK です（すなわち、*_write()* が実際にデータを送る場合）。

get_stdin()

現在処理中のリクエストの `wsgi.input` としての利用に適当な入力ストリームオブジェクトを返します。

get_stderr()

現在処理中のリクエストの `wsgi.errors` としての利用に適当な出力ストリームオブジェクトを返します。

add_cgi_vars()
現在のリクエストの CGI 変数を environ 属性に追加します。

オーバーライドされることの多いメソッド及び属性を以下に挙げます。しかし、このリストは単にサマリであり、オーバーライド可能なすべてのメソッドは含んでいません。カスタマイズした *BaseHandler* サブクラスを作成しようとする前に docstring やソースコードでさらなる情報を調べてください。

WSGI 環境のカスタマイズのための属性とメソッド:

wsgi_multithread
wsgi.multithread 環境変数で使われる値。*BaseHandler* ではデフォルトが true ですが、別のサブクラスではデフォルトで（またはコンストラクタによって設定されて）異なる値を持つことがあります。

wsgi_multiprocess
wsgi.multiprocess 環境変数で使われる値。*BaseHandler* ではデフォルトが true ですが、別のサブクラスではデフォルトで（またはコンストラクタによって設定されて）異なる値を持つことがあります。

wsgi_run_once
wsgi.run_once 環境変数で使われる値。*BaseHandler* ではデフォルトが false ですが、*CGIHandler* はデフォルトでこれを true に設定します。

os_environ
すべてのリクエストの WSGI 環境に含まれるデフォルトの環境変数。デフォルトでは *wsgiref.handlers* がインポートされた時点の os.environ のコピーですが、サブクラスはクラスまたはインスタンスレベルでそれら自身のものを作ることができます。デフォルト値は複数のクラスとインスタンスで共有されるため、この辞書は読み出し専用と考えるべきだという点に注意してください。

server_software
origin_server 属性が設定されている場合、この属性の値がデフォルトの SERVER_SOFTWARE WSGI 環境変数の設定や HTTP レスポンス中のデフォルトの Server: ヘッダの設定に使われます。これは (*BaseCGIHandler* や *CGIHandler* のような) HTTP オリジンサーバでないハンドラでは無視されます。

バージョン 3.3 で変更: ”Python” という語は ”CPython” や ”Jython” などのような個別実装の語に置き換えられました。

get_scheme()
現在のリクエストで使われている URL スキームを返します。デフォルト実装は *wsgiref.util* の guess_scheme() を使い、現在のリクエストの environ 変数に基づいてスキームが”http” か ”https” かを推測します。

setup_environ()
environ 属性を、フル実装 (fully-populated) の WSGI 環境に設定します。デフォルトの実装は、上記すべてのメソッドと属性、加えて *get_stdin()* 、*get_stderr()* 、*add_cgi_vars()* メソッドと *wsgi_file_wrapper* 属性を利用します。これは、キーが存在せず、*origin_server* 属性が

true 値で *server_software* 属性も設定されている場合に SERVER_SOFTWARE を挿入します。

例外処理のカスタマイズのためのメソッドと属性:

log_exception(*exc_info*)
exc_info タプルをサーバログに記録します。*exc_info* は (type, value, traceback) のタプルです。デフォルトの実装は単純にトレースバックをリクエストの wsgi.errors ストリームに書き出してフラッシュします。サブクラスはこのメソッドをオーバーライドしてフォーマットを変更したり出力先の変更、トレースバックを管理者にメールしたりその他適切と思われるいかなるアクションも取ることができます。

traceback_limit
デフォルトの *log_exception()* メソッドで出力されるトレースバック出力に含まれる最大のフレーム数です。None ならば、すべてのフレームが含まれます。

error_output(*environ*, *start_response*)
このメソッドは、ユーザに対してエラーページを出力する WSGI アプリケーションです。これはクライアントにヘッダが送出される前にエラーが発生した場合にのみ呼び出されます。

このメソッドは sys.exc_info() を使って現在のエラー情報にアクセスでき、その情報はこれを呼ぶときに *start_response* に渡すべきです (**PEP 3333** の "Error Handling" セクションに記述があります)。

デフォルト実装は単に *error_status* 、*error_headers* 、*error_body* 属性を出力ページの生成に使います。サブクラスではこれをオーバーライドしてもっと動的なエラー出力をすることができます。

しかし、セキュリティの観点からは診断をあらゆるユーザに吐き出すことは推奨されないことに気をつけてください; 理想的には、診断的な出力を有効にするには何らかの特別なことをする必要があるようにすべきで、これがデフォルト実装では何も含まれていない理由です。

error_status
エラーレスポンスで使われる HTTP ステータスです。これは **PEP 3333** で定義されているステータス文字列です; デフォルトは 500 コードとメッセージです。

error_headers
エラーレスポンスで使われる HTTP ヘッダです。これは **PEP 3333** で述べられているような、WSGI レスポンスヘッダ ((name, value) タプル) のリストであるべきです。デフォルトのリストはコンテントタイプを text/plain にセットしているだけです。

error_body
エラーレスポンスボディ。これは HTTP レスポンスのボディバイト文字列であるべきです。これはデフォルトではプレーンテキストで "A server error occurred. Please contact the administrator." です。

PEP 3333 の " オプションのプラットフォーム固有のファイルハンドリング" 機能のためのメソッドと属性:

wsgi_file_wrapper

wsgi.file_wrapper ファクトリ、または None です。この属性のデフォルト値は *wsgiref.util.FileWrapper* クラスです。

sendfile()
オーバーライドしてプラットフォーム固有のファイル転送を実装します。このメソッドはアプリケーションの戻り値が *wsgi_file_wrapper* 属性で指定されたクラスのインスタンスの場合にのみ呼ばれます。これはファイルの転送が成功できた場合には true を返して、デフォルトの転送コードが実行されないようにするべきです。このデフォルトの実装は単に false 値を返します。

その他のメソッドと属性:

origin_server
この属性はハンドラの *_write()* と *_flush()* が、特別に Status: ヘッダに HTTP ステータスを求めるような CGI 風のゲートウェイプロトコル経由でなく、クライアントと直接通信をするような場合には true 値に設定されているべきです。

この属性のデフォルト値は *BaseHandler* では true ですが、*BaseCGIHandler* と *CGIHandler* では false です。

http_version
origin_server が true の場合、この文字列属性はクライアントへのレスポンスセットの HTTP バージョンの設定に使われます。デフォルトは "1.0" です。

wsgiref.handlers.read_environ()
Transcode CGI variables from os.environ to **PEP 3333** "bytes in unicode" strings, returning a new dictionary. This function is used by *CGIHandler* and *IISCGIHandler* in place of directly using os.environ, which is not necessarily WSGI-compliant on all platforms and web servers using Python 3 -- specifically, ones where the OS's actual environment is Unicode (i.e. Windows), or ones where the environment is bytes, but the system encoding used by Python to decode it is anything other than ISO-8859-1 (e.g. Unix systems using UTF-8).

If you are implementing a CGI-based handler of your own, you probably want to use this routine instead of just copying values out of os.environ directly.

バージョン 3.2 で追加.

21.4.6 使用例

これは動作する "Hello World" WSGI アプリケーションです:

```
from wsgiref.simple_server import make_server

# Every WSGI application must have an application object - a callable
# object that accepts two arguments. For that purpose, we're going to
# use a function (note that you're not limited to a function, you can
# use a class for example). The first argument passed to the function
# is a dictionary containing CGI-style environment variables and the
# second variable is the callable object.
```

(次のページに続く)

(前のページからの続き)

```
def hello_world_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain; charset=utf-8')]  # HTTP Headers
    start_response(status, headers)

    # The returned object is going to be printed
    return [b"Hello World"]

with make_server('', 8000, hello_world_app) as httpd:
    print("Serving on port 8000...")

    # Serve until process is killed
    httpd.serve_forever()
```

Example of a WSGI application serving the current directory, accept optional directory and port number (default: 8000) on the command line:

```
#!/usr/bin/env python3
'''
Small wsgiref based web server. Takes a path to serve from and an
optional port number (defaults to 8000), then tries to serve files.
Mime types are guessed from the file names, 404 errors are raised
if the file is not found. Used for the make serve target in Doc.
'''
import sys
import os
import mimetypes
from wsgiref import simple_server, util

def app(environ, respond):

    fn = os.path.join(path, environ['PATH_INFO'][1:])
    if '.' not in fn.split(os.path.sep)[-1]:
        fn = os.path.join(fn, 'index.html')
    type = mimetypes.guess_type(fn)[0]

    if os.path.exists(fn):
        respond('200 OK', [('Content-Type', type)])
        return util.FileWrapper(open(fn, "rb"))
    else:
        respond('404 Not Found', [('Content-Type', 'text/plain')])
        return [b'not found']

if __name__ == '__main__':
    path = sys.argv[1] if len(sys.argv) > 1 else os.getcwd()
    port = int(sys.argv[2]) if len(sys.argv) > 2 else 8000
    httpd = simple_server.make_server('', port, app)
    print("Serving {} on port {}, control-C to stop".format(path, port))
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
```

(次のページに続く)

(前のページからの続き)

```
        print("Shutting down.")
        httpd.server_close()
```

21.5 urllib --- URL を扱うモジュール群

ソースコード: Lib/urllib/

---

urllib は URL を扱う幾つかのモジュールを集めたパッケージです:

- *urllib.request* は URL を開いて読むためのモジュールです
- *urllib.error* は *urllib.request* が発生させる例外を持っています
- *urllib.parse* は URL をパースするためのモジュールです
- *urllib.robotparser* は robots.txt ファイルをパースするためのモジュールです

21.6 urllib.request --- URL を開くための拡張可能なライブラリ

ソースコード: Lib/urllib/request.py

---

urllib.request モジュールは基本的な認証、暗号化認証、リダイレクション、Cookie、その他の介在する複雑なアクセス環境において (大抵は HTTP で) URL を開くための関数とクラスを定義します。

参考:

より高水準の HTTP クライアントインターフェースとして *Requests package <https://requests.readthedocs.io/en/master/>* がお奨めです。

urllib.request モジュールでは以下の関数を定義しています:

urllib.request.urlopen(*url*, *data=None*[, *timeout*], ***, *cafile=None*, *capath=None*, *cadefault=False*, *context=None*)

URL *url* を開きます。*url* は文字列でも *Request* オブジェクトでもかまいません。

data はサーバーに送信する追加データを指定するオブジェクトであるか None である必要があります。詳細は *Request* を確認してください。

urllib.request モジュールは HTTP/1.1 を使用し、その HTTP リクエストに Connection:close ヘッダーを含みます。

任意引数 *timeout* には接続開始などのブロックする操作におけるタイムアウト時間を秒数で指定します (指定されなかった場合、グローバルのデフォルトタイムアウト時間が利用されます)。この引数は、HTTP, HTTPS, FTP 接続でのみ有効です。

context を指定する場合は、様々な SSL オプションを記述する *ssl.SSLContext* インスタンスでなければなりません。詳細は *HTTPSConnection* を参照してください。

任意引数 *cafile* および *capath* には HTTPS リクエストのための CA 証明書のセットを指定します。*cafile* には CA 証明書のリストを含む 1 個のファイルを指定し、*capath* にはハッシュ化された証明書ファイルが格納されたディレクトリを指定しなければなりません。より詳しい情報は *ssl.SSLContext.load_verify_locations()* を参照してください。

cadefault 引数は無視されます。

この関数は常に **コンテキストマネージャ** として機能するオブジェクトを返します。このオブジェクトには以下のメソッドがあります。

- `geturl()` --- 取得されたリソースの URL を返します。主に、リダイレクトが発生したかどうかを確認するために利用します
- `info()` --- return the meta-information of the page, such as headers, in the form of an *email.message_from_string()* instance (see Quick Reference to HTTP Headers)
- `getcode()` -- レスポンスの HTTP ステータスコードです。

HTTP および HTTPS URL の場合、この関数は、わずかに修正された *http.client.HTTPResponse* オブジェクトを返します。上記の 3 つの新しいメソッドに加えて、msg 属性が *HTTPResponse* のドキュメンテーションで指定されているレスポンスヘッダーの代わりに *reason* 属性 --- サーバーから返された reason フレーズ --- と同じ情報を含んでいます。

FTP 、ファイルおよびデータ URL 、レガシーな *URLopener* や *FancyURLopener* によって明示的に扱われるリクエストの場合、この関数は `urllib.response.addinfourl` オブジェクトを返します。

プロトコルエラー発生時は *URLError* を送出します。

どのハンドラもリクエストを処理しなかった場合には `None` を返すことがあるので注意してください (デフォルトでインストールされる グローバルハンドラの *OpenerDirector* は、*UnknownHandler* を使って上記の問題が起きないようにしています)。

さらに、プロキシ設定が検出された場合 (例えば `http_proxy` のような `*_proxy` 環境変数がセットされているなど) には *ProxyHandler* がデフォルトでインストールされ、これがプロキシを通してリクエストを処理するようにしています。

Python 2.6 以前のレガシーな `urllib.urlopen` 関数は廃止されました。*urllib.request.urlopen()* が過去の `urllib2.urlopen` に相当します。`urllib.urlopen` において辞書型オブジェクトで渡していたプロキシの扱いは、*ProxyHandler* オブジェクトを使用して取得できます。

引数 `fullurl`, `data`, `headers`, `method` を指定して **監査イベント** `urllib.Request` を送出します。

バージョン 3.2 で変更: *cafile* および *capath* が追加されました。

バージョン 3.2 で変更: HTTPS バーチャルホストがサポートされました (*ssl.HAS_SNI* が真の場合のみ)。

バージョン 3.2 で追加: *data* にイテラブルなオブジェクトを指定できるようになりました。

バージョン 3.3 で変更: *cadefault* が追加されました。

バージョン 3.4.3 で変更: *context* が追加されました。

バージョン 3.6 で非推奨: *cafile*, *capath* および *cadefault* は非推奨となったので、*context* を使ってください。代わりに *ssl.SSLContext.load_cert_chain()* を使うか、または *ssl.create_default_context()* にシステムが信頼する CA 証明書を選んでもらうかしてください。

urllib.request.install_opener(*opener*)

指定された *OpenerDirector* のインスタンスを、デフォルトで利用されるグローバルの opener としてインストールします。opener のインストールは、urlopen にその opener を使って欲しいとき以外必要ありません。普段は単に *urlopen()* の代わりに *OpenerDirector.open()* を利用してください。この関数は引数が本当に *OpenerDirector* のインスタンスであるかどうかはチェックしません。適切なインタフェースを持った任意のクラスを利用することができます。

urllib.request.build_opener(*[handler, ...]*)

与えられた順番に URL ハンドラを連鎖させる *OpenerDirector* のインスタンスを返します。*handler* は *BaseHandler* または *BaseHandler* のサブクラスのインスタンスのどちらかです (どちらの場合も、コンストラクトは引数無しで呼び出せるようになっていなければなりません) 。クラス *ProxyHandler* (proxy 設定が検出された場合), *UnknownHandler*, *HTTPHandler*, *HTTPDefaultErrorHandler*, *HTTPRedirectHandler*, *FTPHandler*, *FileHandler*, *HTTPErrorProcessor* については、そのクラスのインスタンスか、そのサブクラスのインスタンスが *handler* に含まれていない限り、*handler* よりも先に連鎖します。

Python が SSL をサポートするように設定してインストールされている場合 (すなわち、*ssl* モジュールを import できる場合) *HTTPSHandler* も追加されます。

BaseHandler サブクラスでも `handler_order` メンバー変数を変更して、ハンドラーリスト内での場所を変更できます。

urllib.request.pathname2url(*path*)

ローカルシステムにおける記法で表されたパス名 *path* を URL におけるパス部分の形式に変換します。これは完全な URL を生成するわけではありません。戻り値は *quote()* 関数によってクオートされています。

urllib.request.url2pathname(*path*)

URL の、パーセントエンコードされたパス部分 *path* をローカルシステムの記法に変換します。これは完全な URL を受け付けません。*path* のデコードには *unquote()* 関数を使用します。

urllib.request.getproxies()

このヘルパー関数はスキーマからプロキシサーバーの URL へのマッピングを行う辞書を返します。この関数はまず、どの OS でも最初に `<scheme>_proxy` という名前の環境変数を大文字小文字を区別せずにスキャンします。そこで見つからなかった場合、Max OS X の場合は Mac OSX システム環境設定を、Windows の場合はシステムレジストリを参照します。もし小文字と大文字の環境変数が両方存在する (そして値が一致しない) なら、小文字の環境変数が優先されます。

注釈: もし環境変数 `REQUEST_METHOD` が設定されていたら (これは通常スクリプトが CGI 環境で動

いていることを示しています)、環境変数 HTTP_PROXY (大文字の _PROXY) は無視されます。その理由は、クライアントが "Proxy:" HTTP ヘッダーを使ってこの環境変数を注入できるからです。もし CGI 環境で HTTP プロキシを使う必要があれば、ProxyHandler を明示的に使用するか、環境変数名を小文字にしてください (あるいは、少なくともサフィックスを _proxy にしてください) 。

以下のクラスが提供されています:

class urllib.request.Request(*url*, *data=None*, *headers={}*, *origin_req_host=None*, *unverifiable=False*, *method=None*)

このクラスは URL リクエストを抽象化したものです。

url は有効な URL を指す文字列でなくてはなりません。

data must be an object specifying additional data to send to the server, or None if no such data is needed. Currently HTTP requests are the only ones that use *data*. The supported object types include bytes, file-like objects, and iterables of bytes-like objects. If no Content-Length nor Transfer-Encoding header field has been provided, *HTTPHandler* will set these headers according to the type of *data*. Content-Length will be used to send bytes objects, while Transfer-Encoding: chunked as specified in **RFC 7230**, Section 3.3.1 will be used to send files and other iterables.

HTTP POST リクエストメソッドでは *data* は標準の *application/x-www-form-urlencoded* 形式のバッファーでなければなりません。*urllib.parse.urlencode()* 関数は、マップ型あるいは 2 タプルのシーケンスを取り、この形式の ASCII 文字列を返します。これは *data* パラメーターとして使用される前に bytes 型にエンコードされなければなりません。

headers は辞書でなければなりません。この辞書は *add_header()* を辞書のキーおよび値を引数として呼び出した時と同じように扱われます。この引数は、多くの場合ブラウザーが何であるかを特定する User-Agent ヘッダーの値を " 偽装" するために用いられます。これは一部の HTTP サーバーが、スクリプトからのアクセスを禁止するために一般的なブラウザーの User-Agent ヘッダーしか許可しないためです。例えば、Mozilla Firefox は User-Agent に "Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11" のように設定し、*urllib* はデフォルトで "Python-urllib/2.6" (Python 2.6 の場合) と設定します。

An appropriate Content-Type header should be included if the *data* argument is present. If this header has not been provided and *data* is not None, Content-Type: application/x-www-form-urlencoded will be added as a default.

次の 2 つの引数は、サードパーティの HTTP クッキーを正しく扱いたい場合にのみ関係してきます:

origin_req_host は、**RFC 2965** で定義されている元のトランザクションにおけるリクエストホスト (request-host of the origin transaction) です。デフォルトの値は http.cookiejar.request_host(self) です。この値は、ユーザーによって開始された元々のリクエストにおけるホスト名や IP アドレスです。例えば、もしリクエストがある HTML ドキュメント内の画像を指していれば、この値は画像を含んでいるページへのリクエストにおけるリクエストホストになるはずです。

unverifiable は、**RFC 2965** の定義において、該当するリクエストが証明不能 (unverifiable) であるかどうかを示します。デフォルトの値は False です。証明不能なリクエストとは、ユーザが受け入れの

可否を選択できないような URL を持つリクエストのことです。例えば、リクエストが HTML ドキュメント中の画像であり、ユーザがこの画像を自動的に取得するか どうかを選択できない場合には、証明不能フラグは True になります。

method は使用される HTTP リクエストメソッド (例: `'HEAD'`) を示す文字列でなければなりません。もし与えられた場合、この値は属性 *method* に格納され、*get_method()* で使用されます。デフォルトは *data* が `None` であれば `'GET'` で、そうでなければ `'POST'` です。サブクラスは *method* 属性をクラス自身に設定して、異なるデフォルトメソッドを示すことができます。

注釈: The request will not work as expected if the data object is unable to deliver its content more than once (e.g. a file or an iterable that can produce the content only once) and the request is retried for HTTP redirects or authentication. The *data* is sent to the HTTP server right away after the headers. There is no support for a 100-continue expectation in the library.

バージョン 3.3 で変更: 引数 *Request.method* が Request クラスに追加されました。

バージョン 3.4 で変更: *Request.method* のデフォルト値はクラスレベルで指定されることがあります。

バージョン 3.6 で変更: Do not raise an error if the `Content-Length` has not been provided and *data* is neither `None` nor a bytes object. Fall back to use chunked transfer encoding instead.

class urllib.request.OpenerDirector

OpenerDirector クラスは、*BaseHandler* の連鎖的に呼び出して URL を開きます。このクラスはハンドラをどのように連鎖させるか、またどのようにエラーをリカバリするかを管理します。

class urllib.request.BaseHandler

このクラスはハンドラ連鎖に登録される全てのハンドラがベースとしているクラスです -- このクラスでは登録のための単純なメカニズムだけを扱います。

class urllib.request.HTTPDefaultErrorHandler

HTTP エラーレスポンスのデフォルトハンドラーを定義するクラスです; すべてのレスポンスは *HTTPError* 例外に変換されます。

class urllib.request.HTTPRedirectHandler

リダイレクションを扱うクラスです。

class urllib.request.HTTPCookieProcessor(*cookiejar=None*)

HTTP Cookie を扱うためのクラスです。

class urllib.request.ProxyHandler(*proxies=None*)

このクラスはプロキシを通過してリクエストを送らせます。引数 *proxies* を与える場合、プロトコル名からプロキシの URL へ対応付ける辞書でなくてはなりません。標準では、プロキシのリストを環境変数 `<protocol>_proxy` から読み出します。プロキシ環境変数が設定されていない場合は、Windows 環境では、レジストリのインターネット設定セクションからプロキシ設定を手に入れ、Mac OS X 環境では、OS X システム設定フレームワーク (System Configuration Framework) からプロキシ情報を取得します。

自動検出された proxy を無効にするには、空の辞書を渡してください。

no_proxy 環境変数は、proxy を利用せずにアクセスするべきホストを指定するために利用されます。設定する場合は、カンマ区切りの、ホストネーム suffix のリストで、オプションとして :port を付けることができます。例えば、cern.ch,ncsa.uiuc.edu,some.host:8080.

注釈: 変数 REQUEST_METHOD が設定されている場合、HTTP_PROXY は無視されます; *getproxies()* のドキュメンテーションを参照してください。

class urllib.request.**HTTPPasswordMgr**

(realm, uri) -> (user, password) の対応付けデータベースを保持します。

class urllib.request.**HTTPPasswordMgrWithDefaultRealm**

(realm, uri) -> (user, password) の対応付けデータベースを保持します。レルム None はその他諸々のレルムを表し、他のレルムが該当しない場合に検索されます。

class urllib.request.**HTTPPasswordMgrWithPriorAuth**

uri -> is_authenticated マッピングのデータベースも持つ *HTTPPasswordMgrWithDefaultRealm* のバリエーションです。最初に 401 レスポンスを待つのではなく直ちに認証情報を送るときの条件を判断するために、BasicAuth ハンドラによって使われます。

バージョン 3.5 で追加.

class urllib.request.**AbstractBasicAuthHandler**(*password_mgr=None*)

これは、リモートホストとプロキシの両方に対して HTTP 認証を行うことを助ける mixin クラスです。*password_mgr* は、もし与えられたら *HTTPPasswordMgr* と互換性のあるオブジェクトでなければなりません; サポートすべきインタフェースに関する情報は *HTTPPasswordMgr* **オブジェクト** 節を参照してください。もし *passwd_mgr* が is_authenticated と update_authenticated メソッドも提供するなら (*HTTPPasswordMgrWithPriorAuth* **オブジェクト** を参照)、ハンドラは与えられた URI に対する is_authenticated の結果を用いてリクエストにおいて認証情報を送るかどうかを決定します。もし is_authenticated がその URI に対して True を返すなら、認証情報が送られます。is_authenticated が False なら認証情報は送られません。そして、もし 401 レスポンスを受け取ったら、認証情報を付けて改めてリクエストが送信されます。もし認証が成功したら、それ以降その URI またはその親 URI に対して行われるリクエストが認証情報を自動的に含むように、URI に対して is_authenticated を True に設定するために update_authenticated が呼ばれます。

バージョン 3.5 で追加: is_authenticated サポートが追加されました。

class urllib.request.**HTTPBasicAuthHandler**(*password_mgr=None*)

遠隔ホストとの間での認証を扱います。*password_mgr* を与える場合、*HTTPPasswordMgr* と互換性がなければなりません; 互換性のためにサポートしなければならないインタフェースについての情報はセクション *HTTPPasswordMgr* **オブジェクト** を参照してください。HTTPBasicAuthHandler は、間違った認証スキーマが与えられると *ValueError* を送出します。

class urllib.request.**ProxyBasicAuthHandler**(*password_mgr=None*)

プロキシとの間での認証を扱います。*password_mgr* を与える場合、*HTTPPasswordMgr* と互換性がなければなりません; 互換性のためにサポートしなければならないインタフェースについての情報はセクション *HTTPPasswordMgr* **オブジェクト** を参照してください。

class urllib.request.**AbstractDigestAuthHandler**(*password_mgr=None*)

このクラスは HTTP 認証を補助するための混ぜ込みクラス (mixin class) です。遠隔ホストとプロキシの両方に対応しています。*password_mgr* を与える場合、*HTTPPasswordMgr* と互換性がなければなりません; 互換性のためにサポートしなければならないインタフェースについての情報はセクション *HTTPPasswordMgr* **オブジェクト** を参照してください。

class urllib.request.**HTTPDigestAuthHandler**(*password_mgr=None*)

リモートホストとの認証を扱います。*password_mgr* を与える場合、*HTTPPasswordMgr* と互換性のあるものでなければなりません。サポートしなければならないインターフェースについての情報は *HTTPPasswordMgr* **オブジェクト** 節を参照してください。Digest 認証ハンドラーと Basic 認証ハンドラーの両方が追加された場合、常に Digest 認証を先に試みます。Digest 認証が 40x のレスポンスを再び返すと、Basic 認証ハンドラーに送信されます。このハンドラーメソッドは、Digest および Basic 以外の認証スキームが存在する場合は *ValueError* を送出します。

バージョン 3.3 で変更: 未サポートの認証スキームでは *ValueError* を送出するようになりました。

class urllib.request.**ProxyDigestAuthHandler**(*password_mgr=None*)

プロキシとの間での認証を扱います。*password_mgr* を与える場合、*HTTPPasswordMgr* と互換性がなければなりません; 互換性のためにサポートしなければならないインタフェースについての情報はセクション *HTTPPasswordMgr* **オブジェクト** を参照してください。

class urllib.request.**HTTPHandler**

HTTP の URL を開きます。

class urllib.request.**HTTPSHandler**(*debuglevel=0*, *context=None*, *check_hostname=None*)

HTTPS で URL を開きます。*context* および *check_hostname* は *http.client.HTTPSConnection* のものと同じ意味です。

バージョン 3.2 で変更: *context* および *check_hostname* が追加されました。

class urllib.request.**FileHandler**

ローカルファイルを開きます。

class urllib.request.**DataHandler**

data URL を開きます。

バージョン 3.4 で追加.

class urllib.request.**FTPHandler**

FTP の URL を開きます。

class urllib.request.**CacheFTPHandler**

FTP の URL を開きます。遅延を最小限にするために、開かれている FTP 接続に対するキャッシュを保持します。

class urllib.request.UnknownHandler
その他諸々のためのクラスで、未知のプロトコルの URL を開きます。

class urllib.request.HTTPErrorProcessor
HTTP エラー応答の処理をします。

21.6.1 Request オブジェクト

以下のメソッドは *Request* の公開インターフェースについて説明しています。これらはすべてサブクラスでオーバーライドできます。また、解析したリクエストを調査するためにクライアントで使用するいくつかの属性も定義します。

Request.full_url
コンストラクターに渡されたオリジナルの URL です。

バージョン 3.4 で変更.

Request.full_url は、setter, getter, deleter を持つプロパティです。もし存在すれば、*full_url* はオリジナルのリクエスト URL フラグメント付きで返します。

Request.type
URI スキームです。

Request.host
URI オーソリティです。通常はホスト名ですが、コロンで区切られたポート番号が付随することもあります。

Request.origin_req_host
リクエストしたオリジナルのホスト名です。ポート番号はつきません。

Request.selector
URI パスです。*Request* がプロキシを使用する場合、セレクターはプロキシに渡される完全な URL になります。

Request.data
リクエストのエンティティボディか、指定されない場合は None になります。

バージョン 3.4 で変更: *Request.data* の値が変更されると、もしそれ以前に "Content-Length" ヘッダーの値が設定または計算されていたらヘッダーが削除されるようになりました。

Request.unverifiable
リクエストが **RFC 2965** で定義された証明不能 (unverifiable) であるかどうかを示す論理値です。

Request.method
HTTP リクエストで使うメソッドです。デフォルト値は *None* で、このときは使うメソッドを *get_method()* が通常の方法で決定するということになります。この値を設定する (従って *get_method()* のデフォルトの決定を上書きする) 方法は、*Request* サブクラスでのクラスレベルの設定処理でデフォルト値を提供するか、*Request* のコンストラクタの *method* 引数へ値を渡すかです。

バージョン 3.3 で追加.

バージョン 3.4 で変更: サブクラスでデフォルト値が設定できるようになりました; 以前はコンストラクタ引数からしか設定できませんでした。

Request.**get_method**()
HTTP リクエストメソッドを示す文字列を返します。*Request.method* が None でなければその値を返します。そうでない場合、*Request.data* が None なら 'GET' を、そうでなければ 'POST' を返します。これは HTTP リクエストに対してのみ意味を持ちます。

バージョン 3.3 で変更: get_method は *Request.method* の値を参照するようになりました。

Request.**add_header**(*key*, *val*)
リクエストに新たなヘッダーを追加します。ヘッダーは HTTP ハンドラ以外のハンドラでは無視されます。HTTP ハンドラでは、引数はサーバに送信される ヘッダーのリストに追加されます。同じ名前を持つヘッダを 2 つ以上持つことはできず、*key* の衝突が生じた場合、後で追加したヘッダーが前に追加したヘッダーを上書きします。現時点では、この機能は HTTP の機能を損ねることはありません。というのは、複数回呼び出したときに意味を 持つようなヘッダーには、どれもただ一つのヘッダーを使って同じ機能を果たすための (ヘッダー特有の) 方法があるからです。

Request.**add_unredirected_header**(*key*, *header*)
リダイレクトされたリクエストには追加されないヘッダーを追加します。

Request.**has_header**(*header*)
インスタンスが名前つきヘッダーであるかどうかを (通常のヘッダーと非リダイレクトヘッダの両方を調べて) 返します。

Request.**remove_header**(*header*)
リクエストインスタンス (の通常のヘッダーと非リダイレクトヘッダーの両方) から名前つきヘッダーを削除します。

バージョン 3.4 で追加.

Request.**get_full_url**()
コンストラクタで与えられた URL を返します。

バージョン 3.4 で変更.

Request.full_url を返します。

Request.**set_proxy**(*host*, *type*)
リクエストがプロキシサーバを経由するように準備します。*host* および *type* はインスタンスのもとの設定と置き換えられ ます。インスタンスのセレクタはコンストラクタに与えたもともとの URL になります。

Request.**get_header**(*header_name*, *default=None*)
指定されたヘッダーの値を返します。ヘッダーがない場合は、*default* の値を返します。

Request.**header_items**()
リクエストヘッダーの値を、タプル (header_name, header_value) のリストで返します。

バージョン 3.4 で変更: 3.3 から非推奨だった Request オブジェクトのメソッド add_data, has_data, get_data, get_type, get_host, get_selector, get_origin_req_host, is_unverifiable が削除されました。

21.6.2 OpenerDirector オブジェクト

OpenerDirector インスタンスは以下のメソッドを持っています:

OpenerDirector.**add_handler**(*handler*)

handler should be an instance of *BaseHandler*. The following methods are searched, and added to the possible chains (note that HTTP errors are a special case). Note that, in the following, *protocol* should be replaced with the actual protocol to handle, for example `http_response()` would be the HTTP protocol response handler. Also *type* should be replaced with the actual HTTP code, for example `http_error_404()` would handle HTTP 404 errors.

- `<protocol>_open()` --- ハンドラーが *protocol* の URL を開く方法を知っているかどうかを調べます。

  詳細は、*BaseHandler.<protocol>_open()* を参照してください。

- `http_error_<type>()` --- ハンドラーが HTTP エラーコード *type* の処理方法を知っていることを示すシグナルです。

  詳細は、*BaseHandler.http_error_<nnn>()* を参照してください。

- `<protocol>_error()` --- ハンドラーが (`http` でない) *protocol* のエラーを処理する方法を知っていることを示すシグナルです。

- `<protocol>_request()` --- ハンドラーが *protocol* リクエストのプリプロセス方法を知っていることを示すシグナルです。

  詳細は、*BaseHandler.<protocol>_request()* を参照してください。

- `<protocol>_response()` --- ハンドラーが *protocol* リクエストのポストプロセス方法を知っていることを示すシグナルです。

  詳細は、*BaseHandler.<protocol>_response()* を参照してください。

OpenerDirector.**open**(*url*, *data=None*[, *timeout*])

与えられた *url* (リクエストオブジェクトでも文字列でもかまいません) を開きます。オプションとして *data* を与えることができます。引数、戻り値、および送出される例外は *urlopen()* と同じです (*urlopen()* の場合、標準でインストールされている グローバルな *OpenerDirector* の *open()* メソッドを呼び出します) 。オプションの *timeout* 引数は、接続開始のようなブロックする処理におけるタイムアウト時間を 秒数で指定します。(指定しなかった場合は、グローバルのデフォルト設定が利用されます) タイムアウト機能は、HTTP, HTTPS, FTP 接続でのみ有効です。

OpenerDirector.**error**(*proto*, **args*)

与えられたプロトコルにおけるエラーを処理します。このメソッドは与えられたプロトコルにおける登録済みのエラーハンドラを (プロトコル固有の) 引数で呼び出します。HTTP プロトコルは特殊なケー

スで、特定のエラーハンドラを選び出すのに HTTP レスポンスコードを使います; ハンドラクラスの `http_error_<type>()` メソッドを参照してください。

戻り値および送出される例外は *urlopen()* と同じものです。

OpenerDirector オブジェクトは、以下の 3 つのステージに分けて URL を開きます:

各ステージで OpenerDirector オブジェクトのメソッドがどのような順で呼び出されるかは、ハンドラインスタンスの並び方で決まります。

1. `<protocol>_request()` 形式のメソッドを持つすべてのハンドラーに対してそのメソッドを呼び出し、リクエストのプリプロセスを行います。

2. `<protocol>_open()` のようなメソッドでハンドラーが呼び出され、リクエストを処理します。このステージではハンドラーが非-*None* 値 (例: レスポンス) か例外 (通常は *URLError*) を返した時点で終了します。例外は伝搬できます。

   実際には、上のアルゴリズムではまず `default_open()` という名前のメソッドを呼び出します。このメソッドがすべて *None* を返す場合、同じアルゴリズムを繰り返して、今度は `<protocol>_open()` 形式のメソッドを試します。メソッドがすべて *None* を返すと、さらに同じアルゴリズムを繰り返して `unknown_open()` を呼び出します。

   これらのメソッドの実装には、親となる *OpenerDirector* インスタンスの `open()` や `error()` といったメソッド呼び出しが入る場合があるので注意してください。

3. `<protocol>_response()` 形式のメソッドを持つすべてのハンドラーに対してそのメソッドを呼び出し、リクエストのポストプロセスを行います。

21.6.3 BaseHandler オブジェクト

BaseHandler オブジェクトは直接的に役に立つ 2 つのメソッドと、その他として派生クラスで使われることを想定したメソッドを 提供します。以下は直接的に使うためのメソッドです:

`BaseHandler.add_parent(`*director*`)`
　親オブジェクトとして、`director` を追加します。

`BaseHandler.close()`
　全ての親オブジェクトを削除します。

以下の属性およびメソッドは *BaseHandler* から派生したクラスでのみ使われます。

注釈: 慣習的に、`<protocol>_request()` や `<protocol>_response()` といったメソッドを定義しているサブクラスは `*Processor` と名づけ、その他は `* Handler` と名づけることになっています

`BaseHandler.parent`
　有効な *OpenerDirector* です。この値は違うプロトコルを使って URL を開く場合やエラーを処理する際に使われます。

BaseHandler.**default_open**(*req*)

このメソッドは *BaseHandler* では定義されて **いません** 。しかし、全ての URL をキャッチさせたいなら、サブクラスで定義する 必要があります。

このメソッドが実装されていれば、*OpenerDirector* の親クラスによって呼び出されます。これは *OpenerDirector* の *open()* の戻り値で表されるファイルライクオブジェクトか、または None を返さなければならず、真に想定外の事態が発生した場合を除き、*URLError* を送出しなければなりません (例えば、*MemoryError* は URLError にマップしてはなりません)。

このメソッドはプロトコル固有のオープンメソッドが呼び出される前に呼び出されます。

BaseHandler.<protocol>_open(req)

このメソッドは *BaseHandler* では定義されて **いません** 。しかしプロトコルの URL をキャッチしたいなら、サブクラスで定義する必要があります。

このメソッドが定義されていた場合、*OpenerDirector* から呼び出されます。戻り値は default_open() と同じでなければなりません。

BaseHandler.**unknown_open**(*req*)

このメソッドは *BaseHandler* では定義されて **いません** 。しかし URL を開くための特定のハンドラが登録されていないような URL をキャッチしたいなら、サブクラスで定義する必要があります。

このメソッドが定義されていた場合、*OpenerDirector* から呼び出されます。戻り値は *default_open()* と同じでなければなりません。

BaseHandler.**http_error_default**(*req*, *fp*, *code*, *msg*, *hdrs*)

このメソッドは *BaseHandler* では定義されて **いません** 。しかしその他の処理されなかった HTTP エラーを処理する機能をもたせたいなら、サブクラスで定義する必要があります。このメソッドはエラーに遭遇した *OpenerDirector* から自動的に呼び出されます。その他の状況では普通呼び出すべきではありません。

req は *Request* オブジェクトで、*fp* は HTTP エラー本体を読み出せるようなファイル類似のオブジェクトに なります。*code* は 3 桁の 10 進数からなるエラーコードで、*msg* ユーザ向けのエラーコード解説です。*hdrs* は エラー応答のヘッダーをマップしたオブジェクトです。

返される値および送出される例外は *urlopen()* と同じものでなければなりません。

BaseHandler.http_error_<nnn>(req, fp, code, msg, hdrs)

nnn は 3 桁の 10 進数からなる HTTP エラーコードでなくてはなりません。このメソッドも *BaseHandler* では定義されていませんが、サブクラスのインスタンスで定義されていた場合、エラーコード *nnn* の HTTP エラーが発生した際に呼び出されます。

特定の HTTP エラーに対する処理を行うためには、このメソッドをサブクラスでオーバライドする必要があります。

引数、返される値、および送出される例外は http_error_default() と同じものでなければなりません。

BaseHandler.<protocol>_request(req)

このメソッドは *BaseHandler* では **定義されていません** が、サブクラスで特定のプロトコルのリクエ

ストのプリプロセスを行いたい場合には定義する必要があります。

このメソッドが定義されていると、親となる *OpenerDirector* から呼び出されます。その際、*req* は *Request* オブジェクトになります。戻り値は *Request* オブジェクトでなければなりません。

BaseHandler.<protocol>_response(req, response)

このメソッドは *BaseHandler* では **定義されていません** が、サブクラスで特定のプロトコルのリクエストのポストプロセスを行いたい場合には定義する必要があります。

このメソッドが定義されていると、親となる *OpenerDirector* から呼び出されます。その際、*req* は *Request* オブジェクトになります。*response* は *urlopen()* の戻り値と同じインタフェースを 実装したオブジェクトになります。戻り値もまた、*urlopen()* の戻り値と同じインタフェースを実装したオブジェクトでなければなりません。

21.6.4 HTTPRedirectHandler オブジェクト

注釈: 一部の HTTP リクエストはこのモジュールのクライアントモードからの操作を要求します。その場合、*HTTPError* が送出されます。さまざまなリダイレクションコードの正確な意味についての詳細は **RFC 2616** を参照してください。

セキュリティ上の配慮として、HTTPRedirectHandler に HTTP、HTTPS、あるいは FTP の URL ではないリダイレクトされた URL が存在した場合、HTTPError 例外が送出されます。

HTTPRedirectHandler.**redirect_request**(*req*, *fp*, *code*, *msg*, *hdrs*, *newurl*)

リダイレクトへのレスポンスの *Request* または None を返します。これはサーバーからリダイレクションを受信した時に http_error_30*() メソッドのデフォルトの実装によって呼び出されます。リダイレクションを行わなければならない場合、*newurl* へのリダイレクトを実行するための http_error_30*() を許可する新しい *Request* を返します。その他の場合、この URL を扱うその他のハンドラーがない場合は *HTTPError* を、他のハンドラーでできそうな場合は None を返します。

注釈: このメソッドのデフォルトの実装は、**RFC 2616** に厳密に従ったものではありません。**RFC 2616** では、POST リクエストに対する 301 および 302 応答が、ユーザの承認なく自動的にリダイレクトされてはならないと述べています。現実には、ブラウザは POST を GET に変更することで、これらの応答に対して自動的にリダイレクトを行えるようにしています。デフォルトの実装でも、この挙動を再現しています。

HTTPRedirectHandler.**http_error_301**(*req*, *fp*, *code*, *msg*, *hdrs*)

Location: か URI: の URL にリダイレクトします。このメソッドは HTTP における 'moved permanently' レスポンスを取得した際に 親オブジェクトとなる *OpenerDirector* によって呼び出されます。

HTTPRedirectHandler.**http_error_302**(*req*, *fp*, *code*, *msg*, *hdrs*)

http_error_301() と同じですが、'found' レスポンスに対して呼び出されます。

HTTPRedirectHandler.**http_error_303**(*req*, *fp*, *code*, *msg*, *hdrs*)

http_error_301() と同じですが、'see other' レスポンスに対して呼び出されます。

HTTPRedirectHandler.**http_error_307**(*req*, *fp*, *code*, *msg*, *hdrs*)

http_error_301() と同じですが、'temporary redirect' レスポンスに対して呼び出されます。

21.6.5 HTTPCookieProcessor オブジェクト

HTTPCookieProcessor インスタンスは属性をひとつだけ持ちます:

HTTPCookieProcessor.**cookiejar**

Cookie の入っている *http.cookiejar.CookieJar* オブジェクトです。

21.6.6 ProxyHandler オブジェクト

ProxyHandler.<protocol>_open(request)

ProxyHandler は、コンストラクターで与えた辞書 *proxies* にプロキシが設定されているような *protocol* すべてについて、メソッド `<protocol>_open()` を持つことになります。このメソッドは `request.set_proxy()` を呼び出して、リクエストがプロキシを通過できるように修正します。その後連鎖するハンドラーの中から次のハンドラーを呼び出して実際にプロトコルを実行します。

21.6.7 HTTPPasswordMgr オブジェクト

以下のメソッドは *HTTPPasswordMgr* および *HTTPPasswordMgrWithDefaultRealm* オブジェクトで利用できます。

HTTPPasswordMgr.**add_password**(*realm*, *uri*, *user*, *passwd*)

uri は単一の URI でも複数の URI からなるシーケンスでもかまいません。*realm* 、*user* および *passwd* は文字列でなくてはなりません。このメソッドによって、*realm* と与えられた URI の上位 URI に対して `(user, passwd)` が認証トークンとして使われるようになります。

HTTPPasswordMgr.**find_user_password**(*realm*, *authuri*)

与えられたレルムおよび URI に対するユーザ名またはパスワードがあればそれを取得します。該当するユーザ名／パスワードが存在しない場合、このメソッドは `(None, None)` を返します。

HTTPPasswordMgrWithDefaultRealm オブジェクトでは、与えられた *realm* に対して該当するユーザ名/パスワードが存在しない場合、レルム `None` が検索されます。

21.6.8 HTTPPasswordMgrWithPriorAuth オブジェクト

このパスワードマネージャは *HTTPPasswordMgrWithDefaultRealm* を継承して、認証の証明書を常に送らないといけない URI を追跡する機能をサポートしています。

HTTPPasswordMgrWithPriorAuth.**add_password**(*realm*, *uri*, *user*, *passwd*, *is_authenticated=False*)

realm, *uri*, *user*, *passwd* は *HTTPPasswordMgr.add_password()* のものと同じです。*is_authenticated* は、与えられた URI や URI のリストの `is_authenticated` フラグの初期値に設定されます。*is_authenticated* に `True` を指定した場合は、*realm* は無視されます。

HTTPPasswordMgrWithPriorAuth.**find_user_password**(*realm*, *authuri*)

HTTPPasswordMgrWithDefaultRealm オブジェクトに対する同名のメソッドと同じです。

HTTPPasswordMgrWithPriorAuth.**update_authenticated**(*self*, *uri*, *is_authenticated=False*)

与えられた *url* や URI のリストの `is_authenticated` フラグを更新します。

HTTPPasswordMgrWithPriorAuth.**is_authenticated**(*self*, *authuri*)

与えられた URI の `is_authenticated` フラグの現在の状態を返します。

21.6.9 AbstractBasicAuthHandler オブジェクト

AbstractBasicAuthHandler.**http_error_auth_reqed**(*authreq*, *host*, *req*, *headers*)

ユーザ名／パスワードを取得し、再度サーバへのリクエストを試みることで、サーバからの認証リクエストを処理します。*authreq* はリクエストにおいて レルムに関する情報が含まれているヘッダーの名前、*host* は認証を行う対象の URL とパスを指定します、*req* は (失敗した) *Request* オブジェクト、そして *headers* はエラーヘッダーでなくてはなりません。

host は、オーソリティ (例 `"python.org"`) か、オーソリティコンポーネントを含む URL (例 `"http://python.org"`) です。どちらの場合も、オーソリティはユーザ情報コンポーネントを含んではいけません (なので、`"python.org"` や `"python.org:80"` は正しく、`"joe:password@python.org"` は不正です) 。

21.6.10 HTTPBasicAuthHandler オブジェクト

HTTPBasicAuthHandler.**http_error_401**(*req*, *fp*, *code*, *msg*, *hdrs*)
認証情報がある場合、認証情報付きで再度リクエストを試みます。

21.6.11 ProxyBasicAuthHandler オブジェクト

ProxyBasicAuthHandler.**http_error_407**(*req*, *fp*, *code*, *msg*, *hdrs*)
認証情報がある場合、認証情報付きで再度リクエストを試みます。

21.6.12 AbstractDigestAuthHandler オブジェクト

AbstractDigestAuthHandler.**http_error_auth_reqed**(*authreq*, *host*, *req*, *headers*)
authreq はリクエストにおいてレルムに関する情報が含まれているヘッダーの名前、*host* は認証を行う対象のホスト名、*req* は (失敗した) *Request* オブジェクト、そして *headers* はエラーヘッダーでなくてはなりません。

21.6.13 HTTPDigestAuthHandler オブジェクト

HTTPDigestAuthHandler.**http_error_401**(*req*, *fp*, *code*, *msg*, *hdrs*)
認証情報がある場合、認証情報付きで再度リクエストを試みます。

21.6.14 ProxyDigestAuthHandler オブジェクト

ProxyDigestAuthHandler.**http_error_407**(*req*, *fp*, *code*, *msg*, *hdrs*)
認証情報がある場合、認証情報付きで再度リクエストを試みます。

21.6.15 HTTPHandler オブジェクト

HTTPHandler.**http_open**(*req*)

HTTP リクエストを送ります。`req.has_data()` に応じて、GET または POST のどちらでも送ることができます。

21.6.16 HTTPSHandler オブジェクト

HTTPSHandler.**https_open**(*req*)

HTTPS リクエストを送ります。`req.has_data()` に応じて、GET または POST のどちらでも送ることができます。

21.6.17 FileHandler オブジェクト

FileHandler.**file_open**(*req*)

ホスト名がない場合、またはホスト名が `'localhost'` の場合にファイルをローカルでオープンします。

バージョン 3.2 で変更: このメソッドはローカルのホスト名に対してのみ適用可能です。リモートホスト名が与えられた場合、*URLError* が送出されます。

21.6.18 DataHandler オブジェクト

DataHandler.**data_open**(*req*)

Read a data URL. This kind of URL contains the content encoded in the URL itself. The data URL syntax is specified in **RFC 2397**. This implementation ignores white spaces in base64 encoded data URLs so the URL may be wrapped in whatever source file it comes from. But even though some browsers don't mind about a missing padding at the end of a base64 encoded data URL, this implementation will raise an *ValueError* in that case.

21.6.19 FTPHandler オブジェクト

FTPHandler.**ftp_open**(*req*)
req で表されるファイルを FTP 越しにオープンします。ログインは常に空のユーザネームおよびパスワードで行われます。

21.6.20 CacheFTPHandler オブジェクト

CacheFTPHandler オブジェクトは *FTPHandler* オブジェクトに以下のメソッドを追加したものです:

CacheFTPHandler.**setTimeout**(*t*)
接続のタイムアウトを *t* 秒に設定します。

CacheFTPHandler.**setMaxConns**(*m*)
キャッシュ付き接続の最大接続数を *m* に設定します。

21.6.21 UnknownHandler オブジェクト

UnknownHandler.**unknown_open**()
URLError 例外を送出します。

21.6.22 HTTPErrorProcessor オブジェクト

HTTPErrorProcessor.**http_response**(*request*, *response*)
HTTP エラー応答の処理をします。

エラーコード 200 の場合、レスポンスオブジェクトを即座に返します。

200 以外のエラーコードの場合、これは単に `http_error_<type>()` ハンドラーメソッドに *OpenerDirector.error()* 経由で処理を渡します。他にそのエラーを処理するハンドラーがない場合、*HTTPDefaultErrorHandler* は最後に *HTTPError* を送出します。

HTTPErrorProcessor.**https_response**(*request*, *response*)
HTTPS エラー応答の処理をします。

振る舞いは *http_response()* と同じです。

21.6.23 使用例

以下の例の他に urllib-howto に多くの例があります。

以下の例では、python.org のメインページを取得して、その最初の 300 バイト分を表示します。

```
>>> import urllib.request
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(300))
...
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">\n\n\n<html
xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">\n\n<head>\n
<meta http-equiv="content-type" content="text/html; charset=utf-8" />\n
<title>Python Programming '
```

urlopen は bytes オブジェクトを返すことに注意してください。これは urlopen が、HTTP サーバーから受信したバイトストリームのエンコーディングを自動的に決定できないためです。一般に、返された bytes オブジェクトを文字列にデコードするためのエンコーディングの決定あるいは推測はプログラム側が行います。

以下の W3C ドキュメント https://www.w3.org/International/O-charset には (X)HTML や XML ドキュメントでそのエンコーディング情報を指定するさまざまな方法の一覧があります。

python.org ウェブサイトでは *utf-8* エンコーディングを使用しており、それをその meta タグで指定していますので、bytes オブジェクトのデコードも同様に行います。

```
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(100).decode('utf-8'))
...
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

コンテキストマネージャー を使用しないアプローチでも同様の結果を得ることができます。

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.python.org/')
>>> print(f.read(100).decode('utf-8'))
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

以下の例では、データストリームを CGI の標準入力へ送信し、返されたデータを読み込みます。この例は Python が SSL をサポートするように設定してインストールされている場合のみ動作します。

```
>>> import urllib.request
>>> req = urllib.request.Request(url='https://localhost/cgi-bin/test.cgi',
...                       data=b'This data is passed to stdin of the CGI')
>>> with urllib.request.urlopen(req) as f:
...     print(f.read().decode('utf-8'))
...
Got Data: "This data is passed to stdin of the CGI"
```

上の例で使われているサンプルの CGI は以下のようになっています:

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print('Content-type: text/plain\n\nGot Data: "%s"' % data)
```

これは *Request* を使った PUT リクエストの例です:

```
import urllib.request
DATA = b'some data'
req = urllib.request.Request(url='http://localhost:8080', data=DATA,method='PUT')
with urllib.request.urlopen(req) as f:
    pass
print(f.status)
print(f.reason)
```

以下はベーシック HTTP 認証の例です:

```
import urllib.request
# Create an OpenerDirector with support for Basic HTTP Authentication...
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-updates.py',
                          user='klem',
                          passwd='kadidd!ehopper')
opener = urllib.request.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib.request.install_opener(opener)
urllib.request.urlopen('http://www.example.com/login.html')
```

build_opener() はデフォルトで沢山のハンドラを提供しており、その中に *ProxyHandler* があります。デフォルトでは、*ProxyHandler* は <scheme>_proxy という環境変数を使います。ここで <scheme> は URL スキームです。例えば、HTTP プロキシの URL を得るには、環境変数 http_proxy を読み出します。

この例では、デフォルトの *ProxyHandler* を置き換えてプログラム的に作成したプロキシ URL を使うようにし、*ProxyBasicAuthHandler* でプロキシ認証サポートを追加します。

```
proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib.request.build_opener(proxy_handler, proxy_auth_handler)
# This time, rather than install the OpenerDirector, we use it directly:
opener.open('http://www.example.com/login.html')
```

以下は HTTP ヘッダーを追加する例です:

headers 引数を使って *Request* コンストラクタを呼び出す方法の他に、以下のようにできます:

```
import urllib.request
req = urllib.request.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
# Customize the default User-Agent header value:
req.add_header('User-Agent', 'urllib-example/0.1 (Contact: . . .)')
r = urllib.request.urlopen(req)
```

OpenerDirector は全ての *Request* に *User-Agent* ヘッダーを自動的に追加します。これを変更するには以下のようにします:

```
import urllib.request
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/')
```

また、*Request* が *urlopen()* (や *OpenerDirector.open()*) に渡される際には、いくつかの標準ヘッダー (*Content-Length*, *Content-Type* および *Host*) も追加されることを忘れないでください。

以下は GET メソッドを使ってパラメータを含む URL を取得するセッションの例です:

```
>>> import urllib.request
>>> import urllib.parse
>>> params = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> url = "http://www.musi-cal.com/cgi-bin/query?%s" % params
>>> with urllib.request.urlopen(url) as f:
...     print(f.read().decode('utf-8'))
...
```

以下の例では、POST メソッドを使用しています。urlencode から出力されたパラメーターは urlopen にデータとして渡される前に bytes にエンコードされていることに注意してください:

```
>>> import urllib.request
>>> import urllib.parse
>>> data = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> data = data.encode('ascii')
>>> with urllib.request.urlopen("http://requestb.in/xrbl82xr", data) as f:
...     print(f.read().decode('utf-8'))
...
```

以下の例では、環境変数による設定内容に対して上書きする形で HTTP プロキシを明示的に設定しています:

```
>>> import urllib.request
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.request.FancyURLopener(proxies)
>>> with opener.open("http://www.python.org") as f:
...     f.read().decode('utf-8')
...
```

以下の例では、環境変数による設定内容に対して上書きする形で、まったくプロキシを使わないよう設定しています:

```
>>> import urllib.request
>>> opener = urllib.request.FancyURLopener({})
>>> with opener.open("http://www.python.org/") as f:
...     f.read().decode('utf-8')
...
```

21.6.24 レガシーインターフェース

以下の関数およびクラスは、Python 2 のモジュール `urllib` (`urllib2` ではありません) から移植されたものです。これらは将来的に廃止されるかもしれません。

`urllib.request.urlretrieve`(*url*, *filename=None*, *reporthook=None*, *data=None*)

URL で表されるネットワークオブジェクトをローカルファイルにコピーします。URL がローカルファイルを示している場合は、ファイル名が与えられない限りオブジェクトはコピーされません。戻り値はタプル `(filename, headers)` になり、*filename* はオブジェクトが保存されたローカルファイル名、*headers* は (リモートオブジェクトに対しては) *urlopen()* が返したオブジェクトの `info()` メソッドが返すものすべてになります。例外は *urlopen()* のものと同じになります。

The second argument, if present, specifies the file location to copy to (if absent, the location will be a tempfile with a generated name). The third argument, if present, is a callable that will be called once on establishment of the network connection and once after each block read thereafter. The callable will be passed three arguments; a count of blocks transferred so far, a block size in bytes, and the total size of the file. The third argument may be `-1` on older FTP servers which do not return a file size in response to a retrieval request.

以下は最も一般的な使用例です:

```
>>> import urllib.request
>>> local_filename, headers = urllib.request.urlretrieve('http://python.org/')
>>> html = open(local_filename)
>>> html.close()
```

url が `http:` スキーム識別子を使用していた場合、任意の引数 *data* は `POST` リクエストの指定に使用される場合があります (通常のリクエストタイプは `GET` です)。引数 *data* は標準 *application/x-www-form-urlencoded* 形式のバイトオブジェクトでなければなりません。*urllib.parse.urlencode()* 関数を参照してください。

urlretrieve() は、予想 (これは *Content-Length* ヘッダーにより通知されるサイズです) よりも取得できるデータ量が少ないことを検知した場合、`ContentTooShortError` を発生します。これは、例えば、ダウンロードが中断された場合などに発生します。

Content-Length はデータ量の下限です: 読み込むデータ量がこれを超えている場合 urlretrieve はそれらも読み込みますが、利用できるデータがこれを下回った場合、例外が送出されます。

このような場合にもダウンロードされたデータを取得することは可能で、これは exception インスタンスの `content` 属性に保存されています。

Content-Length ヘッダーが与えられなければ urlretrieve はダウンロードしたデータのサイズをチェックできません。この場合ダウンロードは正常に完了したとみなすしかありません。

urllib.request.urlcleanup()
以前の *urlretrieve()* 呼び出し後に残っているかもしれない一時ファイルをクリーンアップします。

class urllib.request.URLopener(*proxies=None*, ***x509*)
バージョン 3.3 で非推奨.

URL をオープンし、読み出すためのクラスの基底クラスです。`http:`, `ftp:`, `file:` 以外のスキームを使ったオブジェクトのオープンをサポートしたいのでないかぎり、*FancyURLopener* を使おうと思うことになるでしょう。

デフォルトでは、*URLopener* クラスは *User-Agent* ヘッダーとして `urllib/VVV` を送信します。ここで *VVV* は *urllib* のバージョン番号です。アプリケーションで独自の *User-Agent* ヘッダーを送信したい場合は、*URLopener* かまたは *FancyURLopener* のサブクラスを作成し、サブクラス定義においてクラス属性 *version* を適切な文字列値に設定することで行うことができます。

オプションのパラメーター *proxies* はスキーム名をプロキシの URL にマップする辞書でなければなりません。空の辞書はプロキシ機能を完全にオフにします。デフォルトの値は `None` で、この場合、*urlopen()* の定義で述べたように、プロキシを設定する環境変数が存在するならそれを使います。

追加のキーワードパラメーターは *x509* に集められますが、これは `https:` スキームを使った際のクライアント認証に使われることがあります。キーワード引数 *key_file* および *cert_file* が SSL 鍵と証明書を設定するためにサポートされています; クライアント認証をするには両方が必要です。

URLopener オブジェクトはサーバーがエラーコードを返した場合に *OSError* 例外を送出します。

open(*fullurl*, *data=None*)
適切なプロトコルを使って *fullurl* を開きます。このメソッドはキャッシュとプロキシ情報を設定し、その後適切な open メソッドを入力引数つきで呼び出します。認識できないスキームが与えられた場合、*open_unknown()* が呼び出されます。*data* 引数は *urlopen()* の引数 *data* と同じ意味を持っています。

This method always quotes *fullurl* using *quote()*.

open_unknown(*fullurl*, *data=None*)
オーバライド可能な、未知のタイプの URL を開くためのインタフェースです。

retrieve(*url*, *filename=None*, *reporthook=None*, *data=None*)
url の内容を取得し、*filename* に保存します。戻り値は、ローカルファイル名と、レスポンスヘッダーが含まれる *email.message.Message* (リモート URL の場合) か `None` (ローカル URL の場合) からなるタプルになります。呼び出し側は、その後 *filename* を開いてその内容を読み込まなければなりません。*filename* が与えられず、URL がローカルファイルを参照している場合、入力ファイル名が返されます。URL がローカルでなく、*filename* が与えられていない場合、ファイル名は入力 URL のパスの最後の構成要素のサフィックスとマッチするサフィックスを持つ *tempfile.mktemp()* の出力になります。*reporthook* が与えられている場合、3 つの数値 (チャンク数、読み込んだチャンクの最大サイズ、および総ダウンロードサイズ --- 不明の場合は -1) の引

数を受け取る関数でなければなりません。これは開始時に 1 回と、ネットワークからデータのチャンクを読み込む度に呼び出されます。*reporthook* はローカル URL に対しては無視されます。

url が `http:` スキーム識別子を使用していた場合、任意の引数 *data* は `POST` リクエストの指定に使用される場合があります (通常のリクエストタイプは `GET` です)。引数 *data* は標準 *application/x-www-form-urlencoded* 形式でなければなりません。*urllib.parse.urlencode()* 関数を参照してください。

version

URL をオープンするオブジェクトのユーザエージェントを指定する変数です。*urllib* を特定のユーザエージェントであるとサーバに通知するには、サブクラスの中でこの値をクラス変数として値を設定するか、コンストラクタの中でベースクラスを呼び出す前に値を設定してください。

class urllib.request.FancyURLopener(...)

バージョン 3.3 で非推奨.

FancyURLopener は *URLopener* のサブクラスで、以下の HTTP レスポンスコード: 301、302、303、307、および 401 を取り扱う機能を提供します。レスポンスコード 30x に対しては、*Location* ヘッダーを使って実際の URL を取得します。レスポンスコード 401 (認証が要求されていることを示す) に対しては、BASIC 認証 (basic HTTP authintication) が行われます。レスポンスコード 30x に対しては、最大で *maxtries* 属性に指定された数だけ再帰呼び出しを行うようになっています。この値はデフォルトで 10 です。

その他のレスポンスコードについては、`http_error_default()` が呼ばれます。これはサブクラスでエラーを適切に処理するようにオーバーライドすることができます。

注釈: **RFC 2616** によると、POST 要求に対する 301 および 302 応答はユーザの承認無しに自動的にリダイレクトしてはなりません。実際は、これらの応答に対して自動リダイレクトを許すブラウザでは POST を GET に変更しており、*urllib* でもこの動作を再現します。

コンストラクタに与えるパラメーターは *URLopener* と同じです。

注釈: 基本的な HTTP 認証を行う際、*FancyURLopener* インスタンスは *prompt_user_passwd()* メソッドを呼び出します。このメソッドはデフォルトでは実行を制御している端末上で認証に必要な情報を要求するように実装されています。必要ならば、このクラスのサブクラスにおいてより適切な動作をサポートするために *prompt_user_passwd()* メソッドをオーバライドしてもかまいません。

FancyURLopener クラスはオーバライド可能な追加のメソッドを提供しており、適切な振る舞いをさせることができます:

prompt_user_passwd(*host*, *realm***)**

指定されたセキュリティ領域 (security realm) 下にある与えられたホストにおいて、ユーザー認証に必要な情報を返すための関数です。この関数が返す値は `(user, password)` からなるタプルでなければなりません。値は Basic 認証で使われます。

このクラスでの実装では、端末に情報を入力するようプロンプトを出します; ローカルの環境において適切な形で対話型モデルを使うには、このメソッドをオーバライドしなければなりません。

21.6.25 urllib.request の制限事項

- 現在、次のプロトコルのみサポートされています: HTTP (バージョン 0.9 および 1.0)、FTP、ローカルファイル、およびデータ URL

  バージョン 3.4 で変更: データ URL サポートが追加されました。

- *urlretrieve()* のキャッシュ機能は、誰かが Expiration time ヘッダーの正しい処理をハックする時間を見つけるまで無効にされています。

- ある URL がキャッシュにあるかどうか調べるような関数があればと思っています。

- 後方互換性のため、URL がローカルシステム上のファイルを指しているように見えるにも関わらずファイルを開くことができなければ、URL は FTP プロトコルを使って再解釈されます。この機能は時として混乱を招くエラーメッセージを引き起こします。

- 関数 *urlopen()* および *urlretrieve()* は、ネットワーク接続が確立されるまでの間、一定でない長さの遅延を引き起こすことがあります。このことは、これらの関数を使ってインタラクティブな Web クライアントを構築するのはスレッドなしには難しいことを意味します。

- *urlopen()* あるいは *urlretrieve()* が返すデータはサーバーから返された生データです。これは (画像のような) バイナリ、プレーンテキスト、あるいは (例えば) HTML などになります。HTTP プロトコルはレスポンスヘッダー内でタイプ情報を提供しており、*Content-Type* ヘッダーを見ることで調査できます。返されたデータが HTML の場合、モジュール *html.parser* を使用してこれを解析できます。

- FTP プロトコルを扱うコードでは、ファイルとディレクトリを区別できません。このことから、アクセスできないファイルを指している URL からデータを読み出そうとすると、予期しない動作を引き起こす場合があります。URL が / で終わっていれば、ディレクトリを指しているものとみなして、それに適した処理を行います。しかし、ファイルの読み出し操作が 550 エラー (URL が存在しないか、主にパーミッションの理由でアクセスできない) になった場合、URL がディレクトリを指していて、末尾の / を忘れたケースを処理するため、パスをディレクトリとして扱います。このために、パーミッションのためにアクセスできないファイルを fetch しようとすると、FTP コードはそのファイルを開こうとして 550 エラーに陥り、次にディレクトリ一覧を表示しようとするため、誤解を生むような結果を引き起こす可能性があるのです。よく調整された制御が必要なら、*ftplib* モジュールを使うか、*FancyURLopener* をサブクラス化するか、*_urlopener* を変更して目的に合わせるよう検討してください。

21.7 urllib.response --- urllib で使用するレスポンスクラス

urllib.response モジュールは、read() および readline() を含む 最小限のファイルライクインターフェースを定義する関数およびクラスを定義しています。代表的なレスポンスオブジェクトは addinfourl インスタンスで、レスポンスヘッダーを返す info() メソッドおよび URL を返す geturl() メソッドを定義しています。このモジュールで定義された関数は、*urllib.request* モジュール内で使用されます。

21.8 urllib.parse --- URL を解析して構成要素にする

ソースコード: Lib/urllib/parse.py

このモジュールでは URL (Uniform Resource Locator) 文字列をその構成要素 (アドレススキーム、ネットワーク上の位置、パスその他) に分解したり、構成要素を URL に組みなおしたり、" 相対 URL (relative URL)" を指定した " 基底 URL (base URL)" に基づいて絶対 URL に変換するための標準的なインタフェースを定義しています。

このモジュールは Relative Uniform Resource Locators (相対 URL) に関するインターネット RFC に適合するよう設計されており、次の URL スキームをサポートしています: file, ftp, gopher, hdl, http, https, imap, mailto, mms, news, nntp, prospero, rsync, rtsp, rtspu, sftp, shttp, sip, sips, snews, svn, svn+ssh, telnet, wais, ws, wss

urllib.parse モジュールは、大きく分けると URL の解析を行う関数と URL のクオートを行う関数を定義しています。以下にこれらの詳細を説明します。

21.8.1 URL の解析

URL 解析関数は、URL 文字列を各構成要素に分割するか、あるいは URL の構成要素を組み合わせて URL 文字列を生成します。

urllib.parse.urlparse(*urlstring*, *scheme=''*, *allow_fragments=True*)

Parse a URL into six components, returning a 6-item *named tuple*. This corresponds to the general structure of a URL: scheme://netloc/path;parameters?query#fragment. Each tuple item is a string, possibly empty. The components are not broken up in smaller parts (for example, the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the result, except for a leading slash in the *path* component, which is retained if present. For example:

```
>>> from urllib.parse import urlparse
>>> o = urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
>>> o
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> o.scheme
'http'
```

(次のページに続く)

(前のページからの続き)

```
>>> o.port
80
>>> o.geturl()
'http://www.cwi.nl:80/%7Eguido/Python.html'
```

RFC 1808 にある文法仕様に基づき、urlparse は '//' で始まる場合にのみ netloc を認識します。それ以外の場合は、入力は相対 URL であると推定され、path 部分で始まることになります。

```
>>> from urllib.parse import urlparse
>>> urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('www.cwi.nl/%7Eguido/Python.html')
ParseResult(scheme='', netloc='', path='www.cwi.nl/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('help/Python.html')
ParseResult(scheme='', netloc='', path='help/Python.html', params='',
            query='', fragment='')
```

scheme 引数によってデフォルトのアドレススキームを与えると、アドレススキームを指定していない URL のみに使用されます。常に許されるデフォルトの `''` (`b''` に自動変換出来る) を除き、*urlstring* と同じ型 (テキストもしくはバイト列) であるべきです。

引数 *allow_fragments* が false の場合、フラグメント識別子は認識されません。その代わり、それはパス、パラメータ、またはクエリ要素の一部として解析され、戻り値の `fragment` は空文字に設定されます。

戻り値は 名前付きタプル です。これは、インデックス指定もしくは以下のような名前属性で要素にアクセスできることを意味します:

属性	インデックス	値	指定されなかった場合の値
`scheme`	0	URL スキーム	*scheme* パラメータ
`netloc`	1	ネットワーク上の位置	空文字列
`path`	2	階層的パス	空文字列
`params`	3	最後のパス要素に対するパラメータ	空文字列
`query`	4	クエリ要素	空文字列
`fragment`	5	フラグメント識別子	空文字列
`username`		ユーザ名	*None*
`password`		パスワード	*None*
`hostname`		ホスト名 (小文字)	*None*
`port`		ポート番号を表わす整数 (もしあれば)	*None*

URL 中で不正なポートが指定されている場合、`port` 属性を読みだすと、*ValueError* を送出します。結果オブジェクトのより詳しい情報は 構造化された解析結果 節を参照してください。

`netloc` 属性にマッチしなかった角括弧があると *ValueError* を送出します。

Characters in the `netloc` attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a *ValueError*. If the URL is decomposed before parsing, no error will be raised.

As is the case with all named tuples, the subclass has a few additional methods and attributes that are particularly useful. One such method is `_replace()`. The `_replace()` method will return a new ParseResult object replacing specified fields with new values.

```
>>> from urllib.parse import urlparse
>>> u = urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
>>> u
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> u._replace(scheme='http')
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
```

警告: *urlparse()* does not perform validation. See *URL parsing security* for details.

バージョン 3.2 で変更: IPv6 URL の解析も行えるようになりました。

バージョン 3.3 で変更: **RFC 3986** に従い、fragment はすべての URL スキームに対して解析されるようになりました (*allow_fragment* が偽の場合は除く)。

バージョン 3.6 で変更: 範囲外のポート番号を指定すると、*None* を返す代わりに、*ValueError* を送出するようになりました。

バージョン 3.8 で変更: Characters that affect netloc parsing under NFKC normalization will now raise *ValueError*.

`urllib.parse.parse_qs`(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

文字列引数として渡されたクエリ文字列 (*application/x-www-form-urlencoded* 型のデータ) を解析します。解析されたデータを辞書として返します。辞書のキーは一意なクエリ変数名で、値は各変数名に対する値からなるリストです。

任意の引数 *keep_blank_values* は、パーセントエンコードされたクエリの中の値が入っていないクエリの値を空白文字列と見なすかどうかを示すフラグです。値が真であれば、値の入っていないフィールドは空文字列のままになります。標準では偽で、値の入っていないフィールドを無視し、そのフィールドはクエリに含まれていないものとして扱います。

任意の引数 *strict_parsing* はパース時のエラーをどう扱うかを決めるフラグです。値が偽なら (デフォルトの設定です)、エラーは暗黙のうちに無視します。値が真なら *ValueError* 例外を送出します。

任意のパラメータ *encoding* および *errors* はパーセントエンコードされたシーケンスを Unicode 文字にデコードする方法を指定します。これは *bytes.decode()* メソッドに渡されます。

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then

throws a *ValueError* if there are more than *max_num_fields* fields read.

The optional argument *separator* is the symbol to use for separating the query arguments. It defaults to `&`.

このような辞書をクエリ文字列に変換するには *urllib.parse.urlencode()* 関数を (`doseq` パラメータに `True` を指定して) 使用します。

バージョン 3.2 で変更: *encoding* および *errors* パラメータが追加されました。

バージョン 3.8 で変更: *max_num_fields* 引数が追加されました。

バージョン 3.8.8 で変更: Added *separator* parameter with the default value of `&`. Python versions earlier than Python 3.8.8 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

urllib.parse.parse_qsl(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

文字列引数として渡されたクエリ文字列 (*application/x-www-form-urlencoded* 型のデータ) を解析します。解析されたデータは名前と値のペアからなるリストです。

任意の引数 *keep_blank_values* は、パーセントエンコードされたクエリの中の値が入っていないクエリの値を空白文字列と見なすかどうかを示すフラグです。値が真であれば、値の入っていないフィールドは空文字列のままになります。標準では偽で、値の入っていないフィールドを無視し、そのフィールドはクエリに含まれていないものとして扱います。

任意の引数 *strict_parsing* はパース時のエラーをどう扱うかを決めるフラグです。値が偽なら (デフォルトの設定です)、エラーは暗黙のうちに無視します。値が真なら *ValueError* 例外を送出します。

任意のパラメータ *encoding* および *errors* はパーセントエンコードされたシーケンスを Unicode 文字にデコードする方法を指定します。これは *bytes.decode()* メソッドに渡されます。

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a *ValueError* if there are more than *max_num_fields* fields read.

The optional argument *separator* is the symbol to use for separating the query arguments. It defaults to `&`.

ペアのリストからクエリ文字列を生成する場合には *urllib.parse.urlencode()* 関数を使用します。

バージョン 3.2 で変更: *encoding* および *errors* パラメータが追加されました。

バージョン 3.8 で変更: *max_num_fields* 引数が追加されました。

バージョン 3.8.8 で変更: Added *separator* parameter with the default value of `&`. Python versions earlier than Python 3.8.8 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

urllib.parse.urlunparse(*parts*)

`urlparse()` が返すような形式のタプルから URL を構築します。*parts* 引数は任意の 6 要素イテラブルです。解析された元の URL が、不要な区切り文字を持っていた場合には、多少違いはあるが等価な

URL になるかもしれません (例えばクエリ内容が空の ? のようなもので、RFC はこれらを等価だと述べています)。

urllib.parse.urlsplit(*urlstring*, *scheme=''*, *allow_fragments=True*)

urlparse() に似ていますが、URL から params を切り離しません。このメソッドは通常、URL の *path* 部分において、各セグメントにパラメータ指定をできるようにした最近の URL 構文 (**RFC 2396** 参照) が必要な場合に、*urlparse()* の代わりに使われます。パスセグメントとパラメータを分割するためには分割用の関数が必要です。この関数は 5 要素の term:*named tuple* を返します:

```
(addressing scheme, network location, path, query, fragment identifier).
```

戻り値は **名前付きタプル** で、インデックスによってもしくは名前属性として要素にアクセスできます:

属性	インデックス	値	指定されなかった場合の値
scheme	0	URL スキーム	*scheme* パラメータ
netloc	1	ネットワーク上の位置	空文字列
path	2	階層的パス	空文字列
query	3	クエリ要素	空文字列
fragment	4	フラグメント識別子	空文字列
username		ユーザ名	*None*
password		パスワード	*None*
hostname		ホスト名 (小文字)	*None*
port		ポート番号を表わす整数 (もしあれば)	*None*

URL 中で不正なポートが指定されている場合、port 属性を読みだすと、*ValueError* を送出します。結果オブジェクトのより詳しい情報は **構造化された解析結果** 節を参照してください。

netloc 属性にマッチしなかった角括弧があると *ValueError* を送出します。

Characters in the netloc attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of /, ?, #, @, or : will raise a *ValueError*. If the URL is decomposed before parsing, no error will be raised.

Following some of the WHATWG spec that updates RFC 3986, leading C0 control and space characters are stripped from the URL. \n, \r and tab \t characters are removed from the URL at any position.

警告: *urlsplit()* does not perform validation. See *URL parsing security* for details.

バージョン 3.6 で変更: 範囲外のポート番号を指定すると、*None* を返す代わりに、*ValueError* を送出するようになりました。

バージョン 3.8 で変更: Characters that affect netloc parsing under NFKC normalization will now raise *ValueError*.

バージョン 3.8.10 で変更: ASCII newline and tab characters are stripped from the URL.

バージョン 3.8.17 で変更: Leading WHATWG C0 control and space characters are stripped from the URL.

urllib.parse.**urlunsplit**(*parts*)

urlsplit() が返すような形式のタプル中のエレメントを組み合わせて、文字列の完全な URL にします。*parts* 引数は任意の 5 要素イテラブルです。解析された元の URL が、不要な区切り文字を持っていた場合には、多少違いはあるが等価な URL になるかもしれません (例えばクエリ内容が空の ? のようなもので、RFC はこれらを等価だと述べています)。

urllib.parse.**urljoin**(*base*, *url*, *allow_fragments=True*)

" 基底 URL"(*base*) と別の URL(*url*) を組み合わせて、完全な URL (" 絶対 URL") を構成します。くだけて言えば、この関数は相対 URL にない要素を提供するために基底 URL の要素、特にアドレススキーム、ネットワーク上の位置、およびパス (の一部) を使います。例えば:

```
>>> from urllib.parse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

allow_fragments 引数は *urlparse()* における引数と同じ意味とデフォルトを持ちます。

注釈: *url* が (// か scheme:// で始まっている) 絶対 URL であれば、その *url* のホスト名と / もしくは scheme は結果に反映されます。例えば:

```
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         '//www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

もしこの動作が望みのものでない場合は、*url* を *urlsplit()* と *urlunsplit()* で先に処理して、*scheme* と *netloc* を削除してください。

バージョン 3.5 で変更: **RFC 3986** で定義された意味論とマッチするように挙動がアップデートされました。

urllib.parse.**urldefrag**(*url*)

url がフラグメント識別子を含む場合、フラグメント識別子を持たないバージョンに修正された *url* と、別の文字列に分割されたフラグメント識別子を返します。*url* 中にフラグメント識別子がない場合、そのままの *url* と空文字列を返します。

戻り値は **名前付きタプル** で、インデックスによってもしくは名前属性として要素にアクセスできます:

属性	インデックス	値	指定されなかった場合の値
url	0	フラグメントのない URL	空文字列
fragment	1	フラグメント識別子	空文字列

結果オブジェクトのより詳しい情報は 構造化された解析結果 節を参照してください。

バージョン 3.2 で変更: 結果はシンプルな 2 要素のタプルから構造化オブジェクトに変更されました。

urllib.parse.unwrap(*url*)

Extract the url from a wrapped URL (that is, a string formatted as `<URL:scheme://host/path>`, `<scheme://host/path>`, `URL:scheme://host/path` or `scheme://host/path`). If *url* is not a wrapped URL, it is returned without changes.

21.8.2 URL parsing security

The *urlsplit()* and *urlparse()* APIs do not perform **validation** of inputs. They may not raise errors on inputs that other applications consider invalid. They may also succeed on some inputs that might not be considered URLs elsewhere. Their purpose is for practical functionality rather than purity.

Instead of raising an exception on unusual input, they may instead return some component parts as empty strings. Or components may contain more than perhaps they should.

We recommend that users of these APIs where the values may be used anywhere with security implications code defensively. Do some verification within your code before trusting a returned component part. Does that `scheme` make sense? Is that a sensible `path`? Is there anything strange about that `hostname`? etc.

21.8.3 ASCII エンコードバイト列の解析

URL を解析する関数は元々文字列のみ操作するよう設計されていました。実際のところ、それは URL が正しくクオートされエンコードされた ASCII バイト列を操作できた方が有用でした。結果的にこのモジュールの URL 解析関数はすべて *bytes* および *bytearray* オブジェクトに加えて *str* オブジェクトでも処理するようになりました。

str データが渡された場合、戻り値は *str* データのみを含んだものになります。*bytes* あるいは *bytearray* が渡された場合、戻り値は *bytes* データのみを含んだものになります。

単一の関数を呼び出す時に *bytes* または *bytearray* が混在した *str* を渡した場合、*TypeError* が、非 ASCII バイト値が渡された場合 *UnicodeDecodeError* が送出されます。

str と *bytes* 間で容易に変換を行えるよう、すべての URL 解析関数は `encode()` メソッド (結果に *str* データが含まれる時用) か `decode()` メソッド (結果に *bytes* データが含まれる時用) のどちらかを提供しています。これらメソッドの動作は対応する *str* と *bytes* メソッドが持つものと同じです (ただしデフォルトのエンコーディングは `'utf-8'` ではなく `'ascii'` になります)。それぞれは `encode()` メソッドを持つ *bytes* データか `decode()` メソッドを持つ *str* データのどちらかに対応した型を生成します。

非 ASCII データを含むなど、不適切にクオートされた URL を操作する可能性のあるアプリケーションでは、URL 解析メソッドを呼び出す前に独自にバイト列から文字列にデコードする必要があります。

この項で説明された挙動は URL 解析関数にのみ該当します。URL クオート関数でバイトシーケンスを生成もしくは消化する際には、別に URL クオート関数の項で詳説されている通りのルールに従います。

バージョン 3.2 で変更: URL 解析関数は ASCII エンコードバイトシーケンスも受け付けるようになりました

21.8.4 構造化された解析結果

urlparse()、*urlsplit()*、および *urldefrag()* 関数が返すオブジェクトは *tuple* 型のサブクラスになります。これらサブクラスにはそれぞれの関数で説明されている属性が追加されており、前述のとおりエンコーディングとデコーディングをサポートしています:

urllib.parse.SplitResult.geturl()

オリジナルの URL を再結合した場合は文字列で返されます。これはスキームが小文字に正規化されていたり、空の構成要素が除去されるなど、オリジナルの URL とは異なる場合があります。特に、空のパラメータ、クエリ、およびフラグメント識別子は削除されます。

urldefrag() の戻り値では、空のフラグメント識別子のみ削除されます。*urlsplit()* および *urlparse()* の戻り値では、このメソッドが返す URL には説明されているすべての変更が加えられます。

加えた解析関数を逆に行えばこのメソッドの戻り値は元の URL になります:

```
>>> from urllib.parse import urlsplit
>>> url = 'HTTP://www.Python.org/doc/#'
>>> r1 = urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
>>> r2 = urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

以下のクラスは *str* オブジェクトを操作した場合、構造化された解析結果の実装を提供します:

class urllib.parse.DefragResult(*url*, *fragment*)

urldefrag() の具象クラスの結果には *str* データが含まれます。encode() メソッドは *DefragResultBytes* インスタンスを返します。

バージョン 3.2 で追加.

class urllib.parse.ParseResult(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)

urlparse() の具象クラスの結果には *str* データが含まれます。encode() メソッドは *ParseResultBytes* インスタンスを返します。

class urllib.parse.SplitResult(*scheme*, *netloc*, *path*, *query*, *fragment*)

urlsplit() の具象クラスの結果には *str* データが含まれます。encode() メソッドは *SplitResultBytes* インスタンスを返します。

以下のクラスは *bytes* または *bytearray* オブジェクトを操作した時に解析結果の実装を提供します:

class urllib.parse.DefragResultBytes(*url*, *fragment*)

urldefrag() の具象クラスの結果には *bytes* データが含まれます。decode() メソッドは *DefragResult* インスタンスを返します。

バージョン 3.2 で追加.

class urllib.parse.ParseResultBytes(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)

urlparse() の具象クラスの結果には *bytes* が含まれます。decode() メソッドは *ParseResult* インスタンスを返します。

バージョン 3.2 で追加.

class urllib.parse.SplitResultBytes(*scheme*, *netloc*, *path*, *query*, *fragment*)

urlsplit() の具象クラスの結果には *bytes* データが含まれます。decode() メソッドは *SplitResult* インスタンスを返します。

バージョン 3.2 で追加.

21.8.5 URL のクオート

URL クオート関数は、プログラムデータを取り URL 構成要素として使用できるよう特殊文字をクオートしたり非 ASCII 文字を適切にエンコードすることに焦点を当てています。これらは上述の URL 解析関数でカバーされていない URL 構成要素からオリジナルデータの再作成もサポートしています。

urllib.parse.quote(*string*, *safe='/'*, *encoding=None*, *errors=None*)

Replace special characters in *string* using the %xx escape. Letters, digits, and the characters '_.-~' are never quoted. By default, this function is intended for quoting the path section of URL. The optional *safe* parameter specifies additional ASCII characters that should not be quoted --- its default value is '/'.

string に使用できるのは *str* か *bytes* です。

バージョン 3.7 で変更: Moved from **RFC 2396** to **RFC 3986** for quoting URL strings. "~" is now included in the set of unreserved characters.

任意のパラメータ *encoding* と *errors* は *str.encode()* で受け付けられる非 ASCII 文字への対処法を指定します。*encoding* のデフォルトは 'utf-8'、*errors* のデフォルトは 'strict' で、非サポート文字があると *UnicodeEncodeError* を送出します。*string* が *bytes* の場合 *encoding* と *errors* を指定してはいけません。指定すると *TypeError* が送出されます。

quote(string, safe, encoding, errors) は quote_from_bytes(string.encode(encoding, errors), safe) と等価であることに留意してください。

例: quote('/El Niño/') は '/El%20Ni%C3%B1o/' を返します。

urllib.parse.quote_plus(*string*, *safe=''*, *encoding=None*, *errors=None*)

quote() と似ていますが、クエリ文字列を URL に挿入する時のために HTML フォームの値の空白をプラス記号「+」に置き換えます。オリジナルの文字列に「+」が存在した場合は *safe* に指定されている場合を除きエスケープされます。*safe* にデフォルト値は設定されていません。

例: quote_plus('/El Niño/') は '%2FEl+Ni%C3%B1o%2F' を返します。

urllib.parse.**quote_from_bytes**(*bytes*, *safe='/'*)

quote() と似ていますが、*str* ではなく *bytes* オブジェクトを取り、文字列からバイト列へのエンコードを行いません。

例: `quote_from_bytes(b'a&\xef')` は `'a%26%EF'` を返します。

urllib.parse.**unquote**(*string*, *encoding='utf-8'*, *errors='replace'*)

エスケープされた `%xx` をそれに対応した単一文字に置き換えます。オプション引数の *encoding* と *errors* は *bytes.decode()* メソッドで受け付けられるパーセントエンコードされたシーケンスから Unicode 文字へのデコード法を指定します。

string は *str* でなければなりません。

encoding のデフォルトは `'utf-8'`、*errors* のデフォルトは `'replace'` で、不正なシーケンスはプレースホルダー文字に置き換えられます。

例: `unquote('/El%20Ni%C3%B1o/')` は `'/El Niño/'` を返します。

urllib.parse.**unquote_plus**(*string*, *encoding='utf-8'*, *errors='replace'*)

unquote() と似ていますが、HTML フォームの値のアンクオートのために「+」を空白に置き換えます。

string は *str* でなければなりません。

例: `unquote_plus('/El+Ni%C3%B1o/')` は `'/El Niño/'` を返します。

urllib.parse.**unquote_to_bytes**(*string*)

スケープされた `%xx` をそれに対応した 1 オクテットに置き換え、*bytes* オブジェクトを返します。

string に使用できるのは *str* か *bytes* です。

str だった場合、*string* 内のエスケープされていない非 ASCII 文字は UTF-8 バイト列にエンコードされます。

例: `unquote_to_bytes('a%26%EF')` は `b'a&\xef'` を返します。

urllib.parse.**urlencode**(*query*, *doseq=False*, *safe=''*, *encoding=None*, *errors=None*, *quote_via=quote_plus*)

マッピング型オブジェクトまたは 2 個の要素からなるタプルのシーケンス (*str* か *bytes* オブジェクトが含まれているかもしれません) を、パーセントエンコードされた ASCII 文字列に変換します。戻り値の文字列が *urlopen()* 関数での POST 操作の *data* で使用される場合はバイト列にエンコードしなければなりません。そうでない場合は *TypeError* が送出されます。

戻り値は `'&'` 文字で区切られた `key=value` のペアからなる一組の文字列になります。*key* と *value* は *quote_via* を使用してクオートされます。デフォルトで、値をクォートするために *quote_plus()* が使用されます。つまり、スペースは `'+'` 文字に、'/' 文字は `%2F` にクォートされます。これは GET リクエストの標準に準拠します (`application/x-www-form-urlencoded`)。*quote_via* として渡すことができる別の関数は *quote()* です。それはスペースを `%20` にエンコードし、'/' をエンコードしません。何がクォートされるかを最大限コントロールしたければ、`quote` を使って *safe* に値を指定してください。

引数 *query* が 2 要素のタプルのシーケンスの場合、各タプルの第一要素はキーに、第二要素は値になります。値となる要素はシーケンスを取ることもでき、この場合、オプションのパラメーター *doseq* が `True` と評価されるのであれば、キーに対し値シーケンスの各要素を個別に結び付けた `key=value` のペアを、`'&'` 文字でつないだものを生成します。エンコードされた文字列内のパラメーターの順序はシーケンス内のパラメータータプルの順序と一致します。

safe, *encoding*, および *errors* パラメータは *quote_via* にそのまま渡されます (クエリ要素が *str* の場合は、*encoding* と *errors* パラメータだけが渡されます)。

このエンコード処理の逆を行うには、このモジュールで提供されている *parse_qs()* と *parse_qsl()* を使用して、クエリ文字列を Python データ構造に変換できます。

POST データ、あるいは URL クエリ文字列を生成するために、urlencode メソッドをどのように使えばよいかを見るには、*urllib* **の使用例** を参照してください。

バージョン 3.2 で変更: クエリ文字列にバイト列と文字列オブジェクトをサポートしました。

バージョン 3.5 で追加: *quote_via* 引数。

参考:

WHATWG - URL Living standard　Working Group for the URL Standard that defines URLs, domains, IP addresses, the application/x-www-form-urlencoded format, and their API.

RFC 3986 - Uniform Resource Identifiers　これが現在の標準規格 (STD66) です。urllib.parse モジュールに対するすべての変更はこの規格に準拠していなければなりませんが、若干の逸脱はありえます。これは主には後方互換性のため、また主要なブラウザで一般的に見られる、URL を解析する上でのいくつかの事実上の要件を満たすためです。

RFC 2732 - Format for Literal IPv6 Addresses in URL's.　この規格は IPv6 の URL を解析するときの要求事項を記述しています。

RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax　この RFC では Uniform Resource Name (URN) と Uniform Resource Locator (URL) の両方に対する一般的な文法的要求事項を記述しています。

RFC 2368 - The mailto URL scheme.　mailto URL スキームに対する文法的要求事項です。

RFC 1808 - Relative Uniform Resource Locators　この RFC には絶対 URL と相対 URL を結合するための規則がボーダケースの取扱い方を決定する " 異常な例" つきで収められています。

RFC 1738 - Uniform Resource Locators (URL)　この RFC では絶対 URL の形式的な文法と意味付けを仕様化しています。

21.9 urllib.error --- urllib.request が投げる例外

ソースコード: Lib/urllib/error.py

---

urllib.error は *urllib.request* によって投げられる例外を定義しています。基底クラスは *URLError* です。

urllib.error は必要に応じて以下の例外が送出します:

exception urllib.error.URLError

ハンドラが何らかの問題に遭遇した場合、この例外 (またはこの例外から派生した例外) を送出します。この例外は *OSError* のサブクラスです。

reason

このエラーの理由。メッセージ文字列あるいは他の例外インスタンスです。

バージョン 3.3 で変更: *URLError* は *IOError* の代わりに *OSError* のサブクラスになりました。

exception urllib.error.HTTPError

HTTPError は例外 (*URLError* のサブクラス) ですが、同時に例外ではない file-like な戻り値を返す関数でもあります (*urlopen()* の戻り値と同じです)。これは、例えばサーバからの認証リクエストのように、変わった HTTP エラーを処理するのに役立ちます。

code

RFC 2616 に定義されている HTTP ステータスコード。この数値型の値は、*http.server.BaseHTTPRequestHandler.responses* の辞書に登録されているコードに対応します。

reason

これは通常、このエラーの原因を説明する文字列です。

headers

HTTPError の原因となった HTTP リクエストの HTTP レスポンスヘッダ。

バージョン 3.4 で追加.

exception urllib.error.ContentTooShortError(*msg*, *content*)

この例外は *urlretrieve()* 関数が、ダウンロードされたデータの量が予期した量 (*Content-Length* ヘッダで与えられる) よりも少ないことを検知した際に発生します。content 属性には (恐らく途中までの) ダウンロードされたデータが格納されています。

21.10 urllib.robotparser --- robots.txt のためのパーザ

ソースコード: Lib/urllib/robotparser.py

このモジュールでは単一のクラス、*RobotFileParser* を提供します。このクラスは、特定のユーザエージェントが robots.txt ファイルを公開している Web サイトのある URL を取得可能かどうかの質問に答えます。robots.txt ファイルの構造に関する詳細は http://www.robotstxt.org/orig.html を参照してください。

class urllib.robotparser.**RobotFileParser**(*url=''*)

url の robots.txt に対し読み込み、パーズ、応答するメソッドを提供します。

set_url(*url*)

robots.txt ファイルを参照するための URL を設定します。

read()

robots.txt URL を読み出し、パーザに入力します。

parse(*lines*)

引数 *lines* の内容を解釈します。

can_fetch(*useragent*, *url*)

解釈された robots.txt ファイル中に記載された規則に従ったとき、*useragent* が *url* を取得してもよい場合には True を返します。

mtime()

robots.txt ファイルを最後に取得した時刻を返します。この値は、定期的に新たな robots.txt をチェックする必要がある、長時間動作する Web スパイダープログラムを実装する際に便利です。

modified()

robots.txt ファイルを最後に取得した時刻を現在の時刻に設定します。

crawl_delay(*useragent*)

Returns the value of the Crawl-delay parameter from robots.txt for the *useragent* in question. If there is no such parameter or it doesn't apply to the *useragent* specified or the robots.txt entry for this parameter has invalid syntax, return None.

バージョン 3.6 で追加.

request_rate(*useragent*)

Returns the contents of the Request-rate parameter from robots.txt as a *named tuple* RequestRate(requests, seconds). If there is no such parameter or it doesn't apply to the *useragent* specified or the robots.txt entry for this parameter has invalid syntax, return None.

バージョン 3.6 で追加.

site_maps()

Returns the contents of the Sitemap parameter from robots.txt in the form of a *list()*.

If there is no such parameter or the `robots.txt` entry for this parameter has invalid syntax, return `None`.

バージョン 3.8 で追加.

以下に *RobotFileParser* クラスの利用例を示します。

```
>>> import urllib.robotparser
>>> rp = urllib.robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rrate = rp.request_rate("*")
>>> rrate.requests
3
>>> rrate.seconds
20
>>> rp.crawl_delay("*")
6
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```

21.11 http --- HTTP モジュール群

ソースコード: Lib/http/__init__.py

---

http パッケージはハイパーテキスト転送プロトコルを扱うための幾つかのモジュールを集めたものです:

- *http.client* は低水準の HTTP プロトコルのクライアントです。高水準の URL を開く操作には *urllib.request* を使ってください
- *http.server* は *socketserver* をベースにした基礎的な HTTP サーバーを実装しています
- *http.cookies* は cookie の状態管理を実装するためのユーティリティーを提供しています
- *http.cookiejar* は cookie の永続化機能を提供しています

http は *http.HTTPStatus* 列挙子で多くの HTTP ステータスコードと関連するメッセージを定義しているモジュールでもあります。

class http.HTTPStatus

バージョン 3.5 で追加.

一連の HTTP ステータスコード、理由の表現、英語で書かれた長い記述を定義した *enum.IntEnum* のサブクラスです。

使い方:

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK
<HTTPStatus.OK: 200>
>>> HTTPStatus.OK == 200
True
>>> HTTPStatus.OK.value
200
>>> HTTPStatus.OK.phrase
'OK'
>>> HTTPStatus.OK.description
'Request fulfilled, document follows'
>>> list(HTTPStatus)
[<HTTPStatus.CONTINUE: 100>, <HTTPStatus.SWITCHING_PROTOCOLS: 101>, ...]
```

21.11.1 HTTP ステータスコード

サポートされている IANA に登録された *http.HTTPStatus* で利用可能なステータスコード:

コード	列挙名	詳細
100	CONTINUE	HTTP/1.1 **RFC 7231**, Section 6.2.1
101	SWITCHING_PROTOCOLS	HTTP/1.1 **RFC 7231**, Section 6.2.2
102	PROCESSING	WebDAV **RFC 2518**, Section 10.1
200	OK	HTTP/1.1 **RFC 7231**, Section 6.3.1
201	CREATED	HTTP/1.1 **RFC 7231**, Section 6.3.2
202	ACCEPTED	HTTP/1.1 **RFC 7231**, Section 6.3.3
203	NON_AUTHORITATIVE_INFORMATION	HTTP/1.1 **RFC 7231**, Section 6.3.4
204	NO_CONTENT	HTTP/1.1 **RFC 7231**, Section 6.3.5
205	RESET_CONTENT	HTTP/1.1 **RFC 7231**, Section 6.3.6
206	PARTIAL_CONTENT	HTTP/1.1 **RFC 7233**, Section 4.1
207	MULTI_STATUS	WebDAV **RFC 4918**, Section 11.1
208	ALREADY_REPORTED	WebDAV Binding Extensions **RFC 5842**, Section 7.1 (Experimental
226	IM_USED	Delta Encoding in HTTP **RFC 3229**, Section 10.4.1
300	MULTIPLE_CHOICES	HTTP/1.1 **RFC 7231**, Section 6.4.1
301	MOVED_PERMANENTLY	HTTP/1.1 **RFC 7231**, Section 6.4.2
302	FOUND	HTTP/1.1 **RFC 7231**, Section 6.4.3
303	SEE_OTHER	HTTP/1.1 **RFC 7231**, Section 6.4.4
304	NOT_MODIFIED	HTTP/1.1 **RFC 7232**, Section 4.1
305	USE_PROXY	HTTP/1.1 **RFC 7231**, Section 6.4.5
307	TEMPORARY_REDIRECT	HTTP/1.1 **RFC 7231**, Section 6.4.7
308	PERMANENT_REDIRECT	Permanent Redirect **RFC 7238**, Section 3 (Experimental)
400	BAD_REQUEST	HTTP/1.1 **RFC 7231**, Section 6.5.1
401	UNAUTHORIZED	HTTP/1.1 Authentication **RFC 7235**, Section 3.1

次

表 1 – 前のページからの続き

コード	列挙名	詳細
402	PAYMENT_REQUIRED	HTTP/1.1 **RFC 7231**, Section 6.5.2
403	FORBIDDEN	HTTP/1.1 **RFC 7231**, Section 6.5.3
404	NOT_FOUND	HTTP/1.1 **RFC 7231**, Section 6.5.4
405	METHOD_NOT_ALLOWED	HTTP/1.1 **RFC 7231**, Section 6.5.5
406	NOT_ACCEPTABLE	HTTP/1.1 **RFC 7231**, Section 6.5.6
407	PROXY_AUTHENTICATION_REQUIRED	HTTP/1.1 Authentication **RFC 7235**, Section 3.2
408	REQUEST_TIMEOUT	HTTP/1.1 **RFC 7231**, Section 6.5.7
409	CONFLICT	HTTP/1.1 **RFC 7231**, Section 6.5.8
410	GONE	HTTP/1.1 **RFC 7231**, Section 6.5.9
411	LENGTH_REQUIRED	HTTP/1.1 **RFC 7231**, Section 6.5.10
412	PRECONDITION_FAILED	HTTP/1.1 **RFC 7232**, Section 4.2
413	REQUEST_ENTITY_TOO_LARGE	HTTP/1.1 **RFC 7231**, Section 6.5.11
414	REQUEST_URI_TOO_LONG	HTTP/1.1 **RFC 7231**, Section 6.5.12
415	UNSUPPORTED_MEDIA_TYPE	HTTP/1.1 **RFC 7231**, Section 6.5.13
416	REQUESTED_RANGE_NOT_SATISFIABLE	HTTP/1.1 Range Requests **RFC 7233**, Section 4.4
417	EXPECTATION_FAILED	HTTP/1.1 **RFC 7231**, Section 6.5.14
421	MISDIRECTED_REQUEST	HTTP/2 **RFC 7540**, Section 9.1.2
422	UNPROCESSABLE_ENTITY	WebDAV **RFC 4918**, Section 11.2
423	LOCKED	WebDAV **RFC 4918**, Section 11.3
424	FAILED_DEPENDENCY	WebDAV **RFC 4918**, Section 11.4
426	UPGRADE_REQUIRED	HTTP/1.1 **RFC 7231**, Section 6.5.15
428	PRECONDITION_REQUIRED	Additional HTTP Status Codes **RFC 6585**
429	TOO_MANY_REQUESTS	Additional HTTP Status Codes **RFC 6585**
431	REQUEST_HEADER_FIELDS_TOO_LARGE	Additional HTTP Status Codes **RFC 6585**
451	UNAVAILABLE_FOR_LEGAL_REASONS	An HTTP Status Code to Report Legal Obstacles **RFC 7725**
500	INTERNAL_SERVER_ERROR	HTTP/1.1 **RFC 7231**, Section 6.6.1
501	NOT_IMPLEMENTED	HTTP/1.1 **RFC 7231**, Section 6.6.2
502	BAD_GATEWAY	HTTP/1.1 **RFC 7231**, Section 6.6.3
503	SERVICE_UNAVAILABLE	HTTP/1.1 **RFC 7231**, Section 6.6.4
504	GATEWAY_TIMEOUT	HTTP/1.1 **RFC 7231**, Section 6.6.5
505	HTTP_VERSION_NOT_SUPPORTED	HTTP/1.1 **RFC 7231**, Section 6.6.6
506	VARIANT_ALSO_NEGOTIATES	Transparent Content Negotiation in HTTP **RFC 2295**, Section 8.1 (
507	INSUFFICIENT_STORAGE	WebDAV **RFC 4918**, Section 11.5
508	LOOP_DETECTED	WebDAV Binding Extensions **RFC 5842**, Section 7.2 (Experimental
510	NOT_EXTENDED	An HTTP Extension Framework **RFC 2774**, Section 7 (Experimenta
511	NETWORK_AUTHENTICATION_REQUIRED	Additional HTTP Status Codes **RFC 6585**, Section 6

後方互換性を保つために列挙値は *http.client* にも定数という形で存在します。列挙名は定数名と同じです (すなわち `http.HTTPStatus.OK` は `http.client.OK` としても利用可能です)。

バージョン 3.7 で変更: ステータスコード `421 MISDIRECTED_REQUEST` が追加されました。

バージョン 3.8 で追加: ステータスコード 451 UNAVAILABLE_FOR_LEGAL_REASONS が追加されました。

21.12 http.client --- HTTP プロトコルクライアント

ソースコード: Lib/http/client.py

このモジュールでは HTTP および HTTPS プロトコルのクライアントサイドを実装しているクラスを定義しています。通常、このモジュールは直接使いません --- *urllib.request* モジュールが HTTP や HTTPS を使った URL を扱う上でこのモジュールを使います。

参考:

より高水準の HTTP クライアントインターフェースとして *Requests package <https://requests.readthedocs.io/en/master/>* がお奨めです。

注釈: HTTPS のサポートは、Python が SSL サポート付きでコンパイルされている場合にのみ利用できます (*ssl* モジュールによって)。

このモジュールでは以下のクラスを提供しています:

class http.client.**HTTPConnection**(*host*, *port=None*[, *timeout*], *source_address=None*, *blocksize=8192*)

HTTPConnection インスタンスは、HTTP サーバとの一回のトランザクションを表現します。インスタンスの生成はホスト名とオプションのポート番号を与えて行います。ポート番号を指定しなかった場合、ホスト名文字列が `host:port` の形式であれば、ホスト名からポート番号を抽出し、そうでない場合には標準の HTTP ポート番号 (80) を使います。オプションの引数 *timeout* が渡された場合、ブロックする処理 (コネクション接続など) のタイムアウト時間 (秒数) として利用されます (渡されなかった場合は、グローバルのデフォルトタイムアウト設定が利用されます)。オプションの引数 *source_address* を (host, port) という形式のタプルにすると HTTP 接続の接続元アドレスとして使用します。オプションの *blocksize* 引数は、送信するファイル類メッセージボディのバッファサイズをバイト単位で設定します。

例えば、以下の呼び出しは全て同じサーバの同じポートに接続するインスタンスを生成します:

```
>>> h1 = http.client.HTTPConnection('www.python.org')
>>> h2 = http.client.HTTPConnection('www.python.org:80')
>>> h3 = http.client.HTTPConnection('www.python.org', 80)
>>> h4 = http.client.HTTPConnection('www.python.org', 80, timeout=10)
```

バージョン 3.2 で変更: *source_address* が追加されました。

バージョン 3.4 で変更: *strict* パラメータは廃止されました。HTTP 0.9 の ” シンプルなレスポンス” のような形式はもはやサポートされません。

バージョン 3.7 で変更: *blocksize* 引数が追加されました。

class http.client.HTTPSConnection(*host*, *port=None*, *key_file=None*, *cert_file=None*[, *timeout*], *source_address=None*, *, *context=None*, *check_hostname=None*, *blocksize=8192*)

HTTPConnection のサブクラスはセキュア・サーバとやりとりする為の SSL を使う場合に用います。デフォルトのポート番号は 443 です。*context* が指定されれば、それは様々な SSL オプションを記述する *ssl.SSLContext* インスタンスでなければなりません。

ベストプラクティスに関するより良い情報が **セキュリティで考慮すべき点** にありますのでお読みください。

バージョン 3.2 で変更: *source_address*、*context* そして *check_hostname* が追加されました。

バージョン 3.2 で変更: このクラスは現在、可能であれば (つまり *ssl.HAS_SNI* が真の場合) HTTPS のバーチャルホストをサポートしています。

バージョン 3.4 で変更: *strict* パラメータは廃止されました。HTTP 0.9 の " シンプルなレスポンス" のような形式はもはやサポートされません。

バージョン 3.4.3 で変更: このクラスは今や全ての必要な証明書とホスト名の検証をデフォルトで行うようになりました。昔の、検証を行わない振る舞いに戻したければ、*context* に `ssl._create_unverified_context()` を渡すことで出来ます。

バージョン 3.8 で変更: This class now enables TLS 1.3 *ssl.SSLContext.post_handshake_auth* for the default *context* or when *cert_file* is passed with a custom *context*.

バージョン 3.6 で非推奨: *key_file* および *cert_file* は非推奨となったので、*context* を使ってください。代わりに *ssl.SSLContext.load_cert_chain()* を使うか、または *ssl.create_default_context()* にシステムが信頼する CA 証明書を選んでもらうかしてください。

The *check_hostname* parameter is also deprecated; the *ssl.SSLContext.check_hostname* attribute of *context* should be used instead.

class http.client.HTTPResponse(*sock*, *debuglevel=0*, *method=None*, *url=None*)

コネクションに成功したときに、このクラスのインスタンスが返されます。ユーザーから直接利用されることはありません。

バージョン 3.4 で変更: *strict* パラメータは廃止されました。HTTP 0.9 の " シンプルなレスポンス" のような形式はもはやサポートされません。

このモジュールは以下の関数を提供します:

http.client.parse_headers(*fp*)

Parse the headers from a file pointer *fp* representing a HTTP request/response. The file has to be a `BufferedIOBase` reader (i.e. not text) and must provide a valid **RFC 2822** style header.

This function returns an instance of `http.client.HTTPMessage` that holds the header fields, but no payload (the same as *HTTPResponse.msg* and *http.server.BaseHTTPRequestHandler.headers*). After returning, the file pointer *fp* is ready to read the HTTP body.

注釈: *parse_headers()* does not parse the start-line of a HTTP message; it only parses the `Name:`

value lines. The file has to be ready to read these field lines, so the first line should already be consumed before calling the function.

状況に応じて、以下の例外が送出されます:

exception http.client.HTTPException
このモジュールにおける他の例外クラスの基底クラスです。*Exception* のサブクラスです。

exception http.client.NotConnected
HTTPException サブクラスです。

exception http.client.InvalidURL
HTTPException のサブクラスです。ポート番号を指定したものの、その値が数字でなかったり空のオブジェクトであった場合に送出されます。

exception http.client.UnknownProtocol
HTTPException サブクラスです。

exception http.client.UnknownTransferEncoding
HTTPException サブクラスです。

exception http.client.UnimplementedFileMode
HTTPException サブクラスです。

exception http.client.IncompleteRead
HTTPException サブクラスです。

exception http.client.ImproperConnectionState
HTTPException サブクラスです。

exception http.client.CannotSendRequest
ImproperConnectionState のサブクラスです。

exception http.client.CannotSendHeader
ImproperConnectionState のサブクラスです。

exception http.client.ResponseNotReady
ImproperConnectionState のサブクラスです。

exception http.client.BadStatusLine
HTTPException のサブクラスです。サーバが理解できない HTTP 状態コードで応答した場合に送出されます。

exception http.client.LineTooLong
A subclass of *HTTPException*. Raised if an excessively long line is received in the HTTP protocol from the server.

exception http.client.RemoteDisconnected
A subclass of *ConnectionResetError* and *BadStatusLine*. Raised by *HTTPConnection.*

getresponse() when the attempt to read the response results in no data read from the connection, indicating that the remote end has closed the connection.

バージョン 3.5 で追加: Previously, *BadStatusLine*('') was raised.

このモジュールで定義されている定数は以下の通りです:

http.client.HTTP_PORT
HTTP プロトコルの標準のポート (通常は 80) です。

http.client.HTTPS_PORT
HTTPS プロトコルの標準のポート (通常は 443) です。

http.client.responses
このディクショナリは、HTTP 1.1 ステータスコードを W3C の名前にマップしたものです。

例: `http.client.responses[http.client.NOT_FOUND]` は `'Not Found'` を示します。

See *HTTP ステータスコード* for a list of HTTP status codes that are available in this module as constants.

21.12.1 HTTPConnection オブジェクト

HTTPConnection インスタンスには以下のメソッドがあります:

HTTPConnection.request(*method*, *url*, *body=None*, *headers={}*, ***, *encode_chunked=False*)
このメソッドは、HTTP 要求メソッド *method* およびセレクタ *url* を使って、要求をサーバに送ります。

If *body* is specified, the specified data is sent after the headers are finished. It may be a *str*, a *bytes-like object*, an open *file object*, or an iterable of *bytes*. If *body* is a string, it is encoded as ISO-8859-1, the default for HTTP. If it is a bytes-like object, the bytes are sent as is. If it is a *file object*, the contents of the file is sent; this file object should support at least the `read()` method. If the file object is an instance of *io.TextIOBase*, the data returned by the `read()` method will be encoded as ISO-8859-1, otherwise the data returned by `read()` is sent as is. If *body* is an iterable, the elements of the iterable are sent as is until the iterable is exhausted.

headers 引数は要求と同時に送信される拡張 HTTP ヘッダの内容からなるマップ型でなくてはなりません。

If *headers* contains neither Content-Length nor Transfer-Encoding, but there is a request body, one of those header fields will be added automatically. If *body* is `None`, the Content-Length header is set to `0` for methods that expect a body (`PUT`, `POST`, and `PATCH`). If *body* is a string or a bytes-like object that is not also a *file*, the Content-Length header is set to its length. Any other type of *body* (files and iterables in general) will be chunk-encoded, and the Transfer-Encoding header will automatically be set instead of Content-Length.

The *encode_chunked* argument is only relevant if Transfer-Encoding is specified in *headers*. If *encode_chunked* is `False`, the HTTPConnection object assumes that all encoding is handled by the calling code. If it is `True`, the body will be chunk-encoded.

注釈: Chunked transfer encoding has been added to the HTTP protocol version 1.1. Unless the HTTP server is known to handle HTTP 1.1, the caller must either specify the Content-Length, or must pass a *str* or bytes-like object that is not also a file as the body representation.

バージョン 3.2 で追加: *body* は iterable オブジェクトとして使用できます。

バージョン 3.6 で変更: If neither Content-Length nor Transfer-Encoding are set in *headers*, file and iterable *body* objects are now chunk-encoded. The *encode_chunked* argument was added. No attempt is made to determine the Content-Length for file objects.

HTTPConnection.**getresponse**()

サーバに対して HTTP 要求を送り出した後に呼び出されなければなりません。要求に対する応答を取得します。*HTTPResponse* インスタンスを返します。

注釈: すべての応答を読み込んでからでなければ新しい要求をサーバに送ることはできないことに注意しましょう。

バージョン 3.5 で変更: If a *ConnectionError* or subclass is raised, the *HTTPConnection* object will be ready to reconnect when a new request is sent.

HTTPConnection.**set_debuglevel**(*level*)

Set the debugging level. The default debug level is `0`, meaning no debugging output is printed. Any value greater than `0` will cause all currently defined debug output to be printed to stdout. The `debuglevel` is passed to any new *HTTPResponse* objects that are created.

バージョン 3.1 で追加.

HTTPConnection.**set_tunnel**(*host*, *port=None*, *headers=None*)

HTTP トンネリング接続のホスト名とポート番号を設定します。これによりプロキシサーバを通しての接続を実行できます。

The host and port arguments specify the endpoint of the tunneled connection (i.e. the address included in the CONNECT request, *not* the address of the proxy server).

ヘッダのパラメータは CONNECT リクエストで送信するために他の HTTP ヘッダにマッピングされます。

For example, to tunnel through a HTTPS proxy server running locally on port 8080, we would pass the address of the proxy to the *HTTPSConnection* constructor, and the address of the host that we eventually want to reach to the *set_tunnel()* method:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("localhost", 8080)
>>> conn.set_tunnel("www.python.org")
>>> conn.request("HEAD","/index.html")
```

バージョン 3.2 で追加.

HTTPConnection.connect()
Connect to the server specified when the object was created. By default, this is called automatically when making a request if the client does not already have a connection.

HTTPConnection.close()
サーバへの接続を閉じます。

HTTPConnection.blocksize
Buffer size in bytes for sending a file-like message body.

バージョン 3.7 で追加.

上で説明した request() メソッドを使うかわりに、以下の 4 つの関数を使用して要求をステップバイステップで送信することもできます。

HTTPConnection.putrequest(*method*, *url*, *skip_host=False*, *skip_accept_encoding=False*)
サーバへの接続が確立したら、最初にこのメソッドを呼び出さなくてはなりません。このメソッドは *method* 文字列、*url* 文字列、そして HTTP バージョン (HTTP/1.1) からなる一行を送信します。Host: や Accept-Encoding: ヘッダの自動送信を無効にしたい場合 (例えば別のコンテンツエンコーディングを受け入れたい場合) には、*skip_host* や *skip_accept_encoding* を偽でない値に設定してください。

HTTPConnection.putheader(*header*, *argument*[, ...])
RFC 822 形式のヘッダをサーバに送ります。この処理では、*header* 、コロンとスペース、そして最初の引数からなる 1 行をサーバに送ります。追加の引数を指定した場合、継続して各行にタブ一つと引数の入った引数行が送信されます。

HTTPConnection.endheaders(*message_body=None*, ***, *encode_chunked=False*)
サーバに空行を送り、ヘッダ部が終了したことを通知します。オプションの *message_body* 引数を、リクエストに関連したメッセージボディを渡すのに使うことが出来ます。

If *encode_chunked* is True, the result of each iteration of *message_body* will be chunk-encoded as specified in **RFC 7230**, Section 3.3.1. How the data is encoded is dependent on the type of *message_body*. If *message_body* implements the buffer interface the encoding will result in a single chunk. If *message_body* is a *collections.abc.Iterable*, each iteration of *message_body* will result in a chunk. If *message_body* is a *file object*, each call to .read() will result in a chunk. The method automatically signals the end of the chunk-encoded data immediately after *message_body*.

注釈: Due to the chunked encoding specification, empty chunks yielded by an iterator body will be ignored by the chunk-encoder. This is to avoid premature termination of the read of the request by the target server due to malformed encoding.

バージョン 3.6 で追加: Chunked encoding support. The *encode_chunked* parameter was added.

HTTPConnection.**send**(*data*)
サーバにデータを送ります。このメソッドは *endheaders()* が呼び出された直後で、かつ *getresponse()* が呼び出される前に使わなければなりません。

21.12.2 HTTPResponse オブジェクト

HTTPResponse インスタンスはサーバからの HTTP レスポンスをラップします。これを使用してリクエストヘッダとエンティティボディへアクセスすることができます。レスポンスはイテレート可能なオブジェクトであるので、with 文と使うことが可能です。

バージョン 3.5 で変更: The *io.BufferedIOBase* interface is now implemented and all of its reader operations are supported.

HTTPResponse.**read**([*amt*])
応答の本体全体か、*amt* バイトまで読み出して返します。

HTTPResponse.**readinto**(*b*)
バッファ *b* にレスポンスボディの次のデータを最大 len(b) バイト読み込みます。何バイト読んだかを返します。

バージョン 3.3 で追加.

HTTPResponse.**getheader**(*name*, *default=None*)
Return the value of the header *name*, or *default* if there is no header matching *name*. If there is more than one header with the name *name*, return all of the values joined by ', '. If 'default' is any iterable other than a single string, its elements are similarly returned joined by commas.

HTTPResponse.**getheaders**()
(header, value) のタプルからなるリストを返します。

HTTPResponse.**fileno**()
ソケットの `fileno` を返します。

HTTPResponse.**msg**
A `http.client.HTTPMessage` instance containing the response headers. `http.client.HTTPMessage` is a subclass of *email.message.Message*.

HTTPResponse.**version**
サーバが使用した HTTP プロトコルバージョンです。10 は HTTP/1.0 を、11 は HTTP/1.1 を表します。

HTTPResponse.**status**
サーバから返される状態コードです。

HTTPResponse.**reason**
サーバから返される応答の理由文です。

HTTPResponse.**debuglevel**
A debugging hook. If *debuglevel* is greater than zero, messages will be printed to stdout as the

response is read and parsed.

HTTPResponse.closed
ストリームが閉じている場合 True となります。

21.12.3 使用例

以下は GET リクエストの送信方法を示した例です:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print(r1.status, r1.reason)
200 OK
>>> data1 = r1.read()  # This will return entire content.
>>> # The following example demonstrates reading data in chunks.
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> while chunk := r1.read(200):
...     print(repr(chunk))
b'<!doctype html>\n<!--[if"...
...
>>> # Example of an invalid request
>>> conn = http.client.HTTPSConnection("docs.python.org")
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
>>> print(r2.status, r2.reason)
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

次の例のセッションでは、HEAD メソッドを利用しています。HEAD メソッドは全くデータを返さないことに注目してください。

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("HEAD", "/")
>>> res = conn.getresponse()
>>> print(res.status, res.reason)
200 OK
>>> data = res.read()
>>> print(len(data))
0
>>> data == b''
True
```

以下は POST リクエストの送信方法を示した例です:

```
>>> import http.client, urllib.parse
>>> params = urllib.parse.urlencode({'@number': 12524, '@type': 'issue', '@action': 'show'})
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...            "Accept": "text/plain"}
>>> conn = http.client.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
302 Found
>>> data = response.read()
>>> data
b'Redirecting to <a href="http://bugs.python.org/issue12524">http://bugs.python.org/issue12524
↪</a>'
>>> conn.close()
```

Client side HTTP PUT requests are very similar to POST requests. The difference lies only the server side where HTTP server will allow resources to be created via PUT request. It should be noted that custom HTTP methods are also handled in *urllib.request.Request* by setting the appropriate method attribute. Here is an example session that shows how to send a PUT request using http.client:

```
>>> # This creates an HTTP message
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/file
...
>>> import http.client
>>> BODY = "***filecontents***"
>>> conn = http.client.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200, OK
```

21.12.4 HTTPMessage オブジェクト

http.client.HTTPMessage のインスタンスは HTTP レスポンスヘッダを格納します。*email.message.Message* クラスを利用して実装されています。

21.13 ftplib --- FTP プロトコルクライアント

ソースコード: Lib/ftplib.py

このモジュールでは *FTP* クラスと、それに関連するいくつかの項目を定義しています。*FTP* クラスは、FTP プロトコルのクライアント側の機能を備えています。このクラスを使うと FTP のいろいろな機能の自動化、例えば他の FTP サーバのミラーリングといったことを実行する Python プログラムを書くことができます。また、*urllib.request* モジュールも FTP を使う URL を操作するのにこのクラスを使っています。FTP (File Transfer Protocol) についての詳しい情報は Internet **RFC 959** を参照して下さい。

ftplib モジュールを使ったサンプルを以下に示します:

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.debian.org')     # connect to host, default port
>>> ftp.login()                     # user anonymous, passwd anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')               # change into "debian" directory
>>> ftp.retrlines('LIST')           # list directory contents
-rw-rw-r--    1 1176     1176         1063 Jun 15 10:18 README
...
drwxr-sr-x    5 1176     1176         4096 Dec 19  2000 pool
drwxr-sr-x    4 1176     1176         4096 Nov 17  2008 project
drwxr-xr-x    3 1176     1176         4096 Oct 10  2012 tools
'226 Directory send OK.'
>>> with open('README', 'wb') as fp:
>>>     ftp.retrbinary('RETR README', fp.write)
'226 Transfer complete.'
>>> ftp.quit()
```

このモジュールは以下の項目を定義しています:

class ftplib.**FTP**(*host=''*, *user=''*, *passwd=''*, *acct=''*, *timeout=None*, *source_address=None*)

Return a new instance of the *FTP* class. When *host* is given, the method call `connect(host)` is made. When *user* is given, additionally the method call `login(user, passwd, acct)` is made (where *passwd* and *acct* default to the empty string when not given). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if is not specified, the global default timeout setting will be used). *source_address* is a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.

FTP クラスは `with` 文をサポートしています。例えば:

```
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
...     ftp.login()
...     ftp.dir()
...
'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 .
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 ..
dr-xr-xr-x   5 ftp      ftp          4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp      ftp            18 Jul 10  2008 Fedora
>>>
```

バージョン 3.2 で変更: `with` 構文のサポートが追加されました。

バージョン 3.3 で変更: *source_address* 引数が追加されました。

class ftplib.**FTP_TLS**(*host=''*, *user=''*, *passwd=''*, *acct=''*, *keyfile=None*, *certfile=None*, *context=None*, *timeout=None*, *source_address=None*)

RFC 4217 に記述されている TLS サポートを FTP に加えた *FTP* のサブクラスです。認証の前に FTP コントロール接続を暗黙にセキュアにし、通常通りに port 21 に接続します。データ接続をセキュアにするには、ユーザが *prot_p()* メソッドを呼び出してそれを明示的に要求しなければなりま

せん。*context* は SSL 設定オプション、証明書、秘密鍵を一つの (潜在的に長生きの) 構造にまとめた *ssl.SSLContext* オブジェクトです。ベストプラクティスについての **セキュリティで考慮すべき点** をお読みください。

keyfile と *certfile* は *context* のレガシー版です -- これらは、SSL 接続のための、PEM フォーマットの秘密鍵と証明書チェーンファイル名 (前者が *keyfile* 、後者が *certfile*) を含むことができます。

バージョン 3.2 で追加.

バージョン 3.3 で変更: *source_address* 引数が追加されました。

バージョン 3.4 で変更: このクラスは *ssl.SSLContext.check_hostname* と *Server Name Indication* でホスト名のチェックをサポートしました。(*ssl.HAS_SNI* を参照してください)。

バージョン 3.6 で非推奨: *keyfile* および *certfile* は非推奨となったので、*context* を使ってください。代わりに *ssl.SSLContext.load_cert_chain()* を使うか、または *ssl.create_default_context()* にシステムが信頼する CA 証明書を選んでもらうかしてください。

FTP_TLS クラスを使ったサンプルセッションはこちらです:

```
>>> ftps = FTP_TLS('ftp.pureftpd.org')
>>> ftps.login()
'230 Anonymous user logged in'
>>> ftps.prot_p()
'200 Data protection level set to "private"'
>>> ftps.nlst()
['6jack', 'OpenBSD', 'antilink', 'blogbench', 'bsdcam', 'clockspeed', 'djbdns-jedi',
↪'docs', 'eaccelerator-jedi', 'favicon.ico', 'francotone', 'fugu', 'ignore', 'libpuzzle
↪', 'metalog', 'minidentd', 'misc', 'mysql-udf-global-user-variables', 'php-jenkins-hash
↪', 'php-skein-hash', 'php-webdav', 'phpaudit', 'phpbench', 'pincaster', 'ping', 'posto
↪', 'pub', 'public', 'public_keys', 'pure-ftpd', 'qscan', 'qtc', 'sharedance', 'skycache
↪', 'sound', 'tmp', 'ucarp']
```

exception ftplib.error_reply

サーバから想定外の応答があったときに送出される例外。

exception ftplib.error_temp

一時的エラーを表すエラーコード (400--499 の範囲の応答コード) を受け取った時に発生する例外。

exception ftplib.error_perm

永久エラーを表すエラーコード (500--599 の範囲の応答コード) を受け取った時に発生する例外。

exception ftplib.error_proto

File Transfer Protocol の応答仕様に適合しない、すなわち 1--5 の数字で始まらない応答コードをサーバから受け取った時に発生する例外。

ftplib.all_errors

FTP インスタンスのメソッド実行時、FTP 接続で (プログラミングのエラーと考えられるメソッドの実行によって) 発生する全ての例外 (タプル形式)。この例外には以上の 4 つのエラーはもちろん、*OSError* と *EOFError* も含まれます。

参考:

netrc **モジュール**　.netrc ファイルフォーマットのパーザ。.netrc ファイルは、FTP クライアントがユーザにプロンプトを出す前に、ユーザ認証情報をロードするのによく使われます。

21.13.1 FTP オブジェクト

いくつかのコマンドは 2 つのタイプについて実行します：1 つはテキストファイルで、もう 1 つはバイナリファイルを扱います。これらのメソッドのテキストバージョンでは `lines` 、バイナリバージョンでは `binary` の語がメソッド名の終わりについています。

FTP インスタンスには以下のメソッドがあります:

`FTP.set_debuglevel`(*level*)
インスタンスのデバッグレベルを設定します。この設定によってデバッグ時に出力される量を調節します。デフォルトは `0` で、何も出力されません。`1` なら、一般的に 1 つのコマンドあたり 1 行の適当な量のデバッグ出力を行います。`2` 以上なら、コントロール接続で受信した各行を出力して、最大のデバッグ出力をします。

`FTP.connect`(*host=''*, *port=0*, *timeout=None*, *source_address=None*)
Connect to the given host and port. The default port number is `21`, as specified by the FTP protocol specification. It is rarely needed to specify a different port number. This function should be called only once for each instance; it should not be called at all if a host was given when the instance was created. All other methods can only be used after a connection has been made. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If no *timeout* is passed, the global default timeout setting will be used. *source_address* is a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.

引数 `self`, `host`, `port` を指定して **監査イベント** `ftplib.connect` を送出します。

バージョン 3.3 で変更: *source_address* 引数が追加されました。

`FTP.getwelcome`()
サーバに最初に接続した際に送信される応答中のウェルカムメッセージを返します。(このメッセージには時に、ユーザにとって重要な免責事項や ヘルプ情報が入っています。)

`FTP.login`(*user='anonymous'*, *passwd=''*, *acct=''*)
与えられた *user* でログインします。*passwd* と *acct* のパラメータは省略可能で、デフォルトでは空文字列です。もし *user* が指定されないなら、デフォルトで `'anonymous'` になります。もし *user* が `'anonymous'` なら、デフォルトの *passwd* は `'anonymous@'` になります。この関数は各インスタンスについて一度だけ、接続が確立した後に呼び出さなければなりません。インスタンスが作られた時にホスト名とユーザ名が与えられていたら、このメソッドを実行すべきではありません。ほとんどの FTP コマンドはクライアントがログインした後に実行可能になります。*acct* 引数は "accounting information" を提供します。ほとんどのシステムはこれを実装していません。

`FTP.abort`()
実行中のファイル転送を中止します。これはいつも機能するわけではありませんが、やってみる価値はあります。

FTP.**sendcmd**(*cmd*)
シンプルなコマンド文字列をサーバに送信して、受信した文字列を返します。

引数 self, cmd を指定して **監査イベント** ftplib.sendcmd を送出します。

FTP.**voidcmd**(*cmd*)
シンプルなコマンド文字列をサーバに送信して、その応答を扱います。応答コードが成功に関係するもの (200--299 の範囲にあるコード) なら何も返しません。それ以外は *error_reply* を発生します。

引数 self, cmd を指定して **監査イベント** ftplib.sendcmd を送出します。

FTP.**retrbinary**(*cmd*, *callback*, *blocksize=8192*, *rest=None*)
バイナリ転送モードでファイルを受信します。*cmd* は適切な RETR コマンド: 'RETR filename' でなければなりません。関数 *callback* は、受信したデータブロックのそれぞれに対して、データブロックを 1 つの bytes の引数として呼び出されます。省略可能な引数 *blocksize* は、実際の転送を行うのに作られた低レベルのソケットオブジェクトから読み込む最大のチャンクサイズを指定します（これは *callback* に与えられるデータブロックの最大サイズにもなります)。妥当なデフォルト値が設定されます。*rest* は、*transfercmd()* メソッドと同じものです。

FTP.**retrlines**(*cmd*, *callback=None*)
Retrieve a file or directory listing in ASCII transfer mode. *cmd* should be an appropriate RETR command (see *retrbinary()*) or a command such as LIST or NLST (usually just the string 'LIST'). LIST retrieves a list of files and information about those files. NLST retrieves a list of file names. The *callback* function is called for each line with a string argument containing the line with the trailing CRLF stripped. The default *callback* prints the line to sys.stdout.

FTP.**set_pasv**(*val*)
val が真の場合 ” パッシブ” モードを有効化し、偽の場合は無効化します。デフォルトではパッシブモードです。

FTP.**storbinary**(*cmd*, *fp*, *blocksize=8192*, *callback=None*, *rest=None*)
バイナリ転送モードでファイルを転送します。*cmd* は適切な STOR コマンド："STOR filename" でなければなりません。*fp* は (バイナリモードで開かれた) **ファイルオブジェクト** で、read() メソッドで EOF まで読み込まれ、ブロックサイズ *blocksize* でデータが転送されます。引数 *blocksize* のデフォルト値は 8192 です。*callback* はオプションの引数で、引数を 1 つとる呼び出し可能オブジェクトを渡します。各データブロックが送信された後に、そのブロックを引数にして呼び出されます。*rest* は、*transfercmd()* メソッドにあるものと同じ意味です。

バージョン 3.2 で変更: *rest* パラメタが追加されました。

FTP.**storlines**(*cmd*, *fp*, *callback=None*)
Store a file in ASCII transfer mode. *cmd* should be an appropriate STOR command (see *storbinary()*). Lines are read until EOF from the *file object* *fp* (opened in binary mode) using its *readline()* method to provide the data to be stored. *callback* is an optional single parameter callable that is called on each line after it is sent.

FTP.**transfercmd**(*cmd*, *rest=None*)
データ接続中に転送を初期化します。もし転送中なら、EPRT あるいは PORT コマンドと、*cmd* で指定

したコマンドを送信し、接続を続けます。サーバがパッシブなら、EPSV あるいは PASV コマンドを送信して接続し、転送コマンドを開始します。どちらの場合も、接続のためのソケットを返します。

If optional *rest* is given, a REST command is sent to the server, passing *rest* as an argument. *rest* is usually a byte offset into the requested file, telling the server to restart sending the file's bytes at the requested offset, skipping over the initial bytes. Note however that **RFC 959** requires only that *rest* be a string containing characters in the printable range from ASCII code 33 to ASCII code 126. The *transfercmd()* method, therefore, converts *rest* to a string, but no check is performed on the string's contents. If the server does not recognize the REST command, an *error_reply* exception will be raised. If this happens, simply call *transfercmd()* without a *rest* argument.

FTP.**ntransfercmd**(*cmd*, *rest=None*)

transfercmd() と同様ですが、データと予想されるサイズとのタプルを返します。もしサイズが計算できないなら、サイズの代わりに None が返されます。*cmd* と *rest* は *transfercmd()* のものと同じです。

FTP.**mlsd**(*path=""*, *facts=[]*)

List a directory in a standardized format by using MLSD command (**RFC 3659**). If *path* is omitted the current directory is assumed. *facts* is a list of strings representing the type of information desired (e.g. `["type", "size", "perm"]`). Return a generator object yielding a tuple of two elements for every file found in path. First element is the file name, the second one is a dictionary containing facts about the file name. Content of this dictionary might be limited by the *facts* argument but server is not guaranteed to return all requested facts.

バージョン 3.3 で追加.

FTP.**nlst**(*argument*[, *...*])

NLST コマンドで返されるファイル名のリストを返します。省略可能な *argument* は、リストアップするディレクトリです（デフォルトではサーバのカレントディレクトリです）。NLST コマンドに非標準である複数の引数を渡すことができます。

注釈: If your server supports the command, *mlsd()* offers a better API.

FTP.**dir**(*argument*[, *...*])

LIST コマンドで返されるディレクトリ内のリストを作り、標準出力へ出力します。省略可能な *argument* は、リストアップするディレクトリです（デフォルトではサーバのカレントディレクトリです）。LIST コマンドに非標準である複数の引数を渡すことができます。もし最後の引数が関数なら、*retrlines()* のように *callback* として使われます; デフォルトでは sys.stdout に印字します。このメソッドは None を返します。

注釈: If your server supports the command, *mlsd()* offers a better API.

FTP.**rename**(*fromname*, *toname*)

サーバ上のファイルのファイル名 *fromname* を *toname* へ変更します。

FTP.**delete**(*filename*)
サーバからファイル *filename* を削除します。成功したら応答のテキストを返し、そうでないならパーミッションエラーでは *error_perm* を、他のエラーでは *error_reply* を返します。

FTP.**cwd**(*pathname*)
サーバのカレントディレクトリを設定します。

FTP.**mkd**(*pathname*)
サーバ上に新たにディレクトリを作ります。

FTP.**pwd**()
サーバ上のカレントディレクトリのパスを返します。

FTP.**rmd**(*dirname*)
サーバ上のディレクトリ *dirname* を削除します。

FTP.**size**(*filename*)
サーバ上のファイル *filename* のサイズを尋ねます。成功したらファイルサイズが整数で返され、そうでないなら None が返されます。SIZE コマンドは標準化されていませんが、多くの普通のサーバで実装されていることに注意して下さい。

FTP.**quit**()
サーバに QUIT コマンドを送信し、接続を閉じます。これは接続を閉じるのに" 礼儀正しい" 方法ですが、QUIT コマンドに反応してサーバの例外が発生するかもしれません。この例外は、*close()* メソッドによって *FTP* インスタンスに対するその後のコマンド使用が不可になっていることを示しています (下記参照)。

FTP.**close**()
接続を一方的に閉じます。既に閉じた接続に対して実行すべきではありません (例えば *quit()* を呼び出して成功した後など)。この実行の後、*FTP* インスタンスはもう使用すべきではありません (*close()* あるいは *quit()* を呼び出した後で、*login()* メソッドをもう一度実行して再び接続を開くことはできません)。

21.13.2 FTP_TLS オブジェクト

FTP_TLS クラスは *FTP* を継承し、さらにオブジェクトを定義します:

FTP_TLS.**ssl_version**
使用する SSL のバージョン (デフォルトは *ssl.PROTOCOL_SSLv23*) です。

FTP_TLS.**auth**()
ssl_version 属性で指定されたものに従って、TLS または SSL を使い、セキュアコントロール接続をセットアップします。

バージョン 3.4 で変更: このメソッドは *ssl.SSLContext.check_hostname* と *Server Name Indication* でホスト名のチェックをサポートしました。(*ssl.HAS_SNI* を参照してください)。

FTP_TLS.ccc()
Revert control channel back to plaintext. This can be useful to take advantage of firewalls that know how to handle NAT with non-secure FTP without opening fixed ports.

バージョン 3.3 で追加.

FTP_TLS.prot_p()
セキュアデータ接続をセットアップします。

FTP_TLS.prot_c()
平文データ接続をセットアップします。

21.14 poplib --- POP3 プロトコルクライアント

ソースコード: Lib/poplib.py

このモジュールはクラス *POP3* を定義しています。*POP3* は POP3 サーバへの接続をカプセル化し、**RFC 1939** で定義されているプロトコルを実装しています。*POP3* クラスは **RFC 1939** の最小限のコマンドセットとオプションのコマンドセットをサポートしています。既に確立されている接続で暗号化された通信を行うために、**RFC 2595** で導入された STLS コマンドもサポートされています。

加えて、このモジュールはクラス *POP3_SSL* を提供しています。*POP3_SSL* は SSL を下層のプロトコルレイヤーとして使う POP3 サーバへの接続をサポートしています。

POP3 についての注意事項は、それが広くサポートされているにもかかわらず、既に時代遅れだということです。幾つも実装されている POP3 サーバーの品質は、貧弱なものが多数を占めています。もし、お使いのメールサーバーが IMAP をサポートしているなら、*imaplib.IMAP4* クラスが使えます。IMAP サーバーは、より良く実装されている傾向があります。

poplib モジュールは二つのクラスを提供します:

class poplib.POP3(*host*, *port=POP3_PORT*[, *timeout*])
このクラスが、実際に POP3 プロトコルを実装します。インスタンスが初期化されるときに、コネクションが作成されます。*port* が省略されると、POP3 標準のポート (110) が使われます。オプションの *timeout* 引数は、接続時のタイムアウト時間を秒数で指定します (指定されなかった場合は、グローバルのデフォルトタイムアウト設定が利用されます)。

引数 self, host, port 付きで **監査イベント** poplib.connect を送出します。

引数 self, line を指定して **監査イベント** poplib.putline を送出します。

class poplib.POP3_SSL(*host*, *port=POP3_SSL_PORT*, *keyfile=None*, *certfile=None*, *timeout=None*, *context=None*)
POP3 クラスのサブクラスで、SSL でカプセル化されたソケットによる POP サーバへの接続を提供します。*port* が指定されていない場合、POP3-over-SSL 標準の 995 番ポートが使われます。*timeout* については *POP3* クラスのコンストラクタの引数と同じです。*context* は SSL の設定、証明書、秘密鍵を一つの (POP3_SSL オブジェクトよりも長く存在し続けうる) 構造にまとめた *ssl.SSLContext* オブ

ジェクトで、省略可能です。ベストプラクティスについては **セキュリティで考慮すべき点** を参照してください。

keyfile と *certfile* は *context* のレガシー版です - これらは、SSL 接続のための、PEM フォーマットの秘密鍵と証明書チェーンファイル名 (前者が *keyfile* 、後者が *certfile*) を含むことができます。

引数 `self`, `host`, `port` 付きで **監査イベント** `poplib.connect` を送出します。

引数 `self`, `line` を指定して **監査イベント** `poplib.putline` を送出します。

バージョン 3.2 で変更: *context* 引数が追加されました。

バージョン 3.4 で変更: このクラスは *ssl.SSLContext.check_hostname* と *Server Name Indication* でホスト名のチェックをサポートしました。(*ssl.HAS_SNI* を参照してください)。

バージョン 3.6 で非推奨: *keyfile* および *certfile* は非推奨となったので、*context* を使ってください。代わりに *ssl.SSLContext.load_cert_chain()* を使うか、または *ssl.create_default_context()* にシステムが信頼する CA 証明書を選んでもらうかしてください。

1 つの例外が、*poplib* モジュールのアトリビュートとして定義されています:

exception poplib.**error_proto**
このモジュール内で起こったあらゆるエラーで送出される例外です (*socket* モジュールからのエラーは捕捉されません)。例外の理由は文字列としてコンストラクタに渡されます。

参考:

モジュール *imaplib* 標準 Python IMAP モジュールです。

Frequently Asked Questions About Fetchmail POP/IMAP クライアント `fetchmail` の FAQ。POP プロトコルをベースにしたアプリケーションを書くときに有用な、POP3 サーバの種類や RFC への適合度といった情報を収集しています。

21.14.1 POP3 オブジェクト

POP3 コマンドはすべて、それと同じ名前のメソッドとして lower-case で表現されます。そしてそのほとんどは、サーバからのレスポンスとなるテキストを返します。

POP3 クラスのインスタンスは以下のメソッドを持ちます:

POP3.**set_debuglevel**(*level*)
インスタンスのデバッグレベルを設定します。この設定によってデバッグ時に出力される量を調節します。デフォルトは 0 で、何も出力されません。1 なら、一般的に 1 つのコマンドあたり 1 行の適当な量のデバッグ出力を行います。2 以上なら、コントロール接続で受信した各行を出力して、最大のデバッグ出力をします。

POP3.**getwelcome**()
POP3 サーバーから送られるグリーティングメッセージを返します。

POP3.**capa**()
: **RFC 2449** で規定されている機能についてサーバに問い合わせます。`{'name': ['param'...]}` という形の辞書を返します。

  バージョン 3.4 で追加.

POP3.**user**(*username*)
: user コマンドを送出します。応答はパスワード要求を表示します。

POP3.**pass_**(*password*)
: パスワードを送出します。応答は、メッセージ数とメールボックスのサイズを含みます。注意：サーバー上のメールボックスは `quit()` が呼ばれるまでロックされます。

POP3.**apop**(*user*, *secret*)
: POP3 サーバーにログオンするのに、よりセキュアな APOP 認証を使用します。

POP3.**rpop**(*user*)
: POP3 サーバーにログオンするのに、(UNIX の r-コマンドと同様の) RPOP 認証を使用します。

POP3.**stat**()
: メールボックスの状態を得ます。結果は 2 つの integer からなるタプルとなります。`(message count, mailbox size)`.

POP3.**list**(*[which]*)
: メッセージのリストを要求します。結果は `(response, ['mesg_num octets', ...], octets)` という形式で表されます。*which* が与えられると、それによりメッセージを指定します。

POP3.**retr**(*which*)
: *which* 番のメッセージ全体を取り出し、そのメッセージに既読フラグを立てます。結果は `(response, ['line', ...], octets)` という形式で表されます。

POP3.**dele**(*which*)
: *which* 番のメッセージに削除のためのフラグを立てます。ほとんどのサーバで、QUIT コマンドが実行されるまでは実際の削除は行われません (もっとも良く知られた例外は Eudora QPOP で、その配送メカニズムは RFC に違反しており、どんな切断状況でも削除操作を未解決にしています)。

POP3.**rset**()
: メールボックスの削除マークすべてを取り消します。

POP3.**noop**()
: 何もしません。接続保持のために使われます。

POP3.**quit**()
: サインオフ: 変更をコミットし、メールボックスをアンロックして、接続を破棄します。

POP3.**top**(*which*, *howmuch*)
: メッセージヘッダと *howmuch* で指定した行数のメッセージを、*which* で指定したメッセージ分取り出します。結果は以下のような形式となります。`(response, ['line', ...], octets)`.

  このメソッドは POP3 の TOP コマンドを利用し、RETR コマンドのように、メッセージに既読フラ

グをセットしません。残念ながら、TOP コマンドは RFC では貧弱な仕様しか定義されておらず、しばしばノーブランドのサーバーでは（その仕様が）守られていません。このメソッドを信用してしまう前に、実際に使用する POP サーバーでテストをしてください。

POP3.**uidl**(*which=None*)
（ユニーク ID による）メッセージダイジェストのリストを返します。*which* が設定されている場合、結果はユニーク ID を含みます。それは `'response mesgnum uid` という形式のメッセージ、または `(response, ['mesgnum uid', ...], octets)` という形式のリストとなります。

POP3.**utf8**()
UTF-8 モードへの切り替えを試行します。成功した場合はサーバの応答を返し、失敗した場合は *error_proto* を送出します。**RFC 6856** で規定されています。

バージョン 3.5 で追加.

POP3.**stls**(*context=None*)
アクティブな接続にて **RFC 2595** で定められた方法で TLS セッションを開始します。TLS セッションはユーザ認証を行う前に開始する必要があります。

context は SSL の設定、証明書、秘密鍵を一つの (POP3 オブジェクトよりも長く存在し続けうる) 構造にまとめた *ssl.SSLContext* オブジェクトです。ベストプラクティスについては **セキュリティで考慮すべき点** を参照してください。

このメソッドは *ssl.SSLContext.check_hostname* と *Server Name Indication* でホスト名のチェックをサポートしました。(*ssl.HAS_SNI* を参照してください)。

バージョン 3.4 で追加.

POP3_SSL クラスのインスタンスは追加のメソッドを持ちません。このサブクラスのインターフェイスは親クラスと同じです。

21.14.2 POP3 の例

以下にメールボックスを開き、全てのメッセージを取得して印刷する最小の (エラーチェックをしない) 使用例を示します:

```
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print(j)
```

モジュールの末尾に、より拡張的な使用例が収められたテストセクションがあります。

21.15 imaplib --- IMAP4 プロトコルクライアント

ソースコード: Lib/imaplib.py

このモジュールでは三つのクラス、*IMAP4*, *IMAP4_SSL* と *IMAP4_stream* を定義します。これらのクラスは IMAP4 サーバへの接続をカプセル化し、**RFC 2060** に定義されている IMAP4rev1 クライアントプロトコルの大規模なサブセットを実装しています。このクラスは IMAP4 (**RFC 1730**) 準拠のサーバと後方互換性がありますが、STATUS コマンドは IMAP4 ではサポートされていないので注意してください。

imaplib モジュール内では三つのクラスを提供しており、*IMAP4* は基底クラスとなります:

class imaplib.**IMAP4**(*host=''*, *port=IMAP4_PORT*)

このクラスは実際の IMAP4 プロトコルを実装しています。インスタンスが初期化された際に接続が生成され、プロトコルバージョン (IMAP4 または IMAP4rev1) が決定されます。*host* が指定されていない場合、'' (ローカルホスト) が用いられます。*port* が省略された場合、標準の IMAP4 ポート番号 (143) が用いられます。

The *IMAP4* class supports the `with` statement. When used like this, the IMAP4 LOGOUT command is issued automatically when the `with` statement exits. E.g.:

```
>>> from imaplib import IMAP4
>>> with IMAP4("domain.org") as M:
...     M.noop()
...
('OK', [b'Nothing Accomplished. d25if65hy903weo.87'])
```

バージョン 3.5 で変更: `with` 構文のサポートが追加されました。

例外は *IMAP4* クラスの属性として定義されています:

exception IMAP4.**error**

何らかのエラー発生の際に送出される例外です。例外の理由は文字列としてコンストラクタに渡されます。

exception IMAP4.**abort**

IMAP4 サーバのエラーが生じると、この例外が送出されます。この例外は *IMAP4.error* のサブクラスです。通常、インスタンスを閉じ、新たなインスタンスを再び生成することで、この例外から復旧できます。

exception IMAP4.**readonly**

この例外は書き込み可能なメールボックスの状態がサーバによって変更された際に送出されます。この例外は *IMAP4.error* のサブクラスです。他の何らかのクライアントが現在書き込み権限を獲得しており、メールボックスを開きなおして書き込み権限を再獲得する必要があります。

このモジュールではもう一つ、安全 (secure) な接続を使ったサブクラスがあります:

class imaplib.**IMAP4_SSL**(*host=''*, *port=IMAP4_SSL_PORT*, *keyfile=None*, *certfile=None*, *ssl_context=None*)

This is a subclass derived from *IMAP4* that connects over an SSL encrypted socket (to use this class

you need a socket module that was compiled with SSL support). If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4-over-SSL port (993) is used. *ssl_context* is a *ssl.SSLContext* object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read **セキュリティで考慮すべき点** for best practices.

keyfile and *certfile* are a legacy alternative to *ssl_context* - they can point to PEM-formatted private key and certificate chain files for the SSL connection. Note that the *keyfile*/*certfile* parameters are mutually exclusive with *ssl_context*, a *ValueError* is raised if *keyfile*/*certfile* is provided along with *ssl_context*.

バージョン 3.3 で変更: *ssl_context* parameter added.

バージョン 3.4 で変更: このクラスは *ssl.SSLContext.check_hostname* と *Server Name Indication* でホスト名のチェックをサポートしました。(*ssl.HAS_SNI* を参照してください)。

バージョン 3.6 で非推奨: *keyfile* および *certfile* は非推奨となったので、*ssl_context* を使ってください。代わりに *ssl.SSLContext.load_cert_chain()* を使うか、または *ssl.create_default_context()* にシステムが信頼する CA 証明書を選んでもらうかしてください。

さらにもう一つのサブクラスは、子プロセスで確立した接続を使用する場合に使用します:

class imaplib.**IMAP4_stream**(*command*)

IMAP4 から派生したサブクラスで、*command* を `subprocess.Popen()` に渡して作成される `stdin`/`stdout` ディスクリプタと接続します。

以下のユーティリティ関数が定義されています:

imaplib.**Internaldate2tuple**(*datestr*)

IMAP4 の `INTERNALDATE` 文字列を解析してそれに相当するローカルタイムを返します。戻り値は *time.struct_time* のタプルか、文字列のフォーマットが不正な場合は `None` です。

imaplib.**Int2AP**(*num*)

整数を [`A` .. `P`] からなる文字集合を用いて表現した bytes に変換します。

imaplib.**ParseFlags**(*flagstr*)

IMAP4 `FLAGS` 応答を個々のフラグからなるタプルに変換します。

imaplib.**Time2Internaldate**(*date_time*)

Convert *date_time* to an IMAP4 `INTERNALDATE` representation. The return value is a string in the form: `"DD-Mmm-YYYY HH:MM:SS +HHMM"` (including double-quotes). The *date_time* argument can be a number (int or float) representing seconds since epoch (as returned by *time.time()*), a 9-tuple representing local time an instance of *time.struct_time* (as returned by *time.localtime()*), an aware instance of *datetime.datetime*, or a double-quoted string. In the last case, it is assumed to already be in the correct format.

IMAP4 メッセージ番号は、メールボックスに対する変更が行われた後には変化します; 特に、`EXPUNGE` 命令はメッセージの削除を行いますが、残ったメッセージには再度番号を振りなおします。従って、メッセージ番号ではなく、UID 命令を使い、その UID を利用するよう強く勧めます。

モジュールの末尾に、より拡張的な使用例が収められたテストセクションがあります。

参考:

Documents describing the protocol, sources for servers implementing it, by the University of Washington's IMAP Information Center can all be found at (**Source Code**) https://github.com/uw-imap/imap (**Not Maintained**).

21.15.1 IMAP4 オブジェクト

全ての IMAP4rev1 命令は、同じ名前のメソッドで表されており、大文字のものも小文字のものもあります。

命令に対する引数は全て文字列に変換されます。例外は `AUTHENTICATE` の引数と `APPEND` の最後の引数で、これは IMAP4 リテラルとして渡されます。必要に応じて (IMAP4 プロトコルが感知対象としている文字が文字列に入っており、かつ丸括弧か二重引用符で囲われていなかった場合) 文字列はクオートされます。しかし、`LOGIN` 命令の *password* 引数は常にクオートされます。文字列がクオートされないようにしたい (例えば `STORE` 命令の *flags* 引数) 場合、文字列を丸括弧で囲んでください (例: `r'(\Deleted)'`)。

各命令はタプル: `(type, [data, ...])` を返し、*type* は通常 `'OK'` または `'NO'` です。*data* は命令に対する応答をテキストにしたものか、命令に対する実行結果です。各 *data* は `bytes` かタプルとなります。タプルの場合、最初の要素はレスポンスのヘッダで、次の要素にはデータが格納されます (ie: 'literal' value)。

以下のコマンドにおける *message_set* オプションは、操作の対象となるひとつあるいは複数のメッセージを指す文字列です。単一のメッセージ番号 (`'1'`) かメッセージ番号の範囲 (`'2:4'`)、あるいは連続していないメッセージをカンマでつなげたもの (`'1:3,6:9'`) となります。範囲指定でアスタリスクを使用すると、上限を無限とすることができます (`'3:*'`)。

IMAP4 のインスタンスは以下のメソッドを持っています:

`IMAP4.append`(*mailbox*, *flags*, *date_time*, *message*)
指定された名前のメールボックスに *message* を追加します。

`IMAP4.authenticate`(*mechanism*, *authobject*)
認証命令です --- 応答の処理が必要です。

mechanism は利用する認証メカニズムを与えます。認証メカニズムはインスタンス変数 `capabilities` の中に `AUTH=mechanism` という形式で現れる必要があります。

authobject は呼び出し可能なオブジェクトである必要があります:

```
data = authobject(response)
```

It will be called to process server continuation responses; the *response* argument it is passed will be `bytes`. It should return `bytes` *data* that will be base64 encoded and sent to the server. It should return `None` if the client abort response `*` should be sent instead.

バージョン 3.5 で変更: string usernames and passwords are now encoded to `utf-8` instead of being limited to ASCII.

IMAP4.**check**()
サーバ上のメールボックスにチェックポイントを設定します。

IMAP4.**close**()
現在選択されているメールボックスを閉じます。削除されたメッセージは書き込み可能メールボックスから除去されます。LOGOUT 前に実行することを勧めます。

IMAP4.**copy**(*message_set*, *new_mailbox*)
message_set で指定したメッセージ群を *new_mailbox* の末尾にコピーします。

IMAP4.**create**(*mailbox*)
mailbox と名づけられた新たなメールボックスを生成します。

IMAP4.**delete**(*mailbox*)
mailbox と名づけられた古いメールボックスを削除します。

IMAP4.**deleteacl**(*mailbox*, *who*)
mailbox における who についての ACL を削除 (権限を削除) します。

IMAP4.**enable**(*capability*)
Enable *capability* (see **RFC 5161**). Most capabilities do not need to be enabled. Currently only the UTF8=ACCEPT capability is supported (see **RFC 6855**).

バージョン 3.5 で追加: The *enable()* method itself, and **RFC 6855** support.

IMAP4.**expunge**()
選択されたメールボックスから削除された要素を永久に除去します。各々の削除されたメッセージに対して、EXPUNGE 応答を生成します。返されるデータには EXPUNGE メッセージ番号を受信した順番に並べたリストが入っています。

IMAP4.**fetch**(*message_set*, *message_parts*)
メッセージ (の一部) を取りよせます。*message_parts* はメッセージパートの名前を表す文字列を丸括弧で囲ったもので、例えば: "(UID BODY[TEXT])" のようになります。返されるデータはメッセージパートのエンベロープ情報とデータからなるタプルです。

IMAP4.**getacl**(*mailbox*)
mailbox に対する ACL を取得します。このメソッドは非標準ですが、Cyrus サーバでサポートされています。

IMAP4.**getannotation**(*mailbox*, *entry*, *attribute*)
mailbox に対する ANNOTATION を取得します。このメソッドは非標準ですが、Cyrus サーバでサポートされています。

IMAP4.**getquota**(*root*)
quota *root* により、リソース使用状況と制限値を取得します。このメソッドは **RFC 2087** で定義されている IMAP4 QUOTA 拡張の一部です。

IMAP4.**getquotaroot**(*mailbox*)
mailbox に対して quota *root* を実行した結果のリストを取得します。このメソッドは **RFC 2087** で定義されている IMAP4 QUOTA 拡張の一部です。

IMAP4.**list**(*[directory[, pattern]]*)
pattern にマッチする *directory* メールボックス名を列挙します。*directory* の標準の設定値は最上レベルのメールフォルダで、*pattern* は標準の設定では全てにマッチします。返されるデータには LIST 応答のリストが入っています。

IMAP4.**login**(*user*, *password*)
平文パスワードを使ってクライアントを照合します。*password* はクオートされます。

IMAP4.**login_cram_md5**(*user*, *password*)
パスワードの保護のため、クライアント認証時に CRAM-MD5 だけを使用します。これは、CAPABILITY レスポンスに AUTH=CRAM-MD5 が含まれる場合のみ有効です。

IMAP4.**logout**()
サーバへの接続を遮断します。サーバからの BYE 応答を返します。

バージョン 3.8 で変更: The method no longer ignores silently arbitrary exceptions.

IMAP4.**lsub**(*directory='""'*, *pattern='*'*)
購読しているメールボックス名のうち、ディレクトリ内でパターンにマッチするものを列挙します。*directory* の標準の設定値は最上レベルのメールフォルダで、*pattern* は標準の設定では全てにマッチします。返されるデータには返されるデータはメッセージパートエンベロープ情報とデータからなるタプルです。

IMAP4.**myrights**(*mailbox*)
mailbox における自分の ACL を返します (すなわち自分が mailbox で持っている権限を返します)。

IMAP4.**namespace**()
RFC 2342 で定義される IMAP 名前空間を返します。

IMAP4.**noop**()
サーバに NOOP を送信します。

IMAP4.**open**(*host*, *port*)
host 上の *port* に対するソケットを開きます。このメソッドは *IMAP4* のコンストラクタから暗黙的に呼び出されます。このメソッドで確立された接続オブジェクトは *IMAP4.read()*, *IMAP4.readline()*, *IMAP4.send()*, *IMAP4.shutdown()* メソッドで使われます。このメソッドはオーバライドすることができます。

引数 self, host, port を指定して **監査イベント** imaplib.open を送出します。

IMAP4.**partial**(*message_num*, *message_part*, *start*, *length*)
メッセージの後略された部分を取り寄せます。返されるデータはメッセージパートエンベロープ情報とデータからなるタプルです。

IMAP4.**proxyauth**(*user*)
user として認証されたものとします。認証された管理者がユーザの代理としてメールボックスにアクセスする際に使用します。

IMAP4.**read**(*size*)
遠隔のサーバから *size* バイト読み出します。このメソッドはオーバライドすることができます。

IMAP4.**readline**()
遠隔のサーバから一行読み出します。このメソッドはオーバライドすることができます。

IMAP4.**recent**()
サーバに更新を促します。新たなメッセージがない場合応答は None になり、そうでない場合 RECENT 応答の値になります。

IMAP4.**rename**(*oldmailbox*, *newmailbox*)
oldmailbox という名前のメールボックスを *newmailbox* に名称変更します。

IMAP4.**response**(*code*)
応答 *code* を受信していれば、そのデータを返し、そうでなければ None を返します。通常の形式 (usual type) ではなく指定したコードを返します。

IMAP4.**search**(*charset*, *criterion*[, ...])
Search mailbox for matching messages. *charset* may be None, in which case no CHARSET will be specified in the request to the server. The IMAP protocol requires that at least one criterion be specified; an exception will be raised when the server returns an error. *charset* must be None if the UTF8=ACCEPT capability was enabled using the *enable()* command.

以下はプログラム例です:

```
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

IMAP4.**select**(*mailbox='INBOX'*, *readonly=False*)
メールボックスを選択します。返されるデータは *mailbox* 内のメッセージ数 (EXISTS 応答) です。標準の設定では *mailbox* は 'INBOX' です。*readonly* が設定された場合、メールボックスに対する変更はできません。

IMAP4.**send**(*data*)
遠隔のサーバに data を送信します。このメソッドはオーバライドすることができます。

引数 self, data を指定して 監査イベント imaplib.send を送出します。

IMAP4.**setacl**(*mailbox*, *who*, *what*)
ACL を *mailbox* に設定します。このメソッドは非標準ですが、Cyrus サーバでサポートされています。

IMAP4.**setannotation**(*mailbox*, *entry*, *attribute*[, ...])
ANNOTATION を *mailbox* に設定します。このメソッドは非標準ですが、Cyrus サーバでサポートされています。

IMAP4.**setquota**(*root*, *limits*)
quota *root* のリソースを *limits* に設定します。このメソッドは **RFC 2087** で定義されている IMAP4 QUOTA 拡張の一部です。

IMAP4.**shutdown**()

open で確立された接続を閉じます。*IMAP4.logout()* は暗黙的にこのメソッドを呼び出します。このメソッドはオーバライドすることができます。

IMAP4.**socket**()
サーバへの接続に使われているソケットインスタンスを返します。

IMAP4.**sort**(*sort_criteria*, *charset*, *search_criterion*[, ...])
sort 命令は search に結果の並べ替え (sort) 機能をつけた変種です。返されるデータには、条件に合致するメッセージ番号をスペースで分割したリストが入っています。

sort 命令は *search_criterion* の前に二つの引数を持ちます; *sort_criteria* のリストを丸括弧で囲ったものと、検索時の *charset* です。search と違って、検索時の *charset* は必須です。uid sort 命令もあり、search に対する uid search と同じように sort 命令に対応します。sort 命令はまず、charset 引数の指定に従って searching criteria の文字列を解釈し、メールボックスから与えられた検索条件に合致するメッセージを探します。次に、合致したメッセージの数を返します。

IMAP4rev1 拡張命令です。

IMAP4.**starttls**(*ssl_context=None*)
Send a STARTTLS command. The *ssl_context* argument is optional and should be a *ssl.SSLContext* object. This will enable encryption on the IMAP connection. Please read **セキュリティで考慮すべき点** for best practices.

バージョン 3.2 で追加.

バージョン 3.4 で変更: このメソッドは *ssl.SSLContext.check_hostname* と *Server Name Indication* でホスト名のチェックをサポートしました。(*ssl.HAS_SNI* を参照してください)。

IMAP4.**status**(*mailbox*, *names*)
mailbox の指定ステータス名の状態情報を要求します。

IMAP4.**store**(*message_set*, *command*, *flag_list*)
メールボックス内のメッセージ群のフラグ設定を変更します。*command* は **RFC 2060** のセクション 6.4.6 で指定されているもので、"FLAGS", "+FLAGS", あるいは "-FLAGS" のいずれかとなります。オプションで末尾に ".SILENT" がつくこともあります。

たとえば、すべてのメッセージに削除フラグを設定するには次のようにします:

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
   M.store(num, '+FLAGS', '\\Deleted')
M.expunge()
```

注釈: Creating flags containing ']' (for example: "[test]") violates **RFC 3501** (the IMAP protocol). However, imaplib has historically allowed creation of such tags, and popular IMAP servers, such as Gmail, accept and produce such flags. There are non-Python programs which also create such tags. Although it is an RFC violation and IMAP clients and servers are supposed to be strict, imaplib nonetheless continues to allow such tags to be created for backward compatibility

reasons, and as of Python 3.6, handles them if they are sent from the server, since this improves real-world compatibility.

IMAP4.**subscribe**(*mailbox*)
新たなメールボックスを購読 (subscribe) します。

IMAP4.**thread**(*threading_algorithm*, *charset*, *search_criterion*[, ...])
`thread` コマンドは `search` にスレッドの概念を加えた変形版です。返されるデータは空白で区切られたスレッドメンバのリストを含んでいます。

各スレッドメンバは 0 以上のメッセージ番号からなり、空白で区切られており、親子関係を示しています。

`thread` コマンドは *search_criterion* 引数の前に 2 つの引数を持っています。*threading_algorithm* と *charset* です。`search` コマンドとは違い、*charset* は必須です。`search` に対する `uid search` と同様に、`thread` にも `uid thread` があります。`thread` コマンドはまずメールボックス中のメッセージを、charset を用いた検索条件で検索します。その後マッチしたメッセージを指定されたスレッドアルゴリズムでスレッド化して返します.

`IMAP4rev1` 拡張命令です。

IMAP4.**uid**(*command*, *arg*[, ...])
command args を、メッセージ番号ではなく UID で指定されたメッセージ群に対して実行します。命令内容に応じた応答を返します。少なくとも一つの引数を与えなくてはなりません; 何も与えない場合、サーバはエラーを返し、例外が送出されます。

IMAP4.**unsubscribe**(*mailbox*)
古いメールボックスの購読を解除 (unsubscribe) します。

IMAP4.**xatom**(*name*[, ...])
サーバから `CAPABILITY` 応答で通知された単純な拡張命令を許容 (allow) します。

以下の属性が *IMAP4* のインスタンス上で定義されています:

IMAP4.**PROTOCOL_VERSION**
サーバから返された `CAPABILITY` 応答にある、サポートされている最新のプロトコルです。

IMAP4.**debug**
デバッグ出力を制御するための整数値です。初期値はモジュール変数 `Debug` から取られます。3 以上の値にすると各命令をトレースします。

IMAP4.**utf8_enabled**
Boolean value that is normally `False`, but is set to `True` if an *enable()* command is successfully issued for the `UTF8=ACCEPT` capability.

バージョン 3.5 で追加.

21.15.2 IMAP4 の使用例

以下にメールボックスを開き、全てのメッセージを取得して印刷する最小の (エラーチェックをしない) 使用例を示します:

```
import getpass, imaplib

M = imaplib.IMAP4()
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print('Message %s\n%s\n' % (num, data[0][1]))
M.close()
M.logout()
```

21.16 nntplib --- NNTP プロトコルクライアント

ソースコード: Lib/nntplib.py

This module defines the class *NNTP* which implements the client side of the Network News Transfer Protocol. It can be used to implement a news reader or poster, or automated news processors. It is compatible with **RFC 3977** as well as the older **RFC 977** and **RFC 2980**.

以下にこのモジュールの使い方の小さな例を二つ示します。ニュースグループに関する統計情報を列挙し、最新 10 件の記事を出力するには以下のようにします:

```
>>> s = nntplib.NNTP('news.gmane.io')
>>> resp, count, first, last, name = s.group('gmane.comp.python.committers')
>>> print('Group', name, 'has', count, 'articles, range', first, 'to', last)
Group gmane.comp.python.committers has 1096 articles, range 1 to 1096
>>> resp, overviews = s.over((last - 9, last))
>>> for id, over in overviews:
...     print(id, nntplib.decode_header(over['subject']))
...
1087 Re: Commit privileges for Łukasz Langa
1088 Re: 3.2 alpha 2 freeze
1089 Re: 3.2 alpha 2 freeze
1090 Re: Commit privileges for Łukasz Langa
1091 Re: Commit privileges for Łukasz Langa
1092 Updated ssh key
1093 Re: Updated ssh key
1094 Re: Updated ssh key
1095 Hello fellow committers!
1096 Re: Hello fellow committers!
>>> s.quit()
'205 Bye!'
```

バイナリファイルから記事を投稿するには、以下のようにします。(この例では記事番号は有効な番号を指定していて、あなたがそのニュースグループに投稿する 権限を持っていると仮定しています)

```
>>> s = nntplib.NNTP('news.gmane.io')
>>> f = open('article.txt', 'rb')
>>> s.post(f)
'240 Article posted successfully.'
>>> s.quit()
'205 Bye!'
```

このモジュール自体では以下のクラスを定義しています:

class nntplib.NNTP(*host*, *port=119*, *user=None*, *password=None*, *readermode=None*, *usenetrc=False*[, *timeout*])

Return a new *NNTP* object, representing a connection to the NNTP server running on host *host*, listening at port *port*. An optional *timeout* can be specified for the socket connection. If the optional *user* and *password* are provided, or if suitable credentials are present in `/.netrc` and the optional flag *usenetrc* is true, the `AUTHINFO USER` and `AUTHINFO PASS` commands are used to identify and authenticate the user to the server. If the optional flag *readermode* is true, then a `mode reader` command is sent before authentication is performed. Reader mode is sometimes necessary if you are connecting to an NNTP server on the local machine and intend to call reader-specific commands, such as `group`. If you get unexpected *NNTPPermanentError*s, you might need to set *readermode*. The *NNTP* class supports the `with` statement to unconditionally consume *OSError* exceptions and to close the NNTP connection when done, e.g.:

```
>>> from nntplib import NNTP
>>> with NNTP('news.gmane.io') as n:
...     n.group('gmane.comp.python.committers')
...
('211 1755 1 1755 gmane.comp.python.committers', 1755, 1, 1755, 'gmane.comp.python.
↪committers')
>>>
```

引数 `self`, `host`, `port` を指定して 監査イベント `nntplib.connect` を送出します。

引数 `self`, `line` を指定して 監査イベント `nntplib.putline` を送出します。

バージョン 3.2 で変更: *usenetrc* がデフォルトで `False` になりました。

バージョン 3.3 で変更: `with` 構文のサポートが追加されました。

class nntplib.NNTP_SSL(*host*, *port=563*, *user=None*, *password=None*, *ssl_context=None*, *readermode=None*, *usenetrc=False*[, *timeout*])

Return a new *NNTP_SSL* object, representing an encrypted connection to the NNTP server running on host *host*, listening at port *port*. *NNTP_SSL* objects have the same methods as *NNTP* objects. If *port* is omitted, port 563 (NNTPS) is used. *ssl_context* is also optional, and is a *SSLContext* object. Please read セキュリティで考慮すべき点 for best practices. All other parameters behave the same as for *NNTP*.

Note that SSL-on-563 is discouraged per **RFC 4642**, in favor of STARTTLS as described below.

However, some servers only support the former.

引数 self, host, port を指定して 監査イベント nntplib.connect を送出します。

引数 self, line を指定して 監査イベント nntplib.putline を送出します。

バージョン 3.2 で追加.

バージョン 3.4 で変更: このクラスは *ssl.SSLContext.check_hostname* と *Server Name Indication* でホスト名のチェックをサポートしました。(*ssl.HAS_SNI* を参照してください)。

exception nntplib.NNTPError

標準の例外 *Exception* から派生しており、*nntplib* モジュールが送出する全ての例外の基底クラスです。このクラスのインスタンスは以下の属性を持っています:

response

利用可能な場合サーバの応答です。*str* オブジェクトです。

exception nntplib.NNTPReplyError

サーバから想定外の応答があったときに送出される例外。

exception nntplib.NNTPTemporaryError

400--499 の範囲の応答コードを受け取ったときに送出される例外。

exception nntplib.NNTPPermanentError

500--599 の範囲の応答コードを受け取ったときに送出される例外。

exception nntplib.NNTPProtocolError

1--5 の数字で始まらない応答コードをサーバから受け取ったときに送出される例外。

exception nntplib.NNTPDataError

応答データに何らかのエラーがあったときに送出される例外。

21.16.1 NNTP オブジェクト

接続されたとき *NNTP* および *NNTP_SSL* オブジェクトは以下のメソッドと属性をサポートします:

属性

NNTP.**nntp_version**

サーバがサポートしている NNTP プロトコルのバージョンを表す整数です。実際のところ、この値は **RFC 3977** 準拠を通知しているサーバでは 2 で、他の場合は 1 のはずです。

バージョン 3.2 で追加.

NNTP.**nntp_implementation**

NNTP サーバのソフトウェア名とバージョンを記述する文字列です。サーバが通知していない場合は *None* です。

バージョン 3.2 で追加.

メソッド

全てのメソッドにおける戻り値のタプルで最初の要素となる *response* は、サーバの応答です: この文字列は 3 桁の数字からなるコードで始まります。サーバの応答がエラーを示す場合、上記のいずれかの例外が送出されます。

Many of the following methods take an optional keyword-only argument *file*. When the *file* argument is supplied, it must be either a *file object* opened for binary writing, or the name of an on-disk file to be written to. The method will then write any data returned by the server (except for the response line and the terminating dot) to the file; any list of lines, tuples or objects that the method normally returns will be empty.

バージョン 3.2 で変更: 以下のメソッドの多くは改訂・修正されました。このため、3.1 のものとは互換性がありません。

NNTP.**quit**()
: QUIT 命令を送信し、接続を閉じます。このメソッドを呼び出した後は、NTTP オブジェクトの他のいかなるメソッドも呼び出してはいけません。

NNTP.**getwelcome**()
: サーバに最初に接続した際に送信される応答中のウェルカムメッセージを返します。(このメッセージには時に、ユーザにとって重要な免責事項や ヘルプ情報が入っています。)

NNTP.**getcapabilities**()
: Return the **RFC 3977** capabilities advertised by the server, as a *dict* instance mapping capability names to (possibly empty) lists of values. On legacy servers which don't understand the CAPABILITIES command, an empty dictionary is returned instead.

```
>>> s = NNTP('news.gmane.io')
>>> 'POST' in s.getcapabilities()
True
```

バージョン 3.2 で追加.

NNTP.**login**(*user=None*, *password=None*, *usenetrc=True*)
: Send AUTHINFO commands with the user name and password. If *user* and *password* are None and *usenetrc* is true, credentials from ~/.netrc will be used if possible.

Unless intentionally delayed, login is normally performed during the *NNTP* object initialization and separately calling this function is unnecessary. To force authentication to be delayed, you must not set *user* or *password* when creating the object, and must set *usenetrc* to False.

バージョン 3.2 で追加.

NNTP.**starttls**(*context=None*)
: Send a STARTTLS command. This will enable encryption on the NNTP connection. The *context* argument is optional and should be a *ssl.SSLContext* object. Please read **セキュリティで考慮すべき点** for best practices.

Note that this may not be done after authentication information has been transmitted, and authentication occurs by default if possible during a *NNTP* object initialization. See *NNTP.login()* for information on suppressing this behavior.

バージョン 3.2 で追加.

バージョン 3.4 で変更: このメソッドは *ssl.SSLContext.check_hostname* と *Server Name Indication* でホスト名のチェックをサポートしました。(*ssl.HAS_SNI* を参照してください)。

NNTP.**newgroups**(*date*, ***, *file=None*)

Send a `NEWGROUPS` command. The *date* argument should be a *datetime.date* or *datetime.datetime* object. Return a pair `(response, groups)` where *groups* is a list representing the groups that are new since the given *date*. If *file* is supplied, though, then *groups* will be empty.

```
>>> from datetime import date, timedelta
>>> resp, groups = s.newgroups(date.today() - timedelta(days=3))
>>> len(groups)
85
>>> groups[0]
GroupInfo(group='gmane.network.tor.devel', last='4', first='1', flag='m')
```

NNTP.**newnews**(*group*, *date*, ***, *file=None*)

Send a `NEWNEWS` command. Here, *group* is a group name or `'*'`, and *date* has the same meaning as for *newgroups()*. Return a pair `(response, articles)` where *articles* is a list of message ids.

このコマンドは NNTP サーバ管理者に無効にされていることがよくあります。

NNTP.**list**(*group_pattern=None*, ***, *file=None*)

Send a `LIST` or `LIST ACTIVE` command. Return a pair `(response, list)` where *list* is a list of tuples representing all the groups available from this NNTP server, optionally matching the pattern string *group_pattern*. Each tuple has the form `(group, last, first, flag)`, where *group* is a group name, *last* and *first* are the last and first article numbers, and *flag* usually takes one of these values:

- `y`: Local postings and articles from peers are allowed.
- `m`: The group is moderated and all postings must be approved.
- `n`: No local postings are allowed, only articles from peers.
- `j`: Articles from peers are filed in the junk group instead.
- `x`: No local postings, and articles from peers are ignored.
- `=foo.bar`: Articles are filed in the `foo.bar` group instead.

If *flag* has another value, then the status of the newsgroup should be considered unknown.

This command can return very large results, especially if *group_pattern* is not specified. It is best to cache the results offline unless you really need to refresh them.

バージョン 3.2 で変更: *group_pattern* が追加されました。

NNTP.**descriptions**(*grouppattern*)

Send a `LIST NEWSGROUPS` command, where *grouppattern* is a wildmat string as specified in **RFC 3977** (it's essentially the same as DOS or UNIX shell wildcard strings). Return a pair `(response, descriptions)`, where *descriptions* is a dictionary mapping group names to textual descriptions.

```
>>> resp, descs = s.descriptions('gmane.comp.python.*')
>>> len(descs)
295
>>> descs.popitem()
('gmane.comp.python.bio.general', 'BioPython discussion list (Moderated)')
```

NNTP.**description**(*group*)

単一のグループ *group* から説明文字列を取り出します。('group' が実際には wildmat 文字列で) 複数のグループがマッチした場合、最初にマッチしたものを返します。何もマッチしなければ空文字列を返します。

このメソッドはサーバからの応答コードを省略します。応答コードが必要なら、*descriptions()* を使ってください。

NNTP.**group**(*name*)

Send a `GROUP` command, where *name* is the group name. The group is selected as the current group, if it exists. Return a tuple `(response, count, first, last, name)` where *count* is the (estimated) number of articles in the group, *first* is the first article number in the group, *last* is the last article number in the group, and *name* is the group name.

NNTP.**over**(*message_spec*, ***, *file=None*)

Send an `OVER` command, or an `XOVER` command on legacy servers. *message_spec* can be either a string representing a message id, or a `(first, last)` tuple of numbers indicating a range of articles in the current group, or a `(first, None)` tuple indicating a range of articles starting from *first* to the last article in the current group, or *None* to select the current article in the current group.

Return a pair `(response, overviews)`. *overviews* is a list of `(article_number, overview)` tuples, one for each article selected by *message_spec*. Each *overview* is a dictionary with the same number of items, but this number depends on the server. These items are either message headers (the key is then the lower-cased header name) or metadata items (the key is then the metadata name prepended with `":"`). The following items are guaranteed to be present by the NNTP specification:

- the `subject`, `from`, `date`, `message-id` and `references` headers
- the `:bytes` metadata: the number of bytes in the entire raw article (including headers and body)
- the `:lines` metadata: the number of lines in the article body

The value of each item is either a string, or *None* if not present.

It is advisable to use the *decode_header()* function on header values when they may contain

non-ASCII characters:

```
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, overviews = s.over((last, last))
>>> art_num, over = overviews[0]
>>> art_num
117216
>>> list(over.keys())
['xref', 'from', ':lines', ':bytes', 'references', 'date', 'message-id', 'subject']
>>> over['from']
'=?UTF-8?B?Ik1hcnRpbiB2LiBMw7Z3aXMi?= <martin@v.loewis.de>'
>>> nntplib.decode_header(over['from'])
'"Martin v. Löwis" <martin@v.loewis.de>'
```

バージョン 3.2 で追加.

NNTP.**help**(**, file=None*)

Send a `HELP` command. Return a pair `(response, list)` where *list* is a list of help strings.

NNTP.**stat**(*message_spec=None*)

Send a `STAT` command, where *message_spec* is either a message id (enclosed in `'<'` and `'>'`) or an article number in the current group. If *message_spec* is omitted or *None*, the current article in the current group is considered. Return a triple `(response, number, id)` where *number* is the article number and *id* is the message id.

```
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, number, message_id = s.stat(first)
>>> number, message_id
(9099, '<20030112190404.GE29873@epoch.metaslash.com>')
```

NNTP.**next**()

`NEXT` 命令を送信します。*stat()* のような応答を返します。

NNTP.**last**()

`LAST` 命令を送信します。*stat()* のような応答を返します。

NNTP.**article**(*message_spec=None, *, file=None*)

Send an `ARTICLE` command, where *message_spec* has the same meaning as for *stat()*. Return a tuple `(response, info)` where *info* is a *namedtuple* with three attributes *number*, *message_id* and *lines* (in that order). *number* is the article number in the group (or 0 if the information is not available), *message_id* the message id as a string, and *lines* a list of lines (without terminating newlines) comprising the raw message including headers and body.

```
>>> resp, info = s.article('<20030112190404.GE29873@epoch.metaslash.com>')
>>> info.number
0
>>> info.message_id
'<20030112190404.GE29873@epoch.metaslash.com>'
>>> len(info.lines)
65
```

(次のページに続く)

(前のページからの続き)

```
>>> info.lines[0]
b'Path: main.gmane.org!not-for-mail'
>>> info.lines[1]
b'From: Neal Norwitz <neal@metaslash.com>'
>>> info.lines[-3:]
[b'There is a patch for 2.3 as well as 2.2.', b'', b'Neal']
```

NNTP.**head**(*message_spec=None*, *, *file=None*)

Same as *article()*, but sends a HEAD command. The *lines* returned (or written to *file*) will only contain the message headers, not the body.

NNTP.**body**(*message_spec=None*, *, *file=None*)

Same as *article()*, but sends a BODY command. The *lines* returned (or written to *file*) will only contain the message body, not the headers.

NNTP.**post**(*data*)

Post an article using the POST command. The *data* argument is either a *file object* opened for binary reading, or any iterable of bytes objects (representing raw lines of the article to be posted). It should represent a well-formed news article, including the required headers. The *post()* method automatically escapes lines beginning with . and appends the termination line.

If the method succeeds, the server's response is returned. If the server refuses posting, a *NNTPReplyError* is raised.

NNTP.**ihave**(*message_id*, *data*)

Send an IHAVE command. *message_id* is the id of the message to send to the server (enclosed in '<' and '>'). The *data* parameter and the return value are the same as for *post()*.

NNTP.**date**()

Return a pair (**response, date**). *date* is a *datetime* object containing the current date and time of the server.

NNTP.**slave**()

SLAVE 命令を送信します。サーバの *response* を返します。

NNTP.**set_debuglevel**(*level*)

インスタンスのデバッグレベルを設定します。このメソッドは印字されるデバッグ出力の量を制御します。標準では 0 に設定されていて、これはデバッグ出力を全く印字しません。1 はそこそこの量、一般に NNTP 要求や応答あたり 1 行のデバッグ出力を生成します。値が 2 やそれ以上の場合、(メッセージテキストを含めて) NNTP 接続上で送受信された全ての内容を一行ごとにログ出力する、最大限のデバッグ出力を生成します。

The following are optional NNTP extensions defined in **RFC 2980**. Some of them have been superseded by newer commands in **RFC 3977**.

NNTP.**xhdr**(*hdr*, *str*, *, *file=None*)

Send an XHDR command. The *hdr* argument is a header keyword, e.g. 'subject'. The *str* argument should have the form 'first-last' where *first* and *last* are the first and last article

numbers to search. Return a pair `(response, list)`, where *list* is a list of pairs `(id, text)`, where *id* is an article number (as a string) and *text* is the text of the requested header for that article. If the *file* parameter is supplied, then the output of the `XHDR` command is stored in a file. If *file* is a string, then the method will open a file with that name, write to it then close it. If *file* is a *file object*, then it will start calling `write()` on it to store the lines of the command output. If *file* is supplied, then the returned *list* is an empty list.

`NNTP.xover`(*start*, *end*, ***, *file=None*)

Send an `XOVER` command. *start* and *end* are article numbers delimiting the range of articles to select. The return value is the same of for *over()*. It is recommended to use *over()* instead, since it will automatically use the newer `OVER` command if available.

`NNTP.xpath`(*id*)

Return a pair `(resp, path)`, where *path* is the directory path to the article with message ID *id*. Most of the time, this extension is not enabled by NNTP server administrators.

バージョン 3.3 で非推奨: XPATH 拡張は積極的に使われません。

21.16.2 ユーティリティー関数

このモジュールは以下のユーティリティ関数も定義しています:

`nntplib.decode_header`(*header_str*)

Decode a header value, un-escaping any escaped non-ASCII characters. *header_str* must be a *str* object. The unescaped value is returned. Using this function is recommended to display some headers in a human readable form:

```
>>> decode_header("Some subject")
'Some subject'
>>> decode_header("=?ISO-8859-15?Q?D=E9buter_en_Python?=")
'Débuter en Python'
>>> decode_header("Re: =?UTF-8?B?cHJvYmzDqG1lIGRlIG1hdHJpY2U=?=")
'Re: problème de matrice'
```

21.17 smtplib --- SMTP プロトコルクライアント

ソースコード: Lib/smtplib.py

---

smtplib モジュールは、SMTP または ESMTP のリスナーデーモンを備えた任意のインターネット上のホストにメールを送るために使用することができる SMTP クライアント・セッション・オブジェクトを定義します。SMTP および ESMTP オペレーションの詳細は、**RFC 821** (Simple Mail Transfer Protocol) や **RFC 1869** (SMTP Service Extensions) を調べてください。

`class smtplib.SMTP`(*host=''*, *port=0*, *local_hostname=None*[, *timeout*], *source_address=None*)

SMTP インスタンスは SMTP 接続をカプセル化します。SMTP と ESMTP 操作の完全なレパート

リーをサポートするメソッドがあります。オプションパラメータの host および port が指定されている場合、SMTP の *connect()* メソッドは初期化中にこれらのパラメータを使って呼び出されます。*local_hostname* を指定した場合、それは HELO/EHLO コマンドのローカルホストの FQDN として使用されます。指定しない場合、ローカルホスト名は *socket.getfqdn()* を使用して検索されます。*connect()* 呼び出しが成功コード以外を返すと、*SMTPConnectError* が送出されます。オプションの *timeout* パラメータは、接続試行のようなブロッキング操作のタイムアウトを秒単位で指定します（指定されていない場合、グローバルなデフォルトのタイムアウト設定が使用されます）。タイムアウトが切れると、*socket.timeout* が送出されます。オプションの source_address パラメータを使用すると、複数のネットワークインターフェースを持つマシンの特定の送信元アドレス、かつ（または）特定の送信元 TCP ポートにバインドできます。接続する前にソケットを送信元アドレスとしてバインドするためにタプル (host, port) が必要です。省略された場合（あるいは、host または port がそれぞれ '' かつ/または 0 の場合）、OS のデフォルト動作が使用されます。

普通に使う場合は、初期化と接続を行ってから、*sendmail()* と *SMTP.quit()* メソッドを呼びます。使用例は先の方で記載しています。

The *SMTP* class supports the `with` statement. When used like this, the SMTP `QUIT` command is issued automatically when the `with` statement exits. E.g.:

```
>>> from smtplib import SMTP
>>> with SMTP("domain.org") as smtp:
...     smtp.noop()
...
(250, b'Ok')
>>>
```

引数 `self`, `data` を指定して **監査イベント** `smtplib.send` を送出します。

バージョン 3.3 で変更: `with` 構文のサポートが追加されました。

バージョン 3.3 で変更: source_address 引数が追加されました。

バージョン 3.5 で追加: SMTPUTF8 拡張 (**RFC 6531**) がサポートされました。

class smtplib.**SMTP_SSL**(*host=''*, *port=0*, *local_hostname=None*, *keyfile=None*, *certfile=None*[, *timeout*], *context=None*, *source_address=None*)

SMTP_SSL インスタンスは *SMTP* と全く同じように動作します。*SMTP_SSL* は、接続の始めから SSL が必要であり、`starttls()` が適切でない状況で使用するべきです。*host* が指定されていない場合、ローカルホストが使用されます。*port* が 0 の場合、標準の SMTP-over-SSL ポート（465）が使用されます。オプション引数 *local_hostname*, *timeout*, *source_address* は *SMTP* クラスと同じ意味を持ちます。*context* オプションは *SSLContext* を含むことができ、安全な接続のさまざまな側面を設定することができます。ベストプラクティスについては **セキュリティで考慮すべき点** を読んでください。

keyfile and *certfile* are a legacy alternative to *context*, and can point to a PEM formatted private key and certificate chain file for the SSL connection.

バージョン 3.3 で変更: *context* が追加されました。

バージョン 3.3 で変更: source_address 引数が追加されました。

バージョン 3.4 で変更: このクラスは *ssl.SSLContext.check_hostname* と *Server Name Indication* でホスト名のチェックをサポートしました。(*ssl.HAS_SNI* を参照してください)。

バージョン 3.6 で非推奨: *keyfile* および *certfile* は非推奨となったので、*context* を使ってください。代わりに *ssl.SSLContext.load_cert_chain()* を使うか、または *ssl.create_default_context()* にシステムが信頼する CA 証明書を選んでもらうかしてください。

class smtplib.LMTP(*host=''*, *port=LMTP_PORT*, *local_hostname=None*, *source_address=None*)

The LMTP protocol, which is very similar to ESMTP, is heavily based on the standard SMTP client. It's common to use Unix sockets for LMTP, so our `connect()` method must support that as well as a regular host:port server. The optional arguments local_hostname and source_address have the same meaning as they do in the *SMTP* class. To specify a Unix socket, you must use an absolute path for *host*, starting with a '/'.

認証は、通常の SMTP 機構を利用してサポートされています。Unix ソケットを利用する場合、LMTP は通常認証をサポートしたり要求したりはしません。しかし、あなたが必要であれば、利用することができます。

このモジュールの例外には次のものがあります:

exception smtplib.SMTPException

OSError の派生クラスで、このモジュールが提供する他の全ての例外の基底クラスです。

バージョン 3.4 で変更: SMTPException が *OSError* の派生クラスになりました。

exception smtplib.SMTPServerDisconnected

この例外はサーバが突然接続が切断されるか、*SMTP* インスタンスを生成する前に接続しようとした場合に送出されます。

exception smtplib.SMTPResponseException

SMTP のエラーコードを含んだ例外のクラスです。これらの例外は SMTP サーバがエラーコードを返すときに生成されます。エラーコードは `smtp_code` 属性に格納されます。また、`smtp_error` 属性にはエラーメッセージが格納されます。

exception smtplib.SMTPSenderRefused

送信者のアドレスが弾かれたときに送出される例外です。全ての *SMTPResponseException* 例外に、SMTP サーバが弾いた 'sender' アドレスの文字列がセットされます。

exception smtplib.SMTPRecipientsRefused

全ての受取人アドレスが弾かれたときに送出される例外です。各受取人のエラーは属性 `recipients` によってアクセス可能で、*SMTP.sendmail()* が返す辞書と同じ並びの辞書になっています。

exception smtplib.SMTPDataError

SMTP サーバが、メッセージのデータを受け入れることを拒絶したときに送出される例外です。

exception smtplib.SMTPConnectError

サーバへの接続時にエラーが発生したときに送出される例外です。

exception smtplib.SMTPHeloError

サーバーが HELO メッセージを弾いたときに送出される例外です。

exception smtplib.SMTPNotSupportedError
試行したコマンドやオプションはサーバにサポートされていません。

バージョン 3.5 で追加.

exception smtplib.SMTPAuthenticationError
SMTP 認証が失敗しました。最もあり得るのは、サーバーがユーザ名/パスワードのペアを受け付なかった事です。

参考:

RFC 821 - Simple Mail Transfer Protocol　SMTP のプロトコル定義です。このドキュメントでは SMTP のモデル、操作手順、プロトコルの詳細についてカバーしています。

RFC 1869 - SMTP Service Extensions　SMTP に対する ESMTP 拡張の定義です。このドキュメントでは、新たな命令による SMTP の拡張、サーバによって提供される命令を動的に発見する機能のサポート、およびいくつかの追加命令定義について記述しています。

21.17.1 SMTP オブジェクト

SMTP クラスインスタンスは次のメソッドを提供します:

SMTP.set_debuglevel(*level*)
デバッグ出力レベルを設定します。*level* が 1 や True の場合、接続ならびにサーバとの送受信のメッセージがデバッグメッセージとなります。*level* が 2 の場合、それらのメッセージにタイムスタンプが付きます。

バージョン 3.5 で変更: デバッグレベル 2 が追加されました。

SMTP.docmd(*cmd*, *args=''*)
サーバへコマンド *cmd* を送信します。オプション引数 *args* はスペース文字でコマンドに連結します。戻り値は、整数値のレスポンスコードと、サーバからの応答の値をタプルで返します (サーバからの応答が数行に渡る場合でも一つの大きな文字列で返します)。

数値応答コードと実際の応答行 (複数の応答は 1 つの長い行に結合されます) からなる 2 値タプルを返します。

通常、このメソッドを明示的に使う必要はありません。他のメソッドを実装するのに使い、自分で拡張したものをテストするのに役立つかもしれません。

応答待ちのときにサーバへの接続が切れると *SMTPServerDisconnected* が送出されます。

SMTP.connect(*host='localhost'*, *port=0*)
ホスト名とポート番号をもとに接続します。デフォルトは localhost の標準的な SMTP ポート (25 番) に接続します。もしホスト名の末尾がコロン (':') で、後に番号がついている場合は、「ホスト名:ポート番号」として扱われます。このメソッドはコンストラクタにホスト名及びポート番号が指定されてい

る場合、自動的に呼び出されます。戻り値は、この接続の応答内でサーバによって送信された応答コードとメッセージの 2 要素タプルです。

引数 self, host, port を指定して 監査イベント smtplib.connect を送出します。

SMTP.**helo**(*name=''*)

SMTP サーバに HELO コマンドで身元を示します。デフォルトでは hostname 引数はローカルホストを指します。サーバーが返したメッセージは、オブジェクトの helo_resp 属性に格納されます。

通常は *sendmail()* が呼びだすため、これを明示的に呼び出す必要はありません。

SMTP.**ehlo**(*name=''*)

EHLO を利用し、ESMTP サーバに身元を明かします。デフォルトでは hostname 引数はローカルホストの FQDN です。また、ESMTP オプションのために応答を調べたものは、*has_extn()* に備えて保存されます。また、幾つかの情報を属性に保存します: サーバーが返したメッセージは ehlo_resp 属性に、does_esmtp 属性はサーバーが ESMTP をサポートしているかどうかによって true か false に、esmtp_features 属性は辞書で、サーバーが対応している SMTP サービス拡張の名前と、もしあればそのパラメータを格納します。

メールを送信する前に *has_extn()* を使おうとしない限り、このメソッドを明示的に呼ぶ必要はないはずです。必要な場合は *sendmail()* に暗黙的に呼ばれます。

SMTP.**ehlo_or_helo_if_needed**()

このメソッドは、現在のセッションでまだ EHLO か HELO コマンドが実行されていない場合、*ehlo()* and/or *helo()* メソッドを呼び出します。このメソッドは先に ESMTP EHLO を試します。

SMTPHeloError サーバーが HELO に正しく返答しませんでした。

SMTP.**has_extn**(*name*)

name が拡張 SMTP サービスセットに含まれている場合には *True* を返し、そうでなければ *False* を返します。大小文字は区別されません。

SMTP.**verify**(*address*)

VRFY を利用して SMTP サーバにアドレスの妥当性をチェックします。妥当である場合はコード 250 と完全な **RFC 822** アドレス (人名を含む) のタプルを返します。それ以外の場合は、400 以上のエラーコードとエラー文字列を返します。

注釈: ほとんどのサイトはスパマーの裏をかくために SMTP の VRFY は使用不可になっています。

SMTP.**login**(*user*, *password*, ***, *initial_response_ok=True*)

認証が必要な SMTP サーバにログインします。認証に使用する引数はユーザ名とパスワードです。まだセッションが無い場合は、EHLO または HELO コマンドでセッションを作ります。ESMTP の場合は EHLO が先に試されます。認証が成功した場合は通常このメソッドは戻りますが、例外が起こった場合は以下の例外が上がります:

SMTPHeloError サーバーが HELO に正しく返答しませんでした。

SMTPAuthenticationError サーバがユーザ名/パスワードでの認証に失敗しました。

SMTPNotSupportedError AUTH コマンドはサーバにサポートされていません。

SMTPException 適当な認証方法が見付かりませんでした。

Each of the authentication methods supported by *smtplib* are tried in turn if they are advertised as supported by the server. See *auth()* for a list of supported authentication methods. *initial_response_ok* is passed through to *auth()*.

オプションのキーワード引数 *initial_response_ok* は、それをサポートする認証方法に対して、チャレンジ/レスポンスを要求するのではなく、"AUTH" コマンドとともに **RFC 4954** で指定された" 初期応答" を送信できるかどうかを指定します。

バージョン 3.5 で変更: *SMTPNotSupportedError* が送出される場合があります。*initial_response_ok* 引数が追加されました。

SMTP.**auth**(*mechanism*, *authobject*, ***, *initial_response_ok=True*)

Issue an SMTP AUTH command for the specified authentication *mechanism*, and handle the challenge response via *authobject*.

mechanism specifies which authentication mechanism is to be used as argument to the AUTH command; the valid values are those listed in the auth element of esmtp_features.

authobject must be a callable object taking an optional single argument:

data = authobject(challenge=None)

If optional keyword argument *initial_response_ok* is true, authobject() will be called first with no argument. It can return the **RFC 4954** "initial response" ASCII str which will be encoded and sent with the AUTH command as below. If the authobject() does not support an initial response (e.g. because it requires a challenge), it should return None when called with challenge=None. If *initial_response_ok* is false, then authobject() will not be called first with None.

If the initial response check returns None, or if *initial_response_ok* is false, authobject() will be called to process the server's challenge response; the *challenge* argument it is passed will be a bytes. It should return ASCII str *data* that will be base64 encoded and sent to the server.

The SMTP class provides authobjects for the CRAM-MD5, PLAIN, and LOGIN mechanisms; they are named SMTP.auth_cram_md5, SMTP.auth_plain, and SMTP.auth_login respectively. They all require that the user and password properties of the SMTP instance are set to appropriate values.

User code does not normally need to call auth directly, but can instead call the *login()* method, which will try each of the above mechanisms in turn, in the order listed. auth is exposed to facilitate the implementation of authentication methods not (or not yet) supported directly by *smtplib*.

バージョン 3.5 で追加.

SMTP.**starttls**(*keyfile=None*, *certfile=None*, *context=None*)

TLS (Transport Layer Security) モードで SMTP 接続します。続く全ての SMTP コマンドは暗号化されます。その後、もう一度 *ehlo()* を呼んでください。

If *keyfile* and *certfile* are provided, they are used to create an *ssl.SSLContext*.

Optional *context* parameter is an *ssl.SSLContext* object; This is an alternative to using a keyfile and a certfile and if specified both *keyfile* and *certfile* should be `None`.

もしまだ `EHLO` か `HELO` コマンドが実行されていない場合、このメソッドは ESMTP `EHLO` を先に試します。

バージョン 3.6 で非推奨: *keyfile* および *certfile* は非推奨となったので、*context* を使ってください。代わりに *ssl.SSLContext.load_cert_chain()* を使うか、または *ssl.create_default_context()* にシステムが信頼する CA 証明書を選んでもらうかしてください。

SMTPHeloError サーバーが `HELO` に正しく返答しませんでした。

SMTPNotSupportedError サーバーが STARTTLS 拡張に対応していません。

RuntimeError 実行中の Python インタプリタで、SSL/TLS サポートが利用できません。

バージョン 3.3 で変更: *context* が追加されました。

バージョン 3.4 で変更: このメソッドは `SSLContext.check_hostname` と *Server Name Indicator* でホスト名のチェックをサポートしました。(*HAS_SNI* を参照してください)。

バージョン 3.5 で変更: The error raised for lack of STARTTLS support is now the *SMTPNotSupportedError* subclass instead of the base *SMTPException*.

`SMTP.sendmail`(*from_addr*, *to_addrs*, *msg*, *mail_options=()*, *rcpt_options=()*)

メールを送信します。必要な引数は **RFC 822** の from アドレス文字列、**RFC 822** の to アドレス文字列またはアドレス文字列のリスト、メッセージ文字列です。送信側は `MAIL FROM` コマンドで使用される *mail_options* の ESMTP オプション (`8bitmime` のような) のリストを得るかもしれません。全ての `RCPT` コマンドで使われるべき ESMTP オプション (例えば `DSN` コマンド) は、*rcpt_options* を通して利用することができます。(もし送信先別に ESMTP オプションを使う必要があれば、メッセージを送るために `mail()` 、`rcpt()` 、`data()` といった下位レベルのメソッドを使う必要があります。)

注釈: 配送エージェントは *from_addr*、*to_addrs* 引数を使い、メッセージのエンベロープを構成します。`sendmail` はメッセージヘッダを修正しません。

---

msg may be a string containing characters in the ASCII range, or a byte string. A string is encoded to bytes using the ascii codec, and lone `\r` and `\n` characters are converted to `\r\n` characters. A byte string is not modified.

まだセッションが無い場合は、`EHLO` または `HELO` コマンドでセッションを作ります。ESMTP の場合は `EHLO` が先に試されます。また、サーバが ESMTP 対応ならば、メッセージサイズとそれぞれ指定されたオプションも渡します。(feature オプションがあればサーバの広告をセットします) `EHLO` が失敗した場合は、ESMTP オプションの無い `HELO` が試されます。

このメソッドは最低でも 1 人の受信者にメールが受け取られたときは正常に戻ります。そうでない場合は例外を投げます。つまり、このメソッドが例外を送出しないときは誰かが送信したメールを受け取っ

たはずです。また、例外を送出しない場合は拒絶された受信者ごとに項目のある辞書を返します。各項目はサーバーが送った SMTP エラーコードと付随するエラーメッセージのタプルを持ちます。

mail_options に SMTPUTF8 があり、サーバがサポートしている場合は、*from_addr* と *to_addrs* に非 ASCII 文字を含めることが出来ます。

このメソッドは次の例外を送出することがあります:

SMTPRecipientsRefused 全ての受信を拒否され、誰にもメールが届けられませんでした。例外オブジェクトの recipients 属性は、受信拒否についての情報の入った辞書オブジェクトです。(辞書は少なくとも一つは受信されたときに似ています)。

SMTPHeloError サーバーが HELO に正しく返答しませんでした。

SMTPSenderRefused サーバが *from_addr* を受理しませんでした。

SMTPDataError サーバが予期しないエラーコードを返しました (受信拒否以外)。

SMTPNotSupportedError *mail_options* に SMTPUTF8 が与えられましたが、サーバがサポートしていません。

また、この他の注意として、例外が上がった後もコネクションは開いたままになっています。

バージョン 3.2 で変更: *msg* はバイト文字列でも構いません。

バージョン 3.5 で変更: SMTPUTF8 のサポートが追加されました。SMTPUTF8 が与えられたが、サーバがサポートしていない場合は *SMTPNotSupportedError* が送出されます。

SMTP.**send_message**(*msg*, *from_addr=None*, *to_addrs=None*, *mail_options=()*, *rcpt_options=()*)

This is a convenience method for calling *sendmail()* with the message represented by an *email.message.Message* object. The arguments have the same meaning as for *sendmail()*, except that *msg* is a Message object.

If *from_addr* is None or *to_addrs* is None, send_message fills those arguments with addresses extracted from the headers of *msg* as specified in **RFC 5322**: *from_addr* is set to the *Sender* field if it is present, and otherwise to the *From* field. *to_addrs* combines the values (if any) of the *To*, *Cc*, and *Bcc* fields from *msg*. If exactly one set of *Resent-** headers appear in the message, the regular headers are ignored and the *Resent-** headers are used instead. If the message contains more than one set of *Resent-** headers, a *ValueError* is raised, since there is no way to unambiguously detect the most recent set of *Resent-* headers.

send_message serializes *msg* using *BytesGenerator* with \r\n as the *linesep*, and calls *sendmail()* to transmit the resulting message. Regardless of the values of *from_addr* and *to_addrs*, send_message does not transmit any *Bcc* or *Resent-Bcc* headers that may appear in *msg*. If any of the addresses in *from_addr* and *to_addrs* contain non-ASCII characters and the server does not advertise SMTPUTF8 support, an SMTPNotSupported error is raised. Otherwise the Message is serialized with a clone of its *policy* with the *utf8* attribute set to True, and SMTPUTF8 and BODY=8BITMIME are added to *mail_options*.

バージョン 3.2 で追加.

バージョン 3.5 で追加: 国際化アドレスのサポート (SMTPUTF8)。

SMTP.quit()
SMTP セッションを終了し、接続を閉じます。SMTP QUIT コマンドの結果を返します。

下位レベルのメソッドは標準 SMTP/ESMTP コマンド HELP 、RSET 、NOOP 、MAIL 、RCPT 、DATA に対応しています。通常これらは直接呼ぶ必要はなく、また、ドキュメントもありません。詳細はモジュールのコードを調べてください。

21.17.2 SMTP 使用例

次の例は最低限必要なメールアドレス ('To' と 'From') を含んだメッセージを送信するものです。この例では **RFC 822** ヘッダの加工もしていません。メッセージに含まれるヘッダは、メッセージに含まれる必要があり、特に、明確な 'To'、と 'From' アドレスはメッセージヘッダに含まれている必要があります。

```
import smtplib

def prompt(prompt):
    return input(prompt).strip()

fromaddr = prompt("From: ")
toaddrs  = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")

# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\n\r\n"
       % (fromaddr, ", ".join(toaddrs)))
while True:
    try:
        line = input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line

print("Message length is", len(msg))

server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

注釈: 多くの場合、*email* パッケージの機能を使って email メッセージを構築し、それを、*send_message()* で送信する、という手順を用います。*email:* **使用例** を参照してください。

21.18 smtpd --- SMTP サーバー

ソースコード: Lib/smtpd.py

このモジュールは SMTP (email) サーバを実装するためのいくつかのクラスを提供しています。

参考:

The aiosmtpd package is a recommended replacement for this module. It is based on *asyncio* and provides a more straightforward API. *smtpd* should be considered deprecated.

サーバの実装がいくつかあります。一つはジェネリックで何もしない実装で、オーバーライドすることが出来ます。他の二つは特定のメール送信方策を提供しています。

また、SMTPChannel を拡張して SMTP クライアントとの特定の相互作用挙動を実装することができます。

コードは **RFC 5321** に加え、**RFC 1870** SIZE と **RFC 6531** SMTPUTF8 拡張をサポートしています。

21.18.1 SMTPServer オブジェクト

class smtpd.SMTPServer(*localaddr*, *remoteaddr*, *data_size_limit=33554432*, *map=None*, *enable_SMTPUTF8=False*, *decode_data=False*)

新たな *SMTPServer* オブジェクトを作成し、それをローカルのアドレス *localaddr* に関連づけ (bind) ます。このオブジェクトは *remoteaddr* を上流の SMTP リレー先とします。*localaddr* と *remoteaddr* のどちらも *(host, port)* タプルである必要があります。このクラスは *asyncore.dispatcher* を継承しており、インスタンス化時に自身を *asyncore* のイベントループに登録します。

data_size_limit には `DATA` コマンドが受け取る最大のバイト数を指定します。`None` や `0` の場合上限はありません。

map is the socket map to use for connections (an initially empty dictionary is a suitable value). If not specified the *asyncore* global socket map is used.

enable_SMTPUTF8 determines whether the `SMTPUTF8` extension (as defined in **RFC 6531**) should be enabled. The default is `False`. When `True`, `SMTPUTF8` is accepted as a parameter to the `MAIL` command and when present is passed to *process_message()* in the `kwargs['mail_options']` list. *decode_data* and *enable_SMTPUTF8* cannot be set to `True` at the same time.

decode_data specifies whether the data portion of the SMTP transaction should be decoded using UTF-8. When *decode_data* is `False` (the default), the server advertises the `8BITMIME` extension (**RFC 6152**), accepts the `BODY=8BITMIME` parameter to the `MAIL` command, and when present passes it to *process_message()* in the `kwargs['mail_options']` list. *decode_data* and *enable_SMTPUTF8* cannot be set to `True` at the same time.

process_message(*peer*, *mailfrom*, *rcpttos*, *data*, ***kwargs*)

このクラスでは *NotImplementedError* 例外を送出します。受信したメッセージを使って何か意味のある処理をしたい場合にはこのメソッドをオーバライドしてください。コンストラクタの *remoteaddr* に渡した値は `_remoteaddr` 属性で参照できます。*peer* はリモートホストのアドレス

で、*mailfrom* はメッセージエンベロープの発信元 (envelope originator) 、*rcpttos* はメッセージエンベロープの受信対象、そして *data* は電子メールの内容が入った (**RFC 5321** 形式の) 文字列です。

decode_data コンストラクタ引数が `True` の場合、*data* 引数はユニコード文字列です。`False` の場合は bytes オブジェクトです。

kwargs is a dictionary containing additional information. It is empty if `decode_data=True` was given as an init argument, otherwise it contains the following keys:

mail_options: `MAIL` コマンドが受け取る全ての引数のリストです (要素は大文字の文字列です; 例えば `['BODY=8BITMIME', 'SMTPUTF8']`)。

rcpt_options: `RCPT` コマンドのものである点以外は *mail_options* と同じです。今のところ `RCPT TO` オプションはサポートされていないため、これは常に空のリストです。

将来の仕様改善によって kwargs 辞書にキーが追加される可能性があるため、`process_message` の実装で追加のキーワード引数を受け取るには `**kwargs` シグニチャを使うべきです。

通常の `250 Ok` 応答には `None` を返します。そうでない場合求められる応答を **RFC 5321** 形式の文字列で返します。

channel_class

これを派生クラスでオーバーライドすることで、SMTP クライアントを管理するのにカスタムの *SMTPChannel* を使います。

バージョン 3.4 で追加: *map* コンストラクタ引数。

バージョン 3.5 で変更: *localaddr* および *remoteaddr* は IPv6 アドレスを持てるようになりました。

バージョン 3.5 で追加: The *decode_data* and *enable_SMTPUTF8* constructor parameters, and the *kwargs* parameter to *process_message()* when *decode_data* is `False`.

バージョン 3.6 で変更: *decode_data* がデフォルトで `False` になりました。

21.18.2 DebuggingServer オブジェクト

class smtpd.**DebuggingServer**(*localaddr*, *remoteaddr*)

新たなデバッグ用サーバを生成します。引数は *SMTPServer* と同じです。メッセージが届いても無視し、標準出力に出力します。

21.18.3 PureProxy オブジェクト

class smtpd.**PureProxy**(*localaddr*, *remoteaddr*)

新たな単純プロキシ (pure proxy) サーバを生成します。引数は *SMTPServer* と同じです。全てのメッセージを *remoteaddr* にリレーします。このオブジェクトを動作させるとオープンリレーを作成してしまう可能性が多分にあります。注意してください。

21.18.4 MailmanProxy Objects

class smtpd.**MailmanProxy**(*localaddr*, *remoteaddr*)

新たな単純プロキシサーバを生成します。引数は *SMTPServer* と同じです。全てのメッセージを *remoteaddr* にリレーしますが、ローカルの mailman の設定に *remoteaddr* がある場合には mailman を使って処理します。このオブジェクトを動作させるとオープンリレーを 作成してしまう可能性が多分にあります。注意してください。

21.18.5 SMTPChannel オブジェクト

class smtpd.**SMTPChannel**(*server*, *conn*, *addr*, *data_size_limit=33554432*, *map=None*, *enable_SMTPUTF8=False*, *decode_data=False*)

サーバと一つの SMTP クライアント間の通信を管理する *SMTPChannel* オブジェクトを新たに生成します。

conn and *addr* are as per the instance variables described below.

data_size_limit には `DATA` コマンドが受け取る最大のバイト数を指定します。`None` や `0` の場合上限はありません。

enable_SMTPUTF8 determines whether the `SMTPUTF8` extension (as defined in **RFC 6531**) should be enabled. The default is `False`. *decode_data* and *enable_SMTPUTF8* cannot be set to `True` at the same time.

A dictionary can be specified in *map* to avoid using a global socket map.

decode_data specifies whether the data portion of the SMTP transaction should be decoded using UTF-8. The default is `False`. *decode_data* and *enable_SMTPUTF8* cannot be set to `True` at the same time.

To use a custom SMTPChannel implementation you need to override the *SMTPServer.channel_class* of your *SMTPServer*.

バージョン 3.5 で変更: 引数 *decode_data* と *enable_SMTPUTF8* が追加されました.

バージョン 3.6 で変更: *decode_data* がデフォルトで `False` になりました。

SMTPChannel は以下のインスタンス変数を持っています:

smtp_server
このチャンネルを生成した *SMTPServer* を保持します。

conn
クライアントに接続しているソケットオブジェクトを保持します。

addr
Holds the address of the client, the second value returned by *socket.accept*

received_lines
Holds a list of the line strings (decoded using UTF-8) received from the client. The lines have their `"\r\n"` line ending translated to `"\n"`.

smtp_state
Holds the current state of the channel. This will be either `COMMAND` initially and then `DATA` after the client sends a "DATA" line.

seen_greeting
Holds a string containing the greeting sent by the client in its "HELO".

mailfrom
Holds a string containing the address identified in the "MAIL FROM:" line from the client.

rcpttos
Holds a list of strings containing the addresses identified in the "RCPT TO:" lines from the client.

received_data
Holds a string containing all of the data sent by the client during the DATA state, up to but not including the terminating `"\r\n.\r\n"`.

fqdn
Holds the fully-qualified domain name of the server as returned by *socket.getfqdn()*.

peer
Holds the name of the client peer as returned by `conn.getpeername()` where `conn` is *conn*.

The *SMTPChannel* operates by invoking methods named `smtp_<command>` upon reception of a command line from the client. Built into the base *SMTPChannel* class are methods for handling the following commands (and responding to them appropriately):

コマンド	行う動作
HELO	クライアントのグリーティングを受け取り *seen_greeting* に格納します。サーバを基本コマンドモードに設定します。
EHLO	クライアントのグリーティングを受け取り *seen_greeting* に格納します。サーバを拡張コマンドモードに設定します。
NOOP	何もしません。
QUIT	接続をきれいに閉じます。
MAIL	"MAIL FROM:" シンタックスを受け取り提供されたアドレスを *mailfrom* として保存します。拡張コマンドモードでは **RFC 1870** SIZE 属性を受け取り *data_size_limit* の値に基づき適切に応答します。
RCPT	"RCPT TO:" シンタックスを受け取り提供されたアドレスを *rcpttos* リストに格納します。
RSET	*mailfrom*, *rcpttos*, *received_data* をリセットしますが、グリーティングはリセットしません。
DATA	内部状態を `DATA` に設定し、クライアントからの残りの行を終端子 `"\r\n.\r\n"` を受け取るまで *received_data* に格納します。
HELP	最小の情報をコマンドシンタックスで返します。
VRFY	コード 252 (サーバはアドレスが有効か分かりません) を返します。
EXPN	コマンドが実装されていないことを報告します。

21.19 `telnetlib` --- Telnet クライアント

ソースコード: Lib/telnetlib.py

---

telnetlib モジュールでは、Telnet プロトコルを実装している *Telnet* クラスを提供します。Telnet プロトコルについての詳細は **RFC 854** を参照してください。加えて、このモジュールでは Telnet プロトコルにおける制御文字 (下を参照してください) と、telnet オプションに対するシンボル定数を提供しています。telnet オプションに対するシンボル名は `arpa/telnet.h` の `TELOPT_` がない状態での定義に従います。伝統的に `arpa/telnet.h` に含められていない telnet オプションのシンボル名については、このモジュールのソースコード自体を参照してください。

telnet コマンドのシンボル定数は、IAC、DONT、DO、WONT、WILL、SE (サブネゴシエーション終了)、NOP (何もしない)、DM (データマーク)、BRK (ブレーク)、IP (プロセス中断)、AO (出力停止)、AYT (応答確認)、EC (文字削除)、EL (行削除)、GA (進め)、SB (サブネゴシエーション開始) です。

class `telnetlib.Telnet`(*host=None*, *port=0*[, *timeout*])

Telnet は Telnet サーバへの接続を表現します。デフォルトでは、*Telnet* クラスのインスタンスは最初はサーバに接続していません。接続を確立するには *open()* を使わなければなりません。別の方法として、コンストラクタにホスト名とオプションのポート番号を渡すこともできます。この場合はコンストラクタの呼び出しが返る以前にサーバへの接続が確立されます。オプション引数の *timeout* が渡さ

れた場合、コネクション接続時のタイムアウト時間を秒数で指定します (指定されなかった場合は、グローバルのデフォルトタイムアウト設定が利用されます)。

すでに接続の開かれているンスタンスを再度開いてはいけません。

このクラスは多くの `read_*()` メソッドを持っています。これらのメソッドのいくつかは、接続の終端を示す文字を読み込んだ場合に *EOFError* を送出するので注意してください。例外を送出するのは、これらの関数が終端に到達しなくても空の文字列を返す可能性があるからです。詳しくは下記の個々の説明を参照してください。

Telnet オブジェクトはコンテキストマネージャであり、`with` 文で使えます。`with` ブロックから抜けるとき、*close()* メソッドが呼び出されます:

```
>>> from telnetlib import Telnet
>>> with Telnet('localhost', 23) as tn:
...     tn.interact()
...
```

バージョン 3.6 で変更: コンテキストマネージャーサポートが追加されました。

参考:

RFC 854 - Telnet **プロトコル仕様**　Telnet プロトコルの定義。

21.19.1 Telnet オブジェクト

Telnet インスタンスは以下のメソッドを持っています:

`Telnet.read_until(`*expected*, *timeout=None*`)`

expected で指定されたバイト文字列を読み込むか、*timeout* で指定された秒数が経過するまで読み込みます。

与えられた文字列に一致する部分が見つからなかった場合、読み込むことができたもの全てを返します。これは空のバイト列になる可能性があります。接続が閉じられ、転送処理済みのデータが得られない場合には *EOFError* が送出されます。

`Telnet.read_all()`

EOF に到達するまでの全てのデータをバイト列として読み込みます; 接続が閉じられるまでブロックします。

`Telnet.read_some()`

EOF に到達しない限り、少なくとも 1 バイトの転送処理済みデータを読み込みます。EOF に到達した場合は `b''` を返します。すぐに読み出せるデータが存在しない場合にはブロックします。

`Telnet.read_very_eager()`

I/O によるブロックを起こさずに読み出せる全てのデータを読み込みます (eager モード)。

接続が閉じられており、転送処理済みのデータとして読み出せるものがない場合には *EOFError* が送出されます。それ以外の場合で、単に読み出せるデータがない場合には `b''` を返します。IAC シーケンス操作中でないかぎりブロックしません。

Telnet.**read_eager**()
現在すぐに読み出せるデータを読み出します。

接続が閉じられており、転送処理済みのデータとして読み出せるものがない場合には *EOFError* が送出されます。それ以外の場合で、単に読み出せるデータがない場合には `b''` を返します。IAC シーケンス操作中でないかぎりブロックしません。

Telnet.**read_lazy**()
すでにキューに入っているデータを処理して返します (lazy モード)。

接続が閉じられており、読み出せるデータがない場合には *EOFError* を送出します。それ以外の場合で、転送処理済みのデータで読み出せるものがない場合には `b''` を返します。IAC シーケンス操作中でないかぎりブロックしません。

Telnet.**read_very_lazy**()
すでに処理済みキューに入っているデータを処理して返します (very lazy モード)。

接続が閉じられており、読み出せるデータがない場合には *EOFError* を送出します。それ以外の場合で、転送処理済みのデータで読み出せるものがない場合には `b''` を返します。このメソッドは決してブロックしません。

Telnet.**read_sb_data**()
SB/SE ペア (サブオプション開始／終了) の間に収集されたデータを返します。`SE` コマンドによって起動されたコールバック関数はこれらのデータにアクセスしなければなりません。このメソッドはけっしてブロックしません。

Telnet.**open**(*host*, *port=0*[, *timeout*])
サーバホストに接続します。第二引数はオプションで、ポート番号を指定します。標準の値は通常の Telnet ポート番号 (23) です。オプション引数の *timeout* が渡された場合、コネクション接続時などのブロックする操作のタイムアウト時間を秒数で指定します (指定されなかった場合は、グローバルのデフォルトタイムアウト設定が利用されます)。

すでに接続しているインスタンスで再接続を試みてはいけません。

引数 `self`, `host`, `port` 付きで **監査イベント** `telnetlib.Telnet.open` を送出します。

Telnet.**msg**(*msg*, **args*)
デバッグレベルが `>` 0 のとき、デバッグ用のメッセージを出力します。追加の引数が存在する場合、標準の文字列書式化演算子 `%` を使って *msg* 中の書式指定子に代入されます。

Telnet.**set_debuglevel**(*debuglevel*)
デバッグレベルを設定します。*debuglevel* が大きくなるほど、(`sys.stdout` に) デバッグメッセージがたくさん出力されます。

Telnet.**close**()
コネクションをクローズします。

Telnet.**get_socket**()
内部的に使われているソケットオブジェクトです。

Telnet.fileno()
内部的に使われているソケットオブジェクトのファイル記述子です。

Telnet.write(*buffer*)
ソケットにバイト文字列を書き込みます。このとき IAC 文字については 2 度送信します。接続がブロックした場合、書き込みがブロックする可能性があります。接続が閉じられた場合、*OSError* が送出されるかもしれません。

引数 self, buffer 付きで **監査イベント** telnetlib.Telnet.write を送出します。

バージョン 3.3 で変更: このメソッドは以前は *socket.error* を投げていましたが、これは現在では *OSError* の別名になっています。

Telnet.interact()
非常に低機能の telnet クライアントをエミュレートする対話関数です。

Telnet.mt_interact()
interact() のマルチスレッド版です。

Telnet.expect(*list*, *timeout=None*)
正規表現のリストのうちどれか一つにマッチするまでデータを読みます。

第一引数は正規表現のリストです。コンパイルされたもの (*regex* **オブジェクト**) でも、コンパイルされていないもの (バイト文字列) でもかまいません。オプションの第二引数はタイムアウトで、単位は秒です。標準の値は無期限に設定されています。

3 つの要素からなるタプル: 最初にマッチした正規表現のインデクス; 返されたマッチオブジェクト; マッチ部分を含む、マッチするまでに読み込まれたバイト列、を返します。

ファイル終了子が見つかり、かつ何もバイト列が読み込まれなかった場合、*EOFError* が送出されます。そうでない場合で何もマッチしなかった場合には (-1, None, data) が返されます。ここで *data* はこれまで受信したバイト列です (タイムアウトが発生した場合には空のバイト列になる場合もあります)。

正規表現の末尾が (.* のような) 貪欲マッチングになっている場合や、入力に対して 1 つ以上の正規表現がマッチする場合には、その結果は決定不能で、I/O のタイミングに依存するでしょう。

Telnet.set_option_negotiation_callback(*callback*)
telnet オプションが入力フローから読み込まれるたびに、*callback* が (設定されていれば) 以下の引数形式: callback(telnet socket, command (DO/DONT/WILL/WONT), option) で呼び出されます。その後 telnet オプションに対しては telnetlib は何も行いません。

21.19.2 Telnet Example

典型的な使用例のサンプルコード:

```
import getpass
import telnetlib

HOST = "localhost"
user = input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)

tn.read_until(b"login: ")
tn.write(user.encode('ascii') + b"\n")
if password:
    tn.read_until(b"Password: ")
    tn.write(password.encode('ascii') + b"\n")

tn.write(b"ls\n")
tn.write(b"exit\n")

print(tn.read_all().decode('ascii'))
```

21.20 uuid --- RFC 4122 に基づく UUID オブジェクト

ソースコード: Lib/uuid.py

このモジュールでは immutable (変更不能) な *UUID* オブジェクト (*UUID* クラス) と **RFC 4122** の定めるバージョン 1、3、4、5 の UUID を生成するための *uuid1()*, *uuid3()*, *uuid4()*, *uuid5()*, が提供されています。

もしユニークな ID が必要なだけであれば、おそらく *uuid1()* か *uuid4()* を呼び出せば良いでしょう。*uuid1()* はコンピュータのネットワークアドレスを含む UUID を生成するためにプライバシーを侵害するかもしれない点に注意してください。*uuid4()* はランダムな UUID を生成します。

使用しているプラットフォームによっては、*uuid1()* 関数は安全でない UUID を返す可能性があります。ここでいう「安全な」UUID とは、同期的に生成され、異なるプロセスに同じ UUID が与えられることがないものを言います。*UUID* のすべてのインスタンスは:attr:*is_safe* 属性を持ち、次の項目を使用します:

class uuid.SafeUUID

バージョン 3.7 で追加.

safe

UUID は並列処理に対して安全なプラットフォームで生成された

unsafe

UUID は並列処理に対して安全なプラットフォームで生成されなかった

unknown
プラットフォームは、UUID が安全に生成されたかどうかの情報を提供しなかった

class uuid.UUID(*hex=None*, *bytes=None*, *bytes_le=None*, *fields=None*, *int=None*, *version=None*, ***, *is_safe=SafeUUID.unknown*)
32 桁の 16 進数文字列、*bytes* に 16 バイトのビッグエンディアンの文字列、*bytes_le* 引数に 16 バイトのリトルエンディアンの文字列、*fields* 引数に 6 つの整数のタプル（32 ビット *time_low*, 16 ビット *time_mid*, 16 ビット *time_hi_version*, 8 ビット *clock_seq_hi_variant*, 8 ビット *clock_seq_low*, 48 ビット *node* ）、または *int* に一つの 128 ビット整数のいずれかから UUID を生成します。16 進数が与えられた時、波括弧、ハイフン、それと URN 接頭辞は無視されます。例えば、これらの表現は全て同じ UUID を払い出します:

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes=b'\x12\x34\x56\x78'*4)
UUID(bytes_le=b'\x78\x56\x34\x12\x34\x12\x78\x56' +
              b'\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x567812345678))
UUID(int=0x12345678123456781234567812345678)
```

hex, *bytes*, *bytes_le*, *fields*, または *int* のうち、どれかただ一つだけが与えられなければいけません。*version* 引数はオプションです; 与えられた場合、結果の UUID は与えられた *hex*, *bytes*, *bytes_le*, *fields*, または *int* をオーバーライドして、**RFC 4122** に準拠した variant と version ナンバーのセットを持つことになります。*bytes_le*, *fields*, or *int*.

2 つの UUID オブジェクトの比較は、*UUID.int* 属性の比較によって行われます。UUID ではないオブジェクトとの比較は、*TypeError* を送出します。

`str(uuid)` は 32 桁の 16 進で UUID を表す “12345678-1234-5678-1234-567812345678“ 形式の文字列を返します。

UUID インスタンスは以下の読み出し専用属性を持ちます:

UUID.bytes
16 バイト文字列（バイトオーダーがビッグエンディアンの 6 つの整数フィールドを持つ）の UUID。

UUID.bytes_le
16 バイト文字列（ *time_low*, *time_mid*, *time_hi_version* をリトルエンディアンで持つ）の UUID。

UUID.fields
UUID の 6 つの整数フィールドを持つタプルで、これは 6 つの個別の属性と 2 つの派生した属性としても取得可能です:

フィールド	意味
time_low	UUID の最初の 32 ビット
time_mid	UUID の次の 16 ビット
time_hi_version	UUID の次の 16 ビット
clock_seq_hi_variant	UUID の次の 8 ビット
clock_seq_low	UUID の次の 8 ビット
node	UUID の最後の 48 ビット
time	60 ビットのタイムスタンプ
clock_seq	14 ビットのシーケンス番号

UUID.hex

32 文字の 16 進数文字列での UUID。

UUID.int

128 ビット整数での UUID。

UUID.urn

RFC 4122 で規定される URN での UUID。

UUID.variant

UUID の内部レイアウトを決定する UUID の variant。これは定数 *RESERVED_NCS*, *RFC_4122*, *RESERVED_MICROSOFT*, *RESERVED_FUTURE* のいずれかです。

UUID.version

UUID の version 番号（1 から 5、variant が *RFC_4122* である場合だけ意味があります）。

UUID.is_safe

SafeUUID の列挙で、UUID は並列処理に対して安全なプラットフォームで生成されたかを示す。

バージョン 3.7 で追加.

The *uuid* モジュールには以下の関数があります:

uuid.getnode()

48 ビットの正の整数としてハードウェアアドレスを取得します。最初にこれを起動すると、別個のプログラムが立ち上がって非常に遅くなることがあります。もしハードウェアを取得する試みが全て失敗すると、ランダムな 48 ビットを、**RFC 4122** で推奨されているように、マルチキャストビット (最初のオクテットの最下位ビット) を 1 に設定して使います。” ハードウェアアドレス” とはネットワークインターフェースの MAC アドレスを指します。複数のネットワークインターフェースを持つマシンの場合、全域管理された MAC アドレス (最初のオクテットの下位より 2 番目のビットが **設定されていない** MAC アドレス) が、他の個別管理アドレスよりも優先的に使用されます。この優先順序は保証されません。

バージョン 3.7 で変更: 全域管理された MAC アドレスは、グローバルに固有であると保証されるため、固有である保証のない個別管理アドレスよりも好ましいです。

uuid.uuid1(*node=None*, *clock_seq=None*)

UUID をホスト ID、シーケンス番号、現在時刻から生成します。*node* が与えられなければ、*getnode()*

がハードウェアアドレス取得のために使われます。*clock_seq* が与えられると、これはシーケンス番号として使われます; さもなくば 14 ビットのランダムなシーケンス番号が選ばれます。

uuid.**uuid3**(*namespace*, *name*)
UUID を名前空間識別子 (UUID) と名前 (文字列) の MD5 ハッシュから生成します。

uuid.**uuid4**()
ランダムな UUID を生成します。

uuid.**uuid5**(*namespace*, *name*)
名前空間識別子 (これは UUID です) と名前 (文字列です) の SHA-1 ハッシュから UUID を生成します。

uuid モジュールは *uuid3()* または *uuid5()* で利用するために次の名前空間識別子を定義しています。

uuid.**NAMESPACE_DNS**
この名前空間が指定された場合、*name* 文字列は完全修飾ドメイン名です。

uuid.**NAMESPACE_URL**
この名前空間が指定された場合、*name* 文字列は URL です。

uuid.**NAMESPACE_OID**
この名前空間が指定された場合、*name* 文字列は ISO OID です。

uuid.**NAMESPACE_X500**
この名前空間が指定された場合、*name* 文字列は X.500 DN の DER またはテキスト出力形式です。

The *uuid* モジュールは以下の定数を `variant` 属性が取りうる値として定義しています:

uuid.**RESERVED_NCS**
NCS 互換性のために予約されています。

uuid.**RFC_4122**
RFC 4122 で与えられた UUID レイアウトを指定します。

uuid.**RESERVED_MICROSOFT**
Microsoft の互換性のために予約されています。

uuid.**RESERVED_FUTURE**
将来のために予約されています。

参考:

RFC 4122 - Universally Unique IDentifier (UUID) の URN 名前空間　この仕様は UUID のための Uniform Resource Name 名前空間、UUID の内部フォーマットと UUID の生成方法を定義しています。

21.20.1 使用例

典型的な *uuid* モジュールの利用方法を示します:

```
>>> import uuid

>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')

>>> # make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
>>> x.bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')
```

21.21 socketserver --- ネットワークサーバのフレームワーク

ソースコード: Lib/socketserver.py

---

socketserver モジュールはネットワークサーバを実装するタスクを単純化します。

基本的な具象サーバクラスが 4 つあります:

class socketserver.**TCPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

This uses the Internet TCP protocol, which provides for continuous streams of data between the client and server. If *bind_and_activate* is true, the constructor automatically attempts to invoke

server_bind() and *server_activate()*. The other parameters are passed to the *BaseServer* base class.

class socketserver.**UDPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

This uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The parameters are the same as for *TCPServer*.

class socketserver.**UnixStreamServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

class socketserver.**UnixDatagramServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

These more infrequently used classes are similar to the TCP and UDP classes, but use Unix domain sockets; they're not available on non-Unix platforms. The parameters are the same as for *TCPServer*.

これらの 4 つのクラスは要求を **同期的に** *(synchronously)* 処理します; 各要求は次の要求を開始する前に完結していなければなりません。同期的な処理は、サーバで大量の計算を必要とする、あるいはクライアントが処理するには時間がかかりすぎるような大量のデータを返す、といった理由によってリクエストに長い時間がかかる状況には向いていません。こうした状況の解決方法は別のプロセスを生成するか、個々の要求を扱うスレッドを生成することです; *ForkingMixIn* および *ThreadingMixIn* 配合クラス (mix-in classes) を使えば、非同期的な動作をサポートできます。

サーバの作成にはいくつかのステップが必要です。最初に、*BaseRequestHandler* クラスをサブクラス化して要求処理クラス (request hander class) を生成し、その *handle()* メソッドをオーバーライドしなければなりません; このメソッドは入力される要求を処理します。次に、サーバクラスのうち一つを、サーバのアドレスと要求処理クラスを渡してインスタンス化しなければなりません。サーバを `with` 文中で使うことが推奨されます。そして、*handle_request()* または *serve_forever()* メソッドを呼び出して、一つのまたは多数の要求を処理します。最後に、(`with` 文を使っていなかったなら) *server_close()* を呼び出してソケットを閉じます。

ThreadingMixIn から継承してスレッドを利用した接続を行う場合、突発的な通信切断時の処理を明示的に指定する必要があります。*ThreadingMixIn* クラスには *daemon_threads* 属性があり、サーバがスレッドの終了を待ち合わせるかどうかを指定する事ができます。スレッドが独自の処理を行う場合は、このフラグを明示的に指定します。デフォルトは *False* で、Python は *ThreadingMixIn* クラスが起動した全てのスレッドが終了するまで実行し続けます。

サーバクラス群は使用するネットワークプロトコルに関わらず、同じ外部メソッドおよび属性を持ちます。

21.21.1 サーバ生成に関するノート

継承図にある五つのクラスのうち四つは四種類の同期サーバを表わしています:

```
+------------+
| BaseServer |
+------------+
      |
      v
```

(次のページに続く)

(前のページからの続き)

```
+-----------+        +------------------+
| TCPServer |------->| UnixStreamServer |
+-----------+        +------------------+
      |
      v
+-----------+        +--------------------+
| UDPServer |------->| UnixDatagramServer |
+-----------+        +--------------------+
```

UnixDatagramServer は *UDPServer* から派生していて、*UnixStreamServer* からではないことに注意してください --- IP と Unix ストリームサーバの唯一の違いはアドレスファミリーでそれは両方の Unix サーバクラスで単純に繰り返されています。

class socketserver.**ForkingMixIn**

class socketserver.**ThreadingMixIn**

それぞれのタイプのサーバのフォークしたりスレッド実行したりするバージョンはそれらのミクシン (mix-in) クラスを使って作ることができます。たとえば、*ThreadingUDPServer* は以下のようにして作られます:

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

ミクシンクラスは *UDPServer* で定義されるメソッドをオーバライドするために、最初に来ます。様々な属性を設定することで元になるサーバ機構の振る舞いを変えられます。

ForkingMixIn and the Forking classes mentioned below are only available on POSIX platforms that support *fork()*.

`socketserver.ForkingMixIn.server_close()` waits until all child processes complete, except if `socketserver.ForkingMixIn.block_on_close` attribute is false.

`socketserver.ThreadingMixIn.server_close()` waits until all non-daemon threads complete, except if `socketserver.ThreadingMixIn.block_on_close` attribute is false. Use daemonic threads by setting `ThreadingMixIn.daemon_threads` to `True` to not wait until threads complete.

バージョン 3.7 で変更: `socketserver.ForkingMixIn.server_close()` and `socketserver.ThreadingMixIn.server_close()` now waits until all child processes and non-daemonic threads complete. Add a new `socketserver.ForkingMixIn.block_on_close` class attribute to opt-in for the pre-3.7 behaviour.

class socketserver.**ForkingTCPServer**

class socketserver.**ForkingUDPServer**

class socketserver.**ThreadingTCPServer**

class socketserver.**ThreadingUDPServer**

These classes are pre-defined using the mix-in classes.

サービスの実装には、*BaseRequestHandler* からクラスを派生させてその *handle()* メソッドを再定義しなければなりません。このようにすれば、サーバクラスと要求処理クラスを結合して様々なバージョンのサービ

スを実行することができます。要求処理クラスはデータグラムサービスかストリームサービスかで異なることでしょう。この違いは処理サブクラス *StreamRequestHandler* または *DatagramRequestHandler* を使うという形で隠蔽できます。

もちろん、まだ頭を使わなければなりません! たとえば、サービスがリクエストによっては書き換えられるようなメモリ上の状態を使うならば、フォークするサーバを使うのは馬鹿げています。というのも子プロセスでの書き換えは親プロセスで保存されている初期状態にも親プロセスから分配される各子プロセスの状態にも届かないからです。この場合、スレッド実行するサーバを使うことはできますが、共有データの一貫性を保つためにロックを使わなければならなくなるでしょう。

一方、全てのデータが外部に (たとえばファイルシステムに) 保存される HTTP サーバを作っているのだとすると、同期クラスではどうしても一つの要求が処理されている間サービスが「耳の聞こえない」状態を呈することになります --- この状態はもしクライアントが要求した全てのデータをゆっくり受け取るととても長い時間続きかねません。こういう場合にはサーバをスレッド実行したりフォークすることが適切です。

ある場合には、要求の一部を同期的に処理する一方で、要求データに依って子プロセスをフォークして処理を終了させる、といった方法も適当かもしれません。こうした処理方法は同期サーバを使って要求処理クラスの *handle()* メソッドの中で自分でフォークするようにして実装することができます。

スレッドも *fork()* もサポートされない環境で (もしくはサービスにとってそれらがあまりに高価についたり不適切な場合に) 多数の同時要求を捌くもう一つのアプローチは、部分的に処理し終えた要求のテーブルを自分で管理し、次にどの要求に対処するか (または新しく入ってきた要求を扱うかどうか) を決めるのに *selectors* を使う方法です。これは (もしスレッドやサブプロセスが使えなければ) 特にストリームサービスに対して重要で、そのようなサービスでは各クライアントが潜在的に長く接続し続けます。この問題を管理する別の方法について、*asyncore* モジュールを参照してください。

21.21.2 Server オブジェクト

class socketserver.**BaseServer**(*server_address*, *RequestHandlerClass*)

This is the superclass of all Server objects in the module. It defines the interface, given below, but does not implement most of the methods, which is done in subclasses. The two parameters are stored in the respective *server_address* and *RequestHandlerClass* attributes.

fileno()

サーバが要求待ちを行っているソケットのファイル記述子を整数で返します。この関数は一般的に、同じプロセス中の複数のサーバを監視できるようにするために、*selectors* に渡されます。

handle_request()

単一の要求を処理します。この関数は以下のメソッド: *get_request()* 、*verify_request()* 、および *process_request()* を順番に呼び出します。ハンドラ中でユーザによって提供された *handle()* が例外を送出した場合、サーバの *handle_error()* メソッドが呼び出されます。*timeout* 秒以内にリクエストが来なかった場合、*handle_timeout()* が呼ばれて、*handle_request()* が終了します。

serve_forever(*poll_interval=0.5*)

Handle requests until an explicit *shutdown()* request. Poll for shutdown every *poll_interval*

seconds. Ignores the *timeout* attribute. It also calls *service_actions()*, which may be used by a subclass or mixin to provide actions specific to a given service. For example, the *ForkingMixIn* class uses *service_actions()* to clean up zombie child processes.

バージョン 3.3 で変更: Added `service_actions` call to the `serve_forever` method.

service_actions()

This is called in the *serve_forever()* loop. This method can be overridden by subclasses or mixin classes to perform actions specific to a given service, such as cleanup actions.

バージョン 3.3 で追加.

shutdown()

Tell the *serve_forever()* loop to stop and wait until it does. *shutdown()* must be called while *serve_forever()* is running in a different thread otherwise it will deadlock.

server_close()

サーバをクリーンアップします。上書き出来ます。

address_family

サーバのソケットが属しているプロトコルファミリです。一般的な値は *socket.AF_INET* および *socket.AF_UNIX* です。

RequestHandlerClass

ユーザが提供する要求処理クラスです; 要求ごとにこのクラスのインスタンスが生成されます。

server_address

サーバが要求待ちを行うアドレスです。アドレスの形式はプロトコルファミリによって異なります。詳細は *socket* モジュールを参照してください。インターネットプロトコルでは、この値は例えば `('127.0.0.1', 80)` のようにアドレスを与える文字列と整数のポート番号を含むタプルです。

socket

サーバが入力の要求待ちを行うためのソケットオブジェクトです。

サーバクラスは以下のクラス変数をサポートします:

allow_reuse_address

サーバがアドレスの再使用を許すかどうかを示す値です。この値は標準で *False* で、サブクラスで再使用ポリシを変更するために設定することができます。

request_queue_size

要求待ち行列 (queue) のサイズです。単一の要求を処理するのに長時間かかる場合には、サーバが処理中に届いた要求は最大 *request_queue_size* 個まで待ち行列に置かれます。待ち行列が一杯になると、それ以降のクライアントからの要求は ” 接続拒否 (Connection denied)” エラーになります。標準の値は通常 5 ですが、この値はサブクラスで上書きすることができます。

socket_type

サーバが使うソケットの型です; 一般的な 2 つの値は、*socket.SOCK_STREAM* と *socket.SOCK_DGRAM* です。

timeout
タイムアウト時間 (秒)、もしくは、タイムアウトを望まない場合に *None* 。*handle_request()* がこの時間内にリクエストを受信しない場合、*handle_timeout()* メソッドが呼ばれます。

TCPServer のような基底クラスのサブクラスで上書きできるサーバメソッドは多数あります; これらのメソッドはサーバオブジェクトの外部のユーザにとっては役にたたないものです。

finish_request(*request*, *client_address*)
RequestHandlerClass をインスタンス化し、*handle()* メソッドを呼び出して、実際に要求を処理します。

get_request()
ソケットから要求を受理して、クライアントとの通信に使われる **新しい** ソケットオブジェクト、およびクライアントのアドレスからなる、2 要素のタプルを返します。

handle_error(*request*, *client_address*)
この関数は *RequestHandlerClass* インスタンスの *handle()* メソッドが例外を送出した際に呼び出されます。標準の動作では標準エラー出力へトレースバックを出力し、後続する要求を継続して処理します。

バージョン 3.6 で変更: Now only called for exceptions derived from the *Exception* class.

handle_timeout()
この関数は *timeout* 属性が *None* 以外に設定されて、リクエストがないままタイムアウト秒数が過ぎたときに呼ばれます。fork 型サーバーでのデフォルトの動作は、終了した子プロセスの情報を集めるようになっています。スレッド型サーバーではこのメソッドは何もしません。

process_request(*request*, *client_address*)
finish_request() を呼び出して、*RequestHandlerClass* のインスタンスを生成します。必要なら、この関数から新たなプロセスかスレッドを生成して要求を処理することができます; その処理は *ForkingMixIn* または *ThreadingMixIn* クラスが行います。

server_activate()
Called by the server's constructor to activate the server. The default behavior for a TCP server just invokes *listen()* on the server's socket. May be overridden.

server_bind()
サーバのコンストラクタによって呼び出され、適切なアドレスにソケットをバインドします。このメソッドは上書きできます。

verify_request(*request*, *client_address*)
ブール値を返さなければなりません; 値が *True* の場合には要求が処理され、*False* の場合には要求は拒否されます。サーバへのアクセス制御を実装するためにこの関数を上書きすることができます。標準の実装では常に *True* を返します。

バージョン 3.6 で変更: *context manager* プロトコルのサポートが追加されました。コンテキストマネージャを終了することは、*server_close()* を呼ぶことと同一です。

21.21.3 Request Handler Objects

class socketserver.BaseRequestHandler

This is the superclass of all request handler objects. It defines the interface, given below. A concrete request handler subclass must define a new *handle()* method, and can override any of the other methods. A new instance of the subclass is created for each request.

setup()

handle() メソッドより前に呼び出され、何らかの必要な初期化処理を行います。標準の実装では何も行いません。

handle()

この関数では、クライアントからの要求を実現するために必要な全ての作業を行わなければなりません。デフォルト実装では何もしません。この作業の上で、いくつかのインスタンス属性を利用することができます; クライアントからの要求は `self.request` です; クライアントのアドレスは `self.client_address` です; そしてサーバごとの情報にアクセスする場合には、サーバインスタンスを `self.server` で取得できます。

The type of `self.request` is different for datagram or stream services. For stream services, `self.request` is a socket object; for datagram services, `self.request` is a pair of string and socket.

finish()

handle() メソッドより後に呼び出され、何らかの必要なクリーンアップ処理を行います。標準の実装では何も行いません。*setup()* メソッドが例外を送出した場合、このメソッドは呼び出されません。

class socketserver.StreamRequestHandler

class socketserver.DatagramRequestHandler

These *BaseRequestHandler* subclasses override the *setup()* and *finish()* methods, and provide `self.rfile` and `self.wfile` attributes. The `self.rfile` and `self.wfile` attributes can be read or written, respectively, to get the request data or return data to the client.

The `rfile` attributes of both classes support the *io.BufferedIOBase* readable interface, and `DatagramRequestHandler.wfile` supports the *io.BufferedIOBase* writable interface.

バージョン 3.6 で変更: `StreamRequestHandler.wfile` also supports the *io.BufferedIOBase* writable interface.

21.21.4 使用例

socketserver.TCPServer の例

サーバサイドの例です:

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    The request handler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        # Activate the server; this will keep running until you
        # interrupt the program with Ctrl-C
        server.serve_forever()
```

別の、ストリーム (標準のファイル型のインタフェースを利用して通信をシンプルにしたファイルライクオブジェクト) を使うリクエストハンドラクラスの例です:

```
class MyTCPHandler(socketserver.StreamRequestHandler):

    def handle(self):
        # self.rfile is a file-like object created by the handler;
        # we can now use e.g. readline() instead of raw recv() calls
        self.data = self.rfile.readline().strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # Likewise, self.wfile is a file-like object used to write back
        # to the client
        self.wfile.write(self.data.upper())
```

先ほどとの違いは、readline() の呼び出しが、改行を受け取るまで recv() を複数回呼び出すことです。1 回の recv() の呼び出しは、クライアントから 1 回の sendall() 呼び出しで送信された分しか受け取りません。

クライアントサイドの例:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data + "\n", "utf-8"))

    # Receive data from the server and shut down
    received = str(sock.recv(1024), "utf-8")

print("Sent:     {}".format(data))
print("Received: {}".format(received))
```

この例の出力は次のようになります:

サーバー:

```
$ python TCPServer.py
127.0.0.1 wrote:
b'hello world with TCP'
127.0.0.1 wrote:
b'python is nice'
```

クライアント:

```
$ python TCPClient.py hello world with TCP
Sent:     hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent:     python is nice
Received: PYTHON IS NICE
```

socketserver.UDPServer の例

サーバサイドの例です:

```
import socketserver

class MyUDPHandler(socketserver.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and since
    there is no connection the client address must be given explicitly
    when sending data back via sendto().
```

(次のページに続く)

(前のページからの続き)

```
    """

    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print("{} wrote:".format(self.client_address[0]))
        print(data)
        socket.sendto(data.upper(), self.client_address)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.UDPServer((HOST, PORT), MyUDPHandler) as server:
        server.serve_forever()
```

クライアントサイドの例:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no connections.
# Instead, data is directly sent to the recipient via sendto().
sock.sendto(bytes(data + "\n", "utf-8"), (HOST, PORT))
received = str(sock.recv(1024), "utf-8")

print("Sent:     {}".format(data))
print("Received: {}".format(received))
```

この例の出力は、TCP サーバーの例と全く同じようになります。

非同期処理の Mix-in

複数の接続を非同期に処理するハンドラを作るには、*ThreadingMixIn* か *ForkingMixIn* クラスを利用します。

ThreadingMixIn クラスの利用例:

```
import socket
import threading
import socketserver

class ThreadedTCPRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024), 'ascii')
```

(次のページに続く)

(前のページからの続き)

```
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name, data), 'ascii')
        self.request.sendall(response)

class ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
    pass

def client(ip, port, message):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((ip, port))
        sock.sendall(bytes(message, 'ascii'))
        response = str(sock.recv(1024), 'ascii')
        print("Received: {}".format(response))

if __name__ == "__main__":
    # Port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    with server:
        ip, port = server.server_address

        # Start a thread with the server -- that thread will then start one
        # more thread for each request
        server_thread = threading.Thread(target=server.serve_forever)
        # Exit the server thread when the main thread terminates
        server_thread.daemon = True
        server_thread.start()
        print("Server loop running in thread:", server_thread.name)

        client(ip, port, "Hello World 1")
        client(ip, port, "Hello World 2")
        client(ip, port, "Hello World 3")

        server.shutdown()
```

この例の出力は次のようになります:

```
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

ForkingMixIn クラスは同じように利用することができます。この場合、サーバーはリクエスト毎に新しいプロセスを作成します。*fork()* をサポートする POSIX プラットフォームでのみ利用可能です。

21.22 http.server --- HTTP サーバ

ソースコード: Lib/http/server.py

このモジュールは HTTP (web) サーバを実装するためのクラスを提供しています。

> **警告:** *http.server* is not recommended for production. It only implements *basic security checks*.

クラス *HTTPServer* は *socketserver.TCPServer* のサブクラスです。*HTTPServer* は HTTP ソケットを生成してリクエスト待ち (listen) を行い、リクエストをハンドラに渡します。サーバを作成して動作させるためのコードは以下のようになります:

```
def run(server_class=HTTPServer, handler_class=BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

class http.server.**HTTPServer**(*server_address*, *RequestHandlerClass*)

このクラスは *TCPServer* クラスの上に構築されており、サーバのアドレスをインスタンス変数 server_name および server_port に記憶します。サーバはハンドラからアクセス可能で、通常ハンドラの server インスタンス変数からアクセスします。

class http.server.**ThreadingHTTPServer**(*server_address*, *RequestHandlerClass*)

This class is identical to HTTPServer but uses threads to handle requests by using the *ThreadingMixIn*. This is useful to handle web browsers pre-opening sockets, on which *HTTPServer* would wait indefinitely.

バージョン 3.7 で追加.

HTTPServer と *ThreadingHTTPServer* は *RequestHandlerClass* の初期化のときに与えられなければならず、このモジュールはこのクラスの 3 つの変種を提供しています:

class http.server.**BaseHTTPRequestHandler**(*request*, *client_address*, *server*)

このクラスはサーバに到着したリクエストを処理します。このメソッド自体では、実際のリクエストに応答することはできません; (GET や POST のような) 各リクエストメソッドを処理するためにはサブクラス化しなければなりません。*BaseHTTPRequestHandler* では、サブクラスで使うためのクラスやインスタンス変数、メソッド群を数多く提供しています。

このハンドラはリクエストを解釈し、次いでリクエスト形式ごとに固有のメソッドを呼び出します。メソッド名はリクエストの名称から構成されます。例えば、リクエストメソッド SPAM に対しては、do_SPAM() メソッドが引数なしで呼び出されます。リクエストに関連する情報は全て、ハンドラのインスタンス変数に記憶されています。サブクラスでは __init__() メソッドを上書きしたり拡張したりする必要はありません。

BaseHTTPRequestHandler は以下のインスタンス変数を持っています:

client_address
HTTP クライアントのアドレスを参照している、(host, port) の形式をとるタプルが入っています。

server
server インスタンスが入っています。

close_connection
handle_one_request() が返る前に設定すべき真偽値で、別のリクエストが期待されているかどうか、もしくはコネクションを切断すべきかどうかを指し示しています。

requestline
HTTP リクエスト行の文字列表現を保持しています。末尾の CRLF は除去されています。この属性は *handle_one_request()* によって設定されるべきです。妥当なリクエスト行が 1 行も処理されなかった場合は、空文字列に設定されるべきです。

command
HTTP 命令 (リクエスト形式) が入っています。例えば 'GET' です。

path
Contains the request path. If query component of the URL is present, then path includes the query. Using the terminology of **RFC 3986**, path here includes hier-part and the query.

request_version
リクエストのバージョン文字列が入っています。例えば 'HTTP/1.0' です。

headers
MessageClass クラス変数で指定されたクラスのインスタンスを保持しています。このインスタンスは HTTP リクエストのヘッダを解釈し、管理しています。*http.client* の *parse_headers()* 関数がヘッダを解釈するために使われ、HTTP リクエストが妥当な **RFC 2822** スタイルのヘッダを提供することを要求します。

rfile
An *io.BufferedIOBase* input stream, ready to read from the start of the optional input data.

wfile
Contains the output stream for writing a response back to the client. Proper adherence to the HTTP protocol must be used when writing to this stream in order to achieve successful interoperation with HTTP clients.

バージョン 3.6 で変更: This is an *io.BufferedIOBase* stream.

BaseHTTPRequestHandler は以下の属性を持っています:

server_version
サーバのソフトウェアバージョンを指定します。この値は上書きする必要が生じるかもしれません。書式は複数の文字列を空白で分割したもので、各文字列はソフトウェア名 [/バージョン] の形式をとります。例えば、'BaseHTTP/0.2' です。

sys_version

Python 処理系のバージョンが, *version_string* メソッドや *server_version* クラス変数で利用可能な形式で入っています。例えば 'Python/1.4' です。

error_message_format

Specifies a format string that should be used by *send_error()* method for building an error response to the client. The string is filled by default with variables from *responses* based on the status code that passed to *send_error()*.

error_content_type

エラーレスポンスをクライアントに送信する時に使う Content-Type HTTP ヘッダを指定します。デフォルトでは 'text/html' です。

protocol_version

この値には応答に使われる HTTP プロトコルのバージョンを指定します。'HTTP/1.1' に設定されると、サーバは持続的 HTTP 接続を許可します; しかしその場合、サーバは全てのクライアントに対する応答に、正確な値を持つ Content-Length ヘッダを (*send_header()* を使って) 含め **なければなりません** 。以前のバージョンとの互換性を保つため、標準の設定値は 'HTTP/1.0' です。

MessageClass

HTTP ヘッダを解釈するための *email.message.Message* 類似のクラスを指定します。通常この値が上書きされることはなく、デフォルトの http.client.HTTPMessage になっています。

responses

This attribute contains a mapping of error code integers to two-element tuples containing a short and long message. For example, {code: (shortmessage, longmessage)}. The *shortmessage* is usually used as the *message* key in an error response, and *longmessage* as the *explain* key. It is used by *send_response_only()* and *send_error()* methods.

BaseHTTPRequestHandler インスタンスは以下のメソッドを持っています:

handle()

handle_one_request() を一度だけ (持続的接続が有効になっている場合には複数回) 呼び出して、HTTP リクエストを処理します。このメソッドを上書きする必要はまったくありません; そうする代わりに適切な do_*() を実装してください。

handle_one_request()

このメソッドはリクエストを解釈し、適切な do_*() メソッドに転送します。このメソッドを上書きする必要はまったくありません。

handle_expect_100()

HTTP/1.1 準拠のサーバが Expect: 100-continue リクエストヘッダを受け取ったとき、100 Continue を返し、その後に 200 OK がきます。このメソッドは、サーバがクライアントに継続を要求しない場合、エラーを送出するようにオーバーライドできます。例えば、サーバはレスポンスヘッダとして 417 Expectation Failed を送る選択をし、return False とできます。

バージョン 3.2 で追加.

send_error(*code*, *message=None*, *explain=None*)

Sends and logs a complete error reply to the client. The numeric *code* specifies the HTTP

error code, with *message* as an optional, short, human readable description of the error. The *explain* argument can be used to provide more detailed information about the error; it will be formatted using the *error_message_format* attribute and emitted, after a complete set of headers, as the response body. The *responses* attribute holds the default values for *message* and *explain* that will be used if no value is provided; for unknown codes the default value for both is the string `???`. The body will be empty if the method is HEAD or the response code is one of the following: `1xx`, `204 No Content`, `205 Reset Content`, `304 Not Modified`.

バージョン 3.4 で変更: The error response includes a Content-Length header. Added the *explain* argument.

send_response(*code*, *message=None*)

Adds a response header to the headers buffer and logs the accepted request. The HTTP response line is written to the internal buffer, followed by *Server* and *Date* headers. The values for these two headers are picked up from the *version_string()* and *date_time_string()* methods, respectively. If the server does not intend to send any other headers using the *send_header()* method, then *send_response()* should be followed by an *end_headers()* call.

バージョン 3.3 で変更: Headers are stored to an internal buffer and *end_headers()* needs to be called explicitly.

send_header(*keyword*, *value*)

Adds the HTTP header to an internal buffer which will be written to the output stream when either *end_headers()* or *flush_headers()* is invoked. *keyword* should specify the header keyword, with *value* specifying its value. Note that, after the send_header calls are done, *end_headers()* MUST BE called in order to complete the operation.

バージョン 3.2 で変更: ヘッダは内部バッファに保存されます。

send_response_only(*code*, *message=None*)

Sends the response header only, used for the purposes when `100 Continue` response is sent by the server to the client. The headers not buffered and sent directly the output stream.If the *message* is not specified, the HTTP message corresponding the response *code* is sent.

バージョン 3.2 で追加.

end_headers()

Adds a blank line (indicating the end of the HTTP headers in the response) to the headers buffer and calls *flush_headers()*.

バージョン 3.2 で変更: バッファされたヘッダは出力ストリームに書き出されます。

flush_headers()

最終的にヘッダを出力ストリームに送り、内部ヘッダバッファをフラッシュします。

バージョン 3.3 で追加.

log_request(*code='-'*, *size='-'*)

受理された (成功した) リクエストをログに記録します。*code* にはこの応答に関連付けられた HTTP コード番号を指定します。応答メッセージの大きさを知ることができる場合、*size* パラメタに渡すとよいでしょう。

log_error(*...*)
リクエストを遂行できなかった際に、エラーをログに記録します。標準では、メッセージを *log_message()* に渡します。従って同じ引数 (*format* と追加の値) を取ります。

log_message(*format*, *...*)
任意のメッセージを sys.stderr にログ記録します。このメソッドは通常、カスタムのエラーログ記録機構を作成するために上書きされます。*format* 引数は標準の printf 形式の書式文字列で、*log_message()* に渡された追加の引数は書式化の入力として適用されます。ログ記録される全てのメッセージには、クライアントの IP アドレスおよび現在の日付、時刻が先頭に付けられます。

version_string()
サーバーのソフトウェアのバージョンを示す文字列を返します。その文字列は、*server_version* と *sys_version* の属性が含まれます。

date_time_string(*timestamp=None*)
メッセージヘッダ向けに書式化された、*timestamp* (None または *time.time()* のフォーマットである必要があります) で与えられた日時を返します。もし *timestamp* が省略された場合には、現在の日時が使われます。

出力は 'Sun, 06 Nov 1994 08:49:37 GMT' のようになります。

log_date_time_string()
ログ記録向けに書式化された、現在の日付および時刻を返します。

address_string()
クライアントのアドレスを返します。

バージョン 3.3 で変更: Previously, a name lookup was performed. To avoid name resolution delays, it now always returns the IP address.

class http.server.**SimpleHTTPRequestHandler**(*request*, *client_address*, *server*, *directory=None*)
このクラスは、現在のディレクトリ以下にあるファイルを、HTTP リクエストにおけるディレクトリ構造に直接対応付けて提供します。

リクエストの解釈のような、多くの作業は基底クラス *BaseHTTPRequestHandler* で行われます。このクラスは関数 *do_GET()* および *do_HEAD()* を実装しています。

SimpleHTTPRequestHandler では以下のメンバ変数を定義しています:

server_version
この値は "SimpleHTTP/" + __version__ になります。__version__ はこのモジュールで定義されている値です。

extensions_map
拡張子を MIME 型指定子に対応付ける辞書です。標準の型指定は空文字列で表され、この値は

application/octet-stream と見なされます。対応付けは大小文字の区別をするので、小文字のキーのみを入れるべきです。

directory
If not specified, the directory to serve is the current working directory.

SimpleHTTPRequestHandler では以下のメソッドを定義しています:

do_HEAD()
このメソッドは 'HEAD' 型のリクエスト処理を実行します: すなわち、GET リクエストの時に送信されるものと同じヘッダを送信します。送信される可能性のあるヘッダについての完全な説明は *do_GET()* メソッドを参照してください。

do_GET()
リクエストを現在の作業ディレクトリからの相対的なパスとして解釈することで、リクエストをローカルシステム上のファイルと対応付けます。

リクエストがディレクトリに対応付けられた場合、index.html または index.htm を (この順序で) チェックします。もしファイルを発見できればその内容を、そうでなければディレクトリ一覧を list_directory() メソッドで生成して、返します。このメソッドは *os.listdir()* をディレクトリのスキャンに用いており、*listdir()* が失敗した場合には 404 応答が返されます。

If the request was mapped to a file, it is opened. Any *OSError* exception in opening the requested file is mapped to a 404, 'File not found' error. If there was a 'If-Modified-Since' header in the request, and the file was not modified after this time, a 304, 'Not Modified' response is sent. Otherwise, the content type is guessed by calling the guess_type() method, which in turn uses the *extensions_map* variable, and the file contents are returned.

出力は 'Content-type:' と推測されたコンテントタイプで、その後にファイルサイズを示す 'Content-Length;' ヘッダと、ファイルの更新日時を示す 'Last-Modified:' ヘッダが続きます。

そしてヘッダの終了を示す空白行が続き、さらにその後にファイルの内容が続きます。このファイルはコンテントタイプが text/ で始まっている場合はテキストモードで、そうでなければバイナリモードで開かれます。

For example usage, see the implementation of the *test()* function invocation in the *http.server* module.

バージョン 3.7 で変更: Support of the 'If-Modified-Since' header.

The *SimpleHTTPRequestHandler* class can be used in the following manner in order to create a very basic webserver serving files relative to the current directory:

```
import http.server
import socketserver

PORT = 8000
```

(次のページに続く)

(前のページからの続き)

```
Handler = http.server.SimpleHTTPRequestHandler

with socketserver.TCPServer(("", PORT), Handler) as httpd:
    print("serving at port", PORT)
    httpd.serve_forever()
```

インタプリタの -m スイッチで *http.server* モジュールと **ポート番号** を指定して直接実行することもできます。上の例と同じように、ここで立ち上がったサーバは現在のディレクトリ以下のファイルへのアクセスを提供します。

```
python -m http.server 8000
```

By default, server binds itself to all interfaces. The option `-b/--bind` specifies a specific address to which it should bind. Both IPv4 and IPv6 addresses are supported. For example, the following command causes the server to bind to localhost only:

```
python -m http.server 8000 --bind 127.0.0.1
```

バージョン 3.4 で追加: `--bind` 引数が導入されました。

バージョン 3.8 で追加: `--bind` argument enhanced to support IPv6

By default, server uses the current directory. The option `-d/--directory` specifies a directory to which it should serve the files. For example, the following command uses a specific directory:

```
python -m http.server --directory /tmp/
```

バージョン 3.7 で追加: `--directory` specify alternate directory

class http.server.**CGIHTTPRequestHandler**(*request*, *client_address*, *server*)

このクラスは、現在のディレクトリかその下のディレクトリにおいて、ファイルか CGI スクリプト出力を提供するために使われます。HTTP 階層構造からローカルなディレクトリ構造への対応付けは *SimpleHTTPRequestHandler* と全く同じなので注意してください。

注釈: *CGIHTTPRequestHandler* クラスで実行される CGI スクリプトは HTTP コード 200 (スクリプトの出力が後に続く) を実行に先立って出力される (これがステータスコードになります) ため、リダイレクト (コード 302) を行なうことができません。

このクラスでは、ファイルが CGI スクリプトであると推測された場合、これをファイルとして提供する代わりにスクリプトを実行します。--- 他の一般的なサーバ設定は特殊な拡張子を使って CGI スクリプトであることを示すのに対し、ディレクトリベースの CGI だけが使われます。

`do_GET()` および `do_HEAD()` 関数は、HTTP 要求が `cgi_directories` パス以下のどこかを指している場合、ファイルを提供するのではなく、CGI スクリプトを実行してその出力を提供するように変更されています。

CGIHTTPRequestHandler では以下のデータメンバを定義しています:

cgi_directories
この値は標準で `['/cgi-bin', '/htbin']` であり、CGI スクリプトを含んでいることを示すディレクトリを記述します。

CGIHTTPRequestHandler は以下のメソッドを定義しています:

do_POST()
このメソッドは、CGI スクリプトでのみ許されている `'POST'` 型の HTTP 要求に対するサービスを行います。CGI でない url に対して POST を試みた場合、出力は Error 501, "Can only POST to CGI scripts" になります。

セキュリティ上の理由から、CGI スクリプトはユーザ nobody の UID で動作するので注意してください。CGI スクリプトが原因で発生した問題は、Error 403 に変換されます。

CGIHTTPRequestHandler can be enabled in the command line by passing the `--cgi` option:

```
python -m http.server --cgi 8000
```

21.22.1 Security Considerations

SimpleHTTPRequestHandler will follow symbolic links when handling requests, this makes it possible for files outside of the specified directory to be served.

Earlier versions of Python did not scrub control characters from the log messages emitted to stderr from `python -m http.server` or the default *BaseHTTPRequestHandler* `.log_message` implementation. This could allow remote clients connecting to your server to send nefarious control codes to your terminal.

バージョン 3.8.16 で追加: scrubbing control characters from log messages

21.23 `http.cookies` --- HTTP の状態管理

ソースコード: Lib/http/cookies.py

---

http.cookies モジュールは HTTP の状態管理機能である cookie の概念を抽象化、定義しているクラスです。単純な文字列のみで構成される cookie のほか、シリアル化可能なあらゆるデータ型でクッキーの値を保持するための機能も備えています。

このモジュールは元々 **RFC 2109** と **RFC 2068** に定義されている構文解析の規則を厳密に守っていました。しかし、MSIE 3.0x がこれらの RFC で定義された文字の規則に従っていないことが判明し、また、現代の多くのブラウザとサーバも Cookie の処理に緩い解析をしており、結局、やや厳密さを欠く構文解析規則にせざるを得ませんでした。

文字集合 *string.ascii_letters* 、*string.digits* 、`!#$%&'*+-.^_`|~:` を、このモジュールは Cookie 名 (*key*) として有効と認めています。

バージョン 3.3 で変更: ':' が有効な Cookie 名の文字として認められました。

注釈: 不正な cookie に遭遇した場合、*CookieError* 例外を送出します。そのため、ブラウザから持ってきた cookie データをパースするときには常に不正なデータに備え *CookieError* 例外を捕捉してください。

---

exception http.cookies.**CookieError**

属性や *Set-Cookie* ヘッダが正しくないなど、**RFC 2109** に合致していないときに発生する例外です。

class http.cookies.**BaseCookie**([*input*])

このクラスはキーが文字列、値が *Morsel* インスタンスで構成される辞書風オブジェクトです。値に対するキーを設定するときは、値がキーと値を含む *Morsel* に変換されることに注意してください。

input が与えられたときは、そのまま *load()* メソッドへ渡されます。

class http.cookies.**SimpleCookie**([*input*])

This class derives from *BaseCookie* and overrides `value_decode()` and `value_encode()`. SimpleCookie supports strings as cookie values. When setting the value, SimpleCookie calls the builtin *str()* to convert the value to a string. Values received from HTTP are kept as strings.

参考:

モジュール *http.cookiejar* Web **クライアント** 向けの HTTP クッキー処理です。*http.cookiejar* と *http.cookies* は互いに独立しています。

RFC 2109 - HTTP State Management Mechanism このモジュールが実装している HTTP の状態管理に関する規格です。

21.23.1 Cookie オブジェクト

BaseCookie.**value_decode**(*val*)

Return a tuple `(real_value, coded_value)` from a string representation. `real_value` can be any type. This method does no decoding in *BaseCookie* --- it exists so it can be overridden.

BaseCookie.**value_encode**(*val*)

Return a tuple `(real_value, coded_value)`. *val* can be any type, but `coded_value` will always be converted to a string. This method does no encoding in *BaseCookie* --- it exists so it can be overridden.

通常 *value_encode()* と *value_decode()* はともに *value_decode* の処理内容から逆算した範囲に収まっていなければなりません。

BaseCookie.**output**(*attrs=None*, *header='Set-Cookie:'*, *sep='\r\n'*)

HTTP ヘッダ形式の文字列表現を返します。*attrs* と *header* はそれぞれ *Morsel* の *output()* メソッドに送られます。*sep* はヘッダの連結に用いられる文字で、デフォルトは `'\r\n'` (CRLF) となっています。

BaseCookie.**js_output**(*attrs=None*)

ブラウザが JavaScript をサポートしている場合、HTTP ヘッダを送信した場合と同様に動作する埋め込み可能な JavaScript snippet を返します。

attrs の意味は *output()* と同じです。

BaseCookie.**load**(*rawdata*)

rawdata が文字列であれば、HTTP_COOKIE として処理し、その値を *Morsel* として追加します。辞書の場合は次と同様の処理をおこないます。

```
for k, v in rawdata.items():
    cookie[k] = v
```

21.23.2 Morsel オブジェクト

class http.cookies.**Morsel**

RFC 2109 の属性をキーと値で保持する abstract クラスです。

Morsel は辞書風のオブジェクトで、キーは次のような **RFC 2109** 準拠の定数となっています。

- expires
- path
- comment
- domain
- max-age
- secure
- version
- httponly
- samesite

httponly 属性は、cookie が HTTP リクエストでのみ送信されて、JavaScript からのはアクセスできない事を示します。これはいくつかのクロスサイトスクリプティングの脅威を和らげることを意図しています。

The attribute samesite specifies that the browser is not allowed to send the cookie along with cross-site requests. This helps to mitigate CSRF attacks. Valid values for this attribute are "Strict" and "Lax".

キーの大小文字は区別されません。そのデフォルト値は '' です。

バージョン 3.5 で変更: __eq__() は *key* 及び *value* を考慮するようになりました。

バージョン 3.7 で変更: Attributes *key*, *value* and *coded_value* are read-only. Use *set()* for setting them.

バージョン 3.8 で変更: samesite 属性がサポートされました。

Morsel.value
クッキーの値。

Morsel.coded_value
実際に送信する形式にエンコードされた cookie の値。

Morsel.key
cookie の名前。

Morsel.set(*key*, *value*, *coded_value*)
属性 *key* 、*value* 、*coded_value* に値をセットします。

Morsel.isReservedKey(*K*)
K が *Morsel* のキーであるかどうかを判定します。

Morsel.output(*attrs=None*, *header='Set-Cookie:'*)
Mosel を HTTP ヘッダ形式の文字列表現にして返します。*attrs* を指定しない場合、デフォルトですべての属性を含めます。*attrs* を指定する場合、属性をリストで渡さなければなりません。*header* のデフォルトは "Set-Cookie:" です。

Morsel.js_output(*attrs=None*)
ブラウザが JavaScript をサポートしている場合、HTTP ヘッダを送信した場合と同様に動作する埋め込み可能な JavaScript snippet を返します。

attrs の意味は *output()* と同じです。

Morsel.OutputString(*attrs=None*)
Mosel の文字列表現を HTTP や JavaScript で囲まずに出力します。

attrs の意味は *output()* と同じです。

Morsel.update(*values*)
Morsel 辞書の値を辞書 *values* の値で更新します。*values* 辞書のキーのいずれかが有効な **RFC 2109** 属性でない場合エラーを送出します。

バージョン 3.5 で変更: 不正なキーではエラーが送出されます。

Morsel.copy(*value*)
Morsel オブジェクトの浅いコピーを返します。

バージョン 3.5 で変更: 辞書ではなく Morsel オブジェクトを返します。

Morsel.setdefault(*key*, *value=None*)
キーが有効な **RFC 2109** 属性でない場合エラーを送出します。そうでない場合は *dict.setdefault()* と同じように振る舞います。

21.23.3 使用例

次の例は *http.cookies* の使い方を示したものです。

```
>>> from http import cookies
>>> C = cookies.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print(C) # generate HTTP headers
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print(C.output()) # same thing
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> C = cookies.SimpleCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print(C.output(header="Cookie:"))
Cookie: rocky=road; Path=/cookie
>>> print(C.output(attrs=[], header="Cookie:"))
Cookie: rocky=road
>>> C = cookies.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger") # load from a string (HTTP header)
>>> print(C)
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = cookies.SimpleCookie()
>>> C.load('keebler="E=everybody; L=\\"Loves\\"; fudge=\\012;";')
>>> print(C)
Set-Cookie: keebler="E=everybody; L=\"Loves\"; fudge=\012;"
>>> C = cookies.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print(C)
Set-Cookie: oreo=doublestuff; Path=/
>>> C = cookies.SimpleCookie()
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
>>> C = cookies.SimpleCookie()
>>> C["number"] = 7 # equivalent to C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print(C)
Set-Cookie: number=7
Set-Cookie: string=seven
```

21.24 http.cookiejar --- HTTP クライアント用の Cookie 処理

ソースコード: Lib/http/cookiejar.py

---

http.cookiejar モジュールは HTTP クッキーの自動処理をおこなうクラスを定義します。これは小さなデータの断片 -- **クッキー** -- を要求する web サイトにアクセスする際に有用です。クッキーとは web サーバの HTTP レスポンスによってクライアントのマシンに設定され、のちの HTTP リクエストをおこなうさいにサーバに返されるものです。

標準的な Netscape クッキープロトコルおよび **RFC 2965** で定義されているプロトコルの両方を処理できます。RFC 2965 の処理はデフォルトではオフになっています。**RFC 2109** のクッキーは Netscape クッキーとして解析され、のちに有効な 'ポリシー' に従って Netscape または RFC 2965 クッキーとして処理されます。但し、インターネット上の大多数のクッキーは Netscape クッキーです。*http.cookiejar* はデファクトスタンダードの Netscape クッキープロトコル (これは元々 Netscape が策定した仕様とはかなり異なっています) に従うようになっており、RFC 2109 で導入された `max-age` や `port` などのクッキー属性にも注意を払います。

注釈: *Set-Cookie* や *Set-Cookie2* ヘッダに現れる多種多様なパラメータの名前 (`domain` や `expires` など) は便宜上 **属性** と呼ばれますが、ここでは Python の属性と区別するため、かわりに **クッキー属性** と呼ぶことにします。

このモジュールは以下の例外を定義しています:

exception http.cookiejar.**LoadError**

この例外は *FileCookieJar* インスタンスがファイルからクッキーを読み込むのに失敗した場合に発生します。*LoadError* は *OSError* のサブクラスです。

バージョン 3.3 で変更: LoadError は *IOError* の代わりに *OSError* のサブクラスになりました。

以下のクラスが提供されています:

class http.cookiejar.**CookieJar**(*policy=None*)

policy は *CookiePolicy* インターフェイスを実装するオブジェクトです。

CookieJar クラスには HTTP クッキーを保管します。これは HTTP リクエストに応じてクッキーを取り出し、それを HTTP レスポンスの中で返します。必要に応じて、*CookieJar* インスタンスは保管されているクッキーを自動的に破棄します。このサブクラスは、クッキーをファイルやデータベースに格納したり取り出したりする操作をおこなう役割を負っています。

class http.cookiejar.**FileCookieJar**(*filename*, *delayload=None*, *policy=None*)

policy は *CookiePolicy* インターフェイスを実装するオブジェクトです。これ以外の引数については、該当する属性の説明を参照してください。

FileCookieJar はディスク上のファイルからのクッキーの読み込み、もしくは書き込みをサポートします。実際には、*load()* または *revert()* のどちらかのメソッドが呼ばれるまでクッキーは指定され

たファイルからはロード **されません** 。このクラスのサブクラスは *FileCookieJar* **のサブクラスと** *web* **ブラウザとの連携** 節で説明します。

バージョン 3.8 で変更: *filename* 引数が *path-like object* を受け付けるようになりました。

class http.cookiejar.**CookiePolicy**

このクラスは、あるクッキーをサーバから受け入れるべきか、そしてサーバに返すべきかを決定する役割を負っています。

class http.cookiejar.**DefaultCookiePolicy**(*blocked_domains=None*, *allowed_domains=None*, *netscape=True*, *rfc2965=False*, *rfc2109_as_netscape=None*, *hide_cookie2=False*, *strict_domain=False*, *strict_rfc2965_unverifiable=True*, *strict_ns_unverifiable=False*, *strict_ns_domain=DefaultCookiePolicy.DomainLiberal*, *strict_ns_set_initial_dollar=False*, *strict_ns_set_path=False*, *secure_protocols=("https", "wss")*)

Constructor arguments should be passed as keyword arguments only. *blocked_domains* is a sequence of domain names that we never accept cookies from, nor return cookies to. *allowed_domains* if not *None*, this is a sequence of the only domains for which we accept and return cookies. *secure_protocols* is a sequence of protocols for which secure cookies can be added to. By default *https* and *wss* (secure websocket) are considered secure protocols. For all other arguments, see the documentation for *CookiePolicy* and *DefaultCookiePolicy* objects.

DefaultCookiePolicy は Netscape および **RFC 2965** クッキーの標準的な許可 / 拒絶のルールを実装しています。デフォルトでは、**RFC 2109** のクッキー (*Set-Cookie* の version クッキー属性が 1 で受けとられるもの) は RFC 2965 のルールで扱われます。しかし、RFC 2965 処理が無効に設定されているか *rfc2109_as_netscape* が `True` の場合、RFC 2109 クッキーは *CookieJar* インスタンスによって *Cookie* のインスタンスの `version` 属性を 0 に設定する事で Netscape クッキーに「ダウングレード」されます。また *DefaultCookiePolicy* にはいくつかの細かいポリシー設定をおこなうパラメータが用意されています。

class http.cookiejar.**Cookie**

このクラスは Netscape クッキー、**RFC 2109** のクッキー、および **RFC 2965** のクッキーを表現します。*http.cookiejar* のユーザが自分で *Cookie* インスタンスを作成することは想定されていません。かわりに、必要に応じて *CookieJar* インスタンスの `make_cookies()` を呼ぶことになっています。

参考:

urllib.request **モジュール** クッキーの自動処理をおこない URL を開くモジュールです。

http.cookies **モジュール** HTTP のクッキークラスで、基本的にはサーバサイドのコードで有用です。*http.cookiejar* および *http.cookies* モジュールは互いに依存してはいません。

https://curl.haxx.se/rfc/cookie_spec.html 元祖 Netscape のクッキープロトコルの仕様です。今でもこれが

主流のプロトコルですが、現在のメジャーなブラウザ (と *http.cookiejar*) が実装している「Netscape クッキープロトコル」は cookie_spec.html で述べられているものとおおまかにしか似ていません。

RFC 2109 - HTTP State Management Mechanism **RFC 2965** によって過去の遺物になりました。*Set-Cookie* の version=1 で使います。

RFC 2965 - HTTP State Management Mechanism Netscape プロトコルのバグを修正したものです。*Set-Cookie* のかわりに *Set-Cookie2* を使いますが、普及してはいません。

http://kristol.org/cookie/errata.html **RFC 2965** に対する未完の正誤表です。

RFC 2964 - Use of HTTP State Management

21.24.1 CookieJar および FileCookieJar オブジェクト

CookieJar オブジェクトは保管されている *Cookie* オブジェクトをひとつずつ取り出すための、**イテレータ** プロトコルをサポートしています。

CookieJar は以下のようなメソッドを持っています:

CookieJar.**add_cookie_header**(*request*)

request に正しい *Cookie* ヘッダを追加します。

ポリシーが許すようであれば (*CookieJar* の *CookiePolicy* インスタンスにある属性のうち、rfc2965 および hide_cookie2 がそれぞれ真と偽であるような場合)、必要に応じて *Cookie2* ヘッダも追加されます。

request オブジェクト (通常は *urllib.request.Request* インスタンス) は、*urllib.request* のドキュメントに記されているように、get_full_url(), get_host(), get_type(), unverifiable(), has_header(), get_header(), header_items(), add_unredirected_header() メソッドおよび origin_req_host 属性をサポートしている必要があります。

バージョン 3.3 で変更: *request* object needs origin_req_host attribute. Dependency on a deprecated method get_origin_req_host() has been removed.

CookieJar.**extract_cookies**(*response*, *request*)

HTTP *response* からクッキーを取り出し、ポリシーによって許可されていればこれを *CookieJar* 内に保管します。

CookieJar は *response* 引数の中から許可されている *Set-Cookie* および *Set-Cookie2* ヘッダを探しだし、適切に (*CookiePolicy.set_ok()* メソッドの承認におうじて) クッキーを保管します。

response オブジェクト (通常は *urllib.request.urlopen()* あるいはそれに類似する呼び出しによって得られます) は info() メソッドをサポートしている必要があります。これは *email.message.Message* メソッドのあるオブジェクトを返すものです。

request オブジェクト (通常は *urllib.request.Request* インスタンス) は *urllib.request* のドキュメントに記されているように、get_full_url(), get_host(), unverifiable() メソッドおよび

origin_req_host 属性をサポートしている必要があります。この request はそのクッキーの保存が許可されているかを検査するとともに、クッキー属性のデフォルト値を設定するのに使われます。

バージョン 3.3 で変更: *request* object needs origin_req_host attribute. Dependency on a deprecated method get_origin_req_host() has been removed.

CookieJar.**set_policy**(*policy*)
使用する *CookiePolicy* インスタンスを指定します。

CookieJar.**make_cookies**(*response*, *request*)
response オブジェクトから得られた *Cookie* オブジェクトからなるシーケンスを返します。

response および *request* 引数で要求されるインスタンスについては、*extract_cookies()* の説明を参照してください。

CookieJar.**set_cookie_if_ok**(*cookie*, *request*)
ポリシーが許すのであれば、与えられた *Cookie* を設定します。

CookieJar.**set_cookie**(*cookie*)
与えられた *Cookie* を、それが設定されるべきかどうかのポリシーのチェックを行わずに設定します。

CookieJar.**clear**(*[domain[, path[, name]]]*)
いくつかのクッキーを消去します。

引数なしで呼ばれた場合は、すべてのクッキーを消去します。引数がひとつ与えられた場合、その *domain* に属するクッキーのみを消去します。ふたつの引数が与えられた場合、指定された *domain* と URL *path* に属するクッキーのみを消去します。引数が 3 つ与えられた場合、*domain*, *path* および *name* で指定されるクッキーが消去されます。

与えられた条件に一致するクッキーがない場合は *KeyError* を発生させます。

CookieJar.**clear_session_cookies**()
すべてのセッションクッキーを消去します。

保存されているクッキーのうち、discard 属性が真になっているものすべてを消去します (通常これは max-age または expires のどちらのクッキー属性もないか、あるいは明示的に discard クッキー属性が指定されているものです)。対話的なブラウザの場合、セッションの終了はふつうブラウザのウィンドウを閉じることに相当します。

注意: *ignore_discard* 引数に真を指定しないかぎり、save() メソッドはセッションクッキーは保存しません。

さらに *FileCookieJar* は以下のようなメソッドを実装しています:

FileCookieJar.**save**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
クッキーをファイルに保存します。

この基底クラスは *NotImplementedError* を発生させます。サブクラスはこのメソッドを実装しないままにしておいてもかまいません。

filename はクッキーを保存するファイルの名前です。*filename* が指定されない場合、self.filename

が使用されます (このデフォルト値は、それが存在する場合は、コンストラクタに渡されています)。`self.filename` も *None* の場合は *ValueError* が発生します。

ignore_discard: 破棄されるよう指示されていたクッキーでも保存します。*ignore_expires*: 期限の切れたクッキーでも保存します。

ここで指定されたファイルがもしすでに存在する場合は上書きされるため、以前にあったクッキーはすべて消去されます。保存したクッキーはあとで *load()* または *revert()* メソッドを使って復元することができます。

`FileCookieJar.load`(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)

ファイルからクッキーを読み込みます。

それまでのクッキーは新しいものに上書きされない限り残ります。

ここでの引数の値は *save()* と同じです。

名前のついたファイルはこのクラスがわかるやり方で指定する必要があります。さもないと *LoadError* が発生します。さらに、例えばファイルが存在しないような時に *OSError* が発生する場合があります。

バージョン 3.3 で変更: 以前は *IOError* が送出されました; それは現在 *OSError* のエイリアスです。

`FileCookieJar.revert`(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)

すべてのクッキーを破棄し、保存されているファイルから読み込み直します。

revert() は *load()* と同じ例外を発生させる事ができます。失敗した場合、オブジェクトの状態は変更されません。

FileCookieJar インスタンスは以下のような公開の属性をもっています:

`FileCookieJar.filename`

クッキーを保存するデフォルトのファイル名を指定します。この属性には代入することができます。

`FileCookieJar.delayload`

真であれば、クッキーを読み込むさいにディスクから遅延読み込みします。この属性には代入することができません。この情報は単なるヒントであり、(ディスク上のクッキーが変わらない限りは) インスタンスのふるまいには影響を与えず、パフォーマンスのみに影響します。*CookieJar* オブジェクトはこの値を無視することもあります。標準ライブラリに含まれている *FileCookieJar* クラスで遅延読み込みをおこなうものはありません。

21.24.2 FileCookieJar のサブクラスと web ブラウザとの連携

クッキーの読み書きのために、以下の *CookieJar* サブクラスが提供されています。

`class http.cookiejar.MozillaCookieJar`(*filename*, *delayload=None*, *policy=None*)

Mozilla の `cookies.txt` ファイル形式 (この形式はまた Lynx と Netscape ブラウザによっても使われています) でディスクにクッキーを読み書きするための *FileCookieJar* です。

注釈: このクラスは **RFC 2965** クッキーに関する情報を失います。また、より新しいか、標準でな

い port などのクッキー属性についての情報も失います。

警告: もしクッキーの損失や欠損が望ましくない場合は、クッキーを保存する前にバックアップを取っておくようにしてください (ファイルへの読み込み / 保存をくり返すと微妙な変化が生じる場合があります)。

また、Mozilla の起動中にクッキーを保存すると、Mozilla によって内容が破壊されてしまうことにも注意してください。

class http.cookiejar.**LWPCookieJar**(*filename*, *delayload=None*, *policy=None*)

libwww-perl のライブラリである Set-Cookie3 ファイル形式でディスクにクッキーを読み書きするための *FileCookieJar* です。これはクッキーを人間に可読な形式で保存するのに向いています。

バージョン 3.8 で変更: *filename* 引数が *path-like object* を受け付けるようになりました。

21.24.3 CookiePolicy オブジェクト

CookiePolicy インターフェイスを実装するオブジェクトは以下のようなメソッドを持っています:

CookiePolicy.**set_ok**(*cookie*, *request*)

クッキーがサーバから受け入れられるべきかどうかを表わす boolean 値を返します。

cookie は *Cookie* インスタンスです。*request* は *CookieJar.extract_cookies()* の説明で定義されているインターフェイスを実装するオブジェクトです。

CookiePolicy.**return_ok**(*cookie*, *request*)

クッキーがサーバに返されるべきかどうかを表わす boolean 値を返します。

cookie は *Cookie* インスタンスです。*request* は *CookieJar.add_cookie_header()* の説明で定義されているインターフェイスを実装するオブジェクトです。

CookiePolicy.**domain_return_ok**(*domain*, *request*)

与えられたクッキーのドメインに対して、そこにクッキーを返すべきでない場合には False を返します。

このメソッドは高速化のためのものです。これにより、すべてのクッキーをある特定のドメインに対してチェックする (これには多数のファイル読みこみを伴なう場合があります) 必要がなくなります。*domain_return_ok()* および *path_return_ok()* の両方から true が返された場合、すべての決定は *return_ok()* に委ねられます。

もし、このクッキードメインに対して *domain_return_ok()* が true を返すと、つぎにそのクッキーのパス名に対して *path_return_ok()* が呼ばれます。そうでない場合、そのクッキードメインに対する *path_return_ok()* および *return_ok()* は決して呼ばれることはありません。*path_return_ok()* が true を返すと、*return_ok()* がその *Cookie* オブジェクト自身の全チェックのために呼ばれます。そうでない場合、そのクッキーパス名に対する *return_ok()* は決して呼ばれることはありません。

注意: *domain_return_ok()* は *request* ドメインだけではなく、すべての *cookie* ドメインに対して呼ばれます。たとえば request ドメインが "www.example.com" だった場合、この関数は ".example.com" および "www.example.com" の両方に対して呼ばれることがあります。同じことは *path_return_ok()* にもいえます。

request 引数は *return_ok()* で説明されているとおりです。

CookiePolicy.**path_return_ok**(*path*, *request*)

与えられたクッキーのパス名に対して、そこにクッキーを返すべきでない場合には False を返します。

domain_return_ok() の説明を参照してください。

上のメソッドの実装にくわえて、*CookiePolicy* インターフェイスの実装では以下の属性を設定する必要があります。これはどのプロトコルがどのように使われるべきかを示すもので、これらの属性にはすべて代入することが許されています。

CookiePolicy.**netscape**

Netscape プロトコルを実装していることを示します。

CookiePolicy.**rfc2965**

RFC 2965 プロトコルを実装していることを示します。

CookiePolicy.**hide_cookie2**

Cookie2 ヘッダをリクエストに含めないようにします (このヘッダが存在する場合、私たちは **RFC 2965** クッキーを理解するということをサーバに示すことになります)。

もっとも有用な方法は、*DefaultCookiePolicy* をサブクラス化した *CookiePolicy* クラスを定義して、いくつか (あるいはすべて) のメソッドをオーバーライドすることでしょう。*CookiePolicy* 自体はどのようなクッキーも受け入れて設定を許可する「ポリシー無し」ポリシーとして使うこともできます (これが役に立つことはあまりありませんが)。

21.24.4 DefaultCookiePolicy オブジェクト

クッキーを受けつけ、またそれを返す際の標準的なルールを実装します。

RFC 2965 クッキーと Netscape クッキーの両方に対応しています。デフォルトでは、RFC 2965 の処理はオフになっています。

自分のポリシーを提供するいちばん簡単な方法は、このクラスを継承して、自分用の追加チェックの前にオーバーライドした元のメソッドを呼び出すことです:

```
import http.cookiejar
class MyCookiePolicy(http.cookiejar.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not http.cookiejar.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

CookiePolicy インターフェイスを実装するのに必要な機能に加えて、このクラスではクッキーを受けとったり設定したりするドメインを許可したり拒絶したりできるようになっています。ほかにも、Netscape プロトコルのかなり緩い規則をややきつくするために、いくつかの厳密性のスイッチがついています (いくつかの良性クッキーをブロックする危険性もありますが)。

ドメインのブラックリスト機能やホワイトリスト機能も提供されています (デフォルトではオフになっています)。ブラックリストになく、(ホワイトリスト機能を使用している場合は) ホワイトリストにあるドメインのみがクッキーを設定したり返したりすることを許可されます。コンストラクタの引数 *blocked_domains* 、および `blocked_domains()` と `set_blocked_domains()` メソッドを使ってください (*allowed_domains* に関しても同様の対応する引数とメソッドがあります)。ホワイトリストを設定した場合は、それを *None* にすることでホワイトリスト機能をオフにすることができます。

ブラックリストあるいはホワイトリスト中にあるドメインのうち、ドット (.) で始まっていないものは、正確にそれと一致するドメインのクッキーにしか適用されません。たとえばブラックリスト中のエントリ `"example.com"` は、`"example.com"` にはマッチしますが、`"www.example.com"` にはマッチしません。一方ドット (.) で始まっているドメインは、より特化されたドメインともマッチします。たとえば、`".example.com"` は、`"www.example.com"` と `"www.coyote.example.com"` の両方にマッチします (が、`"example.com"` 自身にはマッチしません)。IP アドレスは例外で、つねに正確に一致する必要があります。たとえば、かりに *blocked_domains* が `"192.168.1.2"` と `".168.1.2"` を含んでいたとして、192.168.1.2 はブロックされますが、193.168.1.2 はブロックされません。

DefaultCookiePolicy は以下のような追加メソッドを実装しています:

`DefaultCookiePolicy.blocked_domains()`
ブロックしているドメインのシーケンスを (タプルとして) 返します。

`DefaultCookiePolicy.set_blocked_domains(`*blocked_domains*`)`
ブロックするドメインを設定します。

`DefaultCookiePolicy.is_blocked(`*domain*`)`
domain がクッキーを授受しないブラックリストに載っているかどうかを返します。

`DefaultCookiePolicy.allowed_domains()`
None あるいは明示的に許可されているドメインを (タプルとして) 返します。

`DefaultCookiePolicy.set_allowed_domains(`*allowed_domains*`)`
許可するドメイン、あるいは *None* を設定します。

`DefaultCookiePolicy.is_not_allowed(`*domain*`)`
domain がクッキーを授受するホワイトリストに載っているかどうかを返します。

DefaultCookiePolicy インスタンスは以下の属性をもっています。これらはすべてコンストラクタから同じ名前の引数をつかって初期化することができ、代入してもかまいません。

`DefaultCookiePolicy.rfc2109_as_netscape`
真の場合、*CookieJar* のインスタンスに **RFC 2109** クッキー (即ち *Set-Cookie* ヘッダのクッキー属性 Version の値が 1 のクッキー) を Netscape クッキーへ、*Cookie* インスタンスの version 属性を 0 に設定する事でダウングレードするように要求します。デフォルトの値は *None* であり、この場合

RFC 2109 クッキーは **RFC 2965** 処理が無効に設定されている場合に限りダウングレードされます。それ故に RFC 2109 クッキーはデフォルトではダウングレードされます。

一般的な厳密性のスイッチ:

DefaultCookiePolicy.strict_domain
サイトに、国別コードとトップレベルドメインだけからなるドメイン名 (`.co.uk`, `.gov.uk`, `.co.nz` など) を設定させないようにします。これは完璧からはほど遠い実装であり、いつもうまくいくとは限りません!

RFC 2965 プロトコルの厳密性に関するスイッチ:

DefaultCookiePolicy.strict_rfc2965_unverifiable
検証不可能なトランザクション (通常これはリダイレクトか、別のサイトがホスティングしているイメージの読み込み要求です) に関する **RFC 2965** の規則に従います。この値が偽の場合、検証可能性を基準にしてクッキーがブロックされることは **決して** ありません

Netscape プロトコルの厳密性に関するスイッチ:

DefaultCookiePolicy.strict_ns_unverifiable
検証不可能なトランザクションに関する **RFC 2965** の規則を Netscape クッキーに対しても適用します。

DefaultCookiePolicy.strict_ns_domain
Netscape クッキーに対するドメインマッチングの規則をどの程度厳しくするかを指示するフラグです。とりうる値については下の説明を見てください。

DefaultCookiePolicy.strict_ns_set_initial_dollar
Set-Cookie: ヘッダで、`'$'` で始まる名前のクッキーを無視します。

DefaultCookiePolicy.strict_ns_set_path
要求した URI にパスがマッチしないクッキーの設定を禁止します。

`strict_ns_domain` はいくつかのフラグの集合です。これはいくつかの値を or することで構成します (たとえば `DomainStrictNoDots|DomainStrictNonDomain` は両方のフラグが設定されていることになります)。

DefaultCookiePolicy.DomainStrictNoDots
クッキーを設定するさい、ホスト名のプレフィクスにドットが含まれるのを禁止します (例: `www.foo.bar.com` は `.bar.com` のクッキーを設定することはできません、なぜなら `www.foo` はドットを含んでいるからです)。

DefaultCookiePolicy.DomainStrictNonDomain
`domain` クッキー属性を明示的に指定していないクッキーは、そのクッキーを設定したドメインと同一のドメインだけに返されます (例: `example.com` からのクッキーに `domain` クッキー属性がない場合、そのクッキーが `spam.example.com` に返されることはありません)。

DefaultCookiePolicy.DomainRFC2965Match
クッキーを設定するさい、**RFC 2965** の完全ドメインマッチングを要求します。

以下の属性は上記のフラグのうちもっともよく使われる組み合わせで、便宜をはかるために提供されています:

DefaultCookiePolicy.DomainLiberal
: 0 と同じです (つまり、上述の Netscape のドメイン厳密性フラグがすべてオフにされます)。

DefaultCookiePolicy.DomainStrict
: DomainStrictNoDots|DomainStrictNonDomain と同じです。

21.24.5 Cookie オブジェクト

Cookie インスタンスは、さまざまなクッキーの標準で規定されている標準的なクッキー属性とおおまかに対応する Python 属性をもっています。しかしデフォルト値を決める複雑なやり方が存在しており、また max-age および expires クッキー属性は同じ値をもつことになっているので、また **RFC 2109** クッキーは *http.cookiejar* によって version 1 から version 0 (Netscape) クッキーへ ' ダウングレード' される場合があるため、この対応は 1 対 1 ではありません。

CookiePolicy メソッド内でのごくわずかな例外を除けば、これらの属性に代入する必要はないはずです。このクラスは内部の一貫性を保つようにはしていないため、代入するのは自分のやっていることを理解している場合のみにしてください。

Cookie.version
: 整数または *None* 。Netscape クッキーはバージョン 0 であり、**RFC 2965** および **RFC 2109** クッキーはバージョン 1 です。しかし、*http.cookiejar* は RFC 2109 クッキーを Netscape クッキー (*version* が 0) に ' ダウングレード' する場合がある事に注意して下さい。

Cookie.name
: クッキーの名前 (文字列)。

Cookie.value
: クッキーの値 (文字列)、あるいは *None* 。

Cookie.port
: ポートあるいはポートの集合をあらわす文字列 (例: '80' または '80,8080')、あるいは *None* 。

Cookie.path
: クッキーのパス名 (文字列、例: '/acme/rocket_launchers')。

Cookie.secure
: そのクッキーを返せるのがセキュアな接続のみならば True を返します。

Cookie.expires
: クッキーの期限が切れる日時をあわらす整数 (エポックから経過した秒数)、あるいは *None* 。*is_expired()* も参照してください。

Cookie.discard
: これがセッションクッキーであれば True を返します。

Cookie.**comment**
: このクッキーの働きを説明する、サーバからのコメント文字列、あるいは *None* 。

Cookie.**comment_url**
: このクッキーの働きを説明する、サーバからのコメントのリンク URL、あるいは *None* 。

Cookie.**rfc2109**
: **RFC 2109** クッキー (即ち *Set-Cookie* ヘッダにあり、かつクッキー属性 Version の値が 1 のクッキー) の場合、True を返します。:mod:“http.cookiejar‘ が RFC 2109 クッキーを Netscape クッキー (*version* が 0) に ’ダウングレード’ する場合があるので、この属性が提供されています。

Cookie.**port_specified**
: サーバがポート、あるいはポートの集合を (*Set-Cookie* / *Set-Cookie2* ヘッダ内で) 明示的に指定していれば True を返します。

Cookie.**domain_specified**
: サーバにより明示的にドメインが指定されていれば True を返します。

Cookie.**domain_initial_dot**
: サーバが明示的に指定したドメインがドット ('.') で始まっていれば True を返します。

クッキーは、オプションとして標準的でないクッキー属性を持つこともできます。これらは以下のメソッドでアクセスできます:

Cookie.**has_nonstandard_attr**(*name*)
: そのクッキーが指定された名前のクッキー属性をもっている場合には True を返します。

Cookie.**get_nonstandard_attr**(*name*, *default=None*)
: クッキーが指定された名前のクッキー属性をもっていれば、その値を返します。そうでない場合は *default* を返します。

Cookie.**set_nonstandard_attr**(*name*, *value*)
: 指定された名前のクッキー属性を設定します。

Cookie クラスは以下のメソッドも定義しています:

Cookie.**is_expired**(*now=None*)
: サーバが要求するクッキーの有効期限を過ぎていれば True を返します。*now* が (エポックからの経過秒で) 指定されているときは、特定の時刻で期限切れかどうかを判定します。

21.24.6 使用例

はじめに、もっとも一般的な *http.cookiejar* の使用例をあげます:

```
import http.cookiejar, urllib.request
cj = http.cookiejar.CookieJar()
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

以下の例では、URL を開く際に Netscape や Mozilla または Lynx のクッキーを使う方法を示しています (クッキーファイルの位置は Unix/Netscape の慣例にしたがうものと仮定しています):

```
import os, http.cookiejar, urllib.request
cj = http.cookiejar.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".netscape", "cookies.txt"))
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

つぎの例は *DefaultCookiePolicy* の使用例です。**RFC 2965** クッキーをオンにし、Netscape クッキーを設定したり返したりするドメインに対してより厳密な規則を適用します。そしていくつかのドメインからクッキーを設定あるいは返還するのをブロックしています:

```
import urllib.request
from http.cookiejar import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=Policy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

21.25 xmlrpc --- XMLRPC サーバーとクライアントモジュール

XML-RPC は HTTP 経由の XML を使って遠隔手続き呼び出し (Remote Procedure Call) を実現する方法です。XML-RPC を使うと、クライアントはリモートサーバー (サーバーは URI で名前付けられます) 上のメソッドを引数付きで呼び出して、構造化されたデータを受け取る事ができます。

xmlrpc パッケージは XML-RPC のサーバーとクライアントを実装したモジュールを持っています。モジュール一覧:

- *xmlrpc.client*
- *xmlrpc.server*

21.26 xmlrpc.client --- XML-RPC クライアントアクセス

ソースコード: Lib/xmlrpc/client.py

XML-RPC は XML を利用した遠隔手続き呼び出し (Remote Procedure Call) の一種で、HTTP(S) をトランスポートとして使用します。XML- RPC では、クライアントはリモートサーバ (URI で指定されたサーバ) 上のメソッドをパラメータを指定して呼び出し、構造化されたデータを取得します。このモジュールは、XML-RPC クライアントの開発をサポートしており、Python オブジェクトに適合する転送用 XML の変換の全てを行います。

警告: *xmlrpc.client* モジュールは悪意を持って構築されたデータに対して安全ではありません。信頼できないデータや認証されていないデータを解析する必要がある場合は、*XML* **の脆弱性** を参照してください。

バージョン 3.5 で変更: HTTPS URI の場合、*xmlrpc.client* は必要な証明書検証とホスト名チェックを全てデフォルトで行います。

class xmlrpc.client.**ServerProxy**(*uri*, *transport=None*, *encoding=None*, *verbose=False*, *allow_none=False*, *use_datetime=False*, *use_builtin_types=False*, ***, *headers=()*, *context=None*)

ServerProxy は、リモートの XML-RPC サーバとの通信を管理するオブジェクトです。最初のパラメータは URI(Uniform Resource Indicator) で、通常はサーバの URL を指定します。2 番目のパラメータにはトランスポート・ファクトリを指定する事ができます。トランスポート・ファクトリを省略した場合、URL が https: ならモジュール内部の `SafeTransport` インスタンスを使用し、それ以外の場合にはモジュール内部の `Transport` インスタンスを使用します。オプションの 3 番目の引数はエンコード方法で、デフォルトでは UTF-8 です。オプションの 4 番目の引数はデバッグフラグです。

The following parameters govern the use of the returned proxy instance. If *allow_none* is true, the Python constant `None` will be translated into XML; the default behaviour is for `None` to raise a *TypeError*. This is a commonly-used extension to the XML-RPC specification, but isn't supported by all clients and servers; see http://ontosys.com/xml-rpc/extensions.php for a description. The *use_builtin_types* flag can be used to cause date/time values to be presented as *datetime.datetime* objects and binary data to be presented as *bytes* objects; this flag is false by default. *datetime.datetime*, *bytes* and *bytearray* objects may be passed to calls. The *headers* parameter is an optional sequence of HTTP headers to send with each request, expressed as a sequence of 2-tuples representing the header name and value. (e.g. *[('Header-Name', 'value')]*). The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

バージョン 3.3 で変更: *use_builtin_types* フラグが追加されました。

バージョン 3.8 で変更: *headers* 引数が追加されました。

HTTP 及び HTTPS 通信の両方で、`http://user:pass@host:port/path` のような HTTP 基本認証のための拡張 URL 構文をサポートしています。`user:pass` は base64 でエンコードして HTTP の'Authorization ' ヘッダとなり、XML-RPC メソッド呼び出し時に接続処理の一部としてリモートサーバに送信されます。リモートサーバが基本認証を要求する場合のみ、この機能を利用する必要があります。HTTPS URL が与えられたなら、*context* は *ssl.SSLContext* にでき、基底の HTTP 接続の SSL 設定を設定します。

生成されるインスタンスはリモートサーバへのプロクシオブジェクトで、RPC 呼び出しを行う為のメソッドを持ちます。リモートサーバがイントロスペクション API をサポートしている場合は、リモートサーバのサポートするメソッドを検索 (サービス検索) やサーバのメタデータの取得なども行えます。

以下の型を XML に変換 (XML を通じてマーシャルする) する事ができます (特別な指定がない限り、逆変換でも同じ型として変換されます):

XML-RPC の型	Python の型
boolean	*bool*
int, i1, i2, i4, i8 または biginteger	*int* in range from -2147483648 to 2147483647. Values get the <int> tag.
double または float	*float*. Values get the <double> tag.
string	*str*
array	適合する要素を持つ *list* または *tuple*。array は *list* として返します。
struct	*dict* 。キーは文字列のみ。全ての値は変換可能でなくてはならない。ユーザー定義型を渡すこともできます。*__dict__* の属性のみ転送されます。
dateTime.iso8601	*DateTime* または *datetime.datetime*。返される型は *use_builtin_types* および *use_datetime* の値に依ります。
base64	*Binary*、*bytes* または *bytearray*。返される型は *use_builtin_types* フラグの値に依ります。
nil	None 定数。*allow_none* が真の場合にのみ渡すことが出来ます。
bigdecimal	*decimal.Decimal*. Returned type only.

上記の XML-RPC でサポートする全データ型を使用することができます。メソッド呼び出し時、XML- RPC サーバエラーが発生すると *Fault* インスタンスを送出し、HTTP/HTTPS トランスポート層でエラーが発生した場合には *ProtocolError* を送出します。Error をベースとする *Fault* と *ProtocolError* の両方が発生します。現在のところ xmlrpclib では組み込み型のサブクラスのインスタンスをマーシャルすることはできません。

文字列を渡す場合、<, >, & などの XML で特殊な意味を持つ文字は自動的にエスケープされます。しかし、ASCII 値 0〜31 の制御文字 (もちろん、タブ'TAB', 改行'LF', リターン'CR' は除く) などの XML で使用することのできない文字を使用することはできず、使用するとその XML-RPC リクエストは well-formed な XML とはなりません。そのようなバイト列を渡す必要がある場合は、*bytes*, *bytearray* クラスまたは後述の *Binary* ラッパクラスを使用してください。

Server は、後方互換性の為に *ServerProxy* の別名として残されています。新しいコードでは *ServerProxy* を使用してください。

バージョン 3.5 で変更: *context* 引数が追加されました。

バージョン 3.6 で変更: Added support of type tags with prefixes (e.g. ex:nil). Added support of unmarshalling additional types used by Apache XML-RPC implementation for numerics: i1, i2, i8, biginteger, float and bigdecimal. See http://ws.apache.org/xmlrpc/types.html for a description.

参考:

XML-RPC HOWTO　数種類のプログラミング言語で記述された XML-RPC の操作とクライアントソフトウェアの素晴らしい説明が掲載されています。XML- RPC クライアントの開発者が知っておくべきことがほとんど全て記載されています。

XML-RPC Introspection　インストロペクションをサポートする、XML-RPC プロトコルの拡張を解説しています。

XML-RPC Specification　公式の仕様

XML-RPC 非公式正誤表　Fredrik Lundh による "unofficial errata, intended to clarify certain details in the XML-RPC specification, as well as hint at 'best practices' to use when designing your own XML-RPC implementations."

21.26.1 ServerProxy オブジェクト

ServerProxy インスタンスの各メソッドはそれぞれ XML-RPC サーバの遠隔手続き呼び出しに対応しており、メソッドが呼び出されると名前と引数をシグネチャとして RPC を実行します (同じ名前のメソッドでも、異なる引数シグネチャによってオーバロードされます)。RPC 実行後、変換された値を返すか、または *Fault* オブジェクトもしくは *ProtocolError* オブジェクトでエラーを通知します。

予約属性 `system` から、XML イントロスペクション API の一般的なメソッドを利用する事ができます:

ServerProxy.system.listMethods()
XML-RPC サーバがサポートするメソッド名 (system 以外) を格納する文字列のリストを返します。

ServerProxy.system.methodSignature(*name*)
XML-RPC サーバで実装されているメソッドの名前を指定し、利用可能なシグネチャの配列を取得します。シグネチャは型のリストで、先頭の型は戻り値の型を示し、以降はパラメータの型を示します。

XML-RPC では複数のシグネチャ (オーバロード) を使用することができるので、単独のシグネチャではなく、シグネチャのリストを返します。

シグネチャは、メソッドが使用する最上位のパラメータにのみ適用されます。例えばあるメソッドのパラメータが構造体の配列で戻り値が文字列の場合、シグネチャは単に"string, array" となります。パラメータが三つの整数で戻り値が文字列の場合は"string, int, int, int" となります。

メソッドにシグネチャが定義されていない場合、配列以外の値が返ります。Python では、この値は list 以外の値となります。

ServerProxy.system.methodHelp(*name*)
XML-RPC サーバで実装されているメソッドの名前を指定し、そのメソッドを解説する文書文字列を取得します。文書文字列を取得できない場合は空文字列を返します。文書文字列には HTML マークアップが含まれます。

バージョン 3.5 で変更: Instances of *ServerProxy* support the *context manager* protocol for closing the underlying transport.

以下は、動作する例です。サーバ側のコード:

```
from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0
```

(次のページに続く)

(前のページからの続き)

```
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(is_even, "is_even")
server.serve_forever()
```

上記のサーバーに対するクライアントコード:

```
import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:8000/") as proxy:
    print("3 is even: %s" % str(proxy.is_even(3)))
    print("100 is even: %s" % str(proxy.is_even(100)))
```

21.26.2 DateTime オブジェクト

class xmlrpc.client.**DateTime**

このクラスは、エポックからの秒数、タプルで表現された時刻、ISO 8601 形式の時間/日付文字列、*datetime.datetime*, のインスタンスのいずれかで初期化することができます。このクラスには以下のメソッドがあり、主にコードをマーシャル/アンマーシャルするための内部処理を行います:

decode(*string*)

文字列をインスタンスの新しい時間を示す値として指定します。

encode(*out*)

出力ストリームオブジェクト *out* に、XML-RPC エンコーディングの *DateTime* 値を出力します。

また、比較と `__repr__()` で定義される演算子を使用することができます。

以下は、動作する例です。サーバ側のコード:

```
import datetime
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def today():
    today = datetime.datetime.today()
    return xmlrpc.client.DateTime(today)

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(today, "today")
server.serve_forever()
```

上記のサーバーに対するクライアントコード:

```
import xmlrpc.client
import datetime
```

(次のページに続く)

(前のページからの続き)

```
proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")

today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print("Today: %s" % converted.strftime("%d.%m.%Y, %H:%M"))
```

21.26.3 Binary オブジェクト

class xmlrpc.client.Binary

このクラスは、バイト列データ (NUL を含む) で初期化することができます。*Binary* の内容は、属性で参照します:

data

Binary インスタンスがカプセル化しているバイナリデータ。このデータは *bytes* オブジェクトです。

Binary オブジェクトは以下のメソッドを持ち、主に内部的にマーシャル/アンマーシャル時に使用されます:

decode(*bytes*)

指定された base64 *bytes* オブジェクトをデコードし、インスタンスのデータとします。

encode(*out*)

バイナリ値を base64 でエンコードし、出力ストリームオブジェクト *out* に出力します。

エンコードされたデータは、**RFC 2045 section 6.8** にある通り、76 文字ごとに改行されます。これは、XMC-RPC 仕様が作成された時のデ・ファクト・スタンダードの base64 です。

また、`__eq__()` および `__ne__()` メソッドで定義される演算子を使用することができます。

バイナリオブジェクトの使用例です。XML-RPC ごしに画像を転送します。

```
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpc.client.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(python_logo, 'python_logo')

server.serve_forever()
```

クライアントは画像を取得して、ファイルに保存します。

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
```

21.26.4 Fault オブジェクト

class xmlrpc.client.**Fault**

Fault オブジェクトは、XML-RPC の fault タグの内容をカプセル化しており、以下の属性を持ちます:

faultCode

失敗のタイプを示す文字列。

faultString

失敗の診断メッセージを含む文字列。

以下のサンプルでは、複素数型のオブジェクトを返そうとして、故意に *Fault* を起こしています。

```
from xmlrpc.server import SimpleXMLRPCServer

# A marshalling error is going to occur because we're returning a
# complex number
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(add, 'add')

server.serve_forever()
```

上記のサーバーに対するクライアントコード:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
try:
    proxy.add(2, 5)
except xmlrpc.client.Fault as err:
    print("A fault occurred")
    print("Fault code: %d" % err.faultCode)
    print("Fault string: %s" % err.faultString)
```

21.26.5 ProtocolError オブジェクト

class xmlrpc.client.ProtocolError

ProtocolError オブジェクトはトランスポート層で発生したエラー (URI で指定したサーバが見つからなかった場合に発生する 404 'not found' など) の内容を示し、以下の属性を持ちます:

url
エラーの原因となった URI または URL。

errcode
エラーコード。

errmsg
エラーメッセージまたは診断文字列。

headers
エラーの原因となった HTTP/HTTPS リクエストを含む辞書。

次の例では、不適切な URI を利用して、故意に *ProtocolError* を発生させています:

```
import xmlrpc.client

# create a ServerProxy with a URI that doesn't respond to XMLRPC requests
proxy = xmlrpc.client.ServerProxy("http://google.com/")

try:
    proxy.some_method()
except xmlrpc.client.ProtocolError as err:
    print("A protocol error occurred")
    print("URL: %s" % err.url)
    print("HTTP/HTTPS headers: %s" % err.headers)
    print("Error code: %d" % err.errcode)
    print("Error message: %s" % err.errmsg)
```

21.26.6 MultiCall オブジェクト

遠隔のサーバに対する複数の呼び出しをひとつのリクエストにカプセル化する方法は、http://www.xmlrpc.com/discuss/msgReader%241208 で示されています。

class xmlrpc.client.MultiCall(*server*)

巨大な (boxcar) メソッド呼び出しに使えるオブジェクトを作成します。*server* には最終的に呼び出しを行う対象を指定します。作成した MultiCall オブジェクトを使って呼び出しを行うと、即座に None を返し、呼び出したい手続き名とパラメタを *MultiCall* オブジェクトに保存するだけに留まります。オブジェクト自体を呼び出すと、それまでに保存しておいたすべての呼び出しを単一の system.multicall リクエストの形で伝送します。呼び出し結果は **ジェネレータ** になります。このジェネレータにわたってイテレーションを行うと、個々の呼び出し結果を返します。

以下は、このクラスの使用例です。サーバ側のコード:

```
from xmlrpc.server import SimpleXMLRPCServer

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x // y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

上記のサーバーに対するクライアントコード:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
multicall = xmlrpc.client.MultiCall(proxy)
multicall.add(7, 3)
multicall.subtract(7, 3)
multicall.multiply(7, 3)
multicall.divide(7, 3)
result = multicall()

print("7+3=%d, 7-3=%d, 7*3=%d, 7//3=%d" % tuple(result))
```

21.26.7 補助関数

xmlrpc.client.**dumps**(*params*, *methodname=None*, *methodresponse=None*, *encoding=None*, *allow_none=False*)

params を XML-RPC リクエストの形式に変換します。*methodresponse* が真の場合、XML-RPC レスポンスの形式に変換します。*params* に指定できるのは、引数からなるタプルか *Fault* 例外クラスのインスタンスです。*methodresponse* が真の場合、単一の値だけを返します。従って、*params* の長さも 1 でなければなりません。*encoding* を指定した場合、生成される XML のエンコード方式になります。デフォルトは UTF-8 です。Python の *None* は標準の XML-RPC には利用できません。*None* を使えるようにするには、*allow_none* を真にして、拡張機能つきにしてください。

xmlrpc.client.loads(*data*, *use_datetime=False*, *use_builtin_types=False*)

XML-RPC リクエストまたはレスポンスを (params, methodname) の形式をとる Python オブジェクトにします。*params* は引数のタプルです。*methodname* は文字列で、パケット中にメソッド名がない場合には None になります。例外条件を示す XML-RPC パケットの場合には、*Fault* 例外を送出します。*use_builtin_types* フラグは *datetime.datetime* のオブジェクトとして日付/時刻を、*bytes* のオブジェクトとしてバイナリデータを表現する時に使用し、デフォルトでは false に設定されています。

非推奨となった *use_datetime* フラグは *use_builtin_types* に似ていますが date/time 値にのみ適用されます。

バージョン 3.3 で変更: *use_builtin_types* フラグが追加されました。

21.26.8 クライアントのサンプル

```
# simple test program (from the XML-RPC specification)
from xmlrpc.client import ServerProxy, Error

# server = ServerProxy("http://localhost:8000") # local server
with ServerProxy("http://betty.userland.com") as proxy:

    print(proxy)

    try:
        print(proxy.examples.getStateName(41))
    except Error as v:
        print("ERROR", v)
```

XML-RPC サーバに HTTP プロキシを経由して接続する場合、カスタムトランスポートを定義する必要があります。以下に例を示します:

```
import http.client
import xmlrpc.client

class ProxiedTransport(xmlrpc.client.Transport):

    def set_proxy(self, host, port=None, headers=None):
        self.proxy = host, port
        self.proxy_headers = headers

    def make_connection(self, host):
        connection = http.client.HTTPConnection(*self.proxy)
        connection.set_tunnel(host, headers=self.proxy_headers)
        self._connection = host, connection
        return connection

transport = ProxiedTransport()
transport.set_proxy('proxy-server', 8080)
server = xmlrpc.client.ServerProxy('http://betty.userland.com', transport=transport)
```

(次のページに続く)

(前のページからの続き)

```
print(server.examples.getStateName(41))
```

21.26.9 クライアントとサーバーの利用例

SimpleXMLRPCServer の例 を参照してください。

脚注

21.27 xmlrpc.server --- 基本的な XML-RPC サーバー

ソースコード: Lib/xmlrpc/server.py

---

xmlrpc.server モジュールは Python で記述された基本的な XML-RPC サーバーフレームワークを提供します。サーバーはスタンドアロンであるか、*SimpleXMLRPCServer* を使うか、*CGIXMLRPCRequestHandler* を使って CGI 環境に組み込まれるかの、いずれかです。

> **警告:** *xmlrpc.server* モジュールは悪意を持って構築されたデータに対して安全ではありません。信頼できないデータや認証されていないデータを解析する必要がある場合は、*XML* の脆弱性 を参照してください。

class xmlrpc.server.**SimpleXMLRPCServer**(*addr*, *requestHandler=SimpleXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=False*)

サーバーインスタンスを新たに作成します。このクラスは XML-RPC プロトコルで呼ばれる関数の登録のためのメソッドを提供します。引数 *requestHandler* にはリクエストハンドラーインスタンスのファクトリーを設定します。デフォルトは *SimpleXMLRPCRequestHandler* です。引数 *addr* と *requestHandler* は *socketserver.TCPServer* のコンストラクターに渡されます。*logRequests* が真の場合 (デフォルト)、リクエストはログに記録されます。この引数を偽にするとはログは記録されません。引数 *allow_none* と *encoding* は *xmlrpc.client* に渡され、サーバーが返す XML-RPC 応答を制御します。*bind_and_activate* 引数はコンストラクタが直ちに `server_bind()` と `server_activate()` を呼ぶかどうかを制御します。デフォルトでは真です。この引数に `False` を設定することで、アドレスを束縛する前に *allow_reuse_address* クラス変数を操作することが出来ます。*use_builtin_types* 引数は *loads()* 関数に渡されます。この引数は日付/時刻の値やバイナリデータを受け取ったときにどの型が処理されるかを制御します。デフォルトでは偽です。

バージョン 3.3 で変更: *use_builtin_types* フラグが追加されました。

class xmlrpc.server.**CGIXMLRPCRequestHandler**(*allow_none=False*, *encoding=None*, *use_builtin_types=False*)

CGI 環境における XML-RPC リクエストハンドラーを新たに作成します。引数 *allow_none* と *encod-*

ing は *xmlrpc.client* に渡され、サーバーが返す XML-RPC 応答を制御します。*use_builtin_types* 引数は *loads()* 関数に渡されます。この引数は日付/時刻の値やバイナリデータを受け取ったときにどの型が処理されるかを制御します。デフォルトは偽です。

バージョン 3.3 で変更: *use_builtin_types* フラグが追加されました。

class xmlrpc.server.**SimpleXMLRPCRequestHandler**

リクエストハンドラーインスタンスを新たに作成します。このリクエストハンドラーは `POST` リクエストをサポートし、*SimpleXMLRPCServer* コンストラクターの引数 *logRequests* に従ってログ出力を行います。

21.27.1 SimpleXMLRPCServer オブジェクト

SimpleXMLRPCServer クラスは *socketserver.TCPServer* のサブクラスで、基本的なスタンドアロンの XML-RPC サーバーを作成する手段を提供します。

SimpleXMLRPCServer.**register_function**(*function=None*, *name=None*)

XML- RPC リクエストに応じる関数を登録します。引数 *name* が与えられている場合はその値が、関数 *function* に関連付けられます。これが与えられない場合は `function.__name__` の値が用いられます。引数 *name* は文字列で、Python で識別子として正しくない文字 (" . " ピリオドなど) を含んでいても構いません。

This method can also be used as a decorator. When used as a decorator, *name* can only be given as a keyword argument to register *function* under *name*. If no *name* is given, `function.__name__` will be used.

バージョン 3.7 で変更: *register_function()* can be used as a decorator.

SimpleXMLRPCServer.**register_instance**(*instance*, *allow_dotted_names=False*)

オブジェクトを登録します。オブジェクトは *register_function()* を使用して登録されていないメソッド名を公開するのに使われます。*instance* に `_dispatch()` メソッドがあった場合、リクエストされたメソッド名と引数で `_dispatch()` を呼び出します。API は `def _dispatch(self, method, params)` (*params* 可変引数リストではないことに注意) です。タスクを実行するのに下層の関数を呼び出す場合、その関数は `func(*params)` のように引数リストを展開して呼び出されます。`_dispatch()` の返り値は結果としてクライアントに返されます。*instance* に `_dispatch()` メソッドがない場合、リクエストされたメソッド名にマッチする属性を検索します。

オプション引数 *allow_dotted_names* が真でインスタンスに `_dispatch()` メソッドがない場合、リクエストされたメソッド名がピリオドを含むなら、メソッド名の各要素が個々に検索され、簡単な階層的検索が行われます。その検索で発見された値をリクエストの引数で呼び出し、クライアントに返り値を返します。

> **警告:** *allow_dotted_names* オプションを有効にすると、侵入者はあなたのモジュールのグローバル変数にアクセスすることができ、あなたのマシンで任意のコードを実行できる可能性があります。このオプションは閉じた安全なネットワークでのみお使い下さい。

SimpleXMLRPCServer.**register_introspection_functions**()
: XML-RPC のイントロスペクション関数、system.listMethods 、system.methodHelp 、system.methodSignature を登録します。

SimpleXMLRPCServer.**register_multicall_functions**()
: XML-RPC における複数の要求を処理する関数 system.multicall を登録します。

SimpleXMLRPCRequestHandler.**rpc_paths**
: この属性値は XML-RPC リクエストを受け付ける URL の有効なパス部分をリストするタプルでなければなりません。これ以外のパスへのリクエストは 404「そのようなページはありません」HTTP エラーになります。このタプルが空の場合は全てのパスが有効であると見なされます。デフォルト値は ('/', '/RPC2') です。

SimpleXMLRPCServer の例

サーバーのコード:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name
    def adder_function(x, y):
        return x + y
    server.register_function(adder_function, 'add')

    # Register an instance; all the methods of the instance are
    # published as XML-RPC methods (in this case, just 'mul').
    class MyFuncs:
        def mul(self, x, y):
            return x * y

    server.register_instance(MyFuncs())

    # Run the server's main loop
    server.serve_forever()
```

以下のクライアントコードは上のサーバーで使えるようになったメソッドを呼び出します:

```
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')
print(s.pow(2,3))  # Returns 2**3 = 8
print(s.add(2,3))  # Returns 5
print(s.mul(5,2))  # Returns 5*2 = 10

# Print list of available methods
print(s.system.listMethods())
```

`register_function()` can also be used as a decorator. The previous server example can register functions in a decorator way:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name, using
    # register_function as a decorator. *name* can only be given
    # as a keyword argument.
    @server.register_function(name='add')
    def adder_function(x, y):
        return x + y

    # Register a function under function.__name__.
    @server.register_function
    def mul(x, y):
        return x * y

    server.serve_forever()
```

`Lib/xmlrpc/server.py` モジュール内にある以下の例はドット付名前を許容し複数呼び出し関数を登録するサーバです。

警告: *allow_dotted_names* オプションを有効にすると、侵入者はあなたのモジュールのグローバル変数にアクセスすることができ、あなたのマシンで任意のコードを実行できる可能性があります。この例は閉じた安全なネットワークでのみお使い下さい。

```
import datetime

class ExampleService:
    def getData(self):
        return '42'

    class currentTime:
        @staticmethod
        def getCurrentTime():
            return datetime.datetime.now()

with SimpleXMLRPCServer(("localhost", 8000)) as server:
    server.register_function(pow)
    server.register_function(lambda x,y: x+y, 'add')
    server.register_instance(ExampleService(), allow_dotted_names=True)
    server.register_multicall_functions()
    print('Serving XML-RPC on localhost port 8000')
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("\nKeyboard interrupt received, exiting.")
        sys.exit(0)
```

この ExampleService デモはコマンドラインから起動することができます。

```
python -m xmlrpc.server
```

上記のサーバとやりとりするクライアントは *Lib/xmlrpc/client.py* にあります:

```
server = ServerProxy("http://localhost:8000")

try:
    print(server.currentTime.getCurrentTime())
except Error as v:
    print("ERROR", v)

multi = MultiCall(server)
multi.getData()
multi.pow(2,9)
multi.add(1,2)
try:
    for response in multi():
        print(response)
except Error as v:
    print("ERROR", v)
```

デモ XMLRPC サーバとやりとりするクライアントは以下のように呼び出します:

```
python -m xmlrpc.client
```

21.27.2 CGIXMLRPCRequestHandler

CGIXMLRPCRequestHandler クラスは、Python の CGI スクリプトに送られた XML-RPC リクエストを処理するときに使用できます。

CGIXMLRPCRequestHandler.**register_function**(*function=None*, *name=None*)

XML- RPC リクエストに応じる関数を登録します。引数 *name* が与えられている場合はその値が、関数 *function* に関連付けられます。これが与えられない場合は `function.__name__` の値が用いられます。引数 *name* は文字列で、Python で識別子として正しくない文字 (" . " ピリオドなど) を含んでいても構いません。

This method can also be used as a decorator. When used as a decorator, *name* can only be given as a keyword argument to register *function* under *name*. If no *name* is given, `function.__name__` will be used.

バージョン 3.7 で変更: *register_function()* can be used as a decorator.

CGIXMLRPCRequestHandler.**register_instance**(*instance*)

オブジェクトを登録します。オブジェクトは *register_function()* を使用して登録されていないメソッド名を公開するのに使われます。*instance* に `_dispatch()` メソッドがあった場合、リクエストされたメソッド名と引数で `_dispatch()` を呼び出します。返り値は結果としてクライアントに返されます。*instance* に `_dispatch()` メソッドがなかった場合、リクエストされたメソッド名にマッチする属性を検索します。リクエストされたメソッド名がピリオドを含む場合、モジュール名の各要素が個々に検索され、簡単な階層的検索が実行されます。その検索で発見された値をリクエストの引数で呼び出し、クライアントに返り値を返します。

CGIXMLRPCRequestHandler.**register_introspection_functions**()

XML-RPC のイントロスペクション関数、`system.listMethods` 、`system.methodHelp` 、`system.methodSignature` を登録します。

CGIXMLRPCRequestHandler.**register_multicall_functions**()

XML-RPC マルチコール関数 `system.multicall` を登録します。

CGIXMLRPCRequestHandler.**handle_request**(*request_text=None*)

XML-RPC リクエストを処理します。与えられた場合、*request_text* は HTTP サーバが提供する POST データでなければなりません。そうでない場合、標準入力の内容が使われます。

以下はプログラム例です:

```
class MyFuncs:
    def mul(self, x, y):
        return x * y


handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
```

(次のページに続く)

(前のページからの続き)

```
handler.register_instance(MyFuncs())
handler.handle_request()
```

21.27.3 XMLRPC サーバの文書化

これらのクラスは HTTP GET 要求への応答内で HTML 文書となるよう上記クラスを拡張します。サーバは独立していても CGI 環境に埋め込まれていてもかまいません。前者では *DocXMLRPCServer* を、後者では *DocCGIXMLRPCRequestHandler* を使用します。

class xmlrpc.server.**DocXMLRPCServer**(*addr*, *requestHandler=DocXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=True*)

サーバ・インスタンスを新たに生成します。全ての引数の意味は *SimpleXMLRPCServer* のものと同じですが、*requestHandler* のデフォルトは *DocXMLRPCRequestHandler* になっています。

バージョン 3.3 で変更: *use_builtin_types* フラグが追加されました。

class xmlrpc.server.**DocCGIXMLRPCRequestHandler**

CGI 環境で XMR-RPC リクエストを処理するインスタンスを新たに生成します。

class xmlrpc.server.**DocXMLRPCRequestHandler**

リクエスト・ハンドラのインスタンスを新たに生成します。このリクエスト・ハンドラは XML-RPC POST 要求とドキュメントの GET 要求をサポートし、*DocXMLRPCServer* コンストラクタに与えられた引数 *logRequests* を優先するためにロギングを変更します。

21.27.4 DocXMLRPCServer オブジェクト

DocXMLRPCServer は *SimpleXMLRPCServer* の派生クラスで、自己文書化するスタンドアローン XML-RPC サーバの作成手段を提供します。HTTP POST リクエストは XML-RPC メソッドの呼び出しとして処理されます。HTTP GET リクエストは pydoc スタイルの HTML 文書の生成に処理されます。これによりサーバは自身の web ベースの文書を提供できます。

DocXMLRPCServer.**set_server_title**(*server_title*)

生成する HTML 文書で使用されるタイトルを設定します。このタイトルは HTML の title 要素内で使われます。

DocXMLRPCServer.**set_server_name**(*server_name*)

生成する HTML 文書内で使用される名前を設定します。この名前は生成した文書冒頭の h1 要素内で使われます。

DocXMLRPCServer.**set_server_documentation**(*server_documentation*)

生成する HTML 文書内で使用される説明を設定します。この説明は文書中のサーバ名の下にパラグラフとして出力されます。

21.27.5 DocCGIXMLRPCRequestHandler

DocCGIXMLRPCRequestHandler は *CGIXMLRPCRequestHandler* の派生クラスで、自己文書化する XML-RPC CGI スクリプトの作成手段を提供します。HTTP POST リクエストは XML-RCP メソッドの呼び出しとして処理されます。HTTP GET リクエストは pydoc スタイルの HTML 文書の生成に処理されます。これによりサーバは自身の web ベースの文書を提供できます。

DocCGIXMLRPCRequestHandler.**set_server_title**(*server_title*)
生成する HTML 文書で使用されるタイトルを設定します。このタイトルは HTML の title 要素内で使われます。

DocCGIXMLRPCRequestHandler.**set_server_name**(*server_name*)
生成する HTML 文書内で使用される名前を設定します。この名前は生成した文書冒頭の h1 要素内で使われます。

DocCGIXMLRPCRequestHandler.**set_server_documentation**(*server_documentation*)
生成する HTML 文書内で使用される説明を設定します。この説明は文書中のサーバ名の下にパラグラフとして出力されます。

21.28 ipaddress --- IPv4/IPv6 操作ライブラリ

ソースコード: Lib/ipaddress.py

---

ipaddress は IPv4 と IPv6 アドレスとネットワークの生成・変更・操作を提供しています。

このモジュールの関数やクラスを使うと、IP アドレスに関する様々なタスク、例えば 2 つのホストが同じサブネットに属しているかどうかをチェックしたり、特定のサブネット内の全てのホストをイテレートしたり、文字列が正しい IP アドレスかネットワークを定義しているかどうかをチェックするなどを簡単に実現することができます。

これはモジュールの完全な API リファレンスです。概要や紹介は ipaddress-howto を参照してください。

バージョン 3.3 で追加.

21.28.1 便利なファクトリ関数

ipaddress モジュールは簡単に IP アドレス、ネットワーク、インターフェースを生成するためのファクトリ関数を提供しています:

ipaddress.**ip_address**(*address*)
引数に渡された IP address に応じて、*IPv4Address* か *IPv6Address* のオブジェクトを返します。IPv4 か IPv6 のアドレスを受け取ります; 2**32 より小さい整数はデフォルトでは IPv4 アドレスだと判断されます。*address* が正しい IPv4, IPv6 アドレスを表現していない場合は *ValueError* を発生させます。

```
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.ip_address('2001:db8::')
IPv6Address('2001:db8::')
```

ipaddress.ip_network(*address*, *strict=True*)

引数に渡された IP address に応じて、*IPv4Network* か *IPv6Network* のオブジェクトを返します。*address* は IP ネットワークを示す文字列あるいは整数です。IPv4 か IPv6 のネットワークを受け取ります; 2**32 より小さい整数はデフォルトでは IPv4 アドレスだと判断されます。*strict* は *IPv4Network* か *IPv6Network* のコンストラクタに渡されます。*address* が正しい IPv4, IPv6 アドレスを表現していない場合や、ネットワークの host bit がセットされていた場合は *ValueError* を発生させます。

```
>>> ipaddress.ip_network('192.168.0.0/28')
IPv4Network('192.168.0.0/28')
```

ipaddress.ip_interface(*address*)

引数に渡された IP address に応じて、*IPv4Interface* か *IPv6Interface* のオブジェクトを返します。*address* は IP ネットワークを示す文字列あるいは整数です。IPv4 か IPv6 のネットワークを受け取ります; 2**32 より小さい整数はデフォルトでは IPv4 アドレスだと判断されます。*address* が正しい IPv4, IPv6 アドレスを表現していない場合は *ValueError* を発生させます。

これらの便利関数を利用するデメリットとして、IPv4 と IPv6 両方のフォーマットを扱う必要性があるために、どちらを期待されていたのかを知ることができず、エラーメッセージが最小限の情報しか提供できないことです。利用したいバージョンの特定のコンストラクタを直接呼ぶことで、より詳細なエラーレポートを得ることができます。

21.28.2 IP アドレス

Address オブジェクト

IPv4Address と *IPv6Address* オブジェクトは多くの共通した属性を持っています。両方の IP バージョンを扱うコードを書きやすくするために、IPv6 アドレスでしか意味が無いいくつかの属性も *IPv4Address* オブジェクトに実装されています。アドレスオブジェクトは *hashable* なので、辞書のキーとして利用できます。

class ipaddress.IPv4Address(*address*)

IPv4 アドレスを構築する。*address* が正しい IPv4 アドレスでない場合、*AddressValueError* を発生させます。

以下のものが正しい IPv4 アドレスを構築します:

1. A string in decimal-dot notation, consisting of four decimal integers in the inclusive range 0--255, separated by dots (e.g. `192.168.0.1`). Each integer represents an octet (byte) in the address. Leading zeroes are not tolerated to prevent confusion with octal notation.

2. 32bit に収まる整数。

3. 大きさ 4 の *bytes* オブジェクトに (最上位オクテットが最初になるように) パックされた整数。

```
>>> ipaddress.IPv4Address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(3232235521)
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(b'\xC0\xA8\x00\x01')
IPv4Address('192.168.0.1')
```

バージョン 3.8 で変更: Leading zeros are tolerated, even in ambiguous cases that look like octal notation.

バージョン 3.8.12 で変更: Leading zeros are no longer tolerated and are treated as an error. IPv4 address strings are now parsed as strict as glibc *inet_pton()*.

version

適切なバージョン番号: IPv4 なら 4, IPv6 なら 6.

max_prefixlen

このバージョンのアドレスを表現するのに必要なビット数: IPv4 なら 32, IPv6 なら 128.

prefix は、アドレスがネットワークに含まれるかどうかを決定するために比較する、アドレスの先頭ビット数を定義します。

compressed

exploded

ドットと 10 進数を使った表現の文字列。この表現には先頭の 0 は含まれません。

IPv4 はアドレスの 0 オクテットを省略する記法を定義していないので、IPv4 アドレスにおいてこれらの 2 つの属性は常に `str(addr)` と等しくなります。これらの属性を用意することで、IPv4 と IPv6 アドレス両方を扱う、表示用コードが書きやすくなります。

packed

このアドレスのバイナリ表現 - 適切な長さをもった *bytes* オブジェクト (最上位オクテットが先頭)。IPv4 では 4 byte で、IPv6 では 16 byte。

reverse_pointer

The name of the reverse DNS PTR record for the IP address, e.g.:

```
>>> ipaddress.ip_address("127.0.0.1").reverse_pointer
'1.0.0.127.in-addr.arpa'
>>> ipaddress.ip_address("2001:db8::1").reverse_pointer
'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa'
```

This is the name that could be used for performing a PTR lookup, not the resolved hostname itself.

バージョン 3.5 で追加.

is_multicast

アドレスがマルチキャスト用に予約されている場合は True 。**RFC 3171** (IPv4) か **RFC 2373** (IPv6) を参照。

is_private
アドレスがプライベートネットワークに割り当てられている場合に True. iana-ipv4-special-registry (for IPv4) か iana-ipv6-special-registry (for IPv6) を参照。

is_global
アドレスがパブリックネットワークに割り当てられている場合に True. iana-ipv4-special-registry (for IPv4) か iana-ipv6-special-registry (for IPv6) を参照。

バージョン 3.4 で追加.

is_unspecified
アドレスが未定義の時に True. **RFC 5735** (IPv4) か **RFC 2373** (IPv6) を参照。

is_reserved
IETF で予約されているアドレスの場合に True。

is_loopback
ループバックアドレスである場合に True. **RFC 3330** (IPv4) か **RFC 2373** (IPv6) を参照。

is_link_local
アドレスがリンクローカル用に予約されいている場合に True. **RFC 3927** を参照。

class ipaddress.IPv6Address(*address*)
IPv6 アドレスを構築する。*address* が正しい IPv6 アドレスでない場合、*AddressValueError* を発生させます。

以下のものが正しい IPv6 アドレスを構築します:

1. 4 桁の 16 進数からなるグループ 8 個で構成された文字列。各グループは 16bit を表現している。グループはコロンで区切られる。これは *exploded* (長い) 記法を表す。文字列は *compressed* (省略) 記法でも良い。詳細は **RFC 4291** を参照。例えば、"0000:0000:0000:0000:0000:0abc:0007:0def" は "::abc:7:def" と省略できる。
2. 128bit に収まる整数。
3. ビッグエンディアンで 16 バイトの長さの *bytes* オブジェクトにパックされた整数。

```
>>> ipaddress.IPv6Address('2001:db8::1000')
IPv6Address('2001:db8::1000')
```

compressed

アドレス表現の短い形式で、グループ内の先頭の 0 を省略し、連続する完全に 0 のグループの一番長いシーケンスを 1 つの空グループに折りたたんだもの。

これは IPv6 アドレスに対して str(addr) が返す値と同じです。

exploded

アドレス表現の長い形式。全てのグループの先頭の 0 は省略されず、完全に 0 のグループも省略されない。

以降の属性については、*IPv4Address* クラスの対応するドキュメントを参照してください:

packed

reverse_pointer

version

max_prefixlen

is_multicast

is_private

is_global

is_unspecified

is_reserved

is_loopback

is_link_local
バージョン 3.4 で追加: is_global

is_site_local
アドレスがサイトローカルな目的のために予約されいている場合に True. サイトローカルアドレスは **RFC 3879** によって廃止されている事に注意してください。アドレスが **RFC 4193** で定義されているユニークローカルアドレスの範囲に含まれているかどうかをテストするには、*is_private* を利用してください。

ipv4_mapped
IPv4 にマップされた (::FFFF/96 で始まる) アドレスの場合、このプロパティは埋め込まれた IPv4 Address を返します。それ以外のアドレスに対しては、このプロパティは None になります。

sixtofour
RFC 3056 で定義された 6to4 (2002::/16 で始まる) アドレスの場合、このプロパティは埋め込まれた IPv4 Address を返します。それ以外のアドレスに対しては、このプロパティは None になります。

teredo
RFC 4380 で定義された Teredo (2001::/32 で始まる) アドレスの場合、このプロパティは埋め込まれた (server, client) IP アドレスペアを返します。それ以外のアドレスに対しては、このプロパティは None になります。

文字列と整数への変換

socket モジュールなどのネットワークインターフェースを利用するには、アドレスを文字列や整数に変換しなければなりません。これには組み込みの *str()* と *int()* 関数を利用します:

```
>>> str(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> int(ipaddress.IPv4Address('192.168.0.1'))
3232235521
>>> str(ipaddress.IPv6Address('::1'))
'::1'
>>> int(ipaddress.IPv6Address('::1'))
1
```

演算子

Address オブジェクトはいくつかの演算子をサポートします。明記されない限り、演算子は互換性のあるオブジェクト間 (つまり IPv4 同士や IPv6 同士) でのみ利用できます。

比較演算子

Address オブジェクトは通常の比較演算子を使って比較することができます。いくつかの例:

```
>>> IPv4Address('127.0.0.2') > IPv4Address('127.0.0.1')
True
>>> IPv4Address('127.0.0.2') == IPv4Address('127.0.0.1')
False
>>> IPv4Address('127.0.0.2') != IPv4Address('127.0.0.1')
True
```

算術演算

アドレスオブジェクトから整数を加減算できます。いくつかの例:

```
>>> IPv4Address('127.0.0.2') + 3
IPv4Address('127.0.0.5')
>>> IPv4Address('127.0.0.2') - 3
IPv4Address('126.255.255.255')
>>> IPv4Address('255.255.255.255') + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ipaddress.AddressValueError: 4294967296 (>= 2**32) is not permitted as an IPv4 address
```

21.28.3 IP ネットワーク定義

IPv4Network と *IPv6Network* オブジェクトは IP ネットワークの定義とインスペクトのための機構を提供します。ネットワーク定義は *mask* と **ネットワークアドレス** からなり、*mask* でマスク (bit ごとの AND) するとネットワークアドレスと同じになる IP アドレスの範囲を定義します。例えば、`255.255.255.0` という *mask* と `192.168.1.0` というネットワークアドレスからなるネットワーク定義は、`192.168.1.0` から `192.168.1.255` を含む範囲を表します。

プリフィックス, ネットマスク、ホストマスク

IP ネットワークマスクを定義する幾つかの等価な方法があります。**プリフィックス** `/<nbits>` は先頭の何 bit がネットワークマスクで立っているかを示します。**ネットマスク** は先頭の幾つかの bit が立っている IP アドレスです。プリフィックス `/24` は IPv4 ではネットマスク `255.255.255.0` と、IPv6 では `ffff:ff00::` と同じになります。加えて、**ネットマスク** と論理が逆の **ホストマスク** があり、ときどき (例えば Cisco のアクセスコントロールリスト) ネットワークマスクを表すために利用されます。`/24` と等しい IPv4 のホストマスクは `0.0.0.255` になります。

Network オブジェクト

address オブジェクトで実装されていた属性は全て network オブジェクトにも実装されています。network はそれに追加で幾つかの属性を実装しています。全ての追加属性は *IPv4Network* と *IPv6Network* で共通なので、重複を避けるために *IPv4Network* にだけドキュメントされています。ネットワークオブジェクトは *hashable* なので、辞書のキーとして使用できます。

class ipaddress.**IPv4Network**(*address*, *strict=True*)

IPv4 ネットワーク定義を構築します。*address* は以下の 1 つです:

1. IP アドレスと、オプションでスラッシュ (`/`) で区切られたマスクを持つ文字列。IP アドレスはネットワークアドレスで、マスクは **プリフィックス** を意味する 1 つの数値か、IPv4 アドレスの文字列表現です。マスクが IPv4 アドレスのとき、非ゼロのフィールドで始まるときは **ネットマスク** として、ゼロのフィールドで始まるときは **ホストマスク** として解釈されます。ただし、すべてのフィールドが 0 の場合は、*ネットマスク*として扱われます。マスクが省略された場合、`/32` が指定されたものとします。

   例えば、次の *address* 指定は全て等しくなります: `192.168.1.0/24`, `192.168.1.0/255.255.255.0` `192.168.1.0/0.0.0.255`.

2. 32bit に収まる整数。これは 1 つのアドレスのネットワークと等しく、ネットワークアドレスが *address* に、マスクが `/32` になります。

3. 4byte の *bytes* オブジェクトにビッグエンディアンでパックされた整数。これは整数の *address* と同じように解釈されます。

4. アドレス記述とネットマスクの 2 要素のタプル。アドレス記述は、文字列、32 ビットの整数、4 バイトのパックされた整数、既存の IPv4Address オブジェクトのいずれかです。ネットマスクは、プレフィックス長を表す整数（例えば “ 24“）またはプレフィックスマスクを表す文字列（例えば “ 255.255.255.0“）です。

address が有効な IPv4 アドレスでない場合に *AddressValueError* 例外を発生させます。マスクが IPv4 アドレスに対して有効でない場合に *NetmaskValueError* 例外を発生させます。

strict が `True` の場合、与えられたアドレスのホストビットが立っていたら *ValueError* を発生させます。そうでない場合、ホストビットをマスクして正しいネットワークアドレスを計算します。

特に明記されない場合、他の network や address を受け取る network のメソッドは、引数の IP バージョンが `self` と異なる場合に *TypeError* を発生させます。

バージョン 3.5 で変更: *address* コンストラクタ引数に 2 要素のタプル形式を追加しました

version

max_prefixlen

IPv4Address の対応する属性のドキュメントを参照してください。

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

These attributes are true for the network as a whole if they are true for both the network address and the broadcast address.

network_address

この network のネットワークアドレス。ネットワークアドレスとプリフィックス長によってユニークにネットワークが定義されます。

broadcast_address

このネットワークのブロードキャストアドレス。ブロードキャストアドレスに投げられたパケットはそのネットワーク内の全てのホストに受信されます。

hostmask

IPv4Address オブジェクトとして表現された ホストマスク。

netmask

IPv4Address オブジェクトとして表現された ネットマスク。

with_prefixlen

compressed

exploded

A string representation of the network, with the mask in prefix notation.

`with_prefixlen` and `compressed` are always the same as `str(network)`. `exploded` uses the exploded form the network address.

with_netmask

A string representation of the network, with the mask in net mask notation.

with_hostmask

A string representation of the network, with the mask in host mask notation.

num_addresses

ネットワーク内のアドレスの総数

prefixlen

ネットワークプレフィックスのビット長。

hosts()

Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the network address itself and the network broadcast address. For networks with a mask length of 31, the network address and network broadcast address are also included in the result. Networks with a mask of 32 will return a list containing the single host address.

```
>>> list(ip_network('192.0.2.0/29').hosts())
[IPv4Address('192.0.2.1'), IPv4Address('192.0.2.2'),
 IPv4Address('192.0.2.3'), IPv4Address('192.0.2.4'),
 IPv4Address('192.0.2.5'), IPv4Address('192.0.2.6')]
>>> list(ip_network('192.0.2.0/31').hosts())
[IPv4Address('192.0.2.0'), IPv4Address('192.0.2.1')]
>>> list(ip_network('192.0.2.1/32').hosts())
[IPv4Address('192.0.2.1')]
```

overlaps(*other*)

`True` if this network is partly or wholly contained in *other* or *other* is wholly contained in this network.

address_exclude(*network*)

Computes the network definitions resulting from removing the given *network* from this one. Returns an iterator of network objects. Raises *ValueError* if *network* is not completely contained in this network.

```
>>> n1 = ip_network('192.0.2.0/28')
>>> n2 = ip_network('192.0.2.1/32')
>>> list(n1.address_exclude(n2))
[IPv4Network('192.0.2.8/29'), IPv4Network('192.0.2.4/30'),
 IPv4Network('192.0.2.2/31'), IPv4Network('192.0.2.0/32')]
```

subnets(*prefixlen_diff=1*, *new_prefix=None*)

The subnets that join to make the current network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be increased by. *new_prefix* is the desired new prefix of the subnets; it must be larger than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns an iterator of network objects.

```
>>> list(ip_network('192.0.2.0/24').subnets())
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
>>> list(ip_network('192.0.2.0/24').subnets(prefixlen_diff=2))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=26))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=23))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise ValueError('new prefix must be longer')
ValueError: new prefix must be longer
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=25))
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
```

supernet(*prefixlen_diff=1*, *new_prefix=None*)

The supernet containing this network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be decreased by. *new_prefix* is the desired new prefix of the supernet; it must be smaller than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns a single network object.

```
>>> ip_network('192.0.2.0/24').supernet()
IPv4Network('192.0.2.0/23')
>>> ip_network('192.0.2.0/24').supernet(prefixlen_diff=2)
IPv4Network('192.0.0.0/22')
>>> ip_network('192.0.2.0/24').supernet(new_prefix=20)
IPv4Network('192.0.0.0/20')
```

subnet_of(*other*)

このネットワークが *other* のサブネットの場合に `True` を返します。

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> b.subnet_of(a)
True
```

バージョン 3.7 で追加.

supernet_of(*other*)

このネットワークが *other* のスーパーネットの場合に `True` を返します。

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> a.supernet_of(b)
True
```

バージョン 3.7 で追加.

compare_networks(*other*)

このネットワークを *other* と比較します。比較ではネットワークアドレスのみが考慮され、ホストアドレスは考慮されません。-1、0、1 のいずれかを返します。

```
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.2/32'))
-1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.0/32'))
1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.1/32'))
0
```

バージョン 3.7 で非推奨: It uses the same ordering and comparison algorithm as "<", "==", and ">"

class ipaddress.IPv6Network(*address*, *strict=True*)

IPv6 ネットワーク定義を構築します。*address* は以下の 1 つです:

1. A string consisting of an IP address and an optional prefix length, separated by a slash (/). The IP address is the network address, and the prefix length must be a single number, the *prefix*. If no prefix length is provided, it's considered to be /128.

   Note that currently expanded netmasks are not supported. That means 2001:db00::0/24 is a valid argument while 2001:db00::0/ffff:ff00:: not.

2. 128bit に収まる整数。これは 1 つのアドレスのネットワークと等しく、ネットワークアドレスが *address* に、マスクが /128 になります。

3. 16byte の *bytes* オブジェクトにビッグエンディアンでパックされた整数。これは整数の *address* と同じように解釈されます。

4. アドレス記述とネットマスクの 2 要素のタプル。アドレス記述は、文字列、128 ビットの整数、16 バイトのパックされた整数、既存の IPv6Address オブジェクトのいずれかです。ネットマスクは、プレフィックス長を表す整数です。

address が有効な IPv6 アドレスでない場合に *AddressValueError* 例外を発生させます。マスクが IPv6 アドレスに対して有効でない場合に *NetmaskValueError* 例外を発生させます。

strict が True の場合、与えられたアドレスのホストビットが立っていたら *ValueError* を発生させます。そうでない場合、ホストビットをマスクして正しいネットワークアドレスを計算します。

バージョン 3.5 で変更: *address* コンストラクタ引数に 2 要素のタプル形式を追加しました

version

max_prefixlen

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

network_address

broadcast_address

hostmask

netmask

with_prefixlen

compressed

exploded

with_netmask

with_hostmask

num_addresses

prefixlen

hosts()

Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the Subnet-Router anycast address. For networks with a mask length of 127, the Subnet-Router anycast address is also included in the result. Networks with a mask of 128 will return a list containing the single host address.

overlaps(*other***)**

address_exclude(*network***)**

subnets(*prefixlen_diff=1, new_prefix=None***)**

supernet(*prefixlen_diff=1, new_prefix=None***)**

subnet_of(*other***)**

supernet_of(*other***)**

compare_networks(*other***)**

IPv4Network の対応する属性のドキュメントを参照してください。

is_site_local

These attribute is true for the network as a whole if it is true for both the network address and the broadcast address.

演算子

Network オブジェクトはいくつかの演算子をサポートします。明記されない限り、演算子は互換性のあるオブジェクト間 (つまり IPv4 同士や IPv6 同士) でのみ利用できます。

論理演算子

Network objects can be compared with the usual set of logical operators. Network objects are ordered first by network address, then by net mask.

イテレーション

Network objects can be iterated to list all the addresses belonging to the network. For iteration, *all* hosts are returned, including unusable hosts (for usable hosts, use the *`hosts()`* method). An example:

```
>>> for addr in IPv4Network('192.0.2.0/28'):
...     addr
...
IPv4Address('192.0.2.0')
IPv4Address('192.0.2.1')
IPv4Address('192.0.2.2')
IPv4Address('192.0.2.3')
IPv4Address('192.0.2.4')
IPv4Address('192.0.2.5')
IPv4Address('192.0.2.6')
IPv4Address('192.0.2.7')
IPv4Address('192.0.2.8')
IPv4Address('192.0.2.9')
IPv4Address('192.0.2.10')
IPv4Address('192.0.2.11')
IPv4Address('192.0.2.12')
IPv4Address('192.0.2.13')
IPv4Address('192.0.2.14')
IPv4Address('192.0.2.15')
```

アドレスのコンテナとしてのネットワーク

ネットワークオブジェクトは、アドレスのコンテナとして振舞えます。いくつか例をあげます:

```
>>> IPv4Network('192.0.2.0/28')[0]
IPv4Address('192.0.2.0')
>>> IPv4Network('192.0.2.0/28')[15]
IPv4Address('192.0.2.15')
>>> IPv4Address('192.0.2.6') in IPv4Network('192.0.2.0/28')
True
>>> IPv4Address('192.0.3.6') in IPv4Network('192.0.2.0/28')
False
```

21.28.4 インターフェイスオブジェクト

インタフェースオブジェクトは *hashable* なので、辞書のキーとして使用できます。

class ipaddress.IPv4Interface(*address*)

Construct an IPv4 interface. The meaning of *address* is as in the constructor of *IPv4Network*, except that arbitrary host addresses are always accepted.

IPv4Interface is a subclass of *IPv4Address*, so it inherits all the attributes from that class. In addition, the following attributes are available:

ip

The address (*IPv4Address*) without network information.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.ip
IPv4Address('192.0.2.5')
```

network

The network (*IPv4Network*) this interface belongs to.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.network
IPv4Network('192.0.2.0/24')
```

with_prefixlen

A string representation of the interface with the mask in prefix notation.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_prefixlen
'192.0.2.5/24'
```

with_netmask

A string representation of the interface with the network as a net mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_netmask
'192.0.2.5/255.255.255.0'
```

with_hostmask

A string representation of the interface with the network as a host mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_hostmask
'192.0.2.5/0.0.0.255'
```

class ipaddress.IPv6Interface(*address*)

Construct an IPv6 interface. The meaning of *address* is as in the constructor of *IPv6Network*, except that arbitrary host addresses are always accepted.

IPv6Interface is a subclass of *IPv6Address*, so it inherits all the attributes from that class. In addition, the following attributes are available:

ip

network

with_prefixlen

with_netmask

with_hostmask

IPv4Interface の対応する属性のドキュメントを参照してください。

演算子

Interface オブジェクトはいくつかの演算子をサポートします。明記されない限り、演算子は互換性のあるオブジェクト間 (つまり IPv4 同士や IPv6 同士) でのみ利用できます。

論理演算子

Interface objects can be compared with the usual set of logical operators.

For equality comparison (`==` and `!=`), both the IP address and network must be the same for the objects to be equal. An interface will not compare equal to any address or network object.

For ordering (`<`, `>`, etc) the rules are different. Interface and address objects with the same IP version can be compared, and the address objects will always sort before the interface objects. Two interface objects are first compared by their networks and, if those are the same, then by their IP addresses.

21.28.5 その他のモジュールレベル関数

このモジュールは以下のモジュールレベル関数も提供しています:

ipaddress.v4_int_to_packed(*address*)

アドレスをネットワークバイトオーダー (ビッグエンディアン) でパックされた 4 バイトで表現します。*address* は IPv4 IP アドレスを整数で表したものです。整数が負だったり IPv4 IP アドレスとして大きすぎる場合は *ValueError* 例外を発生させます。

```
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>> ipaddress.v4_int_to_packed(3221225985)
b'\xc0\x00\x02\x01'
```

ipaddress.v6_int_to_packed(*address*)

アドレスをネットワークバイトオーダー (ビッグエンディアン) でパックされた 16 バイトで表現します。*address* は IPv6 IP アドレスを整数で表したものです。整数が負だったり IPv6 IP アドレスとして大きすぎる場合は *ValueError* 例外を発生させます。

ipaddress.**summarize_address_range**(*first*, *last*)

first と last で指定された IP アドレス帯に対するイテレーターを返します。*first* はアドレス帯の中の最初の *IPv4Address* か *IPv6Address* で、*last* はアドレス帯の中の最後の *IPv4Address* か *IPv6Address* です。*first* か *last* が IP アドレスでない場合や、2 つの型が揃っていない場合に、*TypeError* を発生させます。*last* が *first* より大きくない場合や、*first* アドレスのバージョンが 4 でも 6 でもない場合は *ValueError* を発生させます。

```
>>> [ipaddr for ipaddr in ipaddress.summarize_address_range(
...     ipaddress.IPv4Address('192.0.2.0'),
...     ipaddress.IPv4Address('192.0.2.130'))]
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/31'), IPv4Network('192.0.2.130/32
↪')]
```

ipaddress.**collapse_addresses**(*addresses*)

Return an iterator of the collapsed *IPv4Network* or *IPv6Network* objects. *addresses* is an iterator of *IPv4Network* or *IPv6Network* objects. A *TypeError* is raised if *addresses* contains mixed version objects.

```
>>> [ipaddr for ipaddr in
... ipaddress.collapse_addresses([ipaddress.IPv4Network('192.0.2.0/25'),
... ipaddress.IPv4Network('192.0.2.128/25')])]
[IPv4Network('192.0.2.0/24')]
```

ipaddress.**get_mixed_type_key**(*obj*)

ネットワークとアドレスをソートするための key 関数を返します。アドレスとネットワークは本質的に違うものなので、デフォルトでは比較できません。そのため、次の式は:

```
IPv4Address('192.0.2.0') <= IPv4Network('192.0.2.0/24')
```

理にかなっていません。しかし、それでも *ipaddress* にそれらをソートしてほしい場合があるかもしれません。その場合は、この関数を *sorted()* の *key* 引数に使うことができます。

obj はネットワークオブジェクトかアドレスオブジェクトのどちらかです。

21.28.6 Custom Exceptions

クラスのコンストラクタがより具体的なエラー報告をするために、このモジュールでは以下の例外を定義します:

exception ipaddress.**AddressValueError**(*ValueError*)

Any value error related to the address.

exception ipaddress.**NetmaskValueError**(*ValueError*)

Any value error related to the net mask.

第

TWENTYTWO

マルチメディアサービス

この章で記述されているモジュールは、主にマルチメディアアプリケーションに役立つさまざまなアルゴリズムまたはインターフェイスを実装しています。これらのモジュールはインストール時の自由裁量に応じて利用できます:

22.1 audioop --- 生の音声データを操作する

audioop モジュールは音声データの便利な操作を含んでいます。このモジュールは、*bytes-like* **オブジェクト**に保存された、符号付き整数の、ビット幅が 8, 16, 24, あるいは 32 ビットの音声データを対象として操作します。特に指定されていない限り、すべての波形データ（スカラー要素）は整数です。

バージョン 3.4 で変更: 24 bit サンプルのサポートが追加されました。すべての関数はどんな *bytes-like object* でも使用できます。文字列の入力は即座にエラーになります。

このモジュールは a-LAW、u-LAW そして Intel/DVI ADPCM エンコードをサポートしています。

複雑な操作のうちいくつかはサンプル幅が 16 ビットのデータに対してのみ働きますが、それ以外は常にサンプル幅を操作のパラメタとして (バイト単位で) 渡します。

このモジュールでは以下の変数と関数を定義しています:

exception audioop.error

この例外は、未知のサンプル当たりのバイト数を指定した時など、全般的なエラーに対して送出されます。

audioop.add(*fragment1*, *fragment2*, *width*)

パラメータとして渡された 2 つのサンプルの和のデータを返します。*width* はバイト単位のサンプル幅で、1, 2, 3, 4 のいずれかです。両方のデータは同じ長さでなければなりません。オーバーフローした場合は、切り捨てされます。

audioop.adpcm2lin(*adpcmfragment*, *width*, *state*)

Intel/DVI ADPCM 形式のデータをリニア (linear) 形式にデコードします。ADPCM 符号化方式の詳細については *lin2adpcm()* の説明を参照して下さい。(sample, newstate) からなるタプルを返し、サンプルは *width* に指定した幅になります。

audioop.**alaw2lin**(*fragment*, *width*)

a-LAW 形式のデータをリニア (linear) 形式に変換します。a-LAW 形式は常に 8 ビットのサンプルを使用するので、ここでは *width* は単に出力データのサンプル幅となります。

audioop.**avg**(*fragment*, *width*)

データ中の全サンプルの平均値を返します。

audioop.**avgpp**(*fragment*, *width*)

データ中の全サンプルの平均 peak-peak 振幅を返します。フィルタリングを行っていない場合、このルーチンの有用性は疑問です。

audioop.**bias**(*fragment*, *width*, *bias*)

元の音声データの各サンプルにバイアスを加算した音声データを返します。オーバーフローした場合はラップアラウンドされます。

audioop.**byteswap**(*fragment*, *width*)

fragment のすべてのサンプルを "byteswap" して、修正された fragment を返します。ビッグエンディアンのサンプルをリトルエンディアンに、またはその逆に変換します。

バージョン 3.4 で追加.

audioop.**cross**(*fragment*, *width*)

引数に渡したデータ中のゼロ交差回数を返します。

audioop.**findfactor**(*fragment*, *reference*)

`rms(add(fragment, mul(reference, -F)))` を最小にするような係数 *F*、すなわち、*reference* に乗算したときにもっとも *fragment* に近くなるような値を返します。*fragment* と *reference* のサンプル幅はいずれも 2 バイトでなければなりません。

このルーチンの実行に要する時間は `len(fragment)` に比例します。

audioop.**findfit**(*fragment*, *reference*)

reference を可能な限り *fragment* に一致させようとします (*fragment* は *reference* より長くなければなりません)。この処理は (概念的には) *fragment* からスライスをいくつか取り出し、それぞれについて *findfactor()* を使って最良な一致を計算し、誤差が最小の結果を選ぶことで実現します。*fragment* と *reference* のサンプル幅は両方とも 2 バイトでなければなりません。`(offset, factor)` からなるタプルを返します。*offset* は最適な一致箇所が始まる *fragment* のオフセット値（整数）で、*factor* は *findfactor()* の返す係数 (浮動小数点数) です。

audioop.**findmax**(*fragment*, *length*)

fragment から、長さが *length* サンプル (バイトではありません!) で最大のエネルギーを持つスライス、すなわち、`rms(fragment[i*2:(i+length)* 2])` を最大にするようなスライスを探し、*i* を返します。データのはサンプル幅は 2 バイトでなければなりません。

このルーチンの実行に要する時間は `len(fragment)` に比例します。

audioop.**getsample**(*fragment*, *width*, *index*)

データ中の *index* サンプル目の値を返します。

audioop.**lin2adpcm**(*fragment*, *width*, *state*)

データを 4 ビットの Intel/DVI ADPCM 符号化方式に変換します。ADPCM 符号化方式とは適応符号化方式の一つで、あるサンプルと (可変の) ステップだけ離れたその次のサンプルとの差を 4 ビットの整数で表現する方式です。Intel/DVI ADPCM アルゴリズムは IMA (国際 MIDI 協会) に採用されているので、おそらく標準になるはずです。

state はエンコーダの内部状態が入ったタプルです。エンコーダは `(adpcmfrag, newstate)` のタプルを返し、次に *lin2adpcm()* を呼び出す時に *newstate* を渡さねばなりません。最初に呼び出す時には *state* に `None` を渡してもかまいません。*adpcmfrag* は ADPCM で符号化されたデータで、バイト当たり 2 つの 4 ビット値がパックされています。

audioop.**lin2alaw**(*fragment*, *width*)

音声データのサンプルを a-LAW エンコーディングに変換し、バイトオブジェクトとして返します。a-LAW とは 13 ビットのダイナミックレンジを 8bit だけで表現できる音声エンコーディングです。Sun の音声ハードウェアなどで使われています。

audioop.**lin2lin**(*fragment*, *width*, *newwidth*)

サンプル幅を 1、2、3、4 バイト形式の間で変換します。

注釈: .WAV ファイルのような幾つかのオーディオフォーマットでは、16、24 と 32 bit のサンプルは符号付きですが、8 bit のサンプルは符号なしです。そのため、そのようなフォーマットで 8 bit に変換する場合は、変換結果に 128 を足さなければなりません:

```
new_frames = audioop.lin2lin(frames, old_width, 1)
new_frames = audioop.bias(new_frames, 1, 128)
```

逆に、8 bit から 16、24、32 bit に変換する場合も、同じことが言えます。

---

audioop.**lin2ulaw**(*fragment*, *width*)

音声データのサンプルを u-LAW エンコーディングに変換し、バイトオブジェクトとして返します。u-LAW とは 14 ビットのダイナミックレンジを 8bit だけで表現できる音声エンコーディングです。Sun の音声ハードウェアなどで使われています。

audioop.**max**(*fragment*, *width*)

音声データ全サンプルの **絶対値** の最大値を返します。

audioop.**maxpp**(*fragment*, *width*)

音声データの最大 peak-peak 振幅を返します。

audioop.**minmax**(*fragment*, *width*)

音声データ全サンプル中における最小値と最大値からなるタプルを返します。

audioop.**mul**(*fragment*, *width*, *factor*)

元の音声データの各サンプルに浮動小数点数 *factor* を乗算した音声データを返します。オーバーフローした場合は切り捨てられます。

audioop.**ratecv**(*fragment*, *width*, *nchannels*, *inrate*, *outrate*, *state*[, *weightA*[, *weightB*]])

入力したデータのフレームレートを変換します。

state は変換ルーチンの内部状態を入れたタプルです。変換ルーチンは (newfragment, newstate) を返し、次に *ratecv()* を呼び出す時には *newstate* を渡さなねばなりません。最初の呼び出しでは None を渡します。

引数 *weightA* と *weightB* は単純なデジタルフィルタのパラメタで、デフォルト値はそれぞれ 1 と 0 です。

audioop.**reverse**(*fragment*, *width*)

データ内のサンプルの順序を逆転し、変更されたデータを返します。

audioop.**rms**(*fragment*, *width*)

データの自乗平均根 (root-mean-square)、すなわち sqrt(sum(S_i^2)/n) を返します。

これはオーディオ信号の強度 (power) を測る一つの目安です。

audioop.**tomono**(*fragment*, *width*, *lfactor*, *rfactor*)

ステレオ音声データをモノラル音声データに変換します。左チャネルのデータに *lfactor*、右チャネルのデータに *rfactor* を掛けた後、二つのチャネルの値を加算して単一チャネルの信号を生成します。

audioop.**tostereo**(*fragment*, *width*, *lfactor*, *rfactor*)

モノラル音声データをステレオ音声データに変換します。ステレオ音声データの各サンプル対は、モノラル音声データの各サンプルをそれぞれ左チャネルは *lfactor* 倍、右チャネルは *rfactor* 倍して生成します。

audioop.**ulaw2lin**(*fragment*, *width*)

u-LAW で符号化されている音声データを線形に符号化された音声データに変換します。u-LAW 符号化は常にサンプル当たり 8 ビットを使うため、*width* は出力音声データのサンプル幅にしか使われません。

mul() や *max()* といった操作はモノラルとステレオを区別しない、すなわち全てのデータを平等に扱うということに注意してください。この仕様が問題になるようなら、あらかじめステレオ音声データを二つのモノラル音声データに分割しておき、操作後に再度統合してください。そのような例を以下に示します:

```
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
    rsample = audioop.tomono(sample, width, 0, 1)
    lsample = audioop.mul(lsample, width, lfactor)
    rsample = audioop.mul(rsample, width, rfactor)
    lsample = audioop.tostereo(lsample, width, 1, 0)
    rsample = audioop.tostereo(rsample, width, 0, 1)
    return audioop.add(lsample, rsample, width)
```

もし ADPCM 符号をネットワークパケットの構築に使って独自のプロトコルをステートレスにしたい場合 (つまり、パケットロスを許容したい場合) は、データだけを送信して、ステートを送信すべきではありません。デコーダに従って *initial* ステート (*lin2adpcm()* に渡される値) を送るべきで、最終状態 (符号化器が返す値) を送るべきではないことに注意してください。もし、*struct.Struct* をバイナリでの状態保存に使いたい場合は、最初の要素 (予測値) を 16bit で符号化し、2 番目の要素 (デルタインデックス) を 8bit で符号化できます。

このモジュールの ADPCM 符号のテストは自分自身に対してのみ行っており、他の ADPCM 符号との間で

は行っていません。作者が仕様を誤解している部分もあるかもしれず、それぞれの標準との間で相互運用できない場合もあり得ます。

`find*()` ルーチンは一見滑稽に見えるかもしれません。これらの関数の主な目的はエコー除去 (echo cancellation) にあります。エコー除去を十分高速に行うには、出力サンプル中から最も大きなエネルギーを持った部分を取り出し、この部分が入力サンプル中のどこにあるかを調べ、入力サンプルから出力サンプル自体を減算します:

```
def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 800)    # one tenth second
    out_test = outputdata[pos*2:]
    in_test = inputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test)
    # Optional (for better cancellation):
    # factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)],
    #              out_test)
    prefill = '\0'*(pos+ipos)*2
    postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
    outputdata = prefill + audioop.mul(outputdata, 2, -factor) + postfill
    return audioop.add(inputdata, outputdata, 2)
```

22.2 aifc --- AIFF および AIFC ファイルの読み書き

ソースコード: Lib/aifc.py

このモジュールは AIFF と AIFF-C ファイルの読み書きをサポートします。AIFF（Audio Interchange File Format）はデジタルオーディオサンプルをファイルに保存するためのフォーマットです。AIFF-C は AIFF の新しいバージョンで、オーディオデータの圧縮に対応しています。

オーディオファイルには、オーディオデータについて記述したパラメータがたくさん含まれています。サンプリングレートあるいはフレームレートは、1 秒あたりのオーディオサンプル数です。チャンネル数は、モノラル、ステレオ、4 チャンネルかどうかを示します。フレームはそれぞれ、チャンネルごとに一つのサンプルからなります。サンプルサイズは、一つのサンプルの大きさをバイト数で示したものです。したがって、一つのフレームは `nchannels * samplesize` バイト からなり、1 秒間では `nchannels * samplesize * framerate` バイトで構成されます。

例えば、CD 品質のオーディオは 2 バイト（16 ビット）のサンプルサイズを持っていて、2 チャンネル（ステレオ）であり、44,100 フレーム／秒のフレームレートを持っています。そのため、フレームサイズは 4 バイト（2*2）で、1 秒間では 2*2*44100 バイト（176,400 バイト）になります。

aifc モジュールは以下の関数を定義しています:

aifc.open(*file*, *mode=None*)

AIFF あるいは AIFF-C ファイルを開き、後述するメソッドを持つインスタンスを返します。引数 *file* はファイルを示す文字列か、*file object* のいずれかです。*mode* は、読み込み用に開くときには `'r'` か `'rb'` のどちらかでなければならず、書き込み用に開くときには `'w'` か `'wb'` のどちらかでなければなりません。もし省略されたら、`file.mode` が存在すればそれが使用され、なければ `'rb'` が使われま

す。書き込み用にこのメソッドを使用するときには、これから全部でどれだけのサンプル数を書き込むのか分からなかったり、writeframesraw() と setnframes() を使わないなら、ファイルオブジェクトはシーク可能でなければなりません。*open()* 関数は with 文の中で使われるかもしれません。with ブロックの実行が終了したら、*close()* メソッドが呼び出されます。

バージョン 3.4 で変更: with 構文のサポートが追加されました。

ファイルが *open()* によって読み込み用に開かれたときに返されるオブジェクトには、以下のメソッドがあります:

aifc.getnchannels()
オーディオチャンネル数（モノラルなら 1、ステレオなら 2）を返します。

aifc.getsampwidth()
サンプルサイズをバイト数で返します。

aifc.getframerate()
サンプリングレート（1 秒あたりのオーディオフレーム数）を返します。

aifc.getnframes()
ファイルの中のオーディオフレーム数を返します。

aifc.getcomptype()
オーディオファイルで使用されている圧縮形式を示す 4 バイトの bytes を返します。AIFF ファイルでは b'NONE' が返されます。

aifc.getcompname()
オーディオファイルの圧縮形式を人に判読可能な形に変換できる bytes で返します。AIFF ファイルでは b'not compressed' が返されます。

aifc.getparams()
get*() メソッドが返すのと同じ（nchannels, sampwidth, framerate, nframes, comptype, compname）の *namedtuple()* を返します。

aifc.getmarkers()
オーディオファイルのマーカーのリストを返します。一つのマーカーは三つの要素のタプルです。要素の 1 番目はマーク ID（整数）、2 番目はマーク位置のフレーム数をデータの始めから数えた値（整数）、3 番目はマークの名称（文字列）です。

aifc.getmark(*id*)
与えられた *id* のマークの要素を *getmarkers()* で述べたタプルで返します。

aifc.readframes(*nframes*)
オーディオファイルの次の *nframes* 個のフレームを読み込んで返します。返されるデータは、全チャンネルの圧縮されていないサンプルをフレームごとに文字列にしたものです。

aifc.rewind()
読み込むポインタをデータの始めに巻き戻します。次に *readframes()* を使用すると、データの始めから読み込みます。

aifc.**setpos**(*pos*)
　指定したフレーム数の位置にポインタを設定します。

aifc.**tell**()
　現在のポインタのフレーム位置を返します。

aifc.**close**()
　AIFF ファイルを閉じます。このメソッドを呼び出したあとでは、オブジェクトはもう使用できません。

ファイルが *open()* によって書き込み用に開かれたときに返されるオブジェクトには、readframes() と setpos() を除く上述の全てのメソッドがあります。さらに以下のメソッドが定義されています。get*() メソッドは、対応する set*() を呼び出したあとでのみ呼び出し可能です。最初に writeframes() あるいは writeframesraw() を呼び出す前に、フレーム数を除く全てのパラメータが設定されていなければなりません。

aifc.**aiff**()
　AIFF ファイルを作ります。デフォルトでは AIFF-C ファイルが作られますが、ファイル名が '.aiff' で終わっていれば AIFF ファイルが作られます。

aifc.**aifc**()
　AIFF-C ファイルを作ります。デフォルトでは AIFF-C ファイルが作られますが、ファイル名が '.aiff' で終わっていれば AIFF ファイルが作られます。

aifc.**setnchannels**(*nchannels*)
　オーディオファイルのチャンネル数を設定します。

aifc.**setsampwidth**(*width*)
　オーディオのサンプルサイズをバイト数で設定します。

aifc.**setframerate**(*rate*)
　サンプリングレートを 1 秒あたりのフレーム数で設定します。

aifc.**setnframes**(*nframes*)
　オーディオファイルに書き込まれるフレーム数を設定します。もしこのパラメータが設定されていなかったり正しくなかったら、ファイルはシークに対応していなければなりません。

aifc.**setcomptype**(*type*, *name*)
　圧縮形式を設定します。もし設定しなければ、オーディオデータは圧縮されません。AIFF ファイルは圧縮できません。name 引数は人間が読める圧縮形式の説明を bytes にしたもので、type 引数は 4 バイトの bytes でなければなりません。現在のところ、以下の圧縮形式がサポートされています：b'NONE', b'ULAW', b'ALAW', b'G722'。

aifc.**setparams**(*nchannels*, *sampwidth*, *framerate*, *comptype*, *compname*)
　上の全パラメータを一度に設定します。引数はそれぞれのパラメータからなるタプルです。つまり、*setparams()* の引数として、*getparams()* を呼び出した結果を使うことができます。

aifc.**setmark**(*id*, *pos*, *name*)
　指定した ID（1 以上）、位置、名称でマークを加えます。このメソッドは、*close()* の前ならいつでも呼び出すことができます。

aifc.tell()
出力ファイルの現在の書き込み位置を返します。*setmark()* との組み合わせで使うと便利です。

aifc.writeframes(*data*)
出力ファイルにデータを書き込みます。このメソッドは、オーディオファイルのパラメータを設定したあとでのみ呼び出し可能です。

バージョン 3.4 で変更: どのような *bytes-like object* も使用できるようになりました。

aifc.writeframesraw(*data*)
オーディオファイルのヘッダ情報が更新されないことを除いて、*writeframes()* と同じです。

バージョン 3.4 で変更: どのような *bytes-like object* も使用できるようになりました。

aifc.close()
AIFF ファイルを閉じます。ファイルのヘッダ情報は、オーディオデータの実際のサイズを反映して更新されます。このメソッドを呼び出したあとでは、オブジェクトはもう使用できません。

22.3 sunau --- Sun AU ファイルの読み書き

ソースコード: Lib/sunau.py

---

sunau モジュールは、Sun AU サウンドフォーマットへの便利なインターフェースを提供します。このモジュールは、*aifc* モジュールや *wave* モジュールと互換性のあるインターフェースを備えています。

オーディオファイルはヘッダとそれに続くデータから構成されます。ヘッダのフィールドは以下の通りです:

フィールド	内容
magic word	4 バイト文字列 `.snd`。
header size	info を含むヘッダのサイズをバイト数で示したもの。
data size	データの物理サイズをバイト数で示したもの。
encoding	オーディオサンプルのエンコード形式。
sample rate	サンプリングレート。
# of channels	サンプルのチャンネル数。
info	オーディオファイルについての説明を ASCII 文字列で示したもの（null バイトで埋められます）。

info フィールド以外の全てのヘッダフィールドは 4 バイトの大きさです。ヘッダフィールドは big-endian でエンコードされた、計 32 ビットの符合なし整数です。

sunau モジュールは以下の関数を定義しています:

sunau.open(*file*, *mode*)
file が文字列ならその名前のファイルを開き、そうでないならファイルのようにシーク可能なオブジェクトとして扱います。*mode* は以下のうちのいずれかです

'r'　読み出しのみのモード。

'w'　書き込みのみのモード。

読み込み／書き込み両方のモードで開くことはできないことに注意して下さい。

'r' の *mode* は AU_read オブジェクトを返し、'w' と 'wb' の *mode* は AU_write オブジェクトを返します。

sunau.**openfp**(*file*, *mode*)
open() と同義。後方互換性のために残されています。

Deprecated since version 3.7, will be removed in version 3.9.

sunau モジュールは以下の例外を定義しています:

exception sunau.**Error**
Sun AU の仕様や実装に対する不適切な操作により何か実行不可能となった時に発生するエラー。

sunau モジュールは以下のデータアイテムを定義しています:

sunau.**AUDIO_FILE_MAGIC**
big-endian で保存された正規の Sun AU ファイルは全てこの整数で始まります。これは文字列 .snd を整数に変換したものです。

sunau.**AUDIO_FILE_ENCODING_MULAW_8**
sunau.**AUDIO_FILE_ENCODING_LINEAR_8**
sunau.**AUDIO_FILE_ENCODING_LINEAR_16**
sunau.**AUDIO_FILE_ENCODING_LINEAR_24**
sunau.**AUDIO_FILE_ENCODING_LINEAR_32**
sunau.**AUDIO_FILE_ENCODING_ALAW_8**
AU ヘッダの encoding フィールドの値で、このモジュールでサポートしているものです。

sunau.**AUDIO_FILE_ENCODING_FLOAT**
sunau.**AUDIO_FILE_ENCODING_DOUBLE**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G721**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G722**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G723_3**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G723_5**
AU ヘッダの encoding フィールドの値のうち既知のものとして追加されているものですが、このモジュールではサポートされていません。

22.3.1 AU_read オブジェクト

上述の *open()* によって返される AU_read オブジェクトには、以下のメソッドがあります:

AU_read.close()
ストリームを閉じ、このオブジェクトのインスタンスを使用できなくします。(これはオブジェクトのガベージコレクション時に自動的に呼び出されます。)

AU_read.getnchannels()
オーディオチャンネル数(モノラルなら 1、ステレオなら 2)を返します。

AU_read.getsampwidth()
サンプルサイズをバイト数で返します。

AU_read.getframerate()
サンプリングレートを返します。

AU_read.getnframes()
オーディオフレーム数を返します。

AU_read.getcomptype()
圧縮形式を返します。'ULAW', 'ALAW', 'NONE' がサポートされている形式です。

AU_read.getcompname()
getcomptype() を人に判読可能な形にしたものです。上述の形式に対して、それぞれ 'CCITT G.711 u-law', 'CCITT G.711 A-law', 'not compressed' がサポートされています。

AU_read.getparams()
get*() メソッドが返すのと同じ (nchannels, sampwidth, framerate, nframes, comptype, compname) の *namedtuple()* を返します。

AU_read.readframes(*n*)
n 個のオーディオフレームの値を読み込んで、*bytes* オブジェクトを返します。データは linear 形式で返されます。もし元のデータが u-LAW 形式なら、変換されます。

AU_read.rewind()
ファイルのポインタをオーディオストリームの先頭に戻します。

以下の 2 つのメソッドは共通の" 位置" を定義しています。" 位置" は他の関数とは独立して実装されています。

AU_read.setpos(*pos*)
ファイルのポインタを指定した位置に設定します。*tell()* で返される値を *pos* として使用しなければなりません。

AU_read.tell()
ファイルの現在のポインタ位置を返します。返される値はファイルの実際の位置に対して何も操作はしません。

以下の 2 つのメソッドは *aifc* モジュールとの互換性のために定義されていますが、何も面白いことはしません。

AU_read.getmarkers()
None を返します。

AU_read.getmark(*id*)
エラーを発生します。

22.3.2 AU_write オブジェクト

上述の *open()* によって返される Wave_write オブジェクトには、以下のメソッドがあります:

AU_write.setnchannels(*n*)
チャンネル数を設定します。

AU_write.setsampwidth(*n*)
サンプルサイズを（バイト数で）設定します。

バージョン 3.4 で変更: 24-bit サンプルのサポートが追加されました。

AU_write.setframerate(*n*)
フレームレートを設定します。

AU_write.setnframes(*n*)
フレーム数を設定します。あとからフレームが書き込まれるとフレーム数は変更されます。

AU_write.setcomptype(*type*, *name*)
圧縮形式とその記述を設定します。'NONE' と 'ULAW' だけが、出力時にサポートされている形式です。

AU_write.setparams(*tuple*)
tuple は (nchannels, sampwidth, framerate, nframes, comptype, compname) で、それぞれ set*() のメソッドの値にふさわしいものでなければなりません。全ての変数を設定します。

AU_write.tell()
ファイルの中の現在位置を返します。*AU_read.tell()* と *AU_read.setpos()* メソッドでお断りしたことがこのメソッドにも当てはまります。

AU_write.writeframesraw(*data*)
nframes の修正なしにオーディオフレームを書き込みます。

バージョン 3.4 で変更: どのような *bytes-like object* も使用できるようになりました。

AU_write.writeframes(*data*)
オーディオフレームを書き込んで *nframes* を修正します。

バージョン 3.4 で変更: どのような *bytes-like object* も使用できるようになりました。

AU_write.close()
nframes が正しいか確認して、ファイルを閉じます。

このメソッドはオブジェクトの削除時に呼び出されます。

`writeframes()` や `writeframesraw()` メソッドを呼び出したあとで、どんなパラメータを設定しようとしても不正となることに注意して下さい。

22.4 `wave` --- WAV ファイルの読み書き

ソースコード: Lib/wave.py

---

wave モジュールは、WAV サウンドフォーマットへの便利なインターフェイスを提供するモジュールです。このモジュールは圧縮／展開をサポートしていませんが、モノラル／ステレオには対応しています。

wave モジュールは、以下の関数と例外を定義しています:

`wave.open`(*file*, *mode=None*)

file が文字列ならその名前のファイルを開き、そうでないなら file like オブジェクトとして扱います。*mode* は以下のうちのいずれかです:

`'rb'` 読み出しのみのモード。

`'wb'` 書き込みのみのモード。

WAV ファイルに対して読み込み／書き込み両方のモードで開くことはできないことに注意して下さい。

mode が `'rb'` の場合 `Wave_read` オブジェクトを返し、`'wb'` の場合 `Wave_write` オブジェクトを返します。*mode* が省略されていて、file-like オブジェクトが *file* として渡されると、`file.mode` が *mode* のデフォルト値として使われます。

file like オブジェクトを渡した場合、wave オブジェクトの `close()` メソッドを呼び出してもその file like オブジェクトを close しません。file like オブジェクトの close は呼び出し側の責任になります。

open() 関数は `with` 文とともに使うことができます。`with` ブロックが終わるときに、*Wave_read.close()* メソッドまたは *Wave_write.close()* メソッドが呼ばれます。

バージョン 3.4 で変更: シーク不能なファイルのサポートが追加されました。

`wave.openfp`(*file*, *mode*)

open() と同義。後方互換性のために残されています。

Deprecated since version 3.7, will be removed in version 3.9.

exception `wave.Error`

WAV の仕様を犯したり、実装の欠陥に遭遇して何か実行不可能となった時に発生するエラー。

22.4.1 Wave_read オブジェクト

open() によって返される Wave_read オブジェクトには、以下のメソッドがあります:

Wave_read.close()
wave によって開かれていた場合はストリームを閉じ、このオブジェクトのインスタンスを使用できなくします。これはオブジェクトのガベージコレクション時に自動的に呼び出されます。

Wave_read.getnchannels()
オーディオチャンネル数（モノラルなら 1 、ステレオなら 2 ）を返します。

Wave_read.getsampwidth()
サンプルサイズをバイト数で返します。

Wave_read.getframerate()
サンプリングレートを返します。

Wave_read.getnframes()
オーディオフレーム数を返します。

Wave_read.getcomptype()
圧縮形式を返します（ `'NONE'` だけがサポートされている形式です）。

Wave_read.getcompname()
getcomptype() を人に判読可能な形にしたものです。通常、`'NONE'` に対して `'not compressed'` が返されます。

Wave_read.getparams()
`get*()` メソッドが返すのと同じ (`nchannels`, `sampwidth`, `framerate`, `nframes`, `comptype`, `compname`) の *namedtuple()* を返します。

Wave_read.readframes(*n*)
最大 *n* 個のオーディオフレームを読み込んで、*bytes* オブジェクトとして返します。

Wave_read.rewind()
ファイルのポインタをオーディオストリームの先頭に戻します。

以下の 2 つのメソッドは *aifc* モジュールとの互換性のために定義されており、何も面白いことはしません。

Wave_read.getmarkers()
`None` を返します。

Wave_read.getmark(*id*)
エラーを発生します。

以下の 2 つのメソッドは共通の” 位置” を定義しています。” 位置” は他の関数とは独立して実装されています。

Wave_read.setpos(*pos*)
ファイルのポインタを指定した位置に設定します。

Wave_read.tell()
ファイルの現在のポインタ位置を返します。

22.4.2 Wave_write オブジェクト

seek 可能な出力ストリームでは、実際に書き込まれたフレーム数を反映するように wave ヘッダーは自動的にアップデートされます。seek 不能なストリームでは、最初のフレームデータが書き込まれる時には *nframes* の値は正確でなければなりません。*nframes* の正確な値は、*close()* が呼ばれる前に書き込まれるフレーム数とともに *setnframes()* または *setparams()* を呼んで、その後 *writeframesraw()* を利用してフレームデータを書くか、もしくはすべての書き込むべきフレームデータとともに *writeframes()* を呼び出すことによって達成できます。後者のケースでは、*writeframes()* はフレームデータを書く前に、データ中のフレームの数を計算して、それに応じて *nframes* を設定します。

open() によって返される Wave_write オブジェクトには、以下のメソッドがあります:

バージョン 3.4 で変更: シーク不能なファイルのサポートが追加されました。

Wave_write.close()
nframes が正しいか確認して、ファイルが *wave* によって開かれていた場合は閉じます。このメソッドはオブジェクトがガベージコレクションされるときに呼び出されます。もし出力ストリームがシーク不能で、*nframes* が実際に書き込まれたフレームの数と一致しなければ、例外が起きます。

Wave_write.setnchannels(*n*)
チャンネル数を設定します。

Wave_write.setsampwidth(*n*)
サンプルサイズを *n* バイトに設定します。

Wave_write.setframerate(*n*)
サンプリングレートを *n* に設定します。

バージョン 3.2 で変更: 整数ではない値がこのメソッドに入力された場合は、直近の整数に丸められます。

Wave_write.setnframes(*n*)
フレーム数を *n* に設定します。もし実際に書き込まれたフレームの数と異なるなら、これは後で変更されます (出力ストリームがシーク不能なら、更新しようとした時にエラーが起きます)。

Wave_write.setcomptype(*type*, *name*)
圧縮形式とその記述を設定します。現在のところ、非圧縮を示す圧縮形式 NONE だけがサポートされています。

Wave_write.setparams(*tuple*)
この *tuple* は (nchannels, sampwidth, framerate, nframes, comptype, compname) でなければならず、その値はそれぞれの set*() メソッドで有効でなければなりません。すべてのパラメータを設定します。

Wave_write.tell()

ファイルの中の現在位置を返します。*Wave_read.tell()* と *Wave_read.setpos()* メソッドでお断りしたことがこのメソッドにも当てはまります。

Wave_write.**writeframesraw**(*data*)

nframes の修正なしにオーディオフレームを書き込みます。

バージョン 3.4 で変更: どのような *bytes-like object* も使用できるようになりました。

Wave_write.**writeframes**(*data*)

出力ストリームが seek 不可能で、*data* が書き込まれた後でそれ以前に *nframes* に設定された値と書き込まれた全フレーム数が一致しなければ、エラーを送出します。

バージョン 3.4 で変更: どのような *bytes-like object* も使用できるようになりました。

`writeframes()` や `writeframesraw()` メソッドを呼び出したあとで、どんなパラメータを設定しようとしても不正となることに注意して下さい。そうすると *wave.Error* を発生します。

22.5 chunk --- IFF チャンクデータの読み込み

ソースコード: Lib/chunk.py

このモジュールは EA IFF 85 チャンクを使用しているファイルの読み込みのためのインターフェースを提供します。[*1] このフォーマットは少なくとも、Audio Interchange File Format (AIFF/AIFF-C) と Real Media File Format (RMFF) で使われています。WAVE オーディオファイルフォーマットも厳密に対応しているので、このモジュールで読み込みできます。

チャンクは以下の構造を持っています:

Offset 値	長さ	内容
0	4	チャンク ID
4	4	big- endian で示したチャンクのサイズで、ヘッダは含みません
8	n	バイトデータで、n はこれより先のフィールドのサイズ
$8 + n$	0 or 1	n が奇数ならチャンクの整頓のために埋められるバイト

ID はチャンクの種類を識別する 4 バイトの文字列です。

サイズフィールド（big-endian でエンコードされた 32 ビット値）は、8 バイトのヘッダを含まないチャンクデータのサイズを示します。

普通、IFF タイプのファイルは 1 個かそれ以上のチャンクからなります。このモジュールで定義される *Chunk* クラスの使い方として提案しているのは、それぞれのチャンクの始めにインスタンスを作り、終わりに達するまでそのインスタンスから読み取り、その後で新しいインスタンスを作るということです。ファイルの終わりで新しいインスタンスを作ろうとすると、*EOFError* の例外が発生して失敗します。

[*1] "EA IFF 85" Standard for Interchange Format Files, Jerry Morrison, Electronic Arts, January 1985.

class chunk.Chunk(*file*, *align=True*, *bigendian=True*, *inclheader=False*)

チャンクを表わすクラス。*file* 引数はファイル風オブジェクトであると期待されます。このクラスのインスタンスは特別に許可されます。唯一の必要なメソッドは read() です。メソッド *seek()* および *tell()* が存在し、例外を上げない場合、それらも使用されます。これらのメソッドが存在し、例外を上げる場合、それらのメソッドはオブジェクトを変更しないことが想定されます。オプションの引数 *align* が true の場合、チャンクは 2 バイト境界上で整列されていると仮定されます。*align* が false の場合、整列は仮定されません。デフォルト値は true です。オプションの引数 *bigendian* が false の場合、チャンクサイズはリトルエンディアン順になっていると仮定されます。これは WAVE オーディオファイルに必要とされます。デフォルト値は true です。オプションの引数 *inclheader* が true の場合、チャンクヘッダ中で与えられたサイズはヘッダのサイズを含んでいます。デフォルト値は false です。

Chunk オブジェクトには以下のメソッドが定義されています:

getname()

チャンクの名前（ID）を返します。これはチャンクの始めの 4 バイトです。

getsize()

チャンクのサイズを返します。

close()

オブジェクトを閉じて、チャンクの終わりまで飛びます。これは元のファイル自体は閉じません。

close() メソッドが呼ばれた後で他のメソッドを呼ぶと *OSError* が送出されます。Python 3.3 以前は *IOError* (現在は *OSError* の別名) が送出されていました。

isatty()

False を返します。

seek(*pos*, *whence=0*)

チャンクの現在位置を設定します。引数 *whence* は省略可能で、デフォルト値は 0（ファイルの絶対位置）です; 他に 1（現在位置から相対的にシークします）と 2（ファイルの末尾から相対的にシークします）の値を取ります。何も値は返しません。もし元のファイルがシークに対応していなければ、前方へのシークのみが可能です。

tell()

チャンク内の現在位置を返します。

read(*size=-1*)

チャンクから最大で *size* バイト読み込みます (*size* バイトを読み込むより前にチャンクの最後に行き着いたら、それより少なくなります) 。もし引数 *size* が負か省略されたら、チャンクの最後まで全てのデータを読み込みます。チャンクの最後に行き着いたら、空の bytes オブジェクトを返します。

skip()

チャンクの最後まで飛びます。さらにチャンクの *read()* を呼び出すと、b'' が返されます。もしチャンクの内容に興味がないなら、このメソッドを呼び出してファイルポインタを次のチャンクの始めに設定します。

脚注

22.6 colorsys --- 色体系間の変換

ソースコード: Lib/colorsys.py

---

colorsys モジュールは、計算機のディスプレイモニタで使われている RGB (Red Green Blue) 色空間で表された色と、他の 3 種類の色座標系: YIQ, HLS (Hue Lightness Saturation: 色相、彩度、飽和) および HSV (Hue Saturation Value: 色相、彩度、明度) との間の双方向の色値変換を定義します。これらの色空間における色座標系は全て浮動小数点数で表されます。YIQ 空間では、Y 軸は 0 から 1 ですが、I および Q 軸は正の値も負の値もとり得ます。他の色空間では、各軸は全て 0 から 1 の値をとります。

参考:

色空間に関するより詳細な情報は http://poynton.ca/ColorFAQ.html と https://www.cambridgeincolour.com/tutorials/color-spaces.htm にあります。

colorsys モジュールでは、以下の関数が定義されています:

colorsys.rgb_to_yiq(*r*, *g*, *b*)
RGB から YIQ に変換します。

colorsys.yiq_to_rgb(*y*, *i*, *q*)
YIQ から RGB に変換します。

colorsys.rgb_to_hls(*r*, *g*, *b*)
RGB から HLS に変換します。

colorsys.hls_to_rgb(*h*, *l*, *s*)
HLS から RGB に変換します。

colorsys.rgb_to_hsv(*r*, *g*, *b*)
RGB から HSV に変換します。

colorsys.hsv_to_rgb(*h*, *s*, *v*)
HSV から RGB に変換します。

以下はプログラム例です:

```
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

22.7 imghdr --- 画像の形式を決定する

ソースコード: Lib/imghdr.py

---

imghdr モジュールはファイルやバイトストリームに含まれる画像の形式を決定します。

imghdr モジュールは次の関数を定義しています:

imghdr.**what**(*filename*, *h=None*)

filename という名前のファイル内の画像データをテストし、画像形式を表す文字列を返します。オプションの *h* が与えられた場合は、*filename* は無視され、テストするバイトストリームを含んでいると *h* は仮定されます。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

以下に *what()* からの戻り値とともにリストするように、次の画像形式が認識されます:

値	Image format
'rgb'	SGI ImgLib Files
'gif'	GIF 87a and 89a Files
'pbm'	Portable Bitmap Files
'pgm'	Portable Graymap Files
'ppm'	Portable Pixmap Files
'tiff'	TIFF Files
'rast'	Sun Raster Files
'xbm'	X Bitmap Files
'jpeg'	JPEG data in JFIF or Exif formats
'bmp'	BMP files
'png'	Portable Network Graphics
'webp'	WebP files
'exr'	OpenEXR Files

バージョン 3.5 で追加: フォーマット*exr*と*webp*が追加されました.

この変数に追加することで、あなたは *imghdr* が認識できるファイル形式のリストを拡張できます:

imghdr.**tests**

個別のテストを行う関数のリスト。それぞれの関数は二つの引数をとります: バイトストリームとオープンされたファイルのようにふるまうオブジェクト。*what()* がバイトストリームとともに呼び出されたときは、ファイルのようにふるまうオブジェクトは None でしょう。

テストが成功した場合は、テスト関数は画像形式を表す文字列を返すべきです。あるいは、失敗した場合は None を返すべきです。

以下はプログラム例です:

```
>>> import imghdr
>>> imghdr.what('bass.gif')
'gif'
```

22.8 sndhdr --- サウンドファイルの識別

ソースコード: Lib/sndhdr.py

---

sndhdr モジュールには、ファイルに保存されたサウンドデータの形式を識別するのに便利な関数が定義されています。どんな形式のサウンドデータがファイルに保存されているのか識別可能な場合、これらの関数は 5 つの属性、つまり (`filetype`, `framerate`, `nchannels`, `nframes`, `sampwidth`) で構成される *namedtuple()* を返します。*type* はデータの形式を示す文字列で、`'aifc'`, `'aiff'`, `'au'`, `'hcom'`, `'sndr'`, `'sndt'`, `'voc'`, `'wav'`, `'8svx'`, `'sb'`, `'ub'`, `'ul'` のうちの一つです。*sampling_rate* は実際のサンプリングレート値で、未知の場合や読み取ることが出来なかった場合は `0` です。同様に、*channels* はチャンネル数で、識別できない場合や読み取ることが出来なかった場合は `0` です。*frames* はフレーム数で、識別できない場合は `-1` です。タプルの最後の要素 *bits_per_sample* はサンプルサイズを示すビット数ですが、A-LAW なら `'A'`, u-LAW なら `'U'` です。

sndhdr.what(*filename*)

whathdr() を使って、ファイル *filename* に保存されたサウンドデータの形式を識別します。識別可能なら上記の namedtuple を返し、識別できない場合は `None` を返します。

バージョン 3.5 で変更: 結果が tuple から namedtuple に変更されました。

sndhdr.whathdr(*filename*)

ファイルのヘッダ情報をもとに、保存されたサウンドデータの形式を識別します。ファイル名は *filename* で渡されます。識別可能なら上記の namedtuple を返し、識別できない場合は `None` を返します。

バージョン 3.5 で変更: 結果が tuple から namedtuple に変更されました。

22.9 ossaudiodev --- OSS 互換オーディオデバイスへのアクセス

このモジュールを使うと OSS (Open Sound System) オーディオインターフェースにアクセスできます。OSS はオープンソースあるいは商用の Unix で広く利用でき、Linux (カーネル 2.4 まで) と FreeBSD で標準のオーディオインターフェースです。

バージョン 3.3 で変更: このモジュールの操作で以前は *IOError* が送出されていたところで *OSError* が送出されるようになりました。

参考:

Open Sound System Programmer's Guide　OSS C API の公式ドキュメント

このモジュールでは、OSS デバイスドライバが提供している大量の定数が定義されています; 一覧は Linux や FreeBSD 上の <sys/soundcard.h> を参照してください。

ossaudiodev では以下の変数と関数を定義しています:

exception ossaudiodev.**OSSAudioError**

何らかのエラーのときに送出される例外です。引数は何が誤っているかを示す文字列です。

(*ossaudiodev* が open()、write() 、ioctl() などのシステムコールからエラーを受け取った場合、*OSError* を送出します。*ossaudiodev* によって直接検知されたエラーでは *OSSAudioError* になります。)

(以前のバージョンとの互換性のため、この例外クラスは ossaudiodev.error としても利用できます。)

ossaudiodev.**open**(*mode*)

ossaudiodev.**open**(*device*, *mode*)

オーディオデバイスを開き、OSS オーディオデバイスオブジェクトを返します。このオブジェクトは read() 、write() 、fileno() といったファイル類似オブジェクトのメソッドを数多くサポートしています。(とはいえ、伝統的な Unix の read/write における意味づけと OSS デバイスの read/write との間には微妙な違いがあります)。また、オーディオ特有の多くのメソッドがあります; メソッドの完全なリストについては下記を参照してください。

device は使用するオーディオデバイスファイルネームです。もしこれが指定されないなら、このモジュールは使うデバイスとして最初に環境変数 AUDIODEV を参照します。見つからなければ /dev/dsp を参照します。

mode は読み出し専用アクセスの場合には 'r'、書き込み専用 (プレイバック) アクセスの場合には 'w'、読み書きアクセスの場合には 'rw' にします。多くのサウンドカードは一つのプロセスが一度にレコーダとプレーヤのどちらかしか開けないようにしているため、必要な操作に応じたデバイスだけを開くようにするのがよいでしょう。また、サウンドカードには半二重 (half- duplex) 方式のものがあります: こうしたカードでは、デバイスを読み出しまたは書き込み用に開くことはできますが、両方同時には開けません。

呼び出しの文法が普通と異なることに注意してください: **最初の** 引数は省略可能で、2 番目が必須です。これは *ossaudiodev* にとってかわられた古い linuxaudiodev との互換性のためという歴史的な産物です。

ossaudiodev.**openmixer**([*device*])

ミキサデバイスを開き、OSS ミキサデバイスオブジェクトを返します。*device* は使用するミキサデバイスのファイル名です。*device* を指定しない場合、モジュールはまず環境変数 MIXERDEV を参照して使用するデバイスを探します。見つからなければ、/dev/mixer を参照します。

22.9.1 オーディオデバイスオブジェクト

オーディオデバイスに読み書きできるようになるには、まず 3 つのメソッドを正しい順序で呼び出さねばなりません:

1. `setfmt()` で出力形式を設定し
2. `channels()` でチャンネル数を設定し
3. `speed()` でサンプリングレートを設定します

この代わりに `setparameters()` メソッドを呼び出せば、三つのオーディオパラメタを一度で設定できます。`setparameters()` は便利ですが、多くの状況で柔軟性に欠けるでしょう。

open() の返すオーディオデバイスオブジェクトには以下のメソッドおよび (読み出し専用の) 属性があります:

`oss_audio_device.close()`
: オーディオデバイスを明示的に閉じます。オーディオデバイスは、読み出しや書き込みが終了したら必ず閉じねばなりません。閉じたオブジェクトを再度開くことはできません。

`oss_audio_device.fileno()`
: デバイスに関連付けられているファイル記述子を返します。

`oss_audio_device.read(`*size*`)`
: オーディオ入力から *size* バイトを読みだし、Python 文字列型にして返します。多くの Unix デバイスドライバと違い、ブロックデバイスモード (デフォルト) の OSS オーディオデバイスでは、要求した量のデータ全体を取り込むまで *read()* がブロックします。

`oss_audio_device.write(`*data*`)`
: *bytes-like object* *data* の内容をオーディオデバイスに書き込み、書き込まれたバイト数を返します。オーディオデバイスがブロックモード (デフォルト) の場合、常にデータ全体を書き込みます (前述のように、これは通常の Unix デバイスの振舞いとは異なります)。デバイスが非ブロックモードの場合、データの一部が書き込まれないことがあります --- *writeall()* を参照してください。

  バージョン 3.5 で変更: 書き込み可能な *bytes-like object* を使用できるようになりました。

`oss_audio_device.writeall(`*data*`)`
: *bytes-like object* *data* をオーディオデバイスに書き込みます。オーディオデバイスがデータを受け取れるようになるまで待機し、書き込めるだけのデータを書き込むという操作を、*data* を全て書き込み終わるまで繰り返します。デバイスがブロックモード (デフォルト) の場合には、このメソッドは *write()* と同じです。*writeall()* が有用なのは非ブロックモードだけです。実際に書き込まれたデータの量と渡したデータの量は必ず同じになるので、戻り値はありません。

  バージョン 3.5 で変更: 書き込み可能な *bytes-like object* を使用できるようになりました。

バージョン 3.2 で変更: オーディオデバイスオブジェクトはコンテキストマネージメントプロトコルをサポートしています。つまり `with` 文で使うことができます。

以下の各メソッドは、厳密に 1 つの `ioctl()` システムコールに対応しています。対応関係は明らかです: 例え

ば、setfmt() は ioctl の SNDCTL_DSP_SETFMT に対応し、sync() は SNDCTL_DSP_SYNC に対応します (これは OSS のドキュメントを調べるときに便利です) 。中で呼んでいる ioctl() が失敗した場合は、*OSError* が送出されます。

oss_audio_device.**nonblock**()
デバイスを非ブロックモードにします。いったん非ブロックモードにしたら、ブロックモードは戻せません。

oss_audio_device.**getfmts**()
サウンドカードがサポートしているオーディオ出力形式をビットマスクで返します。以下は OSS でサポートされているフォーマットの一部です:

フォーマット	説明
AFMT_MU_LAW	対数符号化 (Sun の .au 形式や /dev/audio で使われている形式)
AFMT_A_LAW	対数符号化
AFMT_IMA_ADPCM	Interactive Multimedia Association で定義されている 4:1 圧縮形式
AFMT_U8	符号なし 8 ビットオーディオ
AFMT_S16_LE	符号つき 16 ビットオーディオ、リトルエンディアンバイトオーダ (Intel プロセッサで使われている形式)
AFMT_S16_BE	符号つき 16 ビットオーディオ、ビッグエンディアンバイトオーダ (68k、PowerPC、Sparc で使われている形式)
AFMT_S8	符号つき 8 ビットオーディオ
AFMT_U16_LE	符号なし 16 ビットリトルエンディアンオーディオ
AFMT_U16_BE	符号なし 16 ビットビッグエンディアンオーディオ

オーディオ形式の完全なリストは OSS の文書をひもといてください。ただ、ほとんどのシステムは、こうした形式のサブセットしかサポートしていません。古めのデバイスの中には AFMT_U8 だけしかサポートしていないものがあります。現在使われている最も一般的な形式は AFMT_S16_LE です。

oss_audio_device.**setfmt**(*format*)
現在のオーディオ形式を *format* に設定しようと試みます --- *format* については *getfmts()* のリストを参照してください。実際にデバイスに設定されたオーディオ形式を返します。要求通りの形式でないこともあります。AFMT_QUERY を渡すと現在デバイスに設定されているオーディオ形式を返します。

oss_audio_device.**channels**(*nchannels*)
出力チャネル数を *nchannels* に設定します。1 はモノラル、2 はステレオです。いくつかのデバイスでは 2 つより多いチャンネルを持つものもありますし、ハイエンドなデバイスではモノラルをサポートしないものもあります。デバイスに設定されたチャンネル数を返します。

oss_audio_device.**speed**(*samplerate*)
サンプリングレートを 1 秒あたり *samplerate* に設定しようと試み、実際に設定されたレートを返します。たいていのサウンドデバイスでは任意のサンプリングレートをサポートしていません。一般的なレートは以下の通りです:

レート	説明
8000	/dev/audio のデフォルト
11025	会話音声の録音に使われるレート
22050	
44100	(サンプルあたり 16 ビットで 2 チャネルの場合) CD 品質のオーディオ
96000	(サンプル当たり 24 ビットの場合) DVD 品質のオーディオ

oss_audio_device.**sync**()

サウンドデバイスがバッファ内の全てのデータを再生し終えるまで待機します。(デバイスを閉じると暗黙のうちに *sync()* が起こります) OSS のドキュメント上では、*sync()* を使うよりデバイスを一度閉じて開き直すよう勧めています。

oss_audio_device.**reset**()

再生あるいは録音を即座に中止して、デバイスをコマンドを受け取れる状態に戻します。OSS のドキュメントでは、*reset()* を呼び出した後に一度デバイスを閉じ、開き直すよう勧めています。

oss_audio_device.**post**()

ドライバに出力の一時停止 (pause) が起きそうであることを伝え、ドライバが一時停止をより賢く扱えるようにします。短いサウンドエフェクトを再生した直後やユーザ入力待ちの前、またディスク I/O 前などに使うことになるでしょう。

以下のメソッドは、複数の ioctl() を組み合わせたり、ioctl() と単純な計算を組み合わせたりした便宜用メソッドです。

oss_audio_device.**setparameters**(*format*, *nchannels*, *samplerate*[, *strict=False*])

主要なオーディオパラメタ、サンプル形式、チャネル数、サンプルレートを一つのメソッド呼び出しで設定します。*format* 、*nchannels* および *samplerate* には、それぞれ *setfmt()* 、*channels()* および *speed()* と同じやり方で値を設定します。*strict* の値が真の場合、*setparameters()* は値が実際に要求通りにデバイスに設定されたかどうか調べ、違っていれば *OSSAudioError* を送出します。実際にデバイスドライバが設定したパラメタ値を表す (*format*, *nchannels*, *samplerate*) からなるタプルを返します (*setfmt()* 、*channels()* および *speed()* の返す値と同じです)。

例えば、

```
(fmt, channels, rate) = dsp.setparameters(fmt, channels, rate)
```

は以下と同等です

```
fmt = dsp.setfmt(fmt)
channels = dsp.channels(channels)
rate = dsp.rate(rate)
```

oss_audio_device.**bufsize**()

ハードウェアのバッファサイズをサンプル数で返します。

oss_audio_device.**obufcount**()

ハードウェアバッファ上に残っていてまだ再生されていないサンプル数を返します。

oss_audio_device.obuffree()
: ブロックを起こさずにハードウェアの再生キューに書き込めるサンプル数を返します。

オーディオデバイスオブジェクトは読み出し専用の属性もサポートしています:

oss_audio_device.closed
: デバイスが閉じられたかどうかを示す真偽値です。

oss_audio_device.name
: デバイスファイルの名前を含む文字列です。

oss_audio_device.mode
: ファイルの I/O モードで、"r", "rw", "w" のどれかです。

22.9.2 ミキサデバイスオブジェクト

ミキサオブジェクトには、2 つのファイル類似メソッドがあります:

oss_mixer_device.close()
: このメソッドは開いているミキサデバイスオブジェクトを閉じます。このファイルを閉じた後もミキサを使おうとすると、*OSError* が送出されます。

oss_mixer_device.fileno()
: 開かれているミキサデバイスファイルのファイルハンドルナンバを返します。

バージョン 3.2 で変更: ミキサオブジェクトはコンテキストマネージメントプロトコルもサポートします。

以下はオーディオミキシング固有のメソッドです:

oss_mixer_device.controls()
: このメソッドは、利用可能なミキサコントロール (SOUND_MIXER_PCM や SOUND_MIXER_SYNTH のように、ミキシングを行えるチャネル) を指定するビットマスクを返します。このビットマスクは利用可能な全てのミキサコントロールのサブセットです --- 定数 SOUND_MIXER_* はモジュールレベルで定義されています。例えば、もし現在のミキサオブジェクトが PCM ミキサをサポートしているか調べるには、以下の Python コードを実行します:

```
mixer=ossaudiodev.openmixer()
if mixer.controls() & (1 << ossaudiodev.SOUND_MIXER_PCM):
    # PCM is supported
    ... code ...
```

: ほとんどの用途には、SOUND_MIXER_VOLUME (マスタボリューム) と SOUND_MIXER_PCM コントロールがあれば十分でしょう --- とはいえ、ミキサを使うコードを書くときには、コントロールを選ぶ時に柔軟性を持たせるべきです。例えば Gravis Ultrasound には SOUND_MIXER_VOLUME がありません。

oss_mixer_device.stereocontrols()
: ステレオミキサコントロールを示すビットマスクを返します。ビットが立っているコントロールはステレオであることを示し、立っていないコントロールはモノラルか、ミキサがサポートしていないコン

トロールである (どちらの理由かは *controls()* と組み合わせて使うことで判別できます) ことを示します。

ビットマスクから情報を得る例は関数 *controls()* のコード例を参照してください。

oss_mixer_device.**reccontrols**()

録音に使用できるミキサコントロールを特定するビットマスクを返します。ビットマスクから情報を得る例は関数 *controls()* のコード例を参照してください。

oss_mixer_device.**get**(*control*)

指定したミキサコントロールのボリュームを返します。2 要素のタプル (left_volume, right_volume) を返します。ボリュームの値は 0 (無音) から 100 (最大) で示されます。コントロールがモノラルでも 2 要素のタプルが返されますが、2 つの要素の値は同じになります。

不正なコントロールを指定した場合は *OSSAudioError* を、サポートされていないコントロールを指定した場合は *OSError* を送出します。

oss_mixer_device.**set**(*control*, *(left*, *right)*)

指定したミキサコントロールのボリュームを (left,right) に設定します。left と right は整数で、0 (無音) から 100 (最大) の間で指定せねばなりません。呼び出しに成功すると新しいボリューム値を 2 要素のタプルで返します。サウンドカードによっては、ミキサの分解能上の制限から、指定したボリュームと厳密に同じにはならない場合があります。

不正なコントロールを指定した場合や、指定したボリューム値が範囲外であった場合、*OSSAudioError* を送出します。

oss_mixer_device.**get_recsrc**()

現在録音のソースに使われているコントロールを示すビットマスクを返します。

oss_mixer_device.**set_recsrc**(*bitmask*)

録音のソースを指定するために、この関数を呼び出します。成功した場合は、新しい録音のソース (1 つもしくは複数) を示すビットマスクを返します; 不正なソースを指定した場合は、*OSError* を送出します。現在の録音のソースをマイク入力に設定するには以下のように呼び出します:

```
mixer.setrecsrc (1 << ossaudiodev.SOUND_MIXER_MIC)
```

第

TWENTYTHREE

国際化

この章で解説されるモジュールは、プログラムのメッセージで使用される言語を選択したり、または出力を地域の習慣に従って変更するメカニズムを提供して、言語や地域に依存しないソフトの開発を手助けします。

この章で解説されるモジュールのリスト:

23.1 gettext --- 多言語対応に関する国際化サービス

ソースコード: Lib/gettext.py

---

gettext モジュールは、Python のモジュールやアプリケーションの国際化 (I18N, I-nternationalizatio-N) および地域化 (L10N, L-ocalizatio-N) サービスを提供します。このモジュールは GNU **gettext** メッセージカタログの API と、より高水準で Python ファイルに適しているクラス形式の API の両方をサポートしてます。以下で述べるインタフェースを使うことで、モジュールやアプリケーションのメッセージをある自然言語で記述しておき、後から提供する翻訳されたメッセージのカタログによって様々な自然言語環境で実行できます。

ここでは Python のモジュールやアプリケーションを地域化するためのいくつかのヒントも提供しています。

23.1.1 GNU gettext API

gettext モジュールでは、以下の GNU **gettext** API に非常に良く似た API を提供しています。この API を使う場合、アプリケーション全体の翻訳に影響します。アプリケーションが単一の言語しか扱わず、ユーザのロケールに従って言語が選ばれるのなら、たいていはこの API が求めているものです。Python モジュールを地域化していたり、アプリケーションの実行中に言語を切り替える必要がある場合は、この API ではなくおそらくクラス形式の API を使いたくなるでしょう。

gettext.**bindtextdomain**(*domain*, *localedir=None*)

domain をロケールディレクトリ *localedir* に対応付けます。具体的には、*gettext* は与えられたドメインに対するバイナリ形式の `.mo` ファイルを探しに、(Unix では) *localedir*`/`*language*`/LC_MESSAGES/`*domain*`.mo` というパスを見に行きます。ここで *language* はそれぞれ環境変数 `LANGUAGE` 、`LC_ALL` 、`LC_MESSAGES` 、`LANG` の中から検索されます。

localedir が省略されるか None の場合、現在 *domain* に対応付けられているロケールディレクトリが返されます。[*1]

gettext.**bind_textdomain_codeset**(*domain*, *codeset=None*)

domain を *codeset* に対応付け、*lgettext()*, *ldgettext()*, *lngettext()*, *ldngettext()* 関数が返すバイト文字列のエンコード方式を変更します。*codeset* が省略された場合は、現在 *domain* に対応付けられているコードセットを返します。

Deprecated since version 3.8, will be removed in version 3.10.

gettext.**textdomain**(*domain=None*)

現在のグローバルドメインを変更したり調べたりします。*domain* が None の場合、現在のグローバルドメインが返されます。それ以外の場合には、グローバルドメインに *domain* を設定し、その設定されたグローバルドメインを返します。

gettext.**gettext**(*message*)

現在のグローバルドメイン、言語、およびロケールディレクトリに基づいて、*message* の地域化された訳文を返します。通常、この関数はローカルな名前空間にある _() という別名を持ちます (下の例を参照してください)。

gettext.**dgettext**(*domain*, *message*)

gettext() と同様ですが、指定された *domain* からメッセージを探します。

gettext.**ngettext**(*singular*, *plural*, *n*)

gettext() と同様ですが、複数形を考慮しています。翻訳が見つかった場合、複数形の選択公式を *n* に適用し、その結果得られたメッセージを返します (言語によっては二つ以上の複数形があります)。翻訳が見つからなかった場合、*n* が 1 なら *singular* を返します; そうでない場合 *plural* を返します。

複数形の選択公式はカタログのヘッダから取得されます。選択公式は自由変数 *n* を持つ C または Python の式です; その式の評価結果はカタログにある複数形のインデックスになります。.po ファイルで用いられる詳細な文法と、様々な言語における選択公式については GNU gettext ドキュメント を参照してください。

gettext.**dngettext**(*domain*, *singular*, *plural*, *n*)

ngettext() と同様ですが、指定された *domain* からメッセージを探します。

gettext.**pgettext**(*context*, *message*)

gettext.**dpgettext**(*domain*, *context*, *message*)

gettext.**npgettext**(*context*, *singular*, *plural*, *n*)

gettext.**dnpgettext**(*domain*, *context*, *singular*, *plural*, *n*)

Similar to the corresponding functions without the p in the prefix (that is, *gettext()*, *dgettext()*, *ngettext()*, *dngettext()*), but the translation is restricted to the given message *context*.

[*1] 標準でロケールが収められているディレクトリはシステム依存です; 例えば、RedHat Linux では /usr/share/locale ですが、Solaris では /usr/lib/locale です。*gettext* モジュールはこうしたシステム依存の標準設定をサポートしません; その代わりに *sys.base_prefix*/share/locale (*sys.base_prefix* を参照してください) を標準の設定とします。この理由から、常にアプリケーションの開始時に絶対パスで明示的に指定して *bindtextdomain()* を呼び出すのが最良のやり方ということになります。

バージョン 3.8 で追加.

gettext.**lgettext**(*message*)

gettext.**ldgettext**(*domain*, *message*)

gettext.**lngettext**(*singular*, *plural*, *n*)

gettext.**ldngettext**(*domain*, *singular*, *plural*, *n*)

それぞれに対応する先頭の `l` が無い関数 (*gettext()*, *dgettext()*, *ngettext()*, *dngettext()*) と同じですが、エンコーディングが *bind_textdomain_codeset()* を使って明示的に設定されていない場合、翻訳結果は優先システムエンコーディングでエンコードされたバイト文字列として返されます。

> **警告:** これらの関数はエンコードされたバイト列を返すため Python 3 で使うのは避けるべきです。ほとんどの Python アプリケーションでは、人間が読むテキストをバイト列ではなく文字列として扱いたいので、Unicode 文字列を返す代わりの関数を使う方が良いです。さらに言うと、翻訳文字列にエンコーディング上の問題があった場合、Unicode 関連の予期しない例外を受け取るかもしれません。

Deprecated since version 3.8, will be removed in version 3.10.

GNU **gettext** では `dcgettext()` も定義していますが、このメソッドはあまり有用ではないと思われるので、現在のところ実装されていません。

以下にこの API の典型的な使用法を示します:

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print(_('This is a translatable string.'))
```

23.1.2 クラス形式の API

gettext モジュールのクラス形式の API は、GNU **gettext** API よりも高い柔軟性と利便性を持っています。Python のアプリケーションやモジュールを地域化するにはこちらを使うことをお勧めします。`gettext` には、GNU `.mo` 形式のファイルを構文解析する処理の実装と、文字列を返すメソッドを持つ *GNUTranslations* クラスが定義されています。このクラスのインスタンスも、自分自身を組み込み名前空間に関数 `_()` として配置できます。

gettext.**find**(*domain*, *localedir=None*, *languages=None*, *all=False*)

この関数は標準的な `.mo` ファイル検索アルゴリズムを実装しています。*textdomain()* と同じく、*domain* を引数にとります。オプションの *localedir* は *bindtextdomain()* と同じです。またオプションの *languages* は文字列を列挙したリストで、各文字列は言語コードを表します。

localedir が与えられていない場合、標準のシステムロケールディレクトリが使われます。[*2] *languages* が与えられなかった場合、以下の環境変数: LANGUAGE 、LC_ALL 、LC_MESSAGES 、および LANG が検索されます。空でない値を返した最初の候補が *languages* 変数として使われます。この環境変数は言語名をコロンで分かち書きしたリストを含んでいなければなりません。*find()* はこの文字列をコロンで分割し、言語コードの候補リストを生成します。

find() は次に言語コードを展開および正規化し、リストの各要素について、以下のパス構成:

localedir/*language*/LC_MESSAGES/*domain*.mo

からなる実在するファイルの探索を反復的に行います。*find()* は上記のような実在するファイルで最初に見つかったものを返します。該当するファイルが見つからなかった場合、None が返されます。*all* が与えられていれば、全ファイル名のリストが言語リストまたは環境変数で指定されている順番に並べられたものを返します。

gettext.translation(*domain*, *localedir=None*, *languages=None*, *class_=None*, *fallback=False*, *codeset=None*)

domain 、*localedir* 、*languages* に基づいて *Translations インスタンスを返します。*domain* 、*localedir* 、*languages* はまず *find()* に渡され、関連付けられている .mo ファイルパスのリストを取得します。同一の .mo ファイル名を持つインスタンスはキャッシュされます。実際にインスタンス化されるクラスは、*class_* が与えられていた場合はそのクラスで、そうでない場合には *GNUTranslations* です。クラスのコンストラクタは単一の引数として *file object* を取らなければなりません。*codeset* が与えられた場合、*lgettext()* メソッドおよび *lngettext()* メソッドで翻訳文字列のエンコードに使う文字集合を変更します。

複数の .mo ファイルがあった場合、後ろのファイルは前のファイルのフォールバックとして利用されます。フォールバックの設定のために、*copy.copy()* を使いキャッシュから翻訳オブジェクトを複製します; こうすることで、実際のインスタンスデータはキャッシュのものと共有されたままになります。

.mo ファイルが見つからなかった場合、*fallback* が偽 (デフォルト値) ならこの関数は *OSError* を送出し、*fallback* が真なら *NullTranslations* インスタンスが返されます。

バージョン 3.3 で変更: 以前は *OSError* ではなく *IOError* が送出されていました。

Deprecated since version 3.8, will be removed in version 3.10: *codeset* 引数。

gettext.install(*domain*, *localedir=None*, *codeset=None*, *names=None*)

translation() に *domain* 、*localedir* 、および *codeset* を渡してできる関数 _() を Python の組み込み名前空間に組み込みます。

names パラメータについては、翻訳オブジェクトの *install()* メソッドの説明を参照ください。

以下に示すように、通常はアプリケーション中の文字列を関数 _() の呼び出しで包み込んで翻訳対象候補であることを示します:

```
print(_('This string will be translated.'))
```

[*2] 上の *bindtextdomain()* に関する脚注を参照してください。

利便性を高めるためには、_() 関数を Python の組み込み名前空間に組み入れる必要があります。こうすることで、アプリケーション内の全てのモジュールからアクセスできるようになります。

Deprecated since version 3.8, will be removed in version 3.10: *codeset* 引数。

NullTranslations クラス

翻訳クラスは、元のソースファイル中のメッセージ文字列から翻訳されたメッセージ文字列への変換処理が実際に実装されているクラスです。全ての翻訳クラスで基底クラスとして使われているクラスが *NullTranslations* です; このクラスは、独自の翻訳クラスを実装するのに使える基本的なインタフェースを提供しています。以下に NullTranslations のメソッドを示します:

class gettext.NullTranslations(*fp=None*)

オプションの ファイルオブジェクト *fp* を取ります。この引数は基底クラスでは無視されます。このメソッドは ” 保護された (protected)” インスタンス変数 *_info* および *_charset* を初期化します。これらの変数の値は派生クラスで設定することができます。同様に *_fallback* も初期化しますが、この値は *add_fallback()* で設定されます。その後、*fp* が None でない場合 self._parse(fp) を呼び出します。

_parse(*fp*)

基底クラスでは何もしない (no-op) ようになっています。このメソッドの役割はファイルオブジェクト *fp* を引数に取り、ファイルからデータを読み出し、メッセージカタログを初期化することです。サポートされていないメッセージカタログ形式を使っている場合、その形式を解釈するためにはこのメソッドを上書きしなくてはなりません。

add_fallback(*fallback*)

fallback を現在の翻訳オブジェクトの代替オブジェクトとして追加します。翻訳オブジェクトが与えられたメッセージに対して翻訳メッセージを提供できない場合、この代替オブジェクトに問い合わせることになります。

gettext(*message*)

フォールバックが設定されている場合、フォールバックの gettext() に処理を移譲します。そうでない場合、引数として受け取った *message* を返します。派生クラスで上書きするメソッドです。

ngettext(*singular*, *plural*, *n*)

フォールバックが設定されている場合、フォールバックの ngettext() に処理を移譲します。そうでない場合、*n* が 1 なら *singular* を返します; それ以外なら *plural* を返します。派生クラスで上書きするメソッドです。

pgettext(*context*, *message*)

代替オブジェクトが設定されている場合、*pgettext()* を代替オブジェクトに転送します。そうでない場合、翻訳されたメッセージを返します。派生クラスで上書きするメソッドです。

バージョン 3.8 で追加.

npgettext(*context*, *singular*, *plural*, *n*)

代替オブジェクトが設定されている場合、*npgettext()* を代替オブジェクトに転送します。そうでない場合、翻訳されたメッセージを返します。派生クラスで上書きするメソッドです。

バージョン 3.8 で追加.

lgettext(*message*)

lngettext(*singular*, *plural*, *n*)

gettext() および *ngettext()* と同じですが、エンコーディングが *set_output_charset()* で明示的に設定されていない場合、翻訳結果は優先システムエンコーディングでエンコードされたバイト文字列として返されます。派生クラスで上書きするメソッドです。

> **警告:** これらのメソッドは Python 3 で使うのは避けるべきです。*lgettext()* 関数に対する警告を参照してください。

Deprecated since version 3.8, will be removed in version 3.10.

info()

Return the "protected" `_info` variable, a dictionary containing the metadata found in the message catalog file.

charset()

メッセージカタログファイルのエンコーディングを返します。

output_charset()

lgettext() と *lngettext()* の返り値となる翻訳メッセージで使われているエンコーディングを返します。

Deprecated since version 3.8, will be removed in version 3.10.

set_output_charset(*charset*)

返り値の翻訳メッセージで使われるエンコーディングを変更します。

Deprecated since version 3.8, will be removed in version 3.10.

install(*names=None*)

このメソッドは *gettext()* を組み込み名前空間にインストールし、変数 `_` に束縛します。

names パラメータを与える場合には、`_()` 以外では組み込み名前空間に配置したい関数名を列挙したシーケンスでなければなりません。サポートされている名前は `'gettext'`, `'ngettext'`, `'pgettext'`, `'npgettext'`, `'lgettext'`, `'lngettext'` です。

この方法はアプリケーションで `_()` 関数を利用できるようにするための最も便利な方法ですが、唯一の手段でもあるので注意してください。この関数はアプリケーション全体、とりわけ組み込み名前空間に影響するので、地域化されたモジュールで `_()` を組み入れることができないのです。その代わりに、以下のコードを使って `_()` を使えるようにしなければなりません。:

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

この操作は `_()` をモジュール内だけのグローバル名前空間に組み入れるので、モジュール内の `_()` の呼び出しだけに影響します。

バージョン 3.8 で変更: `'pgettext'` と `'npgettext'` が追加されました。

GNUTranslations クラス

gettext モジュールでは *NullTranslations* から派生したもう一つのクラス: *GNUTranslations* を提供しています。このクラスはビッグエンディアン、およびリトルエンディアン両方のバイナリ形式の GNU **gettext** `.mo` ファイルを読み出せるように `_parse()` を上書きしています。

GNUTranslations はまた、翻訳カタログ以外に、オプションのメタデータを読み込んで解釈します。GNU **gettext** では、空の文字列に対する変換先としてメタデータを取り込むことが慣習になっています。このメタデータは **RFC 822** 形式の `key: value` のペアになっており、`Project-Id-Version` キーを含んでいなければなりません。キー `Content-Type` があった場合、`charset` の特性値 (property) は ” 保護された” `_charset` インスタンス変数を初期化するために用いられます。値がない場合には、デフォルトとして `None` が使われます。エンコードに用いられる文字セットが指定されている場合、カタログから読み出された全てのメッセージ id とメッセージ文字列は、指定されたエンコードを用いて Unicode に変換され、そうでなければ ASCII とみなされます。

メッセージ id もユニコード文字列として解釈されるので、すべての `*gettext()` メソッドはメッセージ id をバイト文字列ではなくユニコード文字列と仮定するでしょう。

key/value ペアの集合全体は辞書型データ中に配置され、” 保護された” `_info` インスタンス変数に設定されます。

`.mo` ファイルのマジックナンバーが不正な場合や、メジャーバージョン番号が予期されないものの場合、あるいはその他の問題がファイルの読み出し中に発生した場合、*GNUTranslations* クラスのインスタンス化で *OSError* が送出されることがあります。

class gettext.GNUTranslations

以下のメソッドは基底クラスの実装からオーバライドされています:

gettext(*message*)

カタログから *message* id を検索して、対応するメッセージ文字列を Unicode でエンコードして返します。*message* id に対応するエントリがカタログに存在せず、フォールバックが設定されている場合、検索処理をフォールバックの *gettext()* メソッドに移譲します。それ以外の場合は、*message* id 自体が返されます。

ngettext(*singular*, *plural*, *n*)

メッセージ id に対する複数形を検索します。カタログに対する検索では *singular* がメッセージ id として用いられ、*n* にはどの複数形を用いるかを指定します。返されるメッセージ文字列は Unicode 文字列です。

メッセージ id がカタログ中に見つからず、フォールバックが指定されている場合は、メッセージ検索要求はフォールバックの *ngettext()* メソッドに移譲されます。それ以外の場合、*n* が 1 ならば *singular* が返され、それ以外なら *plural* が返されます。

以下に例を示します。:

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ngettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

pgettext(*context*, *message*)

カタログから *context* と *message* id を検索して、対応するメッセージ文字列を、Unicode でエンコードして返します。*message* id と *context* に対するエントリがカタログに存在せず、フォールバックが設定されている場合、フォールバック検索はオブジェクトの *pgettext()* メソッドに転送されます。そうでない場合、*message* id 自体が返されます。

バージョン 3.8 で追加.

npgettext(*context*, *singular*, *plural*, *n*)

メッセージ id に対する複数形を検索します。カタログに対する検索では *singular* がメッセージ id として用いられ、*n* にはどの複数形を用いるかを指定します。

context に対するメッセージ id がカタログ中に見つからず、フォールバックオブジェクトが指定されている場合、メッセージ検索要求はフォールバックオブジェクトの *npgettext()* メソッドに転送されます。そうでない場合、*n* が 1 ならば *singular* が返され、それ以外に対しては *plural* が返されます。

バージョン 3.8 で追加.

lgettext(*message*)

lngettext(*singular*, *plural*, *n*)

gettext() および *ngettext()* と同じですが、エンコーディングが *set_output_charset()* で明示的に設定されていない場合、翻訳結果は優先システムエンコーディングでエンコードされたバイト文字列として返されます。

> **警告:** これらのメソッドは Python 3 で使うのは避けるべきです。*lgettext()* 関数に対する警告を参照してください。

Deprecated since version 3.8, will be removed in version 3.10.

Solaris メッセージカタログ機構のサポート

Solaris オペレーティングシステムでは、独自の .mo バイナリファイル形式を定義していますが、この形式に関するドキュメントが手に入らないため、現時点ではサポートされていません。

Catalog コンストラクタ

GNOME では、James Henstridge によるあるバージョンの *gettext* モジュールを使っていますが、このバージョンは少し異なった API を持っています。ドキュメントに書かれている利用法は:

```
import gettext
cat = gettext.Catalog(domain, localedir)
_ = cat.gettext
print(_('hello world'))
```

となっています。過去のモジュールとの互換性のために、Catalog() は前述の *translation()* 関数の別名になっています。

このモジュールと Henstridge のバージョンとの間には一つ相違点があります: 彼のカタログオブジェクトはマップ型の API を介したアクセスがサポートされていましたが、この API は使われていないらしく、現在はサポートされていません。

23.1.3 プログラムやモジュールを国際化する

国際化 (I18N, I-nternationalizatio-N) とは、プログラムを複数の言語に対応させる操作を指します。地域化 (L10N, L-ocalizatio-N) とは、すでに国際化されているプログラムを特定地域の言語や文化的な事情に対応させることを指します。Python プログラムに多言語メッセージ機能を追加するには、以下の手順を踏む必要があります:

1. プログラムやモジュールで翻訳対象とする文字列に特殊なマークをつけて準備します
2. マークづけをしたファイルに一連のツールを走らせ、生のメッセージカタログを生成します
3. 特定の言語へのメッセージカタログの翻訳を作成します
4. メッセージ文字列を適切に変換するために *gettext* モジュールを使います

ソースコードを I18N 化する準備として、ファイル内の全ての文字列を探す必要があります。翻訳を行う必要のある文字列はどれも _('...') --- すなわち関数 _() の呼び出しで包むことでマーク付けしなくてはなりません。例えば以下のようにします:

```
filename = 'mylog.txt'
message = _('writing a log message')
with open(filename, 'w') as fp:
    fp.write(message)
```

この例では、文字列 'writing a log message' が翻訳対象候補としてマーク付けされており、文字列 'mylog.txt' および 'w' はされていません。

翻訳対象の文字列を抽出するツールもあります。オリジナルの GNU **gettext** は C と C++ のソースコードしかサポートしませんが、拡張版の **xgettext** は Python を含めた多くの言語で書かれたコードを読み取り、翻訳できる文字列を発見します。Babel は Python の国際化ライブラリで、翻訳文字列の抽出とメッセージカタログのコンパイルを行う file:*pybabel* スクリプトがあります。François Pinard が開発した **xpot** と呼ばれるプログラムは同じような処理を行え、彼の po-utils package の一部として利用可能です。

(Python には **pygettext.py** および **msgfmt.py** という名前の pure-Python 版プログラムもあります; これをインストールしてくれる Python ディストリビューションもあります。**pygettext.py** は **xgettext** に似たプログラムですが Python のソースコードしか理解できず、C や C++ のような他のプログラミング言語を扱えません。**pygettext.py** は **xgettext** と同様のコマンドラインインターフェースをサポートしています; 詳しい使い方については `pygettext.py --help` と実行してください。**msgfmt.py** は GNU **msgfmt** とバイナリ互換性があります。この 2 つのプログラムがあれば、GNU **gettext** パッケージを使わずに Python アプリケーションを国際化できるでしょう。)

xgettext や **pygettext** のようなツールは、メッセージカタログである `.po` ファイルを生成します。このファイルは人間が判読可能な構造をしていて、ソースコード中のマークが着けられた文字列と、その文字列の仮置きの訳文が一緒に書き込まれています。

生成された `.po` ファイルは翻訳者個々人へ頒布され、サポート対象の各自然言語への訳文が書き込まれます。ある言語への翻訳が完了した `<language-name>.po` ファイルは翻訳者により返送され、**msgfmt** を使い機械が読み込みやすい `.mo` バイナリカタログファイルへとコンパイルされます。この `.mo` が *gettext* モジュールによる実行時の実際の翻訳処理で使われます。

gettext モジュールをソースコード中でどのように使うかは単一のモジュールを国際化するのか、それともアプリケーション全体を国際化するのかによります。次のふたつのセクションで、それぞれについて説明します。

モジュールを地域化する

モジュールを地域化する場合、グローバルな変更、例えば組み込み名前空間への変更を行わないように注意しなければなりません。GNU **gettext** API ではなく、クラス形式の API を使うべきです。

仮に対象のモジュール名を "spam" とし、モジュールの各言語における翻訳が収められた `.mo` ファイルが `/usr/share/locale` に GNU **gettext** 形式で置かれているとします。この場合、モジュールの最初で以下のようにします:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.gettext
```

アプリケーションを地域化する

アプリケーションを地域化するのなら、関数 `_()` をグローバルな組み込み名前空間に組み入れなければならず、これは通常アプリケーションの主ドライバ (main driver) ファイルで行います。この操作によって、アプリケーション独自のファイルは明示的に各ファイルで `_()` の組み入れを行わなくても単に `_('...')` を使うだけで済むようになります。

単純な場合では、単に以下の短いコードをアプリケーションの主ドライバファイルに追加するだけです:

```
import gettext
gettext.install('myapplication')
```

ロケールの辞書を設定する必要がある場合、*install()* 関数に渡すことが出来ます:

```
import gettext
gettext.install('myapplication', '/usr/share/locale')
```

動作中 (on the fly) に言語を切り替える

多くの言語を同時にサポートする必要がある場合、複数の翻訳インスタンスを生成して、例えば以下のコードのように、インスタンスを明示的に切り替えてもかまいません。:

```
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

翻訳処理の遅延解決

コードを書く上では、ほとんどの状況で文字列はコードされた場所で翻訳されます。しかし場合によっては、翻訳対象として文字列をマークはするが、その後実際に翻訳が行われるように遅延させる必要が生じます。古典的な例は以下のようなコートです:

```
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python', ]
```

(次のページに続く)

(前のページからの続き)

```
# ...
for a in animals:
    print(a)
```

ここで、リスト animals 内の文字列は翻訳対象としてマークはしたいが、文字列が出力されるまで実際に翻訳を行うのは避けたいとします。

こうした状況を処理する一つの方法を以下に示します:

```
def _(message): return message

animals = [_('mollusk'),
           _('albatross'),
           _('rat'),
           _('penguin'),
           _('python'), ]

del _

# ...
for a in animals:
    print(_(a))
```

ダミーの _() 定義が単に文字列をそのまま返すようになっているので、上のコードはうまく動作します。かつ、このダミーの定義は、組み込み名前空間に置かれた _() の定義で (del 命令を実行するまで) 一時的に上書きすることができます。もしそれまでに _() をローカルな名前空間に持っていたら注意してください。

二つ目の例における _() の使い方では、パラメータが文字列リテラルではないので、**gettext** プログラムが翻訳可能だとは判定されないことに注意してください。

もう一つの処理法は、以下の例のようなやり方です:

```
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print(_(a))
```

この例では、翻訳可能な文字列に N_() でマークを付けているために、_() の定義と衝突しません。しかし、これではメッセージを抽出するプログラムに対して N_() でマークされている翻訳可能な文字列を見付けるように教える必要が出てきます。**xgettext**, **pygettext**, pybabel extract, **xpot** は全て、コマンドラインスイッチ -k を使ってその機能をサポートしています。この例の N_() という名前は好きに選べます; MarkThisStringForTranslation() という名前にしてしまっても構いません。

23.1.4 謝辞

以下の人々が、このモジュールのコード、フィードバック、設計に関する助言、過去の実装、そして有益な経験談による貢献をしてくれました:

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard
- Barry Warsaw
- Gustavo Niemeyer

脚注

23.2 locale --- 国際化サービス

ソースコード: Lib/locale.py

---

locale モジュールは POSIX ロケールデータベースおよびロケール関連機能へのアクセスを提供します。POSIX ロケール機構を使うことで、プログラマはソフトウェアが実行される各国における詳細を知らなくても、アプリケーション上で特定の地域文化に関係する部分を扱うことができます。

locale モジュールは、_locale を被うように実装されており、ANSI C ロケール実装を使っている _locale が利用可能なら、こちらを先に使うようになっています。

locale モジュールでは以下の例外と関数を定義しています:

exception locale.Error

setlocale() に渡されたロケールが認識されない場合例外が送出されます。

locale.setlocale(*category*, *locale=None*)

locale が渡され None でない場合、*setlocale()* は *category* のロケール設定を変更します。利用可能なカテゴリーは下記の表を参照してください。*locale* は文字列か 2 つの文字列の iterable (言語コードと文字コード) です。iterale の場合 locale aliasing engine を用いてロケール名に変換されます。空の文字列はユーザのデフォルトの設定を指定します。ロケールの変更に失敗した場合 *Error* が送出されます。成功した場合新しいロケールの設定が返されます。

locale が省略されたり None の場合、*category* の現在の設定が返されます。

setlocale() はほとんどのシステムでスレッド安全ではありません。アプリケーションを書くとき、大抵は以下のコード

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

から書き始めます。これは全てのカテゴリをユーザの環境における標準設定 (大抵は環境変数 LANG で指定されています) に設定します。その後複数スレッドを使ってロケールを変更したりしない限り、問題は起こらないはずです。

locale.localeconv()

地域的な慣行のデータベースを辞書として返します。辞書は以下の文字列をキーとして持っています:

カテゴリ	キー	意味
LC_NUMERIC	'decimal_point'	小数点を表す文字です。
	'grouping'	'thousands_sep' が来るかもしれない場所を相対的に表した数からなる配列です。配列が *CHAR_MAX* で終端されている場合、それ以上の桁では桁数字のグループ化を行いません。配列が 0 で終端されている場合、最後に指定したグループが反復的に使われます。
	'thousands_sep'	桁グループ間を区切るために使われる文字です。
LC_MONETARY	'int_curr_symbol'	国際通貨を表現する記号です。
	'currency_symbol'	地域的な通貨を表現する記号です。
	'p_cs_precedes/n_cs_precedes'	通貨記号が値の前につくかどうかです (それぞれ正の値、負の値を表します)。
	'p_sep_by_space/n_sep_by_space'	通貨記号と値との間にスペースを入れるかどうかです (それぞれ正の値、負の値を表します)。
	'mon_decimal_point'	金額表示の際に使われる小数点です。
	'frac_digits'	金額を地域的な方法で表現する際の小数点以下の桁数です。
	'int_frac_digits'	金額を国際的な方法で表現する際の小数点以下の桁数です。
	'mon_thousands_sep'	金額表示の際に桁区切り記号です。
	'mon_grouping'	'grouping' と同じで、金額表示の際に使われます。
	'positive_sign'	正の値の金額表示に使われる記号です。
	'negative_sign'	負の値の金額表示に使われる記号です。
	'p_sign_posn/n_sign_posn'	符号の位置です (それぞれ正の値と負の値を表します)。以下を参照してください。

数値形式の値に *CHAR_MAX* を設定すると、そのロケールでは値が指定されていないことを表します。

'p_sign_posn' および 'n_sign_posn' の取り得る値は以下の通りです。

値	説明
0	通貨記号および値は丸括弧で囲われます。
1	符号は値と通貨記号より前に来ます。
2	符号は値と通貨記号の後に続きます。
3	符号は値の直前に来ます。
4	符号は値の直後に来ます。
CHAR_MAX	このロケールでは特に指定しません。

LC_NUMERIC ロケールや LC_MONETARY ロケールが LC_CTYPE ロケールと異なっていて、数値文字列や通貨文字列が非 ASCII 文字列の場合、この関数は一時的に LC_NUMERIC ロケールや LC_MONETARY ロケールに LC_CTYPE ロケールを設定します。この一時的な変更は他のスレッドに影響を及ぼします。

バージョン 3.7 で変更: この関数は、一時的に LC_NUMERIC ロケールに LC_CTYPE ロケールを設定する場合があります。

locale.**nl_langinfo**(*option*)

ロケール特有の情報を文字列として返します。この関数は全てのシステムで利用可能なわけではなく、指定できる *option* もプラットフォーム間で大きく異なります。引数として使えるのは、locale モジュールで利用可能なシンボル定数を表す数字です。

関数 *nl_langinfo()* は以下のキーのうち一つを受理します。ほとんどの記述は GNU C ライブラリ中の対応する説明から引用されています。

locale.**CODESET**

選択されたロケールで用いられている文字エンコーディングの名前を文字列で取得します。

locale.**D_T_FMT**

日付と時刻をロケール特有の方法で表現するために、*time.strftime()* の書式文字列として用いることのできる文字列を取得します。

locale.**D_FMT**

日付をロケール特有の方法で表現するために、*time.strftime()* の書式文字列として用いることのできる文字列を取得します。

locale.**T_FMT**

時刻をロケール特有の方法で表現するために、*time.strftime()* の書式文字列として用いることのできる文字列を取得します。

locale.**T_FMT_AMPM**

時刻を午前／午後の書式で表現するために、*time.strftime()* の書式文字列として用いることのできる文字列を取得します。

DAY_1 ... DAY_7

1 週間中の n 番目の曜日名を取得します。

注釈: ロケール US における、DAY_1 を日曜日とする慣行に従っています。国際的な (ISO 8601) 月曜日を週の初めとする慣行ではありません。

ABDAY_1 ... ABDAY_7
1 週間中の n 番目の曜日名を略式表記で取得します。

MON_1 ... MON_12
n 番目の月の名前を取得します。

ABMON_1 ... ABMON_12
n 番目の月の名前を略式表記で取得します。

locale.RADIXCHAR
基数点 (小数点ドット、あるいは小数点コンマ、等) を取得します。

locale.THOUSEP
1000 単位桁区切り (3 桁ごとのグループ化) の区切り文字を取得します。

locale.YESEXPR
肯定／否定で答える質問に対する肯定回答を正規表現関数で認識するために利用できる正規表現を取得します。

注釈: 表現は C ライブラリの regex() 関数に合ったものでなければならず、これは *re* で使われている構文とは異なるかもしれません。

locale.NOEXPR
肯定／否定で答える質問に対する否定回答を正規表現関数で認識するために利用できる正規表現を取得します。

locale.CRNCYSTR
通貨シンボルを取得します。シンボルを値の前に表示させる場合には ”-”、値の後ろに表示させる場合には ”+”、シンボルを基数点と置き換える場合には ”.” を前につけます。

locale.ERA
現在のロケールで使われている年代を表現する値を取得します。

ほとんどのロケールではこの値を定義していません。この値を設定しているロケールの例は Japanese です。日本には日付の伝統的な表示法として、時の天皇に対応する元号名があります。

通常この値を直接指定する必要はありません。E を書式文字列に指定することで、関数 *time.strftime()* がこの情報を使うようになります。返される文字列の様式は決められていないので、異なるシステム間で様式に関する同じ知識が使えると期待してはいけません。

locale.ERA_D_T_FMT
日付および時間をロケール固有の年代に基づいた方法で表現するために、*time.strftime()* の書式文字列として用いることのできる文字列を取得します。

locale.**ERA_D_FMT**
日付をロケール固有の年代に基づいた方法で表現するために、*time.strftime()* の書式文字列として用いることのできる文字列を取得します。

locale.**ERA_T_FMT**
時刻をロケール固有の年代に基づいた方法で表現するために、*time.strftime()* の書式文字列として用いることのできる文字列を取得します。

locale.**ALT_DIGITS**
返される値は 0 から 99 までの 100 個の値の表現です。

locale.**getdefaultlocale**([*envvars*])
標準のロケール設定を取得しようと試み、結果をタプル (language code, encoding) の形式で返します。

POSIX によると、setlocale(LC_ALL, '') を呼ばなかったプログラムは、移植可能な 'C' ロケール設定を使います。setlocale(LC_ALL, '') を呼ぶことで、LANG 変数で定義された標準のロケール設定を使うようになります。Python では現在のロケール設定に干渉したくないので、上で述べたような方法でその挙動をエミュレーションしています。

他のプラットフォームとの互換性を維持するために、環境変数 LANG だけでなく、引数 *envvars* で指定された環境変数のリストも調べられます。最初に見つかった定義が使われます。*envvars* は標準では GNU gettext で使われている検索順になります; これは常に変数名 LANG を探します。GNU gettext では 'LC_ALL' 、'LC_CTYPE' 、'LANG' 、および 'LANGUAGE' の順に調べられます。

'C' の場合を除き、言語コードは **RFC 1766** に対応します。*language code* および *encoding* が決定できなかった場合、None になるかもしれません。

locale.**getlocale**(*category=LC_CTYPE*)
与えられたロケールカテゴリに対する現在の設定を、*language code* 、*encoding* を含むシーケンスで返します。*category* として *LC_ALL* 以外の LC_* の値の一つを指定できます。標準の設定は *LC_CTYPE* です。

'C' の場合を除き、言語コードは **RFC 1766** に対応します。*language code* および *encoding* が決定できなかった場合、None になるかもしれません。

locale.**getpreferredencoding**(*do_setlocale=True*)
テキストデータをエンコードする方法を、ユーザの設定に基づいて返します。ユーザの設定は異なるシステム間では異なった方法で表現され、システムによってはプログラミング的に得ることができないこともあるので、この関数が返すのはただの推測です。

システムによっては、ユーザの設定を取得するために *setlocale()* を呼び出す必要があるため、この関数はスレッド安全ではありません。*setlocale()* を呼び出す必要がない、または呼び出したくない場合、*do_setlocale* を False に設定する必要があります。

Android 上あるいは UTF-8 モード (-X utf8 オプション) では、常に 'UTF-8' が返され、ロケールや *do_setlocale* 引数は無視されます。

バージョン 3.7 で変更: この関数は、Android 上あるいは UTF-8 モードが有効になっている場合、常

に UTF-8 を返すようになりました。

locale.**normalize**(*localename*)
与えたロケール名を規格化したロケールコードを返します。返されるロケールコードは *setlocale()* で使うために書式化されています。規格化が失敗した場合、もとの名前がそのまま返されます。

与えたエンコードがシステムにとって未知の場合、標準の設定では、この関数は *setlocale()* と同様に、エンコーディングをロケールコードにおける標準のエンコーディングに設定します。

locale.**resetlocale**(*category=LC_ALL*)
category のロケールを標準設定にします。

標準設定は *getdefaultlocale()* を呼ぶことで決定されます。*category* は標準で *LC_ALL* になっています。

locale.**strcoll**(*string1*, *string2*)
現在の *LC_COLLATE* 設定に従って二つの文字列を比較します。他の比較を行う関数と同じように、*string1* が *string2* に対して前に来るか、後に来るか、あるいは二つが等しいかによって、それぞれ負の値、正の値、あるいは 0 を返します。

locale.**strxfrm**(*string*)
文字列を、ロケールを考慮した比較に使える形式に変換します。例えば、`strxfrm(s1) < strxfrm(s2)` は `strcoll(s1, s2) < 0` と等価です。この関数は同じ文字列が何度も比較される場合、例えば文字列からなるシーケンスを順序付けて並べる際に使うことができます。

locale.**format_string**(*format*, *val*, *grouping=False*, *monetary=False*)
数値 *val* を現在の *LC_NUMERIC* の設定に基づいて書式化します。書式は % 演算子の慣行に従います。浮動小数点数については、必要に応じて浮動小数点が変更されます。*grouping* が真なら、ロケールに配慮した桁数の区切りが行われます。

monetary が真なら、桁区切り記号やグループ化文字列を用いて変換を行います。

`format % val` 形式のフォーマット指定子を、現在のロケール設定を考慮したうえで処理します。

バージョン 3.7 で変更: *monetary* キーワード引数が追加されました。

locale.**format**(*format*, *val*, *grouping=False*, *monetary=False*)
この関数は *format_string()* に似た動作をしますが、1 つの `%char` 指定子しかない場合のみ動作します。例えば、`'%f'` や `'%.0f'` はどちらも有効な指定子ですが、`'%f KiB'` は有効ではありません。

文字列全体をフォーマットするには、*format_string()* を使ってください。

バージョン 3.7 で非推奨: 代わりに *format_string()* を使ってください。

locale.**currency**(*val*, *symbol=True*, *grouping=False*, *international=False*)
数値 *val* を、現在の *LC_MONETARY* の設定にあわせてフォーマットします。

symbol が真の場合は、返される文字列に通貨記号が含まれるようになります。これはデフォルトの設定です。*grouping* が真の場合 (これはデフォルトではありません) は、値をグループ化します。*international* が真の場合 (これはデフォルトではありません) は、国際的な通貨記号を使用します。

この関数は'C' ロケールでは動作しないことに注意しましょう。まず最初に *setlocale()* でロケールを設定する必要があります。

locale.**str**(*float*)
浮動小数点数を str(float) と同じように書式化しますが、ロケールに配慮した小数点が使われます。

locale.**delocalize**(*string*)
文字列を *LC_NUMERIC* で設定された慣行に従って正規化された数値文字列に変換します。

バージョン 3.5 で追加.

locale.**atof**(*string*)
文字列を *LC_NUMERIC* で設定された慣行に従って浮動小数点に変換します。

locale.**atoi**(*string*)
文字列を *LC_NUMERIC* で設定された慣行に従って整数に変換します。

locale.**LC_CTYPE**
文字タイプ関連の関数のためのロケールカテゴリです。このカテゴリの設定に従って、モジュール *string* における関数の振る舞いが変わります。

locale.**LC_COLLATE**
文字列を並べ替えるためのロケールカテゴリです。*locale* モジュールの関数 *strcoll()* および *strxfrm()* が影響を受けます。

locale.**LC_TIME**
時刻を書式化するためのロケールカテゴリです。*time.strftime()* はこのカテゴリに設定されている慣行に従います。

locale.**LC_MONETARY**
金額に関係する値を書式化するためのロケールカテゴリです。設定可能なオプションは関数 *localeconv()* で得ることができます。

locale.**LC_MESSAGES**
メッセージ表示のためのロケールカテゴリです。現在 Python はアプリケーション毎にロケールに対応したメッセージを出力する機能はサポートしていません。*os.strerror()* が返すような、オペレーティングシステムによって表示されるメッセージはこのカテゴリによって影響を受けます。

locale.**LC_NUMERIC**
数字を書式化するためのロケールカテゴリです。関数 *format()* 、*atoi()* 、*atof()* および *locale* モジュールの *str()* が影響を受けます。他の数値書式化操作は影響を受けません。

locale.**LC_ALL**
全てのロケール設定を総合したものです。ロケールを変更する際にこのフラグが使われた場合、そのロケールにおける全てのカテゴリを設定しようと試みます。一つでも失敗したカテゴリがあった場合、全てのカテゴリにおいて設定変更を行いません。このフラグを使ってロケールを取得した場合、全てのカテゴリにおける設定を示す文字列が返されます。この文字列は、後に設定を元に戻すために使うことができます。

locale.CHAR_MAX

localeconv() の返す特別な値のためのシンボル定数です。

以下はプログラム例です:

```
>>> import locale
>>> loc = locale.getlocale()  # get current locale
# use German locale; name might vary with platform
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
>>> locale.strcoll('f\xe4n', 'foo')  # compare a string containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '')   # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C')  # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc)  # restore saved locale
```

23.2.1 ロケールの背景、詳細、ヒント、助言および補足説明

C 標準では、ロケールはプログラム全体にわたる特性であり、その変更は高価な処理であるとしています。加えて、頻繁にロケールを変更するようなひどい実装はコアダンプを引き起こすこともあります。このことがロケールを正しく利用する上で苦痛となっています。

最初、プログラムが起動したときには、ユーザの希望するロケールにかかわらずロケールは C です。例外が 1 つあります: *LC_CTYPE* カテゴリは、現在のロケールエンコーディングをユーザーの希望するロケールエンコーディングに設定するために、スタートアップで変更されます。他のカテゴリについては、プログラムは setlocale(LC_ALL, '') を呼び出して、明示的にユーザの希望するロケール設定を行わなければなりません。

setlocale() をライブラリルーチン内で呼ぶことは、それがプログラム全体に及ぼす副作用の面から、一般的によくない考えです。ロケールを保存したり復帰したりするのもよくありません: 高価な処理であり、ロケールの設定が復帰する以前に起動してしまった他のスレッドに悪影響を及ぼすからです。

もし、汎用を目的としたモジュールを作っていて、ロケールによって影響をうけるような操作 (例えば *time.strftime()* の書式の一部) のロケール独立のバージョンが必要ということになれば、標準ライブラリルーチンを使わずに何とかしなければなりません。よりましな方法は、ロケール設定が正しく利用できているか確かめることです。最後の手段は、あなたのモジュールが C ロケール以外の設定には互換性がないとドキュメントに書くことです。

ロケールに従って数値操作を行うための唯一の方法はこのモジュールで特別に定義されている関数: *atof()* 、*atoi()* 、*format()* 、*str()* を使うことです。

ロケールに従ってケース変換や文字分類を行う方法はありません。(ユニコード) テキスト文字列については、これらは文字の値のみによって行われます。その一方、バイト文字列はバイトの ASCII 値に従って変換と分類が行われます。そして、上位ビットが立っているバイト (すなわち、non-ASCII バイト) は、決して変換されず、英字や空白などの文字クラスの一部とみなされることもありません。

23.2.2 Python 拡張の作者と、Python を埋め込むようなプログラムに関して

拡張モジュールは、現在のロケールを調べる以外は、決して *setlocale()* を呼び出してはなりません。しかし、返される値もロケールの復帰のために使えるだけなので、さほど便利とはいえません (例外はおそらくロケールが `C` かどうか調べることでしょう)。

ロケールを変更するために Python コードで *locale* モジュールを使った場合、Python を埋め込んでいるアプリケーションにも影響を及ぼします。Python を埋め込んでいるアプリケーションに影響が及ぶことを望まない場合、`config.c` ファイル内の組み込みモジュールのテーブルから `_locale` 拡張モジュール (ここで全てを行っています) を削除し、共有ライブラリから `_locale` モジュールにアクセスできないようにしてください。

23.2.3 メッセージカタログへのアクセス

`locale.`**`gettext`**`(`*msg*`)`

`locale.`**`dgettext`**`(`*domain*, *msg*`)`

`locale.`**`dcgettext`**`(`*domain*, *msg*, *category*`)`

`locale.`**`textdomain`**`(`*domain*`)`

`locale.`**`bindtextdomain`**`(`*domain*, *dir*`)`

C ライブラリの gettext インタフェースが提供されているシステムでは、locake モジュールでそのインタフェースを公開しています。このインタフェースは関数 `gettext()` 、`dgettext()` 、`dcgettext()` 、`textdomain()` 、`bindtextdomain()` 、`bind_textdomain_codeset()` からなります。これらは *gettext* モジュールの同名の関数に似ていますが、メッセージカタログとして C ライブラリのバイナリフォーマットを使い、メッセージカタログを探すために C ライブラリの検索アルゴリズムを使います。

Python アプリケーションでは、通常これらの関数を呼び出す必要はないはずで、代わりに *gettext* を呼ぶべきです。例外として知られているのは、内部で `gettext()` または `dcgettext()` を呼び出すような C ライブラリにリンクするアプリケーションです。こうしたアプリケーションでは、ライブラリが正しいメッセージカタログを探せるようにテキストドメイン名を指定する必要があります。

第

TWENTYFOUR

プログラムのフレームワーク

この章で解説されるモジュールはあなたのプログラムの大枠を規定するフレームワークです。現状では、ここで解説されるモジュールは全てコマンドラインインタフェースを書くためのものです。

この章で解説されるモジュールの完全な一覧は:

24.1 `turtle` --- タートルグラフィックス

ソースコード: Lib/turtle.py

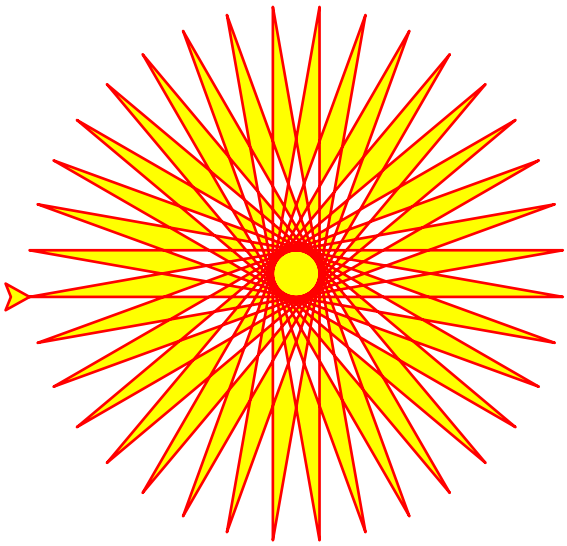
24.1.1 はじめに

タートルグラフィックスは子供向けのプログラミングに入門でよく用いられます。タートルグラフィックスは Wally Feurzeig, Seymour Papert, Cynthia Solomon が 1967 に開発したオリジナルの Logo プログラミング言語の一部分です。

x-y 平面の (0, 0) から動き出すロボット亀を想像してみて下さい。`turtle.forward(15)` という命令を出すと、その亀が (スクリーン上で!) 15 ピクセル分顔を向けている方向に動き、動きに沿って線を引きます。`turtle.left(25)` という命令を出すと、今度はその場で 25 度反時計回りに回ります。

Turtle star

Turtle can draw intricate shapes using programs that repeat simple moves.



```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

これらの命令と他の同様な命令を組み合わせることで、複雑な形や絵が簡単に描けます。

turtle モジュールは同じ名前を持った Python 2.5 までのモジュールの拡張された再実装です。

再実装に際しては古い turtle モジュールのメリットをそのままに、(ほぼ) 100% 互換性を保つようにしました。すなわち、まず第一に、学習中のプログラマがモジュールを `-n` スイッチを付けて走らせている IDLE の中から全てのコマンド、クラス、メソッドを対話的に使えるようにしました。

turtle モジュールはオブジェクト指向と手続き指向の両方の方法でタートルグラフィックス・プリミティブを提供します。グラフィックスの基礎として *tkinter* を使っているために、Tk をサポートした Python のバージョンが必要です。

オブジェクト指向インターフェイスでは、本質的に 2+2 のクラスを使います:

1. *TurtleScreen* クラスはタートルが絵を描きながら走り回る画面を定義します。そのコンストラクタには `tkinter.Canvas` または *ScrolledCanvas* を渡す必要があります。*turtle* をアプリケーションの一部として用いたい場合にはこれを使うべきです。

   Screen() 関数は *TurtleScreen* のサブクラスのシングルトンオブジェクトを返します。*turtle* をグラフィクスを使う一つの独立したツールとして使う場合には、この関数を呼び出すべきです。シングルトンなので、そのクラスからの継承はできません。

TurtleScreen/Screen の全てのメソッドは関数としても、すなわち、手続き指向インターフェイスの一部としても存在しています。

2. *RawTurtle* (別名: *RawPen*) は *TurtleScreen* 上に絵を描く Turtle オブジェクトを定義します。コンストラクタには Canvas, ScrolledCanvas, TurtleScreen のいずれかを引数として渡して RawTurtle オブジェクトがどこに絵を描くかを教えます。

   RawTurtle の派生はサブクラス *Turtle* (別名: `Pen`) で、(既に与えられているのでなければ自動的に作られた) " 唯一の" *Screen* インスタンスに絵を描きます。

   RawTurtle/Turtle の全てのメソッドは関数としても、すなわち、手続き指向インターフェイスの一部としても存在しています。

手続き型インターフェイスでは *Screen* および *Turtle* クラスのメソッドを元にした関数を提供しています。その名前は対応するメソッドと一緒です。Screen のメソッドを元にした関数が呼び出されるといつでも screen オブジェクトが自動的に作られます。Turtle のメソッドを元にした関数が呼び出されるといつでも (名無しの) turtle オブジェクトが自動的に作られます。

複数のタートルを一つのスクリーン上で使いたい場合、オブジェクト指向インターフェイスを使わなければなりません。

注釈: 以下の文書では関数に対する引数リストが与えられています。メソッドでは、勿論、ここでは省略されている *self* が第一引数になります。

24.1.2 Turtle および Screen のメソッドの概観

Turtle のメソッド

Turtle の動き

移動および描画

forward() | *fd()*

backward() | *bk()* | *back()*

right() | *rt()*

left() | *lt()*

goto() | *setpos()* | *setposition()*

setx()

sety()

setheading() | *seth()*

home()

circle()

dot()

stamp()

clearstamp()

clearstamps()

undo()

speed()

Turtle の状態を知る

position() | *pos()*

towards()

xcor()

ycor()

heading()

distance()

設定と計測

degrees()

radians()

Pen の制御

描画状態

pendown() | *pd()* | *down()*

penup() | *pu()* | *up()*

pensize() | *width()*

pen()

isdown()

色の制御

color()

pencolor()

fillcolor()

塗りつぶし

filling()

begin_fill()

end_fill()

さらなる描画の制御

reset()

clear()

write()

タートルの状態

可視性

showturtle() | *st()*

hideturtle() | *ht()*

isvisible()

見た目

shape()

resizemode()

shapesize() | *turtlesize()*

shearfactor()

settiltangle()

tiltangle()

tilt()

shapetransform()

get_shapepoly()

イベントを利用する

onclick()

onrelease()

ondrag()

特別な Turtle のメソッド

begin_poly()

end_poly()

get_poly()

clone()

getturtle() | *getpen()*

getscreen()

setundobuffer()

undobufferentries()

TurtleScreen/Screen のメソッド

ウィンドウの制御

bgcolor()

bgpic()

clear() | *clearscreen()*

reset() | *resetscreen()*

screensize()

setworldcoordinates()

アニメーションの制御

delay()

tracer()

update()

スクリーンイベントを利用する

listen()

onkey() | *onkeyrelease()*

onkeypress()

onclick() | *onscreenclick()*

ontimer()

mainloop() | *done()*

設定と特殊なメソッド

mode()

colormode()

getcanvas()

getshapes()

register_shape() | *addshape()*

turtles()

window_height()

window_width()

入力メソッド

textinput()

numinput()

Screen 独自のメソッド

bye()

exitonclick()

setup()

title()

24.1.3 RawTurtle/Turtle のメソッドと対応する関数

この節のほとんどの例では `turtle` という名前の Turtle インスタンスを使います。

Turtle の動き

`turtle.forward`(*distance*)
`turtle.fd`(*distance*)

パラメータ `distance` -- 数 (整数または浮動小数点数)

タートルが頭を向けている方へ、タートルを距離 *distance* だけ前進させます。

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00,0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00,0.00)
```

`turtle.back`(*distance*)
`turtle.bk`(*distance*)
`turtle.backward`(*distance*)

パラメータ `distance` -- 数

タートルが頭を向けている方と反対方向へ、タートルを距離 *distance* だけ後退させます。タートルの向きは変えません。

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

`turtle.right`(*angle*)
`turtle.rt`(*angle*)

パラメータ `angle` -- 数 (整数または浮動小数点数)

タートルを *angle* 単位だけ右に回します。(単位のデフォルトは度ですが、*degrees()* と *radians()* 関数を使って設定できます。) 角度の向きはタートルのモードによって意味が変わります。*mode()* を参照してください。

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
```

(次のページに続く)

(前のページからの続き)

```
>>> turtle.heading()
337.0
```

turtle.left(*angle*)
turtle.lt(*angle*)

パラメータ **angle** -- 数 (整数または浮動小数点数)

タートルを *angle* 単位だけ左に回します。(単位のデフォルトは度ですが、*degrees()* と *radians()* 関数を使って設定できます。) 角度の向きはタートルのモードによって意味が変わります。*mode()* を参照してください。

```
>>> turtle.heading()
22.0
>>> turtle.left(45)
>>> turtle.heading()
67.0
```

turtle.goto(*x*, *y=None*)
turtle.setpos(*x*, *y=None*)
turtle.setposition(*x*, *y=None*)

パラメータ

- **x** -- 数または数のペア/ベクトル
- **y** -- 数または None

y が None の場合、*x* は座標のペアかまたは *Vec2D* (たとえば *pos()* で返されます) でなければなりません。

タートルを指定された絶対位置に移動します。ペンが下りていれば線を引きます。タートルの向きは変わりません。

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.setpos(60,30)
>>> turtle.pos()
(60.00,30.00)
>>> turtle.setpos((20,80))
>>> turtle.pos()
(20.00,80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00,0.00)
```

turtle.setx(*x*)

パラメータ **x** -- 数 (整数または浮動小数点数)

タートルの第一座標を x にします。第二座標は変わりません。

```
>>> turtle.position()
(0.00,240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00,240.00)
```

turtle.sety(*y*)

パラメータ **y** -- 数 (整数または浮動小数点数)

タートルの第二座標を y にします。第一座標は変わりません。

```
>>> turtle.position()
(0.00,40.00)
>>> turtle.sety(-10)
>>> turtle.position()
(0.00,-10.00)
```

turtle.setheading(*to_angle*)
turtle.seth(*to_angle*)

パラメータ **to_angle** -- 数 (整数または浮動小数点数)

タートルの向きを *to_angle* に設定します。以下はよく使われる方向を度で表わしたものです:

標準モード	logo モード
0 - 東	0 - 北
90 - 北	90 - 東
180 - 西	180 - 南
270 - 南	270 - 西

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

turtle.home()

タートルを原点 -- 座標 (0, 0) -- に移動し、向きを開始方向に設定します (開始方向はモードに依って違います。*mode()* を参照してください)。

```
>>> turtle.heading()
90.0
>>> turtle.position()
(0.00,-10.00)
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
```

(次のページに続く)

(前のページからの続き)

```
>>> turtle.heading()
0.0
```

turtle.**circle**(*radius*, *extent=None*, *steps=None*)

パラメータ

- **radius** -- 数
- **extent** -- 数 (または None)
- **steps** -- 整数 (または None)

半径 *radius* の円を描きます。中心はタートルの左 *radius* ユニットの点です。*extent* -- 角度です -- は円のどの部分を描くかを決定します。*extent* が与えられなければ、デフォルトで完全な円になります。*extent* が完全な円でない場合は、弧の一つの端点は、現在のペンの位置です。*radius* が正の場合、弧は反時計回りに描かれます。そうでなければ、時計回りです。最後にタートルの向きが *extent* 分だけ変わります。

円は内接する正多角形で近似されます。*steps* でそのために使うステップ数を決定します。この値は与えられなければ自動的に計算されます。また、これを正多角形の描画に利用することもできます。

```
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(50)
>>> turtle.position()
(-0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(120, 180)  # draw a semicircle
>>> turtle.position()
(0.00,240.00)
>>> turtle.heading()
180.0
```

turtle.**dot**(*size=None*, **color*)

パラメータ

- **size** -- 1 以上の整数 (与えられる場合には)
- **color** -- 色を表わす文字列またはタプル

直径 *size* の丸い点を *color* で指定された色で描きます。*size* が与えられなかった場合、pensize+4 と 2*pensize の大きい方が使われます。

```
>>> turtle.home()
>>> turtle.dot()
```

(次のページに続く)

(前のページからの続き)

```
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
>>> turtle.position()
(100.00,-0.00)
>>> turtle.heading()
0.0
```

turtle.**stamp**()

キャンバス上の現在タートルがいる位置にタートルの姿のハンコを押します。そのハンコに対して stamp_id が返されますが、これを使うと後で clearstamp(stamp_id) のように呼び出して消すことができます。

```
>>> turtle.color("blue")
>>> turtle.stamp()
11
>>> turtle.fd(50)
```

turtle.**clearstamp**(*stampid*)

パラメータ **stampid** -- 整数で、先立つ *stamp()* 呼出しで返された値でなければなりません

stampid に対応するハンコを消します。

```
>>> turtle.position()
(150.00,-0.00)
>>> turtle.color("blue")
>>> astamp = turtle.stamp()
>>> turtle.fd(50)
>>> turtle.position()
(200.00,-0.00)
>>> turtle.clearstamp(astamp)
>>> turtle.position()
(200.00,-0.00)
```

turtle.**clearstamps**(*n=None*)

パラメータ **n** -- 整数 (または None)

全ての、または最初の/最後の n 個のハンコを消します。n が None の場合、全てのハンコを消します。n が正の場合には最初の n 個、n が負の場合には最後の n 個を消します。

```
>>> for i in range(8):
...     turtle.stamp(); turtle.fd(30)
13
14
15
16
17
18
19
20
```

(次のページに続く)

(前のページからの続き)

```
>>> turtle.clearstamps(2)
>>> turtle.clearstamps(-2)
>>> turtle.clearstamps()
```

turtle.undo()

最後の (繰り返すことにより複数の) タートルの動きを取り消します。取り消しできる動きの最大数は undobuffer のサイズによって決まります。

```
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
...     turtle.undo()
```

turtle.speed(*speed=None*)

パラメータ **speed** -- 0 から 10 までの整数またはスピードを表わす文字列 (以下の説明を参照)

タートルのスピードを 0 から 10 までの範囲の整数に設定します。引数が与えられない場合は現在のスピードを返します。

与えられた数字が 10 より大きかったり 0.5 より小さかったりした場合は、スピードは 0 になります。スピードを表わす文字列は次のように数字に変換されます:

- "fastest": 0
- "fast": 10
- "normal": 6
- "slow": 3
- "slowest": 1

1 から 10 までのスピードを上げていくにつれて線を描いたりタートルが回ったりするアニメーションがだんだん速くなります。

注意: *speed* = 0 はアニメーションを無くします。forward/backward ではタートルがジャンプし、left/right では瞬時に方向を変えます。

```
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
```

Turtle の状態を知る

turtle.**position**()

turtle.**pos**()

タートルの現在位置を (*Vec2D* のベクトルとして) 返します。

```
>>> turtle.pos()
(440.00,-0.00)
```

turtle.**towards**(*x*, *y=None*)

パラメータ

- **x** -- 数または数のペア/ベクトルまたはタートルのインスタンス
- **y** -- *x* が数ならば数、そうでなければ None

タートルの位置から指定された (x,y) への直線の角度を返します。この値はタートルの開始方向にそして開始方向はモード ("standard"/"world" または "logo") に依存します。

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0,0)
225.0
```

turtle.**xcor**()

タートルの x 座標を返します。

```
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28,76.60)
>>> print(round(turtle.xcor(), 5))
64.27876
```

turtle.**ycor**()

タートルの y 座標を返します。

```
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print(turtle.pos())
(50.00,86.60)
>>> print(round(turtle.ycor(), 5))
86.60254
```

turtle.**heading**()

タートルの現在の向きを返します (返される値はタートルのモードに依存します。*mode()* を参照してください)。

```
>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0
```

turtle.**distance**(*x*, *y=None*)

パラメータ

- **x** -- 数または数のペア/ベクトルまたはタートルのインスタンス
- **y** -- *x* が数ならば数、そうでなければ None

タートルから与えられた (x,y) あるいはベクトルあるいは渡されたタートルへの距離を、タートルのステップを単位として測った値を返します。

```
>>> turtle.home()
>>> turtle.distance(30,40)
50.0
>>> turtle.distance((30,40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

設定と計測

turtle.**degrees**(*fullcircle=360.0*)

パラメータ **fullcircle** -- 数

角度を計る単位「度」を、円周を何等分するかという値に指定します。デフォルトは 360 等分で通常の意味での度です。

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0

Change angle measurement unit to grad (also known as gon,
grade, or gradian and equals 1/100-th of the right angle.)
>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
>>> turtle.degrees(360)
>>> turtle.heading()
90.0
```

turtle.**radians**()

角度を計る単位をラジアンにします。degrees(2*math.pi) と同じ意味です。

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966
```

Pen の制御

描画状態

turtle.pendown()
turtle.pd()
turtle.down()
ペンを下ろします -- 動くと線が引かれます。

turtle.penup()
turtle.pu()
turtle.up()
ペンを上げます -- 動いても線は引かれません。

turtle.pensize(*width=None*)
turtle.width(*width=None*)

パラメータ **width** -- 正の数

線の太さを *width* にするか、または現在の太さを返します。resizemode が "auto" でタートルの形が多角形の場合、その多角形も同じ太さで描画されます。引数が渡されなければ、現在の pensize が返されます。

```
>>> turtle.pensize()
1
>>> turtle.pensize(10)   # from here on lines of width 10 are drawn
```

turtle.pen(*pen=None*, ***pendict*)

パラメータ

- **pen** -- 以下にリストされたキーをもった辞書
- **pendict** -- 以下にリストされたキーをキーワードとするキーワード引数

ペンの属性を "pen-dictionary" に以下のキー/値ペアで設定するかまたは返します:

- "shown": True/False
- "pendown": True/False
- "pencolor": 色文字列または色タプル

- ”fillcolor”: 色文字列または色タプル
- ”pensize”: 正の数
- ”speed”: 0 から 10 までの整数
- ”resizemode”: ”auto” または ”user” または ”noresize”
- ”stretchfactor”: (正の数, 正の数)
- ”outline”: 正の数
- ”tilt”: 数

この辞書を以降の *pen()* 呼出しに渡して以前のペンの状態に復旧することができます。さらに一つ以上の属性をキーワード引数として渡すこともできます。一つの文で幾つものペンの属性を設定するのに使えます。

```
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
 ('shearfactor', 0.0), ('shown', True), ('speed', 9),
 ('stretchfactor', (1.0, 1.0)), ('tilt', 0.0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", "")
>>> turtle.penup()
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow')]
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red')]
```

turtle.isdown()

もしペンが下りていれば True を、上がっていれば False を返します。

```
>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
>>> turtle.isdown()
True
```

色の制御

turtle.pencolor(**args*)

ペンの色 (pencolor) を設定するかまたは返します。

4 種類の入力形式が受け入れ可能です:

`pencolor()` 現在のペンの色を色指定文字列またはタプルで返します (例を見て下さい)。次の color/pencolor/fillcolor の呼び出しへの入力に使うこともあるでしょう。

`pencolor(colorstring)` ペンの色を *colorstring* に設定します。その値は Tk の色指定文字列で、`"red"`, `"yellow"`, `"#33cc8c"` のような文字列です。

`pencolor((r, g, b))` ペンの色を *r*, *g*, *b* のタプルで表された RGB の色に設定します。各 *r*, *g*, *b* は 0 から colormode の間の値でなければなりません。ここで colormode は 1.0 か 255 のどちらかです (*colormode()* を参照)。

`pencolor(r, g, b)` ペンの色を *r*, *g*, *b* で表された RGB の色に設定します。各 *r*, *g*, *b* は 0 から colormode の間の値でなければなりません。

タートルの形 (turtleshape) が多角形の場合、多角形の外側が新しく設定された色で描かれます。

```
>>> colormode()
1.0
>>> turtle.pencolor()
'red'
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
'brown'
>>> tup = (0.2, 0.8, 0.55)
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)
>>> colormode(255)
>>> turtle.pencolor()
(51.0, 204.0, 140.0)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
```

`turtle.fillcolor(*args)`

塗りつぶしの色 (fillcolor) を設定するかまたは返します。

4 種類の入力形式が受け入れ可能です:

`fillcolor()` 現在の塗りつぶしの色を色指定文字列またはタプルで返します (例を見て下さい)。次の color/pencolor/fillcolor の呼び出しへの入力に使うこともあるでしょう。

`fillcolor(colorstring)` 塗りつぶしの色を *colorstring* に設定します。その値は Tk の色指定文字列で、`"red"`, `"yellow"`, `"#33cc8c"` のような文字列です。

`fillcolor((r, g, b))` 塗りつぶしの色を *r*, *g*, *b* のタプルで表された RGB の色に設定します。各 *r*, *g*, *b* は 0 から colormode の間の値でなければなりません。ここで colormode は 1.0 か 255 のどちらかです (*colormode()* を参照)。

`fillcolor(r, g, b)` 塗りつぶしの色を *r*, *g*, *b* で表された RGB の色に設定します。各 *r*, *g*, *b* は 0 から colormode の間の値でなければなりません。

タートルの形 (turtleshape) が多角形の場合、多角形の内側が新しく設定された色で描かれます。

```
>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
'violet'
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor((50, 193, 143))  # Integers, not floats
>>> turtle.fillcolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor('#ffffff')
>>> turtle.fillcolor()
(255.0, 255.0, 255.0)
```

turtle.color(*args)

ペンの色 (pencolor) と塗りつぶしの色 (fillcolor) を設定するかまたは返します。

いくつかの入力形式が受け入れ可能です。形式ごとに 0 から 3 個の引数を以下のように使います:

color() 現在のペンの色と塗りつぶしの色を *pencolor()* および *fillcolor()* で返される色指定文字列またはタプルのペアで返します。

color(colorstring), color((r,g,b)), color(r,g,b) *pencolor()* の入力と同じですが、塗りつぶしの色とペンの色、両方を与えられた値に設定します。

color(colorstring1, colorstring2), color((r1,g1,b1), (r2,g2,b2))
pencolor(colorstring1) および fillcolor(colorstring2) を呼び出すのと等価です。もう一つの入力形式についても同様です。

タートルの形 (turtleshape) が多角形の場合、多角形の内側も外側も新しく設定された色で描かれます。

```
>>> turtle.color("red", "green")
>>> turtle.color()
('red', 'green')
>>> color("#285078", "#a0c8f0")
>>> color()
((40.0, 80.0, 120.0), (160.0, 200.0, 240.0))
```

こちらも参照: スクリーンのメソッド *colormode()* 。

塗りつぶし

turtle.filling()

Return fillstate (True if filling, False else).

```
>>> turtle.begin_fill()
>>> if turtle.filling():
...     turtle.pensize(5)
... else:
...     turtle.pensize(3)
```

turtle.begin_fill()

To be called just before drawing a shape to be filled.

turtle.**end_fill**()

最後に呼び出された *begin_fill()* の後に描かれた図形を塗りつぶします。

Whether or not overlap regions for self-intersecting polygons or multiple shapes are filled depends on the operating system graphics, type of overlap, and number of overlaps. For example, the Turtle star above may be either all yellow or have some white regions.

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

さらなる描画の制御

turtle.**reset**()

タートルの描いたものをスクリーンから消し、タートルを中心に戻して、全ての変数をデフォルト値に設定し直します。

```
>>> turtle.goto(0,-22)
>>> turtle.left(100)
>>> turtle.position()
(0.00,-22.00)
>>> turtle.heading()
100.0
>>> turtle.reset()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**clear**()

タートルの描いたものをスクリーンから消します。タートルは動かしません。タートルの状態と位置、それに他のタートルたちの描いたものは影響を受けません。

turtle.**write**(*arg*, *move=False*, *align="left"*, *font=("Arial", 8, "normal")*)

パラメータ

- **arg** -- TurtleScreen に書かれるオブジェクト
- **move** -- True/False
- **align** -- 文字列 "left", "center", right" のどれか
- **font** -- 三つ組み (fontname, fontsize, fonttype)

文字を書きます— *arg* の文字列表現を、現在のタートルの位置に、*align* ("left", "center", "right" のどれか) に従って、与えられたフォントで。もし *move* が真ならば、ペンは書いた文の右下隅に移動します。デフォルトでは、*move* は `False` です。

```
>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)
```

タートルの状態

可視性

turtle.**hideturtle**()

turtle.**ht**()

タートルを見えなくします。複雑な図を描いている途中、タートルが見えないようにするのは良い考えです。というのもタートルを隠すことで描画が目に見えて速くなるからです。

```
>>> turtle.hideturtle()
```

turtle.**showturtle**()

turtle.**st**()

タートルが見えるようにします。

```
>>> turtle.showturtle()
```

turtle.**isvisible**()

タートルが見えている状態ならば `True` を、隠されていれば `False` を返します。

```
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

見た目

turtle.**shape**(*name=None*)

パラメータ **name** -- 形の名前 (shapename) として正しい文字列

タートルの形を与えられた名前 (*name*) の形に設定するか、もしくは名前が与えられなければ現在の形の名前を返します。*name* という名前の形は TurtleScreen の形の辞書に載っていなければなりません。最初は次の多角形が載っています: "arrow", "turtle", "circle", "square", "triangle", "classic"。形についての扱いを学ぶには Screen のメソッド *register_shape()* を参照して下さい。

```
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```

turtle.**resizemode**(*rmode=None*)

パラメータ **rmode** -- 文字列 "auto", "user", "noresize" のどれか

サイズ変更のモード (resizemode) を "auto", "user", "noresize" のどれかに設定します。もし *rmode* が与えられなければ、現在のサイズ変更モードを返します。それぞれのサイズ変更モードは以下の効果を持ちます:

- "auto": ペンのサイズに対応してタートルの見た目を調整します。
- "user": 伸長係数 (stretchfactor) およびアウトライン幅 (outlinewidth) の値に対応してタートルの見た目を調整します。これらの値は *shapesize()* で設定します。
- "noresize": タートルの見た目を調整しません。

`resizemode("user")` は *shapesize()* に引数を渡したときに呼び出されます。

```
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

turtle.**shapesize**(*stretch_wid=None*, *stretch_len=None*, *outline=None*)
turtle.**turtlesize**(*stretch_wid=None*, *stretch_len=None*, *outline=None*)

パラメータ

- **stretch_wid** -- 正の数
- **stretch_len** -- 正の数
- **outline** -- 正の数

ペンの属性 x/y-伸長係数および/またはアウトラインを返すかまたは設定します。サイズ変更のモードは "user" に設定されます。サイズ変更のモードが "user" に設定されたときかつそのときに限り、タートルは伸長係数 (stretchfactor) に従って伸長されて表示されます。*stretch_wid* は進行方向に直交する向きの伸長係数で、*stretch_len* は進行方向に沿ったの伸長係数、*outline* はアウトラインの幅を決めるものです。

```
>>> turtle.shapesize()
(1.0, 1.0, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

turtle.**shearfactor**(*shear=None*)

パラメータ **shear** -- number (optional)

Set or return the current shearfactor. Shear the turtleshape according to the given shearfactor shear, which is the tangent of the shear angle. Do *not* change the turtle's heading (direction of movement). If shear is not given: return the current shearfactor, i. e. the tangent of the shear angle, by which lines parallel to the heading of the turtle are sheared.

```
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.shearfactor(0.5)
>>> turtle.shearfactor()
0.5
```

`turtle.tilt`(*angle*)

パラメータ **angle** -- 数

タートルの形 (turtleshape) を現在の傾斜角から角度 (*angle*) だけ回転します。このときタートルの進む方向は **変わりません** 。

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

`turtle.settiltangle`(*angle*)

パラメータ **angle** -- 数

タートルの形 (turtleshape) を現在の傾斜角に関わらず、指定された角度 (*angle*) の向きに回転します。タートルの進む方向は **変わりません** 。

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.settiltangle(45)
>>> turtle.fd(50)
>>> turtle.settiltangle(-45)
>>> turtle.fd(50)
```

バージョン 3.1 で非推奨.

`turtle.tiltangle`(*angle=None*)

パラメータ **angle** -- a number (optional)

Set or return the current tilt-angle. If angle is given, rotate the turtleshape to point in the direction specified by angle, regardless of its current tilt-angle. Do *not* change the turtle's heading (direction of movement). If angle is not given: return the current tilt-angle, i. e. the angle between the orientation of the turtleshape and the heading of the turtle (its direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

turtle.**shapetransform**(*t11=None*, *t12=None*, *t21=None*, *t22=None*)

パラメータ

- **t11** -- a number (optional)
- **t12** -- a number (optional)
- **t21** -- a number (optional)
- **t12** -- a number (optional)

Set or return the current transformation matrix of the turtle shape.

If none of the matrix elements are given, return the transformation matrix as a tuple of 4 elements. Otherwise set the given elements and transform the turtleshape according to the matrix consisting of first row t11, t12 and second row t21, t22. The determinant t11 * t22 - t12 * t21 must not be zero, otherwise an error is raised. Modify stretchfactor, shearfactor and tiltangle according to the given matrix.

```
>>> turtle = Turtle()
>>> turtle.shape("square")
>>> turtle.shapesize(4,2)
>>> turtle.shearfactor(-0.5)
>>> turtle.shapetransform()
(4.0, -1.0, -0.0, 2.0)
```

turtle.**get_shapepoly**()

Return the current shape polygon as tuple of coordinate pairs. This can be used to define a new shape or components of a compound shape.

```
>>> turtle.shape("square")
>>> turtle.shapetransform(4, -1, 0, 2)
>>> turtle.get_shapepoly()
((50, -20), (30, 20), (-50, 20), (-30, -20))
```

イベントを利用する

turtle.**onclick**(*fun*, *btn=1*, *add=None*)

パラメータ

- **fun** -- 2 引数の関数でキャンバスのクリックされた点の座標を引数として呼び出されるものです
- **btn** -- マウスボタンの番号、デフォルトは 1 (左マウスボタン)
- **add** -- `True` または `False` -- `True` ならば、新しい束縛が追加されますが、そうでなければ、以前の束縛を置き換えます。

fun をタートルのマウスクリック (mouse-click) イベントに束縛します。*fun* が `None` ならば、既存の束縛が取り除かれます。無名タートル、つまり手続き的なやり方の例です:

```
>>> def turn(x, y):
...     left(180)
...
>>> onclick(turn)  # Now clicking into the turtle will turn it.
>>> onclick(None)  # event-binding will be removed
```

turtle.**onrelease**(*fun*, *btn=1*, *add=None*)

パラメータ

- **fun** -- 2 引数の関数でキャンバスのクリックされた点の座標を引数として呼び出されるものです
- **btn** -- マウスボタンの番号、デフォルトは 1 (左マウスボタン)
- **add** -- `True` または `False` -- `True` ならば、新しい束縛が追加されますが、そうでなければ、以前の束縛を置き換えます。

fun をタートルのマウスボタンリリース (mouse-button-release) イベントに束縛します。*fun* が `None` ならば、既存の束縛が取り除かれます。

```
>>> class MyTurtle(Turtle):
...     def glow(self,x,y):
...         self.fillcolor("red")
...     def unglow(self,x,y):
...         self.fillcolor("")
...
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)     # clicking on turtle turns fillcolor red,
>>> turtle.onrelease(turtle.unglow) # releasing turns it to transparent.
```

turtle.**ondrag**(*fun*, *btn=1*, *add=None*)

パラメータ

- **fun** -- 2 引数の関数でキャンバスのクリックされた点の座標を引数として呼び出されるものです
- **btn** -- マウスボタンの番号、デフォルトは 1 (左マウスボタン)
- **add** -- `True` または `False` -- `True` ならば、新しい束縛が追加されますが、そうでなければ、以前の束縛を置き換えます。

fun をタートルのマウスムーブ (mouse-move) イベントに束縛します。*fun* が `None` ならば、既存の束縛が取り除かれます。

注意: 全てのマウスムーブイベントのシーケンスに先立ってマウスクリックイベントが起こります。

```
>>> turtle.ondrag(turtle.goto)
```

この後、タートルをクリックしてドラッグするとタートルはスクリーン上を動きそれによって (ペンが下りていれば) 手書きの線ができあがります。

特別な Turtle のメソッド

turtle.begin_poly()

多角形の頂点の記録を開始します。現在のタートル位置が最初の頂点です。

turtle.end_poly()

多角形の頂点の記録を停止します。現在のタートル位置が最後の頂点です。この頂点が最初の頂点と結ばれます。

turtle.get_poly()

最後に記録された多角形を返します。

```
>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
>>> turtle.left(20)
>>> turtle.fd(30)
>>> turtle.left(60)
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)
```

turtle.clone()

位置、向きその他のプロパティがそっくり同じタートルのクローンを作って返します。

```
>>> mick = Turtle()
>>> joe = mick.clone()
```

turtle.getturtle()

turtle.getpen()

Turtle オブジェクトそのものを返します。唯一の意味のある使い方: 無名タートルを返す関数として

使う:

```
>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
```

turtle.**getscreen**()

タートルが描画中の *TurtleScreen* オブジェクトを返します。TurtleScreen のメソッドをそのオブジェクトに対して呼び出すことができます。

```
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

turtle.**setundobuffer**(*size*)

パラメータ **size** -- 整数または None

アンドゥバッファを設定または無効化します。*size* が整数ならばそのサイズの空のアンドゥバッファを用意します。*size* の値はタートルのアクションを何度 *undo()* メソッド/関数で取り消せるかの最大数を与えます。*size* が None ならば、アンドゥバッファは無効化されます。

```
>>> turtle.setundobuffer(42)
```

turtle.**undobufferentries**()

アンドゥバッファのエントリー数を返します。

```
>>> while undobufferentries():
...     undo()
```

Compound shapes

合成されたタートルの形、つまり幾つかの色の違う多角形から成るような形を使うには、以下のように補助クラス *Shape* を直接使わなければなりません:

1. タイプ "compound" の空の Shape オブジェクトを作ります。
2. addcomponent() メソッドを使って、好きなだけここにコンポーネントを追加します。

   例えば:

```
>>> s = Shape("compound")
>>> poly1 = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s.addcomponent(poly1, "red", "blue")
>>> poly2 = ((0,0),(10,-5),(-10,-5))
>>> s.addcomponent(poly2, "blue", "red")
```

3. こうして作った Shape を Screen の形のリスト (shapelist) に追加して使います:

```
>>> register_shape("myshape", s)
>>> shape("myshape")
```

注釈: *Shape* クラスは *register_shape()* の内部では違った使われ方をします。アプリケーションを書く人が Shape クラスを扱わなければならないのは、上で示したように合成された形を使うとき **だけ** です！

24.1.4 TurtleScreen/Screen のメソッドと対応する関数

この節のほとんどの例では `screen` という名前の TurtleScreen インスタンスを使います。

ウィンドウの制御

`turtle.bgcolor`(**args*)

> **パラメータ** **args** -- 色文字列または 0 から colormode の範囲の数 3 つ、またはそれを三つ組みにしたもの
>
> TurtleScreen の背景色を設定するかまたは返します。

```
>>> screen.bgcolor("orange")
>>> screen.bgcolor()
'orange'
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()
(128.0, 0.0, 128.0)
```

`turtle.bgpic`(*picname=None*)

> **パラメータ** **picname** -- 文字列で gif ファイルの名前 `"nopic"` 、または `None`
>
> 背景の画像を設定するかまたは現在の背景画像 (backgroundimage) の名前を返します。*picname* がファイル名ならば、その画像を背景に設定します。*picname* が `"nopic"` ならば、(もしあれば) 背景画像を削除します。*picname* が `None` ならば、現在の背景画像のファイル名を返します。

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif"
```

`turtle.clear`()

`turtle.clearscreen`()

> 全ての図形と全てのタートルを TurtleScreen から削除します。そして空になった TurtleScreen をリセットして初期状態に戻します: 白い背景、背景画像もイベント束縛もなく、トレーシングはオンです。

注釈: この TurtleScreen メソッドはグローバル関数としては `clearscreen` という名前でだけ使えます。グローバル関数 `clear` は Turtle メソッドの `clear` から派生した別ものです。

`turtle.reset()`
`turtle.resetscreen()`
スクリーン上の全てのタートルをリセットしその初期状態に戻します。

注釈: この TurtleScreen メソッドはグローバル関数としては `resetscreen` という名前でだけ使えます。グローバル関数 `reset` は Turtle メソッドの `reset` から派生した別ものです。

`turtle.screensize(`*canvwidth=None*, *canvheight=None*, *bg=None*`)`

パラメータ

- **canvwidth** -- 正の整数でピクセル単位の新しいキャンバス幅 (canvaswidth)
- **canvheight** -- 正の整数でピクセル単位の新しいキャンバス高さ (canvasheight)
- **bg** -- 色文字列または色タプルで新しい背景色

引数が渡されなければ、現在の (キャンバス幅, キャンバス高さ) を返します。そうでなければタートルが描画するキャンバスのサイズを変更します。描画ウィンドウには影響しません。キャンバスの隠れた部分を見るためにはスクロールバーを使って下さい。このメソッドを使うと、以前はキャンバスの外にあったそうした図形の一部を見えるようにすることができます。

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000,1500)
>>> screen.screensize()
(2000, 1500)
```

逃げ出してしまったタートルを探すためとかね ;-)

`turtle.setworldcoordinates(`*llx*, *lly*, *urx*, *ury*`)`

パラメータ

- **llx** -- 数でキャンバスの左下隅の x-座標
- **lly** -- 数でキャンバスの左下隅の y-座標
- **urx** -- 数でキャンバスの右上隅の x-座標
- **ury** -- 数でキャンバスの右上隅の y-座標

ユーザー定義座標系を準備し必要ならばモードを "world" に切り替えます。この動作は `screen.reset()` を伴います。すでに "world" モードになっていた場合、全ての図形は新しい座標に従って再描画されます。

重要なお知らせ: ユーザー定義座標系では角度が歪むかもしれません。

```
>>> screen.reset()
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...     left(10)
...
>>> for _ in range(8):
...     left(45); fd(2)   # a regular octagon
```

アニメーションの制御

turtle.**delay**(*delay=None*)

パラメータ **delay** -- 正の整数

描画の遅延 (*delay*) をミリ秒単位で設定するかまたはその値を返します。(これは概ね引き続くキャンバス更新の時間間隔です。) 遅延が大きくなると、アニメーションは遅くなります。

オプション引数:

```
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

turtle.**tracer**(*n=None*, *delay=None*)

パラメータ

- **n** -- 非負整数
- **delay** -- 非負整数

タートルのアニメーションをオン・オフし、描画更新の遅延を設定します。*n* が与えられた場合、通常のスクリーン更新のうち 1/n しか実際に実行されません。(複雑なグラフィックスの描画を加速するのに使えます。) 引数なしで呼び出されたなら、現在保存されている n の値を返します。二つ目の引数は遅延の値を設定します (*delay()* も参照)。

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

turtle.**update**()

TurtleScreen の更新を実行します。トレーサーがオフの時に使われます。

RawTurtle/Turtle のメソッド *speed()* も参照して下さい。

スクリーンイベントを利用する

turtle.**listen**(*xdummy=None*, *ydummy=None*)

TurtleScreen に (キー・イベントを収集するために) フォーカスします。ダミー引数は *listen()* を onclick メソッドに渡せるようにするためのものです。

turtle.**onkey**(*fun*, *key*)
turtle.**onkeyrelease**(*fun*, *key*)

パラメータ

- **fun** -- 引数なしの関数または None
- **key** -- 文字列: キー (例 "a") またはキー・シンボル (例 "space")

fun を指定されたキーのキーリリース (key-release) イベントに束縛します。*fun* が None ならばイベント束縛は除かれます。注意: キー・イベントを登録できるようにするためには TurtleScreen はフォーカスを持っていないとなりません (*listen()* を参照)。

```
>>> def f():
...     fd(50)
...     lt(60)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.**onkeypress**(*fun*, *key=None*)

パラメータ

- **fun** -- 引数なしの関数または None
- **key** -- 文字列: キー (例 "a") またはキー・シンボル (例 "space")

Bind *fun* to key-press event of key if key is given, or to any key-press-event if no key is given. Remark: in order to be able to register key-events, TurtleScreen must have focus. (See method *listen()*.)

```
>>> def f():
...     fd(50)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.**onclick**(*fun*, *btn=1*, *add=None*)
turtle.**onscreenclick**(*fun*, *btn=1*, *add=None*)

パラメータ

- **fun** -- 2 引数の関数でキャンバスのクリックされた点の座標を引数として呼び出されるものです
- **btn** -- マウスボタンの番号、デフォルトは 1 (左マウスボタン)

- **add** -- `True` または `False` -- `True` ならば、新しい束縛が追加されますが、そうでなければ、以前の束縛を置き換えます。

fun をタートルのマウスクリック (mouse-click) イベントに束縛します。*fun* が `None` ならば、既存の束縛が取り除かれます。

Example for a `screen` という名の TurtleScreen インスタンスと `turtle` という名前の Turtle インスタンスの例:

```
>>> screen.onclick(turtle.goto) # Subsequently clicking into the TurtleScreen will
>>>                             # make the turtle move to the clicked point.
>>> screen.onclick(None)        # remove event binding again
```

注釈: この TurtleScreen メソッドはグローバル関数としては `onscreenclick` という名前でだけ使えます。グローバル関数 `onclick` は Turtle メソッドの `onclick` から派生した別ものです。

---

turtle.ontimer(*fun*, *t=0*)

パラメータ

- **fun** -- 引数なし関数
- **t** -- 数 >= 0

t ミリ秒後に *fun* を呼び出すタイマーを仕掛けます。

```
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...         screen.ontimer(f, 250)
>>> f()   ### makes the turtle march around
>>> running = False
```

turtle.mainloop()

turtle.done()

Starts event loop - calling Tkinter's mainloop function. Must be the last statement in a turtle graphics program. Must *not* be used if a script is run from within IDLE in -n mode (No subprocess) - for interactive use of turtle graphics.

```
>>> screen.mainloop()
```

入力メソッド

turtle.**textinput**(*title*, *prompt*)

パラメータ

- **title** -- string
- **prompt** -- string

Pop up a dialog window for input of a string. Parameter title is the title of the dialog window, prompt is a text mostly describing what information to input. Return the string input. If the dialog is canceled, return `None`.

```
>>> screen.textinput("NIM", "Name of first player:")
```

turtle.**numinput**(*title*, *prompt*, *default=None*, *minval=None*, *maxval=None*)

パラメータ

- **title** -- string
- **prompt** -- string
- **default** -- number (optional)
- **minval** -- number (optional)
- **maxval** -- number (optional)

Pop up a dialog window for input of a number. title is the title of the dialog window, prompt is a text mostly describing what numerical information to input. default: default value, minval: minimum value for input, maxval: maximum value for input The number input must be in the range minval .. maxval if these are given. If not, a hint is issued and the dialog remains open for correction. Return the number input. If the dialog is canceled, return `None`.

```
>>> screen.numinput("Poker", "Your stakes:", 1000, minval=10, maxval=10000)
```

設定と特殊なメソッド

turtle.**mode**(*mode=None*)

パラメータ **mode** -- 文字列 ”standard”, ”logo”, ”world” のいずれか

タートルのモード (”standard”, ”logo”, ”world” のいずれか) を設定してリセットします。モードが渡されなければ現在のモードが返されます。

モード "standard" は古い *turtle* 互換です。モード "logo" は Logo タートルグラフィックスとほぼ互換です。モード "world" はユーザーの定義した「世界座標 (world coordinates)」を使います。**重要なお知らせ**: このモードでは x/y 比が 1 でないと角度が歪むかもしれません。

モード	タートルの向きの初期値	正の角度
"standard"	右 (東) 向き	反時計回り
"logo"	上 (北) 向き	時計回り

```
>>> mode("logo")   # resets turtle heading to north
>>> mode()
'logo'
```

turtle.colormode(*cmode=None*)

パラメータ **cmode** -- 1.0 か 255 のどちらかの値

色モード (colormode) を返すか、または 1.0 か 255 のどちらかの値に設定します。設定した後は、色トリプルの *r*, *g*, *b* 値は 0 から *cmode* の範囲になければなりません。

```
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
     ...
TurtleGraphicsError: bad color sequence: (240, 160, 80)
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

turtle.getcanvas()

この TurtleScreen の Canvas を返します。Tkinter の Canvas を使って何をするか知っている人には有用です。

```
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas object ...>
```

turtle.getshapes()

現在使うことのできる全てのタートルの形のリストを返します。

```
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

turtle.register_shape(*name*, *shape=None*)

turtle.addshape(*name*, *shape=None*)

この関数を呼び出す三つの異なる方法があります:

(1) *name* が gif ファイルの名前で *shape* が None: 対応する画像の形を取り込みます。

```
>>> screen.register_shape("turtle.gif")
```

注釈: 画像の形はタートルが向きを変えても **回転しません** ので、タートルがどちらを向いているか見ても判りません!

---

(2) *name* が任意の文字列で *shape* が座標ペアのタプル: 対応する多角形を取り込みます。

```
>>> screen.register_shape("triangle", ((5,-3), (0,5), (-5,-3)))
```

(3) *name* が任意の文字列で *shape* が (合成形の) *Shape* オブジェクト: 対応する合成形を取り込みます。

タートルの形を TurtleScreen の形リスト (shapelist) に加えます。このように登録された形だけが `shape(shapename)` コマンドに使えます。

turtle.turtles()

スクリーン上のタートルのリストを返します。

```
>>> for turtle in screen.turtles():
...     turtle.color("red")
```

turtle.window_height()

タートルウィンドウの高さを返します。

```
>>> screen.window_height()
480
```

turtle.window_width()

タートルウィンドウの幅を返します。

```
>>> screen.window_width()
640
```

Screen 独自のメソッド、TurtleScreen から継承したもの以外

turtle.bye()

タートルグラフィックス (turtlegraphics) のウィンドウを閉じます。

turtle.exitonclick()

スクリーン上のマウスクリックに `bye()` メソッドを束縛します。

設定辞書中の "using_IDLE" の値が `False` (デフォルトです) の場合、さらにメインループ (mainloop) に入ります。注意: もし IDLE が `-n` スイッチ (サブプロセスなし) 付きで使われているときは、この値は `turtle.cfg` の中で `True` とされているべきです。この場合、IDLE のメインループもクライアントスクリプトから見てアクティブです。

turtle.**setup**(*width=_CFG["width"], height=_CFG["height"], startx=_CFG["leftright"], starty=_CFG["topbottom"]*)

メインウィンドウのサイズとポジションを設定します。引数のデフォルト値は設定辞書に収められており、turtle.cfg ファイルを通じて変更できます。

パラメータ

- **width** -- 整数ならばピクセル単位のサイズ、浮動小数点数ならばスクリーンに対する割合 (スクリーンの 50% がデフォルト)
- **height** -- 整数ならばピクセル単位の高さ、浮動小数点数ならばスクリーンに対する割合 (スクリーンの 75% がデフォルト)
- **startx** -- 正の数ならばスクリーンの左端からピクセル単位で測った開始位置、負の数ならば右端から、None ならば水平方向に真ん中
- **starty** -- 正の数ならばスクリーンの上端からピクセル単位で測った開始位置、負の数ならば下端から、None ならば垂直方向に真ん中

```
>>> screen.setup (width=200, height=200, startx=0, starty=0)
>>>              # sets window to 200x200 pixels, in upper left of screen
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
>>>              # sets window to 75% of screen by 50% of screen and centers
```

turtle.**title**(*titlestring*)

パラメータ **titlestring** -- タートルグラフィックスウィンドウのタイトルバーに表示される文字列

ウインドウのタイトルを *titlestring* に設定します。

```
>>> screen.title("Welcome to the turtle zoo!")
```

24.1.5 Public classes

class turtle.**RawTurtle**(*canvas*)
class turtle.**RawPen**(*canvas*)

パラメータ **canvas** -- tkinter.Canvas, *ScrolledCanvas*, *TurtleScreen* のいずれか

タートルを作ります。タートルには上の「Turtle/RawTurtle のメソッド」で説明した全てのメソッドがあります。

class turtle.**Turtle**

RawTurtle のサブクラスで同じインターフェイスを持ちますが、最初に必要になったとき自動的に作られる *Screen* オブジェクトに描画します。

class turtle.**TurtleScreen**(*cv*)

パラメータ **cv** -- tkinter.Canvas

上で説明した `setbg()` のようなスクリーン向けのメソッドを提供します。

class turtle.Screen

TurtleScreen のサブクラスで *4 つのメソッドが加わっています* 。

class turtle.ScrolledCanvas(*master*)

パラメータ **master** -- この ScrolledCanvas すなわちスクロールバーの付いた Tkinter canvas を収める Tkinter ウィジェット

タートルたちが遊び回る場所として自動的に ScrolledCanvas を提供する Screen クラスによって使われます。

class turtle.Shape(*type_*, *data*)

パラメータ **type_** -- 文字列 "polygon", "image", "compound" のいずれか

形をモデル化するデータ構造。ペア `(type_, data)` は以下の仕様に従わなければなりません:

*type_*	*data*
"polygon"	多角形タプル、すなわち座標ペアのタプル
"image"	画像 (この形式は内部的にのみ使用されます!)
"compound"	`None` (合成形は *addcomponent()* メソッドを使って作らなければなりません)

addcomponent(*poly*, *fill*, *outline=None*)

パラメータ

- **poly** -- 多角形、すなわち数のペアのタプル
- **fill** -- *poly* を塗りつぶす色
- **outline** -- *poly* のアウトラインの色 (与えられた場合)

例:

```
>>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... add more components and then use register_shape()
```

Compound shapes を参照。

class turtle.Vec2D(*x*, *y*)

2 次元ベクトルのクラスで、タートルグラフィックスを実装するための補助クラス。タートルグラフィックスを使ったプログラムでも有用でしょう。タプルから派生しているので、ベクターはタプルです!

以下の演算が使えます (*a*, *b* はベクトル、*k* は数):

- `a + b` ベクトル和

- `a - b` ベクトル差
- `a * b` 内積
- `k * a` および `a * k` スカラー倍
- `abs(a)` a の絶対値
- `a.rotate(angle)` 回転

24.1.6 ヘルプと設定

ヘルプの使い方

Screen と Turtle クラスのパブリックメソッドはドキュメント文字列で網羅的に文書化されていますので、Python のヘルプ機能を通じてオンラインヘルプとして利用できます:

- IDLE を使っているときは、打ち込んだ関数/メソッド呼び出しのシグニチャとドキュメント文字列の一行目がツールチップとして表示されます。
- *help()* をメソッドや関数に対して呼び出すとドキュメント文字列が表示されます:

```
>>> help(Screen.bgcolor)
Help on method bgcolor in module turtle:

bgcolor(self, *args) unbound turtle.Screen method
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.


      >>> screen.bgcolor("orange")
      >>> screen.bgcolor()
      "orange"
      >>> screen.bgcolor(0.5,0,0.5)
      >>> screen.bgcolor()
      "#800080"

>>> help(Turtle.penup)
Help on method penup in module turtle:

penup(self) unbound turtle.Turtle method
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    >>> turtle.penup()
```

- メソッドに由来する関数のドキュメント文字列は変更された形をとります:

```
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.

    Example::

      >>> bgcolor("orange")
      >>> bgcolor()
      "orange"
      >>> bgcolor(0.5,0,0.5)
      >>> bgcolor()
      "#800080"

>>> help(penup)
Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    Example:
    >>> penup()
```

これらの変更されたドキュメント文字列はインポート時にメソッドから導出される関数定義と一緒に自動的に作られます。

ドキュメント文字列の翻訳

Screen と Turtle クラスのパブリックメソッドについて、キーがメソッド名で値がドキュメント文字列である辞書を作るユーティリティがあります。

turtle.write_docstringdict(*filename="turtle_docstringdict"*)

パラメータ **filename** -- ファイル名として使われる文字列

ドキュメント文字列辞書 (docstring-dictionary) を作って与えられたファイル名の Python スクリプトに書き込みます。この関数はわざわざ呼び出さなければなりません (タートルグラフィックスのクラスから使われることはありません)。ドキュメント文字列辞書は *filename*.py という Python スクリプトに書き込まれます。ドキュメント文字列の異なった言語への翻訳に対するテンプレートとして使われることを意図したものです。

もしあなたが (またはあなたの生徒さんが) *turtle* を自国語のオンラインヘルプ付きで使いたいならば、ドキュメント文字列を翻訳してできあがったファイルをたとえば turtle_docstringdict_german.py という

名前で保存しなければなりません。

さらに turtle.cfg ファイルで適切な設定をしておけば、このファイルがインポート時に読み込まれて元の英語のドキュメント文字列を置き換えます。

この文書を書いている時点ではドイツ語とイタリア語のドキュメント文字列辞書が存在します。(glingl@aon.at にリクエストして下さい。)

Screen および Turtle の設定方法

初期デフォルト設定では古い turtle の見た目と振る舞いを真似るようにして、互換性を最大限に保つようにしています。

このモジュールの特性を反映した、あるいは個々人の必要性 (たとえばクラスルームでの使用) に合致した、異なった設定を使いたい場合、設定ファイル turtle.cfg を用意してインポート時に読み込ませその設定に従わせることができます。

初期設定は以下の turtle.cfg に対応します:

```
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

いくつかピックアップしたエントリーの短い説明:

- 最初の 4 行は Screen.setup() メソッドの引数に当たります。
- 5 行目 6 行目は Screen.screensize() メソッドの引数に当たります。
- *shape* は最初から用意されている形ならどれでも使えます (arrow, turtle など)。詳しくは help(shape) をお試し下さい。
- 塗りつぶしの色 (fillcolor) を使いたくない (つまりタートルを透明にしたい) 場合、fillcolor = "" と書かなければなりません (しかし全ての空でない文字列は cfg ファイル中で引用符を付けてはいけません)。

- タートルにその状態を反映させるためには `resizemode = auto` とします。
- たとえば `language = italian` とするとドキュメント文字列辞書 (docstringdict) として `turtle_docstringdict_italian.py` がインポート時に読み込まれます (もしそれがインポートパス、たとえば *turtle* と同じディレクトリにあれば)。
- *exampleturtle* および *examplescreen* はこれらのオブジェクトのドキュメント文字列内での呼び名を決めます。メソッドのドキュメント文字列から関数のドキュメント文字列に変換する際に、これらの名前は取り除かれます。
- *using_IDLE*: IDLE とその -n スイッチ (サブプロセスなし) を常用するならば、この値を `True` に設定して下さい。これにより *exitonclick()* がメインループ (mainloop) に入るのを阻止します。

`turtle.cfg` ファイルは *turtle* の保存されているディレクトリと現在の作業ディレクトリに追加的に存在し得ます。後者が前者の設定をオーバーライドします。

`Lib/turtledemo` ディレクトリにも `turtle.cfg` ファイルがあります。デモを実際に (できればデモビュワーからでなく) 実行してそこに書かれたものとその効果を学びましょう。

24.1.7 `turtledemo` --- デモスクリプト

'turtledemo'パッケージには一連のデモスクリプトが含まれています。これらのスクリプトは以下のように、付属のデモビューアを使用して実行および表示できます:

```
python -m turtledemo
```

あるいは、個別にデモスクリプトを実行できます。たとえば 、:

```
python -m turtledemo.bytedesign
```

'turtledemo'パッケージのディレクトリには次のものが含まれます:

- ソースコードを眺めつつスクリプトを実行できるデモビュワー `__main__.py` 。
- Multiple scripts demonstrating different features of the *turtle* module. Examples can be accessed via the Examples menu. They can also be run standalone.
- 設定ファイルの書き方や使い方の例として参考にできる `turtle.cfg` ファイル。

デモスクリプトは以下の通りです:

名前	説明	フィーチャー
bytedesign	複雑な古典的タートルグラフィックスパターン	`tracer()`, delay, `update()`
chaos	verhust 力学系のグラフ化, コンピュータの計算が常識的な予想に反する場合があることを示します。	世界座標系
clock	コンピュータの時間を示すアナログ時計	タートルが時計の針, ontimer
colormixer	r, g, b の実験	`ondrag()`
forest	3 breadth-first trees	randomization
fractalcurves	Hilbert & Koch 曲線	再帰
lindenmayer	民俗的数学 (インド kolams)	L-システム
minimal_hanoi	ハノイの塔	ハノイ盤として正方形のタートル (shape, shapesize)
nim	play the classical nim game with three heaps of sticks against the computer.	turtles as nimsticks, event driven (mouse, keyboard)
paint	超極小主義的描画プログラム	`onclick()`
peace	初歩的	turtle: 見た目とアニメーション
penrose	凧と矢による非周期的タイリング	`stamp()`
planet_and_moon	重力系のシミュレーション	合成形, `Vec2D`
round_dance	dancing turtles rotating pairwise in opposite direction	compound shapes, clone shapesize, tilt, get_shapepoly, update
sorting_animate	visual demonstration of different sorting methods	simple alignment, randomization
tree	(図形的) 幅優先木 (ジェネレータを使って)	`clone()`
two_canvases	simple design	turtles on two canvases
wikipedia	タートルグラフィックスについての wikipedia の記事の例	`clone()`, `undo()`
yinyang	もう一つの初歩的な例	`circle()`

楽しんでね!

24.1.8 python 2.6 からの変更点

- The methods `Turtle.tracer()`, `Turtle.window_width()` and `Turtle.window_height()` have been eliminated. Methods with these names and functionality are now available only as methods of `Screen`. The functions derived from these remain available. (In fact already in Python 2.6 these methods were merely duplications of the corresponding `TurtleScreen`/`Screen`-methods.)
- The method `Turtle.fill()` has been eliminated. The behaviour of `begin_fill()` and `end_fill()` have changed slightly: now every filling-process must be completed with an `end_fill()` call.

- A method `Turtle.filling()` has been added. It returns a boolean value: `True` if a filling process is under way, `False` otherwise. This behaviour corresponds to a `fill()` call without arguments in Python 2.6.

24.1.9 python 3.0 からの変更点

- The methods `Turtle.shearfactor()`, `Turtle.shapetransform()` and `Turtle.get_shapepoly()` have been added. Thus the full range of regular linear transforms is now available for transforming turtle shapes. `Turtle.tiltangle()` has been enhanced in functionality: it now can be used to get or set the tiltangle. `Turtle.settiltangle()` has been deprecated.
- The method `Screen.onkeypress()` has been added as a complement to `Screen.onkey()` which in fact binds actions to the keyrelease event. Accordingly the latter has got an alias: `Screen.onkeyrelease()`.
- The method `Screen.mainloop()` has been added. So when working only with Screen and Turtle objects one must not additionally import `mainloop()` anymore.
- Two input methods has been added `Screen.textinput()` and `Screen.numinput()`. These popup input dialogs and return strings and numbers respectively.
- Two example scripts `tdemo_nim.py` and `tdemo_round_dance.py` have been added to the `Lib/turtledemo` directory.

24.2 cmd --- 行指向のコマンドインタープリタのサポート

ソースコード: Lib/cmd.py

Cmd クラスでは、行指向のコマンドインタープリタを書くための簡単なフレームワークを提供します。テストハーネスや管理ツール、そして、後により洗練されたインターフェイスでラップするプロトタイプとして、こうしたインタープリタはよく役に立ちます。

class cmd.**Cmd**(*completekey='tab'*, *stdin=None*, *stdout=None*)

Cmd インスタンス、あるいはサブクラスのインスタンスは、行指向のインタープリタ・フレームワークです。*Cmd* 自身をインスタンス化することはありません。むしろ、*Cmd* のメソッドを継承したり、アクションメソッドをカプセル化するために、あなたが自分で定義するインタープリタクラスのスーパークラスとしての便利です。

オプション引数 *completekey* は、補完キーの *readline* 名です。デフォルトは `Tab` です。*completekey* が *None* でなく、*readline* が利用できるならば、コマンド補完は自動的に行われます。

オプション引数の *stdin* と *stdout* には、Cmd またはそのサブクラスのインスタンスが入出力に使用するファイルオブジェクトを指定します。省略時には *sys.stdin* と *sys.stdout* が使用されます。

引数に渡した *stdin* を使いたい場合は、インスタンスの *use_rawinput* 属性を False にセットしてください。そうしないと *stdin* は無視されます。

24.2.1 Cmd オブジェクト

Cmd インスタンスは、次のメソッドを持ちます:

Cmd.cmdloop(*intro=None*)

プロンプトを繰り返し出力し、入力を受け取り、受け取った入力から取り去った先頭の語を解析し、その行の残りを引数としてアクションメソッドへディスパッチします。

オプションの引数は、最初のプロンプトの前に表示されるバナーあるいはイントロ用の文字列です (これはクラス属性 *intro* をオーバーライドします)。

readline モジュールがロードされているなら、入力は自動的に bash のような履歴リスト編集機能を受け継ぎます (例えば、Control-P は直前のコマンドへのスクロールバック、Control-N は次のものへ進む、Control-F はカーソルを右へ非破壊的に進める、Control-B はカーソルを非破壊的に左へ移動させる等)。

入力のファイル終端は、文字列 'EOF' として渡されます。

メソッド do_foo() を持っている場合に限って、インタープリタのインスタンスはコマンド名 foo を認識します。特別な場合として、文字 '?' で始まる行はメソッド do_help() へディスパッチします。他の特別な場合として、文字 '!' で始まる行はメソッド do_shell() へディスパッチします (このようなメソッドが定義されている場合)。

このメソッドは *postcmd()* メソッドが真を返したときに return します。*postcmd()* に対する *stop* 引数は、このコマンドが対応する do_*() メソッドからの返り値です。

補完が有効になっているなら、コマンドの補完が自動的に行われます。また、コマンド引数の補完は、引数 *text*, *line*, *begidx*, および *endidx* と共に complete_foo() を呼び出すことによって行われます。*text* は、マッチしようとしている文字列の先頭の語です。返されるマッチは全てそれで始まっていなければなりません。*line* は始めの空白を除いた現在の入力行です。*begidx* と *endidx* は先頭のテキストの始まりと終わりのインデックスで、引数の位置に依存した異なる補完を提供するのに使えます。

Cmd のすべてのサブクラスは、定義済みの do_help() を継承します。このメソッドは、(引数 'bar' と共に呼ばれたとすると) 対応するメソッド help_bar() を呼び出します。そのメソッドが存在しない場合、do_bar() の docstring があればそれを表示します。引数がなければ、do_help() は、すべての利用可能なヘルプ見出し (すなわち、対応する help_*() メソッドを持つすべてのコマンドまたは docstring を持つコマンド) をリストアップします。また、文書化されていないコマンドでも、すべてリストアップします。

Cmd.onecmd(*str*)

プロンプトに答えてタイプしたかのように引数を解釈実行します。これをオーバーライドすることがあるかもしれませんが、通常は必要ないでしょう。便利な実行フックについては、*precmd()* と *postcmd()* メソッドを参照してください。戻り値は、インタープリタによるコマンドの解釈実行をやめるかどうかを示すフラグです。コマンド *str* に対応する do_*() メソッドがある場合、そのメソッド

の返り値が返されます。そうでない場合は *default()* メソッドからの返り値が返されます。

Cmd.emptyline()
: プロンプトに空行が入力されたときに呼び出されるメソッド。このメソッドがオーバーライドされていないなら、最後に入力された空行でないコマンドが繰り返されます。

Cmd.default(*line*)
: コマンドの先頭の語が認識されないときに、入力行に対して呼び出されます。このメソッドがオーバーライドされていないなら、エラーメッセージを表示して戻ります。

Cmd.completedefault(*text*, *line*, *begidx*, *endidx*)
: 利用可能なコマンド固有の `complete_*()` が存在しないときに、入力行を補完するために呼び出されるメソッド。デフォルトでは、空行を返します。

Cmd.precmd(*line*)
: コマンド行 *line* が解釈実行される直前、しかし入力プロンプトが作られ表示された後に実行されるフックメソッド。このメソッドは *Cmd* 内のスタブであって、サブクラスでオーバーライドされるために存在します。戻り値は *onecmd()* メソッドが実行するコマンドとして使われます。*precmd()* の実装では、コマンドを書き換えるかもしれないし、あるいは単に変更していない *line* を返すかもしれません。

Cmd.postcmd(*stop*, *line*)
: コマンドディスパッチが終わった直後に実行されるフックメソッド。このメソッドは *Cmd* 内のスタブで、サブクラスでオーバーライドされるために存在します。*line* は実行されたコマンド行で、*stop* は *postcmd()* の呼び出しの後に実行を停止するかどうかを示すフラグです。これは *onecmd()* メソッドの戻り値です。このメソッドの戻り値は、*stop* に対応する内部フラグの新しい値として使われます。偽を返すと、実行を続けます。

Cmd.preloop()
: *cmdloop()* が呼び出されたときに一度だけ実行されるフックメソッド。このメソッドは *Cmd* 内のスタブであって、サブクラスでオーバーライドされるために存在します。

Cmd.postloop()
: *cmdloop()* が戻る直前に一度だけ実行されるフックメソッド。このメソッドは *Cmd* 内のスタブであって、サブクラスでオーバーライドされるために存在します。

Cmd のサブクラスのインスタンスは、公開されたインスタンス変数をいくつか持っています:

Cmd.prompt
: 入力を求めるために表示されるプロンプト。

Cmd.identchars
: コマンドの先頭の語として受け入れられる文字の文字列。

Cmd.lastcmd
: 最後の空でないコマンド接頭辞。

Cmd.cmdqueue
: キューに入れられた入力行のリスト。cmdqueue リストは新たな入力が必要な際に *cmdloop()* 内でチェックされます; これが空でない場合、その要素は、あたかもプロンプトから入力されたかのように

順に処理されます。

Cmd.intro
: イントロあるいはバナーとして表示される文字列。*cmdloop()* メソッドに引数を与えるために、オーバーライドされるかもしれません。

Cmd.doc_header
: ヘルプ出力に文書化されたコマンドのセクションがある場合に表示するヘッダ。

Cmd.misc_header
: ヘルプの出力にその他のヘルプ見出しがある (すなわち、do_*() メソッドに対応していない help_*() メソッドが存在する) 場合に表示するヘッダ。

Cmd.undoc_header
: ヘルプ出力に文書化されていないコマンドのセクションがある (すなわち、対応する help_*() メソッドを持たない do_*() メソッドが存在する) 場合に表示するヘッダ。

Cmd.ruler
: ヘルプメッセージのヘッダの下に、区切り行を表示するために使われる文字。空のときは、ルーラ行が表示されません。デフォルトでは、'=' です。

Cmd.use_rawinput
: フラグで、デフォルトでは真です。真ならば、*cmdloop()* はプロンプトを表示して次のコマンド読み込むために *input()* を使います。偽ならば、sys.stdout.write() と sys.stdin.readline() が使われます。(これが意味するのは、*readline* を import することによって、それをサポートするシステム上では、インタープリタが自動的に **Emacs** 形式の行編集とコマンド履歴のキーストロークをサポートするということです。)

24.2.2 Cmd の例

cmd モジュールは、ユーザーがプログラムと対話的に連携できるカスタムシェルを構築するのに主に役立ちます。

この節では、*turtle* モジュールのいくつかのコマンドを持ったシェルの作成方法の簡単な例を示します。

forward() のような基本的な turtle コマンドは *Cmd* のサブクラスに do_forward() と名付けられたメソッドで追加されます。引数は数値に変換され、turtle モジュールに振り分けられます。ドキュメント文字列はシェルによって提供されるヘルプユーティリティで使用されます。

この例には、基本的な記録機能と再実行機能が含まれていて、入力を小文字に変換しコマンドをファイルに書き込む責務を持たせた *precmd()* メソッドに実装されています。do_playback() メソッドはファイルを読み込み、そこに記録されているコマンドをすぐに再実行するために cmdqueue に追加します:

```
import cmd, sys
from turtle import *

class TurtleShell(cmd.Cmd):
    intro = 'Welcome to the turtle shell.   Type help or ? to list commands.\n'
```

(次のページに続く)

(前のページからの続き)

```
    prompt = '(turtle) '
    file = None

    # ----- basic turtle commands -----
    def do_forward(self, arg):
        'Move the turtle forward by the specified distance:  FORWARD 10'
        forward(*parse(arg))
    def do_right(self, arg):
        'Turn turtle right by given number of degrees:  RIGHT 20'
        right(*parse(arg))
    def do_left(self, arg):
        'Turn turtle left by given number of degrees:  LEFT 90'
        left(*parse(arg))
    def do_goto(self, arg):
        'Move turtle to an absolute position with changing orientation.  GOTO 100 200'
        goto(*parse(arg))
    def do_home(self, arg):
        'Return turtle to the home position:  HOME'
        home()
    def do_circle(self, arg):
        'Draw circle with given radius an options extent and steps:  CIRCLE 50'
        circle(*parse(arg))
    def do_position(self, arg):
        'Print the current turtle position:  POSITION'
        print('Current position is %d %d\n' % position())
    def do_heading(self, arg):
        'Print the current turtle heading in degrees:  HEADING'
        print('Current heading is %d\n' % (heading(),))
    def do_color(self, arg):
        'Set the color:  COLOR BLUE'
        color(arg.lower())
    def do_undo(self, arg):
        'Undo (repeatedly) the last turtle action(s):  UNDO'
    def do_reset(self, arg):
        'Clear the screen and return turtle to center:  RESET'
        reset()
    def do_bye(self, arg):
        'Stop recording, close the turtle window, and exit:  BYE'
        print('Thank you for using Turtle')
        self.close()
        bye()
        return True

    # ----- record and playback -----
    def do_record(self, arg):
        'Save future commands to filename:  RECORD rose.cmd'
        self.file = open(arg, 'w')
    def do_playback(self, arg):
        'Playback commands from a file:  PLAYBACK rose.cmd'
        self.close()
        with open(arg) as f:
```

(次のページに続く)

(前のページからの続き)

```
            self.cmdqueue.extend(f.read().splitlines())
    def precmd(self, line):
        line = line.lower()
        if self.file and 'playback' not in line:
            print(line, file=self.file)
        return line
    def close(self):
        if self.file:
            self.file.close()
            self.file = None

def parse(arg):
    'Convert a series of zero or more numbers to an argument tuple'
    return tuple(map(int, arg.split()))

if __name__ == '__main__':
    TurtleShell().cmdloop()
```

以下は、turtle シェルでの機能のヘルプ表示、空行によるコマンドの繰り返し、単純な記録と再実行のセッション例です:

```
Welcome to the turtle shell.   Type help or ? to list commands.

(turtle) ?

Documented commands (type help <topic>):
========================================
bye     color    goto     home  playback  record  right
circle  forward  heading  left  position  reset   undo

(turtle) help forward
Move the turtle forward by the specified distance:  FORWARD 10
(turtle) record spiral.cmd
(turtle) position
Current position is 0 0

(turtle) heading
Current heading is 0

(turtle) reset
(turtle) circle 20
(turtle) right 30
(turtle) circle 40
(turtle) right 30
(turtle) circle 60
(turtle) right 30
(turtle) circle 80
(turtle) right 30
(turtle) circle 100
(turtle) right 30
(turtle) circle 120
```

(次のページに続く)

(前のページからの続き)

```
(turtle) right 30
(turtle) circle 120
(turtle) heading
Current heading is 180

(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 500
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 300
(turtle) playback spiral.cmd
Current position is 0 0

Current heading is 0

Current heading is 180

(turtle) bye
Thank you for using Turtle
```

24.3 shlex --- 単純な字句解析

ソースコード: Lib/shlex.py

---

shlex クラスは Unix シェルに似た、単純な構文に対する字句解析器を簡単に書けるようにします。このクラスはしばしば、Python アプリケーションのための実行制御ファイルのような小規模言語を書く上で便利です。

shlex モジュールは以下の関数を定義しています:

shlex.**split**(*s*, *comments=False*, *posix=True*)

シェルに似た文法を使って、文字列 *s* を分割します。*comments* が *False* (デフォルト値) の場合、受理した文字列内のコメントを解析しません (*shlex* インスタンスの *commenters* メンバの値を空文字列にします)。この関数はデフォルトでは POSIX モードで動作し、*posix* 引数が偽の場合は非 POSIX モードで動作します。

注釈: *split()* 関数は *shlex* クラスのインスタンスを利用するので、*s* に None を渡すと標準入力から分割する文字列を読み込みます。

---

shlex.join(*split_command*)

Concatenate the tokens of the list *split_command* and return a string. This function is the inverse of *split()*.

```
>>> from shlex import join
>>> print(join(['echo', '-n', 'Multiple words']))
echo -n 'Multiple words'
```

The returned value is shell-escaped to protect against injection vulnerabilities (see *quote()*).

バージョン 3.8 で追加.

shlex.quote(*s*)

文字列 *s* をシェルエスケープして返します。戻り値は、リストを使えないようなケースで、シェルコマンドライン内で一つのトークンとして安全に利用出来る文字列です。

以下のイディオムは安全ではないかもしれません:

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print(command)  # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

quote() がそのセキュリティホールをふさぎます:

```
>>> from shlex import quote
>>> command = 'ls -l {}'.format(quote(filename))
>>> print(command)
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print(remote_command)
ssh home 'ls -l '"'"'somefile; rm -rf ~'"'"''
```

クォーティングは UNIX シェルならびに *split()* と互換です:

```
>>> from shlex import split
>>> remote_command = split(remote_command)
>>> remote_command
['ssh', 'home', "ls -l 'somefile; rm -rf ~'"]
>>> command = split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

バージョン 3.3 で追加.

shlex モジュールは以下のクラスを定義します。

class shlex.shlex(*instream=None*, *infile=None*, *posix=False*, *punctuation_chars=False*)

A *shlex* instance or subclass instance is a lexical analyzer object. The initialization argument, if present, specifies where to read characters from. It must be a file-/stream-like object with *read()* and *readline()* methods, or a string. If no argument is given, input will be taken from

sys.stdin. The second optional argument is a filename string, which sets the initial value of the *infile* attribute. If the *instream* argument is omitted or equal to sys.stdin, this second argument defaults to "stdin". The *posix* argument defines the operational mode: when *posix* is not true (default), the *shlex* instance will operate in compatibility mode. When operating in POSIX mode, *shlex* will try to be as close as possible to the POSIX shell parsing rules. The *punctuation_chars* argument provides a way to make the behaviour even closer to how real shells parse. This can take a number of values: the default value, False, preserves the behaviour seen under Python 3.5 and earlier. If set to True, then parsing of the characters ();<>|& is changed: any run of these characters (considered punctuation characters) is returned as a single token. If set to a non-empty string of characters, those characters will be used as the punctuation characters. Any characters in the *wordchars* attribute that appear in *punctuation_chars* will be removed from *wordchars*. See *Improved Compatibility with Shells* for more information. *punctuation_chars* can be set only upon *shlex* instance creation and can't be modified later.

バージョン 3.6 で変更: *punctuation_chars* 引数が追加されました。

参考:

configparser **モジュール** Windows .ini ファイルに似た設定ファイルのパーザ。

24.3.1 shlex オブジェクト

shlex インスタンスは以下のメソッドを持っています:

shlex.**get_token**()
トークンを一つ返します。トークンが *push_token()* で使ってスタックに積まれていた場合、トークンをスタックからポップします。そうでない場合、トークンを一つ入力ストリームから読み出します。読み出し即時にファイル終了子に遭遇した場合、*eof* (非 POSIX モードでは空文字列 (''), POSIX モードでは None) が返されます。

shlex.**push_token**(*str*)
トークンスタックに引数文字列をスタックします。

shlex.**read_token**()
生 (raw) のトークンを読み出します。プッシュバックスタックを無視し、かつソースリクエストを解釈しません (通常これは便利なエントリポイントではありません。完全性のためにここで記述されています)。

shlex.**sourcehook**(*filename*)
shlex がソースリクエスト (下の *source* を参照してください) を検出した際、このメソッドはその後に続くトークンを引数として渡され、ファイル名と開かれたファイル類似オブジェクトからなるタプルを返すとされています。

通常、このメソッドはまず引数から何らかのクオートを取り除きます。処理後の引数が絶対パス名であった場合か、以前に有効になったソースリクエストが存在しない場合か、以前のソースが (sys.stdin のような) ストリームであった場合、この結果はそのままにされます。そうでない場合で、処理後の引数が相対パス名の場合、ソースインクルードスタックにある直前のファイル名からディレクトリ部分が取

り出され、相対パスの前の部分に追加されます (この動作は C 言語プリプロセッサにおける `#include "file.h"` の扱いと同様です)。

これらの操作の結果はファイル名として扱われ、タプルの最初の要素として返されます。同時にこのファイル名で *open()* を呼び出した結果が二つ目の要素になります (注意: インスタンス初期化のときとは引数の並びが逆になっています！)

このフックはディレクトリサーチパスや、ファイル拡張子の追加、その他の名前空間に関するハックを実装できるようにするために公開されています。'close' フックに対応するものはありませんが、shlex インスタンスはソースリクエストされている入力ストリームが EOF を返した時には *close()* を呼び出します。

ソーススタックをより明示的に操作するには、*push_source()* および *pop_source()* メソッドを使ってください。

shlex.push_source(*newstream*, *newfile=None*)
入力ソースストリームを入力スタックにプッシュします。ファイル名引数が指定された場合、以後のエラーメッセージ中で利用することができます。*sourcehook()* メソッドが内部で使用しているのと同じメソッドです。

shlex.pop_source()
最後にプッシュされた入力ソースを入力スタックからポップします。字句解析器がスタック上の入力ストリームの EOF に到達した際に利用するメソッドと同じです。

shlex.error_leader(*infile=None*, *lineno=None*)
このメソッドはエラーメッセージの論述部分を Unix C コンパイラエラーラベルの形式で生成します; この書式は `'"%s", line %d: '` で、`%s` は現在のソースファイル名で置き換えられ、`%d` は現在の入力行番号で置き換えられます (オプションの引数を使ってこれらを上書きすることもできます)。

このやり方は、*shlex* のユーザに対して、Emacs やその他の Unix ツール群が解釈できる一般的な書式でのメッセージを生成することを推奨するために提供されています。

shlex サブクラスのインスタンスは、字句解析を制御したり、デバッグに使えるような public なインスタンス変数を持っています:

shlex.commenters
コメントの開始として認識される文字列です。コメントの開始から行末までのすべてのキャラクタ文字は無視されます。標準では単に `'#'` が入っています。

shlex.wordchars
The string of characters that will accumulate into multi-character tokens. By default, includes all ASCII alphanumerics and underscore. In POSIX mode, the accented characters in the Latin-1 set are also included. If *punctuation_chars* is not empty, the characters `~-./*?=`, which can appear in filename specifications and command line parameters, will also be included in this attribute, and any characters which appear in `punctuation_chars` will be removed from `wordchars` if they are present there. If *whitespace_split* is set to `True`, this will have no effect.

shlex.whitespace
空白と見なされ、読み飛ばされる文字群です。空白はトークンの境界を作ります。標準では、スペース、

タブ、改行 (linefeed) および復帰 (carriage-return) が入っています。

shlex.escape
エスケープ文字と見なされる文字群です。これは POSIX モードでのみ使われ、デフォルトでは '\' だけが入っています。

shlex.quotes
文字列引用符と見なされる文字群です。トークンを構成する際、同じクオートが再び出現するまで文字をバッファに蓄積します (すなわち、異なるクオート形式はシェル中で互いに保護し合う関係にあります)。標準では、ASCII 単引用符および二重引用符が入っています。

shlex.escapedquotes
quotes のうち、*escape* で定義されたエスケープ文字を解釈する文字群です。これは POSIX モードでのみ使われ、デフォルトでは '"' だけが入っています。

shlex.whitespace_split
If `True`, tokens will only be split in whitespaces. This is useful, for example, for parsing command lines with *shlex*, getting tokens in a similar way to shell arguments. When used in combination with *punctuation_chars*, tokens will be split on whitespace in addition to those characters.

バージョン 3.8 で変更: The *punctuation_chars* attribute was made compatible with the *whitespace_split* attribute.

shlex.infile
現在の入力ファイル名です。クラスのインスタンス化時に初期設定されるか、その後のソースリクエストでスタックされます。エラーメッセージを構成する際にこの値を調べると便利なことがあります。

shlex.instream
shlex インスタンスが文字を読み出している入力ストリームです。

shlex.source
このメンバ変数は標準で *None* を取ります。この値に文字列を代入すると、その文字列は多くのシェルにおける `source` キーワードに似た、字句解析レベルでのインクルード要求として認識されます。すなわち、その直後に現れるトークンをファイル名として新たなストリームを開き、そのストリームを入力として、EOF に到達するまで読み込まれます。新たなストリームの EOF に到達した時点で *close()* が呼び出され、入力は元の入力ストリームに戻されます。ソースリクエストは任意のレベルの深さまでスタックしてかまいません。

shlex.debug
このメンバ変数が数値で、かつ 1 またはそれ以上の値の場合、*shlex* インスタンスは動作に関する冗長な進捗報告を出力します。この出力を使いたいなら、モジュールのソースコードを読めば詳細を学ぶことができます。

shlex.lineno
ソース行番号 (遭遇した改行の数に 1 を加えたもの) です。

shlex.token
トークンバッファです。例外を捕捉した際にこの値を調べると便利なことがあります。

shlex.eof
: ファイルの終端を決定するのに使われるトークンです。非 POSIX モードでは空文字列 (`''`) 、POSIX モードでは `None` が入ります。

shlex.punctuation_chars
: A read-only property. Characters that will be considered punctuation. Runs of punctuation characters will be returned as a single token. However, note that no semantic validity checking will be performed: for example, '»>' could be returned as a token, even though it may not be recognised as such by shells.

  バージョン 3.6 で追加.

24.3.2 解析規則

非 POSIX モードで動作中の *shlex* は以下の規則に従おうとします。

- ワード内の引用符を認識しない (`Do"Not"Separate` は単一ワード `Do"Not"Separate` として解析されます)
- エスケープ文字を認識しない
- 引用符で囲まれた文字列は、引用符内の全ての文字リテラルを保持する
- 閉じ引用符でワードを区切る (`"Do"Separate` は、`"Do"` と `Separate` であると解析されます)
- *whitespace_split* が `False` の場合、wordchar、whitespace または quote として宣言されていない全ての文字を、単一の文字トークンとして返す。`True` の場合、*shlex* は空白文字でのみ単語を区切る。
- 空文字列 (`''`) で EOF を送出する
- 引用符に囲んであっても、空文字列を解析しない

POSIX モードで動作中の *shlex* は以下の解析規則に従おうとします。

- 引用符を取り除き、引用符で単語を分解しない (`"Do"Not"Separate"` は単一ワード `DoNotSeparate` として解析されます)
- 引用符で囲まれないエスケープ文字群 (`'\'` など) は直後に続く文字のリテラル値を保持する
- *escapedquotes* でない引用符文字 (`'"'` など) で囲まれている全ての文字のリテラル値を保持する
- 引用符に囲まれた *escapedquotes* に含まれる文字 (`'"'` など) は、*escape* に含まれる文字を除き、全ての文字のリテラル値を保持する。エスケープ文字群は使用中の引用符、または、そのエスケープ文字自身が直後にある場合のみ、特殊な機能を保持する。他の場合にはエスケープ文字は普通の文字とみなされる。
- *None* で EOF を送出する
- 引用符に囲まれた空文字列 (`''`) を許す。

24.3.3 Improved Compatibility with Shells

バージョン 3.6 で追加.

The *shlex* class provides compatibility with the parsing performed by common Unix shells like `bash`, `dash`, and `sh`. To take advantage of this compatibility, specify the `punctuation_chars` argument in the constructor. This defaults to `False`, which preserves pre-3.6 behaviour. However, if it is set to `True`, then parsing of the characters `();<>|&` is changed: any run of these characters is returned as a single token. While this is short of a full parser for shells (which would be out of scope for the standard library, given the multiplicity of shells out there), it does allow you to perform processing of command lines more easily than you could otherwise. To illustrate, you can see the difference in the following snippet:

```
>>> import shlex
>>> text = "a && b; c && d || e; f >'abc'; (def \"ghi\")"
>>> s = shlex.shlex(text, posix=True)
>>> s.whitespace_split = True
>>> list(s)
['a', '&&', 'b;', 'c', '&&', 'd', '||', 'e;', 'f', '>abc;', '(def', 'ghi)']
>>> s = shlex.shlex(text, posix=True, punctuation_chars=True)
>>> s.whitespace_split = True
>>> list(s)
['a', '&&', 'b', ';', 'c', '&&', 'd', '||', 'e', ';', 'f', '>', 'abc', ';',
'(', 'def', 'ghi', ')']
```

Of course, tokens will be returned which are not valid for shells, and you'll need to implement your own error checks on the returned tokens.

Instead of passing `True` as the value for the punctuation_chars parameter, you can pass a string with specific characters, which will be used to determine which characters constitute punctuation. For example:

```
>>> import shlex
>>> s = shlex.shlex("a && b || c", punctuation_chars="|")
>>> list(s)
['a', '&', '&', 'b', '||', 'c']
```

注釈: When `punctuation_chars` is specified, the *wordchars* attribute is augmented with the characters `~-./*?=`. That is because these characters can appear in file names (including wildcards) and command-line arguments (e.g. `--color=auto`). Hence:

```
>>> import shlex
>>> s = shlex.shlex('~/a && b-c --color=auto || d *.py?',
...                 punctuation_chars=True)
>>> list(s)
['~/a', '&&', 'b-c', '--color=auto', '||', 'd', '*.py?']
```

However, to match the shell as closely as possible, it is recommended to always use `posix` and

whitespace_split when using *punctuation_chars*, which will negate *wordchars* entirely.

For best effect, `punctuation_chars` should be set in conjunction with `posix=True`. (Note that `posix=False` is the default for *shlex*.)

第

TWENTYFIVE

TK を用いたグラフィカルユーザインターフェイス

Tk/Tcl は長きにわたり Python の不可欠な一部でありつづけています。Tk/Tcl は頑健でプラットホームに依存しないウィンドウ構築ツールキットであり、Python プログラマは *tkinter* パッケージやその拡張の *tkinter.tix* および *tkinter.ttk* モジュールを使って利用できます。

tkinter パッケージは Tcl/Tk 上に作られた軽量なオブジェクト指向のレイヤです。*tkinter* を使うために Tcl コードを書く必要はありませんが、Tk のドキュメントや、場合によっては Tcl のドキュメントを調べる必要があるでしょう。*tkinter* は Tk のウィジェットを Python のクラスとして実装しているラッパをまとめたものです。加えて、内部モジュール `_tkinter` は、Python と Tcl とがやり取りできるスレッドセーフなメカニズムを提供しています。

tkinter の一番素晴らしい点は、速く、そしてほとんどの Python に付属していることです。標準ドキュメントが頼りないものだとしても、代わりとなる、リファレンス、チュートリアル、書籍その他が入手可能です。*tkinter* は古臭いルックアンドフィールでも有名ですが、その点は Tk 8.5 で大きく改善されました。とはいえ、他にも興味を引きそうな GUI ライブラリは多数あります。それらについてのより詳しい情報は **他のグラフィカルユーザインタフェースパッケージ** 節を参照してください。

25.1 `tkinter` --- Tcl/Tk の Python インタフェース

ソースコード: Lib/tkinter/___init___.py

---

tkinter パッケージ ("Tk インタフェース") は、Tk GUI ツールキットに対する標準の Python インタフェースです。Tk と *tkinter* はほとんどの Unix プラットフォームの他、Windows システム上でも利用できます。(Tk 自体は Python の一部ではありません。Tk は ActiveState で保守されています。)

コマンドラインから `python -m tkinter` を実行すると簡素な Tk インターフェースを表示するウィンドウが開き、システムに *tkinter* が正しくインストールされたことが分かり、さらにインストールされた Tcl/Tk がどのバーションなのかが表示されるので、そのバージョンの Tcl/Tk ドキュメントを選んで読めます。

参考:

Tkinter ドキュメント:

Python Tkinter Resources　Python Tkinter Topic Guide では、Tk を Python から利用する上での情報と、その他の Tk にまつわる情報源を数多く提供しています。

TKDocs　豊富なチュートリアルといくつかのウィジットについてのよりフレンドリなページ。

Tkinter 8.5 reference: a GUI for Python　オンラインリファレンス資料です。

Tkinter docs from effbot　effbot.org が提供している tkinter のオンラインリファレンスです。

Programming Python　マーク・ルッツによる書籍で、Tkinter についても広く取り上げています。

Modern Tkinter for Busy Python Developers　Mark Roseman による書籍で、Python と Tkinter を使用した魅力的でモダンなグラフィカルユーザーインターフェースの構築方法を説明しています。

Python and Tkinter Programming　John Grayson による解説書 (ISBN 1-884777-81-3) です。

Tcl/Tk ドキュメント:

Tk commands　ほとんどのコマンドは *tkinter* あるいは *tkinter.ttk* クラスから利用可能です。手元のインストールされている Tcl/Tk のバージョンに合うように '8.6' の部分を読み替えてください。

Tcl/Tk の最近の man pages　www.tcl.tk の最近の Tcl/Tk マニュアルです。

ActiveState Tcl ホームページ　Tk/Tcl の開発は ActiveState で大々的に行われています。

Tcl and the Tk Toolkit　Tcl の発明者である John Ousterhout による本です。

Practical Programming in Tcl and Tk　Brent Welch の百科事典のような本です。

25.1.1 Tkinter モジュール

ほとんどの場合、本当に必要となるのは *tkinter* モジュールだけですが、他にもいくつかの追加モジュールを利用できます。Tk インタフェース自体はバイナリモジュール `_tkinter` 内にあります。このモジュールに入っているのは Tk への低レベルインタフェースであり、アプリケーションプログラマが直接使ってはなりません。`_tkinter` は通常共有ライブラリ (や DLL) になっていますが、Python インタプリタに静的にリンクされていることもあります。

Tk インタフェースモジュールの他にも *tkinter* には数多くの Python モジュールが入っています。その中でも `tkinter.constants` は最も重要なモジュールの一つですが、*tkinter* をインポートすると `tkinter.constants` は自動的にインポートされるので、以下のように *tkinter* をインポートするだけで Tkinter を使えるようになります:

```
import tkinter
```

あるいは、よく使うやり方で以下のようにします:

```
from tkinter import *
```

class `tkinter.Tk`(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=1*)

Tk クラスは引数なしでインスタンス化します。これは Tk のトップレベルウィジェットを生成します。通常、トップレベルウィジェットはアプリケーションのメインウィンドウになります。それぞれのインスタンスごとに固有の Tcl インタプリタが関連づけられます。

tkinter.Tcl(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=0*)
Tcl() はファクトリ関数で、*Tk* クラスで生成するオブジェクトとよく似たオブジェクトを生成します。ただし Tk サブシステムを初期化しません。この関数は、余分なトップレベルウィンドウを作る必要がなかったり、(X サーバを持たない Unix/Linux システムなどのように) 作成できない環境において Tcl インタプリタを駆動したい場合に便利です。*Tcl()* で生成したオブジェクトに対して `loadtk()` メソッドを呼び出せば、トップレベルウィンドウを作成 (して、Tk サブシステムを初期化) します。

Tk をサポートしているモジュールには、他にも以下のようなモジュールがあります:

tkinter.scrolledtext 垂直スクロールバー付きのテキストウィジェットです。

`tkinter.colorchooser` ユーザに色を選択させるためのダイアログです。

`tkinter.commondialog` このリストの他のモジュールが定義しているダイアログの基底クラスです。

`tkinter.filedialog` ユーザが開きたいファイルや保存したいファイルを指定できるようにする共通のダイアログです。

`tkinter.font` フォントの扱いを補助するためのユーティリティです。

`tkinter.messagebox` 標準的な Tk のダイアログボックスにアクセスします。

`tkinter.simpledialog` 基本的なダイアログと便宜関数 (convenience function) です。

`tkinter.dnd` *tkinter* 用のドラッグアンドドロップのサポートです。実験的なサポートで、Tk DND に置き替わった時点で撤廃されるはずです。

turtle Tk ウィンドウ上でタートルグラフィックスを実現します。

25.1.2 Tkinter お助け手帳

この節は、Tk や Tkinter を全て網羅したチュートリアルを目指しているわけではありません。むしろ、Tkinter のシステムを学ぶ上での指針を示すための、その場しのぎ的なマニュアルです。

謝辞:

- Tk は John Ousterhout が Berkeley の在籍中に作成しました。
- Tkinter は Steen Lumholt と Guido van Rossum が作成しました。
- この Life Preserver は Virginia 大学の Matt Conway 他が執筆しました。
- HTML レンダリングやたくさんの編集は、Ken Manheimer が FrameMaker 版から行いました。
- Fredrik Lundh はクラスインタフェース詳細な説明を書いたり内容を改訂したりして、現行の Tk 4.2 に合うようにしました。
- Mike Clarkson はドキュメントを LaTeX 形式に変換し、リファレンスマニュアルのユーザインタフェースの章をコンパイルしました。

この節の使い方

この節は二つの部分で構成されています: 前半では、背景となることがらを (大雑把に) 網羅しています。後半は、キーボードの横に置けるような手軽なリファレンスになっています。

「○○するにはどうしたらよいですか」という形の問いに答えようと思うなら、まず Tk で「○○する」方法を調べてから、このドキュメントに戻ってきてその方法に対応する *tkinter* の関数呼び出しに変換するのが多くの場合最善の方法になります。Python プログラマが Tk ドキュメンテーションを見れば、たいてい正しい Python コマンドの見当をつけられます。従って、Tkinter を使うには Tk についてほんの少しだけ知っていればよいということになります。このドキュメントではその役割を果たせないので、次善の策として、すでにある最良のドキュメントについていくつかヒントを示しておくことにしましょう:

- Tk の man ページのコピーを手に入れるよう強く勧めます。とりわけ最も役立つのは `manN` ディレクトリ内にあるマニュアルです。`man3` のマニュアルページは Tk ライブラリに対する C インタフェースについての説明なので、スクリプト書きにとって取り立てて役に立つ内容ではありません。
- Addison-Wesley は John Ousterhout の書いた Tcl and the Tk Toolkit (ISBN 0-201-63337-X) という名前の本を出版しています。この本は初心者向けの Tcl と Tk の良い入門書です。内容は網羅的ではなく、詳細の多くは man ページ任せにしています。
- たいていの場合、`tkinter/__init__.py` は参照先としての最後の手段ですが、それ以外の手段で調べても分からない場合には救いの地になるかもしれません。

簡単な Hello World プログラム

```
import tkinter as tk

class Application(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.pack()
        self.create_widgets()

    def create_widgets(self):
        self.hi_there = tk.Button(self)
        self.hi_there["text"] = "Hello World\n(click me)"
        self.hi_there["command"] = self.say_hi
        self.hi_there.pack(side="top")

        self.quit = tk.Button(self, text="QUIT", fg="red",
                              command=self.master.destroy)
        self.quit.pack(side="bottom")

    def say_hi(self):
        print("hi there, everyone!")

root = tk.Tk()
app = Application(master=root)
app.mainloop()
```

25.1.3 Tcl/Tk を (本当に少しだけ) 見渡してみる

クラス階層は複雑に見えますが、実際にプログラムを書く際には、アプリケーションプログラマはほとんど常にクラス階層の最底辺にあるクラスしか参照しません。

注釈:

- クラスのいくつかは、特定の関数を一つの名前空間下にまとめるために提供されています。こうしたクラスは個別にインスタンス化するためのものではありません。
- *Tk* クラスはアプリケーション内で一度だけインスタンス化するようになっています。アプリケーションプログラマが明示的にインスタンス化する必要はなく、他のクラスがインスタンス化されると常にシステムが作成します。
- `Widget` クラスもまた、インスタンス化して使うようにはなっていません。このクラスはサブクラス化して「実際の」ウィジェットを作成するためのものです。(C++ で言うところの、' 抽象クラス (abstract class)' です)。

このリファレンス資料を活用するには、Tk の短いプログラムを読んだり、Tk コマンドの様々な側面を知っておく必要がままあるでしょう。(下の説明の *tkinter* 版は、**基本的な *Tk* プログラムと *Tkinter* との対応関係** 節を参照してください。)

Tk スクリプトは Tcl プログラムです。全ての Tcl プログラムに同じく、Tk スクリプトはトークンをスペースで区切って並べます。Tk ウィジェットとは、ウィジェットの **クラス** 、ウィジェットの設定を行う **オプション** 、そしてウィジェットに役立つことをさせる **アクション** を組み合わせたものに過ぎません。

Tk でウィジェットを作るには、常に次のような形式のコマンドを使います:

```
classCommand newPathname options
```

classCommand どの種類のウィジェット (ボタン、ラベル、メニュー、...) を作るかを表します。

newPathname 作成するウィジェットにつける新たな名前です。Tk 内の全ての名前は一意になっていなければなりません。一意性を持たせる助けとして、Tk 内のウィジェットは、ファイルシステムにおけるファイルと同様、**パス名** *(pathname)* を使って名づけられます。トップレベルのウィジェット、すなわち **ルート** は `.` (ピリオド) という名前になり、その子ウィジェット階層もピリオドで区切ってゆきます。ウィジェットの名前は、例えば `.myApp.controlPanel.okButton` のようになります。

options ウィジェットの見た目を設定します。場合によってはウィジェットの挙動も設定します。オプションはフラグと値がリストになった形式をとります。Unix のシェルコマンドのフラグと同じように、フラグの前には '-' がつき、複数の単語からなる値はクオートで囲まれます。

例えば:

```
button   .fred   -fg red -text "hi there"
   ^       ^     \_____________________/
   |       |                |
 class    new            options
command  widget  (-opt val -opt val ...)
```

ウィジェットを作成すると、ウィジェットへのパス名は新しいコマンドになります。この新たな *widget command* は、プログラマが新たに作成したウィジェットに *action* を実行させる際のハンドル (handle) になります。C では someAction(fred, someOptions) と表し、C++ では fred.someAction(someOptions) と表すでしょう。Tk では:

```
.fred someAction someOptions
```

のようにします。オブジェクト名 `.fred` はドットから始まっている点に注意してください。

読者の想像の通り、*someAction* に指定できる値はウィジェットのクラスに依存しています: fred がボタンなら `.fred disable` はうまくいきます (fred はグレーになります) が、fred がラベルならうまくいきません (Tk ではラベルの無効化をサポートしていないからです)。

someOptions に指定できる値はアクションの内容に依存しています。`disable` のようなアクションは引数を必要としませんが、テキストエントリボックスの `delete` コマンドのようなアクションにはテキストを削除する範囲を指定するための引数が必要になります。

25.1.4 基本的な Tk プログラムと Tkinter との対応関係

Tk のクラスコマンドは、Tkinter のクラスコンストラクタに対応しています。

```
button .fred                =====>  fred = Button()
```

オブジェクトの親 (master) は、オブジェクトの作成時に指定した新たな名前から非明示的に決定されます。Tkinter では親を明示的に指定します。

```
button .panel.fred          =====>  fred = Button(panel)
```

Tk の設定オプションは、ハイフンをつけたタグと値の組からなるリストで指定します。Tkinter では、オプションはキーワード引数にしてインスタンスのコンストラクタに指定したり、`config()` にキーワード引数を指定して呼び出したり、インデクス指定を使ってインスタンスに代入したりして設定します。オプションの設定については **オプションの設定** 節を参照してください。

```
button .fred -fg red        =====>  fred = Button(panel, fg="red")
.fred configure -fg red     =====>  fred["fg"] = red
                            OR ==>  fred.config(fg="red")
```

Tk でウィジェットにアクションを実行させるには、ウィジェット名をコマンドにして、その後にアクション名を続け、必要に応じて引数 (オプション) を続けます。Tkinter では、クラスインスタンスのメソッドを呼び出して、ウィジェットのアクションを呼び出します。あるウィジェットがどんなアクション (メソッド) を実行できるかは、`tkinter/__init__.py` モジュール内にリストされています。

```
.fred invoke                =====>  fred.invoke()
```

Tk でウィジェットを packer (ジオメトリマネージャ) に渡すには、pack コマンドをオプション引数付きで呼び出します。Tkinter では Pack クラスがこの機能すべてを握っていて、様々な pack の形式がメソッドとして実装されています。*tkinter* のウィジェットはすべて packer からサブクラス化されているため、pack 操

作にまつわるすべてのメソッドを継承しています。Form ジオメトリマネージャに関する詳しい情報については *tkinter.tix* モジュールのドキュメントを参照してください。

```
pack .fred -side left       =====>  fred.pack(side="left")
```

25.1.5 Tk と Tkinter はどのように関わっているのか

上から下に、呼び出しの階層構造を説明してゆきます:

あなたのアプリケーション (Python)　まず、Python アプリケーションが *tkinter* を呼び出します。

tkinter (Python **パッケージ**)　上記の呼び出し (例えば、ボタンウィジェットの作成) は、*tkinter* パッケージ内で実装されており、Python で書かれています。この Python で書かれた関数は、コマンドと引数を解析して変換し、あたかもコマンドが Python スクリプトではなく Tk スクリプトから来たようにみせかけます。

_tkinter (C)　上記のコマンドと引数は `_tkinter` (先頭にアンダースコアです) 拡張モジュール内の C 関数に渡されます。

Tk **ウィジェット** (C **および** Tcl)　上記の C 関数は、Tk ライブラリを構成する C 関数の入った別の C モジュールへの呼び出しを行えるようになっています。Tk は C と少しの Tcl を使って実装されています。Tk ウィジェットの Tcl 部分は、ウィジェットのデフォルト動作をバインドするために使われ、Python *tkinter* パッケージがインポートされた時に一度だけ実行されます (ユーザがこの過程を目にすることはありません)。

Tk (C)　Tk ウィジェットの Tk 部分で実装されている最終的な対応付け操作によって...

Xlib (C)　Xlib ライブラリがスクリーン上にグラフィックスを描きます。

25.1.6 簡単なリファレンス

オプションの設定

オプションは、色やウィジェットの境界線幅などを制御します。オプションの設定には三通りの方法があります:

オブジェクト生成時、キーワード引数を使用する

```
fred = Button(self, fg="red", bg="blue")
```

オブジェクト生成後、オプション名を辞書インデックスのように扱う

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

オブジェクト生成後に、config() **メソッドを使って複数の属性を更新する**

```
fred.config(fg="red", bg="blue")
```

オプションとその振る舞いに関する詳細な説明は、該当するウィジェットの Tk の man ページを参照してください。

man ページには、各ウィジェットの "STANDARD OPTIONS (標準オプション)" と "WIDGET SPECIFIC OPTIONS (ウィジェット固有のオプション)" がリストされていることに注意してください。前者は多くのウィジェットに共通のオプションのリストで、後者は特定のウィジェットに特有のオプションです。標準オプションの説明は man ページの *options(3)* にあります。

このドキュメントでは、標準オプションとウィジェット固有のオプションを区別していません。オプションによっては、ある種のウィジェットに適用できません。あるウィジェットがあるオプションに対応しているかどうかは、ウィジェットのクラスによります。例えばボタンには `command` オプションがありますが、ラベルにはありません。

あるウィジェットがどんなオプションをサポートしているかは、ウィジェットの man ページにリストされています。また、実行時にウィジェットの `config()` メソッドを引数なしで呼び出したり、`keys()` メソッドを呼び出したりして問い合わせることもできます。メソッド呼び出しを行うと辞書型の値を返します。この辞書は、オプションの名前がキー (例えば `'relief'`) になっていて、値が 5 要素のタプルになっています。

`bg` のように、いくつかのオプションはより長い名前を持つ共通のオプションに対する同義語になっています (`bg` は "background" を短縮したものです)。短縮形のオプション名を `config()` に渡すと、5 要素ではなく 2 要素のタプルを返します。このタプルには、同義語の名前と「本当の」オプション名が入っています (例えば `('bg', 'background')`)。

インデックス	意味	使用例
0	オプション名	`'relief'`
1	データベース検索用のオプション名	`'relief'`
2	データベース検索用のオプションクラス	`'Relief'`
3	デフォルト値	`'raised'`
4	現在の値	`'groove'`

以下はプログラム例です:

```
>>> print(fred.config())
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

もちろん、実際に出力される辞書には利用可能なオプションが全て表示されます。上の表示例は単なる例にすぎません。

Packer

packer は Tk のジオメトリ管理メカニズムの一つです。ジオメトリマネージャは、複数のウィジェットの位置を、それぞれのウィジェットを含むコンテナ - 共通の **マスタ** *(master)* からの相対で指定するために使います。やや扱いにくい *placer* (あまり使われないのでここでは取り上げません) と違い、packer は定性的な関係を表す指定子 - **上** *(above)* 、**〜の左** *(to the left of)* 、**引き延ばし** *(filling)* など - を受け取り、厳密な配置座標の決定を全て行ってくれます。

どんな **マスタ** ウィジェットでも、大きさは内部の ” スレイブ (slave) ウィジェット” の大きさで決まります。packer は、スレイブウィジェットを pack 先のマスタウィジェット中のどこに配置するかを制御するために使われます。望みのレイアウトを達成するには、ウィジェットをフレームにパックし、そのフレームをまた別のフレームにパックできます。さらに、一度パックを行うと、それ以後の設定変更に合わせて動的に並べ方を調整します。

ジオメトリマネージャがウィジェットのジオメトリを確定するまで、ウィジェットは表示されないので注意してください。初心者のころにはよくジオメトリの確定を忘れてしまい、ウィジェットを生成したのに何も表示されず驚くことになります。ウィジェットは、(例えば packer の `pack()` メソッドを適用して) ジオメトリを確定した後で初めて表示されます。

pack() メソッドは、キーワード引数つきで呼び出せます。キーワード引数は、ウィジェットをコンテナ内のどこに表示するか、メインのアプリケーションウィンドウをリサイズしたときにウィジェットがどう振舞うかを制御します。以下に例を示します:

```
fred.pack()                     # defaults to side = "top"
fred.pack(side="left")
fred.pack(expand=1)
```

Packer のオプション

packer と packer の取りえるオプションについての詳細は、man ページや John Ousterhout の本の 183 ページを参照してください。

anchor　アンカーの型です。packer が区画内に各スレイブを配置する位置を示します。

expand　ブール値で、0 または 1 になります。

fill　指定できる値は `'x'` 、`'y'` 、`'both'` 、`'none'` です。

ipadx **および** ipady　スレイブウィジェットの各側面の内側に行うパディング幅を表す長さを指定します。

padx **および** pady　スレイブウィジェットの各側面の外側に行うパディング幅を表す長さを指定します。

side　指定できる値は `'left'`, `'right'`, `'top'`, `'bottom'` です。

ウィジェット変数を関連付ける

ウィジェットによっては、(テキスト入力ウィジェットのように) 特殊なオプションを使って、現在設定されている値をアプリケーション内の変数に直接関連付けできます。このようなオプションには `variable`, `textvariable`, `onvalue`, `offvalue` および `value` があります。この関連付けは双方向に働きます: 変数の値が何らかの理由で変更されると、関連付けされているウィジェットも更新され、新しい値を反映します。

残念ながら、現在の *tkinter* の実装では、`variable` や `textvariable` オプションでは任意の Python の値をウィジェットに渡せません。この関連付け機能がうまく働くのは、*tkinter* 内で Variable というクラスからサブクラス化されている変数によるオプションだけです。

Variable には、`StringVar`, `IntVar`, `DoubleVar`, `BooleanVar` といった便利なサブクラスがすでにすでに数多く定義されています。こうした変数の現在の値を読み出したければ、`get()` メソッドを呼び出します。また、値を変更したければ `set()` メソッドを呼び出します。このプロトコルに従っている限り、それ以上なにも手を加えなくてもウィジェットは常に現在値に追従します。

例えば:

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        self.pack()

        self.entrythingy = tk.Entry()
        self.entrythingy.pack()

        # Create the application variable.
        self.contents = tk.StringVar()
        # Set it to some value.
        self.contents.set("this is a variable")
        # Tell the entry widget to watch this variable.
        self.entrythingy["textvariable"] = self.contents

        # Define a callback for when the user hits return.
        # It prints the current value of the variable.
        self.entrythingy.bind('<Key-Return>',
                              self.print_contents)

    def print_contents(self, event):
        print("Hi. The current entry content is:",
              self.contents.get())

root = tk.Tk()
myapp = App(root)
myapp.mainloop()
```

ウィンドウマネージャ

Tk には、ウィンドウマネジャとやり取りするための wm というユーティリティコマンドがあります。wm コマンドにオプションを指定すると、タイトルや配置、アイコンビットマップなどを操作できます。*tkinter* では、こうしたコマンドは Wm クラスのメソッドとして実装されています。トップレベルウィジェットは Wm クラスからサブクラス化されているので、Wm のメソッドを直接呼び出せます。

あるウィジェットの入っているトップレベルウィンドウを取得したい場合、大抵は単にウィジェットのマスタを参照するだけですみます。とはいえ、ウィジェットがフレーム内にパックされている場合、マスタはトップレベルウィンドウではありません。任意のウィジェットの入っているトップレベルウィンドウを知りたければ _root() メソッドを呼び出してください。このメソッドはアンダースコアがついていますが、これはこの関数が Tkinter の実装の一部であり、Tk の機能に対するインタフェースではないことを示しています。

以下に典型的な使い方の例をいくつか挙げます:

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()

# create the application
myapp = App()

#
# here are method calls to the window manager class
#
myapp.master.title("My Do-Nothing Application")
myapp.master.maxsize(1000, 400)

# start the program
myapp.mainloop()
```

Tk オプションデータ型

anchor　指定できる値はコンパスの方位です: "n" 、"ne" 、"e" 、"se" 、"s" 、"sw" 、"w" 、"nw" 、および "center" 。

bitmap　八つの組み込み、名前付きビットマップ: 'error' 、'gray25' 、'gray50' 、'hourglass' 、'info' 、'questhead' 、'question' 、'warning' 。X ビットマップファイル名を指定するために、"@/usr/contrib/bitmap/gumby.bit" のような @ を先頭に付けたファイルへの完全なパスを与えてください。

boolean　整数 0 または 1 、あるいは、文字列 "yes" または "no" を渡すことができます。

callback　これは引数を取らない Python 関数ならどれでも構いません。例えば:

```
def print_it():
    print("hi there")
fred["command"] = print_it
```

color　色は rgb.txt ファイルの X カラーの名前か、または RGB 値を表す文字列として与えられます。RGB 値を表す文字列は、4 ビット: `"#RGB"`, 8 bit: `"#RRGGBB"`, 12 bit: `"#RRRGGGBBB"`, あるいは、16 bit `"#RRRRGGGGBBBB"` の範囲を取ります。ここでは、R,G,B は適切な十六進数ならどんなものでも表します。詳細は、Ousterhout の本の 160 ページを参照してください。

cursor　`cursorfont.h` の標準 X カーソル名を、接頭語 `XC_` 無しで使うことができます。例えば、hand カーソル (`XC_hand2`) を得るには、文字列 `"hand2"` を使ってください。あなた自身のビットマップとマスクファイルを指定することもできます。Ousterhout の本の 179 ページを参照してください。

distance　スクリーン上の距離をピクセルか絶対距離のどちらかで指定できます。ピクセルは数値として与えられ、絶対距離は文字列として与えられます。絶対距離を表す文字列は、単位を表す終了文字 (センチメートルには `c` 、インチには `i` 、ミリメートルには `m` 、プリンタのポイントには `p`) を伴います。例えば、3.5 インチは `"3.5i"` と表現します。

font　Tk はフォント名の形式に `{courier 10 bold}` のようなリストを使います。正の数のフォントサイズはポイント単位で使用され、負の数のサイズはピクセル単位と見なされます。

geometry　これは `widthxheight` 形式の文字列です。ここでは、ほとんどのウィジェットに対して幅と高さピクセル単位で (テキストを表示するウィジェットに対しては文字単位で) 表されます。例えば: `fred["geometry"] = "200x100"` 。

justify　指定できる値は文字列です: `"left"` 、`"center"` 、`"right"` 、そして `"fill"` 。

region　これは空白で区切られた四つの要素をもつ文字列です。各要素は指定可能な距離です (以下を参照)。例えば: `"2 3 4 5"` と `"3i 2i 4.5i 2i"` と `"3c 2c 4c 10.43c"` は、すべて指定可能な範囲です。

relief　ウィジェットのボーダのスタイルが何かを決めます。指定できる値は: `"raised"` 、`"sunken"` 、`"flat"` 、`"groove"` 、と `"ridge"` 。

scrollcommand　これはほとんどの場合スクロールバーウィジェットの `set()` メソッドですが、1 個の引数を取るあらゆるウィジェットにもなりえます。

wrap:　次の中の一つでなければなりません: `"none"` 、`"char"` 、あるいは `"word"` 。

バインドとイベント

ウィジェットコマンドからの bind メソッドによって、あるイベントを待つことと、そのイベント型が起きたときにコールバック関数を呼び出すことができるようになります。bind メソッドの形式は:

```
def bind(self, sequence, func, add=''):
```

ここでは:

sequence　は対象とするイベントの型を示す文字列です (詳細については、bind の man ページと John Ousterhout の本の 201 ページを参照してください)。

func　は一引数を取り、イベントが起きるときに呼び出される Python 関数です。イベント・インスタンスが引数として渡されます。(このように実施される関数は、一般に *callbacks* として知られています。)

add　はオプションで、'' か '+' のどちらかです。空文字列を渡すことは、このイベントが関係する他のどんなバインドをもこのバインドが置き換えることを意味します。'+' を使う仕方は、この関数がこのイベント型にバインドされる関数のリストに追加されることを意味しています。

例えば:

```
def turn_red(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turn_red)
```

イベントのウィジェットフィールドが `turn_red()` コールバック内でどのようにアクセスされているかに注目してください。このフィールドは X イベントを捕らえたウィジェットを含んでいます。以下の表はアクセスできる他のイベントフィールドとそれらの Tk での表現方法の一覧です。Tk man ページを参照するときに役に立つでしょう。

Tk	Tkinter イベントフィールド	Tk	Tkinter イベントフィールド
%f	focus	%A	char
%h	高さ	%E	send_event
%k	keycode	%K	keysym
%s	state	%N	keysym_num
%t	time	%T	type
%w	幅	%W	widget
%x	x	%X	x_root
%y	y	%Y	y_root

index パラメータ

多くのウィジェットにはパラメータ "index" を渡す必要があります。これらはテキストウィジェット内の特定の位置や、エントリウィジェット内の特定の文字、あるいはメニューウィジェット内の特定のメニューアイテムを指定するために使用されます。

エントリウィジェットのインデックス (インデックス、ビューインデックスなど)　エントリウィジェットは表示されているテキスト内の文字位置を参照するオプションを持っています。テキストウィジェットにおけるこれらの特別な位置にアクセスするために、次の *tkinter* 関数を使うことができます:

テキストウィジェットのインデックス　テキストウィジェットに対するインデックス記法はとても機能が豊富で、Tk man ページでよく説明されています。

メニューのインデックス (menu.invoke()、menu.entryconfig() など)　メニューに対するいくつかのオプションとメソッドは特定のメニュー項目を操作します。メニューインデックスはオプションまたはパラメータのために必要とされるときはいつでも、以下のものを渡すことができます:

- ウィジェット内の数字の先頭からの位置を指す整数。先頭は 0。

- 文字列 "active"。現在カーソルがあるメニューの位置を指します。
- 文字列 "last" 。最後のメニューを指します。
- @6 のような @ が前に来る整数。ここでは、整数がメニューの座標系における y ピクセル座標として解釈されます。
- 文字列 "none"。どんなメニューエントリもまったく指しておらず、ほとんどの場合、すべてのエントリの動作を停止させるために menu.activate() と一緒に使われます。そして、最後に、
- メニューの先頭から一番下までスキャンしたときに、メニューエントリのラベルに一致したパターンであるテキスト文字列。このインデックス型は他すべての後に考慮されることに注意してください。その代わりに、それは last 、active または none とラベル付けされたメニュー項目への一致は上のリテラルとして解釈されることを意味します。

画像

様々な形式の画像を、それに対応する tkinter.Image のサブクラスを使って作成できます:

- XBM 形式の画像のための BitmapImage 。
- PGM, PPM, GIF, PNG 形式の画像のための PhotoImage 。最後のは Tk 8.6 からサポートされるようになりました。

画像のどちらの型でも file または data オプションを使って作られます (その上、他のオプションも利用できます)。

image オプションがウィジェットにサポートされるところならどこでも、画像オブジェクトを使うことができます (例えば、ラベル、ボタン、メニュー)。これらの場合では、Tk は画像への参照を保持しないでしょう。画像オブジェクトへの最後の Python の参照が削除されたときに、画像データも削除されます。そして、どこで画像が使われていようとも、Tk は空の箱を表示します。

参考:

Pillow パッケージにより、BMP, JPEG, TIFF, WebP などの形式のサポートが追加されました。

25.1.7 ファイルハンドラ

Tk を使うとコールバック関数の登録や解除ができ、ファイルディスクリプタに対する入出力が可能なときに、Tk のメインループからその関数が呼ばれます。ファイルディスクリプタ 1 つにつき、1 つだけハンドラは登録されます。コード例です:

```
import tkinter
widget = tkinter.Tk()
mask = tkinter.READABLE | tkinter.WRITABLE
widget.tk.createfilehandler(file, mask, callback)
...
widget.tk.deletefilehandler(file)
```

これらの機能は Windows では利用できません。

読み込みに使えるバイト数は分からないので、*BufferedIOBase* クラスや *TextIOBase* クラスの *read()* メソッドおよび *readline()* メソッドを使おうとしないでください。これらは読み込みの際に、あらかじめ決められたバイト数を要求するのです。ソケットには、*recv()* や *recvfrom()* メソッドを使うといいです。その他のファイルには、raw 読み込みか `os.read(file.fileno(), maxbytecount)` を使ってください。

`Widget.tk.createfilehandler`(*file*, *mask*, *func*)

ファイルハンドラであるコールバック関数 *func* を登録します。*file* 引数は、(ファイルやソケットオブジェクトのような) *fileno()* メソッドを持つオブジェクトか、整数のファイルディスクリプタとなります。*mask* 引数は、以下にある 3 つの定数の組み合わせの OR を取ったものです。コールバックは次のように呼ばれます:

```
callback(file, mask)
```

`Widget.tk.deletefilehandler`(*file*)

ファイルハンドラの登録を解除します。

`tkinter.READABLE`

`tkinter.WRITABLE`

`tkinter.EXCEPTION`

mask 引数で使う定数です。

25.2 tkinter.ttk --- Tk のテーマ付きウィジェット

ソースコード: Lib/tkinter/ttk.py

---

tkinter.ttk モジュールは Tk 8.5 で導入された Tk のテーマ付きウィジェットへのアクセスを提供します。Tk 8.5 が無い環境で Python がコンパイルされていた場合でも、*Tile* がインストールされていればこのモジュールにアクセスできます。前者のメソッドは Tk 8.5 を使うことで X11 上のフォントのアンチエイリアスや透過ウィンドウ (X11 ではコンポジションウィンドウマネージャが必要です) など、恩恵が増えます。

tkinter.ttk の基本的なアイディアは、拡張可能性のためにウィジェットの動作を実装するコードと見た目を記述するコードを分離することです。

参考:

Tk Widget Styling Support　Tk のテーマサポートを紹介するドキュメント

25.2.1 Ttk を使う

Ttk を使い始めるために、モジュールをインポートします:

```
from tkinter import ttk
```

基本的な Tk ウィジェットを上書きするため、次のように Tk のインポートに続けてインポートを行います:

```
from tkinter import *
from tkinter.ttk import *
```

このように書くと、いくつかの *tkinter.ttk* ウィジェット (Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale, Scrollbar) は自動的に Tk ウィジェットを置き換えます。

これにはプラットフォームをまたいでより良い見た目を得られるという、直接的な利益がありますが、ウィジェットは完全な互換性を持っているわけではありません。一番の違いは "fg" や "bg" やその他のスタイルに関係するウィジェットのオプションが Ttk ウィジェットから無くなっていることです。同じ (もしくはより良い) 見た目にするためには ttk.Style を使ってください。

参考:

Converting existing applications to use Tile widgets　アプリケーションを移行して新しいウィジェットを使用する際に出くわす典型的な差異について (Tcl の用語を使って) 書かれている研究論文

25.2.2 Ttk ウィジェット

Ttk には 18 のウィジェットがあり、そのうち 12 は Tkinter に既にあるものです: Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale, Scrollbar, *Spinbox* 。新しい 6 つは次のものです: *Combobox*, *Notebook*, *Progressbar*, Separator, Sizegrip, *Treeview* 。そして、これらは全て *Widget* の子クラスです。

Ttk ウィジェットを使用すると、アプリケーションの見た目と使いやすさが向上します。上述の通り、書式をコーディングする方法は異なります。

Tk のコード

```
l1 = tkinter.Label(text="Test", fg="black", bg="white")
l2 = tkinter.Label(text="Test", fg="black", bg="white")
```

Ttk のコード:

```
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")

l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

TtkStyling についての詳細は *Style* クラスの文書を読んでください。

25.2.3 ウィジェット

ttk.Widget は Tk のテーマ付きウィジェットがサポートしている標準のオプションやメソッドを定義するもので、これを直接インスタンス化するものではありません。

標準オプション

全ての ttk ウィジェットは以下のオプションを受け付けます:

オプション	説明
クラス	ウィンドウクラスを指定します。このクラスはオプションデータベースにウィンドウの他のオプションについて問い合わせを行うときに使われ、これによりウィンドウのデフォルトのバインドタグを決定したり、ウィジェットのデフォルトのレイアウトやスタイルを選択します。このオプションは読み出し専用で、ウィンドウが作成されるときにのみ指定できます。
cursor	このウィジェットで使うマウスカーソルを指定します。空文字列 (デフォルト) が設定されている場合は、カーソルは親ウィジェットのものを引き継ぎます。
takefocus	キーボードによる移動のときにウィンドウがフォーカスを受け入れるかを決定します。0、1、空文字列のいずれかを返します。0 が返されると、キーボードによる移動でそのウィンドウは常にスキップされます。1 なら、そのウィンドウが表示されているときに限り入力フォーカスを受け入れます。そして空文字列は、移動スクリプトによってウィンドウにフォーカスを当てるかどうかが決まることを意味します。
style	独自のウィジェットスタイルを指定するのに使われます。

スクロール可能ウィジェットのオプション

以下のオプションはスクロールバーで操作されるウィジェットが持っているオプションです。

オプション	説明
xscrollcommand	水平方向のスクロールバーとのやり取りに使われます。 ウィジェットのウィンドウが再描画されたとき, ウィジェットは scrollcommand に基いて Tcl コマンドを生成します。 通常このオプションにはあるスクロールバーのメソッド Scrollbar.set() が設定されます。こうすると、ウィンドウの見た目が変わったときにスクロールバーの状態も更新されます。
yscrollcommand	垂直方向のスクロールバーとのやり取りに使われます。詳しくは、上記を参照してください。

ラベルオプション

以下のオプションはラベルやボタンやボタンに類似したウィジェットが持っているオプションです。

オプション	説明
text	ウィジェットに表示される文字列を指定します。
textvariable	text オプションの代わりに使う値の変数名を指定します。
underline	このオプションを設定すると、文字列の中で下線を引く文字のインデックス (0 基点) を指定します。下線が引かれた文字はショートカットとして使われます。
image	表示する画像を指定します。これは 1 つ以上の要素を持つリストです。先頭の要素はデフォルトの画像名です。残りの要素は *Style.map()* で定義されているような状態名と値のペアの並びで、ウィジェットがある状態、もしくはある状態の組み合わせにいるときに使用する別の画像を指定します。このリストにある全ての画像は同じサイズでなければなりません。
compound	text オプションと image オプションが両方とも指定されていた場合に、テキストに対して画像をどう配置するかを指定します。適当な値は: • text: テキストのみ表示する • image: 画像のみ表示する • top, bottom, left, right: それぞれ画像をテキストの上、下、左、右に配置する。 • none: デフォルト。もしあれば画像を表示し、そうでなければテキストを表示する。
幅	0 より大きい場合、テキストラベルを作成するのにどれくらいのスペースを使うかを文字の幅で指定します。0 より小さい場合、最小の幅が指定されます。0 もしくは無指定の場合、テキストラベルに対して自然な幅が使われます。

互換性オプション

オプション	説明
state	”normal” か ”disabled” に設定され、”disabled” 状態のビットをコントロールします。これは書き込み専用のオプションです: これを設定するとウィジェットの状態を変更できますが、*Widget.state()* メソッドはこのオプションに影響を及ぼしません。

ウィジェットの状態

ウィジェットの状態は独立した状態フラグのビットマップです。

Flag	説明
active	マウスカーソルがウィジェットの上にあり、マウスのボタンをクリックすることで何らかの動作をさせられます
disabled	プログラムによってウィジェットは無効化されています
focus	ウィジェットにキーボードフォーカスがあります
pressed	ウィジェットは押されています
selected	チェックボタンやラジオボタンのようなウィジェットでの ” オン” や ” チェック有” や ” 選択中” に当たります
background	Windows と Mac には ” アクティブな” もしくは最前面のウィンドウという概念があります。背面のウィンドウにあるウィジェットには *background* 状態が設定され、最前面のウィンドウにあるウィジェットでは解除されます
readonly	ウィジェットはユーザからの変更を受け付けません
alternate	ウィジェット特有の切り替え表示になっています
invalid	ウィジェットの値が不正です

状態仕様は状態名の並びになっていて、状態名の先頭にはビットがオフになっていることを示す感嘆符が付くことがあります。

ttk.Widget

以下に書かれているメソッドに加えて、`ttk.Widget` は `tkinter.Widget.cget()` メソッドと `tkinter.Widget.configure()` メソッドをサポートしています。

class `tkinter.ttk.Widget`

identify(*x*, *y*)
: *x* *y* の位置にある要素の名前、もしくはその位置に要素が無ければ空文字列を返します。

  x と *y* はウィジェットに対するピクセル単位の座標です。

instate(*statespec*, *callback=None*, **args*, ***kw*)
: ウィジェットの状態をチェックします。コールバックが指定されていない場合、ウィジェットの状態が *statespec* に一致していれば `True` 、そうでなければ `False` を返します。コールバックが指定されていて、ウィジェットの状態が *statespec* に一致している場合、引数に args を指定してそのコールバックを呼び出します。

state(*statespec=None*)
: ウィジェットの状態を変更したり、取得したりします。*statespec* が指定されている場合、それに応じてウィジェットの状態を設定し、どのフラグが変更されたかを示す新しい *statespec* を返します。*statespec* が指定されていない場合、現在の状態フラグを返します。

通常 *statespec* はリストもしくはタプルです。

25.2.4 コンボボックス

ttk.Combobox ウィジェットはテキストフィールドと値のポップダウンリストを結び付けます。このウィジェットは Entry の子クラスです。

Widget から継承したメソッド (Widget.cget(), Widget.configure(), *Widget.identify()*, *Widget.instate()*, *Widget.state()*) と以下の Entry から継承したメソッド (Entry.bbox(), Entry.delete(), Entry.icursor(), Entry.index(), Entry.insert(), Entry.selection(), Entry.xview()) に加え、このクラスには ttk.Combobox で説明するメソッドがあります。

オプション

このウィジェットは以下の固有のオプションを受け付けます:

オプション	説明
exportselection	真偽値を取る。設定されている場合、ウィジェットの選択はウィンドウマネージャの選択とリンクしています (例えば、Misc.selection_get を実行することで得られます)。
justify	ウィジェットの中でテキストをどう配置するかを指定します。”left”, ”center”, ”right” のうちのどれか 1 つです。
高さ	ポップダウンリストの高さを行数で指定します。
postcommand	コンボボックスの値を表示する直前に呼び出される、(Misc.register などで登録した) スクリプトです。どの値を表示するかについても指定できます。
state	”normal”, ”readonly”, ”disabled” のどれか 1 つです。”readonly” 状態では、直接入力値を編集することはできず、ユーザはドロップダウンリストから値を 1 つ選ぶことしかできません。”normal” 状態では、テキストフィールドは直接編集できます。”disabled” 状態では、コンボボックスは一切反応しません。
textvariable	コンボボックスの値とリンクさせる変数名を指定します。その変数の値が変更されたとき、ウィジェットの値は更新されます。ウィジェットの値が更新されたときも同様です。tkinter.StringVar を参照してください。
values	ドロップダウンリストに表示する値のリストを指定します。
幅	入力ウィンドウに必要な幅をウィジェットのフォントの平均的なサイズの文字で測った、文字数を指定します。

仮想イベント

コンボボックスウィジェットは、ユーザが値のリストから 1 つ選んだときに仮想イベント **«ComboboxSelected»** を生成します。

ttk.Combobox

class tkinter.ttk.Combobox

current(*newindex=None*)
newindex が指定されている場合、コンボボックスの値がドロップダウンリストの *newindex* の位置にある値に設定されます。そうでない場合、現在の値のインデックスを、もしくは現在の値がリストに含まれていないなら -1 を返します。

get()
コンボボックスの現在の値を返します。

set(*value*)
コンボボックスの値を *value* に設定します。

25.2.5 Spinbox

The `ttk.Spinbox` widget is a `ttk.Entry` enhanced with increment and decrement arrows. It can be used for numbers or lists of string values. This widget is a subclass of `Entry`.

Widget から継承したメソッド: `Widget.cget()`, `Widget.configure()`, *Widget.identify()*, *Widget.instate()*, *Widget.state()* と、以下の `Entry` から継承したメソッド: `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.xview()` に加え、このクラスには `ttk.Spinbox` で説明するいくつかのメソッドがあります。

オプション

このウィジェットは以下の固有のオプションを受け付けます:

オプション	説明
from	Float value. If set, this is the minimum value to which the decrement button will decrement. Must be spelled as `from_` when used as an argument, since `from` is a Python keyword.
to	Float value. If set, this is the maximum value to which the increment button will increment.
increment	Float value. Specifies the amount which the increment/decrement buttons change the value. Defaults to 1.0.
`values`	Sequence of string or float values. If specified, the increment/decrement buttons will cycle through the items in this sequence rather than incrementing or decrementing numbers.
wrap	Boolean value. If `True`, increment and decrement buttons will cycle from the `to` value to the `from` value or the `from` value to the `to` value, respectively.
format	String value. This specifies the format of numbers set by the increment/decrement buttons. It must be in the form "%W.Pf", where W is the padded width of the value, P is the precision, and '%' and 'f' are literal.
command	Python callable. Will be called with no arguments whenever either of the increment or decrement buttons are pressed.

仮想イベント

The spinbox widget generates an **«Increment»** virtual event when the user presses <Up>, and a **«Decrement»** virtual event when the user presses <Down>.

ttk.Spinbox

`class tkinter.ttk.Spinbox`

`get()`
Returns the current value of the spinbox.

`set(`*value*`)`
Sets the value of the spinbox to *value*.

25.2.6 ノートブック

ノートブックウィジェットは複数のウィンドウを管理し、同時に 1 つのウィンドウを表示します。それぞれの子ウィンドウはタブの関連付けられていて、ユーザはそれを選択して表示されているウィンドウを切り替えます。

オプション

このウィジェットは以下の固有のオプションを受け付けます:

オプション	説明
高さ	0 より大きな値が設定されている場合、(内部のパディングやタブを含まない) ペイン領域に必要な高さを指定します。設定されていない場合、全てのペインの高さの最大値が使われます。
padding	ノートブックの外周に付け足す追加の領域の量を指定します。パディングは最大 4 個の長さ指定のリストです: 左、上、右、下の順で指定します。4 個より少ない場合、デフォルトで下は上と、右は左と、上は左と同じ値が、それぞれ使われます。
幅	0 より大きな値が指定されている場合、(内部のパディングを含まない) ペイン領域に必要な幅を指定します。設定されていない場合、全てのペインの幅の最大値が使われます。

タブオプション

タブ用のオプションもあります:

オプション	説明
state	”normal”, ”disabled”, ”hidden” のうちどれか 1 つです。”disabled” の場合、タブは選択することができません。”hidden” の場合、タブは表示されません。
sticky	ペイン領域の中に子ウィンドウがどう置かれるかを指定します。指定する値は ”n”, ”s”, ”e”, ”w” からなる 0 文字以上の文字列です。配置マネージャの `grid()` と同様に、それぞれの文字は子ウィンドウが (北、南、東、西の) どの辺に対して追随するかに対応しています。
padding	ノートブックとこのペインの間に付け足す追加の領域の量を指定します。文法はこのウィジェットの padding オプションと同じです。
text	タブに表示するテキストを指定します。
image	タブに表示する画像を指定します。*`Widget`* のオプション image の説明を参照してください。
compound	text オプションと image オプションが両方指定されているときにテキストに対して画像をどう表示するかを指定します。指定する値については *Label Options* を参照してください。
underline	テキスト中の下線を引く文字のインデックス (0 基点) を指定します。*`Notebook.enable_traversal()`* が呼ばれていた場合、下線が引かれた文字はショートカットとして使われます。

タブ識別子

`ttk.Notebook` のいくつかのメソッドにある *tab_id* は以下の形式を取ります:

- 0 からタブの数の間の整数
- 子ウィンドウの名前
- タブを指し示す "@x,y" という形式の位置指定
- 現在選択されているタブを指し示すリテラル文字列 "current"
- タブ数を返すリテラル文字列 "end" (*`Notebook.index()`* でのみ有効)

仮想イベント

このウィジェットは新しいタブが選択された後に仮想イベント **«NotebookTabChanged»** を生成します。

ttk.Notebook

class tkinter.ttk.Notebook

add(*child*, ***kw*)
ノートブックに新しいタブを追加します。

ウィンドウが現在ノートブックによって管理されているが隠れている場合、以前の位置に復元します。

利用可能なオプションのリストについては *Tab Options* を参照してください。

forget(*tab_id*)
tab_id で指定されたタブを削除します。関連付けられていたウィンドウは切り離され、管理対象でなくなります。

hide(*tab_id*)
tab_id で指定されたタブを隠します。

タブは表示されませんが、関連付いているウィンドウはノートブックによって保持されていて、その設定も記憶されています。隠れたタブは *`add()`* コマンドで復元できます。

identify(*x*, *y*)
x *y* の位置にあるタブの名前を、そこにタブが無ければ空文字列を返します。

index(*tab_id*)
tab_id で指定されたタブのインデックスを、*tab_id* が文字列の "end" だった場合はタブの総数を返します。

insert(*pos*, *child*, ***kw*)
指定された位置にペインを挿入します。

pos は文字列の "end" か整数のインデックスか管理されている子ウィンドウの名前です。*child* が既にノートブックの管理対象だった場合、指定された場所に移動させます。

利用可能なオプションのリストについては *Tab Options* を参照してください。

select(*tab_id=None*)
指定された *tab_id* を選択します。

関連付いている子ウィンドウは表示され、直前に選択されていたウィンドウは (もし異なれば) 表示されなくなります。*tab_id* が指定されていない場合は、現在選択されているペインのウィジェット名を返します。

tab(*tab_id*, *option=None*, ***kw*)
指定された *tab_id* のオプションを問い合わせたり、変更したりします。

kw が与えられなかった場合、タブのオプション値の辞書を返します。*option* が指定されていた場合、その *option* の値を返します。それ以外の場合は、オプションに対応する値が設定されます。

tabs()
ノートブックに管理されているウィンドウのリストを返します。

enable_traversal()
このノートブックを含む最上位にあるウィンドウでのキーボード移動を可能にします。

これによりノートブックを含んだ最上位にあるウィンドウに対し、以下のキーバインディングが追加されます:

- Control-Tab: 現在選択されているタブの 1 つ次のタブを選択します。
- Shift-Control-Tab: 現在選択されているタブの 1 つ前のタブを選択します。
- Alt-K: K があるタブの (下線が引かれた) ショートカットキーだとして、そのタブを選択します。

ネストしたノートブックも含め、1 つのウィンドウの最上位にある複数のノートブックのキーボード移動が可能になることもあります。しかしノートブック上の移動は、全てのペインが同じノートブックを親としているときのみ正しく動作します。

25.2.7 プログレスバー

ttk.Progressbar ウィジェットは長く走る処理の状態を表示します。このウィジェットは 2 つのモードで動作します: 1) 決定的モードでは、全ての処理の総量のうち完了した量を表示します。2) 非決定的モードでは、今作業が進んでいることをユーザに示します。

オプション

このウィジェットは以下の固有のオプションを受け付けます:

オプション	説明
orient	”horizontal” もしくは ”vertical” のいずれかです。プログレスバーの方向を指定します。
length	プログレスバーの長さを指定します。(水平方向の場合は幅、垂直方向の場合は高さです)
mode	”determinate” か ”indeterminate” のいずれかです。
maximum	最大値を数値で指定します。デフォルトは 100 です。
value	プログレスバーの現在値です。決定的 (”determinate”) モードでは、完了した処理の量を表します。非決定的 (”indeterminate”) モードでは、*maximum* を法として解釈され、値が *maximum* に達したときにプログレスバーは 1 ” サイクル” を完了したことになります。
variable	value オプションとリンクさせる変数名です。指定されている場合、変数の値が変更されるとプログレスバーの値は自動的にその値に設定されます。
phase	読み出し専用のオプションです。このウィジェットの値が 0 より大きく、かつ決定的モードでは最大値より小さいときに、ウィジェットが定期的にこのオプションの値を増加させます。このオプションは現在の画面テーマが追加のアニメーション効果を出すのに使います。

ttk.Progressbar

`class tkinter.ttk.Progressbar`

`start`(*interval=None*)
自動増加モードを開始します: *interval* ミリ秒ごとに *Progressbar.step()* を繰り返し呼び出すタイマーイベントを設定します。引数で指定しない場合は、*interval* はデフォルトで 50 ミリ秒になります。

`step`(*amount=None*)
プログレスバーの値を *amount* だけ増加させます。

引数で指定しない場合は、*amount* はデフォルトで 1.0 になります。

`stop`()
自動増加モードを停止します: このプログレスバーの *Progressbar.start()* で開始された繰り返しのタイマーイベントを全てキャンセルします。

25.2.8 セパレータ

ttk.Separator ウィジェットは水平もしくは垂直のセパレータを表示します。

ttk.Widget から継承したメソッド以外にメソッドを持ちません。

オプション

このウィジェットは次の特定のオプションを受け付けます。

オプション	説明
orient	”horizontal” か ”vertical” のいずれかです。セパレータの方向を指定します。

25.2.9 サイズグリップ

(グローボックスとしても知られる) ttk.Sizegrip ウィジェットを使用すると、つまみ部分を押してドラッグすることで、このウィジェットを含む最上位のウィンドウのサイズを変更できます。

このウィジェットは ttk.Widget から継承したもの以外のオプションとメソッドを持ちません。

プラットフォーム固有のメモ

- Mac OS X (訳注: 今後は macOS と呼ばれます) では、最上位のウィンドウにはデフォルトで組み込みのサイズグリップが含まれています。組み込みのグリップが Sizegrip を隠してしまうので、Sizegrip を追加しても問題ありません。

バグ

- このウィジェットを含む最上位のウィンドウの位置がスクリーンの右端や下端に対して相対的に指定されている場合 (例: ....) 、Sizegrip ウィジェットはウィンドウのサイズ変更をしません。
- このウィジェットは ” 南東” 方向のサイズ変更しかサポートしていません。

25.2.10 ツリービュー

ttk.Treeview ウィジェットは階層のある要素 (アイテム) の集まりを表示します。それぞれの要素はテキストラベル、オプションの画像、オプションのデータのリストを持っています。データはラベルの後に続くカラムに表示されます。

データが表示される順序はウィジェットの displaycolumns オプションで制御されます。ツリーウィジェットはカラムヘッダを表示することもできます。カラムには数字もしくはウィジェットの columns オプションにある名前でアクセスできます。*Column Identifiers* を参照してください。

それぞれの要素は一意な名前で識別されます。要素の作成時に識別子が与えられなかった場合、ウィジェットが要素の識別子を生成します。このウィジェットには {} という名前の特別なルート要素があります。ルート要素自身は表示されません; その子要素たちが階層の最上位に現れます。

それぞれの要素はタグのリストも持っていて、イベントバインディングと個別の要素を関連付け、要素の見た目を管理するのに使えます。

ツリービューウィジェットは水平方向と垂直方向のスクロールをサポートしていて、*Scrollable Widget Options* に記述してあるオプションと *Treeview.xview()* メソッドおよび *Treeview.yview()* メソッドが使えます。

オプション

このウィジェットは以下の固有のオプションを受け付けます:

オプション	説明
columns	カラム数とその名前を指定するカラム識別子のリストです。
displaycolumns	どのデータカラムをどの順序で表示するかを指定する、(名前もしくは整数のインデックスの) カラム識別子のリストか、文字列 "#all" です。
高さ	表示する行数を指定します。メモ: 表示に必要な幅はカラム幅の合計から決定されます。
padding	ウィジェットの内部のパディングのサイズを指定します。パディングは最大 4 個の長さ指定のリストです。
selectmode	組み込みのクラスバインディングが選択状態をどう管理するかを指定します。設定する値は "extended", "browse", "none" のどれか 1 つです。"extended" に設定した場合 (デフォルト)、複数の要素が選択できます。"browse" に設定した場合、同時に 1 つの要素しか選択できません。"none" に設定した場合、選択を変更することはできません。 このオプションの値によらず、アプリケーションのコードと関連付けられているタグからは好きなように選択状態を設定できます。
show	ツリーのどの要素を表示するかを指定する、以下にある値を 0 個以上含むリストです。 • tree: カラム #0 にツリーのラベルを表示します。 • headings: ヘッダ行を表示します。 デフォルトは "tree headings" 、つまり全ての要素を表示します。 **メモ**: show="tree" が指定されていない場合でも、カラム #0 は常にツリーカラムを参照します。

要素オプション

以下の要素オプションは、ウィジェットの insert コマンドと item コマンドで要素に対して指定できます。

オプション	説明
text	アイテムに表示するテキストラベルです。
image	ラベルの左に表示される Tk 画像です。
`values`	要素に関連付けられている値のリストです。 それぞれの要素はウィジェットの columns オプションと同じ数の値を持たなければいけません。columns オプションより少ない場合、残りの値は空として扱われます。 columns オプションより多い場合、余計な値は無視されます。
open	要素の子供を表示するか隠すかを指示する `True`/`False` 値です。
tags	この要素に関連付いているタグのリストです。

タグオプション

以下のオプションはタグに対して設定できます:

オプション	説明
foreground	テキストの色を指定します。
background	セルや要素の背景色を指定します。
font	テキストを描画するときに使うフォントを指定します。
image	要素の image オプションが空だった場合に使用する画像を指定します。

カラム識別子

カラム識別子は以下のいずれかの形式を取ります:

- columns オプションのリストにある名前。
- n 番目のデータカラムを指し示す整数 n 。
- n を整数として n 番目の表示されているカラムを指し示す #n という形式の文字列。

注釈:

- 要素のオプション値は実際に格納されている順序とは違った順序で表示されることがあります。
- show=”tree” が指定されていない場合でも、カラム #0 は常にツリーカラムを指しています。

データカラムを指す数字は、要素の values オプションのリストのインデックスです; 表示カラムを指す数字は、値が表示されているツリーのカラム番号です。ツリーラベルはカラム #0 に表示されます。displaycolumns オプションが設定されていない場合は、n 番目のデータカラムはカラム #n+1 に表示されます。再度言っておくと、**カラム #0 は常にツリーカラムを指します** 。

仮想イベント

ツリービューは以下の仮想イベントを生成します。

Event	説明
«TreeviewSelect»	選択状態が変更されたときに生成されます。
«TreeviewOpen»	フォーカスが当たっている要素に open=True が設定される直前に生成されます。
«TreeviewClose»	フォーカスが当たっている要素に open=False が設定された直後に生成されます。

Treeview.focus() メソッドと *Treeview.selection()* メソッドは変更を受けた要素を判別するのに使えます。

ttk.Treeview

class tkinter.ttk.Treeview

bbox(*item*, *column=None*)

(ツリービューウィジェットのウィンドウを基準として) 指定された *item* のバウンディングボックス情報を (x 座標, y 座標, 幅, 高さ) の形式で返します。

column が指定されている場合は、セルのバウンディングボックスを返します。(例えば、閉じた状態の要素の子供であったり、枠外にスクロールされていて) *item* が見えなくなっている場合は、空文字列が返されます。

get_children(*item=None*)

item の子要素のリストを返します。

item が指定されていなかった場合は、ルート要素の子供が返されます。

set_children(*item*, **newchildren*)

item の子要素を *newchildren* で置き換えます。

item にいる子供のうち *newchildren* にないものはツリーから切り離されます。*newchildren* にあるどの要素も *item* の祖先であってはいけません。*newchildren* を指定しなかった場合は、*item* の子要素が全て切り離されることに注意してください。

column(*column*, *option=None*, ***kw*)

指定した *column* のオプションを問い合わせたり、変更したりします。

kw が与えられなかった場合は、カラムのオプション値の辞書が返されます。*option* が指定されていた場合は、その *option* の値が返されます。それ以外の場合は、オプションに値を設定します。

設定できるオプションとその値は次の通りです:

- id　カラム名を返します。これは読み出し専用のオプションです。

- anchor: **標準の** Tk anchor **の値**　このカラムでセルに対してテキストをどう配置するかを指定します。
- minwidth: **幅**　カラムの最小幅をピクセル単位で表したものです。ツリービューウィジェットは、ウィジェットのサイズが変更されたりカラムをユーザがドラッグして移動させたりしたときに、このオプションで指定した幅より狭くすることはありません。
- stretch: `True` **もしくは** `False`　ウィジェットがサイズ変更されたとき、カラムの幅をそれに合わせるかどうかを指定します。
- width: **幅**　カラムの幅をピクセル単位で表したものです。

ツリーカラムの設定を行うには、column = "#0" を付けてこのメソッドを呼び出してください。

`delete`(**items*)
指定された *items* とその子孫たち全てを削除します。

ルート要素は削除されません。

`detach`(**items*)
指定された *items* を全てツリーから切り離します。

その要素と子孫たちは依然として存在していて、ツリーの別の場所に再度挿入することができますが、隠された状態になり表示はされません。

ルート要素は切り離されません。

`exists`(*item*)
指定された *item* がツリーの中にあれば `True` を返します。

`focus`(*item=None*)
item が指定されていた場合は、*item* にフォーカスを当てます。そうでない場合は、現在フォーカスが当たっている要素が、どの要素にもフォーカスが当たっていない場合は '' が返されます。

`heading`(*column*, *option=None*, ***kw*)
指定された *column* の heading のオプションを問い合わせたり、変更したりします。

kw が与えられなかった場合は、見出しのオプション値の辞書が返されます。*option* が指定されている場合は、*option* の値が返されます。それ以外の場合は、オプションに値を設定します。

設定できるオプションとその値は次の通りです:

- text: **テキスト**　カラムの見出しに表示するテキスト。
- image: **画像名**　カラムの見出しの右に表示する画像を指定します。
- anchor: anchor　見出しのテキストをどう配置するかを指定します。標準の Tk anchor の値です。
- command: **コールバック**　見出しラベルがクリックされたときに実行されるコールバックです。

ツリーカラムの見出しの設定を行うには、column = "#0" を付けてこのメソッドを呼び出してください。

identify(*component*, *x*, *y*)

x *y* で与えられた場所にある指定された *component* の説明を返します。その場所に指定された *component* が無い場合は空文字列を返します。(訳注: component には "region", "item", "column", "row", "element" が指定でき、それぞれ "cell", "heading" などの場所の名前、要素の識別子、#n という形式のカラム名、その行にある要素の識別子、"text", "padding" などの画面構成要素の名前を返します。)

identify_row(*y*)

y 座標が *y* の位置にある要素の識別子を返します。

identify_column(*x*)

x 座標が *x* の位置にあるセルのデータカラムの識別子を返します。

ツリーカラムは #0 という識別子を持ちます。

identify_region(*x*, *y*)

以下のうち 1 つを返します:

region	意味
heading	ツリーの見出し領域
separator	2 つのカラム見出しの間のスペース
tree	ツリーの領域
cell	データセル

使用可能バージョン: Tk 8.6

identify_element(*x*, *y*)

x *y* の位置にある画面構成要素の名前を返します。

使用可能バージョン: Tk 8.6

index(*item*)

親要素の子要素リストの中での *item* のインデックスを返します。

insert(*parent*, *index*, *iid=None*, ***kw*)

新しい要素を作り、その要素の識別子を返します。

parent は親となる要素の識別子で、空文字列にすると新しい要素を最上位に作成します。*index* は整数もしくは "end" という値で、それによって親要素の子要素リストのどこに新しい要素を挿入するかを指定します。*index* が 0 以下だった場合は、新しい要素は先頭に挿入されます; *index* が現在の子要素の数以上だった場合は末尾に挿入されます。*iid* が指定された場合は、要素の識別子として使われます; *iid* はまだツリーに存在していないものに限ります。それ以外の場合は、一意な識別子が生成されます。

使用できるオプションのリストについては *Item Options* を参照してください。

item(*item*, *option=None*, ***kw*)
指定された *item* のオプションを問い合わせたり、変更したりします。

オプションが与えられなかった場合は、要素のオプションと値が辞書の形で返されます。*option* が指定された場合は、そのオプションの値が返されます。それ以外の場合は、*kw* で与えられたようにオプションに値が設定されます。

move(*item*, *parent*, *index*)
item を *parent* の子要素リストの *index* の位置に移動します。

要素を自身の子孫の下に移動させるのは許されていません。*index* が 0 以下の場合、*item* は先頭に移動されます; 子要素の数以上だった場合、末尾に移動されます。*item* が切り離された状態の場合は、再度取り付けられます。

next(*item*)
item の 1 つ下の兄弟の識別子を、*item* が親にとって一番下の子供だった場合 '' を返します。

parent(*item*)
item の親の識別子を、*item* が階層の最上位にいた場合 '' を返します。

prev(*item*)
item の 1 つ上の兄弟の識別子を、*item* が親にとって一番上の子供だった場合 '' を返します。

reattach(*item*, *parent*, *index*)
Treeview.move() のエイリアスです。

see(*item*)
item を見える状態にします。

item の全ての子孫の open オプションを `True` にし、必要であれば *item* がツリーの見える範囲に来るようにウィジェットをスクロールさせます。

selection()
Returns a tuple of selected items.

バージョン 3.8 で変更: `selection()` no longer takes arguments. For changing the selection state use the following selection methods.

selection_set(**items*)
新しく選択状態の要素が *items* になります。

バージョン 3.6 で変更: *items* は 1 つのタプルとしてだけでなく、別々の引数としても渡せます。

selection_add(**items*)
選択状態の要素として *items* を追加します。

バージョン 3.6 で変更: *items* は 1 つのタプルとしてだけでなく、別々の引数としても渡せます。

selection_remove(**items*)
選択状態の要素から *items* を取り除きます。

バージョン 3.6 で変更: *items* は 1 つのタプルとしてだけでなく、別々の引数としても渡せます。

selection_toggle(**items*)
items のそれぞれの要素の選択状態を入れ替えます。

バージョン 3.6 で変更: *items* は 1 つのタプルとしてだけでなく、別々の引数としても渡せます。

set(*item*, *column=None*, *value=None*)
1 引数で呼び出された場合、指定された *item* のカラムと値のペアからなる辞書を返します。2 引数で呼び出された場合、指定された *column* の現在の値を返します。3 引数で呼び出された場合、与えられた *item* の *column* を指定された値 *value* に設定します。

tag_bind(*tagname*, *sequence=None*, *callback=None*)
与えられたイベント *sequence* 用のコールバックをタグ *tagname* にバインドします。イベントが要素に渡ってきたときに、要素の tags オプションのそれぞれのコールバックが呼び出されます。

tag_configure(*tagname*, *option=None*, ***kw*)
指定された *tagname* のオプションを問い合わせたり、変更したりします。

kw が与えられなかった場合、*tagname* のオプション設定を辞書の形で返します。*option* が指定された場合、指定された *tagname* の *option* の値を返します。それ以外の場合、与えられた *tagname* のオプションに値を設定します。

tag_has(*tagname*, *item=None*)
item が指定されていた場合、指定された *item* が与えられた *tagname* を持っているかどうかに従って 1 または 0 が返されます。そうでない場合、指定されたタグを持つ全ての要素のリストを返します。

使用可能バージョン: Tk 8.6

xview(**args*)
ツリービューの水平方向の位置を問い合わせたり、変更したりします。

yview(**args*)
ツリービューの垂直方向の位置を問い合わせたり、変更したりします。

25.2.11 Ttk スタイル

ttk のそれぞれのウィジェットにはスタイルが関連付けられていて、それと動的もしくはデフォルトで設定される要素のオプションによってウィジェットを構成する要素とその配置を指定します。デフォルトではスタイル名はウィジェットのクラス名と同じですが、ウィジェットの style オプションで上書きすることができます。ウィジェットのクラス名が分からない場合は、**Misc.winfo_class()** (somewidget.winfo_class()) メソッドを使ってください。

参考:

Tcl'2004 conference **のプレゼンテーション**　この文書ではテーマエンジンがどう動くかを説明しています

class tkinter.ttk.Style
このクラスはスタイルデータベースを操作するために使われます。

configure(*style*, *query_opt=None*, ***kw*)

style の指定されたオプションのデフォルト値を問い合わせたり、設定したりします。

kw のそれぞれのキーはオプション名で値はそのオプションの値の文字列です。

例えば、全てのデフォルトのボタンをパディングのある平らな見た目にし、背景の色を変更するには以下のようにします:

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

ttk.Style().configure("TButton", padding=6, relief="flat",
   background="#ccc")

btn = ttk.Button(text="Sample")
btn.pack()

root.mainloop()
```

map(*style*, *query_opt=None*, ***kw*)

style の指定されたオプションの動的な値を問い合わせたり、設定したりします。

kw のそれぞれのキーはオプション名で、値はタプルやリストや何か他のお好みのものでグループ化された状態仕様 (statespec) を要素とするリストやタプルです。状態仕様は 1 つもしくは複数の状態と値の組み合わせです。

以下のように、例を示す方がわかりやすいでしょう:

```
import tkinter
from tkinter import ttk

root = tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=[('pressed', 'red'), ('active', 'blue')],
    background=[('pressed', '!disabled', 'black'), ('active', 'white')]
    )

colored_btn = ttk.Button(text="Test", style="C.TButton").pack()

root.mainloop()
```

あるオプションに対する状態と値の組 (states, value) の並び順はスタイルに影響を与えることに注意してください。例えば、foreground オプションの順序を `[('active', 'blue'), ('pressed', 'red')]` に変更した場合、ウィジェットがアクティブもしくは押された状態のとき前面が青くなります。

lookup(*style*, *option*, *state=None*, *default=None*)

style の指定された *option* の値を返します。

state を指定する場合は、1 つ以上の状態名の並びである必要があります。*default* 引数が指定されていた場合は、オプション指定が見付からなかったときに代わりに返される値として使われます。

デフォルトでボタンがどのフォントを使うかを調べるには、以下のように実行します:

```
from tkinter import ttk

print(ttk.Style().lookup("TButton", "font"))
```

layout(*style*, *layoutspec=None*)

与えられた *style* でのウィジェットのレイアウトを定義します。*layoutspec* が省略されていた場合は、与えられたスタイルのレイアウト仕様を返します。

layoutspec を指定する場合は、リストもしくは (文字列を除いた) 何か他のシーケンス型である必要があります。それぞれの要素はタプルで、レイアウト名を 1 番目の要素とし、2 番目の要素は *Layouts* で説明されているフォーマットである必要があります。

フォーマットを理解するために以下の例を見てください (何かを使い易くするための例ではありません):

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.layout("TMenubutton", [
   ("Menubutton.background", None),
   ("Menubutton.button", {"children":
       [("Menubutton.focus", {"children":
           [("Menubutton.padding", {"children":
               [("Menubutton.label", {"side": "left", "expand": 1})]
           })]
       })]
   }),
])

mbtn = ttk.Menubutton(text='Text')
mbtn.pack()
root.mainloop()
```

element_create(*elementname*, *etype*, **args*, ***kw*)

与えられた *etype* ("image", "from", "vsapi" のいずれか) の現在のテーマに新しい要素を作成します。最後の "vsapi" は Windows XP と Vista の Tk 8.6a のみで使用可能でここでは説明しません。

"image" が使われた場合、*args* はデフォルトの画像名の後ろに状態仕様と値のペア (これが画像仕様です) を並べたものである必要があります。*kw* には以下のオプションが指定できます:

- border=padding　padding は 4 個以下の整数のリストで、それぞれ左、上、右、下の縁の幅を指定します。

- height=height　要素の最小の高さを指定します。0 より小さい場合は、画像の高さをデフォルトとして使用します。
- padding=padding　要素の内部のパディングを指定します。指定されない場合は、border の値がデフォルトとして使われます。
- sticky=spec　1 つ外側の枠に対し画像をどう配置するかを指定します。spec は ”n”, ”s”, ”w”, ”e” の文字を 0 個以上含みます。
- width=width　要素の最小の幅を指定します。0 より小さい場合は、画像の幅をデフォルトとして使用します。

etype の値として ”from” が使われた場合は、*element_create()* が現在の要素を複製します。*args* は要素の複製元のテーマの名前と、オプションで複製する要素を含んでいる必要があります。複製元の要素が指定されていなかった場合、空要素が使用され、*kw* は破棄されます。

element_names()
現在のテーマに定義されている要素のリストを返します。

element_options(*elementname*)
elementname のオプションのリストを返します。

theme_create(*themename*, *parent=None*, *settings=None*)
新しいテーマを作成します。

themename が既に存在していた場合はエラーになります。*parent* が指定されていた場合は、新しいテーマは親テーマからスタイルや要素やレイアウトを継承します。*settings* が指定された場合は、*theme_settings()* で使われるのと同じ形式である必要があります。

theme_settings(*themename*, *settings*)
一時的に現在のテーマを *themename* に設定し、指定された *settings* を適用した後、元のテーマを復元します。

settings のそれぞれのキーはスタイル名で値はさらに ’configure’, ’map’, ’layout’, ’element create’ をキーとして持ち、その値はそれぞれ *Style.configure()*, *Style.map()*, *Style.layout()*, *Style.element_create()* メソッドで指定するのと同じ形式である必要があります。

例として、コンボボックスの default テーマを少し変更してみましょう

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
   "TCombobox": {
       "configure": {"padding": 5},
       "map": {
           "background": [("active", "green2"),
                          ("!disabled", "green4")],
```

(次のページに続く)

(前のページからの続き)

```
            "fieldbackground": [("!disabled", "green3")],
            "foreground": [("focus", "OliveDrab1"),
                           ("!disabled", "OliveDrab2")]
        }
    }
})

combo = ttk.Combobox().pack()

root.mainloop()
```

theme_names()
全ての既存のテーマのリストを返します。

theme_use(*themename=None*)
themename が与えられなかった場合は、現在使用中のテーマ名を返します。そうでない場合は、現在のテーマを *themename* に設定し、全てのウィジェットを再描画し、«ThemeChanged» イベントを発生させます。

レイアウト

レイアウトはオプションを取らない場合はただの None にできますし、そうでない場合は要素をどう配置するかを指定するオプションの辞書になります。レイアウト機構は単純化したジオメトリマネージャを使っています: 最初に空間が与えられ、それぞれの要素に分割された空間が配分されます。設定できるオプションと値は次の通りです:

- side: **辺の名前**　要素を空間のどちら側に配置するかを指定します; top, right, bottom, left のどれか 1 つです。省略された場合は、要素は空間全体を占めます。
- sticky: n, s, w, e **から 0 個以上**　配分された空間の内部に要素をどう配置するかを指定します。
- unit: 0 **か** 1　1 に設定されると、*Widget.identify()* などには要素とその子で単一の要素として扱われます。これは、グリップのついたスクロールバーサムのようなものに使われます。
- children: [**内部レイアウト**...]　要素の内部に配置する要素のリストを指定します。リストのそれぞれの要素はタプル (もしくは他のシーケンス型) で、それの 1 番目の要素はレイアウト名でそれ以降は *Layout* です。

25.3 tkinter.tix --- Tk の拡張ウィジェット

ソースコード: Lib/tkinter/tix.py

バージョン 3.6 で非推奨: この Tk 拡張は保守されておらず、新しいコードでは使うべきではありません。その代わりに *tkinter.ttk* を使ってください。

---

tkinter.tix (Tk Interface Extension) モジュールは豊富な追加ウィジェットを提供します。標準 Tk ライブ

ラリには多くの有用なウィジェットがありますが、決して完全ではありません。*tkinter.tix* ライブラリは、*HList* 、*ComboBox* 、*Control* (別名 SpinBox) および各種のスクロール可能なウィジェットなど、標準 Tk にはないが、一般的に必要とされるウィジェットの大部分を提供します。*tkinter.tix* には、class:*NoteBook* 、*FileEntry* 、*PanedWindow* など、一般的に幅広い用途に役に立つたくさんのウィジェットも含まれています。それらは 40 以上あります。

これら多くの新しいウィジェットと使うと、より便利でより直感的なユーザインタフェースを作成し、新しい相互作用テクニックをアプリケーションに導入することができます。アプリケーションとユーザに特有の要求に合うように、最も適切なアプリケーションウィジェットを選んでアプリケーションを設計できます。

参考:

Tix Homepage　Tix のホームページ。ここには追加ドキュメントとダウンロードへのリンクがあります。

Tix Man Pages　man ページと参考資料のオンライン版。

Tix Programming Guide　プログラマ用参考資料のオンライン版。

Tix Development Applications　Tix と Tkinter プログラムの開発のための Tix アプリケーション。Tide アプリケーションは Tk または Tkinter に基づいて動作します。また、リモートで Tix/Tk/Tkinter アプリケーションを変更やデバッグするためのインスペクタ **TixInspect** が含まれます。

25.3.1 Tix を使う

class tkinter.tix.Tk(*screenName=None*, *baseName=None*, *className='Tix'*)

主にアプリケーションのメインウィンドウを表す Tix のトップレベルウィジェット。Tcl インタープリタが関連付けられます。

tkinter.tix モジュールのクラスは *tkinter* モジュールのクラスをサブクラス化します。前者は後者をインポートするので、モジュールを一つインポートするだけで *tkinter.tix* を Tkinter と一緒に使うことができます。一般的に、*tkinter.tix* をインポートし、トップレベルでの *tkinter.Tk* への呼び出しは `tix.Tk` に置き換えるだけで済みます。次に例を示します:

```
from tkinter import tix
from tkinter.constants import *
root = tix.Tk()
```

tkinter.tix を使うためには、通常 Tk ウィジェットのインストールと平行して、Tix ウィジェットをインストールしなければなりません。インストールをテストするために、次のことを試してください:

```
from tkinter import tix
root = tix.Tk()
root.tk.eval('package require Tix')
```

25.3.2 Tix ウィジェット

Tix は 40 個以上のウィジェットクラスを *tkinter* のレパートリーに導入します。

基本ウィジェット

class tkinter.tix.Balloon

ヘルプを提示するためにウィジェット上にポップアップする Balloon。ユーザがカーソルを Balloon ウィジェットが束縛されているウィジェット内部へ移動させたとき、説明のメッセージが付いた小さなポップアップウィンドウがスクリーン上に表示されます。

class tkinter.tix.ButtonBox

ButtonBox ウィジェットは、`Ok Cancel` のためだけに普通は使われるようなボタンボックスを作成します。

class tkinter.tix.ComboBox

ComboBox ウィジェットは MS Windows のコンボボックスコントロールに似ています。ユーザはエントリ・サブウィジェットでタイプするか、リストボックス・サブウィジェットから選択するかのどちらかで選択肢を選びます。

class tkinter.tix.Control

Control ウィジェットは `SpinBox` ウィジェットとしても知られています。ユーザは二つの矢印ボタンを押すか、またはエントリに直接値を入力して値を調整します。新しい値をユーザが定義した上限と下限に対してチェックします。

class tkinter.tix.LabelEntry

LabelEntry ウィジェットはエントリウィジェットとラベルを一つのメガウィジェットにまとめたものです。" 記入形式" 型のインタフェースの作成を簡単に行うために使うことができます。

class tkinter.tix.LabelFrame

LabelFrame ウィジェットはフレームウィジェットとラベルを一つのメガウィジェットにまとめたものです。LabelFrame ウィジェット内部にウィジェットを作成するためには、`frame` サブウィジェットに対して新しいウィジェットを作成し、それらを `frame` サブウィジェット内部で取り扱います。

class tkinter.tix.Meter

Meter ウィジェットは実行に時間のかかるバックグラウンド・ジョブの進み具合を表示するために使用できます。

class tkinter.tix.OptionMenu

OptionMenu はオプションのメニューボタンを作成します。

class tkinter.tix.PopupMenu

PopupMenu ウィジェットは `tk_popup` コマンドの代替品として使用できます。`Tix` *PopupMenu* ウィジェットの利点は、操作に必要なアプリケーション・コードが少ないことです。

class tkinter.tix.Select

Select ウィジェットはボタン・サブウィジェットのコンテナです。ユーザに対する選択オプションのラジオボックスまたはチェックボックス形式を提供するために利用することができます。

class tkinter.tix.StdButtonBox

StdButtonBox ウィジェットは、Motif に似たダイアログボックスのための標準的なボタンのグループです。

ファイルセレクタ

class tkinter.tix.DirList

DirList ウィジェットは、ディレクトリのリストビュー (上の階層のディレクトリとサブディレクトリ) を表示します。ユーザはリスト内に表示されたディレクトリの一つを選択したり、他のディレクトリへ変更したりできます。

class tkinter.tix.DirTree

DirTree ウィジェットはディレクトリのツリービュー (上の階層のディレクトリとそのサブディレクトリ) を表示します。ユーザはリスト内に表示されたディレクトリの一つを選択したり、他のディレクトリへ変更したりできます。

class tkinter.tix.DirSelectDialog

DirSelectDialog ウィジェットは、ダイアログウィンドウにファイルシステム内のディレクトリを提示します。ユーザはこのダイアログウィンドウを使用して、ファイルシステム内を移動して目的のディレクトリを選択することができます。

class tkinter.tix.DirSelectBox

DirSelectBox は標準 Motif(TM) ディレクトリ選択ボックスに似ています。ユーザがディレクトリを選択するために一般的に使われます。DirSelectBox は主に最近 ComboBox ウィジェットに選択されたディレクトリを保存し、すばやく再選択できるようにします。

class tkinter.tix.ExFileSelectBox

ExFileSelectBox ウィジェットは、たいてい tixExFileSelectDialog ウィジェット内に組み込まれます。ユーザがファイルを選択するのに便利なメソッドを提供します。*ExFileSelectBox* ウィジェットのスタイルは、MS Windows 3.1 の標準ファイルダイアログにとてもよく似ています。

class tkinter.tix.FileSelectBox

FileSelectBox は標準的な Motif(TM) ファイル選択ボックスに似ています。通常、ユーザがファイルを選択するために使われます。FileSelectBox は、直近に *ComboBox* ウィジェットに選択されたファイルを保存し、素早く再選択できるようにします。

class tkinter.tix.FileEntry

FileEntry ウィジェットは、ファイル名を入力するために使うことができます。ユーザはファイル名を手入力できます。その代わりに、エントリの横に並んでいるボタンウィジェットを押すと表示されるファイル選択ダイアログを使用することもできます。

階層的リストボックス

class tkinter.tix.HList

HList ウィジェットは階層構造をもつどんなデータ (例えば、ファイルシステムディレクトリツリー) でも表示するために使用できます。リストエントリは字下げされ、階層のそれぞれの場所に応じて分岐線で接続されます。

class tkinter.tix.CheckList

CheckList ウィジェットは、ユーザが選ぶ項目のリストを表示します。CheckList は Tk のチェックリストやラジオボタンより多くの項目を扱うことができることを除いて、チェックボタンあるいはラジオボタンウィジェットと同じように動作します。

class tkinter.tix.Tree

Tree ウィジェットは階層的なデータをツリー形式で表示するために使うことができます。ユーザはツリーの一部を開いたり閉じたりすることによって、ツリーの見えを調整できます。

表的リストボックス

class tkinter.tix.TList

TList ウィジェットは、表形式でデータを表示するために使うことができます。*TList* ウィジェットのリスト・エントリは、Tk のリストボックス・ウィジェットのエントリに似ています。主な差は、(1) *TList* ウィジェットはリスト・エントリを二次元形式で表示でき、(2) リスト・エントリに対して複数の色やフォントだけでなく画像も使うことができるということです。

管理ウィジェット

class tkinter.tix.PanedWindow

PanedWindow ウィジェットは、ユーザがいくつかのペインのサイズを対話的に操作できるようにします。ペインは垂直または水平のどちらかに配置されます。ユーザは二つのペインの間でリサイズ・ハンドルをドラッグしてペインの大きさを変更します。

class tkinter.tix.ListNoteBook

ListNoteBook ウィジェットは、TixNoteBook ウィジェットにとてもよく似ています。ノートのメタファを使って限られた空間をに多くのウィンドウを表示するために使われます。ノートはたくさんのページ (ウィンドウ) に分けられています。ある時には、これらのページの一つしか表示できません。ユーザは hlist サブウィジェットの中の望みのページの名前を選択することによって、これらのページを切り替えることができます。

class tkinter.tix.NoteBook

NoteBook ウィジェットは、ノートのメタファを多くのウィンドウを表示することができます。ノートはたくさんのページに分けられています。ある時には、これらのページの一つしか表示できません。ユーザは NoteBook ウィジェットの一番上にある目に見える” タブ” を選択することで、これらのページを切り替えることができます。

画像タイプ

tkinter.tix モジュールは次のものを追加します:

- 全ての *tkinter.tix* と *tkinter* ウィジェットに対して XPM ファイルからカラー画像を作成する pixmap 機能。
- Compound 画像タイプは複数の水平方向の線から構成される画像を作成するために使うことができます。それぞれの線は左から右に並べられた一連のアイテム (テキスト、ビットマップ、画像あるいは空白) から作られます。例えば、Tk の Button ウィジェットの中にビットマップとテキスト文字列を同時に表示するために compound 画像は使うことができます。

その他のウィジェット

class tkinter.tix.InputOnly

InputOnly ウィジェットは、ユーザから入力を受け付けます。それは、bind コマンドを使って行われます (Unix のみ)。

ジオメトリマネジャを作る

加えて、*tkinter.tix* は次のものを提供することで *tkinter* を補強します:

class tkinter.tix.Form

すべての Tk ウィジェットに対する接続ルールに基づいた Form ジオメトリマネジャ。

25.3.3 Tix コマンド

class tkinter.tix.tixCommand

tix コマンド は Tix の内部状態と Tix アプリケーション・コンテキストのいろいろな要素へのアクセスを提供します。これらのメソッドによって操作される情報の大部分は、特定のウィンドウというよりむしろアプリケーション全体かスクリーンあるいはディスプレイに関するものです。

現在の設定を見るための一般的な方法は:

```
from tkinter import tix
root = tix.Tk()
print(root.tix_configure())
```

tixCommand.tix_configure(*cnf=None*, ***kw*)

Tix アプリケーション・コンテキストの設定オプションを問い合わせたり、変更したりします。オプションが指定されなければ、利用可能なオプションすべてのディクショナリを返します。オプションが値なしで指定された場合は、メソッドは指定されたオプションを説明するリストを返します (このリストはオプションが指定されていない場合に返される値に含まれている、指定されたオプションに対応するサブリストと同一です)。一つ以上のオプション-値のペアが指定された場合は、メソッドは与えられたオプションが与えられた値を持つように変更します。この場合は、メソッドは空文字列を返します。オプションは設定オプションのどれでも構いません。

tixCommand.**tix_cget**(*option*)

option によって与えられた設定オプションの現在の値を返します。オプションは設定オプションのどれでも構いません。

tixCommand.**tix_getbitmap**(*name*)

ビットマップディレクトリの一つの中の `name.xpm` または `name` と言う名前のビットマップファイルの場所を見つけ出します (*tix_addbitmapdir()* メソッドを参照してください)。*tix_getbitmap()* を使うことで、アプリケーションにビットマップファイルのパス名をハードコーディングすることを避けることができます。成功すれば、文字 `@` を先頭に付けたビットマップファイルの完全なパス名を返します。戻り値を Tk と Tix ウィジェットの `bitmap` オプションを設定するために使うことができます。

tixCommand.**tix_addbitmapdir**(*directory*)

Tix は *tix_getimage()* と *tix_getbitmap()* メソッドが画像ファイルを検索する検索先ディレクトリのリストを保持しています。標準ビットマップディレクトリは `$TIX_LIBRARY/bitmaps` です。*tix_addbitmapdir()* メソッドは *directory* をこのリストに追加します。そのメソッドを使うことによって、*tix_getimage()* または *tix_getbitmap()* メソッドを使ってアプリケーションの画像ファイルも見つけることができます。

tixCommand.**tix_filedialog**(*[dlgclass]*)

このアプリケーションからの異なる呼び出しの間で共有される可能性があるファイル選択ダイアログを返します。最初に呼ばれた時に、このメソッドはファイル選択ダイアログ・ウィジェットを作成します。このダイアログはその後のすべての *tix_filedialog()* への呼び出しで返されます。オプションの dlgclass パラメータは、要求されているファイル選択ダイアログ・ウィジェットの型を指定するために文字列として渡されます。指定可能なオプションは `tix` 、`FileSelectDialog` あるいは `tixExFileSelectDialog` です。

tixCommand.**tix_getimage**(*self*, *name*)

ビットマップディレクトリの一つの中の `name.xpm` 、`name.xbm` または `name.ppm` という名前の画像ファイルの場所を見つけ出します (上の *tix_addbitmapdir()* メソッドを参照してください)。同じ名前 (だが異なる拡張子) のファイルが一つ以上ある場合は、画像のタイプが X ディスプレイの深さに応じて選択されます。xbm 画像はモノクロディスプレイの場合に選択され、カラー画像はカラーディスプレイの場合に選択されます。*tix_getimage()* を使うことによって、アプリケーションに画像ファイルのパス名をハードコーディングすることを避けられます。成功すれば、このメソッドは新たに作成した画像の名前を返し、Tk と Tix ウィジェットの `image` オプションを設定するためにそれを使うことができます。

tixCommand.**tix_option_get**(*name*)

Tix のスキーム・メカニズムによって保持されているオプションを得ます。

tixCommand.**tix_resetoptions**(*newScheme*, *newFontSet*[, *newScmPrio*])

Tix アプリケーションのスキームとフォントセットをそれぞれ *newScheme* と *newFontSet* に再設定します。この設定は、この呼び出し後に作成されたウィジェットだけに影響します。そのため、Tix アプリケーションのどのウィジェットを作成する前にも resetoptions メソッドを呼び出すのが良い方法です。

オプション・パラメータ *newScmPrio* を、Tix スキームによって設定される Tk オプションの優先度レベルを再設定するために与えることができます。

Tk が X オプションデータベースを扱う方法のため、Tix がインポートされ初期化された後に、カラースキームとフォントセットを tix_config() メソッドを使って再設定することはできません。したがって、*tix_resetoptions()* メソッドを代わりに使わなければならないのです。

25.4 tkinter.scrolledtext --- スクロールするテキストウィジェット

ソースコード: Lib/tkinter/scrolledtext.py

---

tkinter.scrolledtext モジュールは” 正しい動作” をするように設定された垂直スクロールバーをもつ基本的なテキストウィジェットを実装する同じ名前のクラスを提供します。ScrolledText クラスを使うことは、テキストウィジェットとスクロールバーを直接設定するより簡単です。コンストラクタは tkinter.Text クラスのものを同じです。

テキストウィジェットとスクロールバーは Frame の中に一緒に pack され、Grid と Pack ジオメトリマネジャのメソッドは Frame オブジェクトから得られます。これによって、もっとも標準的なジオメトリマネジャの振る舞いにするために、直接 ScrolledText ウィジェットを使えるようになります。

特定の制御が必要ならば、以下の属性が利用できます:

ScrolledText.frame
: テキストとスクロールバーウィジェットを取り囲むフレーム。

ScrolledText.vbar
: スクロールバーウィジェット。

25.5 IDLE

ソースコード: Lib/idlelib/

IDLE は Python の統合開発環境で、学習用環境です。

IDLE は次のような特徴があります:

- *tkinter* GUI ツールキットを使って、100% ピュア Python でコーディングされています
- クロスプラットホーム: Windows, Unix, macOS で動作します
- コード入力、出力、エラーメッセージの色付け機能を持った Python shell (対話的インタプリタ) ウィンドウ
- 多段 Undo、Python 対応の色づけ、自動的な字下げ、呼び出し情報の表示、自動補完、他たくさんの機能をもつマルチウィンドウ・テキストエディタ
- 任意のウィンドウ内での検索、エディタウィンドウ内での置換、複数ファイルを跨いだ検索 (grep)
- 永続的なブレイクポイント、ステップ実行、グローバルとローカル名前空間の視覚化機能を持ったデバッガ

- 設定、ブラウザ群、ほかダイアログ群

25.5.1 メニュー

IDLE には 2 種類のメインウィンドウ、Shell ウィンドウと Editor ウィンドウがあります。Editor ウィンドウは同時に複数開けます。Windows と Linux では、それぞれ個別のトップメニューがあります。後で解説するそれぞれのメニューは、ウィンドウがどちらのウィンドウタイプなのかを示しています。

ファイル内の編集 (Edit) => 検索 (Find) で使われるような出力ウィンドウ (Output ウィンドウ) は editor ウィンドウのサブタイプで、Editor ウィンドウと同じトップメニューを持ちますが、デフォルトタイトルとコンテキストメニューが違います。

On macOS, there is one application menu. It dynamically changes according to the window currently selected. It has an IDLE menu, and some entries described below are moved around to conform to Apple guidelines.

File メニュー (Shell ウィンドウ、Editor ウィンドウ)

New File [**新規ファイル**] 新しいファイル編集ウィンドウを作成します。

Open... [**開く**...] Open ダイアログを使って既存のファイルをオープンします。

Recent Files [**最近使ったファイル**] 最近使ったファイルのリストを開きます。ファイルを一つクリックするとそれを開きます。

Open Module... [**モジュールを開く**...] 既存のモジュールをオープンします (sys.path を検索します)。

Class Browser [**クラスブラウザ**] 現在 Editor が開いているファイル内にあるクラス、関数、メソッドを木構造で可視化します。Shell からの場合は、先にモジュール選択のダイアログが開きます。

Path Browser [**パスブラウザ**] sys.path ディレクトリ、モジュール、クラスおよびメソッドを木構造で可視化します。

Save [**保存**] 現在のウィンドウを対応するファイルがあればそこに保存します。開いてから、または最後に保存したのちに編集があった場合のウィンドウには、ウィンドウタイトルの前後に * が付けられます。対応するファイルがなければ代わりに Save As が実行されます。

Save As... [**名前を付けて保存**...] Save As ダイアログを使って現在のウィンドウを保存します。保存されたファイルがウィンドウに新しく対応するファイルになります。

Save Copy As... [**コピーとして保存**...] 現在のウィンドウを対応するファイルを変えずに異なるファイルに保存します。

Print Window [**ウィンドウを印刷**] 現在のウィンドウをデフォルトプリンタで印刷します。

Close [**閉じる**] 現在のウィンドウを閉じます (未保存の場合は保存するか質問します)。

Exit [**終了**] すべてのウィンドウを閉じて IDLE を終了します (未保存の場合は保存するか質問します)。

Edit メニュー (Shell ウィンドウ、Editor ウィンドウ)

Undo [**元に戻す**] 現在のウィンドウに対する最後の変更を Undo (取り消し) します。最大で 1000 個の変更が Undo できます。

Redo [**やり直し**] 現在のウィンドウに対する最後に undo された変更を Redo(再スタート) します。

Cut [**切り取り**] システムのクリップボードへ選択された部分をコピーします。それから選択された部分を削除します。

Copy [**コピー**] 選択された部分をシステムのクリップボードへコピーします。

Paste [**貼り付け**] システムのクリップボードの内容をカレントウィンドウへ挿入します。

クリップボードの機能はコンテキストメニューからも使えます。

Select All [**全て選択**] カレントウィンドウの内容全体を選択します。

Find... [**検索**...] たくさんのオプションをもつ検索ダイアログボックスを開きます。

Find Again [**再検索**] 直前の検索があれば、それを繰り返します。

Find Selection [**現在の選択を検索**] 現在選択された文字列があれば、それを検索します。

Find in Files... [**ファイルから検索**...] ファイル検索ダイアログを開きます。結果を新しい出力ウィンドウに出力します。

Replace... [**置換**...] 検索と置換ダイアログを開きます。

Go to Line [**指定行へジャンプ**] Move the cursor to the beginning of the line requested and make that line visible. A request past the end of the file goes to the end. Clear any selection and update the line and column status.

Show Completions [**補完候補の一覧**] 既存の名前が選択できるスクロール可能なリストを開きます。下の編集とナビゲーションの節にある **補完** の項を参照してください。

Expand Word [**語の展開**] 先頭だけタイプしたものを、同一ウィンドウ内の完全な語と合致するものに展開します。異なる展開を得るためには繰り返します。

Show call tip [**呼び出し方ヒントの表示**] 関数の開き括弧の後ろで、関数パラメータについてのヒントを表示する小さなウィンドウを開きます。下の編集とナビゲーションの節にある **呼び出しヒント** の項を参照してください。

Show surrounding parens [**囲んでいる括弧の強調**] 囲んでいる括弧をハイライトします。

Format メニュー (Shell ウィンドウ、Editor ウィンドウ)

Indent Region [**領域をインデント**] 選択された行を右へインデント幅分シフトします (デフォルトは空白 4 個)。

Dedent Region [**領域をインデント解除**] 選択された行を左へインデント幅分シフトします (デフォルトは空白 4 個)。

Comment Out Region [**領域をコメントアウト**] 選択された行の先頭に ## を挿入します。

Uncomment Region [**領域のコメントを解除**] 選択された行から先頭の # あるいは ## を取り除きます。

Tabify Region [**領域のタブ化**] **先頭** の一続きの空白をタブに置き換えます (注意: Python コードのインデントには 4 つの空白を使うことをお勧めします)。

Untabify Region [**領域の非タブ化**] **すべての** タブを適切な数の空白に置き換えます。

Toggle Tabs [**タブの切り替え**] 字下げのために空白を使うかタブを使うかを切り替えるダイアログを開きます。

New Indent Width [**新しいインデント幅**] インデント幅を変更するダイアログを開きます。Python コミュニティによって受け容れられているデフォルトは空白 4 個です。

Format Paragraph [**パラグラフのフォーマット**] コメント内、マルチライン文字列リテラル内、あるいは選択行の現在位置の (空行区切り) パラグラフの再整形。パラグラフ内の全ての行は N カラム (デフォルトは 72) 以内で再整形されます。

Strip trailing whitespace [**末尾の空白を取り除く**] Remove trailing space and other whitespace characters after the last non-whitespace character of a line by applying str.rstrip to each line, including lines within multiline strings. Except for Shell windows, remove extra newlines at the end of the file.

Run メニュー (Editor ウィンドウのみ)

Run Module [**モジュールの実行**] *Check Module* を実行します。エラーがなければ Shell の環境をクリーンにして再スタートした上で、モジュールを実行します。出力は Shell ウィンドウに表示されます。`print` や `write` しない限り、この出力はされません。モジュール実行が完了すると Shell はフォーカスされた状態のままで、プロンプトを表示します。これにより対話的に実行結果を調べることができます。この機能は、コマンドラインからファイルを `python -i file` で実行することに相当します。

Run... Customized Same as *Run Module*, but run the module with customized settings. *Command Line Arguments* extend *`sys.argv`* as if passed on a command line. The module can be run in the Shell without restarting.

Check Module [**モジュールのチェック**] Editor ウィンドウでいま開いているモジュールを構文チェックします。モジュールが未保存の場合、*Options -> Configure IDLE -> General* の "Autosave Preferences" の設定にもとづき、確認を求められるか自動的に保存します。構文エラーが見つかると Editor ウィンドウでそのおよその位置に移動します。

Python Shell [Python **シェル**] Python Shell ウィンドウを開くか、起こします。

Shell メニュー (Shell ウィンドウのみ)

View Last Restart [最後のリスタートを表示する] 最後に Shell のリスタートを行った場所まで Shell ウインドウをスクロールします。

Restart Shell [Shell のリスタート] Restart the shell to clean the environment and reset display and exception handling.

Previous History Cycle through earlier commands in history which match the current entry.

Next History Cycle through later commands in history which match the current entry.

Interrupt Execution [実行の中断] プログラムの実行を停止します。

Debug メニュー (Shell ウィンドウのみ)

Go to File/Line [ファイル/行へ移動] カーソルのある現在行の上にファイル名と行番号が見つかれば、(開かれていなければ) そのファイルを開いてその行に飛びます。例外のトレースバックが参照しているソース行を見るのにこれを使いましょう。そのようなファイル名・行番号表示をしている行を見つけるのには Find を使えます。この機能は Shell ウィンドウと Output ウィンドウのコンテキストメニューからも使えます。

Debugger [デバッガ] (トグル切り替え) アクティブにすると、Shell または Editor に入力したコードをデバッガのもとで実行します。Editor ではコンテキストメニューからブレイクポイントをセット出来ます。この機能はまだ不完全で、少々実験的です。

Stack Viewer [スタックビューア] 最後の例外のスタックトレースと locals 辞書、globals 辞書をツリーウィジットで表示します。

Auto-open Stack Viewer [スタックビューアの自動オープン] 未捕捉の例外時にスタックビューアを自動的に開くかどうかを切り替えます。

Options メニュー (Shell ウィンドウ、Editor ウィンドウ)

Configure IDLE [IDLE の設定] Open a configuration dialog and change preferences for the following: fonts, indentation, keybindings, text color themes, startup windows and size, additional help sources, and extensions. On macOS, open the configuration dialog by selecting Preferences in the application menu. For more details, see *Setting preferences* under Help and preferences.

Most configuration options apply to all windows or all future windows. The option items below only apply to the active window.

Show/Hide Code Context (Editor Window only) Editor ウィンドウの上部にペインが開き、そこにウィンドウの一番上のコードのブロックコンテキスト (訳注: コードが含まれているブロックのインデント構造) が表示されます。下の編集とナビゲーションの節にある **コードコンテキスト** の項を参照してください。

Show/Hide Line Numbers (Editor Window only) Open a column to the left of the edit window which shows the number of each line of text. The default is off, which may be changed in the preferences (see *Setting preferences*).

Zoom/Restore Height　Toggles the window between normal size and maximum height. The initial size defaults to 40 lines by 80 chars unless changed on the General tab of the Configure IDLE dialog. The maximum height for a screen is determined by momentarily maximizing a window the first time one is zoomed on the screen. Changing screen settings may invalidate the saved height. This toggle has no effect when a window is maximized.

Window メニュー (Shell ウィンドウ、Editor ウィンドウ)

Lists the names of all open windows; select one to bring it to the foreground (deiconifying it if necessary).

Help メニュー (Shell ウィンドウ、Editor ウィンドウ)

About IDLE [IDLE **について**]　バージョン、コピーライト、ライセンス、クレジット、その他を表示します。

IDLE Help [IDLE **ヘルプ**]　メニューオプション、基本的な編集やナビゲーションその他チップスを詳しく書いた IDLE のドキュメントを表示します。

Python Docs [Python **ドキュメント**]　Python ドキュメントがローカルにインストールされていればそれを開きます。もしくはウェブブラウザで最新の Python ドキュメント (https://docs.python.org) を開きます。(訳注: いずれも何もしていなければ英語。下記参照。)

Turtle Demo [Turtle **デモ**]　Python コード例とタートルによる描画を使って turtledemo モジュールを実行します。

General タブの下の方にある Configure IDLE ダイアログで、追加のヘルプソースを Help メニューに加えられます。下の編集とナビゲーションの節にある **ヘルプソース** の項を参照してください。

コンテキストメニュー

ウィンドウ内で右クリック (macOS では Control-クリック) でコンテキストメニューが開きます。コンテキストメニューには Edit メニューにもある標準的なクリップボード機能が含まれています。

Cut [**切り取り**]　システムのクリップボードへ選択された部分をコピーします。それから選択された部分を削除します。

Copy [**コピー**]　選択された部分をシステムのクリップボードへコピーします。

Paste [**貼り付け**]　システムのクリップボードの内容をカレントウィンドウへ挿入します。

Editor ウィンドウではさらにブレイクポイント機能が使えます。ブレイクポイントがセットされた行には、特別に印がつきます。ブレイクポイントはデバッガのもとでの実行にだけ影響します。ファイルに付けたブレイクポイントはユーザの `.idlerc` ディレクトリに保存されます。

Set Breakpoint [**ブレークポイントのセット**]　現在行にブレークポイントをセットします。

Clear Breakpoint [**ブレークポイントのクリア**]　その行のブレークポイントをクリアします。

Shell ウィンドウや Output ウィンドウには次のメニューもあります。

Go to file/line [**ファイル/行へ移動**]　Debug メニューと同じものです。

The Shell window also has an output squeezing facility explained in the *Python Shell window* subsection below.

Squeeze　If the cursor is over an output line, squeeze all the output between the code above and the prompt below down to a 'Squeezed text' label.

25.5.2 編集とナビゲーション

Editor windows

IDLE may open editor windows when it starts, depending on settings and how you start IDLE. Thereafter, use the File menu. There can be only one open editor window for a given file.

The title bar contains the name of the file, the full path, and the version of Python and IDLE running the window. The status bar contains the line number ('Ln') and column number ('Col'). Line numbers start with 1; column numbers with 0.

IDLE assumes that files with a known .py* extension contain Python code and that other files do not. Run Python code with the Run menu.

Key bindings

ここでの説明で 'C' は、Windows と Unix の場合は `Control` キー、macOS では `Command` キーを示します。

- `Backspace` は左側を削除し、`Del` は右側を削除します。
- `C-Backspace` は語単位で左側を削除、`C-Del` は語単位で右側を削除します。
- 矢印キーと `Page Up`/`Page Down` はそれぞれその通りに移動します。
- `C-LeftArrow` と `C-RightArrow` は語単位で移動します。
- `Home`/`End` は行の始め/終わりへ移動します。
- `C-Home`/`C-End` はファイルの始め/終わりへ移動します。
- いくつかの有用な Emacs バインディングが Tcl/Tk から継承されています:
  - `C-a` で行頭へ移動。
  - `C-e` で行末へ移動。
  - `C-k` で行を削除 (ただしクリップボードには入りません)。
  - `C-l` で挿入ポイントをウィンドウの中心にする。
  - `C-b` で一文字分文字削除なしで戻る (通常、これはカーソルキーでもできます)。
  - `C-f` で一文字分文字削除なしで進む (通常、これはカーソルキーでもできます)。
  - `C-p` で一行上へ移動 (通常、これはカーソルキーでもできます)。

– C-d で次の文字を削除。

標準的なキーバインディング (C-c がコピーで C-v がペースト、など) は機能するかもしれません。キーバインディングは Configure IDLE ダイアログで選択します。

自動的な字下げ

ブロックの始まりの文の後、次の行は 4 つの空白 (Python Shell ウィンドウでは、一つのタブ) で字下げされます。あるキーワード (break、return など) の後では、次の行は字下げが解除 (dedent) されます。先頭の字下げでは、Backspace は 4 つの空白があれば削除します。Tab はインデント幅に対応する数の空白 (Python Shell ウィンドウでは一つのタブ) を挿入します。現在、タブは Tcl/Tk の制約のため 4 つの空白に固定されています。

Format メニュー の indent/dedent region コマンドも参照してください。

補完 (Completions)

Completions are supplied, when requested and available, for module names, attributes of classes or functions, or filenames. Each request method displays a completion box with existing names. (See tab completions below for an exception.) For any box, change the name being completed and the item highlighted in the box by typing and deleting characters; by hitting Up, Down, PageUp, PageDown, Home, and End keys; and by a single click within the box. Close the box with Escape, Enter, and double Tab keys or clicks outside the box. A double click within the box selects and closes.

One way to open a box is to type a key character and wait for a predefined interval. This defaults to 2 seconds; customize it in the settings dialog. (To prevent auto popups, set the delay to a large number of milliseconds, such as 100000000.) For imported module names or class or function attributes, type '.'. For filenames in the root directory, type *os.sep* or *os.altsep* immediately after an opening quote. (On Windows, one can specify a drive first.) Move into subdirectories by typing a directory name and a separator.

Instead of waiting, or after a box is closed, open a completion box immediately with Show Completions on the Edit menu. The default hot key is C-space. If one types a prefix for the desired name before opening the box, the first match or near miss is made visible. The result is the same as if one enters a prefix after the box is displayed. Show Completions after a quote completes filenames in the current directory instead of a root directory.

Hitting Tab after a prefix usually has the same effect as Show Completions. (With no prefix, it indents.) However, if there is only one match to the prefix, that match is immediately added to the editor text without opening a box.

Invoking 'Show Completions', or hitting Tab after a prefix, outside of a string and without a preceding '.' opens a box with keywords, builtin names, and available module-level names.

When editing code in an editor (as oppose to Shell), increase the available module-level names by running your code and not restarting the Shell thereafter. This is especially useful after adding imports at the top of a file. This also increases possible attribute completions.

Completion boxes intially exclude names beginning with '_' or, for modules, not included in '__all__'. The hidden names can be accessed by typing '_' after '.', either before or after the box is opened.

呼び出しヒント (Calltips)

A calltip is shown automatically when one types `(` after the name of an *accessible* function. A function name expression may include dots and subscripts. A calltip remains until it is clicked, the cursor is moved out of the argument area, or `)` is typed. Whenever the cursor is in the argument part of a definition, select Edit and "Show Call Tip" on the menu or enter its shortcut to display a calltip.

The calltip consists of the function's signature and docstring up to the latter's first blank line or the fifth non-blank line. (Some builtin functions lack an accessible signature.) A '/' or '*' in the signature indicates that the preceding or following arguments are passed by position or name (keyword) only. Details are subject to change.

In Shell, the accessible functions depends on what modules have been imported into the user process, including those imported by Idle itself, and which definitions have been run, all since the last restart.

For example, restart the Shell and enter `itertools.count(`. A calltip appears because Idle imports itertools into the user process for its own use. (This could change.) Enter `turtle.write(` and nothing appears. Idle does not itself import turtle. The menu entry and shortcut also do nothing. Enter `import turtle`. Thereafter, `turtle.write(` will display a calltip.

In an editor, import statements have no effect until one runs the file. One might want to run a file after writing import statements, after adding function definitions, or after opening an existing file.

Code Context

Within an editor window containing Python code, code context can be toggled in order to show or hide a pane at the top of the window. When shown, this pane freezes the opening lines for block code, such as those beginning with `class`, `def`, or `if` keywords, that would have otherwise scrolled out of view. The size of the pane will be expanded and contracted as needed to show the all current levels of context, up to the maximum number of lines defined in the Configure IDLE dialog (which defaults to 15). If there are no current context lines and the feature is toggled on, a single blank line will display. Clicking on a line in the context pane will move that line to the top of the editor.

The text and background colors for the context pane can be configured under the Highlights tab in the Configure IDLE dialog.

Python Shell ウィンドウ

With IDLE's Shell, one enters, edits, and recalls complete statements. Most consoles and terminals only work with a single physical line at a time.

When one pastes code into Shell, it is not compiled and possibly executed until one hits `Return`. One may edit pasted code first. If one pastes more that one statement into Shell, the result will be a *SyntaxError* when multiple statements are compiled as if they were one.

The editing features described in previous subsections work when entering code interactively. IDLE's Shell window also responds to the following keys.

- `C-c` で実行中のコマンドを中断します。
- `C-d` でファイル終端 (end-of-file) を送り、`>>>` プロンプトでタイプしていた場合はウィンドウを閉じます。
- `Alt-/` (語を展開します) もタイピングを減らすのに便利です。

  コマンド履歴

  - `Alt-p` は、以前のコマンドから検索します。macOS では `C-p` を使ってください。
  - `Alt-n` は、次を取り出します。macOS では `C-n` を使ってください。
  - `Return` は、以前のコマンドを取り出しているときは、そのコマンドを取り出します。

テキストの色

IDLE の表示はデフォルトで白背景に黒字ですが、以下のような特別な意味を持ったテキストには色が付きます。Shell では shell 出力、shell エラー、ユーザエラー。Python コードでは Shell プロンプト内や Editor でのキーワード、組み込みクラスや組み込み関数の名前、`class` や `def` に続く名前、文字列、そしてコメント。どんなテキストウィンドウでも、カーソル (あれば)、検索で合致したテキスト (あれば)、そして選択されているテキストには色が付きます。

このテキストの色付けはバックグラウンドで行われるため、たまに色が付いてない状態が見えてしまいます。カラースキームは、Configure IDLE [IDLE の設定] ダイアログの Highlighting タブで変更できます。ただし、エディタ内のデバッガブレークポイント行のマーキングと、ポップアップとダイアログないのテキストの色は、ユーザーにより変更することはできません。

25.5.3 スタートアップとコードの実行

`-s` オプションとともに起動すると、IDLE は環境変数 `IDLESTARTUP` か `PYTHONSTARTUP` で参照されているファイルを実行します。IDLE はまず `IDLESTARTUP` をチェックし、あれば参照しているファイルを実行します。`IDLESTARTUP` が無ければ、IDLE は `PYTHONSTARTUP` をチェックします。これらの環境変数で参照されているファイルは、IDLE シェルでよく使う関数を置いたり、一般的なモジュールの import 文を実行するのに便利です。

加えて、Tk もスタートアップファイルがあればそれをロードします。その Tk のファイルは無条件にロードされることに注意してください。このファイルは .Idle.py で、ユーザーのホームディレクトリから探されます。このファイルの中の文は Tk の名前空間で実行されるので、IDLE の Python シェルで使う関数を import するのには便利ではありません。

コマンドラインの使い方

```
idle.py [-c command] [-d] [-e] [-h] [-i] [-r file] [-s] [-t title] [-] [arg] ...

-c command  run command in the shell window
-d          enable debugger and open shell window
-e          open editor window
-h          print help message with legal combinations and exit
-i          open shell window
-r file     run file in shell window
-s          run $IDLESTARTUP or $PYTHONSTARTUP first, in shell window
-t title    set title of shell window
-           run stdin in shell (- must be last option before args)
```

引数がある場合 (訳注: 以下の説明、たぶん実情に反してますが一応訳しています):

- -, -c, -r のどれかを使う場合、全ての引数は sys.argv[1:...] に入り、sys.argv[0] には '', '-c', '-r' の、与えたものが入ります。Options ダイアログでデフォルトだったとしても Editor ウィンドウが開くことはありません。
- これ以外の場合は引数は編集対象のファイルとして開かれて、sys.argv には IDLE そのものに渡された引数が反映されます。

Startup failure

IDLE uses a socket to communicate between the IDLE GUI process and the user code execution process. A connection must be established whenever the Shell starts or restarts. (The latter is indicated by a divider line that says 'RESTART'). If the user process fails to connect to the GUI process, it usually displays a Tk error box with a 'cannot connect' message that directs the user here. It then exits.

One specific connection failure on Unix systems results from misconfigured masquerading rules somewhere in a system's network setup. When IDLE is started from a terminal, one will see a message starting with `** Invalid host:`. The valid value is `127.0.0.1 (idlelib.rpc.LOCALHOST)`. One can diagnose with `tcpconnect -irv 127.0.0.1 6543` in one terminal window and `tcplisten <same args>` in another.

A common cause of failure is a user-written file with the same name as a standard library module, such as *random.py* and *tkinter.py*. When such a file is located in the same directory as a file that is about to be run, IDLE cannot import the stdlib file. The current fix is to rename the user file.

Though less common than in the past, an antivirus or firewall program may stop the connection. If the program cannot be taught to allow the connection, then it must be turned off for IDLE to work. It is safe to allow this internal connection because no data is visible on external ports. A similar problem is a network mis-configuration that blocks connections.

Python installation issues occasionally stop IDLE: multiple versions can clash, or a single installation might need admin access. If one undo the clash, or cannot or does not want to run as admin, it might be easiest to completely remove Python and start over.

A zombie pythonw.exe process could be a problem. On Windows, use Task Manager to check for one and stop it if there is. Sometimes a restart initiated by a program crash or Keyboard Interrupt (control-C) may fail to connect. Dismissing the error box or using Restart Shell on the Shell menu may fix a temporary problem.

When IDLE first starts, it attempts to read user configuration files in `~/.idlerc/` (~ is one's home directory). If there is a problem, an error message should be displayed. Leaving aside random disk glitches, this can be prevented by never editing the files by hand. Instead, use the configuration dialog, under Options. Once there is an error in a user configuration file, the best solution may be to delete it and start over with the settings dialog.

If IDLE quits with no message, and it was not started from a console, try starting it from a console or terminal (`python -m idlelib`) and see if this results in an error message.

On Unix-based systems with tcl/tk older than `8.6.11` (see `About IDLE`) certain characters of certain fonts can cause a tk failure with a message to the terminal. This can happen either if one starts IDLE to edit a file with such a character or later when entering such a character. If one cannot upgrade tcl/tk, then re-configure IDLE to use a font that works better.

Running user code

With rare exceptions, the result of executing Python code with IDLE is intended to be the same as executing the same code by the default method, directly with Python in a text-mode system console or terminal window. However, the different interface and operation occasionally affect visible results. For instance, `sys.modules` starts with more entries, and `threading.active_count()` returns 2 instead of 1.

By default, IDLE runs user code in a separate OS process rather than in the user interface process that runs the shell and editor. In the execution process, it replaces `sys.stdin`, `sys.stdout`, and `sys.stderr` with objects that get input from and send output to the Shell window. The original values stored in `sys.__stdin__`, `sys.__stdout__`, and `sys.__stderr__` are not touched, but may be `None`.

Sending print output from one process to a text widget in another is slower than printing to a system terminal in the same process. This has the most effect when printing multiple arguments, as the string for each argument, each separator, the newline are sent separately. For development, this is usually not a problem, but if one wants to print faster in IDLE, format and join together everything one wants displayed together and then print a single string. Both format strings and *`str.join()`* can help combine fields and lines.

IDLE's standard stream replacements are not inherited by subprocesses created in the execution process, whether directly by user code or by modules such as multiprocessing. If such subprocess use `input` from sys.stdin or `print` or `write` to sys.stdout or sys.stderr, IDLE should be started in a command line window. The secondary subprocess will then be attached to that window for input and output.

If `sys` is reset by user code, such as with `importlib.reload(sys)`, IDLE's changes are lost and input from the keyboard and output to the screen will not work correctly.

When Shell has the focus, it controls the keyboard and screen. This is normally transparent, but functions that directly access the keyboard and screen will not work. These include system-specific functions that determine whether a key has been pressed and if so, which.

The IDLE code running in the execution process adds frames to the call stack that would not be there otherwise. IDLE wraps `sys.getrecursionlimit` and `sys.setrecursionlimit` to reduce the effect of the additional stack frames.

When user code raises SystemExit either directly or by calling sys.exit, IDLE returns to a Shell prompt instead of exiting.

User output in Shell

When a program outputs text, the result is determined by the corresponding output device. When IDLE executes user code, `sys.stdout` and `sys.stderr` are connected to the display area of IDLE's Shell. Some of its features are inherited from the underlying Tk Text widget. Others are programmed additions. Where it matters, Shell is designed for development rather than production runs.

For instance, Shell never throws away output. A program that sends unlimited output to Shell will eventually fill memory, resulting in a memory error. In contrast, some system text windows only keep the last n lines of output. A Windows console, for instance, keeps a user-settable 1 to 9999 lines, with 300 the default.

A Tk Text widget, and hence IDLE's Shell, displays characters (codepoints) in the BMP (Basic Multilingual Plane) subset of Unicode. Which characters are displayed with a proper glyph and which with a replacement box depends on the operating system and installed fonts. Tab characters cause the following text to begin after the next tab stop. (They occur every 8 'characters'). Newline characters cause following text to appear on a new line. Other control characters are ignored or displayed as a space, box, or something else, depending on the operating system and font. (Moving the text cursor through such output with arrow keys may exhibit some surprising spacing behavior.)

```
>>> s = 'a\tb\a<\x02><\r>\bc\nd'  # Enter 22 chars.
>>> len(s)
14
>>> s  # Display repr(s)
'a\tb\x07<\x02><\r>\x08c\nd'
>>> print(s, end='')  # Display s as is.
# Result varies by OS and font.  Try it.
```

The `repr` function is used for interactive echo of expression values. It returns an altered version of the input string in which control codes, some BMP codepoints, and all non-BMP codepoints are replaced with escape codes. As demonstrated above, it allows one to identify the characters in a string, regardless of how they are displayed.

Normal and error output are generally kept separate (on separate lines) from code input and each other.

They each get different highlight colors.

For SyntaxError tracebacks, the normal '^' marking where the error was detected is replaced by coloring the text with an error highlight. When code run from a file causes other exceptions, one may right click on a traceback line to jump to the corresponding line in an IDLE editor. The file will be opened if necessary.

Shell has a special facility for squeezing output lines down to a 'Squeezed text' label. This is done automatically for output over N lines (N = 50 by default). N can be changed in the PyShell section of the General page of the Settings dialog. Output with fewer lines can be squeezed by right clicking on the output. This can be useful lines long enough to slow down scrolling.

Squeezed output is expanded in place by double-clicking the label. It can also be sent to the clipboard or a separate view window by right-clicking the label.

Developing tkinter applications

IDLE is intentionally different from standard Python in order to facilitate development of tkinter programs. Enter `import tkinter as tk; root = tk.Tk()` in standard Python and nothing appears. Enter the same in IDLE and a tk window appears. In standard Python, one must also enter `root.update()` to see the window. IDLE does the equivalent in the background, about 20 times a second, which is about every 50 milliseconds. Next enter `b = tk.Button(root, text='button'); b.pack()`. Again, nothing visibly changes in standard Python until one enters `root.update()`.

Most tkinter programs run `root.mainloop()`, which usually does not return until the tk app is destroyed. If the program is run with `python -i` or from an IDLE editor, a `>>>` shell prompt does not appear until `mainloop()` returns, at which time there is nothing left to interact with.

When running a tkinter program from an IDLE editor, one can comment out the mainloop call. One then gets a shell prompt immediately and can interact with the live application. One just has to remember to re-enable the mainloop call when running in standard Python.

サブプロセスを起こさずに起動する

デフォルトでは、IDLE でのユーザコードの実行は、内部的なループバックインターフェイスを使用する、ソケット経由の分離されたサブプロセスで行われます。この接続は外部からは見えませんし、インターネットとのデータの送受信は行われません。ファイアウォールソフトウェアの警告が発生しても、無視して構いません。

ソケット接続の確立を試みて失敗した場合、IDLE によって通知されます。このような失敗は一過性の場合もありますが、永続的に失敗する場合は、ファイアウォールが接続をブロックしているか、特定のシステムの設定が誤っていることが原因かもしれません。問題が解決するまでは、IDLE をコマンドラインオプション `-n` で起動することもできます。

IDLE を `-n` コマンドラインスイッチを使って開始した場合、IDLE は単一のプロセス内で動作し、RPC Python 実行サーバを走らせるサブプロセスを作りません。これは、プラットフォーム上で Python がサブプロセスや RPC ソケットインターフェイスを作れない場合に有用かもしれません。ただし、このモードではユーザコードが IDLE 自身から隔離されませんし、Run/Run Module (F5) 選択時に環境が再起動されて

まっさらな状態になることもありません。コードを変更した場合、影響するモジュールを reload() しないといけませんし、変更を反映するには、すべての特定の項目 (`from foo import baz` など) を再インポートしないといけません。これらの理由から、可能なら常にデフォルトのサブプロセスを起こすモードで IDLE を起動するのが吉です。

バージョン 3.4 で非推奨.

25.5.4 ヘルプとお好み設定

Help sources

Help menu entry "IDLE Help" displays a formatted html version of the IDLE chapter of the Library Reference. The result, in a read-only tkinter text window, is close to what one sees in a web browser. Navigate through the text with a mousewheel, the scrollbar, or up and down arrow keys held down. Or click the TOC (Table of Contents) button and select a section header in the opened box.

Help menu entry "Python Docs" opens the extensive sources of help, including tutorials, available at `docs.python.org/x.y`, where 'x.y' is the currently running Python version. If your system has an off-line copy of the docs (this may be an installation option), that will be opened instead.

Selected URLs can be added or removed from the help menu at any time using the General tab of the Configure IDLE dialog.

Setting preferences [お好み設定]

The font preferences, highlighting, keys, and general preferences can be changed via Configure IDLE on the Option menu. Non-default user settings are saved in a `.idlerc` directory in the user's home directory. Problems caused by bad user configuration files are solved by editing or deleting one or more of the files in `.idlerc`.

On the Font tab, see the text sample for the effect of font face and size on multiple characters in multiple languages. Edit the sample to add other characters of personal interest. Use the sample to select monospaced fonts. If particular characters have problems in Shell or an editor, add them to the top of the sample and try changing first size and then font.

On the Highlights and Keys tab, select a built-in or custom color theme and key set. To use a newer built-in color theme or key set with older IDLEs, save it as a new custom theme or key set and it well be accessible to older IDLEs.

IDLE on macOS

Under System Preferences: Dock, one can set "Prefer tabs when opening documents" to "Always". This setting is not compatible with the tk/tkinter GUI framework used by IDLE, and it breaks a few IDLE features.

Extensions [拡張]

IDLE contains an extension facility. Preferences for extensions can be changed with the Extensions tab of the preferences dialog. See the beginning of config-extensions.def in the idlelib directory for further information. The only current default extension is zzdummy, an example also used for testing.

25.6 他のグラフィカルユーザインタフェースパッケージ

主要なクロスプラットフォーム (Windows, Mac OS X, Unix 系) GUI ツールキットで Python でも使えるものは:

参考:

PyGObject PyGObject は GObject の GObject introspection を利用するバインディングです。これらのライブラリの一つが GTK+ 3 ウィジットセットです。GTK+ には、Tkinter が提供するよりも多くのウィジェットが付属しています。オンラインで利用できる Python GTK+ 3 Tutorial があります。

PyGTK PyGTK はより古いバージョンであるライブラリ GTK+ 2 に対するバインディングを提供しています。それは C での実装よりはやや高級なオブジェクト指向インターフェイスを提供しています。GNOME へのバインディングもあります。オンラインで利用できる チュートリアル があります。

PyQt PyQt は `sip` でラップされた Qt ツールキットへのバインディングです。Qt は Unix、Windows および Mac OS X で利用できる大規模な C++ GUI ツールキットです。`sip` は C++ ライブラリに対するバインディングを Python クラスとして生成するためのツールで、Python 用に特化して設計されています。

PySide2 Python プロジェクトのための Qt としても知られている PySide2 は、Qt ツールキットへのより新しいバインディングです。これは The Qt Company から提供されており、PySide から Qt 5 への完全な移植を提供することを目標としています。PyQt と比較して、非オープンソースのアプリケーションでより扱いやすいライセンス形態になっています。

wxPython wxPython はクロスプラットフォームの Python 用 GUI ツールキットで、人気のある wxWidgets (旧名 wxWindows) C++ ツールキットに基づいて作られています。このツールキットは Windows, Mac OS X および Unix システムのアプリケーションに、それぞれのプラットフォームのネイティブなウィジェットを可能ならば利用して (Unix 系のシステムでは GTK+)、ネイティブなルック&フィールを提供します。多彩なウィジェットの他に、オンラインドキュメントや場面に応じたヘルプ、印刷、HTML 表示、低級デバイスコンテキスト描画、ドラッグ&ドロップ、システムクリップボードへのアクセス、XML に基づいたリソースフォーマット、さらにユーザ寄贈のモジュールからなる成長し続けているライブラリ等々を wxPython は提供しています。

PyGTK、PyQt、PySide2、wxPython は全て現代的なルック&フィールをそなえ、Tkinter より豊富なウィジェットを持ちます。これらに加えて、他にも Python 用 GUI ツールキットが、クロスプラットフォームのもの、プラットフォーム固有のものを含め、沢山あります。Python Wiki の GUI Programming ページも参照してください。もっとずっと完全なリストや、GUI ツールキット同士の比較をしたドキュメントへのリンクがあります。

第

TWENTYSIX

開発ツール

この章で紹介されるモジュールはソフトウェアを書くことを支援します。たとえば、*pydoc* モジュールはモジュールの内容からドキュメントを生成します。*doctest* と *unittest* モジュールでは、自動的に実行して予想通りの出力が生成されるか確認するユニットテストを書くことができます。`2to3` は Python2.x 用のソースコードを正当な Python 3.x コードに翻訳できます。

この章で解説されるモジュールのリスト:

26.1 typing --- 型ヒントのサポート

バージョン 3.5 で追加.

ソースコード: Lib/typing.py

注釈: Python ランタイムは、関数や変数の型アノテーションを強制しません。型アノテーションは、型チェッカー、IDE、linter などのサードパーティーツールで使われます。

このモジュールは **PEP 484**, **PEP 526**, **PEP 544**, **PEP 586**, **PEP 589**, **PEP 591** によって規定された型ヒントへのランタイムサポートを提供します。最も基本的なサポートとして *Any*、*Union*、*Tuple*、*Callable*、*TypeVar* および *Generic* 型を含みます。完全な仕様は **PEP 484** を参照してください。型ヒントの簡単な導入は **PEP 483** を参照してください。

以下の関数は文字列を受け取って文字列を返す関数で、次のようにアノテーションがつけられます:

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

関数 `greeting` で、実引数 `name` の型は *str* であり、返り値の型は *str* であることが期待されます。サブタイプも実引数として許容されます。

26.1.1 型エイリアス

型エイリアスは型をエイリアスに代入することで定義されます。この例では Vector と List[float] は交換可能な同義語として扱われます。

```
from typing import List
Vector = List[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# typechecks; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

型エイリアスは複雑な型シグネチャを単純化するのに有用です。例えば:

```
from typing import Dict, Tuple, Sequence

ConnectionOptions = Dict[str, str]
Address = Tuple[str, int]
Server = Tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: Sequence[Server]) -> None:
    ...

# The static type checker will treat the previous type signature as
# being exactly equivalent to this one.
def broadcast_message(
        message: str,
        servers: Sequence[Tuple[Tuple[str, int], Dict[str, str]]]) -> None:
    ...
```

型ヒントとしての None は特別なケースであり、type(None) によって置き換えられます。

26.1.2 NewType

異なる型を作るためには *NewType()* ヘルパー関数を使います:

```
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

静的型検査器は新しい型を元々の型のサブクラスのように扱います。この振る舞いは論理的な誤りを見つける手助けとして役に立ちます。

```
def get_user_name(user_id: UserId) -> str:
    ...
```

(次のページに続く)

(前のページからの続き)

```
# typechecks
user_a = get_user_name(UserId(42351))

# does not typecheck; an int is not a UserId
user_b = get_user_name(-1)
```

UserId 型の変数も int の全ての演算が行えますが、その結果は常に int 型になります。この振る舞いにより、int が期待されるところに UserId を渡せますが、不正な方法で UserId を作ってしまうことを防ぎます。

```
# 'output' is of type 'int', not 'UserId'
output = UserId(23413) + UserId(54341)
```

これらのチェックは静的型検査器のみによって強制されるということに注意してください。実行時に Derived = NewType('Derived', Base) という文は渡された仮引数をただちに返す Derived 関数を作ります。つまり Derived(some_value) という式は新しいクラスを作ることはなく、通常の関数呼び出し以上のオーバーヘッドがないということを意味します。

より正確に言うと、式 some_value is Derived(some_value) は実行時に常に真を返します。

これは Derived のサブタイプを作ることが出来ないということも意味しています。Derived は実行時には恒等関数になっていて、実際の型ではないからです:

```
from typing import NewType

UserId = NewType('UserId', int)

# Fails at runtime and does not typecheck
class AdminUserId(UserId): pass
```

しかし、'derived' である NewType をもとにした *NewType()* は作ることが出来ます:

```
from typing import NewType

UserId = NewType('UserId', int)

ProUserId = NewType('ProUserId', UserId)
```

そして ProUserId に対する型検査は期待通りに動作します。

より詳しくは **PEP 484** を参照してください。

注釈: 型エイリアスの使用は二つの型が互いに **等価** だと宣言している、ということを思い出してください。Alias = Original とすると、静的型検査器は Alias をすべての場合において Original と **完全に等価** なものとして扱います。これは複雑な型シグネチャを単純化したい時に有用です。

これに対し、NewType はある型をもう一方の型の **サブタイプ** として宣言します。Derived = NewType('Derived', Original) とすると静的型検査器は Derived を Original の **サブクラス** とし

て扱います。つまり Original 型の値は Derived 型の値が期待される場所で使うことが出来ないということです。これは論理的な誤りを最小の実行時のコストで防ぎたい時に有用です。

バージョン 3.5.2 で追加.

26.1.3 呼び出し可能オブジェクト

特定のシグネチャを持つコールバック関数を要求されるフレームワークでは、Callable[[Arg1Type, Arg2Type], ReturnType] を使って型ヒントを付けます。

例えば:

```
from typing import Callable

def feeder(get_next_item: Callable[[], str]) -> None:
    # Body

def async_query(on_success: Callable[[int], None],
                on_error: Callable[[int, Exception], None]) -> None:
    # Body
```

型ヒントの実引数の型を ellipsis で置き換えることで呼び出しシグニチャを指定せずに callable の戻り値の型を宣言することができます: Callable[..., ReturnType]。

26.1.4 ジェネリクス

コンテナ内のオブジェクトについての型情報は一般的な方法では静的に推論できないため、抽象基底クラスを継承したクラスが実装され、期待されるコンテナの要素の型を示すために添字表記をサポートするようになりました。

```
from typing import Mapping, Sequence

def notify_by_email(employees: Sequence[Employee],
                    overrides: Mapping[str, str]) -> None: ...
```

ジェネリクスは、typing にある *TypeVar* と呼ばれる新しいファクトリを使ってパラメータ化することができます。

```
from typing import Sequence, TypeVar

T = TypeVar('T')      # Declare type variable

def first(l: Sequence[T]) -> T:   # Generic function
    return l[0]
```

26.1.5 ユーザー定義のジェネリック型

ユーザー定義のクラスを、ジェネリッククラスとして定義できます。

```
from typing import TypeVar, Generic
from logging import Logger

T = TypeVar('T')

class LoggedVar(Generic[T]):
    def __init__(self, value: T, name: str, logger: Logger) -> None:
        self.name = name
        self.logger = logger
        self.value = value

    def set(self, new: T) -> None:
        self.log('Set ' + repr(self.value))
        self.value = new

    def get(self) -> T:
        self.log('Get ' + repr(self.value))
        return self.value

    def log(self, message: str) -> None:
        self.logger.info('%s: %s', self.name, message)
```

`Generic[T]` を基底クラスにすることで、`LoggedVar` クラスが 1 つの型引数 `T` をとる、と定義できます。この定義により、クラスの本体の中でも `T` が型として有効になります。

基底クラス *Generic* には `LoggedVar[t]` が型として有効になるように `__class_getitem__()` メソッドが定義されています:

```
from typing import Iterable

def zero_all_vars(vars: Iterable[LoggedVar[int]]) -> None:
    for var in vars:
        var.set(0)
```

ジェネリック型は任意の数の型変数をとれます、また型変数に制約をつけられます:

```
from typing import TypeVar, Generic
...

T = TypeVar('T')
S = TypeVar('S', int, str)

class StrangePair(Generic[T, S]):
    ...
```

Generic の引数のそれぞれの型変数は別のものでなければなりません。このため次のクラス定義は無効です:

```
from typing import TypeVar, Generic
...

T = TypeVar('T')

class Pair(Generic[T, T]):   # INVALID
    ...
```

Generic を用いて多重継承が可能です:

```
from typing import TypeVar, Generic, Sized

T = TypeVar('T')

class LinkedList(Sized, Generic[T]):
    ...
```

ジェネリッククラスを継承するとき、いくつかの型変数を固定することが出来ます:

```
from typing import TypeVar, Mapping

T = TypeVar('T')

class MyDict(Mapping[str, T]):
    ...
```

この場合では MyDict は仮引数 T を 1 つとります。

型引数を指定せずにジェネリッククラスを使う場合、それぞれの型引数に *Any* を与えられたものとして扱います。次の例では、MyIterable はジェネリックではありませんが Iterable[Any] を暗黙的に継承しています:

```
from typing import Iterable

class MyIterable(Iterable): # Same as Iterable[Any]
```

ユーザ定義のジェネリック型エイリアスもサポートされています。例:

```
from typing import TypeVar, Iterable, Tuple, Union
S = TypeVar('S')
Response = Union[Iterable[S], int]

# Return type here is same as Union[Iterable[str], int]
def response(query: str) -> Response[str]:
    ...

T = TypeVar('T', int, float, complex)
Vec = Iterable[Tuple[T, T]]

def inproduct(v: Vec[T]) -> T: # Same as Iterable[Tuple[T, T]]
    return sum(x*y for x, y in v)
```

バージョン 3.7 で変更: *Generic* にあった独自のメタクラスは無くなりました。

ユーザーが定義したジェネリッククラスはメタクラスの衝突を起こすことなく基底クラスに抽象基底クラスをとれます。ジェネリックメタクラスはサポートされません。パラメータ化を行うジェネリクスの結果はキャッシュされていて、typing モジュールのほとんどの型はハッシュ化でき、等価比較できます。

26.1.6 Any 型

Any は特別な種類の型です。静的型検査器はすべての型を *Any* と互換として扱い、*Any* をすべての型と互換として扱います。

これは、*Any* 型の値では、任意の演算やメソッドの呼び出しが行えることを意味します:

```
from typing import Any

a = None    # type: Any
a = []      # OK
a = 2       # OK

s = ''      # type: str
s = a       # OK

def foo(item: Any) -> int:
    # Typechecks; 'item' could be any type,
    # and that type might have a 'bar' method
    item.bar()
    ...
```

Any 型の値をより詳細な型に代入する時に型検査が行われないことに注意してください。例えば、静的型検査器は `a` を `s` に代入する時、`s` が *str* 型として宣言されていて実行時に *int* の値を受け取るとしても、エラーを報告しません。

さらに、返り値や引数の型のないすべての関数は暗黙的に *Any* を使用します。

```
def legacy_parser(text):
    ...
    return data

# A static type checker will treat the above
# as having the same signature as:
def legacy_parser(text: Any) -> Any:
    ...
    return data
```

この挙動により、動的型付けと静的型付けが混在したコードを書かなければならない時に *Any* を **非常口** として使用することができます。

Any の挙動と *object* の挙動を対比しましょう。*Any* と同様に、すべての型は *object* のサブタイプです。しかしながら、*Any* と異なり、逆は成り立ちません: *object* はすべての他の型のサブタイプでは **ありません**。

これは、ある値の型が *object* のとき、型検査器はこれについてのほとんどすべての操作を拒否し、これをより特殊化された変数に代入する (または返り値として利用する) ことは型エラーになることを意味します。例えば:

```
def hash_a(item: object) -> int:
    # Fails; an object does not have a 'magic' method.
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Typechecks
    item.magic()
    ...

# Typechecks, since ints and strs are subclasses of object
hash_a(42)
hash_a("foo")

# Typechecks, since Any is compatible with all types
hash_b(42)
hash_b("foo")
```

object は、ある値が型安全な方法で任意の型として使えることを示すために使用します。*Any* はある値が動的に型付けられることを示すために使用します。

26.1.7 名前的部分型 vs 構造的部分型

初めは **PEP 484** は Python の静的型システムを **名前的部分型** を使って定義していました。名前的部分型とは、クラス `B` が期待されているところにクラス `A` が許容されるのは `A` が `B` のサブクラスの場合かつその場合に限る、ということです。

前出の必要条件は、*Iterable* などの抽象基底クラスにも当て嵌まります。この型付け手法の問題は、この手法をサポートするためにクラスに明確な型付けを行う必要があることで、これは pythonic ではなく、普段行っている 慣用的な Python コードへの動的型付けとは似ていません。例えば、次のコードは **PEP 484** に従ったものです

```
from typing import Sized, Iterable, Iterator

class Bucket(Sized, Iterable[int]):
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...
```

PEP 544 によって上にあるようなクラス定義で基底クラスを明示しないコードをユーザーが書け、静的型チェッカーで `Bucket` が `Sized` と `Iterable[int]` 両方のサブタイプだと暗黙的に見なせるようになり、この問題が解決しました。これは *structural subtyping (***構造的部分型***)* (あるいは、静的ダックタイピング) として知られています:

```
from typing import Iterator, Iterable

class Bucket:  # Note: no base classes
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...

def collect(items: Iterable[int]) -> int: ...
result = collect(Bucket())  # Passes type check
```

さらに、特別なクラス *Protocol* のサブクラスを作ることで、新しい独自のプロトコルを作って構造的部分型というものを満喫できます。

26.1.8 クラス、関数、およびデコレータ

このモジュールでは以下のクラス、関数、デコレータを定義します:

class typing.**TypeVar**

型変数です。

使い方:

```
T = TypeVar('T')  # Can be anything
A = TypeVar('A', str, bytes)  # Must be str or bytes
```

型変数は主として静的型検査器のために存在します。型変数はジェネリック型やジェネリック関数の定義の引数として役に立ちます。ジェネリック型についての詳細は Generic クラスを参照してください。ジェネリック関数は以下のように動作します:

```
def repeat(x: T, n: int) -> Sequence[T]:
    """Return a list containing n references to x."""
    return [x]*n

def longest(x: A, y: A) -> A:
    """Return the longest of two strings."""
    return x if len(x) >= len(y) else y
```

後者の例のシグネチャは本質的に `(str, str) -> str` と `(bytes, bytes) -> bytes` のオーバーロードです。もし引数が *str* のサブクラスのインスタンスの場合、返り値は普通の *str* であることに注意して下さい。

実行時に、`isinstance(x, T)` は *TypeError* を送出するでしょう。一般的に、*isinstance()* と *issubclass()* は型に対して使用するべきではありません。

型変数は `covariant=True` または `contravariant=True` を渡すことによって共変または反変であることを示せます。詳細は **PEP 484** を参照して下さい。デフォルトの型変数は不変です。あるいは、型変数は `bound=<type>` を使うことで上界を指定することが出来ます。これは、型変数に (明示的または非明示的に) 代入された実際の型が境界の型のサブクラスでなければならないということを意味しま

す、**PEP 484** も参照。

class typing.Generic

ジェネリック型のための抽象基底クラスです。

ジェネリック型は典型的にはこのクラスを 1 つ以上の型変数によってインスタンス化したものを継承することによって宣言されます。例えば、ジェネリックマップ型は次のように定義することが出来ます:

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

このクラスは次のように使用することが出来ます:

```
X = TypeVar('X')
Y = TypeVar('Y')

def lookup_name(mapping: Mapping[X, Y], key: X, default: Y) -> Y:
    try:
        return mapping[key]
    except KeyError:
        return default
```

class typing.Protocol(*Generic*)

プロトコルクラスの基底クラス。プロトコルクラスは次のように定義されます:

```
class Proto(Protocol):
    def meth(self) -> int:
        ...
```

このようなクラスは主に構造的部分型 (静的ダックタイピング) を認識する静的型チェッカーが使います。例えば:

```
class C:
    def meth(self) -> int:
        return 0

def func(x: Proto) -> int:
    return x.meth()

func(C())  # Passes static type check
```

詳細については pep:*544* を参照してください。*runtime_checkable()* (後で説明します) でデコレートされたプロトコルクラスは、与えられたメソッドがあることだけを確認し、その型シグネチャは全く見ない安直な動作をする実行時プロトコルとして振る舞います。

プロトコルクラスはジェネリックにもできます。例えば:

```
class GenProto(Protocol[T]):
    def meth(self) -> T:
        ...
```

バージョン 3.8 で追加.

class typing.Type(*Generic[CT_co]*)

C と注釈が付けされた変数は C 型の値を受理します。一方で Type[C] と注釈が付けられた変数は、そのクラス自身を受理します -- 具体的には、それは C の **クラスオブジェクト** を受理します。例:

```
a = 3         # Has type 'int'
b = int       # Has type 'Type[int]'
c = type(a)   # Also has type 'Type[int]'
```

Type[C] は共変であることに注意してください:

```
class User: ...
class BasicUser(User): ...
class ProUser(User): ...
class TeamUser(User): ...

# Accepts User, BasicUser, ProUser, TeamUser, ...
def make_new_user(user_class: Type[User]) -> User:
    # ...
    return user_class()
```

Type[C] が共変だということは、C の全てのサブクラスは、C と同じシグネチャのコンストラクタとクラスメソッドを実装すべきだということになります。型チェッカーはこの規則への違反に印を付けるべきですが、サブクラスでのコンストラクタ呼び出しで、指定された基底クラスのコンストラクタ呼び出しに適合するものは許可すべきです。この特別な場合を型チェッカーがどう扱うべきかについては、**PEP 484** の将来のバージョンで変更されるかもしれません。

Type で許されているパラメータは、クラス、*Any* 、**型変数** あるいは、それらの直和型だけです。例えば次のようになります:

```
def new_non_team_user(user_class: Type[Union[BasicUser, ProUser]]): ...
```

Type[Any] は Type と等価で、同様に Type は type と等価です。type は Python のメタクラス階層のルートです。

バージョン 3.5.2 で追加.

class typing.Iterable(*Generic[T_co]*)

collections.abc.Iterable のジェネリック版です。

class typing.Iterator(*Iterable[T_co]*)

collections.abc.Iterator のジェネリック版です。

class typing.Reversible(*Iterable[T_co]*)

collections.abc.Reversible のジェネリック版です。

class typing.SupportsInt
: 抽象メソッド `__int__` を備えた ABC です。

class typing.SupportsFloat
: 抽象メソッド `__float__` を備えた ABC です。

class typing.SupportsComplex
: 抽象メソッド `__complex__` を備えた ABC です。

class typing.SupportsBytes
: 抽象メソッド `__bytes__` を備えた ABC です。

class typing.SupportsIndex
: 抽象メソッド `__index__` を備えた ABC です。

  バージョン 3.8 で追加.

class typing.SupportsAbs
: 返り値の型と共変な抽象メソッド `__abs__` を備えた ABC です。

class typing.SupportsRound
: 返り値の型と共変な抽象メソッド `__round__` を備えた ABC です。

class typing.Container(*Generic[T_co]*)
: *collections.abc.Container* のジェネリック版です。

class typing.Hashable
: *collections.abc.Hashable* へのエイリアス

class typing.Sized
: *collections.abc.Sized* へのエイリアス

class typing.Collection(*Sized, Iterable[T_co], Container[T_co]*)
: *collections.abc.Collection* のジェネリック版です。

  バージョン 3.6.0 で追加.

class typing.AbstractSet(*Sized, Collection[T_co]*)
: *collections.abc.Set* のジェネリック版です。

class typing.MutableSet(*AbstractSet[T]*)
: *collections.abc.MutableSet* のジェネリック版です。

class typing.Mapping(*Sized, Collection[KT], Generic[VT_co]*)
: *collections.abc.Mapping* のジェネリック版です。この型は次のように使えます:

```
def get_position_in_index(word_list: Mapping[str, int], word: str) -> int:
    return word_list[word]
```

class typing.MutableMapping(*Mapping[KT, VT]*)
: *collections.abc.MutableMapping* のジェネリック版です。

class typing.**Sequence**(*Reversible[T_co], Collection[T_co]*)

collections.abc.Sequence のジェネリック版です。

class typing.**MutableSequence**(*Sequence[T]*)

collections.abc.MutableSequence のジェネリック版です。

class typing.**ByteString**(*Sequence[int]*)

collections.abc.ByteString のジェネリック版です。

この型は *bytes* と *bytearray* 、バイト列の *memoryview* を表します。

この型の省略形として、*bytes* を上に挙げた任意の型の引数にアノテーションをつけることに使えます。

class typing.**Deque**(*deque, MutableSequence[T]*)

collections.deque のジェネリック版です。

バージョン 3.5.4 で追加.

バージョン 3.6.1 で追加.

class typing.**List**(*list, MutableSequence[T]*)

list のジェネリック版です。返り値の型のアノテーションをつけるのに便利です。引数にアノテーションをつけるためには、*Sequence* や *Iterable* のような抽象コレクション型を使うことが好ましいです。

この型は次のように使えます:

```
T = TypeVar('T', int, float)

def vec2(x: T, y: T) -> List[T]:
    return [x, y]

def keep_positives(vector: Sequence[T]) -> List[T]:
    return [item for item in vector if item > 0]
```

class typing.**Set**(*set, MutableSet[T]*)

builtins.set のジェネリック版です。返り値の型のアノテーションをつけるのに便利です。引数にアノテーションをつけるためには、*AbstractSet* のような抽象コレクション型を使うことが好ましいです。

class typing.**FrozenSet**(*frozenset, AbstractSet[T_co]*)

builtins.frozenset のジェネリック版です。

class typing.**MappingView**(*Sized, Iterable[T_co]*)

collections.abc.MappingView のジェネリック版です。

class typing.**KeysView**(*MappingView[KT_co], AbstractSet[KT_co]*)

collections.abc.KeysView のジェネリック版です。

class typing.**ItemsView**(*MappingView, Generic[KT_co, VT_co]*)

collections.abc.ItemsView のジェネリック版です。

class typing.ValuesView(*MappingView[VT_co]*)
collections.abc.ValuesView のジェネリック版です。

class typing.Awaitable(*Generic[T_co]*)
collections.abc.Awaitable のジェネリック版です。

バージョン 3.5.2 で追加.

class typing.Coroutine(*Awaitable[V_co], Generic[T_co, T_contra, V_co]*)
collections.abc.Coroutine のジェネリック版です。変性と型変数の順序は Generator のものと対応しています。例えば次のようになります:

```
from typing import List, Coroutine
c = None # type: Coroutine[List[str], str, int]
...
x = c.send('hi') # type: List[str]
async def bar() -> None:
    x = await c # type: int
```

バージョン 3.5.3 で追加.

class typing.AsyncIterable(*Generic[T_co]*)
collections.abc.AsyncIterable のジェネリック版です。

バージョン 3.5.2 で追加.

class typing.AsyncIterator(*AsyncIterable[T_co]*)
collections.abc.AsyncIterator のジェネリック版です。

バージョン 3.5.2 で追加.

class typing.ContextManager(*Generic[T_co]*)
contextlib.AbstractContextManager のジェネリック版です。

バージョン 3.5.4 で追加.

バージョン 3.6.0 で追加.

class typing.AsyncContextManager(*Generic[T_co]*)
contextlib.AbstractAsyncContextManager のジェネリック版です。

バージョン 3.5.4 で追加.

バージョン 3.6.2 で追加.

class typing.Dict(*dict, MutableMapping[KT, VT]*)
dict のジェネリック版です。返り値の型のアノテーションをつけることに便利です。引数にアノテーションをつけるためには、Mapping のような抽象コレクション型を使うことが好ましいです。

この型は次のように使えます:

```
def count_words(text: str) -> Dict[str, int]:
    ...
```

class typing.DefaultDict(*collections.defaultdict, MutableMapping[KT, VT]*)

collections.defaultdict のジェネリック版です。

バージョン 3.5.2 で追加.

class typing.OrderedDict(*collections.OrderedDict, MutableMapping[KT, VT]*)

collections.OrderedDict のジェネリック版です。

バージョン 3.7.2 で追加.

class typing.Counter(*collections.Counter, Dict[T, int]*)

collections.Counter のジェネリック版です。

バージョン 3.5.4 で追加.

バージョン 3.6.1 で追加.

class typing.ChainMap(*collections.ChainMap, MutableMapping[KT, VT]*)

collections.ChainMap のジェネリック版です。

バージョン 3.5.4 で追加.

バージョン 3.6.1 で追加.

class typing.Generator(*Iterator[T_co], Generic[T_co, T_contra, V_co]*)

ジェネレータはジェネリック型 `Generator[YieldType, SendType, ReturnType]` によってアノテーションを付けられます。例えば:

```
def echo_round() -> Generator[int, float, str]:
    sent = yield 0
    while sent >= 0:
        sent = yield round(sent)
    return 'Done'
```

typing モジュールの多くの他のジェネリクスと違い *Generator* の `SendType` は共変や不変ではなく、反変として扱われることに注意してください。

もしジェネレータが値を返すだけの場合は、`SendType` と `ReturnType` に `None` を設定してください:

```
def infinite_stream(start: int) -> Generator[int, None, None]:
    while True:
        yield start
        start += 1
```

代わりに、ジェネレータを `Iterable[YieldType]` や `Iterator[YieldType]` という返り値の型でアノテーションをつけることもできます:

```
def infinite_stream(start: int) -> Iterator[int]:
    while True:
        yield start
        start += 1
```

class typing.AsyncGenerator(*AsyncIterator[T_co], Generic[T_co, T_contra]*)

非同期ジェネレータはジェネリック型 AsyncGenerator[YieldType, SendType] によってアノテーションを付けられます。例えば:

```
async def echo_round() -> AsyncGenerator[int, float]:
    sent = yield 0
    while sent >= 0.0:
        rounded = await round(sent)
        sent = yield rounded
```

通常のジェネレータと違って非同期ジェネレータは値を返せないので、ReturnType 型引数はありません。*Generator* と同様に、SendType は反変的に振る舞います。

ジェネレータが値を yield するだけなら、SendType を None にします:

```
async def infinite_stream(start: int) -> AsyncGenerator[int, None]:
    while True:
        yield start
        start = await increment(start)
```

あるいは、ジェネレータが AsyncIterable[YieldType] と AsyncIterator[YieldType] のいずれかの戻り値型を持つとアノテートします:

```
async def infinite_stream(start: int) -> AsyncIterator[int]:
    while True:
        yield start
        start = await increment(start)
```

バージョン 3.6.1 で追加.

class typing.Text

Text は str のエイリアスです。これは Python 2 のコードの前方互換性を提供するために設けられています: Python 2 では Text は unicode のエイリアスです。

Text は Python 2 と Python 3 の両方と互換性のある方法で値が unicode 文字列を含んでいなければならない場合に使用してください。

```
def add_unicode_checkmark(text: Text) -> Text:
    return text + u' \u2713'
```

バージョン 3.5.2 で追加.

class typing.IO

class typing.TextIO

class typing.BinaryIO

ジェネリック型 IO[AnyStr] とそのサブクラスの TextIO(IO[str]) および BinaryIO(IO[bytes]) は、*open()* 関数が返すような I/O ストリームの型を表します。

class typing.Pattern

class typing.Match

これらの型エイリアスは *re.compile()* と *re.match()* の返り値の型に対応します。これらの型 (と対応する関数) は AnyStr についてジェネリックで、Pattern[str] 、Pattern[bytes] 、Match[str] 、Match[bytes] と書くことで具体型にできます。

class typing.NamedTuple

collections.namedtuple() の型付き版です。

使い方:

```
class Employee(NamedTuple):
    name: str
    id: int
```

これは次と等価です:

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

フィールドにデフォルト値を与えるにはクラス本体で代入してください:

```
class Employee(NamedTuple):
    name: str
    id: int = 3

employee = Employee('Guido')
assert employee.id == 3
```

デフォルト値のあるフィールドはデフォルト値のないフィールドの後でなければなりません。

最終的に出来上がるクラスには、フィールド名をフィールド型へ対応付ける辞書を提供する __annotations__ 属性が追加されています。(フィールド名は _fields 属性に、デフォルト値は _field_defaults 属性に格納されていて、両方とも名前付きタプル API の一部分です。)

NamedTuple のサブクラスは docstring やメソッドも持てます:

```
class Employee(NamedTuple):
    """Represents an employee."""
    name: str
    id: int = 3

    def __repr__(self) -> str:
        return f'<Employee {self.name}, id={self.id}>'
```

後方互換な使用法:

```
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

バージョン 3.6 で変更: **PEP 526** 変数アノテーションのシンタックスが追加されました。

バージョン 3.6.1 で変更: デフォルト値、メソッド、ドキュメンテーション文字列への対応が追加されました。

Deprecated since version 3.8, will be removed in version 3.9: `_field_types` 属性は非推奨となりました。代わりに同じ情報を持つより標準的な `__annotations__` 属性を使ってください。

バージョン 3.8 で変更: `_field_types` 属性および `__annotations__` 属性は `OrderedDict` インスタンスではなく普通の辞書になりまいた。

class typing.**TypedDict**(*dict*)

シンプルな型付き名前空間です。実行時には素の *dict* と同等のものになります。

`TypedDict` は、その全てのインスタンスにおいてキーの集合が固定されていて、各キーに対応する値が全てのインスタンスで同じ型を持つことが期待される辞書型を作成します。この期待は実行時にはチェックされず、型チェッカーでのみ強制されます。使用方法は次の通りです:

```
class Point2D(TypedDict):
    x: int
    y: int
    label: str

a: Point2D = {'x': 1, 'y': 2, 'label': 'good'}  # OK
b: Point2D = {'z': 3, 'label': 'bad'}           # Fails type check

assert Point2D(x=1, y=2, label='first') == dict(x=1, y=2, label='first')
```

内観のための型情報には `Point2D.__annotations__` や `Point2D.__total__` を通してアクセスできます。**PEP 526** をサポートしていない古いバージョンの Python でこの機能を使えるようにするために、`TypedDict` はこれと同等の 2 つの文法形式を追加でサポートしています:

```
Point2D = TypedDict('Point2D', x=int, y=int, label=str)
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label': str})
```

By default, all keys must be present in a TypedDict. It is possible to override this by specifying totality. Usage:

```
class point2D(TypedDict, total=False):
    x: int
    y: int
```

This means that a point2D TypedDict can have any of the keys omitted. A type checker is only expected to support a literal False or True as the value of the total argument. True is the default, and makes all items defined in the class body be required.

他の例や、`TypedDict` を扱う詳細な規則については **PEP 589** を参照してください。

バージョン 3.8 で追加.

class typing.**ForwardRef**

文字列による前方参照の内部的な型付け表現に使われるクラスです。例えば、`List["SomeClass"]` は暗黙的に `List[ForwardRef("SomeClass")]` に変換されます。このクラスはユーザーがインスタンス化するべきではなく、イントロスペクションツールに使われるものです。

バージョン 3.7.4 で追加.

typing.**NewType**(*name*, *tp*)

異なる型であることを型チェッカーに教えるためのヘルパー関数です。*NewType* を参照してください。実行時には、その引数を返す関数を返します。使い方は次のようになります:

```
UserId = NewType('UserId', int)
first_user = UserId(1)
```

バージョン 3.5.2 で追加.

typing.**cast**(*typ*, *val*)

値をある型にキャストします。

この関数は値を変更せずに返します。型検査器に対して、返り値が指定された型を持っていることを通知しますが、実行時には意図的に何も検査しません。(その理由は、処理をできる限り速くしたかったためです。)

typing.**get_type_hints**(*obj*[, *globals*[, *locals*]])

関数、メソッド、モジュールまたはクラスのオブジェクトの型ヒントを含む辞書を返します。

この辞書はたいてい `obj.__annotations__` と同じものです。それに加えて、文字列リテラルにエンコードされた順方向参照は `globals` 名前空間および `locals` 名前空間で評価されます。必要であれば、`None` と等価なデフォルト値が設定されている場合に、関数とメソッドのアノテーションに `Optional[t]` が追加されます。クラス `C` については、`C.__mro__` の逆順に沿って全ての `__annotations__` を合併して構築された辞書を返します。

typing.**get_origin**(*tp*)

typing.**get_args**(*tp*)

ジェネリック型や特殊な型付け形式についての基本的な内観を提供します。

For a typing object of the form `X[Y, Z, ...]` these functions return `X` and `(Y, Z, ...)`. If `X` is a generic alias for a builtin or *collections* class, it gets normalized to the original class. For unsupported objects return `None` and `()` correspondingly. Examples:

```
assert get_origin(Dict[str, int]) is dict
assert get_args(Dict[int, str]) == (int, str)

assert get_origin(Union[int, str]) is Union
assert get_args(Union[int, str]) == (int, str)
```

バージョン 3.8 で追加.

@typing.**overload**

`@overload` デコレータを使うと、引数の型の複数の組み合わせをサポートする関数やメソッドを書けるようになります。`@overload` 付きの定義を並べた後ろに、(同じ関数やメソッドの) `@overload` 無しの定義が来なければなりません。`@overload` 付きの定義は型チェッカーのためでしかありません。というのも、`@overload` 付きの定義は `@overload` 無しの定義で上書きされるからです。後者は実行時に使われますが、型チェッカーからは無視されるべきなのです。実行時には、`@overload` 付きの関数を直接呼び出すと *NotImplementedError* を送出します。次のコードはオーバーロードを使うことで直

和型や型変数を使うよりもより正確な型が表現できる例です:

```
@overload
def process(response: None) -> None:
    ...
@overload
def process(response: int) -> Tuple[int, str]:
    ...
@overload
def process(response: bytes) -> str:
    ...
def process(response):
    <actual implementation>
```

詳細と他の型付け意味論との比較は **PEP 484** を参照してください。

@typing.final

A decorator to indicate to type checkers that the decorated method cannot be overridden, and the decorated class cannot be subclassed. For example:

```
class Base:
    @final
    def done(self) -> None:
        ...
class Sub(Base):
    def done(self) -> None:  # Error reported by type checker
          ...

@final
class Leaf:
    ...
class Other(Leaf):  # Error reported by type checker
    ...
```

この機能は実行時には検査されません。詳細については **PEP 591** を参照してください。

バージョン 3.8 で追加.

@typing.no_type_check

アノテーションが型ヒントでないことを示すデコレータです。

これはクラス *decorator* または関数 *decorator* として動作します。クラス *decorator* として動作する場合は、そのクラス内に定義されたすべてのメソッドに対して再帰的に適用されます。(ただしスーパークラスやサブクラス内に定義されたメソッドには適用されません。)

これは関数を適切に変更します。

@typing.no_type_check_decorator

別のデコレータに *no_type_check()* の効果を与えるデコレータです。

これは何かの関数をラップするデコレータを *no_type_check()* でラップします。

@typing.type_check_only
実行時に使えなくなるクラスや関数に印を付けるデコレータです。

このデコレータ自身は実行時には使えません。このデコレータは主に、実装がプライベートクラスのインスタンスを返す場合に、型スタブファイルに定義されているクラスに対して印を付けるためのものです:

```
@type_check_only
class Response:  # private or not available at runtime
    code: int
    def get_header(self, name: str) -> str: ...

def fetch_response() -> Response: ...
```

プライベートクラスのインスタンスを返すのは推奨されません。そのようなクラスは公開クラスにするのが望ましいです。

@typing.runtime_checkable
Mark a protocol class as a runtime protocol.

Such a protocol can be used with *isinstance()* and *issubclass()*. This raises *TypeError* when applied to a non-protocol class. This allows a simple-minded structural check, very similar to "one trick ponies" in *collections.abc* such as *Iterable*. For example:

```
@runtime_checkable
class Closable(Protocol):
    def close(self): ...

assert isinstance(open('/some/file'), Closable)
```

Warning: this will check only the presence of the required methods, not their type signatures!

バージョン 3.8 で追加.

typing.Any
制約のない型であることを示す特別な型です。

- 任意の型は *Any* と互換です。
- *Any* は任意の型と互換です。

typing.NoReturn
関数が返り値を持たないことを示す特別な型です。例えば次のように使います:

```
from typing import NoReturn

def stop() -> NoReturn:
    raise RuntimeError('no way')
```

バージョン 3.5.4 で追加.

バージョン 3.6.2 で追加.

typing.Union
: ユニオン型; Union[X, Y] は X または Y を表します。

  ユニオン型を定義します、例えば Union[int, str] のように使います。詳細:

  - 引数は型でなければならず、少なくとも一つ必要です。
  - ユニオン型のユニオン型は平滑化されます。例えば:

    ```
    Union[Union[int, str], float] == Union[int, str, float]
    ```

  - 引数が一つのユニオン型は消えます。例えば:

    ```
    Union[int] == int  # The constructor actually returns int
    ```

  - 冗長な実引数は飛ばされます。例えば:

    ```
    Union[int, str, int] == Union[int, str]
    ```

  - ユニオン型を比較するとき引数の順序は無視されます。例えば:

    ```
    Union[int, str] == Union[str, int]
    ```

  - ユニオン型のサブクラスを作成したり、インスタンスを作成することは出来ません。
  - Union[X][Y] と書くことは出来ません。
  - Optional[X] を Union[X, None] の略記として利用することが出来ます。

  バージョン 3.7 で変更: 明示的に書かれているサブクラスを、実行時に直和型から取り除かなくなりました。

typing.Optional
: オプショナル型。

  Optional[X] は Union[X, None] と同値です。

  これがデフォルト値を持つオプション引数とは同じ概念ではないということに注意してください。デフォルト値を持つオプション引数はオプション引数であるために、型アノテーションに Optional 修飾子は必要ありません。例えば次のようになります:

  ```
  def foo(arg: int = 0) -> None:
      ...
  ```

  それとは逆に、None という値が許されていることが明示されている場合は、引数がオプションであろうとなかろうと、Optional を使うのが好ましいです。例えば次のようになります:

  ```
  def foo(arg: Optional[int] = None) -> None:
      ...
  ```

typing.Tuple

タプル型; Tuple[X, Y] は、最初の要素の型が X で、2 つ目の要素の型が Y であるような、2 つの要素を持つタプルの型です。空のタプルの型は Tuple[()] と書けます。

例: Tuple[T1, T2] は型変数 T1 と T2 に対応する 2 つの要素を持つタプルです。Tuple[int, float, str] は int と float、string のタプルです。

同じ型の任意の長さのタプルを指定するには ellipsis リテラルを用います。例: Tuple[int, ...]。ただの *Tuple* は Tuple[Any, ...] と等価で、さらに *tuple* と等価です。.

typing.Callable

呼び出し可能型; Callable[[int], str] は (int) -> str の関数です。

添字表記は常に 2 つの値とともに使われなければなりません: 実引数のリストと返り値の型です。実引数のリストは型のリストか ellipsis でなければなりません; 返り値の型は単一の型でなければなりません。

オプショナル引数やキーワード引数を表すための文法はありません; そのような関数型はコールバックの型として滅多に使われません。Callable[..., ReturnType] (リテラルの Ellipsis) は任意の個数の引数をとり ReturnType を返す型ヒントを与えるために使えます。普通の *Callable* は Callable[..., Any] と同等で、*collections.abc.Callable* でも同様です。

typing.Literal

型チェッカーに、変数や関数引数と対応する与えられたリテラル (あるいはいくつかあるリテラルのうちの 1 つ) が同等な値を持つことを表すのに使える型です。

```
def validate_simple(data: Any) -> Literal[True]:  # always returns True
    ...

MODE = Literal['r', 'rb', 'w', 'wb']
def open_helper(file: str, mode: MODE) -> str:
    ...

open_helper('/some/path', 'r')  # Passes type check
open_helper('/other/path', 'typo')  # Error in type checker
```

Literal[...] はサブクラスにはできません。実行時に、任意の値が Literal[...] の型引数として使えますが、型チェッカーが制約を課すことがあります。リテラル型についてより詳しいことは **PEP 586** を参照してください。

バージョン 3.8 で追加.

typing.ClassVar

クラス変数であることを示す特別な型構築子です。

PEP 526 で導入された通り、ClassVar でラップされた変数アノテーションによって、ある属性はクラス変数として使うつもりであり、そのクラスのインスタンスから設定すべきではないということを示せます。使い方は次のようになります:

```
class Starship:
    stats: ClassVar[Dict[str, int]] = {} # class variable
    damage: int = 10                     # instance variable
```

ClassVar は型のみを受け入れ、それ以外は受け付けられません。

ClassVar はクラスそのものではなく、*isinstance()* や *issubclass()* で使うべきではありません。*ClassVar* は Python の実行時の挙動を変えませんが、サードパーティの型検査器で使えます。例えば、型チェッカーは次のコードをエラーとするかもしれません:

```
enterprise_d = Starship(3000)
enterprise_d.stats = {} # Error, setting class variable on instance
Starship.stats = {}     # This is OK
```

バージョン 3.5.3 で追加.

typing.**Final**

特別な型付けの構成要素で、名前の割り当て直しやサブクラスでのオーバーライドができないことを型チェッカーに示すためのものです。例えば:

```
MAX_SIZE: Final = 9000
MAX_SIZE += 1  # Error reported by type checker

class Connection:
    TIMEOUT: Final[int] = 10

class FastConnector(Connection):
    TIMEOUT = 1  # Error reported by type checker
```

この機能は実行時には検査されません。詳細については **PEP 591** を参照してください。

バージョン 3.8 で追加.

typing.**AnyStr**

AnyStr は `AnyStr = TypeVar('AnyStr', str, bytes)` として定義される型変数です。

他の種類の文字列を混ぜることなく、任意の種類の文字列を許す関数によって使われることを意図しています。

```
def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b

concat(u"foo", u"bar")  # Ok, output has type 'unicode'
concat(b"foo", b"bar")  # Ok, output has type 'bytes'
concat(u"foo", b"bar")  # Error, cannot mix unicode and bytes
```

typing.**TYPE_CHECKING**

サードパーティーの静的型検査器が `True` と仮定する特別な定数です。実行時には `False` になります。使用例:

```
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: 'expensive_mod.SomeType') -> None:
    local_var: expensive_mod.AnotherType = other_fun()
```

1 つ目の型アノテーションは引用符で囲って " 前方参照 (forward reference)" にし、インタプリタのランタイムから expensive_mod への参照を隠さなければなりません。ローカル変数への型アノテーションは評価されないので、2 つ目のアノテーションは引用符で囲う必要はありません。

バージョン 3.5.2 で追加.

26.2 pydoc --- ドキュメント生成とオンラインヘルプシステム

ソースコード: Lib/pydoc.py

---

pydoc モジュールは、Python モジュールから自動的にドキュメントを生成します。生成されたドキュメントをテキスト形式でコンソールに表示したり、Web ブラウザにサーバとして提供したり、HTML ファイルとして保存したりできます。

モジュール、クラス、関数、メソッドについての表示されるドキュメンテーションは、オブジェクトの docstring (つまり __doc__ 属性) に基き、またそれのドキュメント可能なメンバが再帰的に続きます。docstring がない場合、*pydoc* は、クラス、関数、メソッドについてはその定義の直前の、モジュールについてはファイル先頭の、ソースファイルのコメント行のブロックから記述を取得しようと試みます (*inspect.getcomments()* を参照してください)。

組み込み関数の *help()* を使うことで、対話型インタプリタからオンラインヘルプを起動することができます。コンソール用のテキスト形式のドキュメントをつくるのにオンラインヘルプでは *pydoc* を使っています。**pydoc** を Python インタプリタからはなく、オペレーティングシステムのコマンドプロンプトから起動した場合でも、同じテキスト形式のドキュメントを見ることができます。例えば、以下の実行を

```
pydoc sys
```

シェルから行うと *sys* モジュールのドキュメントを、Unix の **man** コマンドのような形式で表示させることができます。**pydoc** の引数として与えることができるのは、関数名・モジュール名・パッケージ名、また、モジュールやパッケージ内のモジュールに含まれるクラス・メソッド・関数へのドット形式での参照です。**pydoc** への引数がパスと解釈されるような場合で (オペレーティングシステムのパス区切り記号を含む場合です。例えば Unix ならばスラッシュ含む場合になります)、さらに、そのパスが Python のソースファイルを指しているなら、そのファイルに対するドキュメントが生成されます。

注釈: オブジェクトとそのドキュメントを探すために、*pydoc* はドキュメント対象のモジュールを import します。そのため、モジュールレベルのコードはそのときに実行されます。`if __name__ == '__main__':` ガードを使って、ファイルがスクリプトとして実行したときのみコードを実行し、import されたときには実

行されないようにして下さい。

コンソールへの出力時には、**pydoc** は読みやすさのために出力をページ化しようと試みます。環境変数 PAGER がセットされていれば **pydoc** はその値で設定されたページ化プログラムを使います。

引数の前に **-w** フラグを指定すると、コンソールにテキストを表示させるかわりにカレントディレクトリに HTML ドキュメントを生成します。

引数の前に **-k** フラグを指定すると、引数をキーワードとして利用可能な全てのモジュールの概要を検索します。検索のやりかたは、Unix の **man** コマンドと同様です。モジュールの概要というのは、モジュールのドキュメントの一行目のことです。

また、**pydoc** を使うことでローカルマシンに Web browser から閲覧可能なドキュメントを提供する HTTP サーバーを起動することもできます。**pydoc -p 1234** とすると、HTTP サーバーをポート 1234 に起動します。これで、お好きな Web ブラウザを使って `http://localhost:1234/` からドキュメントを見ることができます。ポート番号に 0 を指定すると、任意の空きポートが選択されます。

pydoc -n <hostname> は、与えられたホスト名で listen するサーバーを起動します。デフォルトではホスト名は 'localhost' ですが、他のマシンからサーバーへ疎通できるようにしたい場合は、サーバーが応答するホスト名を変更したいと思うでしょう。開発作業中に、コンテナ内で pydoc を走らせたい場合は、これは特に便利です。

pydoc -b では、サーバとして起動するとともにブラウザも起動し、モジュールインデクスページを開きます。提供されるページには、個別のヘルプページに飛ぶための *Get* ボタン、全モジュールから概要行に基くキーワード検索するための *Search* ボタン、と、*Module index*, *Topics* , *Keywords* へのそれぞれリンクがついたナビゲーションバーがページの一番上に付きます。

pydoc でドキュメントを生成する場合、その時点での環境とパス情報に基づいてモジュールがどこにあるのか決定されます。そのため、**pydoc spam** を実行した場合につくられるドキュメントは、Python インタプリタを起動して `import spam` と入力したときに読み込まれるモジュールに対するドキュメントになります。

コアモジュールのドキュメントは https://docs.python.org/X.Y/library/ にあると仮定されています。X, Y はそれぞれ Python インタプリタのメジャー、マイナーバージョン番号です。環境変数 PYTHONDOCS を設定することでこれをオーバライドすることが出来、ライブラリリファレンスページを含む別の URL やローカルディレクトリを指定出来ます。

バージョン 3.2 で変更: **-b** オプションが追加されました。

バージョン 3.3 で変更: **-g** コマンドラインオプションが削除されました。

バージョン 3.4 で変更: *pydoc* は、callable からシグニチャの情報を得るのに *inspect.getfullargspec()* ではなく *inspect.signature()* を使うようになりました。

バージョン 3.7 で変更: **-n** オプションが追加されました。

26.3 doctest --- 対話的な実行例をテストする

ソースコード: Lib/doctest.py

---

doctest モジュールは、対話的 Python セッションのように見えるテキストを探し出し、セッションの内容を実行して、そこに書かれている通りに振舞うかを調べます。*doctest* は以下のような用途によく使われています:

- モジュールの docstring (ドキュメンテーション文字列) 中にある対話実行例のすべてが書かれている通りに動作するか検証することで、docstring の内容が最新かどうかチェックする。
- テストファイルやテストオブジェクト中の対話実行例が期待通りに動作するかを検証することで、回帰テストを実現します。
- 入出力例を豊富に使ったパッケージのチュートリアルドキュメントが書けます。入出力例と解説文のどちらに注目するかによって、ドキュメントは「読めるテスト」にも「実行できるドキュメント」にもなります。

以下に完全かつ短い実行例を示します:

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000

    It must also not be ridiculously large:
```

(次のページに続く)

(前のページからの続き)

```
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

example.py をコマンドラインから直接実行すると、*doctest* はその魔法を働かせます:

```
$ python example.py
$
```

出力は何もありません！ しかしこれが正常で、すべての実行例が正しく動作することを意味しています。スクリプトに -v を与えると、*doctest* は何を行おうとしているのかを記録した詳細なログを出力し、最後にまとめを出力します:

```
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

といった具合で、最後には:

```
Trying:
    factorial(1e100)
```

(次のページに続く)

(前のページからの続き)

```
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
2 items passed all tests:
   1 tests in __main__
   8 tests in __main__.factorial
9 tests in 2 items.
9 passed and 0 failed.
Test passed.
$
```

doctest の生産的な利用を始めるために知る必要があるのはこれだけです！ さあやってみましょう。詳細な事柄は後続の各節ですべて説明しています。doctest の例は、標準の Python テストスイートやライブラリ中に沢山あります。標準のテストファイル `Lib/test/test_doctest.py` には、特に役に立つ例があります。

26.3.1 簡単な利用法: docstring 中の実行例をチェックする

doctest を試す簡単な方法 (とはいえ、いつもそうする必要はないのですが) は、各モジュール `M` の最後を、以下のようにして締めくくることです:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

こうすると、*doctest* は `M` 中の docstring を検査します。

モジュールをスクリプトとして実行すると、docstring 中の実行例が実行され、検証されます:

```
python M.py
```

docstring に書かれた実行例の実行が失敗しない限り、何も表示されません。失敗すると、失敗した実行例と、その原因が (場合によっては複数) 標準出力に印字され、最後に `***Test Failed*** N failures.` という行を出力します。ここで、*N* は失敗した実行例の数です。

一方、`-v` スイッチをつけて走らせると:

```
python M.py -v
```

実行を試みたすべての実行例について詳細に報告し、最後に各種まとめを行った内容が標準出力に印字されます。

`verbose=True` を *testmod()* に渡せば、詳細報告 (verbose) モードを強制できます。また、`verbose=False` にすれば禁止できます。どちらの場合にも、*testmod()* は `sys.argv` 上のスイッチを調べません。(したがって、`-v` をつけても効果はありません)。

testmod() を実行するコマンドラインショートカットもあります。Python インタプリタに doctest モジュールを標準ライブラリから直接実行して、テストするモジュール名をコマンドライン引数に与えます:

```
python -m doctest -v example.py
```

こうすると example.py を単体モジュールとしてインポートして、それに対して *testmod()* を実行します。このファイルがパッケージの一部で他のサブモジュールをそのパッケージからインポートしている場合はうまく動かないことに注意してください。

testmod() の詳しい情報は **基本** *API* 節を参照してください。

26.3.2 簡単な利用法: テキストファイル中の実行例をチェックする

doctest のもう一つの簡単な用途は、テキストファイル中にある対話実行例に対するテストです。これには *testfile()* 関数を使います:

```
import doctest
doctest.testfile("example.txt")
```

この短いスクリプトは、example.txt というファイルの中に入っている対話モードの Python 操作例すべてを実行して、その内容を検証します。ファイルの内容は一つの巨大な docstring であるかのように扱われます; ファイルが Python プログラムである必要はありません！ 例えば、example.txt には以下のような内容が入っているとします:

```
The ``example`` module
======================

Using ``factorial``
-------------------

This is an example text file in reStructuredText format.  First import
``factorial`` from the ``example`` module:

    >>> from example import factorial

Now use it:

    >>> factorial(6)
    120
```

doctest.testfile("example.txt") を実行すると、このドキュメント内のエラーを見つけ出します:

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

testmod() と同じく、*testfile()* は実行例が失敗しない限り何も表示しません。実行例が失敗すると、失敗した実行例とその原因が (場合によっては複数) *testmod()* と同じ書式で標準出力に書き出されます。

デフォルトでは、*testfile()* は自分自身を呼び出したモジュールのあるディレクトリを探します。その他の場所にあるファイルを見に行くように *testfile()* に指示するためのオプション引数についての説明は **基本** *API* 節を参照してください。

testmod() と同様に *testfile()* の冗長性 (verbosity) はコマンドラインスイッチ `-v` またはオプションのキーワード引数 *verbose* によって指定できます。

testfile() を実行するコマンドラインショートカットもあります。Python インタプリタに doctest モジュールを標準ライブラリから直接実行して、テストするモジュール名をコマンドライン引数に与えます:

```
python -m doctest -v example.txt
```

ファイル名が `.py` で終っていないので、*doctest* は *testmod()* ではなく *testfile()* を使って実行するのだと判断します。

testfile() の詳細は **基本** *API* 節を参照してください。

26.3.3 doctest のからくり

この節では、doctest のからくり: どの docstring を見に行くのか、どのように対話実行例を見つけ出すのか、どんな実行コンテキストを使うのか、例外をどう扱うか、上記の振る舞いを制御するためにどのようなオプションフラグを使うか、について詳しく吟味します。こうした情報は、doctest に対応した実行例を書くために必要な知識です; 書いた実行例に対して実際に doctest を実行する上で必要な情報については後続の節を参照してください。

どの docstring が検証されるのか?

モジュールの docstring と、すべての関数、クラスおよびメソッドの docstring が検索されます。モジュールに import されたオブジェクトは検索されません。

加えて、`M.__test__` が存在し、" 真の値を持つ" 場合、この値は辞書でなければならず、辞書の各エントリは (文字列の) 名前を関数オブジェクト、クラスオブジェクト、または文字列へとマップします。`M.__test__` から得られた関数およびクラスオブジェクトの docstring は、その名前がプライベートなものでも検索され、文字列の場合にはそれが docstring であるかのように扱われます。出力においては、`M.__test__` におけるキー `K` は、以下の名前で表示されます

```
<name of M>.__test__.K
```

検索中に見つかったクラスも同様に再帰的に検索が行われ、クラスに含まれているメソッドおよびネストされたクラスについて docstring のテストが行われます。

CPython implementation detail: Prior to version 3.4, extension modules written in C were not fully searched by doctest.

docstring 内の実行例をどのように認識するのか?

ほとんどの場合、対話コンソールセッション上でのコピー／ペーストはうまく動作します。とはいえ、*doctest* は特定の Python シェルの振る舞いを正確にエミュレーションしようとするわけではありません。

```
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print("yes")
... else:
...     print("no")
...     print("NO")
...     print("NO!!!")
...
no
NO
NO!!!
>>>
```

コードを含む最後の '>>> ' または '... ' 行の直下に期待する出力結果が置かれます。(出力結果がもしあれば) それは次の '>>> ' 行か、すべて空白文字の行まで続きます。

詳細事項:

- 期待する出力結果には、空白だけの行が入っていてはなりません。そのような行は期待する出力結果の終了を表すと見なされるからです。もし期待する出力結果の内容に空白行が入っている場合には、空白行が入るべき場所すべてに <BLANKLINE> を入れてください。
- 全てのタブ文字は 8 カラムのタブ位置でスペースに展開されます。テスト対象のコードが作成した出力にあるタブは変更されません。出力例にあるどんなタブも展開 **される** ので、コードの出力がハードタブを含んでいた場合、doctest が通るには *NORMALIZE_WHITESPACE* オプションか *directive* を有効にするしかありません。そうする代わりに、テストが出力を読み取りテストの一部として期待される値と正しく比較するように、テストを書き直すこともできます。このソース中のタブの扱いは試行錯誤の結果であり、タブの扱いの中で最も問題の起きにくいものだということが分かっています。タブの扱いについては、独自の *DocTestParser* クラスを実装して、別のアルゴリズムを使うこともできます。
- 標準出力への出力は取り込まれますが、標準エラーは取り込まれません (例外発生時のトレースバックは別の方法で取り込まれます)。
- 対話セッションにおいて、バックスラッシュを用いて次の行に続ける場合や、その他の理由でバックスラッシュを用いる場合、raw docstring を使ってバックスラッシュを入力どおりに扱わせるようにしなければなりません:

  ```
  >>> def f(x):
  ...     r'''Backslashes in a raw docstring: m\n'''
  >>> print(f.__doc__)
  Backslashes in a raw docstring: m\n
  ```

  そうしない場合は、バックスラッシュは文字列の一部として解釈されます。例えば、上の \n は改行文

字として解釈されてしまいます。それ以外の方法では、doctest にあるバックスラッシュを二重にする (そして raw string を使わない) という方法もあります:

```
>>> def f(x):
...     '''Backslashes in a raw docstring: m\\n'''
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

- 開始カラムはどこでもかまいません:

```
>>> assert "Easy!"
      >>> import math
          >>> math.floor(1.9)
          1
```

期待する出力結果の先頭部にある空白文字列は、実行例の開始部分にあたる '>>> ' 行の先頭にある空白文字列と同じだけ取り除かれます。

実行コンテキストとは何か?

デフォルトでは、*doctest* はテストを行うべき docstring を見つけるたびに M のグローバル名前空間の **浅いコピー** を使い、テストの実行によってモジュールの実際のグローバル名前空間を変更しないようにし、かつ M 内で行ったテストが痕跡を残して偶発的に別のテストを誤って動作させないようにしています。したがって、実行例中では M 内のトップレベルで定義されたすべての名前と、docstring が動作する以前に定義された名前を自由に使えます。個々の実行例は他の docstring 中で定義された名前を参照できません。

testmod() や *testfile()* に globs=your_dict を渡し、自前の辞書を実行コンテキストとして使うこともできます。

例外はどう扱えばよいか?

トレースバックが実行例によって生成される唯一の出力なら問題ありません。単にトレースバックを貼り付けてください。[*1] トレースバックには、頻繁に変更されがちな情報 (例えばファイルパスや行番号など) が入っているものなので、これは受け入れるテスト結果に柔軟性を持たせようと doctest が苦労している部分の一つです。

簡単な例を示しましょう:

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

この doctest は、*ValueError* が送出され、その詳細情報が list.remove(x): x not in list である場合に成功します。

[*1] 期待する出力結果と例外の両方を含んだ例はサポートされていません。一方の終わりと他方の始まりを見分けようとするのはエラーの元になりがちですし、解りにくいテストになってしまいます。

例外が発生したときの期待する出力はトレースバックヘッダから始まっていなければなりません。トレースバックの形式は以下の二通りの行のいずれかで、実行例の最初の行と同じインデントでなければりません:

```
Traceback (most recent call last):
Traceback (innermost last):
```

トレースバックヘッダの後ろにトレースバックスタックが続いてもかまいませんが、doctest はその内容を無視します。普通はトレースバックスタックを省略するか、対話セッションからそのままコピーしてきます。

トレースバックスタックの後ろにはもっとも有意義な部分、例外の型と詳細情報の入った行があります。これは通常トレースバックの最後の行ですが、例外が複数行の詳細情報を持っている場合、複数の行にわたることもあります:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: multi
    line
detail
```

上の例では、最後の 3 行 (*ValueError* から始まる行) における例外の型と詳細情報だけが比較され、それ以外の部分は無視されます。

例外を扱うコツは、実行例をドキュメントとして読む上で明らかに価値のある情報でない限り、トレースバックスタックは省略する、ということです。したがって、先ほどの例は以下のように書くべきでしょう:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
    ...
ValueError: multi
    line
detail
```

トレースバックの扱いは非常に特殊なので注意してください。特に、上の書き直した実行例では、`...` の扱いは doctest の *ELLIPSIS* オプションとは独立しています。この例での省略記号は何かの省略を表しているかもしれませんし、コンマや数字が 3 個 (または 300 個) かもしれませんし、Monty Python のスキットをインデントして書き写したものかもしれません。

以下の詳細はずっと覚えておく必要はないのですが、一度目を通しておいてください:

- doctest は期待する出力の出所が print 文なのか例外なのかを推測できません。したがって、例えば期待する出力が `ValueError: 42 is prime` であるような実行例は、*ValueError* が実際に送出された場合と、万が一期待する出力と同じ文字列を print した場合の両方で成功してしまいます。現実的には、通常の出力がトレースバックヘッダから始まることはないので、実際に問題になることはないでしょう。

- トレースバックスタック (がある場合) の各行は、実行例の最初の行よりも深くインデントされているか、**または** 英数文字以外で始まっていなければなりません。トレースバックヘッダ以後に現れる行のうち、インデントが等しく英数文字で始まる最初の行は例外の詳細情報が書かれた行とみなされるからです。もちろん、本物のトレースバックでは正しく動作します。

- doctest のオプション *IGNORE_EXCEPTION_DETAIL* を指定した場合、最も左端のコロン以後の全ての内容と、例外名の中の全てのモジュール情報が無視されます。

- 対話シェルでは、*SyntaxError* の場合にトレースバックヘッダが省略されることがあります。しかし doctest にとっては、例外を例外でないものと区別するためにトレースバックヘッダが必要です。そこで、トレースバックヘッダを省略するような *SyntaxError* をテストする必要があるというごく稀なケースでは、実行例にトレースバックヘッダを手作業で追加する必要があるでしょう。

- *SyntaxError* の場合、Python は構文エラーの起きた場所を ^ マーカで表示します:

```
>>> 1 1
  File "<stdin>", line 1
    1 1
      ^
SyntaxError: invalid syntax
```

  例外の型と詳細情報の前にエラー位置を示す行がくるため、doctest はこの行を調べません。例えば、以下の例では、間違った場所に ^ マーカを入れても成功してしまいます:

```
>>> 1 1
Traceback (most recent call last):
  File "<stdin>", line 1
    1 1
    ^
SyntaxError: invalid syntax
```

オプションフラグ

doctest の振る舞いは、数多くのオプションフラグによって様々な側面から制御されています。フラグのシンボル名はモジュールの定数として提供され、ビット単位論理和 でつないで様々な関数に渡すことができます。シンボル名は *doctest directives* でも使用でき、doctest コマンドラインインタフェースに `-o` を通して与えることができます。

バージョン 3.4 で追加: `-o` コマンドラインオプション

最初に説明するオプション群は、テストのセマンティクスを決めます。すなわち、実際にテストを実行したときの出力と実行例中の期待する出力とが一致しているかどうかを doctest がどのように判断するかを制御します:

`doctest.DONT_ACCEPT_TRUE_FOR_1`
: デフォルトでは、期待する出力ブロックに単に `1` だけが入っており、実際の出力ブロックに `1` または `True` だけが入っていた場合、これらの出力は一致しているとみなされます。`0` と `False` の場合も同様です。*DONT_ACCEPT_TRUE_FOR_1* を指定すると、こうした値の読み替えを行いません。デフォルトの挙動で読み替えを行うのは、最近の Python で多くの関数の戻り値型が整数型からブール型に変更されたことに対応するためです; 読み替えを行う場合、”通常の整数”の出力を期待する出力とするような doctest も動作します。このオプションはそのうちなくなるでしょうが、ここ数年はそのままでしょう。

`doctest.DONT_ACCEPT_BLANKLINE`
: デフォルトでは、期待する出力ブロックに `<BLANKLINE>` だけの入った行がある場合、その行は実際の

出力における空行に一致するようになります。完全な空行を入れてしまうと期待する出力がそこで終わっているとみなされてしまうため、期待する出力に空行を入れたい場合にはこの方法を使わなければなりません。*DONT_ACCEPT_BLANKLINE* を指定すると、<BLANKLINE> の読み替えを行わなくなります。

doctest.NORMALIZE_WHITESPACE

このフラグを指定すると、連続する空白 (空白と改行文字) は互いに等価であるとみなします。期待する出力における任意の空白列は実際の出力における任意の空白と一致します。デフォルトでは、空白は厳密に一致しなければなりません。*NORMALIZE_WHITESPACE* は、期待する出力の内容が非常に長いために、ソースコード中でその内容を複数行に折り返して書きたい場合に特に便利です。

doctest.ELLIPSIS

このフラグを指定すると、期待する出力中の省略記号マーカ (...) が実際の出力中の任意の部分文字列と一致するようになります。部分文字列は行境界にわたるものや空文字列を含みます。したがって、このフラグを使うのは単純な内容を対象にする場合にとどめましょう。複雑な使い方をすると、正規表現に .* を使ったときのように " しまった、マッチしすぎた！ (match too much!)" と驚くことになりかねません。

doctest.IGNORE_EXCEPTION_DETAIL

このフラグを指定すると、期待する実行結果に例外が入るような実行例で、期待通りの型の例外が送出された場合に、例外の詳細情報が一致していなくてもテストが成功します。例えば、期待する出力が ValueError: 42 であるような実行例は、実際に送出された例外が ValueError: 3*14 でも成功しますが、*TypeError* が送出されるといった場合には成功しません。

また Python 3 の doctest レポートにあるモジュール名も無視します。従ってこのフラグを指定すると、テストを Python 2.7 と Python 3.2 (もしくはそれ以降) のどちらで行ったかに関係無く、どちらに対しても正しく動作します:

```
>>> raise CustomError('message')
Traceback (most recent call last):
CustomError: message

>>> raise CustomError('message')
Traceback (most recent call last):
my_module.CustomError: message
```

なお、*ELLIPSIS* を使っても例外メッセージの詳細を無視することができますが、モジュールの詳細が例外名の一部として表示されるかどうかに依存するようなテストは、やはり失敗します。また、*IGNORE_EXCEPTION_DETAIL* と Python 2.3 の詳細情報を使うことが、例外の詳細に影響されず、なおかつ Python 2.3 以前の Python (これらのリリースは *doctest* **ディレクティブ** をサポートせず、これらを無関係なコメントとして無視します) で成功する doctest を書くための、唯一の明確な方法です。例えば、例外の詳細情報は 2.4 で変更され、"doesn't" の代わりに "does not" と書くようになりましたが:

```
>>> (1, 2)[3] = 'moo'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object doesn't support item assignment
```

とすると、Python 2.3 や Python 2.4 以降の Python バージョンでテストを成功させることができます。

バージョン 3.2 で変更: *doctest* モジュールの *IGNORE_EXCEPTION_DETAIL* フラグが、テストされている例外を含むモジュールの名前を無視するようになりました。

doctest.SKIP
: このフラグを指定すると、実行例は一切実行されません。こうした機能は doctest の実行例がドキュメントとテストを兼ねていて、ドキュメントのためには含めておかなければならないけれどチェックされなくても良い、というような文脈で役に立ちます。例えば、実行例の出力がランダムであるとか、テストドライバーには利用できないリソースに依存している場合などです。

  SKIP フラグは一時的に実行例を” コメントアウト” するのにも使えます。

doctest.COMPARISON_FLAGS
: 上記の比較フラグすべての論理和をとったビットマスクです。

二つ目のオプション群は、テストの失敗を報告する方法を制御します:

doctest.REPORT_UDIFF
: このオプションを指定すると、期待する出力および実際の出力が複数行になるときにテストの失敗結果を unified diff 形式を使って表示します。

doctest.REPORT_CDIFF
: このオプションを指定すると、期待する出力および実際の出力が複数行になるときにテストの失敗結果を context diff 形式を使って表示します。

doctest.REPORT_NDIFF
: このオプションを指定すると、期待する出力と実際の出力との間の差分を `difflib.Differ` を使って算出します。使われているアルゴリズムは有名な `ndiff.py` ユーティリティと同じです。これは、行単位の差分と同じように行内の差分にマーカをつけられるようにする唯一の手段です。例えば、期待する出力のある行に数字の `1` が入っていて、実際の出力には `l` が入っている場合、不一致の起きているカラム位置を示すキャレットの入った行が一行挿入されます。

doctest.REPORT_ONLY_FIRST_FAILURE
: このオプションを指定すると、各 doctest で最初にエラーの起きた実行例だけを表示し、それ以後の実行例の出力を抑制します。これにより、正しく書かれた実行例が、それ以前の実行例の失敗によっておかしくなってしまった場合に、doctest がそれを報告しないようになります。とはいえ、最初に失敗を引き起こした実行例とは関係なく誤って書かれた実行例の報告も抑制してしまいます。*REPORT_ONLY_FIRST_FAILURE* を指定した場合、実行例がどこかで失敗しても、それ以後の実行例を続けて実行し、失敗したテストの総数を報告します; 出力が抑制されるだけです。

doctest.FAIL_FAST
: このオプションを指定すると、最初に実行例が失敗した時点で終了し、残りの実行例を実行しません。つまり、報告される失敗の数はたかだか 1 つになります。最初の失敗より後の例はデバッグ出力を生成しないため、このフラグはデバッグの際に有用でしょう。

  doctest コマンドラインは `-f` を `-o FAIL_FAST` の短縮形として受け付けます。

バージョン 3.4 で追加.

doctest.**REPORTING_FLAGS**

上記のエラー報告に関するフラグすべての論理和をとったビットマスクです。

サブクラス化で *doctest* 内部を拡張するつもりがなければ役に立ちませんが、別の新しいオプションフラグ名を登録する方法もあります:

doctest.**register_optionflag**(*name*)

名前 *name* の新たなオプションフラグを作成し、作成されたフラグの整数値を返します。*register_optionflag()* は *OutputChecker* や *DocTestRunner* をサブクラス化して、その中で新たに作成したオプションをサポートさせる際に使います。*register_optionflag()* は以下のような定形文で呼び出さなければなりません:

```
MY_FLAG = register_optionflag('MY_FLAG')
```

ディレクティブ (Directives)

doctest ディレクティブは個々の実行例の **オプションフラグ** を操作するために使われます。doctest ディレクティブは後のソースコード例にあるような特殊な Python コメントです:

```
directive             ::=   "#" "doctest:" directive_options
directive_options     ::=   directive_option ("," directive_option)\*
directive_option      ::=   on_or_off directive_option_name
on_or_off             ::=   "+" \| "-"
directive_option_name ::=   "DONT_ACCEPT_BLANKLINE" \| "NORMALIZE_WHITESPACE" \| ...
```

+ や - とディレクティブオプション名の間に空白を入れてはなりません。ディレクティブオプション名は上で説明したオプションフラグ名のいずれかです。

ある実行例の doctest ディレクティブは、その実行例だけの doctest の振る舞いを変えます。ある特定の挙動を有効にしたければ + を、無効にしたければ - を使います。

例えば、以下のテストは成功します:

```
>>> print(list(range(20)))
[0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
10,  11, 12, 13, 14, 15, 16, 17, 18, 19]
```

ディレクティブがない場合、実際の出力には一桁の数字の間に二つスペースが入っていないこと、実際の出力は 1 行になることから、テストは成功しないはずです。別のディレクティブを使って、このテストを成功させることもできます:

```
>>> print(list(range(20)))
[0, 1, ..., 18, 19]
```

複数のディレクティブは、一つの物理行の中にコンマで区切って指定できます:

```
>>> print(list(range(20)))
[0,    1, ...,   18,    19]
```

一つの実行例中で複数のディレクティブコメントを使った場合、それらは組み合わされます:

```
>>> print(list(range(20)))
...
[0,    1, ...,   18,    19]
```

この実行例で分かるように、実行例にはディレクティブだけを含む ... 行を追加することができます。この書きかたは、実行例が長すぎるためにディレクティブを同じ行に入れると収まりが悪い場合に便利です:

```
>>> print(list(range(5)) + list(range(10, 20)) + list(range(30, 40)))
...
[0, ..., 4, 10, ..., 19, 30, ..., 39]
```

フォルトではすべてのオプションが無効になっており、ディレクティブは特定の実行例だけに影響を及ぼすので、通常意味があるのは有効にするためのオプション (+ のついたディレクティブ) だけです。とはいえ、doctest を実行する関数はオプションフラグを指定してデフォルトとは異なった挙動を実現できるので、そのような場合には - を使った無効化オプションも意味を持ちます。

警告

doctest では、期待する出力に対する完全一致を厳格に求めます。一致しない文字が一文字でもあると、テストは失敗してしまいます。このため、Python が出力に関して何を保証していて、何を保証していないかを正確に知っていないと混乱させられることがあるでしょう。例えば、集合を出力する際、Python は要素がある特定の順番で並ぶよう保証してはいません。したがって、以下のようなテスト

```
>>> foo()
{"Hermione", "Harry"}
```

は失敗するかもしれないのです! 回避するには

```
>>> foo() == {"Hermione", "Harry"}
True
```

とするのが一つのやり方です。別のやり方は

```
>>> d = sorted(foo())
>>> d
['Harry', 'Hermione']
```

注釈: Python 3.6 以前では、辞書を表示するときに、Python はキーと値のペアはある特定の順序で並ぶよう保証はされていませんでした。

他のやり方もありますが、あとは自分で考えてみてください。

以下のように、オブジェクトアドレスを埋め込むような結果を print するのもよくありません

```
>>> id(1.0) # certain to fail some of the time
7948648
>>> class C: pass
>>> C()   # the default repr() for instances embeds an address
<__main__.C instance at 0x00AC18F0>
```

ELLIPSIS ディレクティブを使うと、上のような例をうまく解決できます:

```
>>> C()
<__main__.C instance at 0x...>
```

浮動小数点数もまた、プラットフォーム間での微妙な出力の違いの原因となります。というのも、Python は浮動小数点の書式化をプラットフォームの C ライブラリに委ねており、この点では、C ライブラリはプラットフォーム間で非常に大きく異なっているからです。

```
>>> 1./7  # risky
0.14285714285714285
>>> print(1./7) # safer
0.142857142857
>>> print(round(1./7, 6)) # much safer
0.142857
```

`I/2.**J` の形式になる数値はどのプラットフォームでもうまく動作するので、私はこの形式の数値を生成するように doctest の実行例を工夫しています:

```
>>> 3./4  # utterly safe
0.75
```

単純な分数は人間にとっても理解しやすく、良いドキュメントを書くために役に立ちます。

26.3.4 基本 API

数 *testmod()* と *testfile()* は、ほとんどの基本的な用途に十分な doctest インタフェースを提供しています。これら二つの関数についてあまり形式的でない入門が読みたければ、**簡単な利用法: *docstring* 中の実行例をチェックする** 節や **簡単な利用法: テキストファイル中の実行例をチェックする** 節を参照してください。

`doctest.testfile`(*filename*, *module_relative=True*, *name=None*, *package=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *parser=DocTestParser()*, *encoding=None*)

filename 以外の引数はすべてオプションで、キーワード引数形式で指定しなければなりません。

filename に指定したファイル内にある実行例をテストします。`(failure_count, test_count)` を返します。

オプション引数の *module_relative* は、ファイル名をどのように解釈するかを指定します:

- *module_relative* が `True` (デフォルト) の場合、*filename* は OS に依存しないモジュールの相対

パスになります。デフォルトでは、このパスは関数 *testfile()* を呼び出しているモジュールからの相対パスになります; ただし、*package* 引数を指定した場合には、パッケージからの相対になります。OS への依存性を除くため、*filename* ではパスを分割する文字に / を使わなければならず、絶対パスにしてはなりません (パス文字列を / で始めてはなりません)。

- *module_relative* が `False` の場合、*filename* は OS 依存のパスを示します。パスは絶対パスでも相対パスでもかまいません; 相対パスにした場合、現在の作業ディレクトリを基準に解決します。

オプション引数 *name* には、テストの名前を指定します; デフォルトの場合や `None` を指定した場合、`os.path.basename(filename)` になります。

オプション引数 *package* には、Python パッケージを指定するか、モジュール相対のファイル名の場合には相対の基準ディレクトリとなる Python パッケージの名前を指定します。パッケージを指定しない場合、関数を呼び出しているモジュールのディレクトリを相対の基準ディレクトリとして使います。*module_relative* を `False` に指定している場合、*package* を指定するとエラーになります。

オプション引数 *globs* には辞書を指定します。この辞書は、実行例を実行する際のグローバル変数として用いられます。doctest はこの辞書の浅いコピーを生成するので、実行例は白紙の状態からスタートします。デフォルトの場合や `None` を指定した場合、新たな空の辞書になります。

オプション引数 *extraglobs* には辞書を指定します。この辞書は、実行例を実行する際にグローバル変数にマージされます。マージは *dict.update()* のように振舞います: *globs* と *extraglobs* との間に同じキー値がある場合、両者を合わせた辞書中には *extraglobs* の方の値が入ります。デフォルト、または `None` であるとき、追加のグローバル変数は使われません。この仕様は、パラメータ付きで doctest を実行するという、やや進んだ機能です。例えば、一般的な名前を使って基底クラス向けに doctest を書いておき、その後で辞書で一般的な名前からテストしたいサブクラスへの対応付けを行う辞書を *extraglobs* に渡して、様々なサブクラスをテストできます。

オプション引数 *verbose* が真の場合、様々な情報を出力します。偽の場合にはテストの失敗だけを報告します。デフォルトの場合や `None` を指定した場合、`sys.argv` に `'-v'` を指定しない限りこの値は真になりません。

オプション引数 *report* が真の場合、テストの最後にサマリを出力します。それ以外の場合には何も出力しません。verbose モードの場合、サマリには詳細な情報を出力しますが、そうでない場合にはサマリはとても簡潔になります (実際には、すべてのテストが成功した場合には何も出力しません)。

オプション引数 *optionflags* (デフォルト値 0) は、各オプションフラグの ビット単位論理和 を取ります。**オプションフラグ** 節を参照してください。

オプション引数 *raise_on_error* の値はデフォルトでは偽です。真にすると、最初のテスト失敗や予期しない例外が起きたときに例外を送出します。このオプションを使うと、失敗の原因を検死デバッグ (post-mortem debug) できます。デフォルトの動作では、実行例の実行を継続します。

オプション引数 *parser* には、*DocTestParser* (またはそのサブクラス) を指定します。このクラスはファイルから実行例を抽出するために使われます。デフォルトでは通常のパーザ (`DocTestParser()`) です。

オプション引数 *encoding* にはファイルをユニコードに変換する際に使われるエンコーディングを指定します。

doctest.**testmod**(*m=None*, *name=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *exclude_empty=False*)

引数はすべてオプションで、*m* 以外の引数はキーワード引数として指定しなければなりません。

モジュール *m* (*m* を指定しないか None にした場合には *__main__*) から到達可能な関数およびクラスの docstring 内にある実行例をテストします。m.__doc__ 内の実行例からテストを開始します。

また、辞書 m.__test__ が存在し、None でない場合、この辞書から到達できる実行例もテストします。m.__test__ は、(文字列の) 名前から関数、クラスおよび文字列への対応付けを行っています。関数およびクラスの場合には、その docstring 内から実行例を検索します。文字列の場合には、docstring と同じようにして実行例の検索を直接実行します。

モジュール *m* に属するオブジェクトにつけられた docstring のみを検索します。

(failure_count, test_count) を返します。

オプション引数 *name* には、モジュールの名前を指定します。デフォルトの場合や None を指定した場合には、m.__name__ を使います。

オプション引数 *exclude_empty* はデフォルトでは偽になっています。この値を真にすると、doctest を持たないオブジェクトを考慮から外します。デフォルトの設定は依存のバージョンとの互換性を考えたハックであり、doctest.master.summarize() と *testmod()* を合わせて利用しているようなコードでも、テスト実行例を持たないオブジェクトから出力を得るようにしています。新たに追加された *DocTestFinder* のコンストラクタの *exclude_empty* はデフォルトで真になります。

オプション引数 *extraglobs*, *verbose*, *report*, *optionflags*, *raise_on_error*, および *globs* は上で説明した *testfile()* の引数と同じです。ただし、*globs* のデフォルト値は m.__dict__ になります。

doctest.**run_docstring_examples**(*f*, *globs*, *verbose=False*, *name="NoName"*, *compileflags=None*, *optionflags=0*)

オブジェクト *f* に関連付けられた実行例をテストします。*f* は文字列、モジュール、関数、またはクラスオブジェクトです。

引数 *globs* に辞書を指定すると、その浅いコピーを実行コンテキストに使います。

オプション引数 *name* はテスト失敗時のメッセージに使われます。デフォルトの値は 'NoName' です。

オプション引数 *verbose* の値を真にすると、テストが失敗しなくても出力を生成します。デフォルトでは、実行例のテストに失敗したときのみ出力を生成します。

オプション引数 *compileflags* には、実行例を実行するときに Python バイトコードコンパイラが使うフラグを指定します。デフォルトの場合や None を指定した場合、フラグは *globs* 内にある future 機能セットに対応したものになります。

オプション引数 *optionflags* は、上で述べた *testfile()* と同様の働きをします。

26.3.5 単体テスト API

doctest が付けられたモジュールが大きくなるにつれ、全ての doctest を組織的に実行したくなるでしょう。*doctest* モジュールは 2 つの関数を提供していて、*unittest* のテストスイートを作り、doctest を含んだテキストファイルを作成するのに使えます。*unittest* のテストディスカバリと組み合わせるには、テストモジュールに `load_tests()` 関数を書いておいてください:

```
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
    return tests
```

doctest の入ったテキストファイルやモジュールから *unittest.TestSuite* インスタンスを生成するための主な関数は二つあります:

`doctest.DocFileSuite`(**paths*, *module_relative=True*, *package=None*, *setUp=None*, *tearDown=None*, *globs=None*, *optionflags=0*, *parser=DocTestParser()*, *encoding=None*)

単一または複数のテキストファイルに入っている doctest 形式のテストを、*unittest.TestSuite* インスタンスに変換します。

この関数の返す *unittest.TestSuite* インスタンスは、unittest フレームワークで動作させ、各ファイルの実行例を対話的に実行するためのものです。ファイル内の何らかの実行例の実行に失敗すると、この関数で生成した単体テストは失敗し、該当するテストの入っているファイルの名前と、(場合によりだいたいの) 行番号の入った `failureException` 例外を送出します。

関数には、テストを行いたい一つまたは複数のファイルへのパスを (文字列で) 渡します。

DocFileSuite() には、キーワード引数でオプションを指定できます:

オプション引数 *module_relative* は *paths* に指定したファイル名をどのように解釈するかを指定します:

- *module_relative* が `True` (デフォルト) の場合、*paths* 中のファイル名は OS に依存しないモジュールの相対パスになります。デフォルトでは、このパスは呼び出し元のモジュールのディレクトリからの相対パスになります; ただし、*package* 引数を指定した場合には、パッケージからの相対になります。OS への依存性を除くため、各ファイル名はパスを分割するのに / 文字を使わなければならず、絶対パスにしてはなりません (パス文字列を / で始めてはなりません)。
- *module_relative* が `False` の場合、*filename* は OS 依存のパスを示します。パスは絶対パスでも相対パスでもかまいません; 相対パスにした場合、現在の作業ディレクトリを基準に解決します。

オプション引数 *package* には、Python パッケージを指定するか、モジュール相対のファイル名の場合には相対の基準ディレクトリとなる Python パッケージの名前を指定します。パッケージを指定しない場合、関数を呼び出しているモジュールのディレクトリを相対の基準ディレクトリとして使います。*module_relative* を `False` に指定している場合、*package* を指定するとエラーになります。

オプション引数 *setUp* には、テストスイートのセットアップに使う関数を指定します。この関数は、各ファイルのテストを実行する前に呼び出されます。*setUp* 関数は *DocTest* オブジェクトに引き渡されます。*setUp* は *globs* 属性を介してテストのグローバル変数にアクセスできます。

オプション引数 *tearDown* には、テストを解体 (tear-down) するための関数を指定します。この関数は、各ファイルのテストの実行を終了するたびに呼び出されます。*tearDown* 関数は *DocTest* オブジェクトに引き渡されます。*tearDown* は *globs* 属性を介してテストのグローバル変数にアクセスできます。

オプション引数 *globs* は辞書で、テストのグローバル変数の初期値が入ります。この辞書は各テストごとに新たにコピーして使われます。デフォルトでは *globs* は空の新たな辞書です。

オプション引数 *optionflags* には、テストを実行する際にデフォルトで適用される doctest オプションを OR で結合して指定します。**オプションフラグ** 節を参照してください。結果レポートに関するオプションを指定するより適切な方法は下記の *set_unittest_reportflags()* の説明を参照してください。

オプション引数 *parser* には、*DocTestParser* (またはそのサブクラス) を指定します。このクラスはファイルから実行例を抽出するために使われます。デフォルトでは通常のパーザ (`DocTestParser()`) です。

オプション引数 *encoding* にはファイルをユニコードに変換する際に使われるエンコーディングを指定します。

DocFileSuite() を使用してテキストファイルからロードされた doctests に提供される globals に、グローバル変数 `__file__` が追加されます。

`doctest.DocTestSuite`(*module=None*, *globs=None*, *extraglobs=None*, *test_finder=None*, *setUp=None*, *tearDown=None*, *checker=None*)

doctest のテストを *unittest.TestSuite* に変換します。

この関数の返す *unittest.TestSuite* インスタンスは、unittest フレームワークで動作させ、モジュール内の各 doctest を実行するためのものです。何らかの doctest の実行に失敗すると、この関数で生成した単体テストは失敗し、該当するテストの入っているファイルの名前と、(場合によりだいたいの) 行番号の入った `failureException` 例外を送出します。

オプション引数 *module* には、テストしたいモジュールの名前を指定します。*module* にはモジュールオブジェクトまたは (ドット表記の) モジュール名を指定できます。*module* を指定しない場合、この関数を呼び出しているモジュールになります。

オプション引数 *globs* は辞書で、テストのグローバル変数の初期値が入ります。この辞書は各テストごとに新たにコピーして使われます。デフォルトでは *globs* は空の新たな辞書です。

オプション引数 *extraglobs* には追加のグローバル変数セットを指定します。この変数セットは *globs* に統合されます。デフォルトでは、追加のグローバル変数はありません。

オプション引数 *test_finder* は、モジュールから doctest を抽出するための *DocTestFinder* オブジェクト (またはその代替となるオブジェクト) です。

オプション引数 *setUp* 、*tearDown* 、および *optionflags* は上の *DocFileSuite()* と同じです。

この関数は *testmod()* と同じ検索方法を使います。

バージョン 3.5 で変更: *module* がドキュメンテーション文字列を含まない場合には、*DocTestSuite()* は *ValueError* を送出するのではなく空の *unittest.TestSuite* を返します。

裏側では *DocTestSuite()* は doctest.DocTestCase インスタンスから *unittest.TestSuite* を作成しており、DocTestCase は *unittest.TestCase* のサブクラスになっています。DocTestCase についてはここでは説明しません (これは内部実装上の詳細だからです) が、そのコードを調べてみれば、*unittest* の組み込みの詳細に関する疑問を解決できるはずです。

同様に、*DocFileSuite()* は doctest.DocFileCase インスタンスから *unittest.TestSuite* を作成し、DocFileCase は DocTestCase のサブクラスになっています。

そのため、*unittest.TestSuite* クラスを生成するどちらの方法も DocTestCase のインスタンスを実行します。これは次のような微妙な理由で重要です: *doctest* 関数を自分で実行する場合、オプションフラグを *doctest* 関数に渡すことで、*doctest* のオプションを直接操作できます。しかしながら、*unittest* フレームワークを書いている場合には、いつどのようにテストを動作させるかを *unittest* が完全に制御してしまいます。フレームワークの作者はたいてい、*doctest* のレポートオプションを (コマンドラインオプションで指定するなどして) 操作したいと考えますが、*unittest* を介して *doctest* のテストランナーにオプションを渡す方法は存在しないのです。

このため、*doctest* では、以下の関数を使って、*unittest* サポートに特化したレポートフラグ表記方法もサポートしています:

doctest.**set_unittest_reportflags**(*flags*)

doctest のレポートフラグをセットします。

引数 *flags* はオプションフラグの ビット単位論理和 を取ります。**オプションフラグ** 節を参照してください。「レポートフラグ」しか使えません。

この関数で設定した内容はモジュール全体にわたるものであり、関数呼び出し以後に *unittest* モジュールから実行されるすべての doctest に影響します: DocTestCase の runTest() メソッドは、DocTestCase インスタンスが作成された際に、現在のテストケースに指定されたオプションフラグを見に行きます。レポートフラグが指定されていない場合 (通常の場合で、望ましいケースです)、*doctest* の *unittest* レポートフラグが bit ごとの OR で結合され、doctest を実行するために作成される *DocTestRunner* インスタンスに渡されます。DocTestCase インスタンスを構築する際に何らかのレポートフラグが指定されていた場合、*doctest* の *unittest* レポートフラグは無視されます。

この関数は、関数を呼び出す前に有効になっていた *unittest* レポートフラグの値を返します。

26.3.6 拡張 API

基本 API は、doctest を使いやすくするための簡単なラッパであり、柔軟性があってほとんどのユーザの必要を満たしています; とはいえ、もっとテストをきめ細かに制御したい場合や、doctest の機能を拡張したい場合、拡張 API (advanced API) を使わなければなりません。

拡張 API は、doctest ケースから抽出した対話モードでの実行例を記憶するための二つのコンテナクラスを中心に構成されています:

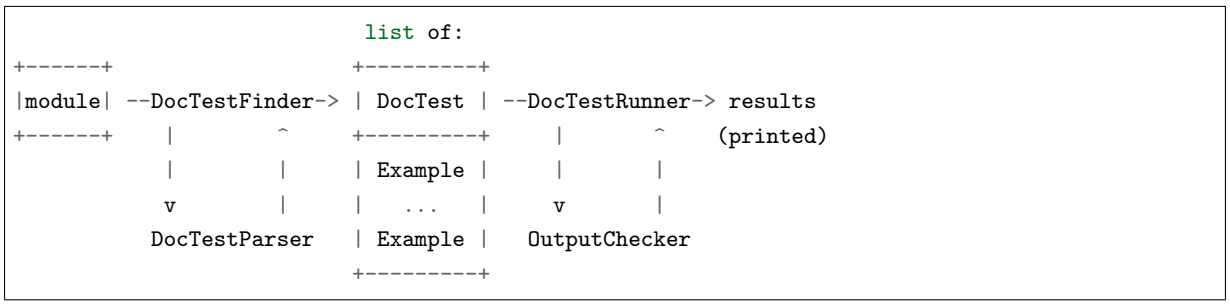
- *Example*: 1 つの Python **文** と、その期待する出力をペアにしたもの。

- *DocTest*: *Example* の集まり。通常一つの docstring やテキストファイルから抽出されます。

その他に、doctest の実行例を検索、構文解析、実行、チェックするための処理クラスが以下のように定義されています:

- *DocTestFinder*: 与えられたモジュールからすべての docstring を検索し、*DocTestParser* を使って対話モードでの実行例が入ったすべての docstring から *DocTest* を生成します。
- *DocTestParser*: (オブジェクトの docstring 等の) 文字列から *DocTest* オブジェクトを生成します。
- *DocTestRunner*: *DocTest* 内の実行例を実行し、*OutputChecker* を使って出力を検証します。
- *OutputChecker*: doctest 実行例から実際に出力された結果を期待する出力と比較し、両者が一致するか判別します。

これらの処理クラスの関係を図にまとめると、以下のようになります:



DocTest オブジェクト

class doctest.**DocTest**(*examples*, *globs*, *name*, *filename*, *lineno*, *docstring*)

単一の名前空間内で実行される doctest 実行例の集まりです。コンストラクタの引数は *DocTest* インスタンス中の同名の属性の初期化に使われます。

DocTest では、以下の属性を定義しています。これらの変数はコンストラクタで初期化されます。直接変更してはなりません。

examples

対話モードにおける実行例それぞれをエンコードしていて、テストで実行される、*Example* オブジェクトからなるリストです。

globs

実行例を実行する名前空間 (いわゆるグローバル変数) です。このメンバは、名前から値への対応付けを行っている辞書です。実行例が名前空間に対して (新たな変数を束縛するなど) 何らかの変更を行った場合、*globs* への反映はテストの実行後に起こります。

name

DocTest を識別する名前の文字列です。通常、この値はテストを取り出したオブジェクトかファイルの名前になります。

filename

DocTest を取り出したファイルの名前です; ファイル名が未知の場合や *DocTest* をファイルから

取り出したのでない場合には None になります。

lineno
filename 中で *DocTest* のテスト実行例が始まっている行の行番号で、行番号が利用できなければ None です。行番号は、ファイルの先頭を 0 として数えます。

docstring
テストを取り出した docstring 自体を現す文字列です。docstring 文字列を得られない場合や、文字列からテスト実行例を取り出したのでない場合には None になります。

Example オブジェクト

class doctest.**Example**(*source*, *want*, *exc_msg=None*, *lineno=0*, *indent=0*, *options=None*)
ひとつの Python 文と、それに対する期待する出力からなる、単一の対話的モードの実行例です。コンストラクタの引数は *Example* インスタンス中の同名の属性の初期化に使われます。

Example では、以下の属性を定義しています。これらの変数はコンストラクタで初期化されます。直接変更してはなりません。

source
実行例のソースコードが入った文字列です。ソースコードは単一の Python で、末尾は常に改行です。コンストラクタは必要に応じて改行を追加します。

want
実行例のソースコードを実行した際の期待する出力 (標準出力と、例外が生じた場合にはトレースバック) です。*want* の末尾は、期待する出力がまったくない場合を除いて常に改行になります。期待する出力がない場合には空文字列になります。コンストラクタは必要に応じて改行を追加します。

exc_msg
実行例が例外を生成すると期待される場合の例外メッセージです。例外を送出しない場合には None です。この例外メッセージは、*traceback.format_exception_only()* の戻り値と比較されます。値が None でない限り、*exc_msg* は改行で終わっていなければなりません; コンストラクタは必要に応じて改行を追加します。

lineno
この実行例を含む文字列における実行例が始まる行番号です。行番号は文字列の先頭を 0 として数えます。

indent
実行例の入っている文字列のインデント、すなわち実行例の最初のプロンプトより前にある空白文字の数です。

options
オプションフラグを True または False に対応付けている辞書です。実行例に対するデフォルトオプションを上書きするために用いられます。この辞書に入っていないオプションフラグはデフォルトの状態 (*DocTestRunner* の optionflags の内容) のままになります。

DocTestFinder オブジェクト

class doctest.DocTestFinder(*verbose=False*, *parser=DocTestParser()*, *recurse=True*, *exclude_empty=True*)

与えられたオブジェクトについて、そのオブジェクト自身の docstring か、そのオブジェクトに含まれるオブジェクトの docstring から *DocTest* を抽出する処理クラスです。モジュール、クラス、関数、メソッド、静的メソッド、クラスメソッド、プロパティから *DocTest* を抽出できます。

オプション引数 *verbose* を使うと、抽出処理の対象となるオブジェクトを表示できます。デフォルトは False (出力を行わない) です。

オプション引数 *parser* には、docstring から *DocTest* を抽出するのに使う *DocTestParser* オブジェクト (またはその代替となるオブジェクト) を指定します。

オプション引数 *recurse* が偽の場合、*DocTestFinder.find()* は与えられたオブジェクトだけを調べ、そのオブジェクトに含まれる他のオブジェクトを調べません。

オプション引数 *exclude_empty* が偽の場合、*DocTestFinder.find()* は空の docstring を持つオブジェクトもテスト対象に含めます。

DocTestFinder では以下のメソッドを定義しています:

find(*obj[, name][, module][, globs][, extraglobs]*)

obj または *obj* 内に入っているオブジェクトの docstring 中で定義されている *DocTest* のリストを返します。

オプション引数 *name* には、オブジェクトの名前を指定します。この名前は、関数が返す *DocTest* の名前になります。*name* を指定しない場合、obj.__name__ を使います。

オプションのパラメータ *module* は、指定したオブジェクトを収めているモジュールを指定します。*module* を指定しないか、None を指定した場合には、正しいモジュールを自動的に決定しようと試みます。オブジェクトのモジュールは以下のような役割を果たします:

- *globs* を指定していない場合、オブジェクトのモジュールはデフォルトの名前空間になります。
- 他のモジュールから import されたオブジェクトに対して *DocTestFinder* が *DocTest* を抽出するのを避けるために使います。(*module* 由来でないオブジェクトを無視します。)
- オブジェクトの入っているファイル名を調べるために使います。
- オブジェクトがファイル内の何行目にあるかを調べる手助けにします。

module が False の場合には、モジュールの検索を試みません。これは正確さを欠くような使い方で、通常 doctest 自体のテストにしか使いません。*module* が False の場合、または *module* が None で自動的に的確なモジュールを見つけ出せない場合には、すべてのオブジェクトは (non-existent) モジュールに属するとみなされ、そのオブジェクト内のすべてのオブジェクトに対して (再帰的に) doctest の検索を行います。

各 *DocTest* のグローバル変数は、*globs* と *extraglobs* を合わせたもの (*extraglobs* 内の束縛が *globs* 内の束縛を上書きする) になります。各々の *DocTest* に対して、グローバル変数を表す辞書の新たな浅いコピーを生成します。*globs* を指定しない場合に使われるのデフォルト値は、モ

ジュールを指定していればそのモジュールの *__dict__* になり、指定していなければ {} になります。*extraglobs* を指定しない場合、デフォルトの値は {} になります。

DocTestParser オブジェクト

class doctest.**DocTestParser**

対話モードの実行例を文字列から抽出し、それを使って *DocTest* オブジェクトを生成するために使われる処理クラスです。

DocTestParser では以下のメソッドを定義しています:

get_doctest(*string*, *globs*, *name*, *filename*, *lineno*)

指定した文字列からすべての doctest 実行例を抽出し、*DocTest* オブジェクト内に集めます。

globs, *name*, *filename*, および *lineno* は新たに作成される *DocTest* オブジェクトの属性になります。詳しくは *DocTest* のドキュメントを参照してください。

get_examples(*string*, *name='<string>'*)

指定した文字列からすべての doctest 実行例を抽出し、*Example* オブジェクトからなるリストにして返します。各 *Example* の行番号は 0 から数えます。オプション引数 *name* はこの文字列につける名前で、エラーメッセージにしか使われません。

parse(*string*, *name='<string>'*)

指定した文字列を、実行例とその間のテキストに分割し、実行例を *Example* オブジェクトに変換し、*Example* と文字列からなるリストにして返します。各 *Example* の行番号は 0 から数えます。オプション引数 *name* はこの文字列につける名前で、エラーメッセージにしか使われません。

DocTestRunner オブジェクト

class doctest.**DocTestRunner**(*checker=None*, *verbose=None*, *optionflags=0*)

DocTest 内の対話モード実行例を実行し、検証する際に用いられる処理クラスです。

期待する出力と実際の出力との比較は *OutputChecker* で行います。比較は様々なオプションフラグを使ってカスタマイズできます; 詳しくは **オプションフラグ** を参照してください。オプションフラグでは不十分な場合、コンストラクタに *OutputChecker* のサブクラスを渡して比較方法をカスタマイズできます。

テストランナーの表示出力の制御には二つの方法があります。一つ目は、TestRunner.run() に出力用の関数を渡すというものです。この関数は、表示すべき文字列を引数にして呼び出されます。デフォルトは sys.stdout.write です。出力を取り込んで処理するだけでは不十分な場合、*DocTestRunner* をサブクラス化し、*report_start()*, *report_success()*, *report_unexpected_exception()*, および *report_failure()* をオーバライドすればカスタマイズできます。

オプションのキーワード引数 *checker* には、*OutputChecker* オブジェクト (またはその代替となるオブジェクト) を指定します。このオブジェクトは doctest 実行例の期待する出力と実際の出力との比較を行う際に使われます。

オプションのキーワード引数 *verbose* は、*DocTestRunner* の出すメッセージの冗長性を制御します。

verbose が True の場合、各実行例を実行する都度、その実行例についての情報を出力します。*verbose* が False の場合、テストの失敗だけを出力します。*verbose* を指定しない場合や None を指定した場合、コマンドラインスイッチ -v を使った場合にのみ *verbose* 出力を適用します。

オプションのキーワード引数 *optionflags* を使うと、テストランナーが期待される出力と実際の出力を比較する方法や、テストの失敗を表示する方法を制御できます。詳しくは **オプションフラグ** 節を参照してください。

DocTestParser では以下のメソッドを定義しています:

report_start(*out*, *test*, *example*)

テストランナーが実行例を処理しようとしているときにレポートを出力します。*DocTestRunner* の出力をサブクラスでカスタマイズできるようにするためのメソッドです。直接呼び出してはなりません。

example は処理する実行例です。*test* は *example* の入っているテストです。*out* は出力用の関数で、*DocTestRunner.run()* に渡されます。

report_success(*out*, *test*, *example*, *got*)

与えられた実行例が正しく動作したことを報告します。このメソッドは *DocTestRunner* のサブクラスで出力をカスタマイズできるようにするために提供されています; 直接呼び出してはなりません。

example は処理する実行例です。*got* は実行例から実際に得られた出力です。*test* は *example* の入っているテストです。*out* は出力用の関数で、*DocTestRunner.run()* に渡されます。

report_failure(*out*, *test*, *example*, *got*)

与えられた実行例が正しく動作しなかったことを報告します。このメソッドは *DocTestRunner* のサブクラスで出力をカスタマイズできるようにするために提供されています; 直接呼び出してはなりません。

example は処理する実行例です。*got* は実行例から実際に得られた出力です。*test* は *example* の入っているテストです。*out* は出力用の関数で、*DocTestRunner.run()* に渡されます。

report_unexpected_exception(*out*, *test*, *example*, *exc_info*)

与えられた実行例が期待とは違う例外を送出したことを報告します。このメソッドは *DocTestRunner* のサブクラスで出力をカスタマイズできるようにするために提供されています; 直接呼び出してはなりません。

example は処理する実行例です。*exc_info* には予期せず送出された例外の情報を入れたタプル (*sys.exc_info()* の返す内容) になります。*test* は *example* の入っているテストです。*out* は出力用の関数で、*DocTestRunner.run()* に渡されます。

run(*test*, *compileflags=None*, *out=None*, *clear_globs=True*)

test 内の実行例 (*DocTest* オブジェクト) を実行し、その結果を出力用の関数 *out* を使って表示します。

実行例は名前空間 test.globs の下で実行されます。*clear_globs* が真 (デフォルト) の場合、名前空間はテストの実行後に消去され、ガベージコレクションを促します。テストの実行完了後にそ

の内容を調べたければ、*clear_globs=False* としてください。

compileflags には、実行例を実行する際に Python コンパイラに適用するフラグセットを指定します。*compileflags* を指定しない場合、デフォルト値は *globs* で適用されている future-import フラグセットになります。

各実行例の出力は *DocTestRunner* の出力チェッカで検査され、その結果は `DocTestRunner.report_*()` メソッドで書式化されます。

summarize(*verbose=None*)

この DocTestRunner が実行したすべてのテストケースのサマリを出力し、**名前付きタプル** `TestResults(failed, attempted)` を返します。

オプションの *verbose* 引数を使うと、どのくらいサマリを詳しくするかを制御できます。冗長度を指定しない場合、*DocTestRunner* 自体の冗長度を使います。

OutputChecker オブジェクト

class doctest.OutputChecker

doctest 実行例を実際に実行したときの出力が期待する出力と一致するかどうかをチェックするために使われるクラスです。*OutputChecker* では、与えられた二つの出力を比較して、一致する場合には `True` を返す *check_output()* と、二つの出力間の違いを説明する文字列を返す *output_difference()* の、二つのメソッドがあります。

OutputChecker では以下のメソッドを定義しています:

check_output(*want*, *got*, *optionflags*)

実行例から実際に得られた出力 (*got*) と、期待する出力 (*want*) が一致する場合にのみ `True` を返します。二つの文字列がまったく同一の場合には常に一致するとみなしますが、テストランナーの使っているオプションフラグにより、厳密には同じ内容になっていなくても一致するとみなす場合もあります。オプションフラグについての詳しい情報は **オプションフラグ** 節を参照してください。

output_difference(*example*, *got*, *optionflags*)

与えられた実行例 (*example*) の期待する出力と、実際に得られた出力 (*got*) の間の差異を解説している文字列を返します。*optionflags* は *want* と *got* を比較する際に使われるオプションフラグのセットです。

26.3.7 デバッグ

doctest では、doctest 実行例をデバッグするメカニズムをいくつか提供しています:

- doctest を実行可能な Python プログラムに変換し、Python デバッガ *pdb* で実行できるようにするための関数がいくつかあります。
- *DocTestRunner* のサブクラス *DebugRunner* クラスがあります。このクラスは、最初に失敗した実行例に対して例外を送出します。例外には実行例に関する情報が入っています。この情報は実行例の検死デバッグに利用できます。

- *DocTestSuite()* の生成する *unittest* テストケースは、*debug()* メソッドをサポートしています。*debug()* は *unittest.TestCase* で定義されています。

- *pdb.set_trace()* を doctest 実行例の中で呼び出しておけば、その行が実行されたときに Python デバッガが組み込まれます。デバッガを組み込んだあとは、変数の現在の値などを調べられます。たとえば、以下のようなモジュールレベルの docstring の入ったファイル `a.py` があるとします:

```
"""
>>> def f(x):
...     g(x*2)
>>> def g(x):
...     print(x+3)
...     import pdb; pdb.set_trace()
>>> f(3)
9
"""
```

対話セッションは以下のようになるでしょう:

```
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
  1     def g(x):
  2         print(x+3)
  3  ->     import pdb; pdb.set_trace()
[EOF]
(Pdb) p x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
-> g(x*2)
(Pdb) list
  1     def f(x):
  2  ->     g(x*2)
[EOF]
(Pdb) p x
3
(Pdb) step
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>>
```

以下は、doctest を Python コードに変換して、できたコードをデバッガ下で実行できるようにするための関数です:

doctest.**script_from_examples**(*s*)

実行例の入ったテキストをスクリプトに変換します。

引数 *s* は doctest 実行例の入った文字列です。この文字列は Python スクリプトに変換され、その中では *s* の doctest 実行例が通常のコードに、それ以外は Python のコメント文になります。生成したスクリプトを文字列で返します。例えば、

```
import doctest
print(doctest.script_from_examples(r"""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print(x+y)
    3
"""))
```

は:

```
# Set x and y to 1 and 2.
x, y = 1, 2
#
# Print their sum:
print(x+y)
# Expected:
## 3
```

この関数は内部的に他の関数から使われていますが (下記参照)、対話セッションを Python スクリプトに変換したいような場合にも便利でしょう。

doctest.**testsource**(*module*, *name*)

あるオブジェクトの doctest をスクリプトに変換します。

引数 *module* はモジュールオブジェクトか、対象の doctest を持つオブジェクトの入ったモジュールのドット表記名です。引数 *name* は対象の doctest を持つオブジェクトの (モジュール内の) 名前です。対象オブジェクトの docstring を上記の *script_from_examples()* で説明した方法で Python スクリプトに変換してできた文字列を返します。例えば、`a.py` モジュールのトップレベルに関数 `f()` がある場合、以下のコード

```
import a, doctest
print(doctest.testsource(a, "a.f"))
```

を実行すると、`f()` の docstring から doctest をコードに変換し、それ以外をコメントにしたスクリプトを出力します。

doctest.**debug**(*module*, *name*, *pm=False*)

オブジェクトの持つ doctest をデバッグします。

module および *name* 引数は上の *testsource()* と同じです。指定したオブジェクトの docstring から合成された Python スクリプトは一時ファイルに書き出され、その後 Python デバッガ *pdb* の制御

下で実行されます。

ローカルおよびグローバルの実行コンテキストには、module.__dict__ の浅いコピーが使われます。

オプション引数 *pm* は、検死デバッグを行うかどうかを指定します。*pm* が真の場合、スクリプトファイルは直接実行され、スクリプトが送出した例外が処理されないまま終了した場合にのみデバッガが立ち入ります。その場合、*pdb.post_mortem()* によって検死デバッグを起動し、処理されなかった例外から得られたトレースバックオブジェクトを渡します。*pm* を指定しないか値を偽にした場合、*pdb.run()* に適切な *exec()* 呼び出しを渡して、最初からデバッガの下でスクリプトを実行します。

doctest.**debug_src**(*src*, *pm=False*, *globs=None*)

文字列中の doctest をデバッグします。

上の *debug()* に似ていますが、doctest の入った文字列は *src* 引数で直接指定します。

オプション引数 *pm* は上の *debug()* と同じ意味です。

オプション引数 *globs* には、ローカルおよびグローバルな実行コンテキストの両方に使われる辞書を指定します。*globs* を指定しない場合や None にした場合、空の辞書を使います。辞書を指定した場合、実際の実行コンテキストには浅いコピーが使われます。

DebugRunner クラス自体や *DebugRunner* クラスが送出する特殊な例外は、テストフレームワークの作者にとって非常に興味のあるところですが、ここでは概要しか述べられません。詳しくはソースコード、とりわけ *DebugRunner* の docstring (それ自体 doctest です!) を参照してください:

class doctest.**DebugRunner**(*checker=None*, *verbose=None*, *optionflags=0*)

テストの失敗に遭遇するとすぐに例外を送出するようになっている *DocTestRunner* のサブクラスです。予期しない例外が生じると、*UnexpectedException* 例外を送出します。この例外には、テスト、実行例、もともと送出された例外が入っています。期待する出力と実際の出力が一致しないために失敗した場合には、*DocTestFailure* 例外を送出します。この例外には、テスト、実行例、実際の出力が入っています。

コンストラクタのパラメータやメソッドについては、**拡張** *API* 節の *DocTestRunner* のドキュメントを参照してください。

DebugRunner インスタンスの送出する例外には以下の二つがあります:

exception doctest.**DocTestFailure**(*test*, *example*, *got*)

doctest 実行例の実際の出力が期待する出力と一致しなかったことを示すために *DocTestRunner* が送出する例外です。コンストラクタの引数は、インスタンスの同名の属性を初期化するために使われます。

DocTestFailure では以下の属性を定義しています:

DocTestFailure.**test**

実行例が失敗した時に実行されていた *DocTest* オブジェクトです。

DocTestFailure.**example**

失敗した *Example* オブジェクトです。

DocTestFailure.**got**

実行例の実際の出力です。

exception doctest.**UnexpectedException**(*test*, *example*, *exc_info*)

doctest 実行例が予期しない例外を送出したことを示すために *DocTestRunner* が送出する例外です。コンストラクタの引数は、インスタンスの同名の属性を初期化するために使われます。

UnexpectedException では以下の属性を定義しています:

UnexpectedException.test

実行例が失敗した時に実行されていた *DocTest* オブジェクトです。

UnexpectedException.example

失敗した *Example* オブジェクトです。

UnexpectedException.exc_info

予期しない例外についての情報の入ったタプルで、*sys.exc_info()* が返すのと同じものです。

26.3.8 アドバイス

冒頭でも触れたように、*doctest* は、以下の三つの主な用途を持つようになりました:

1. docstring 内の実行例をチェックする。
2. 回帰テストを行う。
3. 実行可能なドキュメント/読めるテストの実現。

これらの用途にはそれぞれ違った要求があるので、区別して考えるのが重要です。特に、docstring を曖昧なテストケースに埋もれさせてしまうとドキュメントとしては最悪です。

docstring の例は注意深く作成してください。doctest の作成にはコツがあり、きちんと学ぶ必要があります --- 最初はすんなりできないでしょう。実行例はドキュメントに本物の価値を与えます。良い例は、たくさんの言葉と同じ価値を持つことがしばしばあります。注意深くやれば、例はユーザにとっては非常に有益であり、時を経るにつれて、あるいは状況が変わった際に、何度も修正するのにかかる時間を節約するという形で、きっと見返りを得るでしょう。私は今でも、自分の *doctest* 実行例が ” 無害な” 変更を行った際にうまく動作しなくなることに驚いています。

説明テキストの作成をけちらなければ、*doctest* は回帰テストの優れたツールにもなり得ます。説明文と実行例を交互に記述していけば、実際に何をどうしてテストしているのかもっと簡単に把握できるようになるでしょう。もちろん、コードベースのテストに詳しくコメントを入れるのも手ですが、そんなことをするプログラマはほとんどいません。多くの人々が、*doctest* のアプローチをとった方がきれいにテストを書けると気づいています。おそらく、これは単にコード中にコメントを書くのが少し面倒だからという理由でしょう。私はもう少し穿った見方もしています。doctest ベースのテストを書くときの自然な態度は、自分のソフトウェアのよい点を説明しようとして、実行例を使って説明しようとするときの態度そのものだからだ、という理由です。それゆえに、テストファイルは自然と単純な機能の解説から始め、論理的により複雑で境界条件的なケースに進むような形になります。結果的に、一見ランダムに見えるような個別の機能をテストしている個別の関数の集まりではなく、首尾一貫した説明ができるようになるのです。*doctest* によるテストの作成はまったく別の取り組み方であり、テストと説明の区別をなくして、まったく違う結果を生み出すのです。

回帰テストは特定のオブジェクトやファイルにまとめておくのがよいでしょう。回帰テストの組み方にはいく

つか選択肢があります:

- テストケースを対話モードの実行例にして入れたテキストファイルを書き、*testfile()* や *DocFileSuite()* を使ってそのファイルをテストします。この方法をお勧めします。最初から doctest を使うようにしている新たなプロジェクトでは、この方法が一番簡単です。
- `_regrtest_topic` という名前の関数を定義します。この関数には、あるトピックに対応するテストケースの入った docstring が一つだけ入っています。この関数はモジュールと同じファイルの中にも置けますし、別のテストファイルに分けてもかまいません。
- 回帰テストのトピックをテストケースの入った docstring に対応付けた辞書 `__test__` 辞書を定義します。

テストコードをモジュールに組み込むことで、そのモジュールは、モジュール自身がテストランナーになることができます。テストが失敗した場合には、問題箇所をデバッグする際に、失敗した doctest だけを再度実行することで、作成したテストランナーを修正することができます。これがこのようなテストランナーの最小例となります。

```
if __name__ == '__main__':
    import doctest
    flags = doctest.REPORT_NDIFF|doctest.FAIL_FAST
    if len(sys.argv) > 1:
        name = sys.argv[1]
        if name in globals():
            obj = globals()[name]
        else:
            obj = __test__[name]
        doctest.run_docstring_examples(obj, globals(), name=name,
                                       optionflags=flags)
    else:
        fail, total = doctest.testmod(optionflags=flags)
        print("{} failures out of {} tests".format(fail, total))
```

脚注

26.4 unittest --- ユニットテストフレームワーク

ソースコード: Lib/unittest/__init__.py

(すでにテストの基本概念について詳しいようでしたら、この部分をとばして **アサートメソッド一覧** に進むと良いでしょう。)

unittest ユニットテストフレームワークは元々 JUnit に触発されたもので、他の言語の主要なユニットテストフレームワークと同じような感じです。テストの自動化、テスト用のセットアップやシャットダウンのコードの共有、テストのコレクション化、そして報告フレームワークからのテストの独立性をサポートしています。

これを実現するために、*unittest* はいくつかの重要な概念をオブジェクト指向の方法でサポートしています:

テストフィクスチャ (test fixture)　テストフィクスチャ (*test fixture*) とは、テスト実行のために必要な準備や終了処理を指します。例: テスト用データベースの作成・ディレクトリ・サーバプロセスの起動など。

テストケース (test case)　テストケース (*test case*) はテストの独立した単位で、各入力に対する結果をチェックします。テストケースを作成する場合は、*unittest* が提供する *TestCase* クラスを基底クラスとして利用することができます。

テストスイート (test suite)　テストスイート (*test suite*) はテストケースとテストスイートの集まりで、同時に実行しなければならないテストをまとめる場合に使用します。

テストランナー (test runner)　テストランナー (*test runner*) はテストの実行を管理し結果を提供する要素です。ランナーはグラフィカルインターフェースやテキストインターフェースを使用しても構いませんし、テストの実行結果を示す特別な値を返しても構いません。

参考:

doctest **モジュール**　テストをサポートするもうひとつのモジュールで、このモジュールとは趣きがだいぶ異なります。

Simple Smalltalk Testing: With Patterns　Kent Beck のテスティングフレームワークに関する原論文で、ここに記載されたパターンを *unittest* が使用しています。

'pytest '_　サードパーティのユニットテストフレームワークでより軽量な構文でテストを書くことができます。例えば、`assert func(10) == 42` のように書きます。

The Python Testing Tools Taxonomy　多くの Python のテストツールが一覧で紹介されています。ファンクショナルテストのフレームワークやモックライブラリも掲載されています。

Testing in Python **メーリングリスト**　Python でテストやテストツールについての議論に特化したグループです。

Python のソースコード配布物にあるスクリプト `Tools/unittestgui/unittestgui.py` はテストディスカバリとテスト実行のための GUI ツールです。主な目的は単体テストの初心者が簡単に使えるようにすることです。実際の生産環境では、'Buildbot '_ 、'Jenkins '_ 、'Travis-CI '_, 'AppVeyor '_ のような継続的インテグレーションシステムでテストを実行することを推奨します。

26.4.1 基本的な例

unittest モジュールには、テストの開発や実行の為の優れたツールが用意されており、この節では、その一部を紹介します。ほとんどのユーザとっては、ここで紹介するツールだけで十分でしょう。

以下は、三つの文字列メソッドをテストするスクリプトです:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')
```

(次のページに続く)

(前のページからの続き)

```
    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

テストケースは、*unittest.TestCase* のサブクラスとして作成します。メソッド名が test で始まる三つのメソッドがテストです。テストランナーはこの命名規約によってテストを行うメソッドを検索します。

これらのテスト内では、予定の結果が得られていることを確かめるために *assertEqual()* を、条件のチェックに *assertTrue()* や *assertFalse()* を、例外が発生する事を確認するために *assertRaises()* をそれぞれ呼び出しています。assert 文の代わりにこれらのメソッドを使用すると、テストランナーでテスト結果を集計してレポートを作成する事ができます。

setUp() および *tearDown()* メソッドによって各テストメソッドの前後に実行する命令を実装することが出来ます。詳細は **テストコードの構成** を参照してください。

最後のブロックは簡単なテストの実行方法を示しています。*unittest.main()* は、テストスクリプトのコマンドライン用インターフェースを提供します。コマンドラインから起動された場合、上記のスクリプトは以下のような結果を出力します:

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

-v オプションをテストスクリプトに渡すことで *unittest.main()* はより冗長になり、以下のような出力をします:

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

上の例が *unittest* モジュールで最もよく使われる機能で、ほとんどのテストではこれで十分です。以下では全ての機能を一から解説しています。

26.4.2 コマンドラインインターフェイス

ユニットテストモジュールはコマンドラインから使って、モジュール、クラス、あるいは個別のテストメソッドで定義されたテストを実行することが出来ます:

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

モジュール名ならびに完全修飾されたクラス名やメソッド名の任意の組み合わせを一覧で渡すことが出来ます。

テストモジュールはファイルパスで指定することも出来ます:

```
python -m unittest tests/test_something.py
```

そのため、テストモジュールを指定するのにシェルのファイル名補完が使えます。指定されたファイルはやはりモジュールとしてインポート可能でなければなりません。パスから '.py' を取り除き、パスセパレータを '.' に置き換えることでモジュール名に変換されます。モジュールとしてインポート可能でないテストファイルを実行したい場合は、代わりにそのファイルを直接実行するのが良いでしょう。

テスト実行時に (より冗長な) 詳細を表示するには -v フラグを渡します:

```
python -m unittest -v test_module
```

引数無しで実行すると テストディスカバリ が開始されます:

```
python -m unittest
```

コマンドラインプションの一覧を表示するには以下のコマンドを実行します:

```
python -m unittest -h
```

バージョン 3.2 で変更: 以前のバージョンでは、個々のテストメソッドしか実行することができず、モジュール単位やクラス単位で実行することは不可能でした。

コマンドラインオプション

unittest には以下のコマンドラインオプションがあります:

-b, --buffer
: 標準出力と標準エラーのストリームをテストの実行中にバッファします。テストが成功している間は結果の出力は破棄されます。テストの失敗やエラーの場合、出力は通常通り表示され、エラーメッセージに追加されます。

-c, --catch
: Control-C を実行中のテストが終了するまで遅延させ、そこまでの結果を出力します。二回目の Control-C は、通常通り *KeyboardInterrupt* の例外を発生させます。

この機能の仕組みについては、シグナルハンドリング を参照してください。

-f, --failfast
初回のエラーもしくは失敗の時にテストを停止します。

-k
Only run test methods and classes that match the pattern or substring. This option may be used multiple times, in which case all test cases that match of the given patterns are included.

ワイルドカード (*) を含むパターンは *fnmatch.fnmatchcase()* を使用してテスト名と照合され、それ以外の場合は単純な大文字と小文字を区別した部分文字列マッチングが使用されます。

パターンは、テストローダがインポートする時の完全修飾されたテストメソッド名と照合されます。

たとえば、`-k foo` は `foo_tests.SomeTest.test_something`, `bar_tests.SomeTest.test_foo` にマッチし、`bar_tests.FooTest.test_something` はマッチしません。

--locals
トレースバック内の局所変数を表示します。

バージョン 3.2 で追加: コマンドラインオプションの `-b`、`-c`、`-f` が追加されました。

バージョン 3.5 で追加: コマンドラインオプション `--locals`。

バージョン 3.7 で追加: コマンドラインオプション `-k`。

コマンドラインによってテストディスカバリ、すなわちプロジェクトの全テストを実行したりサブセットのみを実行したりすることも出来ます。

26.4.3 テストディスカバリ

バージョン 3.2 で追加.

unittest はシンプルなテストディスカバリをサポートします。テストディスカバリに対応するには、全テストファイルはプロジェクトの最上位のディスカバリからインポート可能な モジュール か (名前空間パッケージ を含む) パッケージ でなければなりません (つまりそれらのファイル名は有効な 識別子 でなければなりません)。

テストディスカバリは *TestLoader.discover()* で実装されていますが、コマンドラインから使う事も出来ます。その基本的な使い方は:

```
cd project_directory
python -m unittest discover
```

注釈: `python -m unittest` は `python -m unittest discover` と等価なショートカットです。テストディスカバリに引数を渡したい場合は、`discover` サブコマンドを明示的に使用しなければなりません。

`discover` サブコマンドには以下のオプションがあります:

-v, --verbose
詳細な出力

-s, --start-directory directory
ディスカバリを開始するディレクトリ (デフォルトは .)

-p, --pattern pattern
テストファイル名を識別するパターン (デフォルトは test*.py)

-t, --top-level-directory directory
プロジェクトの最上位のディスカバリのディレクトリ (デフォルトは開始のディレクトリ)

-s 、*-p* 、および *-t* オプションは、この順番であれば位置引数として渡す事ができます。以下の二つのコマンドは等価です:

```
python -m unittest discover -s project_directory -p "*_test.py"
python -m unittest discover project_directory "*_test.py"
```

パスと同様にパッケージ名を、例えば myproject.subpackage.test のように、開始ディレクトリとして渡すことができます。指定したパッケージ名はインポートされ、そのファイルシステム上の場所が開始ディレクトリとして使われます。

ご用心: テストディスカバリはインポートによりテストを読み込みます。一旦テストディスカバリが指定された開始ディレクトリから全テストファイルを見付けると、パスはインポートするパッケージ名に変換されます。例えば、foo/bar/baz.py は foo.bar.baz としてインポートされます。

グローバルにインストールされたパッケージがあり、それとは異なるコピーでディスカバリしようとしたとき、誤った場所からインポートが行われる **かもしれません**。その場合テストディスカバリは警告し、停止します。

ディレクトリのパスではなくパッケージ名を開始ディレクトリに指定した場合、ディスカバリはインポートするいずれの場所も意図した場所とするため、警告を受けないはずです。

テストモジュールとパッケージは、*load_tests* **プロトコル** によってテストのロードとディスカバリをカスタマイズすることができます。

バージョン 3.4 で変更: Test discovery supports *namespace packages* for start directory. Note that you need to the top level directory too. (e.g. python -m unittest discover -s root/namespace -t root).

26.4.4 テストコードの構成

ユニットテストの基本的な構成要素は、**テストケース** --- 設定され正しさのためにチェックされるべき単独のシナリオ --- です。*unittest* では、テストケースは *unittest.TestCase* クラスのインスタンスで表現されます。独自のテストケースを作成するには *TestCase* のサブクラスを記述するか、*FunctionTestCase* を使用しなければなりません。

TestCase インスタンスのテストコードは完全に独立していなければなりません。すなわち単独でか、他の様々なテストケースの任意の組み合わせのいずれかで実行可能でなければなりません。

最も単純な *TestCase* のサブクラスは、特定のテストコードを実行するためのテストメソッド (すなわち名前が `test` で始まるメソッド) を実装するだけで簡単に書くことができます:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def test_default_widget_size(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50))
```

何らかのテストを行うには、*TestCase* ベースクラスが提供する `assert*()` メソッドのうちの一つを使用してください。テストが失敗した場合は、例外が説明のメッセージとともに送出され、*unittest* はテスト結果を *failure* とします。その他の例外は *error* として扱われます。

テストは多くなり、それらの設定は繰り返しになるかもしれません。幸いにも、*setUp()* メソッドを実装することで設定コードをくくり出すことができます。テストフレームワークは実行するテストごとに自動的に *setUp()* を呼びます:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

注釈: いろいろなテストが実行される順序は、文字列の組み込みの順序でテストメソッド名をソートすることで決まります。

テスト中に *setUp()* メソッドで例外が発生した場合、フレームワークはそのテストに問題があるとみなし、そのテストメソッドは実行されません。

同様に、テストメソッド実行後に片付けをする *tearDown()* メソッドを提供出来ます:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
```

setUp() が成功した場合、テストメソッドが成功したかどうかに関わらず *tearDown()* が実行されます。

そのようなテストコードのための作業環境は **テストフィクスチャ** *(test fixture)* と呼ばれます。新しい TestCase インスタンスはある単一のテストフィクスチャとして作成され、個々のテストメソッドを実行するのに使われます。従って、*setUp()*, *tearDown()*„ `__init__()` は 1 回のテストにつき 1 回だけ呼び出されます。

テストケースの実装では、テストする機能に従ってテストをまとめるのをお勧めします。*unittest* はこのための機構、*unittest* の *TestSuite* クラスで表現される *test suite*、を提供します。たいていの場合 *unittest.main()* を呼び出しは正しい処理を行い、モジュールの全テストケースを集めて実行します。

しかし、テストスイートの構築をカスタマイズしたい場合、自分ですることができます:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_widget_size'))
    suite.addTest(WidgetTestCase('test_widget_resize'))
    return suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    runner.run(suite())
```

テストケースやテストコードの定義を (`widget.py` のような) テスト対象コードと同じモジュールに置くことが出来ますが、テストコードを (`test_widget.py` のような) 独立したモジュールに置くのには以下のような利点があります:

- テストモジュールだけをコマンドラインから独立に実行することができる。
- テストコードと出荷するコードをより簡単に分ける事ができる。
- 余程のことがない限り、テスト対象のコードに合わせてテストコードを変更することになりにくい。
- テストコードは、テスト対象コードほど頻繁に変更されない。
- テストコードをより簡単にリファクタリングすることができる。
- C で書いたモジュールのテストはどうせ独立したモジュールなのだから、同様にしない理由がない
- テストの方策を変更した場合でも、ソースコードを変更する必要がない。

26.4.5 既存テストコードの再利用

既存のテストコードが有るとき、このテストを *unittest* で実行しようとするために古いテスト関数をいちいち *TestCase* クラスのサブクラスに変換するのは大変です。

このような場合は、*unittest* では *TestCase* のサブクラスである *FunctionTestCase* クラスを使い、既存のテスト関数をラップします。初期設定と終了処理も行なえます。

以下のテストコードがあった場合:

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

オプションの set-up と tear-down メソッドを持った同等のテストケースインスタンスは次のように作成します:

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

注釈: *FunctionTestCase* を使って既存のテストを *unittest* ベースのテスト体系に変換することができますが、この方法は推奨されません。時間を掛けて *TestCase* のサブクラスに書き直した方が将来的なテストのリファクタリングが限りなく易しくなります。

既存のテストが *doctest* を使って書かれている場合もあるでしょう。その場合、*doctest* は DocTestSuite クラスを提供します。このクラスは、既存の *doctest* ベースのテストから、自動的に *unittest.TestSuite* のインスタンスを作成します。

26.4.6 テストのスキップと予期された失敗

バージョン 3.1 で追加.

unittest は特定のテストメソッドやテストクラス全体をスキップする仕組みを備えています。さらに、この機能はテスト結果を「予期された失敗 (expected failure)」とすることができ、テストが失敗しても *TestResult* の失敗数にはカウントされなくなります。

テストをスキップするには、*skip()* デコレータかその条件付きバージョンの一つを使うか、*setUp()* やテストメソッドの中で *TestCase.skipTest()* を呼び出すか、あるいは直接 *SkipTest* を送出するだけです。

基本的なスキップは以下のようになります:

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
```

(次のページに続く)

(前のページからの続き)

```
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass

    def test_maybe_skipped(self):
        if not external_resource_available():
            self.skipTest("external resource not available")
        # test code that depends on the external resource
        pass
```

このサンプルを冗長モードで実行すると以下のように出力されます:

```
test_format (__main__.MyTestCase) ... skipped 'not supported in this library version'
test_nothing (__main__.MyTestCase) ... skipped 'demonstrating skipping'
test_maybe_skipped (__main__.MyTestCase) ... skipped 'external resource not available'
test_windows_support (__main__.MyTestCase) ... skipped 'requires Windows'

----------------------------------------------------------------------
Ran 4 tests in 0.005s

OK (skipped=4)
```

テストクラスは以下のようにメソッドをスキップすることができます:

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

TestCase.setUp() もスキップすることができます。この機能はセットアップの対象のリソースが使用不可能な時に便利です。

予期された失敗の機能を使用するには *expectedFailure()* デコレータを使います。

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

独自のスキップ用のデコレータの作成は簡単です。そのためには、独自のデコレータのスキップしたい時点で *skip()* を呼び出します。以下のデコレータはオブジェクトに指定した属性が無い場合にテストをスキップし

ます:

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj, attr))
```

以下のデコレータと例外はテストのスキップと予期された失敗を実装しています:

@unittest.skip(*reason*)
デコレートしたテストを無条件でスキップします。*reason* にはテストをスキップした理由を記載します。

@unittest.skipIf(*condition*, *reason*)
condition が真の場合、デコレートしたテストをスキップします。

@unittest.skipUnless(*condition*, *reason*)
condition が偽の場合、デコレートしたテストをスキップします。

@unittest.expectedFailure
Mark the test as an expected failure or error. If the test fails or errors it will be considered a success. If the test passes, it will be considered a failure.

exception unittest.SkipTest(*reason*)
この例外はテストをスキップするために送出されます。

ふつうはこれを直接送出する代わりに *TestCase.skipTest()* やスキッピングデコレータの一つを使用出来ます。

スキップしたテストの前後では、*setUp()* および *tearDown()* は実行されません。同様に、スキップしたクラスの前後では、*setUpClass()* および *tearDownClass()* は実行されません。スキップしたモジュールの前後では、setUpModule() および tearDownModule() は実行されません。

26.4.7 サブテストを利用して繰り返しテストの区別を付ける

バージョン 3.4 で追加.

テストの間にとても小さな差異がある場合（例えばいくつかのパラメータなど）、unittest では *subTest()* コンテキストマネージャを使用してテストメソッドの内部でそれらを区別することができます。

例えば以下のテストは:

```
class NumbersTest(unittest.TestCase):

    def test_even(self):
        """
        Test that numbers between 0 and 5 are all even.
        """
        for i in range(0, 6):
```

(次のページに続く)

(前のページからの続き)

```
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

以下の出力をします:

```
======================================================================
FAIL: test_even (__main__.NumbersTest) (i=1)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest) (i=3)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest) (i=5)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

サブテスト無しの場合、最初の失敗で実行は停止し、`i` の値が表示されないためエラーの原因を突き止めるのは困難になります:

```
======================================================================
FAIL: test_even (__main__.NumbersTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

26.4.8 クラスと関数

この節では、*unittest* モジュールの API の詳細について説明します。

テストクラス

class unittest.TestCase(*methodName='runTest'*)

TestCase クラスのインスタンスは、*unittest* の世界における論理的なテストの単位を示します。このクラスをベースクラスとして使用し、必要なテストを具象サブクラスに実装します。*TestCase* クラスでは、テストランナーがテストを実行するため のインターフェースと、各種の失敗をチェックしレポートするためのメソッドを実装しています。

TestCase の各インスタンスは *methodName* という名前の単一の基底メソッドを実行します。*TestCase* を使用する大半の場合 *methodName* を変更したりデフォルトの `runTest()` メソッドを再実装することはありません。

バージョン 3.2 で変更: *TestCase* が *methodName* を指定しなくてもインスタンス化できるようになりました。これにより対話的インタプリタから *TestCase* を簡単に試せるようになりました。

TestCase のインスタンスのメソッドは 3 種類のグループを提供します。1 つ目のグループはテストの実行で使用されます。2 つ目のグループは条件のチェックおよび失敗のレポートを行うテストの実装で使用されます。3 つ目のグループである問い合わせ用のメソッドによってテスト自身の情報が収集されます。

はじめのグループ (テスト実行) に含まれるメソッドは以下の通りです:

setUp()

テストフィクスチャの準備のために呼び出されるメソッドです。テストメソッドの直前に呼び出されます。このメソッドで *AssertionError* や *SkipTest* 以外の例外が発生した場合、テストの失敗ではなくエラーとされます。デフォルトの実装では何も行いません。

tearDown()

テストメソッドが実行され、結果が記録された直後に呼び出されるメソッドです。このメソッドはテストメソッドで例外が投げられても呼び出されます。そのため、サブクラスでこのメソッドを実装する場合は、内部状態を確認することが必要になるでしょう。このメソッドで *AssertionError* や *SkipTest* 以外の例外が発生した場合、テストの失敗とは別のエラーとみなされます (従って報告されるエラーの総数は増えます)。このメソッドは、テストの結果に関わらず *setUp()* が成功した場合にのみ呼ばれます。デフォルトの実装では何も行いません。

setUpClass()

個別のクラス内のテストが実行される前に呼び出されるクラスメソッドです。`setUpClass` はクラスを唯一の引数として取り、*classmethod()* でデコレートされていなければなりません:

```
@classmethod
def setUpClass(cls):
    ...
```

詳しくは クラスとモジュールのフィクスチャ を参照してください。

バージョン 3.2 で追加.

tearDownClass()

個別のクラス内のテストが実行された後に呼び出されるクラスメソッドです。tearDownClass はクラスを唯一の引数として取り、*classmethod()* でデコレーションされていなければなりません:

```
@classmethod
def tearDownClass(cls):
    ...
```

詳しくは **クラスとモジュールのフィクスチャ** を参照してください。

バージョン 3.2 で追加.

run(*result=None*)

テストを実行し、テスト結果を *result* に指定された *TestResult* オブジェクトにまとめます。*result* が省略されるか None が渡された場合、(*defaultTestResult()* メソッドを呼んで) 一時的な結果オブジェクトを生成し、使用します。結果オブジェクトは *run()* の呼び出し元に返されます。

このメソッドは、単に *TestCase* インスタンスを呼び出した場合と同様に振る舞います。

バージョン 3.3 で変更: 以前のバージョンの run は結果オブジェクトを返しませんでした。また *TestCase* インスタンスを呼び出した場合も同様でした。

skipTest(*reason*)

テストメソッドや *setUp()* が現在のテストをスキップする間に呼ばれます。詳細については、**テストのスキップと予期された失敗** を参照してください。

バージョン 3.1 で追加.

subTest(*msg=None*, ***params*)

このメソッドを囲っているブロックをサブテストとして実行するコンテキストマネージャを返します。*msg* と *params* はサブテストが失敗したときに表示されるオプションの任意の値で、どんな値が使われたかを明確にするものです。

テストケースには subtest 宣言を幾らでも含めることができ、任意にネストすることができます。

詳細は **サブテストを利用して繰り返しテストの区別を付ける** を参照してください。

バージョン 3.4 で追加.

debug()

テスト結果を収集せずにテストを実行します。例外が呼び出し元に通知されます。また、テストをデバッガで実行することができます。

TestCase クラスは失敗の検査と報告を行う多くのメソッドを提供しています。以下の表は最も一般的に使われるメソッドを列挙しています (より多くのアサートメソッドについては表の下を見てください):

メソッド	確認事項	初出
assertEqual(a, b)	`a == b`	
assertNotEqual(a, b)	`a != b`	
assertTrue(x)	`bool(x) is True`	
assertFalse(x)	`bool(x) is False`	
assertIs(a, b)	`a is b`	3.1
assertIsNot(a, b)	`a is not b`	3.1
assertIsNone(x)	`x is None`	3.1
assertIsNotNone(x)	`x is not None`	3.1
assertIn(a, b)	`a in b`	3.1
assertNotIn(a, b)	`a not in b`	3.1
assertIsInstance(a, b)	`isinstance(a, b)`	3.2
assertNotIsInstance(a, b)	`not isinstance(a, b)`	3.2

全てのアサートメソッドは *msg* 引数を受け取り、指定された場合、失敗時のエラーメッセージとして使われます。(*longMessage* も参照してください)。*msg* キーワード引数は *assertRaises()* 、*assertRaisesRegex()* 、*assertWarns()* 、*assertWarnsRegex()* には、そのメソッドをコンテキストマネージャとして使った場合にのみ使えます。

assertEqual(*first*, *second*, *msg=None*)

first と *second* が等しいことをテストします。両者が等しくない場合、テストは失敗です。

さらに、*first* と *second* が厳密に同じ型であり、list、tuple、dict、set、frozenset もしくは str のいずれか、またはサブクラスが *addTypeEqualityFunc()* に登録されている任意の型の場合、より有用なデフォルトのエラーメッセージを生成するために、その型特有の比較関数が呼ばれます (**型固有のメソッドの一覧** も参照してください)。

バージョン 3.1 で変更: 自動で型固有の比較関数が呼ばれるようになりました。

バージョン 3.2 で変更: 文字列比較のデフォルトの比較関数として *assertMultiLineEqual()* が追加されました。

assertNotEqual(*first*, *second*, *msg=None*)

first と *second* が等しくないことをテストします。両者が等しい場合、テストは失敗です。

assertTrue(*expr*, *msg=None*)

assertFalse(*expr*, *msg=None*)

expr が真 (偽) であることをテストします。

このメソッドは、`bool(expr) is True` と等価であり、`expr is True` と等価ではないことに注意が必要です (後者のためには、`assertIs(expr, True)` が用意されています)。また、専用のメソッドが使用できる場合には、そちらを使用してください (例えば `assertTrue(a == b)` の代わりに `assertEqual(a, b)` を使用してください)。そうすることにより、テスト失敗時のエラーメッセージを詳細に表示することができます。

assertIs(*first*, *second*, *msg=None*)

assertIsNot(*first*, *second*, *msg=None*)

first と *second* が同じオブジェクトであること (またはそうでないこと) をテストします。

バージョン 3.1 で追加.

assertIsNone(*expr*, *msg=None*)

assertIsNotNone(*expr*, *msg=None*)

expr が `None` であること (および、そうでないこと) をテストします。

バージョン 3.1 で追加.

assertIn(*member*, *container*, *msg=None*)

assertNotIn(*member*, *container*, *msg=None*)

member が *container* に含まれること (またはそうでないこと) をテストします。

バージョン 3.1 で追加.

assertIsInstance(*obj*, *cls*, *msg=None*)

assertNotIsInstance(*obj*, *cls*, *msg=None*)

obj が *cls* のインスタンスであること (あるいはそうでないこと) をテストします (この *cls* は、*isinstance()* が扱うことのできる、クラスもしくはクラスのタプルである必要があります)。正確な型をチェックするためには、*assertIs(type(obj), cls)* を使用してください。

バージョン 3.2 で追加.

以下のメソッドを使用して例外、警告、およびログメッセージの発生を確認することが出来ます:

メソッド	確認事項	初出
*assertRaises(exc, fun, *args, **kwds)*	`fun(*args, **kwds)` が *exc* を送出する	
*assertRaisesRegex(exc, r, fun, *args, **kwds)*	`fun(*args, **kwds)` が *exc* を送出してメッセージが正規表現 *r* とマッチする	3.1
*assertWarns(warn, fun, *args, **kwds)*	`fun(*args, **kwds)` が *warn* を送出する	3.2
*assertWarnsRegex(warn, r, fun, *args, **kwds)*	`fun(*args, **kwds)` が *warn* を送出してメッセージが正規表現 *r* とマッチする	3.2
assertLogs(logger, level)	`with` ブロックが 最低 *level* で *logger* を使用する	3.4

assertRaises(*exception*, *callable*, **args*, ***kwds*)

assertRaises(*exception*, ***, *msg=None*)

callable を呼び出した時に例外が発生することをテストします。*assertRaises()* で指定した位置パラメータとキーワードパラメータを該当メソッドに渡します。*exception* が送出された場合、テストは成功です。また、他の例外が投げられた場合はエラー、例外が送出されなかった場合は失敗になります。複数の例外をキャッチする場合には、例外クラスのタプルを *exception* に指定してください。

exception 引数のみ（またはそれに加えて *msg* 引数）が渡された場合には、コンテキストマネージャが返されます。これにより関数名を渡す形式ではなく、インラインでテスト対象のコードを書くことができます:

```
with self.assertRaises(SomeException):
    do_something()
```

コンテキストマネージャとして使われたときは、*assertRaises()* は加えて *msg* キーワード引数も受け付けます。

このコンテキストマネージャは exception で指定されたオブジェクトを格納します。これにより、例外発生時の詳細な確認をおこなうことができます:

```
with self.assertRaises(SomeException) as cm:
    do_something()

the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

バージョン 3.1 で変更: *assertRaises()* がコンテキストマネージャとして使えるようになりました。

バージョン 3.2 で変更: exception 属性が追加されました。

バージョン 3.3 で変更: コンテキストマネージャとして使用したときに *msg* キーワード引数が追加されました。

assertRaisesRegex(*exception*, *regex*, *callable*, **args*, ***kwds*)

assertRaisesRegex(*exception*, *regex*, ***, *msg=None*)

assertRaises() と同等ですが、例外の文字列表現が *regex* にマッチすることもテストします。*regex* は正規表現オブジェクトか、*re.search()* が扱える正規表現が書かれた文字列である必要があります。例えば以下のようになります:

```
self.assertRaisesRegex(ValueError, "invalid literal for.*XYZ'$",
                       int, 'XYZ')
```

もしくは:

```
with self.assertRaisesRegex(ValueError, 'literal'):
   int('XYZ')
```

バージョン 3.1 で追加: “assertRaisesRegexp“という名前で追加されました。

バージョン 3.2 で変更: *assertRaisesRegex()* にリネームされました。

バージョン 3.3 で変更: コンテキストマネージャとして使用したときに *msg* キーワード引数が追加されました。

assertWarns(*warning*, *callable*, **args*, ***kwds*)

assertWarns(*warning*, *, *msg=None*)

callable を呼び出した時に警告が発生することをテストします。*assertWarns()* で指定した位置パラメータとキーワードパラメータを該当メソッドに渡します。*warning* が発生した場合にテストが成功し、そうでなければ失敗になります。例外が送出された場合はエラーになります。複数の警告を捕捉する場合には、警告クラスのタプルを *warnings* に指定してください。

warning 引数のみ（またはそれに加えて *msg* 引数）が渡された場合には、コンテキストマネージャが返されます。これにより関数名を渡す形式ではなく、インラインでテスト対象のコードを書くことができます:

```
with self.assertWarns(SomeWarning):
    do_something()
```

コンテキストマネージャとして使われたときは、*assertWarns()* は加えて *msg* キーワード引数も受け付けます。

このコンテキストマネージャは、捕捉した警告オブジェクトを `warning` 属性に、警告が発生したソース行を `filename` 属性と `lineno` 属性に格納します。これは警告発生時に捕捉された警告に対して追加の確認を行いたい場合に便利です:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()

self.assertIn('myfile.py', cm.filename)
self.assertEqual(320, cm.lineno)
```

このメソッドは呼び出されたときに警告フィルタを無視して動作します。

バージョン 3.2 で追加.

バージョン 3.3 で変更: コンテキストマネージャとして使用したときに *msg* キーワード引数が追加されました。

assertWarnsRegex(*warning*, *regex*, *callable*, **args*, ***kwds*)

assertWarnsRegex(*warning*, *regex*, *, *msg=None*)

assertWarns() と同等ですが、警告メッセージが *regex* にマッチすることもテストします。*regex* は正規表現オブジェクトか、*re.search()* が扱える正規表現が書かれた文字列である必要があります。例えば以下のようになります:

```
self.assertWarnsRegex(DeprecationWarning,
                      r'legacy_function\(\) is deprecated',
                      legacy_function, 'XYZ')
```

もしくは:

```
with self.assertWarnsRegex(RuntimeWarning, 'unsafe frobnicating'):
    frobnicate('/etc/passwd')
```

バージョン 3.2 で追加.

バージョン 3.3 で変更: コンテキストマネージャとして使用したときに *msg* キーワード引数が追加されました。

assertLogs(*logger=None*, *level=None*)

logger かその子ロガーのうちの 1 つに、少なくとも 1 つのログメッセージが少なくとも与えられた *level* で出力されることをテストするコンテキストマネージャです。

If given, *logger* should be a *logging.Logger* object or a *str* giving the name of a logger. The default is the root logger, which will catch all messages that were not blocked by a non-propagating descendent logger.

level が与えられた場合、ログレベルを表す数値もしくはそれに相当する文字列 (例えば `"ERROR"` もしくは `logging.ERROR`) であるべきです。デフォルトは `logging.INFO` です。

`with` ブロック内で出たメッセージの少なくとも一つが *logger* および *level* 条件に合っている場合、このテストをパスします。それ以外の場合は失敗です。

コンテキストマネージャから返されるオブジェクトは、条件に該当するログメッセージを追跡し続ける記録のためのヘルパーです。このオブジェクトには 2 つの属性があります:

records
該当するログメッセージを表す *logging.LogRecord* オブジェクトのリスト。

output
該当するメッセージ出力をフォーマットした *str* オブジェクトのリスト。

以下はプログラム例です:

```
with self.assertLogs('foo', level='INFO') as cm:
   logging.getLogger('foo').info('first message')
   logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

バージョン 3.4 で追加.

より具体的な確認を行うために以下のメソッドが用意されています:

メソッド	確認事項	初出
assertAlmostEqual(a, b)	`round(a-b, 7) == 0`	
assertNotAlmostEqual(a, b)	`round(a-b, 7) != 0`	
assertGreater(a, b)	`a > b`	3.1
assertGreaterEqual(a, b)	`a >= b`	3.1
assertLess(a, b)	`a < b`	3.1
assertLessEqual(a, b)	`a <= b`	3.1
assertRegex(s, r)	`r.search(s)`	3.1
assertNotRegex(s, r)	`not r.search(s)`	3.2
assertCountEqual(a, b)	*a* と *b* に、順番によらず同じ要素が同じ数だけある。	3.2

assertAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)
assertNotAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)
first と *second* が近似的に等しい (等しくない) ことをテストします。これは、*places* (デフォルト 7) で指定した小数位で丸めた差分をゼロと比較することで行われます。これらのメソッドは (*round()* と同様に) **小数位** を指定するのであって、**有効桁数** を指定するのではないことに注意してください。

places の代わりに *delta* が渡された場合には、*first* と *second* の差分が *delta* 以下 (以上) であることをテストします。

delta と *places* の両方が指定された場合は *TypeError* が送出されます。

バージョン 3.2 で変更: *assertAlmostEqual()* は、オブジェクトが等しい場合には自動で近似的に等しいとみなすようになりました。*assertNotAlmostEqual()* は、オブジェクトが等しい場合には自動的に失敗するようになりました。*delta* 引数が追加されました。

assertGreater(*first*, *second*, *msg=None*)
assertGreaterEqual(*first*, *second*, *msg=None*)
assertLess(*first*, *second*, *msg=None*)
assertLessEqual(*first*, *second*, *msg=None*)
first が *second* と比べて、メソッド名に対応して >, >=, < もしくは <= であることをテストします。そうでない場合はテストは失敗です:

```
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or equal to "4"
```

バージョン 3.1 で追加.

assertRegex(*text*, *regex*, *msg=None*)
assertNotRegex(*text*, *regex*, *msg=None*)
regex の検索が *text* とマッチする (またはマッチしない) ことをテストします。テスト失敗時には、エラーメッセージにパターンと *text* が表示されます（もしくは、パターンと意図しないかたちでマッチした *text* の一部が表示されます)。*regex* は正規表現オブジェクトか、*re.search()* が扱える正規表現が書かれた文字列である必要があります。

バージョン 3.1 で追加: “assertRegexpMatches“という名前で追加されました。

バージョン 3.2 で変更: メソッド `assertRegexpMatches()` は *assertRegex()* にリネームされました。

バージョン 3.2 で追加: *assertNotRegex()*。

バージョン 3.5 で追加: `assertNotRegexpMatches` は *assertNotRegex()* のエイリアスであることから非推奨となります。

assertCountEqual(*first*, *second*, *msg=None*)
シーケンス *first* が *second* と同じ要素を含んでいることをテストします。要素の順序はテスト結果に影響しません。要素が含まれていない場合には、シーケンスの差分がエラーメッセージとして表示されます。

first と *second* の比較では、重複した要素は無視 **されません**。両者に同じ数の要素が含まれていることを検証します。このメソッドは `assertEqual(Counter(list(first)), Counter(list(second)))` と同等に振る舞うことに加えて、ハッシュ化できないオブジェクトのシーケンスでも動作します。

バージョン 3.2 で追加.

assertEqual() メソッドは、同じ型のオブジェクトの等価性確認のために、型ごとに特有のメソッドにディスパッチします。これらのメソッドは、ほとんどの組み込み型用のメソッドは既に実装されています。さらに、*addTypeEqualityFunc()* を使う事で新たなメソッドを登録することができます:

addTypeEqualityFunc(*typeobj*, *function*)

assertEqual() で呼び出される型特有のメソッドを登録します。登録するメソッドは、比較する 2 つのオブジェクトの型が厳密に *typeobj* と同じ (サブクラスでもいけません) の場合に等価性を確認します。*function* は *assertEqual()* と同様に、2 つの位置引数と、3 番目に msg=None のキーワード引数を取れる必要があります。このメソッドは、始めの 2 つに指定したパラメータ間の差分を検出した時に *self.failureException(msg)* の例外を投げる必要があります。この例外を投げる際は、出来る限り、エラーの内容が分かる有用な情報と差分の詳細をエラーメッセージに含めてください。

バージョン 3.1 で追加.

assertEqual() が自動的に呼び出す型特有のメソッドの概要を以下の表示に記載しています。これらのメソッドは通常は直接呼び出す必要がないことに注意が必要です。

メソッド	比較の対象	初出
assertMultiLineEqual(a, b)	文字列	3.1
assertSequenceEqual(a, b)	シーケンス	3.1
assertListEqual(a, b)	リスト	3.1
assertTupleEqual(a, b)	タプル	3.1
assertSetEqual(a, b)	set または frozenset	3.1
assertDictEqual(a, b)	辞書	3.1

assertMultiLineEqual(*first*, *second*, *msg=None*)

複数行の文字列 *first* が文字列 *second* と等しいことをテストします。等しくない場合には、両者の差分がハイライトされてエラーメッセージに表示されます。このメソッドは、デフォルトで、*assertEqual()* が string を比較するときに自動的に使用します。

バージョン 3.1 で追加.

assertSequenceEqual(*first*, *second*, *msg=None*, *seq_type=None*)

2 つのシーケンスが等しいことをテストします。*seq_type* が指定された場合、*first* と *second* が *seq_type* のインスタンスで無い場合にはテストが失敗します。シーケンスどうしが異なる場合には、両者の差分がエラーメッセージに表示されます。

このメソッドは直接 *assertEqual()* からは呼ばれませんが、*assertListEqual()* と *assertTupleEqual()* の実装で使われています。

バージョン 3.1 で追加.

assertListEqual(*first*, *second*, *msg=None*)
assertTupleEqual(*first*, *second*, *msg=None*)
2 つのリストまたはタプルが等しいかどうかをテストします。等しくない場合には、両者の差分を表示します。2 つのパラメータの型が異なる場合にはテストがエラーになります。このメソッドは、デフォルトで、*assertEqual()* が list または tuple を比較するときに自動的に使用します。

バージョン 3.1 で追加.

assertSetEqual(*first*, *second*, *msg=None*)
2 つのセットが等しいかどうかをテストします。等しくない場合には、両者の差分を表示します。このメソッドは、デフォルトで、*assertEqual()* が set もしくは frozenset を比較するときに自動的に使用します。

first or *second* のいずれかに `set.difference()` が無い場合にはテストは失敗します。

バージョン 3.1 で追加.

assertDictEqual(*first*, *second*, *msg=None*)
2 つの辞書が等しいかどうかをテストします。等しくない場合には、両者の差分を表示します。このメソッドは、デフォルトで、*assertEqual()* が dict を比較するときに自動的に使用します。

バージョン 3.1 で追加.

最後に、*TestCase* の残りのメソッドと属性を紹介します:

fail(*msg=None*)
無条件にテストを失敗させます。エラーメッセージの表示に、*msg* または `None` が使われます。

failureException
test() メソッドが送出する例外を指定するクラス属性です。例えばテストフレームワークで追加情報を付した特殊な例外が必要になる場合、この例外のサブクラスとして作成します。この属性の初期値は *AssertionError* です。

longMessage
このクラス属性は、失敗した assertXYY の呼び出しで独自の失敗時のメッセージが msg 引数として渡されていたときにどうするかを決定します。`True` がデフォルト値です。この場合、標準の失敗時のメッセージの後に独自のメッセージが追記されます。`False` に設定したときは、標準のメッセージを独自のメッセージで置き換えます。

アサートメソッドを呼び出す前に、個別のテストメソッドの中でインスタンス属性 self.longMessage を `True` または `False` に設定して、この設定を上書きできます。

このクラスの設定はそれぞれのテストを呼び出す前にリセットされます。

バージョン 3.1 で追加.

maxDiff
この属性は、アサーションメソッドが失敗をレポートする時に表示する差分の長さをコントロールします。デフォルトは 80*8 文字です。この属性が影響するメソッド

は、*assertSequenceEqual()* (およびこのメソッドに委譲するシーケンス比較メソッド)、*assertDictEqual()* と *assertMultiLineEqual()* です。

maxDiff を None に設定すると差分表示の上限がなくなります。

バージョン 3.2 で追加.

テストフレームワークは、テスト情報を収集するために以下のメソッドを使用します:

countTestCases()
テストオブジェクトに含まれるテストの数を返します。*TestCase* インスタンスは常に 1 を返します。

defaultTestResult()
このテストケースクラスで使われるテスト結果クラスのインスタンスを (もし *run()* メソッドに他の結果インスタンスが提供されないならば) 返します。

TestCase インスタンスに対しては、いつも *TestResult* のインスタンスですので、*TestCase* のサブクラスでは必要に応じてこのメソッドをオーバライドしてください。

id()
テストケースを特定する文字列を返します。通常、*id* はモジュール名・クラス名を含む、テストメソッドのフルネームを指定します。

shortDescription()
テストの説明を一行分、または説明がない場合には None を返します。デフォルトでは、テストメソッドの docstring の先頭の一行、または None を返します。

バージョン 3.1 で変更: 3.1 で docstring があったとしても、返される短い説明文字列にテスト名が付けられるようになりました。この変更によって unittest 拡張に互換性の問題が発生し、Python 3.2 でテスト名が追加される場所は *TextTestResult* へ移動しました。

addCleanup(*function*, **args*, ***kwargs*)
tearDown() の後に呼び出される関数を追加します。この関数はリソースのクリーンアップのために使用します。追加された関数は、追加された順と逆の順番で呼び出されます（LIFO)。*addCleanup()* に渡された引数とキーワード引数が追加された関数にも渡されます。

setUp() が失敗した場合、つまり *tearDown()* が呼ばれなかった場合でも、追加されたクリーンアップ関数は呼び出されます。

バージョン 3.1 で追加.

doCleanups()
このメソッドは、*tearDown()* の後、もしくは、*setUp()* が例外を投げた場合は *setUp()* の後に、無条件で呼ばれます。

このメソッドは、*addCleanup()* で追加された関数を呼び出す責務を担います。もし、クリーンアップ関数を *tearDown()* より前に呼び出す必要がある場合には、*doCleanups()* を明示的に呼び出してください。

doCleanups() は、どこで呼び出されても、クリーンアップ関数をスタックから削除して実行します。

バージョン 3.1 で追加.

classmethod addClassCleanup(*function*, */*, **args*, ***kwargs*)

tearDownClass() の後に呼び出される関数を追加します。この関数はリソースのクリーンアップのために使用します。追加された関数は、追加された順と逆の順番で呼び出されます（LIFO）。*addClassCleanup()* に渡された引数とキーワード引数が追加された関数にも渡されます。

setUpClass() が失敗した場合、つまり *tearDownClass()* が呼ばれなかった場合でも、追加されたクリーンアップ関数は呼び出されます。

バージョン 3.8 で追加.

classmethod doClassCleanups()

このメソッドは、*tearDownClass()* の後、もしくは、*setUpClass()* が例外を投げた場合は *setUpClass()* の後に、無条件で呼ばれます。

このメソッドは、*addClassCleanup()* で追加された関数を呼び出す責務を担います。もし、クリーンアップ関数を *tearDownClass()* より前に呼び出す必要がある場合には、*doClassCleanups()* を明示的に呼び出してください。

doClassCleanups() は、どこで呼び出されても、クリーンアップ関数をスタックから削除して実行します。

バージョン 3.8 で追加.

class unittest.IsolatedAsyncioTestCase(*methodName='runTest'*)

このクラスは:class:*TestCase* と似た API を提供し、テスト関数としてコルーチンも許容します。

バージョン 3.8 で追加.

coroutine asyncSetUp()

テストフィクスチャを用意するために呼び出されるメソッドです。これは `setUp()`. の後に呼び出されます。これはテストメソッドを呼び出す直前に呼び出されます。*AssertionError* と *SkipTest* を除いて、このメソッドのによって送出されたあらゆる例外はテストの失敗ではなくエラーとみなされます。デフォルトの実装では何もしません。

coroutine asyncTearDown()

テストメソッドが呼び出され、その結果が記録された直後に呼び出されるメソッドです。これは `tearDown`の前に呼び出されます。これはテストメソッドが例外を送出した場合でも呼び出されるので、サブクラスの実装では内部状態のチェックに特に気を付ける必要があります。このメソッドで送出された:exc:`AssertionError() と SkipTest`以外の例外は、テストの失敗ではなく追加のエラーとみなされます（そのため、報告されるエラーの総数が増えることになります）。このメソッドはテストメソッドの結果に関係なく、:meth:`asyncSetUp が成功した場合にのみ呼び出されます。デフォルトの実装では何もしません。

addAsyncCleanup(*function*, */*, **args*, ***kwargs*)

このメソッドはクリーンアップ関数として使用できるコルーチンを受け入れます。

run(*result=None*)

テストを実行するための新しいイベントループを作成し、*result*として渡された:class:‘TestResult‘オブジェクトに結果を収集します。*result*が省略された場合や “None“の場合は、一時的な result オブジェクトが (:meth:‘defaultTestResult‘メソッドの呼び出しによって) 作成され、使用されます。この result オブジェクトは :meth:‘run‘の呼び出し元に返されます。テスト終了時には、イベントループ内のすべてのタスクがキャンセルされます。

順番を示す例です:

```
from unittest import IsolatedAsyncioTestCase

events = []


class Test(IsolatedAsyncioTestCase):


    def setUp(self):
        events.append("setUp")

    async def asyncSetUp(self):
        self._async_connection = await AsyncConnection()
        events.append("asyncSetUp")

    async def test_response(self):
        events.append("test_response")
        response = await self._async_connection.get("https://example.com")
        self.assertEqual(response.status_code, 200)
        self.addAsyncCleanup(self.on_cleanup)

    def tearDown(self):
        events.append("tearDown")

    async def asyncTearDown(self):
        await self._async_connection.close()
        events.append("asyncTearDown")

    async def on_cleanup(self):
        events.append("cleanup")

if __name__ == "__main__":
    unittest.main()
```

テスト実行後、“events“には “[”setUp”, ”asyncSetUp”, ”test_response”, ”asyncTearDown”, ”tearDown”, ”cleanup”]“が含まれます。

class unittest.FunctionTestCase(*testFunc*, *setUp=None*, *tearDown=None*, *description=None*)

このクラスでは *TestCase* インターフェースの内、テストランナーがテストを実行するためのインターフェースだけを実装しており、テスト結果のチェックやレポートに関するメソッドは実装していません。既存のテストコードを *unittest* によるテストフレームワークに組み込むために使用します。

非推奨のエイリアス

歴史的な経緯で、*TestCase* のいくつかのエイリアスは非推奨となりました。以下の表に、非推奨のエイリアスをまとめます:

メソッド名	非推奨のエイリアス	非推奨のエイリアス
assertEqual()	failUnlessEqual	assertEquals
assertNotEqual()	failIfEqual	assertNotEquals
assertTrue()	failUnless	assert_
assertFalse()	failIf	
assertRaises()	failUnlessRaises	
assertAlmostEqual()	failUnlessAlmostEqual	assertAlmostEquals
assertNotAlmostEqual()	failIfAlmostEqual	assertNotAlmostEquals
assertRegex()		assertRegexpMatches
assertNotRegex()		assertNotRegexpMatches
assertRaisesRegex()		assertRaisesRegexp

バージョン 3.1 で非推奨: 2 列目に記載されている fail* エイリアスは非推奨になりました。

バージョン 3.2 で非推奨: 3 列目に記載されている assert* エイリアスは非推奨になりました。

バージョン 3.2 で非推奨: `assertRegexpMatches` は *assertRegex()* に、`assertRaisesRegexp` は *assertRaisesRegex()* にメソッド名が変更されました

バージョン 3.5 で非推奨: “assertNotRegexpMatches“という名前は非推奨となり、:meth:‘.assertNotRegex‘になりました。

テストのグループ化

class unittest.TestSuite(*tests=()*)

このクラスは、個々のテストケースやテストスイートの集合を表現しています。通常のテストケースと同じようにテストランナーで実行するためのインタフェースを備えています。*TestSuite* インスタンスを実行することはスイートをイテレートして得られる個々のテストを実行することと同じです。

引数 *tests* が指定された場合、それはテストケースに亘る繰り返し可能オブジェクトまたは内部でスイートを組み立てるための他のテストスイートでなければなりません。後からテストケースやスイートをコレクションに付け加えるためのメソッドも提供されています。

TestSuite は *TestCase* オブジェクトのように振る舞います。違いは、スイートにはテストを実装しない点にあります。代わりに、テストをまとめてグループ化して、同時に実行します。*TestSuite* のインスタンスにテスト追加するためのメソッドが用意されています:

addTest(*test*)

TestCase 又は *TestSuite* のインスタンスをスイートに追加します。

addTests(*tests*)

イテラブル *tests* に含まれる全ての *TestCase* 又は *TestSuite* のインスタンスをスイートに追加します。

このメソッドは *tests* 上のイテレーションをしながらそれぞれの要素に *addTest()* を呼び出すのと等価です。

TestSuite クラスは *TestCase* と以下のメソッドを共有します:

run(*result*)

スイート内のテストを実行し、結果を *result* で指定した結果オブジェクトに収集します。*TestCase.run()* と異なり、*TestSuite.run()* では必ず結果オブジェクトを指定する必要があります。

debug()

このスイートに関連づけられたテストを結果を収集せずに実行します。これによりテストで送出された例外は呼び出し元に伝わるようになり、デバッガの下でのテスト実行をサポートできるようになります。

countTestCases()

このテストオブジェクトによって表現されるテストの数を返します。これには個別のテストと下位のスイートも含まれます。

__iter__()

TestSuite でグループ化されたテストは反復アクセスできます。サブクラスは *__iter__()* をオーバーライドすることで、遅延処理でテストを提供できます。1 つのスイート内でこのメソッドは何度も呼ばれる可能性があることに注意してください (例えば、テスト数のカウントや等価性の比較)。そのため、*TestSuite.run()* を実行する前に反復アクセスを何度繰り返しても同じテスト群を返すようにしなければなりません。呼び出し側が TestSuite._removeTestAtIndex() をオーバーライドしたサブクラスを使いテストへの参照を保存していない限り、*TestSuite.run()* を実行した後はこのメソッドが返すテスト群を信頼すべきではありません。

バージョン 3.2 で変更: 以前のバージョンでは *TestSuite* はイテレータではなく、直接テストにアクセスしていました。そのため、*__iter__()* をオーバーラードしてもテストにアクセスできませんでした。

バージョン 3.4 で変更: 以前のバージョンでは、*TestSuite.run()* の実行後は *TestSuite* が各 *TestCase* への参照を保持していました。サブクラスで TestSuite._removeTestAtIndex() をオーバーライドすることでこの振る舞いを復元できます。

通常、*TestSuite* の *run()* メソッドは TestRunner が起動するため、ユーザが直接実行する必要はありません。

テストのロードと起動

class unittest.TestLoader

TestLoader クラスはクラスとモジュールからテストスイートを生成します。通常、このクラスのインスタンスを明示的に生成する必要はありません。*unittest* モジュールの *unittest.defaultTestLoader* を共用インスタンスとして使用することができます。しかし、このクラスのサブクラスやインスタンスで、属性をカスタマイズすることができます。

TestLoader オブジェクトには以下の属性があります:

errors

テストの読み込み中に起きた致命的でないエラーのリストです。どの時点でもローダーからリセットされることはありません。致命的なエラーは適切なメソッドが例外を送出して、呼び出し元に通知します。致命的でないエラーも、実行したときのエラーを総合テストが通知してくれます。

バージョン 3.5 で追加.

TestLoader のオブジェクトには以下のメソッドがあります:

loadTestsFromTestCase(*testCaseClass*)

TestCase の派生クラス testCaseClass に含まれる全テストケースのスイートを返します。

getTestCaseNames() で指定されたメソッドに対し、テストケースインスタンスが作成されます。デフォルトでは test で始まる名前のメソッド群です。*getTestCaseNames()* がメソッド名を返さなかったが、runTest() メソッドが実装されている場合は、そのメソッドに対するテストケースが代わりに作成されます。

loadTestsFromModule(*module*, *pattern=None*)

指定したモジュールに含まれる全テストケースのスイートを返します。このメソッドは *module* 内の *TestCase* 派生クラスを検索し、見つかったクラスのテストメソッドごとにクラスのインスタンスを作成します。

注釈: *TestCase* クラスを基底クラスとしてクラス階層を構築するとテストフィクスチャや補助的な関数をうまく共用することができますが、基底クラスに直接インスタンス化できないテストメソッドがあると、この loadTestsFromModule() を使うことができません。この場合でも、fixture が全て別々で定義がサブクラスにある場合は使用することができます。

モジュールが load_tests 関数を用意している場合、この関数がテストの読み込みに使われます。これによりテストの読み込み処理がカスタマイズできます。これが *load_tests* **プロトコル** です。*pattern* 引数は load_tests に第 3 引数として渡されます。

バージョン 3.2 で変更: load_tests のサポートが追加されました。

バージョン 3.5 で変更: ドキュメントにない、非公式の *use_load_tests* デフォルト引数は非推奨で、後方互換性のために残されていますが無視されます。また、このメソッドはキーワード専用引数 *pattern* を受け取るようになりました。これは load_tests の第三引数に渡されます。

loadTestsFromName(*name*, *module=None*)

文字列で指定される全テストケースを含むスイートを返します。

name には " ドット修飾名" でモジュールかテストケースクラス、テストケースクラス内のメソッド、*TestSuite* インスタンスまたは *TestCase* か *TestSuite* のインスタンスを返す呼び出し可能オブジェクトを指定します。このチェックはここで挙げた順番に行なわれます。すなわち、候補テストケースクラス内のメソッドは「呼び出し可能オブジェクト」としてではなく「テストケースクラス内のメソッド」として拾い出されます。

例えば SampleTests モジュールに *TestCase* から派生した SampleTestCase クラスがあり、SampleTestCase にはテストメソッド test_one() ・ test_two() ・ test_three() があるとします。この場合、*name* に 'SampleTests.SampleTestCase' と指定すると、SampleTestCase の三つのテストメソッドを実行するテストスイートが作成されます。'SampleTests.SampleTestCase.test_two' と指定すれば、test_two() だけを実行するテストスイートが作成されます。インポートされていないモジュールやパッケージ名を含んだ名前を指定した場合は自動的にインポートされます。

また、*module* を指定した場合、*module* 内の *name* を取得します。

バージョン 3.5 で変更: *name* を巡回している間に *ImportError* か *AttributeError* が発生した場合、実行するとその例外を発生させるようなテストを合成して返します。それらのエラーは self.errors に集められます。

loadTestsFromNames(*names*, *module=None*)

loadTestsFromName() と同じですが、名前を一つだけ指定するのではなく、複数の名前のシーケンスを指定する事ができます。戻り値は *names* 中の名前で指定されるテスト全てを含むテストスイートです。

getTestCaseNames(*testCaseClass*)

testCaseClass 中の全てのメソッド名を含むソート済みシーケンスを返します。*testCaseClass* は *TestCase* のサブクラスでなければなりません。

discover(*start_dir*, *pattern='test*.py'*, *top_level_dir=None*)

指定された開始ディレクトリからサブディレクトリに再帰することですべてのテストモジュールを検索し、それらを含む TestSuite オブジェクトを返します。*pattern* にマッチしたテストファイルだけがロードの対象になります。(シェルスタイルのパターンマッチングが使われます)。その中で、インポート可能なもジュール (つまり Python の識別子として有効であるということです) がロードされます。

すべてのテストモジュールはプロジェクトのトップレベルからインポート可能である必要があります。開始ディレクトリがトップレベルディレクトリでない場合は、トップレベルディレクトリを個別に指定しなければなりません。

シンタックスエラーなどでモジュールのインポートに失敗した場合、エラーが記録され、ディスカバリ自体は続けられます。import の失敗が *SkipTest* 例外が発生したためだった場合は、そのモジュールはエラーではなく skip として記録されます。

パッケージ (__init__.py という名前のファイルがあるディレクトリ) が見付かった場合、

そのパッケージに load_tests 関数があるかをチェックします。関数があった場合、次に package.load_tests(loader, tests, pattern) が呼ばれます。テストの検索の実行では、たとえ load_tests 関数自身が loader.discover を呼んだとしても、パッケージのチェックは 1 回のみとなることが保証されています。

load_tests が存在して、ディスカバリがパッケージ内を再帰的な検索を続けている途中で **ない** 場合、load_tests はそのパッケージ内の全てのテストをロードする責務を担います。

意図的にパターンはローダの属性として保持されないようになっています。それにより、パッケージが自分自身のディスカバリを続ける事ができます。*top_level_dir* は保持されるため、load_tests はこの引数を loader.discover() に渡す必要はありません。

start_dir はドット付のモジュール名でもディレクトリでも構いません。

バージョン 3.2 で追加.

バージョン 3.4 で変更: インポート時に *SkipTest* を送出するモジュールはエラーではなくスキップとして記録されます。

バージョン 3.4 で変更: *start_dir*に:term:‘名前空間パッケージ <namespace package>‘を指定できます。

バージョン 3.4 で変更: ファイルシステムの順序がファイル名に従わないとしても実行順序が一定になるように、パスはインポートする前にソートされます。

バージョン 3.5 で変更: パッケージ名がデフォルトのパターンに適合するのは不可能なので、パスが *pattern* に適合するかどうかに関係無く、見付けたパッケージに load_tests があるかをチェックするようになりました。

以下の属性は、サブクラス化またはインスタンスの属性値を変更して *TestLoader* をカスタマイズする場合に使用します:

testMethodPrefix
テストメソッドの名前と判断されるメソッド名の接頭語を示す文字列。デフォルト値は 'test' です。

この値は *getTestCaseNames()* と全ての loadTestsFrom*() メソッドに影響を与えます。

sortTestMethodsUsing
getTestCaseNames() および全ての loadTestsFrom*() メソッドでメソッド名をソートする際に使用する比較関数。

suiteClass
テストのリストからテストスイートを構築する呼び出し可能オブジェクト。メソッドを持つ必要はありません。デフォルト値は *TestSuite* です。

この値は全ての loadTestsFrom*() メソッドに影響を与えます。

testNamePatterns
テストメソッドがテストスイートに含まれるためにマッチしなければならない Unix シェルスタイルのワイルドカードテスト名パターンのリストです (“-v“オプションを参照)。

この属性が “None“ (デフォルト) でない場合、テストスイートに含まれるすべてのテストメソッドはこのリストのパターンのいずれかにマッチしていなければなりません。マッチは常に `fnmatch.fnmatchcase`を使用して実行されるため、`()-v“オプションに渡されるパターンとは異なり、単純な部分文字列パターンは “*“ワイルドカードを使用して変換する必要があることに注意してください。

この値は全ての `loadTestsFrom*()` メソッドに影響を与えます。

バージョン 3.7 で追加.

class unittest.TestResult

このクラスはどのテストが成功しどのテストが失敗したかという情報を収集するのに使います。

TestResult は、複数のテスト結果を記録します。*TestCase* クラスと *TestSuite* クラスのテスト結果を正しく記録しますので、テスト開発者が独自にテスト結果を管理する処理を開発する必要はありません。

unittest を利用したテストフレームワークでは、`TestRunner.run()` が返す *TestResult* インスタンスを参照し、テスト結果をレポートします。

TestResult インスタンスの以下の属性は、テストの実行結果を検査する際に使用することができます:

errors

TestCase と例外のトレースバック情報をフォーマットした文字列の 2 要素タプルからなるリスト。それぞれのタプルは予想外の例外を送出したテストに対応します。

failures

TestCase と例外のトレースバック情報をフォーマットした文字列の 2 要素タプルからなるリスト。それぞれのタプルは `TestCase.assert*()` メソッドを使って見つけ出した失敗に対応します。

skipped

TestCase インスタンスとテストをスキップした理由を保持する文字列の 2 要素タプルからなるリストです。

バージョン 3.1 で追加.

expectedFailures

TestCase と例外のトレースバック情報をフォーマット済の文字列の 2 要素タプルからなるリストです。それぞれのタプルは予期された失敗またはエラーに対応します。

unexpectedSuccesses

予期された失敗とされていながら成功してしまった *TestCase* のインスタンスのリスト。

shouldStop

`True` が設定されると *stop()* によりテストの実行が停止します。

testsRun

これまでに実行したテストの総数です。

buffer

True が設定されると、sys.stdout と sys.stderr は、*startTest()* から *stopTest()* が呼ばれるまでの間バッファリングされます。実際に、結果が sys.stdout と sys.stderr に出力されるのは、テストが失敗するかエラーが発生した時になります。表示の際には、全ての失敗 / エラーメッセージが表示されます。

バージョン 3.2 で追加.

failfast

真の場合 *stop()* が始めの失敗もしくはエラーの時に呼び出され、テストの実行が終了します。

バージョン 3.2 で追加.

tb_locals

真の場合、局所変数がトレースバックに表示されます。

バージョン 3.5 で追加.

wasSuccessful()

これまでに実行したテストが全て成功していれば True を、それ以外なら False を返します。

バージョン 3.4 で変更: *expectedFailure()* デコレータでマークされたテストに *unexpectedSuccesses* があった場合 False を返します。

stop()

このメソッドを呼び出して *TestResult* の *shouldStop* 属性に True をセットすることで、実行中のテストは中断しなければならないというシグナルを送ることができます。TestRunner オブジェクトはこのフラグを順守してそれ以上のテストを実行することなく復帰しなければなりません。

たとえばこの機能は、ユーザのキーボード割り込みを受け取って *TextTestRunner* クラスがテストフレームワークを停止させるのに使えます。TestRunner の実装を提供する対話的なツールでも同じように使用することができます。

TestResult クラスの以下のメソッドは内部データ管理用のメソッドですが、対話的にテスト結果をレポートするテストツールを開発する場合などにはサブクラスで拡張することができます。

startTest(*test***)**

test を実行する直前に呼び出されます。

stopTest(*test***)**

test の実行直後に、テスト結果に関わらず呼び出されます。

startTestRun()

全てのテストが実行される前に一度だけ実行されます。

バージョン 3.1 で追加.

stopTestRun()

全てのテストが実行された後に一度だけ実行されます。

バージョン 3.1 で追加.

addError(*test*, *err*)
テスト *test* 実行中に、想定外の例外が発生した場合に呼び出されます。*err* は *sys.exc_info()* が返すタプル (type, value, traceback) です。

デフォルトの実装では、タプル、(test, formatted_err) をインスタンスの *errors* 属性に追加します。ここで、*formatted_err* は、*err* から導出される、整形されたトレースバックです。

addFailure(*test*, *err*)
テストケース *test* が失敗した場合に呼び出されます。*err* は *sys.exc_info()* が返すタプル (type, value, traceback) です。

デフォルトの実装では、タプル、(test, formatted_err) をインスタンスの *failures* 属性に追加します。ここで、*formatted_err* は、*err* から導出される、整形されたトレースバックです。

addSuccess(*test*)
テストケース *test* が成功した場合に呼び出されます。

デフォルトの実装では何もしません。

addSkip(*test*, *reason*)
test がスキップされた時に呼び出されます。*reason* はスキップの際に渡された理由の文字列です。

デフォルトの実装では、(test, reason) のタプルをインスタンスの *skipped* 属性に追加します。

addExpectedFailure(*test*, *err*)
expectedFailure() のデコレータでマークされた *test* が失敗またはエラーの時に呼び出されます。

デフォルトの実装では (test, formatted_err) のタプルをインスタンスの *expectedFailures* に追加します。ここで *formatted_err* は *err* から派生した整形されたトレースバックです。

addUnexpectedSuccess(*test*)
expectedFailure() のデコレータでマークされた *test* が成功した時に呼び出されます。

デフォルトの実装ではテストをインスタンスの *unexpectedSuccesses* 属性に追加します。

addSubTest(*test*, *subtest*, *outcome*)
サブテストが終了すると呼ばれます。*test* はテストメソッドに対応するテストケースです。*subtest* はサブテストを記述するカスタムの *TestCase* インスタンスです。

outcome が *None* の場合サブテストは成功です。それ以外の場合は失敗で、*sys.exc_info()* が返す形式 (type, value, traceback) の *outcome* を持つ例外を伴います。

結果が成功の場合デフォルトの実装では何もせず、サブテストの失敗を通常の失敗として報告します。

バージョン 3.4 で追加.

class unittest.TextTestResult(*stream*, *descriptions*, *verbosity*)
TextTestRunner に使用される *TestResult* の具象実装です。

バージョン 3.2 で追加: このクラスは以前 _TextTestResult という名前でした。以前の名前はエイリアスとして残っていますが非推奨です。

unittest.defaultTestLoader

TestLoader のインスタンスで、共用することが目的です。*TestLoader* をカスタマイズする必要がなければ、新しい *TestLoader* オブジェクトを作らずにこのインスタンスを使用します。

class unittest.TextTestRunner(*stream=None*, *descriptions=True*, *verbosity=1*, *failfast=False*, *buffer=False*, *resultclass=None*, *warnings=None*, ***, *tb_locals=False*)

結果をストリームに出力する、基本的なテストランナーの実装です。*stream* が None の場合、デフォルトで *sys.stderr* が出力ストリームとして使われます。このクラスはいくつかの設定項目があるだけで、基本的に非常に単純です。グラフィカルなテストの実行アプリケーションでは、独自のテストランナーを実装してください。テストランナーの実装は、unittest に新しい機能が追加されランナーを構築するインターフェースが変更されたときに備えて **kwargs を受け取れるようにするべきです。

デフォルトで無視 に設定されているとしても、このランナーのデフォルトでは *DeprecationWarning*, *PendingDeprecationWarning*, *ResourceWarning*, *ImportWarning* を表示します。*unittest* **の非推奨メソッド** で起きた非推奨警告も特別な場合として扱われ、警告フィルタが 'default' もしくは 'always' だったとき、対象の警告メッセージが出ないようにモジュールごとに 1 回だけ表示されます。Python の -Wd オプションや -Wa オプション (警告の制御 を参照してください) を使ったり、*warnings* を None にしたりしておくと、この動作を上書きできます。

バージョン 3.2 で変更: warnings 引数が追加されました。

バージョン 3.2 で変更: インポート時でなくインスタンス化時にデフォルトのストリームが *sys.stderr* に設定されます。

バージョン 3.5 で変更: tb_locals 引数が追加されました。

_makeResult()

このメソッドは *run()* で使われる TestResult のインスタンスを返します。このメソッドは明示的に呼び出す必要はありませんが、サブクラスで TestResult をカスタマイズすることができます。

_makeResult() は、TextTestRunner のコンストラクタで resultclass 引数として渡されたクラスもしくはコーラブルオブジェクトをインスタンス化します。resultclass が指定されていない場合には、デフォルトで *TextTestResult* が使用されます。結果のクラスは以下の引数が渡されインスタンス化されます:

```
stream, descriptions, verbosity
```

run(*test*)

このメソッドは TextTestRunner へのメインの公開インターフェースです。このメソッドは *TestSuite* インスタンスあるいは *TestCase* インスタンスを受け取ります。*_makeResult()* が呼ばれて *TestResult* が作成され、テストが実行され、結果が標準出力に表示されます。

unittest.main(*module='__main__', defaultTest=None, argv=None, testRunner=None, testLoader=unittest.defaultTestLoader, exit=True, verbosity=1, failfast=None, catchbreak=None, buffer=None, warnings=None*)

module から複数のテストを読み込んで実行するためのコマンドラインプログラム。この関数を使えば、簡単に実行可能なテストモジュールを作成する事ができます。一番簡単なこの関数の使い方は、以下の行をテストスクリプトの最後に置くことです:

```
if __name__ == '__main__':
    unittest.main()
```

より詳細な情報は verbosity 引数を指定して実行すると得られます:

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

defaultTest 引数は、*argv* にテスト名が指定されていない場合に実行する、ある 1 つのテストの名前もしくはテスト名のイテラブルです。この引数を指定しないか None を指定し、かつ *argv* にテスト名が与えられない場合は、*module* にある全てのテストを実行します。

argv 引数には、プログラムに渡されたオプションのリストを、最初の要素がプログラム名のままで渡せます。指定しないか None の場合は *sys.argv* が使われます。

引数、*testRunner* は、test runner class、あるいは、そのインスタンスのどちらでも構いません。デフォルトでは main はテストが成功したか失敗したかに対応した終了コードと共に *sys.exit()* を呼び出します。

testLoader 引数は *TestLoader* インスタンスでなければなりません。デフォルトは *defaultTestLoader* です。

main は、exit=False を指定する事で対話的なインタプリタから使用することもできます。この引数を指定すると、*sys.exit()* を呼ばずに、結果のみを出力します:

```
>>> from unittest import main
>>> main(module='test_module', exit=False)
```

failfast, *catchbreak*, *buffer* は、**コマンドラインオプション** にある同名のオプションと同じ効果のあるパラメータです。

warnings 引数では、テストの実行中に使うべき **警告フィルタ** を指定します。この引数が指定されない場合には、-W オプションが python に渡されていなければ None のまま (警告の制御 を参照してください) で、そうでなければ 'default' が設定されます。

main を呼び出すと、TestProgram のインスタンスが返されます。このインスタンスは、result 属性にテスト結果を保持します。

バージョン 3.1 で変更: *exit* パラメータが追加されました。

バージョン 3.2 で変更: *verbosity*、*failfast*、*catchbreak*、*buffer*、*warnings* 引数が追加されました。

バージョン 3.4 で変更: *defaultTest* 引数がテスト名のイテラブルも受け取るようになりました。

load_tests プロトコル

バージョン 3.2 で追加.

モジュールやパッケージには、load_tests と呼ばれる関数を実装できます。これにより、通常のテスト実行時やテストディスカバリ時のテストのロードされ方をカスタマイズできます。

テストモジュールが load_tests を定義していると、それが *TestLoader.loadTestsFromModule()* から呼ばれます。引数は以下です:

```
load_tests(loader, standard_tests, pattern)
```

pattern は loadTestsFromModule からそのまま渡されます。デフォルトは None です。

これは *TestSuite* を返すべきです。

loader はローディングを行う *TestLoader* のインスタンスです。*standard_tests* は、そのモジュールからデフォルトでロードされるテストです。これは、テストの標準セットのテストの追加や削除のみを行いたいテストモジュールに一般に使われます。第三引数は、パッケージをテストディスカバリの一部としてロードするときに使われます。

特定の *TestCase* クラスのセットからテストをロードする典型的な load_tests 関数は、このようになります:

```
test_cases = (TestCase1, TestCase2, TestCase3)

def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite
```

コマンドラインからでも *TestLoader.discover()* の呼び出しでも、パッケージを含むディレクトリで検索を始めた場合、そのパッケージの __init__.py をチェックして load_tests を探します。その関数が存在しない場合、他のディレクトリであるかのようにパッケージの中を再帰的に検索します。その関数が存在した場合、パッケージのテストの検索をそちらに任せ、load_tests が次の引数で呼び出されます:

```
load_tests(loader, standard_tests, pattern)
```

これはパッケージ内のすべてのテストを表す *TestSuite* を返すべきです。(standard_tests には、__init__.py から収集されたテストのみが含まれます。)

パターンは load_tests に渡されるので、パッケージは自由にテストディスカバリを継続 (必要なら変更) できます。テストパッケージに ' 何もしない' load_tests 関数は次のようになります:

```
def load_tests(loader, standard_tests, pattern):
    # top level directory cached on loader instance
    this_dir = os.path.dirname(__file__)
```

(次のページに続く)

(前のページからの続き)

```
    package_tests = loader.discover(start_dir=this_dir, pattern=pattern)
    standard_tests.addTests(package_tests)
    return standard_tests
```

バージョン 3.5 で変更: パッケージ名がデフォルトのパターンに適合するのが不可能なため、検索ではパッケージ名が *pattern* に適合するかのチェックは行われなくなりました。

26.4.9 クラスとモジュールのフィクスチャ

クラスレベルとモジュールレベルのフィクスチャが *TestSuite* に実装されました。テストスイートが新しいクラスのテストを始める時、以前のクラス (あれば) の `tearDownClass()` を呼び出し、その後に新しいクラスの `setUpClass()` を呼び出します。

同様に、今回のテストのモジュールが前回のテストとは異なる場合、以前のモジュールの `tearDownModule` を実行し、次に新しいモジュールの `setUpModule` を実行します。

すべてのテストが実行された後、最後の `tearDownClass` と `tearDownModule` が実行されます。

なお、共有フィクスチャは、テストの並列化などの [潜在的な] 機能と同時にはうまくいかず、テストの分離を壊すので、気をつけて使うべきです。

unittest テストローダによるテスト作成のデフォルトの順序では、同じモジュールやクラスからのテストはすべて同じグループにまとめられます。これにより、`setUpClass` / `setUpModule` (など) は、一つのクラスやモジュールにつき一度だけ呼ばれます。この順序をバラバラにし、異なるモジュールやクラスのテストが並ぶようにすると、共有フィクスチャ関数は、一度のテストで複数回呼ばれるようにもなります。

共有フィクスチャは標準でない順序で実行されることを意図していません。共有フィクスチャをサポートしたくないフレームワークのために、`BaseTestSuite` がまだ存在しています。

共有フィクスチャ関数のいずれかで例外が発生した場合、そのテストはエラーとして報告されます。そのとき、対応するテストインスタンスが無いので (*TestCase* と同じインタフェースの) `_ErrorHolder` オブジェクトが生成され、エラーを表します。標準 unittest テストランナーを使っている場合はこの詳細は問題になりませんが、あなたがフレームワークの作者である場合は注意してください。

setUpClass と tearDownClass

これらはクラスメソッドとして実装されなければなりません:

```
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
```

(次のページに続く)

(前のページからの続き)

```
    def tearDownClass(cls):
        cls._connection.destroy()
```

基底クラスの setUpClass および tearDownClass を使いたいなら、それらを自分で呼び出さなければなりません。*TestCase* の実装は空です。

setUpClass の中で例外が送出されたら、クラス内のテストは実行されず、tearDownClass も実行されません。スキップされたクラスは setUpClass も tearDownClass も実行されません。例外が *SkipTest* 例外であれば、そのクラスはエラーではなくスキップされたものとして報告されます。

setUpModule と tearDownModule

これらは関数として実装されなければなりません:

```
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

setUpModule の中で例外が送出されたら、モジュール内のテストは実行されず、tearDownModule も実行されません。例外が *SkipTest* 例外であれば、そのモジュールはエラーではなくスキップされたものとして報告されます。

例外が発生した場合でも実行しなければならないクリーンアップコードを追加するには “addModuleCleanup“を使用します:

unittest.addModuleCleanup(*function*, */*, **args*, ***kwargs*)

tearDownModule() の後に呼び出される関数を追加します。この関数はリソースのクリーンアップのために使用します。追加された関数は、追加された順と逆の順番で呼び出されます（LIFO）。*addModuleCleanup()* に渡された引数とキーワード引数が追加された関数にも渡されます。

setUpModule() が失敗した場合、つまり tearDownModule() が呼ばれなかった場合でも、追加されたクリーンアップ関数は呼び出されます。

バージョン 3.8 で追加.

unittest.doModuleCleanups()

この関数は、tearDownModule() の後、もしくは、setUpModule() が例外を投げた場合は setUpModule() の後に、無条件で呼ばれます。

このメソッドは、addCleanupModule() で追加された関数を呼び出す責務を担います。もし、クリーンアップ関数を tearDownModule() より前に呼び出す必要がある場合には、*doModuleCleanups()* を明示的に呼び出してください。

doModuleCleanups() は、どこで呼び出されても、クリーンアップ関数をスタックから削除して実行します。

バージョン 3.8 で追加.

26.4.10 シグナルハンドリング

バージョン 3.2 で追加.

unittest の *-c/--catch* コマンドラインオプションや、*unittest.main()* の catchbreak パラメタは、テスト実行中の control-C の処理をよりフレンドリーにします。中断捕捉動作を有効である場合、control-C が押されると、現在実行されているテストまで完了され、そのテストランが終わると今までの結果が報告されます。control-C がもう一度押されると、通常通り *KeyboardInterrupt* が送出されます。

シグナルハンドラを処理する control-c は、独自の *signal.SIGINT* ハンドラをインストールするコードやテストの互換性を保とうとします。unittest ハンドラが呼ばれ、それがインストールされた *signal.SIGINT* ハンドラで **なければ** 、すなわちテスト中のシステムに置き換えられて移譲されたなら、それはデフォルトのハンドラを呼び出します。インストールされたハンドラを置き換えて委譲するようなコードは、通常その動作を期待するからです。unittest の control-c 処理を無効にしたいような個別のテストには、*removeHandler()* デコレータが使えます。

フレームワークの作者がテストフレームワーク内で control-c 処理を有効にするための、いくつかのユーティリティ関数があります。

unittest.installHandler()

control-c ハンドラをインストールします。(主にユーザが control-c を押したことにより) *signal.SIGINT* が受け取られると、登録した結果すべてに *stop()* が呼び出されます。

unittest.registerResult(*result*)

control-c 処理のために *TestResult* を登録します。結果を登録するとそれに対する弱参照が格納されるので、結果がガベージコレクトされるのを妨げません。

control-c 処理が有効でなければ、*TestResult* オブジェクトの登録には副作用がありません。ですからテストフレームワークは、処理が有効か無効かにかかわらず、作成する全ての結果を無条件に登録できます。

unittest.removeResult(*result*)

登録された結果を削除します。一旦結果が削除されると、control-c が押された際にその結果オブジェクトに対して *stop()* が呼び出されなくなります。

unittest.removeHandler(*function=None*)

引数なしで呼び出されると、この関数は Ctrl+C のシグナルハンドラを（それがインストールされていた場合）削除します。また、この関数はテストが実行されている間、Ctrl+C のハンドラを一時的に削除するテストデコレーターとしても使用できます。

```
@unittest.removeHandler
def test_signal_handling(self):
    ...
```

26.5 unittest.mock --- モックオブジェクトライブラリ

バージョン 3.3 で追加.

ソースコード: Lib/unittest/mock.py

---

unittest.mock は Python におけるソフトウェアテストのためのライブラリです。テスト中のシステムの一部をモックオブジェクトで置き換え、それらがどのように使われるかをアサートすることができます。

unittest.mock はコア *Mock* クラスを提供しており、それによってテストスイート内でたくさんのスタブを作成しなくてすみます 。アクションの実行後、メソッドや属性の使用や実引数についてアサートできます。また通常の方法で戻り値を明記したり、必要な属性を設定することもできます。

加えて、mock はテストのスコープ内にあるモジュールやクラスの属性を変更する *patch()* デコレータを提供します。さらに、ユニークなオブジェクトの作成には *sentinel* が利用できます。*Mock* や *MagicMock* 、*patch()* の利用例は *quick guide* を参照してください。

Mock はとても使いやすく、*unittest* で利用するために設計されています。Mock は多くのモックフレームワークで使われる 'record -> replay' パターンの代わりに、'action -> assertion' パターンに基づいています。

以前の Python バージョン向けにバックポートされた *unittest.mock* があり、mock on PyPI として利用可能です。

26.5.1 クイックガイド

Mock および *MagicMock* オブジェクトはアクセスしたすべての属性とメソッドを作成し、どのように使用されたかについての詳細な情報を格納します。戻り値を指定したり利用できる属性を制限するために *Mock* および *MagicMock* を設定でき、どのよう使用されたかについてアサートできます:

```
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

side_effect によって、モック呼び出し時の例外発生などの副作用を実行できます:

```
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
 ...
KeyError: 'foo'
```

```
>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]
```

(次のページに続く)

(前のページからの続き)

```
...
>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
>>> mock(), mock(), mock()
(5, 4, 3)
```

モックには多くの設定法や挙動の制御法があります。例えば *spec* 引数によって別のオブジェクトからの仕様を受け取るよう設定できます。spec にないモックの属性やメソッドにアクセスを試みた場合、*AttributeError* で失敗します。

patch() デコレータ / コンテキストマネージャーによってテスト対象のモジュール内のクラスやオブジェクトを簡単にモックできます。指定したオブジェクトはテスト中はモック (または別のオブジェクト) に置換され、テスト終了時に復元されます:

```
>>> from unittest.mock import patch
>>> @patch('module.ClassName2')
... @patch('module.ClassName1')
... def test(MockClass1, MockClass2):
...     module.ClassName1()
...     module.ClassName2()
...     assert MockClass1 is module.ClassName1
...     assert MockClass2 is module.ClassName2
...     assert MockClass1.called
...     assert MockClass2.called
...
>>> test()
```

注釈: patch デコレータをネストした場合、モックは適用されるときと同じ順番 (デコレータを適用するときの通常の *Python* の順番) でデコレートされた関数に渡されます。つまり下から順に適用されるため、上の例では `module.ClassName1` のモックが最初に渡されます。

patch() では探索される名前空間内のオブジェクトにパッチをあてることが重要です。通常は単純ですが、クイックガイドには *where-to-patch* を読んでください。

---

patch() デコレータと同様に with 文でコンテキストマネージャーとして使用できます:

```
>>> with patch.object(ProductionClass, 'method', return_value=None) as mock_method:
...     thing = ProductionClass()
...     thing.method(1, 2, 3)
...
>>> mock_method.assert_called_once_with(1, 2, 3)
```

また、*patch.dict()* を使うと、スコープ内だけで辞書に値を設定し、テスト終了時には元の状態に復元されます:

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

mock は Python の **マジックメソッド** のモックをサポートしています。マジックメソッドのもっとも簡単な利用法は *MagicMock* クラスと使うことです。以下のように利用します:

```
>>> mock = MagicMock()
>>> mock.__str__.return_value = 'foobarbaz'
>>> str(mock)
'foobarbaz'
>>> mock.__str__.assert_called_with()
```

mock によってマジックメソッドに関数 (あるいは他の Mock インスタンス) を割り当てることができ、それらは適切に呼び出されます。*MagicMock* クラスは、すべてのマジックメソッドがあらかじめ作成されている点を除けば *Mock* クラスと一緒です (まあ、とにかく便利ってこと)。

以下は通常の Mock クラスでマジックメソッドを利用する例です:

```
>>> mock = Mock()
>>> mock.__str__ = Mock(return_value='wheeeeee')
>>> str(mock)
'wheeeeee'
```

テスト内のモックオブジェクトが置換するオブジェクトと同じ API を持つことを保証するには、*auto-speccing* を使うことができます。パッチをあてる *autospec* 引数、または *create_autospec()* 関数を通じて auto-speccing は行われます。auto-speccing は置換するオブジェクトと同じ属性とメソッドを持つモックオブジェクトを作成し、すべての関数および (コンストラクタを含む) メソッドは本物のオブジェクトと同じ呼び出しシグネチャを持ちます。

誤って使用された場合、モックは製品コードと同じように失敗されることが保証されています:

```
>>> from unittest.mock import create_autospec
>>> def function(a, b, c):
...     pass
...
>>> mock_function = create_autospec(function, return_value='fishy')
>>> mock_function(1, 2, 3)
'fishy'
>>> mock_function.assert_called_once_with(1, 2, 3)
>>> mock_function('wrong arguments')
Traceback (most recent call last):
 ...
TypeError: <lambda>() takes exactly 3 arguments (1 given)
```

create_autospec() はクラスにおいても利用でき、`__init__` メソッドのシグニチャをコピーします。また、呼び出し可能オブジェクトについても `__call__` メソッドのシグニチャをコピーします。

26.5.2 Mock クラス

Mock は、コードにおけるスタブの使用やテストダブルを置き換えるための柔軟なモックオブジェクトです。モックは呼び出し可能で、属性にアクセスした場合それを新たなモックとして作成します[*1]。同じ属性にアクセスした場合は常に同じモックを返します。モックはどのように使われたかを記録するので、コードがモックに行うことについてアサートできます。

MagicMock は *Mock* のサブクラスで、すべてのマジックメソッドが事前に作成され、利用できます。また、呼び出し不可能なモックを作成する場合には、呼び出し不能な変種の *NonCallableMock* や *NonCallableMagicMock* があります。

patch() デコレータによって特定のモジュール内のクラスを *Mock* オブジェクトで一時的に置換することが簡単にできます。デフォルトでは *patch()* は *MagicMock* を作成します。*patch()* に渡す *new_callable* 引数によって、別の *Mock* クラスを指定できます。

class unittest.mock.**Mock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, ***kwargs*)

新しい *Mock* オブジェクトを作成します。*Mock* はモックオブジェクトの挙動を指定するオプション引数をいくつか取ります:

- *spec*: モックオブジェクトの仕様として働く文字列のリストもしくは存在するオブジェクト (クラスもしくはインスタンス) を指定します。オブジェクトを渡した場合には、dir 関数によって文字列のリストが生成されます (サポートされない特殊属性や特殊メソッドは除く) 。このリストにない属性にアクセスした際には *AttributeError* が発生します。

  spec が (文字列のリストではなく) オブジェクトの場合、*__class__* はスペックオブジェクトのクラスを返します。これによってモックが *isinstance()* テストに通るようになります。

- *spec_set*: より厳しい *spec* です。こちらを利用した場合、*spec_set* に渡されたオブジェクトに存在しない属性に対し **設定** や取得をしようとした際に *AttributeError* が発生します。

- *side_effect*: モックが呼び出された際に呼び出される関数を指定します。*side_effect* 属性を参考にしてください。例外を発生させたり、動的に戻り値を変更する場合に便利です。関数には、モックと同じ引数が渡され、*DEFAULT* を返さない限りはこの関数の戻り値が使われます。

  一方で、*side_effect* には、例外クラスやインスタンスを指定できます。この場合は、モックが呼び出された際に指定された例外を発生します。

  もし、*side_effect* にイテレート可能オブジェクトを指定した場合には、モックの呼び出しごとに順に値を返します。

  side_effect に `None` を指定した場合には、設定がクリアされます。

- *return_value*: モックが呼び出された際に返す値です。デフォルトでは (最初にアクセスされた際に生成される) 新しい Mock を返します。*return_value* を参照してください。

[*1] 例外は特殊メソッドと属性だけです (これらは 2 つのアンダースコアで開始・終了します)。モックはこれらの代わりに *AttributeError* を発生させます。これは、インタープリタが暗黙的にこれらのメソッドを要求するためであり、特殊メソッドを予測する際に **非常に** 混乱してしまいます。もし特殊メソッドのサポートが必要な場合は、*magic methods* を参照してください。

- *unsafe*: デフォルトでは、何らかの属性が *assert* または *assret* で始まると *AttributeError* が上がります。unsafe=True を渡すと、これらの属性へのアクセスが許可されます。

  バージョン 3.5 で追加.

- *wraps*: ラップするモックオブジェクトを指定します。もし *wraps* に None 以外を指定した場合には、モックの呼び出し時に指定したオブジェクトを呼び出します (戻り値は実際の結果を返します)。属性へのアクセスは、ラップされたオブジェクトと対応するモックを返します (すなわち、存在しない属性にアクセスしようとした場合は *AttributeError* が発生します)。

  return_value が設定されている場合には、ラップ対象は呼び出されず、*return_value* が返されます。

- *name*: もし、モックが *name* を持つ場合には、モックの repr として使われます。デバッグの際に役立つでしょう。この値は、子のモックにも伝播します。

モックは、任意のキーワード引数を与えることができます。これらはモックの生成後、属性の設定に使われます。詳細は *configure_mock()* を参照してください。

assert_called()

モックが少なくとも一度は呼び出されたことをアサートします。

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called()
```

バージョン 3.6 で追加.

assert_called_once()

モックが一度だけ呼び出されたことをアサートします。

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
Traceback (most recent call last):
...
AssertionError: Expected 'method' to have been called once. Called 2 times.
```

バージョン 3.6 で追加.

assert_called_with(**args*, ***kwargs*)

This method is a convenient way of asserting that the last call has been made in a particular way:

```
>>> mock = Mock()
>>> mock.method(1, 2, 3, test='wow')
```

(次のページに続く)

(前のページからの続き)

```
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_with(1, 2, 3, test='wow')
```

assert_called_once_with(**args*, ***kwargs*)

Assert that the mock was called exactly once and that that call was with the specified arguments.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar='baz')
>>> mock.assert_called_once_with('foo', bar='baz')
>>> mock('other', bar='values')
>>> mock.assert_called_once_with('other', bar='values')
Traceback (most recent call last):
  ...
AssertionError: Expected 'mock' to be called once. Called 2 times.
```

assert_any_call(**args*, ***kwargs*)

モックが特定の引数で呼び出されたことがあるのをアサートします。

The assert passes if the mock has *ever* been called, unlike *assert_called_with()* and *assert_called_once_with()* that only pass if the call is the most recent one, and in the case of *assert_called_once_with()* it must also be the only call.

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, arg='thing')
>>> mock('some', 'thing', 'else')
>>> mock.assert_any_call(1, 2, arg='thing')
```

assert_has_calls(*calls*, *any_order=False*)

モックが特定の呼び出しで呼ばれたことをアサートします。呼び出しでは *mock_calls* のリストがチェックされます。

any_order が false の場合、呼び出しは連続していなければなりません。指定された呼び出しの前、あるいは呼び出しの後に余分な呼び出しがある場合があります。

any_order が true の場合、呼び出しは任意の順番でも構いませんが、それらがすべて *mock_calls* に現われなければなりません。

```
>>> mock = Mock(return_value=None)
>>> mock(1)
>>> mock(2)
>>> mock(3)
>>> mock(4)
>>> calls = [call(2), call(3)]
>>> mock.assert_has_calls(calls)
>>> calls = [call(4), call(2), call(3)]
>>> mock.assert_has_calls(calls, any_order=True)
```

assert_not_called()

モックが一度も呼ばれなかったことをアサートします。

```
>>> m = Mock()
>>> m.hello.assert_not_called()
>>> obj = m.hello()
>>> m.hello.assert_not_called()
Traceback (most recent call last):
  ...
AssertionError: Expected 'hello' to not have been called. Called 1 times.
```

バージョン 3.5 で追加.

reset_mock(*, *return_value=False*, *side_effect=False*)

モックオブジェクトのすべての呼び出し属性をリセットします:

```
>>> mock = Mock(return_value=None)
>>> mock('hello')
>>> mock.called
True
>>> mock.reset_mock()
>>> mock.called
False
```

バージョン 3.6 で変更: 2 つのキーワード専用引数が reset_mock 関数に追加されました。

This can be useful where you want to make a series of assertions that reuse the same object. Note that *reset_mock() doesn't* clear the return value, *side_effect* or any child attributes you have set using normal assignment by default. In case you want to reset *return_value* or *side_effect*, then pass the corresponding parameter as `True`. Child mocks and the return value mock (if any) are reset as well.

注釈: *return_value* と *side_effect* はキーワード専用引数です。

---

mock_add_spec(*spec*, *spec_set=False*)

モックに仕様を追加します。*spec* にはオブジェクトもしくは文字列のリストを指定してください。*spec* で設定した属性は、モックの属性としてのみアクセスできます。

spec_set が真なら、spec 以外の属性は設定できません。

attach_mock(*mock*, *attribute*)

属性として mock を設定して、その名前と親を入れ替えます。設定されたモックの呼び出しは、*method_calls* や *mock_calls* 属性に記録されます。

configure_mock(***kwargs*)

モックの属性をキーワード引数で設定します。

属性に加え、子の戻り値や副作用もドット表記を用いて設定でき、辞書はメソッドの呼び出し時にアンパックされます:

```
>>> mock = Mock()
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock.configure_mock(**attrs)
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

コンストラクタの呼び出しでも同様に行うことができます:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

configure_mock() は、モック生成後のコンフィギュレーションを容易に行うために存在します。

__dir__()

Mock オブジェクトは、有用な結果を得るために dir(some_mock) の結果を制限します。*spec* を設定したモックに対しては、許可された属性のみを含みます。

このフィルタが何をしていて、どのように停止させるかは、*FILTER_DIR* を参照してください。

_get_child_mock(kw)**

子のモックを作成し、その値を返すようにしてください。デフォルトでは親と同じタイプで作成されます。サブクラスで子モックの作成される方法をカスタマイズしたい場合には、このメソッドをオーバーライドします。

呼び出し不可能なモックに対しては、(カスタムのサブクラスではなく) 呼び出し可能なモックが使われます。

called

このモックが呼び出されたかどうかを表します:

```
>>> mock = Mock(return_value=None)
>>> mock.called
False
>>> mock()
>>> mock.called
True
```

call_count

このモックオブジェクトが呼び出された回数を返します:

```
>>> mock = Mock(return_value=None)
>>> mock.call_count
0
>>> mock()
>>> mock()
>>> mock.call_count
2
```

return_value

モックが呼び出された際に返す値を設定します:

```
>>> mock = Mock()
>>> mock.return_value = 'fish'
>>> mock()
'fish'
```

デフォルトの戻り値はモックオブジェクトです。通常の方法で設定することもできます:

```
>>> mock = Mock()
>>> mock.return_value.attribute = sentinel.Attribute
>>> mock.return_value()
<Mock name='mock()()' id='...'>
>>> mock.return_value.assert_called_with()
```

return_value は生成時にも設定可能です:

```
>>> mock = Mock(return_value=3)
>>> mock.return_value
3
>>> mock()
3
```

side_effect

このモックが呼ばれた際に呼び出される関数、イテラブル、もしくは発生させる例外 (クラスまたはインスタンス) を設定できます。

関数を渡した場合はモックと同じ引数で呼び出され、*DEFAULT* を返さない限りはその関数の戻り値が返されます。関数が *DEFAULT* を返した場合は (*return_value* によって) モックの通常の値を返します。

iterable が渡された場合、その値はイテレータを取り出すために使用されます。イテレータは毎回の呼び出しにおいて値を yield しなければなりません。この値は、送出される例外インスタンスか、呼び出しからモックに返される値のいずれかです (*DEFAULT* の処理は関数の場合と同一です)。

以下はモックが (API による例外の扱いをテストするために) 例外を発生させる例です:

```
>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
>>> mock()
Traceback (most recent call last):
```

(次のページに続く)

(前のページからの続き)

```
  ...
Exception: Boom!
```

side_effect を使用して連続的に値を返します:

```
>>> mock = Mock()
>>> mock.side_effect = [3, 2, 1]
>>> mock(), mock(), mock()
(3, 2, 1)
```

呼び出し可能オブジェクトを使います:

```
>>> mock = Mock(return_value=3)
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> mock.side_effect = side_effect
>>> mock()
3
```

side_effect は生成時にも設定可能です。呼び出し時の値に 1 を加えて返す例を以下に示します:

```
>>> side_effect = lambda value: value + 1
>>> mock = Mock(side_effect=side_effect)
>>> mock(3)
4
>>> mock(-8)
-7
```

side_effect に None を設定した場合はクリアされます:

```
>>> m = Mock(side_effect=KeyError, return_value=3)
>>> m()
Traceback (most recent call last):
 ...
KeyError
>>> m.side_effect = None
>>> m()
3
```

call_args

This is either **None** (if the mock hasn't been called), or the arguments that the mock was last called with. This will be in the form of a tuple: the first member, which can also be accessed through the **args** property, is any ordered arguments the mock was called with (or an empty tuple) and the second member, which can also be accessed through the **kwargs** property, is any keyword arguments (or an empty dictionary).

```
>>> mock = Mock(return_value=None)
>>> print(mock.call_args)
```

(次のページに続く)

(前のページからの続き)

```
None
>>> mock()
>>> mock.call_args
call()
>>> mock.call_args == ()
True
>>> mock(3, 4)
>>> mock.call_args
call(3, 4)
>>> mock.call_args == ((3, 4),)
True
>>> mock.call_args.args
(3, 4)
>>> mock.call_args.kwargs
{}
>>> mock(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args
call(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args.args
(3, 4, 5)
>>> mock.call_args.kwargs
{'key': 'fish', 'next': 'w00t!'}
```

call_args は、*call_args_list* や *method_calls* 、*mock_calls* と同様、*call* オブジェクトです。これらはタプルとしてアンパックすることで個別に取り出すことができます。そして、より複雑なアサーションを行うことができます。*calls as tuples* を参照してください。

バージョン 3.8 で変更: args と kwargs 属性が追加されました。

call_args_list

モックの呼び出しを順に記録したリストです (よって、このリストの長さはモックが呼び出された回数と等しくなります)。モックを作成してから一度も呼び出しを行なっていない場合は、空のリストが返されます。*call* オブジェクトは、*call_args_list* の比較対象となる呼び出しのリストを作成する際に便利です。

```
>>> mock = Mock(return_value=None)
>>> mock()
>>> mock(3, 4)
>>> mock(key='fish', next='w00t!')
>>> mock.call_args_list
[call(), call(3, 4), call(key='fish', next='w00t!')]
>>> expected = [(), ((3, 4),), ({'key': 'fish', 'next': 'w00t!'},)]
>>> mock.call_args_list == expected
True
```

call_args_list のメンバは *call* オブジェクトです。タプルとしてアンパックすることで個別に取り出すことができます。*calls as tuples* を参照してください。

method_calls

自身の呼び出しと同様に、モックはメソッドや属性、そして **それらの** メソッドや属性の呼び出し

も追跡します:

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.property.method.attribute()
<Mock name='mock.property.method.attribute()' id='...'>
>>> mock.method_calls
[call.method(), call.property.method.attribute()]
```

method_calls のメンバは *call* オブジェクトです。タプルとしてアンパックすることで個別に取り出すことができます。*calls as tuples* を参照してください。

mock_calls

mock_calls は、メソッド、特殊メソッド、**そして** 戻り値のモックまで、モックオブジェクトに対する **すべての** 呼び出しを記録します。

```
>>> mock = MagicMock()
>>> result = mock(1, 2, 3)
>>> mock.first(a=3)
<MagicMock name='mock.first()' id='...'>
>>> mock.second()
<MagicMock name='mock.second()' id='...'>
>>> int(mock)
1
>>> result(1)
<MagicMock name='mock()()' id='...'>
>>> expected = [call(1, 2, 3), call.first(a=3), call.second(),
... call.__int__(), call()(1)]
>>> mock.mock_calls == expected
True
```

mock_calls のメンバは *call* オブジェクトです。タプルとしてアンパックすることで個別に取り出すことができます。*calls as tuples* を参照してください。

注釈: The way *mock_calls* are recorded means that where nested calls are made, the parameters of ancestor calls are not recorded and so will always compare equal:

```
>>> mock = MagicMock()
>>> mock.top(a=3).bottom()
<MagicMock name='mock.top().bottom()' id='...'>
>>> mock.mock_calls
[call.top(a=3), call.top().bottom()]
>>> mock.mock_calls[-1] == call.top(a=-1).bottom()
True
```

__class__

通常、オブジェクトの *__class__* 属性はその型を返します。spec を設定したオブジェクトの場合、__class__ は代わりに spec のクラスを返します。これにより、置き換え / 偽装しているオ

ブジェクトに対する *isinstance()* も通過することができます:

```
>>> mock = Mock(spec=3)
>>> isinstance(mock, int)
True
```

__class__ は書き換え可能で、*isinstance()* を通るために必ず spec を使う必要はありません:

```
>>> mock = Mock()
>>> mock.__class__ = dict
>>> isinstance(mock, dict)
True
```

class unittest.mock.NonCallableMock(*spec=None*, *wraps=None*, *name=None*, *spec_set=None*, ***kwargs*)

呼び出しができない *Mock* です。コンストラクタのパラメータは *Mock* と同様ですが、*return_value* や *side_effect* は意味を持ちません。

spec か spec_set にクラスかインスタンスを渡した mock は *isinstance()* テストをパスします:

```
>>> mock = Mock(spec=SomeClass)
>>> isinstance(mock, SomeClass)
True
>>> mock = Mock(spec_set=SomeClass())
>>> isinstance(mock, SomeClass)
True
```

Mock クラスは、特殊メソッドをサポートしています。すべての詳細は *magic methods* を参照してください。

モッククラスや *patch()* デコレータは、任意のキーワード引数を設定できます。*patch()* デコレータへのキーワード引数は、モックが作られる際のコンストラクタに渡されます。キーワード引数は、モックの属性を設定します:

```
>>> m = MagicMock(attribute=3, other='fish')
>>> m.attribute
3
>>> m.other
'fish'
```

子の戻り値や副作用も、ドットで表記することで同様に設定できます。呼び出し時に直接ドットのついた名前を使用できないので、作成した辞書を ** でアンパックする必要があります:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
```

(次のページに続く)

(前のページからの続き)

```
  ...
KeyError
```

spec (または *spec_set*) によって作成された呼び出し可能なモックは、モックへの呼び出しがマッチしたときに仕様オブジェクトのシグネチャを内省します。したがって、引数を位置引数として渡したか名前で渡したかどうかに関わらず、実際の呼び出しの引数とマッチすることができます:

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, c=3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(1, 2, 3)
>>> mock.assert_called_with(a=1, b=2, c=3)
```

これは *assert_called_with()*, *assert_called_once_with()*, *assert_has_calls()* and *assert_any_call()* にも適用されます。*autospec* **を使う** と、モックオブジェクトのメソッド呼び出しにも適用されます。

バージョン 3.4 で変更: spec や autospec を用いて生成されたモックオブジェクトは、シグネチャを考慮するようになりました。

class unittest.mock.**PropertyMock**(**args*, ***kwargs*)

プロパティもしくはディスクリプタとして使われるためのモックです。*PropertyMock* は、`__get__()` と `__set__()` メソッドを提供し、戻り値を指定することができます。

オブジェクトから *PropertyMock* のインスタンスを取得することは、引数を与えないモックの呼び出しに相当します。設定は、設定する値を伴った呼び出しになります。:

```
>>> class Foo:
...     @property
...     def foo(self):
...         return 'something'
...     @foo.setter
...     def foo(self, value):
...         pass
...
>>> with patch('__main__.Foo.foo', new_callable=PropertyMock) as mock_foo:
...     mock_foo.return_value = 'mockity-mock'
...     this_foo = Foo()
...     print(this_foo.foo)
...     this_foo.foo = 6
...
mockity-mock
>>> mock_foo.mock_calls
[call(), call(6)]
```

PropertyMock を直接モックに取り付ける方法は、モックの属性を保存する方法によりうまく動作しません。代わりに、モック型に取り付けてください:

```
>>> m = MagicMock()
>>> p = PropertyMock(return_value=3)
>>> type(m).foo = p
>>> m.foo
3
>>> p.assert_called_once_with()
```

class unittest.mock.**AsyncMock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, ***kwargs*)

An asynchronous version of *Mock*. The *AsyncMock* object will behave so the object is recognized as an async function, and the result of a call is an awaitable.

```
>>> mock = AsyncMock()
>>> asyncio.iscoroutinefunction(mock)
True
>>> inspect.isawaitable(mock())
True
```

The result of `mock()` is an async function which will have the outcome of `side_effect` or `return_value` after it has been awaited:

- if `side_effect` is a function, the async function will return the result of that function,
- if `side_effect` is an exception, the async function will raise the exception,
- if `side_effect` is an iterable, the async function will return the next value of the iterable, however, if the sequence of result is exhausted, `StopAsyncIteration` is raised immediately,
- if `side_effect` is not defined, the async function will return the value defined by `return_value`, hence, by default, the async function returns a new *AsyncMock* object.

Setting the *spec* of a *Mock* or *MagicMock* to an async function will result in a coroutine object being returned after calling.

```
>>> async def async_func(): pass
...
>>> mock = MagicMock(async_func)
>>> mock
<MagicMock spec='function' id='...'>
>>> mock()
<coroutine object AsyncMockMixin._mock_call at ...>
```

Setting the *spec* of a *Mock*, *MagicMock*, or *AsyncMock* to a class with asynchronous and synchronous functions will automatically detect the synchronous functions and set them as *MagicMock* (if the parent mock is *AsyncMock* or *MagicMock*) or *Mock* (if the parent mock is *Mock*). All asynchronous functions will be *AsyncMock*.

```
>>> class ExampleClass:
...     def sync_foo():
```

(次のページに続く)

(前のページからの続き)

```
...         pass
...     async def async_foo():
...         pass
...
>>> a_mock = AsyncMock(ExampleClass)
>>> a_mock.sync_foo
<MagicMock name='mock.sync_foo' id='...'>
>>> a_mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
>>> mock = Mock(ExampleClass)
>>> mock.sync_foo
<Mock name='mock.sync_foo' id='...'>
>>> mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
```

バージョン 3.8 で追加.

assert_awaited()

Assert that the mock was awaited at least once. Note that this is separate from the object having been called, the `await` keyword must be used:

```
>>> mock = AsyncMock()
>>> async def main(coroutine_mock):
...     await coroutine_mock
...
>>> coroutine_mock = mock()
>>> mock.called
True
>>> mock.assert_awaited()
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited.
>>> asyncio.run(main(coroutine_mock))
>>> mock.assert_awaited()
```

assert_awaited_once()

Assert that the mock was awaited exactly once.

```
>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.assert_awaited_once()
>>> asyncio.run(main())
>>> mock.method.assert_awaited_once()
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

assert_awaited_with(**args*, ***kwargs*)

Assert that the last await was with the specified arguments.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_with('foo', bar='bar')
>>> mock.assert_awaited_with('other')
Traceback (most recent call last):
...
AssertionError: expected call not found.
Expected: mock('other')
Actual: mock('foo', bar='bar')
```

assert_awaited_once_with(**args*, ***kwargs*)

Assert that the mock was awaited exactly once and with the specified arguments.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_once_with('foo', bar='bar')
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_once_with('foo', bar='bar')
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

assert_any_await(**args*, ***kwargs*)

Assert the mock has ever been awaited with the specified arguments.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> asyncio.run(main('hello'))
>>> mock.assert_any_await('foo', bar='bar')
>>> mock.assert_any_await('other')
Traceback (most recent call last):
...
AssertionError: mock('other') await not found
```

assert_has_awaits(*calls*, *any_order=False*)

Assert the mock has been awaited with the specified calls. The *await_args_list* list is checked for the awaits.

If *any_order* is false then the awaits must be sequential. There can be extra calls before or after the specified awaits.

If *any_order* is true then the awaits can be in any order, but they must all appear in *await_args_list*.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> calls = [call("foo"), call("bar")]
>>> mock.assert_has_awaits(calls)
Traceback (most recent call last):
...
AssertionError: Awaits not found.
Expected: [call('foo'), call('bar')]
Actual: []
>>> asyncio.run(main('foo'))
>>> asyncio.run(main('bar'))
>>> mock.assert_has_awaits(calls)
```

assert_not_awaited()

Assert that the mock was never awaited.

```
>>> mock = AsyncMock()
>>> mock.assert_not_awaited()
```

reset_mock(**args*, ***kwargs*)

See *Mock.reset_mock()*. Also sets *await_count* to 0, *await_args* to None, and clears the *await_args_list*.

await_count

An integer keeping track of how many times the mock object has been awaited.

```
>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.await_count
1
>>> asyncio.run(main())
>>> mock.await_count
2
```

await_args

This is either `None` (if the mock hasn't been awaited), or the arguments that the mock was last awaited with. Functions the same as *Mock.call_args*.

```
>>> mock = AsyncMock()
>>> async def main(*args):
...     await mock(*args)
...
```

(次のページに続く)

(前のページからの続き)

```
>>> mock.await_args
>>> asyncio.run(main('foo'))
>>> mock.await_args
call('foo')
>>> asyncio.run(main('bar'))
>>> mock.await_args
call('bar')
```

await_args_list

This is a list of all the awaits made to the mock object in sequence (so the length of the list is the number of times it has been awaited). Before any awaits have been made it is an empty list.

```
>>> mock = AsyncMock()
>>> async def main(*args):
...     await mock(*args)
...
>>> mock.await_args_list
[]
>>> asyncio.run(main('foo'))
>>> mock.await_args_list
[call('foo')]
>>> asyncio.run(main('bar'))
>>> mock.await_args_list
[call('foo'), call('bar')]
```

呼び出し

モックオブジェクトは呼び出し可能です。呼び出しの戻り値は *return_value* 属性に設定された値です。デフォルトでは新しいモックオブジェクトを返します。この新しいモックは、属性に最初にアクセスした際に作成されます (明示もしくはモックの呼び出しによって)。そしてそれは保存され、それ以降は同じものが返されます。

呼び出しはオブジェクトとして *call_args* や *call_args_list* に記録されます。

もし *side_effect* が設定されている場合は、その呼び出しが記録された後に呼び出されます。よって、もし side_effect が例外を発生させても、その呼び出しは記録されます。

呼び出された際に例外を発生させるモックを作成するためには、*side_effect* を例外クラスかインスタンスにする方法が最もシンプルです:

```
>>> m = MagicMock(side_effect=IndexError)
>>> m(1, 2, 3)
Traceback (most recent call last):
  ...
IndexError
>>> m.mock_calls
[call(1, 2, 3)]
```

(次のページに続く)

(前のページからの続き)

```
>>> m.side_effect = KeyError('Bang!')
>>> m('two', 'three', 'four')
Traceback (most recent call last):
  ...
KeyError: 'Bang!'
>>> m.mock_calls
[call(1, 2, 3), call('two', 'three', 'four')]
```

もし side_effect が関数だった場合には、その関数の戻り値がモックを呼び出した際の戻り値になります。side_effect 関数には、モックの呼び出し時に与えられた引数と同じ物があたえられます。これにより、入力によって動的に値を返すことができます:

```
>>> def side_effect(value):
...     return value + 1
...
>>> m = MagicMock(side_effect=side_effect)
>>> m(1)
2
>>> m(2)
3
>>> m.mock_calls
[call(1), call(2)]
```

もし、モックにデフォルトの戻り値 (新しいモック) や設定した値を返して欲しい場合は、2 つの方法があります。side_effect の内部で mock.return_value を返すか *DEFAULT* を返します:

```
>>> m = MagicMock()
>>> def side_effect(*args, **kwargs):
...     return m.return_value
...
>>> m.side_effect = side_effect
>>> m.return_value = 3
>>> m()
3
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> m.side_effect = side_effect
>>> m()
3
```

side_effect を削除し、デフォルトの挙動を行うようにするには、side_effect に None を設定します:

```
>>> m = MagicMock(return_value=6)
>>> def side_effect(*args, **kwargs):
...     return 3
...
>>> m.side_effect = side_effect
>>> m()
3
```

(次のページに続く)

(前のページからの続き)

```
>>> m.side_effect = None
>>> m()
6
```

side_effect には、イテレート可能オブジェクトを設定できます。モックが呼び出されるごとに、イテレート可能オブジェクトから戻り値を得ます (イテレート可能オブジェクトが尽きて *StopIteration* が発生するまで):

```
>>> m = MagicMock(side_effect=[1, 2, 3])
>>> m()
1
>>> m()
2
>>> m()
3
>>> m()
Traceback (most recent call last):
  ...
StopIteration
```

もしイテレート可能オブジェクトの要素が例外だった場合には、戻り値として返される代わりに例外が発生します:

```
>>> iterable = (33, ValueError, 66)
>>> m = MagicMock(side_effect=iterable)
>>> m()
33
>>> m()
Traceback (most recent call last):
 ...
ValueError
>>> m()
66
```

属性の削除

モックオブジェクトは要求に応じて属性を生成することで，任意のオブジェクトとして振る舞うことができます。

hasattr() の呼び出しの際に False を返したり，属性にアクセスした際に *AttributeError* を発生させたりしたい場合，spec を用いる方法があります．しかし，この方法は必ずしも便利ではありません．

属性を削除することで，*AttributeError* を発生させ，アクセスを ”妨げる” ようになります．

```
>>> mock = MagicMock()
>>> hasattr(mock, 'm')
True
>>> del mock.m
```

(次のページに続く)

(前のページからの続き)

```
>>> hasattr(mock, 'm')
False
>>> del mock.f
>>> mock.f
Traceback (most recent call last):
    ...
AttributeError: f
```

Mock の名前と name 属性

”name” は *Mock* コンストラクタの引数なので、モックオブジェクトが ”name” 属性を持つことを望む場合、単に生成時にそれを渡すことはできません。2 つの選択肢があります。1 つのオプションは *configure_mock()* を使用することです:

```
>>> mock = MagicMock()
>>> mock.configure_mock(name='my_name')
>>> mock.name
'my_name'
```

より単純なオプションはモックの生成後に単に ”name” 属性をセットすることです:

```
>>> mock = MagicMock()
>>> mock.name = "foo"
```

属性として設定されるモック

属性 (もしくは戻り値) に他のモックを設定した場合、このモックは ” 子” になります。この子に対する呼び出しは、親の *method_calls* や *mock_calls* に記録されます。これは、子のモックを構成し、親にそのモックを設定する際に有用です。また、親に対して設定したすべての子の呼び出しを記録するため、それらの間の順番を確認する際にも有用です:

```
>>> parent = MagicMock()
>>> child1 = MagicMock(return_value=None)
>>> child2 = MagicMock(return_value=None)
>>> parent.child1 = child1
>>> parent.child2 = child2
>>> child1(1)
>>> child2(2)
>>> parent.mock_calls
[call.child1(1), call.child2(2)]
```

モックが名前をもつ場合は、例外的に扱われます。何らかの理由で ” 子守り” が発生してほしくないときに、それを防ぐことができます。

```
>>> mock = MagicMock()
>>> not_a_child = MagicMock(name='not-a-child')
```

(次のページに続く)

(前のページからの続き)

```
>>> mock.attribute = not_a_child
>>> mock.attribute()
<MagicMock name='not-a-child()' id='...'>
>>> mock.mock_calls
[]
```

patch() を用いて作成したモックには、自動的に名前が与えられます。名前を持つモックを設定したい場合には、親に対して *attach_mock()* メソッドを呼び出します:

```
>>> thing1 = object()
>>> thing2 = object()
>>> parent = MagicMock()
>>> with patch('__main__.thing1', return_value=None) as child1:
...     with patch('__main__.thing2', return_value=None) as child2:
...         parent.attach_mock(child1, 'child1')
...         parent.attach_mock(child2, 'child2')
...         child1('one')
...         child2('two')
...
>>> parent.mock_calls
[call.child1('one'), call.child2('two')]
```

26.5.3 patcher

patch デコレータは、その関数のスコープ内でパッチを適用するオブジェクトに対して使用されます。たとえ例外が発生したとしても、パッチは自動的に解除されます。これらすべての機能は文やクラスのデコレータとしても使用できます。

patch

注釈: *patch()* を使うのは簡単です。重要なのは正しい名前空間に対して patch することです。*where to patch* セクションを参照してください。

---

unittest.mock.**patch**(*target*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

patch() は関数デコレータ、クラスデコレータ、コンテキストマネージャーとして利用できます。関数や with 文の body では、*target* は *new* オブジェクトにパッチされます。関数/with 文が終了すると、パッチは元に戻されます。

If *new* is omitted, then the target is replaced with an *AsyncMock* if the patched object is an async function or a *MagicMock* otherwise. If *patch()* is used as a decorator and *new* is omitted, the created mock is passed in as an extra argument to the decorated function. If *patch()* is used as a context manager the created mock is returned by the context manager.

target は `'package.module.ClassName'` の形式の文字列でなければなりません。*target* はインポー

トされ、指定されたオブジェクトが *new* オブジェクトに置き換えられます。なので、*target* は *patch()* を呼び出した環境からインポート可能でなければなりません。target がインポートされるのは、デコレートした時ではなく、デコレートされた関数が呼び出された時です。

patch が *MagicMock* を生成する場合、*spec* と *spec_set* キーワード引数は *MagicMock* に渡されます。

加えて、`spec=True` もしくは `spec_set=True` を渡すことで、モック対象のオブジェクトが spec/spec_set に渡されます。

new_callable allows you to specify a different class, or callable object, that will be called to create the *new* object. By default *AsyncMock* is used for async functions and *MagicMock* for the rest.

より強力な *spec* の形は *autospec* です。`autospec=True` を指定した場合、mock は置換対象となるオブジェクトから得られる spec で生成されます。mock のすべての属性もまた置換対象となるオブジェクトの属性に応じた spec を持ちます。mock されたメソッドや関数は引数をチェックし、間違ったシグネチャで呼び出された場合は *TypeError* を発生させます。クラスを置き換える mock の場合、その戻り値 (つまりインスタンス) はそのクラスと同じ spec を持ちます。*create_autospec()* 関数と *autospec* **を使う** を参照してください。

置換対象ではなく任意のオブジェクトを spec として使うために、`autospec=True` の代わりに、`autospec=some_object` と指定することができます。

By default *patch()* will fail to replace attributes that don't exist. If you pass in `create=True`, and the attribute doesn't exist, patch will create the attribute for you when the patched function is called, and delete it again after the patched function has exited. This is useful for writing tests against attributes that your production code creates at runtime. It is off by default because it can be dangerous. With it switched on you can write passing tests against APIs that don't actually exist!

注釈: バージョン 3.5 で変更: モジュールのビルトインにパッチを当てようとしているなら、`create=True` を渡す必要はありません。それはデフォルトで追加されます。

---

patch は `TestCase` のクラスデコレータとして利用できます。この場合そのクラスの各テストメソッドをデコレートします。これによりテストメソッドが同じ patch を共有している場合に退屈なコードを減らすことができます。*patch()* は `patch.TEST_PREFIX` で始まるメソッド名のメソッドを探します。デフォルトではこれは `'test'` で、*unittest* がテストを探す方法とマッチしています。`patch.TEST_PREFIX` を設定することで異なる prefix を指定することもできます。

patch は with 文を使ってコンテキストマネージャーとして使うこともできます。その場合パッチは with 文のブロック内でのみ適用されます。"as" を使って、"as" に続いて指定した変数にパッチされたオブジェクトが代入されます。これは *patch()* が mock オブジェクトを生成するときに便利です。

patch() は任意のキーワード引数を受け取り、それを *Mock* (あるいは *new_callable*) の生成時に渡します。

異なるユースケースのために、`patch.dict(...)`, `patch.multiple(...)`, `patch.object(...)` が用意されています。

patch() を関数デコレータとして利用し、mock を生成してそれをデコレートされた関数に渡します:

```
>>> @patch('__main__.SomeClass')
... def function(normal_argument, mock_class):
...     print(mock_class is SomeClass)
...
>>> function(None)
True
```

クラスをパッチするとそのクラスを *MagicMock* の **インスタンス** に置き換えます。テスト中のコードからそのクラスがインスタンス化される場合、mock の *return_value* が利用されます。

クラスが複数回インスタンス化される場合、*side_effect* を利用して毎回新しい mock を返すようにできます。もしくは、*return_value* に、何でも好きなものを設定できます。

パッチしたクラスの **インスタンス** のメソッドの戻り値をカスタマイズしたい場合、return_value に対して設定しなければなりません。例:

```
>>> class Class:
...     def method(self):
...         pass
...
>>> with patch('__main__.Class') as MockClass:
...     instance = MockClass.return_value
...     instance.method.return_value = 'foo'
...     assert Class() is instance
...     assert Class().method() == 'foo'
...
```

spec か *spec_set* を指定し、*patch()* が **クラス** を置換するとき、生成された mock の戻り値は同じ spec を持ちます。:

```
>>> Original = Class
>>> patcher = patch('__main__.Class', spec=True)
>>> MockClass = patcher.start()
>>> instance = MockClass()
>>> assert isinstance(instance, Original)
>>> patcher.stop()
```

new_callable 引数は、デフォルトの *MagicMock* の代わりに別のクラスをモックとして生成したい場合に便利です。例えば、*NonCallableMock* を利用したい場合:

```
>>> thing = object()
>>> with patch('__main__.thing', new_callable=NonCallableMock) as mock_thing:
...     assert thing is mock_thing
...     thing()
...
Traceback (most recent call last):
  ...
TypeError: 'NonCallableMock' object is not callable
```

別のユースケースとしては、オブジェクトを *io.StringIO* インスタンスで置き換えることがあります:

```
>>> from io import StringIO
>>> def foo():
...     print('Something')
...
>>> @patch('sys.stdout', new_callable=StringIO)
... def test(mock_stdout):
...     foo()
...     assert mock_stdout.getvalue() == 'Something\n'
...
>>> test()
```

patch() に mock を生成させる場合、たいてい最初に必要なことはその mock をセットアップすることです。幾らかのセットアップは patch の呼び出しから行うことができます。任意のキーワード引数が、生成された mock の属性に設定されます:

```
>>> patcher = patch('__main__.thing', first='one', second='two')
>>> mock_thing = patcher.start()
>>> mock_thing.first
'one'
>>> mock_thing.second
'two'
```

生成された mock の属性のさらに属性、*return_value* *side_effect* などもセットアップできます。これらはキーワード引数のシンタックスでは直接指定できませんが、それらをキーとする辞書を ** を使って *patch()* に渡すことができます:

```
>>> config = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> patcher = patch('__main__.thing', **config)
>>> mock_thing = patcher.start()
>>> mock_thing.method()
3
>>> mock_thing.other()
Traceback (most recent call last):
  ...
KeyError
```

By default, attempting to patch a function in a module (or a method or an attribute in a class) that does not exist will fail with *AttributeError*:

```
>>> @patch('sys.non_existing_attribute', 42)
... def test():
...     assert sys.non_existing_attribute == 42
...
>>> test()
Traceback (most recent call last):
  ...
AttributeError: <module 'sys' (built-in)> does not have the attribute 'non_existing'
```

but adding `create=True` in the call to *patch()* will make the previous example work as expected:

```
>>> @patch('sys.non_existing_attribute', 42, create=True)
... def test(mock_stdout):
...     assert sys.non_existing_attribute == 42
...
>>> test()
```

バージョン 3.8 で変更: *patch()* now returns an *AsyncMock* if the target is an async function.

patch.object

patch.object(*target*, *attribute*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

オブジェクト (*target*) の指定された名前のメンバー (*attribute*) を mock オブジェクトでパッチします。

patch.object() はデコレータ、クラスデコレータ、コンテキストマネージャーとして利用できます。引数の *new*, *spec*, *create*, *spec_set*, *autospec*, *new_callable* は *patch()* と同じ意味を持ちます。*patch()* と同じく、*patch.object()* も mock を生成するための任意のキーワード引数を受け取ります。

クラスデコレータとして利用する場合、*patch.object()* は patch.TEST_PREFIX にしたがってラップするメソッドを選択します。

patch.object() の呼び出しには 3 引数の形式と 2 引数の形式があります。3 引数の場合、patch 対象のオブジェクト、属性名、その属性を置き換えるオブジェクトを取ります。

2 引数の形式では、置き換えるオブジェクトを省略し、生成された mock がデコレート対象となる関数に追加の引数として渡されます:

```
>>> @patch.object(SomeClass, 'class_method')
... def test(mock_method):
...     SomeClass.class_method(3)
...     mock_method.assert_called_with(3)
...
>>> test()
```

spec, *create* やその他の *patch.object()* の引数は *patch()* の引数と同じ意味を持ちます。

patch.dict

patch.dict(*in_dict*, *values=()*, *clear=False*, ***kwargs*)

辞書や辞書のようなオブジェクトにパッチし、テスト後に元の状態に戻します。

in_dict は辞書やその他のマップ型のコンテナです。マップ型の場合、最低限 get, set, del 操作とキーに対するイテレートをサポートしている必要があります。

in_dict に辞書を指定する文字列を渡した場合、それをインポートして取得します。

values は対象の辞書にセットする値を含む、辞書か (key, value) ペアの iterable です。

clear が true なら、新しい値が設定される前に辞書がクリアされます。

patch.dict() はまた、任意のキーワード引数を受け取って辞書に設定します。

バージョン 3.8 で変更: *patch.dict()* now returns the patched dictionary when used as a context manager.

patch.dict() can be used as a context manager, decorator or class decorator:

```
>>> foo = {}
>>> @patch.dict(foo, {'newkey': 'newvalue'})
... def test():
...     assert foo == {'newkey': 'newvalue'}
>>> test()
>>> assert foo == {}
```

When used as a class decorator *patch.dict()* honours `patch.TEST_PREFIX` (default to `'test'`) for choosing which methods to wrap:

```
>>> import os
>>> import unittest
>>> from unittest.mock import patch
>>> @patch.dict('os.environ', {'newkey': 'newvalue'})
... class TestSample(unittest.TestCase):
...     def test_sample(self):
...         self.assertEqual(os.environ['newkey'], 'newvalue')
```

If you want to use a different prefix for your test, you can inform the patchers of the different prefix by setting `patch.TEST_PREFIX`. For more details about how to change the value of see *TEST_PREFIX*.

patch.dict() を使うと、辞書にメンバーを追加するか、または単にテストが辞書を変更して、その後テストが終了した時にその辞書が確実に復元されるようにすることができます。

```
>>> foo = {}
>>> with patch.dict(foo, {'newkey': 'newvalue'}) as patched_foo:
...     assert foo == {'newkey': 'newvalue'}
...     assert patched_foo == {'newkey': 'newvalue'}
...     # You can add, update or delete keys of foo (or patched_foo, it's the same dict)
...     patched_foo['spam'] = 'eggs'
...
>>> assert foo == {}
>>> assert patched_foo == {}
```

```
>>> import os
>>> with patch.dict('os.environ', {'newkey': 'newvalue'}):
...     print(os.environ['newkey'])
...
newvalue
>>> assert 'newkey' not in os.environ
```

patch.dict() を呼び出すときにキーワード引数を使って辞書に値をセットすることができます。

```
>>> mymodule = MagicMock()
>>> mymodule.function.return_value = 'fish'
>>> with patch.dict('sys.modules', mymodule=mymodule):
...     import mymodule
...     mymodule.function('some', 'args')
...
'fish'
```

patch.dict() は辞書ではなくても辞書ライクなオブジェクトに対して使うことができます。対象となるオブジェクトは最低限、get, set, del そしてイテレーションかメンバーのテストのどちらかをサポートする必要があります。これはマジックメソッドの `__getitem__()`, `__setitem__()`, `__delitem__()` そして `__iter__()` か `__contains__()` のどちらかに対応します。

```
>>> class Container:
...     def __init__(self):
...         self.values = {}
...     def __getitem__(self, name):
...         return self.values[name]
...     def __setitem__(self, name, value):
...         self.values[name] = value
...     def __delitem__(self, name):
...         del self.values[name]
...     def __iter__(self):
...         return iter(self.values)
...
>>> thing = Container()
>>> thing['one'] = 1
>>> with patch.dict(thing, one=2, two=3):
...     assert thing['one'] == 2
...     assert thing['two'] == 3
...
>>> assert thing['one'] == 1
>>> assert list(thing) == ['one']
```

patch.multiple

patch.**multiple**(*target*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

1 回の呼び出しで複数のパッチを実行します。パッチ対象のオブジェクト (あるいはそのオブジェクトをインポートするための文字列) と、パッチ用のキーワード引数を取ります:

```
with patch.multiple(settings, FIRST_PATCH='one', SECOND_PATCH='two'):
    ...
```

patch.multiple() に mock を生成させたい場合、キーワード引数の値に *DEFAULT* を指定します。この場合生成されたモックはデコレート対象の関数にキーワード引数として渡され、コンテキストマネージャーとして利用された場合は辞書として返します。

patch.multiple() はデコレータ、クラスデコレータ、コンテキストマネージャーとして使えます。引

数の *spec*, *spec_set*, *create*, *autospec*, *new_callable* は *patch()* の引数と同じ意味を持ちます。これらの引数は *patch.multiple()* によって適用される **すべての** パッチに対して適用されます。

クラスデコレータとして利用する場合、*patch.multiple()* は patch.TEST_PREFIX にしたがってラップするメソッドを選択します。

patch.multiple() に mock を生成させたい場合、キーワード引数の値に *DEFAULT* を指定します。この場合生成されたモックはデコレート対象の関数にキーワード引数として渡されます。:

```
>>> thing = object()
>>> other = object()

>>> @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(thing, other):
...     assert isinstance(thing, MagicMock)
...     assert isinstance(other, MagicMock)
...
>>> test_function()
```

patch.multiple() は他の patch デコレータとネストして利用できますが、キーワード引数は *patch()* によって作られる通常の引数の **後ろ** で受け取る必要があります:

```
>>> @patch('sys.exit')
... @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(mock_exit, other, thing):
...     assert 'other' in repr(other)
...     assert 'thing' in repr(thing)
...     assert 'exit' in repr(mock_exit)
...
>>> test_function()
```

patch.multiple() がコンテキストマネージャーとして利用される場合、コンテキストマネージャーが返す値は、名前がキーで値が生成された mock となる辞書です:

```
>>> with patch.multiple('__main__', thing=DEFAULT, other=DEFAULT) as values:
...     assert 'other' in repr(values['other'])
...     assert 'thing' in repr(values['thing'])
...     assert values['thing'] is thing
...     assert values['other'] is other
...
```

patch のメソッド: start と stop

すべての patcher は start() と stop() メソッドを持ちます。これを使うと、with 文やデコレータをネストさせずに、setUp メソッドで複数のパッチをシンプルに適用させることができます。

これらのメソッドを使うには、*patch()*, *patch.object*, *patch.dict()* を通常の関数のように呼び出して、戻り値の patcher オブジェクトを保持します。その start() メソッドでパッチを適用し、stop() メソッドで巻き戻すことができます。

patch() に mock を生成させる場合、patcher.start の呼び出しの戻り値として mock を受け取れます。:

```
>>> patcher = patch('package.module.ClassName')
>>> from package import module
>>> original = module.ClassName
>>> new_mock = patcher.start()
>>> assert module.ClassName is not original
>>> assert module.ClassName is new_mock
>>> patcher.stop()
>>> assert module.ClassName is original
>>> assert module.ClassName is not new_mock
```

この方式の典型的なユースケースは、TestCase の setUp メソッドで複数のパッチを行うことです:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         self.patcher1 = patch('package.module.Class1')
...         self.patcher2 = patch('package.module.Class2')
...         self.MockClass1 = self.patcher1.start()
...         self.MockClass2 = self.patcher2.start()
...
...     def tearDown(self):
...         self.patcher1.stop()
...         self.patcher2.stop()
...
...     def test_something(self):
...         assert package.module.Class1 is self.MockClass1
...         assert package.module.Class2 is self.MockClass2
...
>>> MyTest('test_something').run()
```

ご用心: この方式を使う場合、必ず stop メソッドを呼び出してパッチが解除する必要があります。setUp の中で例外が発生した場合 tearDown が呼び出されないので、これは意外に面倒です。*unittest.TestCase.addCleanup()* を使うと簡単にできます:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         patcher = patch('package.module.Class')
...         self.MockClass = patcher.start()
...         self.addCleanup(patcher.stop)
...
...     def test_something(self):
...         assert package.module.Class is self.MockClass
...
```

この方式を使うと patcher オブジェクトの参照を維持する必要がなくなるという特典も付きます。

patch.stopall() を利用してすべての start されたパッチを stop することもできます。

```
patch.stopall()
```

すべての有効なパッチを stop します。start で開始したパッチしか stop しません。

ビルトインをパッチする

モジュール内の任意のビルトインに対してパッチを当てることができます。以下の例はビルトインの *ord()* を修正します:

```
>>> @patch('__main__.ord')
... def test(mock_ord):
...     mock_ord.return_value = 101
...     print(ord('c'))
...
>>> test()
101
```

TEST_PREFIX

すべての patcher はクラスデコレータとして利用できます。この場合、そのクラスのすべてのテストメソッドをラップします。patcher は 'test' で始まる名前のメソッドをテストメソッドだと認識します。これはデフォルトで *unittest.TestLoader* がテストメソッドを見つける方法と同じです。

他の prefix を使う事もできます。その場合、patcher にその prefix を patch.TEST_PREFIX に設定することで教えることができます:

```
>>> patch.TEST_PREFIX = 'foo'
>>> value = 3
>>>
>>> @patch('__main__.value', 'not three')
... class Thing:
...     def foo_one(self):
...         print(value)
...     def foo_two(self):
...         print(value)
...
>>>
>>> Thing().foo_one()
not three
>>> Thing().foo_two()
not three
>>> value
3
```

patch デコレータをネストする

複数のパッチを行いたい場合、シンプルにデコレータを重ねることができます。

次のパターンのように patch デコレータを重ねることができます:

```
>>> @patch.object(SomeClass, 'class_method')
... @patch.object(SomeClass, 'static_method')
... def test(mock1, mock2):
...     assert SomeClass.static_method is mock1
...     assert SomeClass.class_method is mock2
...     SomeClass.static_method('foo')
...     SomeClass.class_method('bar')
...     return mock1, mock2
...
>>> mock1, mock2 = test()
>>> mock1.assert_called_once_with('foo')
>>> mock2.assert_called_once_with('bar')
```

デコレータは下から上へと適用されることに注意してください。これは Python がデコレータを適用する標準的な方法です。テスト関数に渡される生成された mock の順番もこれに一致します。

どこにパッチするか

patch() は (一時的に) ある **名前** が参照しているオブジェクトを別のものに変更することで適用されます。任意のオブジェクトには、それを参照するたくさんの名前が存在しえます。なので、必ずテスト対象のシステムが使っている名前に対して patch しなければなりません。

基本的な原則は、オブジェクトが **ルックアップ** されるところにパッチすることです。その場所はオブジェクトが定義されたところとは限りません。これを説明するためにいくつかの例を挙げます。

次のような構造を持ったプロジェクトをテストしようとしていると仮定してください:

```
a.py
    -> Defines SomeClass

b.py
    -> from a import SomeClass
    -> some_function instantiates SomeClass
```

いま、some_function をテストしようとしていて、そのために SomeClass を *patch()* を使って mock しようとしています。モジュール b をインポートした時点で、b は SomeClass を a からインポートしています。この状態で a.SomeClass を *patch()* を使って mock out してもテストには影響しません。モジュール b はすでに **本物の** SomeClass への参照を持っていて、パッチの影響を受けないからです。

重要なのは、SomeClass が使われている (もしくはルックアップされている) 場所にパッチすることです。この場合、some_function はモジュール b の中にインポートされた SomeClass をルックアップしています。なのでパッチは次のようにしなければなりません:

```
@patch('b.SomeClass')
```

ですが、別のシナリオとして、module b が `from a import SomeClass` ではなく `import a` をしていて、`some_function` が `a.SomeClass` を利用していたとします。どちらのインポートも一般的なものです。この場合、パッチしたいクラスはそのモジュールからルックアップされているので、`a.SomeClass` をパッチする必要があります:

```
@patch('a.SomeClass')
```

デスクリプタやプロキシオブジェクトにパッチする

patch と *patch.object* はどちらも デスクリプタ (クラスメソッド、static メソッド、プロパティ) を正しく patch できます。デスクリプタに patch する場合、インスタンスではなく *class* にパッチする必要があります。これらはまた **幾らかの** 属性アクセスをプロキシするオブジェクト、例えば django の setttings オブジェクト に対しても機能します。

26.5.4 MagicMock と magic method のサポート

magick method をモックする

Mock は、"magic method" とも呼ばれる、Python のプロトコルメソッドに対する mock もサポートしています。これによりコンテナやその他 Python のプロトコルを実装しているオブジェクトを mock することが可能になります。

magic method は通常のメソッドとはルックアップ方法が異なるので[*2], magic method のサポートは特別に実装されています。そのため、サポートされているのは特定の magic method のみです。**ほとんど** すべてのメソッドをサポートしていますが、足りないものを見つけたら私達に教えてください。

magic method を mock するには、対象の method に対して関数や mock のインスタンスをセットします。もし関数を使う場合、それは第一引数に `self` を取る **必要があります**[*3].

```
>>> def __str__(self):
...     return 'fooble'
...
>>> mock = Mock()
>>> mock.__str__ = __str__
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__str__ = Mock()
```

(次のページに続く)

[*2] Magic method はインスタンスではなくクラスからルックアップされるはずです。Python のバージョンによってこのルールが適用されるかどうかに違いがあります。サポートされているプロトコルメソッドは、サポートされているすべての Python のバージョンで動作するはずです。

[*3] 関数はクラスまで hook しますが、各 `Mock` インスタンス間の独立性は保たれます。

(前のページからの続き)

```
>>> mock.__str__.return_value = 'fooble'
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__iter__ = Mock(return_value=iter([]))
>>> list(mock)
[]
```

ユースケースの 1 つは with 文の中でコンテキストマネージャーとして使われるオブジェクトを mock することです。

```
>>> mock = Mock()
>>> mock.__enter__ = Mock(return_value='foo')
>>> mock.__exit__ = Mock(return_value=False)
>>> with mock as m:
...     assert m == 'foo'
...
>>> mock.__enter__.assert_called_with()
>>> mock.__exit__.assert_called_with(None, None, None)
```

magic method の呼び出しは *method_calls* に含まれませんが、*mock_calls* には記録されます。

注釈: mock を生成するのに *spec* キーワード引数を使った場合、spec に含まれない magic method を設定しようとすると *AttributeError* が発生します。

サポートしている magic method の完全なリスト:

- __hash__, __sizeof__, __repr__, __str__
- __dir__, __format__, __subclasses__
- __round__, __floor__, __trunc__ と __ceil__
- 比較: __lt__, __gt__, __le__, __ge__, __eq__, __ne__
- コンテナメソッド: __getitem__, __setitem__, __delitem__, __contains__, __len__, __iter__, __reversed__, __missing__
- Context manager: __enter__, __exit__, __aenter__ and __aexit__
- 単項算術メソッド: __neg__, __pos__, __invert__
- 算術メソッド (右辺や in-place のものも含む): __add__, __sub__, __mul__, __matmul__, __div__, __truediv__, __floordiv__, __mod__, __divmod__, __lshift__, __rshift__, __and__, __xor__, __or__, __pow__
- 算術変換メソッド: __complex__, __int__, __float__, __index__

- デスクリプタメソッド: `__get__`, `__set__`, `__delete__`
- pickle: `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__`, `__setstate__`
- File system path representation: `__fspath__`
- Asynchronous iteration methods: `__aiter__` and `__anext__`

バージョン 3.8 で変更: Added support for *os.PathLike.__fspath__()*.

バージョン 3.8 で変更: Added support for `__aenter__`, `__aexit__`, `__aiter__` and `__anext__`.

以下のメソッドは存在しますが、mock が利用している、動的に設定不可能、その他の問題が発生する可能性があるなどの理由で **サポートされていません**:

- `__getattr__`, `__setattr__`, `__init__`, `__new__`
- `__prepare__`, `__instancecheck__`, `__subclasscheck__`, `__del__`

Magic Mock

MagicMock 系のクラスは 2 種類あります: *MagicMock* と *NonCallableMagicMock* です。

class unittest.mock.**MagicMock**(**args*, ***kw*)

MagicMock は *Mock* のサブクラスで、ほとんどの magic method のデフォルト実装を提供しています。自分で magic method を構成しなくても MagicMock を使うことができます。

コンストラクタの引数は *Mock* と同じ意味を持っています。

spec か *spec_set* 引数を利用した場合、spec に存在する magic method **のみ** が生成されます。

class unittest.mock.**NonCallableMagicMock**(**args*, ***kw*)

callable でないバージョンの *MagicMock*

コンストラクタの引数は *MagicMock* と同じ意味を持ちますが、*return_value* と *side_effect* は callable でない mock では意味を持ちません。

MagicMock が magic method をセットアップするので、あとは通常の方法で構成したり利用したりできます:

```
>>> mock = MagicMock()
>>> mock[3] = 'fish'
>>> mock.__setitem__.assert_called_with(3, 'fish')
>>> mock.__getitem__.return_value = 'result'
>>> mock[2]
'result'
```

デフォルトでは、多くのプロトコルメソッドは特定の型の戻り値を要求されます。それらのメソッドではその型のデフォルトの戻り値がデフォルトの戻り値として設定されているので、戻り値に興味が有る場合以外は何もしなくても利用可能です。デフォルトの値を変更したい場合は手動で戻り値を設定可能です。

メソッドとそのデフォルト値:

- __lt__: NotImplemented
- __gt__: NotImplemented
- __le__: NotImplemented
- __ge__: NotImplemented
- __int__: 1
- __contains__: False
- __len__: 0
- __iter__: iter([])
- __exit__: False
- __aexit__: False
- __complex__: 1j
- __float__: 1.0
- __bool__: True
- __index__: 1
- __hash__: mock のデフォルトの hash
- __str__: mock のデフォルトの str
- __sizeof__: mock のデフォルトの sizeof

例えば:

```
>>> mock = MagicMock()
>>> int(mock)
1
>>> len(mock)
0
>>> list(mock)
[]
>>> object() in mock
False
```

2 つの比較メソッド `__eq__()` と `__ne__()` は特別です。それらは、もし戻り値として何か別のものを返すように変更していなければ、*side_effect* 属性を使用して、同一性に基づくデフォルトの同値比較を行います:

```
>>> MagicMock() == 3
False
>>> MagicMock() != 3
True
>>> mock = MagicMock()
>>> mock.__eq__.return_value = True
```

(次のページに続く)

(前のページからの続き)

```
>>> mock == 3
True
```

MagickMock.__iter__() の return_value は任意の iterable で、イテレータである必要はありません:

```
>>> mock = MagicMock()
>>> mock.__iter__.return_value = ['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
```

return_value が iterator であった場合、最初のイテレートでその iterator を消費してしまい、2 回目以降のイテレートの結果が空になってしまいます:

```
>>> mock.__iter__.return_value = iter(['a', 'b', 'c'])
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
[]
```

MagicMock はいくつかの曖昧であったり時代遅れなものをのぞいて、対応している magic method を事前にセットアップします。自動でセットアップされていないものも必要なら手動でセットアップすることができます。

MagicMock がサポートしているもののデフォルトではセットアップしない magic method:

- __subclasses__
- __dir__
- __format__
- __get__, __set__, __delete__
- __reversed__, __missing__
- __reduce__, __reduce_ex__, __getinitargs__, __getnewargs__, __getstate__, __setstate__
- __getformat__, __setformat__

26.5.5 ヘルパー

sentinel

unittest.mock.**sentinel**

sentinel オブジェクトはテストで必要なユニークなオブジェクトを簡単に提供します。

属性はアクセス時にオンデマンドで生成されます。同じ属性に複数回アクセスすると必ず同じオブジェクトが返されます。返されるオブジェクトは、テスト失敗のメッセージがわかりやすくなるように気が

利いた repr を持ちます。

バージョン 3.7 で変更: The `sentinel` attributes now preserve their identity when they are *copied* or *pickled*.

特定のオブジェクトが他のメソッドに引数として渡されることをテストしたり、返されることをテストしたい場合があります。このテストのために名前がついた sentinel オブジェクトを作るのが一般的です。*sentinel* はこのようなオブジェクトを生成し、同一性をテストするのに便利な方法を提供します。

次の例では、`method` が `sentinel.some_object` を返すようにモンキーパッチしています:

```
>>> real = ProductionClass()
>>> real.method = Mock(name="method")
>>> real.method.return_value = sentinel.some_object
>>> result = real.method()
>>> assert result is sentinel.some_object
>>> sentinel.some_object
sentinel.some_object
```

DEFAULT

unittest.mock.**DEFAULT**

DEFAULT オブジェクトは事前に生成された sentinel (実際には `sentinel.DEFAULT`) オブジェクトです。*side_effect* 関数が、通常の戻り値を使うことを示すために使います。

call

unittest.mock.**call**(**args*, ***kwargs*)

call() は *call_args*, *call_args_list*, *mock_calls*, *method_calls* と比較してより下端に assert できるようにするためのヘルパーオブジェクトです。*call()* は *assert_has_calls()* と組み合わせて使うこともできます。

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, a='foo', b='bar')
>>> m()
>>> m.call_args_list == [call(1, 2, a='foo', b='bar'), call()]
True
```

call.**call_list**()

複数回の呼び出しを表す call オブジェクトに対して、*call_list()* はすべての途中の呼び出しと最終の呼び出しを含むリストを返します。

`call_list` は特に ”chained call” に対して assert するのに便利です。”chained call” は 1 行のコードにおける複数の呼び出しです。この結果は mock の *mock_calls* に複数の call エントリとして格納されます。この call のシーケンスを手動で構築するのは退屈な作業になります。

call_list() は同じ chained call からその call のシーケンスを構築することができます:

```
>>> m = MagicMock()
>>> m(1).method(arg='foo').other('bar')(2.0)
<MagicMock name='mock().method().other()()' id='...'>
>>> kall = call(1).method(arg='foo').other('bar')(2.0)
>>> kall.call_list()
[call(1),
 call().method(arg='foo'),
 call().method().other('bar'),
 call().method().other()(2.0)]
>>> m.mock_calls == kall.call_list()
True
```

call オブジェクトは、どう構築されたかによって、(位置引数、キーワード引数) のタプルか、(名前、位置引数、キーワード引数) のタプルになります。自分で call オブジェクトを構築するときはこれを意識する必要はありませんが、*Mock.call_args*, *Mock.call_args_list*, *Mock.mock_calls* 属性の中の call オブジェクトを解析して個々の引数を解析することができます。

Mock.call_args と *Mock.call_args_list* の中の call オブジェクトは (位置引数, キーワード引数) のタプルで、*Mock.mock_calls* の中の call オブジェクトや自分で構築したオブジェクトは (名前, 位置引数, キーワード引数) のタプルになります。

call オブジェクトの ” タプル性” を使って、より複雑な内省とアサートを行うために各引数を取り出しすことができます。位置引数はタプル (位置引数が存在しない場合は空のタプル) で、キーワード引数は辞書になります:

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, 3, arg='one', arg2='two')
>>> kall = m.call_args
>>> kall.args
(1, 2, 3)
>>> kall.kwargs
{'arg': 'one', 'arg2': 'two'}
>>> kall.args is kall[0]
True
>>> kall.kwargs is kall[1]
True
```

```
>>> m = MagicMock()
>>> m.foo(4, 5, 6, arg='two', arg2='three')
<MagicMock name='mock.foo()' id='...'>
>>> kall = m.mock_calls[0]
>>> name, args, kwargs = kall
>>> name
'foo'
>>> args
(4, 5, 6)
>>> kwargs
{'arg': 'two', 'arg2': 'three'}
>>> name is m.mock_calls[0][0]
True
```

create_autospec

unittest.mock.**create_autospec**(*spec*, *spec_set=False*, *instance=False*, ***kwargs*)

他のオブジェクトを spec として利用して mock オブジェクトを作ります。mock の属性も、*spec* オブジェクトの該当する属性を spec として利用します。

mock された関数やメソッドは、正しいシグネチャで呼び出されたことを確認するために引数をチェックします。

spec_set が `True` のとき、spec オブジェクトにない属性をセットしようとすると *AttributeError* を発生させます。

spec にクラスが指定された場合、mock の戻り値 (そのクラスのインスタンス) は同じ spec を持ちます。`instance=True` を指定すると、インスタンスオブジェクトの spec としてクラスを利用できます。返される mock は、モックのインスタンスが callable な場合にだけ callable となります。

create_autospec() は任意のキーワード引数を受け取り、生成する mock のコンストラクタに渡します。

create_autospec() や、*patch()* の *autospec* 引数で autospec を使うサンプルは *autospec* **を使う** を参照してください。

バージョン 3.8 で変更: *create_autospec()* now returns an *AsyncMock* if the target is an async function.

ANY

unittest.mock.**ANY**

mock の呼び出しのうち **幾つか** の引数に対して assert したいけれども、それ以外の引数は気にしない、あるいは *call_args* から個別に取り出してより高度な assert を行いたい場合があります。

特定の引数を無視するために、**すべて** と等しくなるオブジェクトを使うことができます。そうすると、*assert_called_with()* と *assert_called_once_with()* は、実際の引数が何であったかに関わらず成功します。

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar=object())
>>> mock.assert_called_once_with('foo', bar=ANY)
```

mock_calls などの call list との比較に *ANY* を使うこともできます:

```
>>> m = MagicMock(return_value=None)
>>> m(1)
>>> m(1, 2)
>>> m(object())
>>> m.mock_calls == [call(1), call(1, 2), ANY]
True
```

FILTER_DIR

unittest.mock.FILTER_DIR

FILTER_DIR は mock オブジェクトが *dir()* に何を返すかを制御するためのモジュールレベル変数です。(Python 2.6 以上でのみ有効) デフォルトは True で、以下に示すフィルタリングを行い、有用なメンバーだけを表示します。このフィルタリングが嫌な場合や、何かの診断のためにフィルタリングを切りたい場合は、mock.FILTER_DIR = False と設定してください。

フィルタリングが有効な場合、dir(some_mock) は有用な属性だけを表示し、また通常は表示されない動的に生成される属性も表示します。mock が *spec* を使って (もちろん *autospec* でも) 生成された場合、元のオブジェクトのすべての属性が、まだアクセスされていなかったとしても、表示されます。

```
>>> dir(Mock())
['assert_any_call',
 'assert_called',
 'assert_called_once',
 'assert_called_once_with',
 'assert_called_with',
 'assert_has_calls',
 'assert_not_called',
 'attach_mock',
 ...
>>> from urllib import request
>>> dir(Mock(spec=request))
['AbstractBasicAuthHandler',
 'AbstractDigestAuthHandler',
 'AbstractHTTPHandler',
 'BaseHandler',
 ...
```

多くのあまり有用ではない (mock 対象のものではなく、*Mock* 自身のプライベートな) 属性は、アンダースコアと 2 つのアンダースコアで prefix された属性は *Mock* に対して *dir()* した結果からフィルタリングされます。この動作が嫌な場合は、モジュールレベルの *FILTER_DIR* スイッチを設定することでフィルターを切ることができます。

```
>>> from unittest import mock
>>> mock.FILTER_DIR = False
>>> dir(mock.Mock())
['_NonCallableMock__get_return_value',
 '_NonCallableMock__get_side_effect',
 '_NonCallableMock__return_value_doc',
 '_NonCallableMock__set_return_value',
 '_NonCallableMock__set_side_effect',
 '__call__',
 '__class__',
 ...
```

mock.FILTER_DIR によるフィルタリングをバイパスしたい場合、var(my_mock) (インスタンスメンバー)、dir(type(my_mock)) (型メンバー) を代わりに使うことができます。

mock_open

unittest.mock.**mock_open**(*mock=None*, *read_data=None*)

open() の利用を置き換えるための mock を作るヘルパー関数。*open()* を直接呼んだりコンテキストマネージャーとして利用する場合に使うことができます。

mock 引数は構成する mock オブジェクトです。None (デフォルト) なら、通常のファイルハンドルと同じ属性やメソッドに API が制限された *MagicMock* が生成されます。

read_data は、ファイルハンドルの read(), *readline()*, そして *readlines()* のメソッドが返す文字列です。これらのメソッドを呼び出すと、読み出し終わるまで *read_data* からデータが読み出されます。これらモックのメソッドはとても単純化されています: *mock* が呼ばれるたびに *read_data* は先頭に巻き戻されます。テストコードに与えるデータをさらにコントロールするには自分自身でモックをカスタマイズする必要があります。それでも不十分な場合は、PyPI にあるインメモリファイルシステムパッケージのうちのどれかを使えば、テストのための本物のファイルシステムが得られるでしょう。

バージョン 3.4 で変更: *readline()* および *readlines()* のサポートが追加されました。read() のモックは、個々の呼び出しで *read_data* を返すのではなく、それを消費するように変わりました。

バージョン 3.5 で変更: *read_data* は *mock* を呼び出す度に毎回リセットされるようになりました。

バージョン 3.8 で変更: Added __iter__() to implementation so that iteration (such as in for loops) correctly consumes *read_data*.

open() をコンテキストマネージャーとして使う方法は、ファイルが必ず適切に閉じられるようにする素晴らしい方法で、今では一般的になっています:

```
with open('/some/path', 'w') as f:
    f.write('something')
```

問題は、*open()* をモックアウトしたところで、コンテキストマネージャーが使われる (__enter__() と __exit__() が呼ばれる) のはその **戻り値** だということです。

コンテキストマネージャーを *MagicMock* でモックするのは一般的かつ面倒なので、ヘルパー関数を用意しています。:

```
>>> m = mock_open()
>>> with patch('__main__.open', m):
...     with open('foo', 'w') as h:
...         h.write('some stuff')
...
>>> m.mock_calls
[call('foo', 'w'),
 call().__enter__(),
 call().write('some stuff'),
 call().__exit__(None, None, None)]
>>> m.assert_called_once_with('foo', 'w')
>>> handle = m()
>>> handle.write.assert_called_once_with('some stuff')
```

ファイルの読み込みをモックする例:

```
>>> with patch('__main__.open', mock_open(read_data='bibble')) as m:
...     with open('foo') as h:
...         result = h.read()
...
>>> m.assert_called_once_with('foo')
>>> assert result == 'bibble'
```

autospec を使う

autospec は mock の spec 機能を基盤にしています。autospec は mock の API を元のオブジェクト (spec) に制限しますが、再帰的に適用される (lazy に実装されている) ので、mock の属性も spec の属性と同じ API だけを持つようになります。さらに、mock された関数/メソッドは元と同じシグネチャを持ち、正しくない引数で呼び出されると *TypeError* を発生させます。

autospec の動作について説明する前に、それが必要となる背景から説明していきます。

Mock は非常に強力で柔軟なオブジェクトですが、テスト対象のシステムをモックアウトするときに 2 つの欠点があります。1 つ目の欠点は *Mock* の API に関したもので、もう一つは mock オブジェクトを使う場合のもっと一般的な問題です。

まず *Mock* 独自の問題から解説します。*Mock* は便利な 2 つのメソッド、*assert_called_with()* と *assert_called_once_with()* を持っています。

```
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
Traceback (most recent call last):
 ...
AssertionError: Expected 'mock' to be called once. Called 2 times.
```

mock が属性をオンデマンドに自動生成し、それを任意の引数で呼び出せるため、それらの assert メソッドのいずれかをミススペルするとその assert の効果が消えてしまいます:

```
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assret_called_once_with(4, 5, 6)
```

typo のために、テストは不正確に、かつ暗黙に成功してしまいます。

2 つ目の問題はもっと一般的なものです。なにかのコードをリファクタし、メンバの名前を変更したとします。**古い** *API* を利用したコードに対するテストが、mock を利用しているとするとテストは通り続けます。このため、コードが壊れていてもテストがすべて通ってしまう可能性があります。

各ユニットが互いにどのように接続されるかをテストしない場合、依然としてテストで見つけることができるバグの余地が多く残っています。

mock はこの問題に対処するために spec と呼ばれる機能を提供しています。何かクラスかインスタンスを

spec として mock に渡すと、実際のクラスに存在する属性にしか、mock に対してもアクセスできなくなります:

```
>>> from urllib import request
>>> mock = Mock(spec=request.Request)
>>> mock.assret_called_with
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assret_called_with'
```

spec はその mock 自体にしか適用されません。なので、同じ問題がその mock のすべてのメソッドに対して発生します:

```
>>> mock.has_data()
<mock.Mock object at 0x...>
>>> mock.has_data.assret_called_with()
```

autospec はこの問題を解決します。*patch()* か *patch.object()* に autospec=True を渡すか、*create_autospec()* 関数を使って spec をもとに mock を作ることができます。*patch()* の引数に autospec=True を渡した場合、置換対象のオブジェクトが spec オブジェクトとして利用されます。spec は遅延処理される (mock の属性にアクセスされた時に spec が生成される) ので、非常に複雑だったり深くネストしたオブジェクト (例えばモジュールをインポートするモジュールをインポートするモジュール) に対しても大きなパフォーマンスの問題なしに autospec を使うことができます。

autospec の利用例:

```
>>> from urllib import request
>>> patcher = patch('__main__.request', autospec=True)
>>> mock_request = patcher.start()
>>> request is mock_request
True
>>> mock_request.Request
<MagicMock name='request.Request' spec='Request' id='...'>
```

request.Request が spec を持っているのが分かります。request.Request のコンストラクタは 2 つの引数を持っています (片方は *self* です)。コンストラクタを間違って呼び出した時に何が起こるでしょうか:

```
>>> req = request.Request()
Traceback (most recent call last):
  ...
TypeError: <lambda>() takes at least 2 arguments (1 given)
```

spec はクラスがインスタンス化されたとき (つまり spec が適用された mock の戻り値) にも適用されます:

```
>>> req = request.Request('foo')
>>> req
<NonCallableMagicMock name='request.Request()' spec='Request' id='...'>
```

Request オブジェクトは callable ではないので、request.Request の mock から返されたインスタンスの mock は callable ではなくなります。spec があれば、assert のミススペルは正しいエラーを発生させます:

```
>>> req.add_header('spam', 'eggs')
<MagicMock name='request.Request().add_header()' id='...'>
>>> req.add_header.assret_called_with
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assret_called_with'
>>> req.add_header.assert_called_with('spam', 'eggs')
```

多くの場合、単に既存の *patch()* 呼び出しに autospec=True を加えるだけで、ミススペルや API 変更に伴うバグから守られます。

patch() を経由する以外にも、*create_autospec()* を使って *autospec* が適用された mock を直接作る方法もあります:

```
>>> from urllib import request
>>> mock_request = create_autospec(request)
>>> mock_request.Request('foo', 'bar')
<NonCallableMagicMock name='mock.Request()' spec='Request' id='...'>
```

とはいえ、autospec には注意しなければならない点や制限があり、そのためデフォルトでは無効になっています。spec オブジェクトでどんな属性が使えるかどうかを調べるために、autospec は spec オブジェクトをイントロスペクト (実際に属性にアクセスする) 必要があります。mock の属性を利用するとき、水面下で元のオブジェクトに対しても同じ属性の探索が行われます。spec したオブジェクトのどれかがコードを実行するプロパティやデスクリプタを持っている場合、autospec は正しく動きません。もしくは、イントロスペクションしても安全なようにオブジェクトを設計するのがよいでしょう[*4]。

より重大な問題は、インスタンス属性が __init__() で生成され、クラスには全く存在しない場合です。*autospec* は動的に生成される属性については知ることができず、API を検出できる属性だけに制限してしまいます。:

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

この問題を解決するためにいくつかの方法があります。一番簡単な、ただし一番面倒でないとは限らない方法は、必要とされる属性を mock が生成された後に設定することです。*autospec* は属性を参照することを禁止しますが、設定することは禁止していません:

[*4] これはクラスやすでにインスタンス化されたオブジェクトにだけ当てはまります。mock されたクラスを呼び出して mock インスタンスを作っても、実際のオブジェクトのインスタンスは生成されません。mock は属性を - *dir()* を呼び出して - 検索するだけです。

```
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a = 33
...
```

spec と *autospec* にはよりアグレッシブなバージョンがあり、存在しない属性への設定も禁止します。これはコードが正しい属性にのみ代入することを保証したいときに便利ですが、もちろん先ほどの方法も制限されてしまいます:

```
>>> with patch('__main__.Something', autospec=True, spec_set=True):
...   thing = Something()
...   thing.a = 33
...
Traceback (most recent call last):
 ...
AttributeError: Mock object has no attribute 'a'
```

この問題を解決するベストな方法は、`__init__()` で初期化されるインスタンスメンバに対する初期値をクラス属性として追加することかもしれません。`__init__()` メソッドの中でデフォルトの属性を代入しているだけなら、それをクラス属性にする (この属性はもちろんインスタンス間で共有されます) 方が速くなるのもメリットです。例:

```
class Something:
    a = 33
```

クラス属性を使ってもまた別の問題があります。メンバーのデフォルト値に `None` を利用し、後から別の型のオブジェクトを代入するのは比較的よくあるパターンです。spec として `None` を使うと **すべての** 属性やメソッドへのアクセスも許されなくなるので使い物になりません。`None` を spec にすることが有用な場面は **決して** なく、おそらくそのメンバーは他の何かの型のメンバーになることを示すので、autospec は `None` に設定されているメンバーには spec を使いません。その属性は通常の mock (MagicMocks) になります。

```
>>> class Something:
...     member = None
...
>>> mock = create_autospec(Something)
>>> mock.member.foo.bar.baz()
<MagicMock name='mock.member.foo.bar.baz()' id='...'>
```

すでに利用されているクラスにデフォルト値属性を追加するのが嫌な場合は、他の選択肢もあります。選択肢の 1 つは、クラスではなくインスタンスを spec に使うことです。別の選択肢は、実際のクラスのサブクラスを作り、実際に利用されている方に影響を与えずにデフォルト値属性を追加することです。どちらの方法も spec として代替オブジェクトを利用することが必要です。*patch()* はこれをサポートしていて、*autospec* 引数に代替オブジェクトを渡すことができます:

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
```

(次のページに続く)

(前のページからの続き)

```
>>> class SomethingForTest(Something):
...   a = 33
...
>>> p = patch('__main__.Something', autospec=SomethingForTest)
>>> mock = p.start()
>>> mock.a
<NonCallableMagicMock name='Something.a' spec='int' id='...'>
```

Sealing mocks

unittest.mock.**seal**(*mock*)

Seal will disable the automatic creation of mocks when accessing an attribute of the mock being sealed or any of its attributes that are already mocks recursively.

If a mock instance with a name or a spec is assigned to an attribute it won't be considered in the sealing chain. This allows one to prevent seal from fixing part of the mock object.

```
>>> mock = Mock()
>>> mock.submock.attribute1 = 2
>>> mock.not_submock = mock.Mock(name="sample_name")
>>> seal(mock)
>>> mock.new_attribute  # This will raise AttributeError.
>>> mock.submock.attribute2  # This will raise AttributeError.
>>> mock.not_submock.attribute2  # This won't raise.
```

バージョン 3.7 で追加.

26.6 unittest.mock --- 入門

バージョン 3.3 で追加.

26.6.1 Mock を使う

Mock のパッチ用メソッド

一般的な *Mock* の使い方の中には次のものがあります:

- メソッドにパッチを当てる
- オブジェクトに対するメソッド呼び出しを記録する

システムの他の部分からメソッドが正しい引数で呼び出されたかどうかを確認するために、そのオブジェクトのメソッドを置き換えることができます:

```
>>> real = SomeClass()
>>> real.method = MagicMock(name='method')
```

(次のページに続く)

(前のページからの続き)

```
>>> real.method(3, 4, 5, key='value')
<MagicMock name='method()' id='...'>
```

モック (上の例では real.method) が利用された場合、どう使われたかを assert できるようにする属性やメソッドがモックにあります。

注釈: この例のような場合、たいてい *Mock* と *MagicMock* は互換です。MagicMock の方が強力なので、デフォルトではこちらを使うといいでしょう。

モックが呼び出されると、その *called* 属性が True に設定されます。そして *assert_called_with()* や *assert_called_once_with()* メソッドを使ってそのメソッドが正しい引数で呼び出されたかどうかをチェックできます。

次の例では ProductionClass().method が something メソッドを呼び出したことをテストしています:

```
>>> class ProductionClass:
...     def method(self):
...         self.something(1, 2, 3)
...     def something(self, a, b, c):
...         pass
...
>>> real = ProductionClass()
>>> real.something = MagicMock()
>>> real.method()
>>> real.something.assert_called_once_with(1, 2, 3)
```

オブジェクトのメソッド呼び出しに対するモック

上の例ではオブジェクトのメソッドに対して直接パッチを当てて、それが正しく呼び出されていたかどうかをテストしていました。もう一つのよくあるユースケースが、モックをメソッド (またはテスト対象のシステムのどこか) に渡して、それが正しく利用されたかどうかをチェックする方法です。

次の例で、ProductionClass は closer メソッドを持っています。このメソッドは渡されたオブジェクトの close メソッドを呼び出します。

```
>>> class ProductionClass:
...     def closer(self, something):
...         something.close()
...
```

ですからこれををテストするには、close メソッドを持ったオブジェクトを渡して、それが正しく呼び出されたかどうかをテストしなければなりません。

```
>>> real = ProductionClass()
>>> mock = Mock()
```

(次のページに続く)

(前のページからの続き)

```
>>> real.closer(mock)
>>> mock.close.assert_called_with()
```

モックに *close* メソッドを持たせるために何か準備する必要はありません。*close* メソッドにアクセスすると自動的にそれが作られます。なので、もし *close* が呼び出されなかったとしてもテスト時に生成されるのですが、*assert_called_with()* が failure 例外を発生させます。

クラスをモックする

他のよくあるユースケースが、テスト対象のコードによってインスタンス化されているクラスをモックに置き換えることです。クラスに patch すると、そのクラスがモックに置き換えられます。インスタンスは **クラスを呼び出した時に** 作られます。なので、モックの戻り値を使うことで、「モックのインスタンス」にアクセスできます。

次の例では、some_function という関数が Foo をインスタンス化し、その method を呼び出しています。*patch()* を呼び出すと Foo クラスをモックに置き換えます。Foo のインスタンスはモックを呼び出して作られるので、モックの *return_value* を変更することでカスタマイズできます。:

```
>>> def some_function():
...     instance = module.Foo()
...     return instance.method()
...
>>> with patch('module.Foo') as mock:
...     instance = mock.return_value
...     instance.method.return_value = 'the result'
...     result = some_function()
...     assert result == 'the result'
```

モックに名前をつける

モックに名前をつけると便利なことがあります。その名前はモックを repr したときに表示されるので、モックがテスト失敗のメッセージ内に現れた時に便利です。また、モックの名前はそのモックの属性やメソッドにも伝播します:

```
>>> mock = MagicMock(name='foo')
>>> mock
<MagicMock name='foo' id='...'>
>>> mock.method
<MagicMock name='foo.method' id='...'>
```

全ての呼び出しのトラッキング

メソッドの複数回の呼び出しをトラックしたいことがあります。*mock_calls* 属性は、そのモックの子属性やさらにその子孫に対する呼び出しすべてを記録しています。

```
>>> mock = MagicMock()
>>> mock.method()
<MagicMock name='mock.method()' id='...'>
>>> mock.attribute.method(10, x=53)
<MagicMock name='mock.attribute.method()' id='...'>
>>> mock.mock_calls
[call.method(), call.attribute.method(10, x=53)]
```

mock_calls に対して assert すると、予期していないメソッド呼び出しがあったときにその assert が失敗します。これはあるメソッド呼び出しが期待通りに実行されたかどうかだけでなく、その呼び出し順序や期待した以外の呼び出しが起こらなかったことまでテストできるので便利です:

mock_calls と比較するリストを作るために *call* オブジェクトを利用できます:

```
>>> expected = [call.method(), call.attribute.method(10, x=53)]
>>> mock.mock_calls == expected
True
```

However, parameters to calls that return mocks are not recorded, which means it is not possible to track nested calls where the parameters used to create ancestors are important:

```
>>> m = Mock()
>>> m.factory(important=True).deliver()
<Mock name='mock.factory().deliver()' id='...'>
>>> m.mock_calls[-1] == call.factory(important=False).deliver()
True
```

戻り値や属性を設定する

モックオブジェクトに戻り値を設定するのはとっても簡単です:

```
>>> mock = Mock()
>>> mock.return_value = 3
>>> mock()
3
```

もちろん同じことがモックのメソッドに対しても行えます:

```
>>> mock = Mock()
>>> mock.method.return_value = 3
>>> mock.method()
3
```

コンストラクターで戻り値を設定することもできます:

```
>>> mock = Mock(return_value=3)
>>> mock()
3
```

モックに属性を設定したかったら、普通に設定するだけです:

```
>>> mock = Mock()
>>> mock.x = 3
>>> mock.x
3
```

mock.connection.cursor().execute("SELECT 1") のような複雑なケースでモックを使いたい場合もあります。この呼出があるリストを返すようにしたい場合、このネストした呼び出しを構成しなければなりません。

call を使って "chained call" 内の呼び出しを構成して、assert で使うことができます:

```
>>> mock = Mock()
>>> cursor = mock.connection.cursor.return_value
>>> cursor.execute.return_value = ['foo']
>>> mock.connection.cursor().execute("SELECT 1")
['foo']
>>> expected = call.connection.cursor().execute("SELECT 1").call_list()
>>> mock.mock_calls
[call.connection.cursor(), call.connection.cursor().execute('SELECT 1')]
>>> mock.mock_calls == expected
True
```

call object を chained call を表す list にするために .call_list() を使います。

モックから例外を発生させる

side_effect という便利な属性があります。この属性に例外クラスやそのインスタンスを設定すると、モックが呼ばれた時にその例外を発生させます。

```
>>> mock = Mock(side_effect=Exception('Boom!'))
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

side_effect の関数と iterable

side_effect には関数や iterable を設定することもできます。side_effect に iterable を設定するユースケースは、そのモックが複数回呼び出され、そのたびに違う値を返したい場合です。side_effect に iterable を設定すると、そのモックに対するすべての呼び出しは iterable の次の値を返します:

```
>>> mock = MagicMock(side_effect=[4, 5, 6])
>>> mock()
4
>>> mock()
5
>>> mock()
6
```

より高度なユースケースとして、mock が呼び出された時の引数によって戻り値を変化させたい場合は、side_effect に関数を設定することができます。その関数は mock と同じ引数で呼び出されます。その関数の戻り値がそのモック呼び出しの戻り値になります:

```
>>> vals = {(1, 2): 1, (2, 3): 2}
>>> def side_effect(*args):
...     return vals[args]
...
>>> mock = MagicMock(side_effect=side_effect)
>>> mock(1, 2)
1
>>> mock(2, 3)
2
```

非同期イテレータをモックする

Python 3.8 以降、AsyncMock と MagicMock は "___aiter___" を通じて:ref:*async-iterators* のモックをサポートしています。"*___aiter___*" の:attr:*~Mock.return_value* 属性はイテレーションに使用される戻り値を設定するために使用できます。

```
>>> mock = MagicMock()  # AsyncMock also works here
>>> mock.__aiter__.return_value = [1, 2, 3]
>>> async def main():
...     return [i async for i in mock]
...
>>> asyncio.run(main())
[1, 2, 3]
```

非同期コンテキストマネージャをモックする

Python 3.8 以降、AsyncMock と MagicMock は “___aenter___“と “___aexit___“を通じて:ref:*async-context-managers*‘のモックをサポートしています。デフォルトでは、“*___aenter___*‘ と “___aexit___“は非同期関数を返す “AsyncMock“インスタンスです。

```
>>> class AsyncContextManager:
...     async def __aenter__(self):
...         return self
...     async def __aexit__(self, exc_type, exc, tb):
...         pass
...
>>> mock_instance = MagicMock(AsyncContextManager())  # AsyncMock also works here
>>> async def main():
...     async with mock_instance as result:
...         pass
...
>>> asyncio.run(main())
>>> mock_instance.__aenter__.assert_awaited_once()
>>> mock_instance.__aexit__.assert_awaited_once()
```

既存のオブジェクトから Mock を作る

mock を使いすぎることの問題の一つは、テストが実際のコードではなく mock の実装をテストするようになってしまうことです。some_method というメソッドを実装したクラスがあるとします。他のクラスをテストするときに、some_method を提供する mock を使います。最初のクラスをリファクタリングして some_method がなくなった時、コードは壊れているのにテストは通る状態になってしまいます

Mock は *spec* というキーワード引数で mock の定義となるオブジェクトを指定できます。定義オブジェクトに存在しないメソッドや属性にアクセスすると AttributeError を発生させます。定義となるクラスの実装を変更した場合、テストの中でそのクラスをインスタンス化させなくても、テストを失敗させる事ができます。

```
>>> mock = Mock(spec=SomeClass)
>>> mock.old_method()
Traceback (most recent call last):
   ...
AttributeError: object has no attribute 'old_method'
```

また、定義を指定することで、パラメータが位置引数と名前付き引数のどちらで渡されたかに関わらず、モックへの呼び出しをよりスマートに照合することができます。

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, 3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(a=1, b=2, c=3)
```

このよりスマートなマッチングをモックのメソッド呼び出しにも適用したい場合は、:ref:‘auto-speccing <auto-speccing>‘を使用します。

任意の属性の参照だけでなく代入も禁止するより強い定義を利用したい場合は、*spec* の代わりに *spec_set* を使います。

26.6.2 patch デコレータ

注釈: *patch()* では探索される名前空間内のオブジェクトにパッチをあてることが重要です。通常は単純ですが、クイックガイドには *where-to-patch* を読んでください。

テストの中でクラス属性やモジュール属性、例えば組み込み関数や、テスト対象モジュールにあるインスタンス化されるクラスに対してパッチしたいことがあります。モジュールやクラスは実際はグローバルなので、パッチするときは必ずテスト後にパッチを解除しないと、そのパッチが永続化されて他のテストに影響を与え、解析しにくい問題になります。

mock はこのために 3 つの便利なデコレータを提供しています: *patch()*, *patch.object()*, *patch.dict()* です。patch はパッチ対象を指定する package.module.Class.attribute の形式の文字列を引数に取ります。オプションでその属性 (やクラスなど) を置き換えるオブジェクトを渡すことができます。'patch.object' はオブジェクトとパッチしたい属性名、それにオプションで置き換えるオブジェクトを受け取ります。

patch.object:

```
>>> original = SomeClass.attribute
>>> @patch.object(SomeClass, 'attribute', sentinel.attribute)
... def test():
...     assert SomeClass.attribute == sentinel.attribute
...
>>> test()
>>> assert SomeClass.attribute == original

>>> @patch('package.module.attribute', sentinel.attribute)
... def test():
...     from package.module import attribute
...     assert attribute is sentinel.attribute
...
>>> test()
```

モジュール (*builtins* を含む) をパッチしようとする場合、*patch.object()* の代わりに *patch()* を使用してください:

```
>>> mock = MagicMock(return_value=sentinel.file_handle)
>>> with patch('builtins.open', mock):
...     handle = open('filename', 'r')
...
>>> mock.assert_called_with('filename', 'r')
>>> assert handle == sentinel.file_handle, "incorrect file handle returned"
```

モジュール名は必要に応じて package.module のようにドットを含むことができます:

```
>>> @patch('package.module.ClassName.attribute', sentinel.attribute)
... def test():
...     from package.module import ClassName
...     assert ClassName.attribute == sentinel.attribute
...
>>> test()
```

テストメソッド自体をデコレートするのは良いパターンです:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'attribute', sentinel.attribute)
...     def test_something(self):
...         self.assertEqual(SomeClass.attribute, sentinel.attribute)
...
>>> original = SomeClass.attribute
>>> MyTest('test_something').test_something()
>>> assert SomeClass.attribute == original
```

Mock を使ってパッチしたい場合は、*patch()* を 1 引数で (または *patch.object()* を 2 引数で) 使うことができます。mock が自動で生成され、テスト関数/メソッドに渡されます:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'static_method')
...     def test_something(self, mock_method):
...         SomeClass.static_method()
...         mock_method.assert_called_with()
...
>>> MyTest('test_something').test_something()
```

次のパターンのように patch デコレータを重ねることができます:

```
>>> class MyTest(unittest.TestCase):
...     @patch('package.module.ClassName1')
...     @patch('package.module.ClassName2')
...     def test_something(self, MockClass2, MockClass1):
...         self.assertIs(package.module.ClassName1, MockClass1)
...         self.assertIs(package.module.ClassName2, MockClass2)
...
>>> MyTest('test_something').test_something()
```

patch デコレータをネストした際、モックは (デコレータを適用する *Python* の通常の) 順に適用されます。つまり引数は下から上の順になり、よって上記の例では `test_module.ClassName2` が先になります。

また、*patch.dict()* を使うと、スコープ内だけで辞書に値を設定し、テスト終了時には元の状態に復元されます:

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
```

(次のページに続く)

(前のページからの続き)

```
...
>>> assert foo == original
```

patch, patch.object, patch.dict は全てコンテキストマネージャーとしても利用できます。

patch() に mock を生成させる場合、その参照を with 文の as を使って受け取れます:

```
>>> class ProductionClass:
...     def method(self):
...         pass
...
>>> with patch.object(ProductionClass, 'method') as mock_method:
...     mock_method.return_value = None
...     real = ProductionClass()
...     real.method(1, 2, 3)
...
>>> mock_method.assert_called_with(1, 2, 3)
```

他の方法として、patch, patch.object, patch.dict はクラスデコレータとしても利用できます。その場合、”test” で始まる全てのメソッドにデコレータを適用するのと同じになります。

26.6.3 さらなる例

より高度なシナリオを想定した例をあげていきます。

chained call をモックする

chained call を mock するのは、一度 *return_value* 属性を理解してしまえば簡単です。mock が最初に呼ばれた時や、呼び出される前に return_value を参照した場合、新しい *Mock* が生成されます。

つまり、戻り値のオブジェクトがどう利用されたかは、return_value mock を調べれば分かります:

```
>>> mock = Mock()
>>> mock().foo(a=2, b=3)
<Mock name='mock().foo()' id='...'>
>>> mock.return_value.foo.assert_called_with(a=2, b=3)
```

これをもとに、mock を構成して chained call に対する assert を行うのは簡単です。もちろん、元のコードを最初からもっとテストしやすく書くという選択肢もありますが...

では、例として次のようなコードがあるとします:

```
>>> class Something:
...     def __init__(self):
...         self.backend = BackendProvider()
...     def method(self):
```

(次のページに続く)

(前のページからの続き)

```
...         response = self.backend.get_endpoint('foobar').create_call('spam', 'eggs').start_
↪call()
...         # more code
```

BackendProvider はすでに十分テストされているとします。method() をどうテストしましょうか？ 特に、response オブジェクトを使う # more code の部分のコードをテストしたいとします。

この chained call はインスタンス属性を起点にしているので、Something インスタンスの backend 属性に対してモンキーパッチすることができます。今回の場合だと、最後の start_call の呼び出しが返す値にだけ興味があるので、あまり多くの構成は必要ありません。このメソッドが返すのが 'file-like' オブジェクトだとしましょう。そうすると、response オブジェクトは組み込みの *open()* を spec として利用できます。

これをするために、backend のモックとしてモックインスタンスを作成し、それに対するモックの response オブジェクトを作成します。最終的な start_call の返り値として response をセットすると、このようにすることができます:

```
mock_backend.get_endpoint.return_value.create_call.return_value.start_call.return_value = mock_
↪response
```

これより少し良いやり方として、返り値を直接セットする *configure_mock()* メソッドを使用して次のようにすることができます:

```
>>> something = Something()
>>> mock_response = Mock(spec=open)
>>> mock_backend = Mock()
>>> config = {'get_endpoint.return_value.create_call.return_value.start_call.return_value':␣
↪mock_response}
>>> mock_backend.configure_mock(**config)
```

これらによって「モックの backend」をその場で monkey patch して、実際の呼び出しを行うことができます:

```
>>> something.backend = mock_backend
>>> something.method()
```

mock_calls を使用すると、チェーンされた呼び出しを単一のアサーションでチェックすることができます。チェーンされた呼び出しは、1 行のコードの中で行われる複数の呼び出しです。したがって、mock_calls には複数のエントリーがあるでしょう。*call.call_list()* を使用することで、この呼び出しのリストを作成することができます:

```
>>> chained = call.get_endpoint('foobar').create_call('spam', 'eggs').start_call()
>>> call_list = chained.call_list()
>>> assert mock_backend.mock_calls == call_list
```

部分的なモック

テストによっては、*datetime.date.today()* の呼び出しを既知の date を返すようにモックしたいと思うことがありました。しかし、テスト対象のコードが新しい date オブジェクトを生成するのを妨げたくはありません。不運にも、*datetime.date* は C で書かれています。したがって、静的な `date.today()` メソッドを単にモンキーパッチすることはできませんでした。

私はこれを行う単純な方法を見つけました。それは、date クラスをモックで事実上ラップして、しかしコンストラクタの呼び出しを実際のクラスへと素通りさせる (そして、実際のインスタンスを返す) というものです。

ここで、テスト対象のモジュール中の `date` クラスをモックするために *patch decorator* が使用されています。そして、モックの date クラスの `side_effect` 属性は、本物の date を返すラムダ関数にセットされます。モックの date クラスが呼ばれる時、`side_effect` によって本物の date が構築されて返されます。:

```
>>> from datetime import date
>>> with patch('mymodule.date') as mock_date:
...     mock_date.today.return_value = date(2010, 10, 8)
...     mock_date.side_effect = lambda *args, **kw: date(*args, **kw)
...
...     assert mymodule.date.today() == date(2010, 10, 8)
...     assert mymodule.date(2009, 6, 8) == date(2009, 6, 8)
```

グローバルに:class:*datetime.date*‘**にパッチするのではなく、それを使用するモジュールの中の** *“date‘* にパッチしていることに留意してください。**どこにパッチするか**. を参照してください。

When `date.today()` is called a known date is returned, but calls to the `date(...)` constructor still return normal dates. Without this you can find yourself having to calculate an expected result using exactly the same algorithm as the code under test, which is a classic testing anti-pattern.

date コンストラクタの呼び出しは、“mock_date“の属性 (“call_count“とその仲間) に記録され、テストに役立つこともあります。

日付や他の組み込みクラスのモックに対処する別の方法は このブログエントリ. で議論されています。

ジェネレータ method をモックする

Python のジェネレータは、反復処理された時に一連の値を返す :keyword:‘yield‘文を使用した関数やメソッドです [#]_。

ジェネレータメソッド/関数はジェネレータオブジェクトを返すために呼び出されます。そしてこのジェネレータオブジェクトが反復処理されます。イテレーションのためのプロトコルメソッドは :meth:‘~container.__iter__‘なので、:class:‘MagicMock‘を使ってこれをモックすることができます。

ここでは”iter” メソッドをジェネレータとして実装したクラスの例を紹介します:

```
>>> class Foo:
...     def iter(self):
...         for i in [1, 2, 3]:
...             yield i
...
```

(次のページに続く)

(前のページからの続き)

```
>>> foo = Foo()
>>> list(foo.iter())
[1, 2, 3]
```

このクラス、特に ”iter” メソッドをどのようにモックするのでしょうか。

(:class:‘list‘の呼び出しの中での暗黙的な) イテレーションから返される値を設定するには、“foo.iter()“の呼び出しで返されるオブジェクトを設定する必要がある。

```
>>> mock_foo = MagicMock()
>>> mock_foo.iter.return_value = iter([1, 2, 3])
>>> list(mock_foo.iter())
[1, 2, 3]
```

同じパッチを全てのメソッドに適用する

If you want several patches in place for multiple test methods the obvious way is to apply the patch decorators to every method. This can feel like unnecessary repetition. For Python 2.6 or more recent you can use *patch()* (in all its various forms) as a class decorator. This applies the patches to all test methods on the class. A test method is identified by methods whose names start with `test`:

```
>>> @patch('mymodule.SomeClass')
... class MyTest(unittest.TestCase):
...
...     def test_one(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def test_two(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def not_a_test(self):
...         return 'something'
...
>>> MyTest('test_one').test_one()
>>> MyTest('test_two').test_two()
>>> MyTest('test_two').not_a_test()
'something'
```

パッチを管理する別の方法として、*patch* **のメソッド**: *start* **と** *stop*. を使用する方法があります。これらを使うと、パッチを `setUp` と “tearDown“メソッドに移すことができます。

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         self.patcher = patch('mymodule.foo')
...         self.mock_foo = self.patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
```

(次のページに続く)

(前のページからの続き)

```
...
...     def tearDown(self):
...         self.patcher.stop()
...
>>> MyTest('test_foo').run()
```

この方式を使う場合、必ず stop メソッドを呼び出してパッチが解除する必要があります。setUp の中で例外が発生した場合 tearDown が呼び出されないので、これは意外に面倒です。*unittest.TestCase.addCleanup()* を使うと簡単にできます:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         patcher = patch('mymodule.foo')
...         self.addCleanup(patcher.stop)
...         self.mock_foo = patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
>>> MyTest('test_foo').run()
```

Unbound メソッドをモックする

Whilst writing tests today I needed to patch an *unbound method* (patching the method on the class rather than on the instance). I needed self to be passed in as the first argument because I want to make asserts about which objects were calling this particular method. The issue is that you can't patch with a mock for this, because if you replace an unbound method with a mock it doesn't become a bound method when fetched from the instance, and so it doesn't get self passed in. The workaround is to patch the unbound method with a real function instead. The *patch()* decorator makes it so simple to patch out methods with a mock that having to create a real function becomes a nuisance.

If you pass `autospec=True` to patch then it does the patching with a *real* function object. This function object has the same signature as the one it is replacing, but delegates to a mock under the hood. You still get your mock auto-created in exactly the same way as before. What it means though, is that if you use it to patch out an unbound method on a class the mocked function will be turned into a bound method if it is fetched from an instance. It will have `self` passed in as the first argument, which is exactly what I wanted:

```
>>> class Foo:
...   def foo(self):
...     pass
...
>>> with patch.object(Foo, 'foo', autospec=True) as mock_foo:
...   mock_foo.return_value = 'foo'
...   foo = Foo()
...   foo.foo()
...
```

(次のページに続く)

(前のページからの続き)

```
'foo'
>>> mock_foo.assert_called_once_with(foo)
```

autospec=True を使用しない場合、束縛されていないメソッドは代わりに Mock インスタンスでパッチされ、“self“で呼び出されることはありません。

モックで複数回の呼び出しをチェックする

モックには、モックオブジェクトがどのように使用されるかをアサートするための素晴らしい API があります。

```
>>> mock = Mock()
>>> mock.foo_bar.return_value = None
>>> mock.foo_bar('baz', spam='eggs')
>>> mock.foo_bar.assert_called_with('baz', spam='eggs')
```

もしモックが一度しか呼ばれていないのであれば、call_count`が 1 であることも保証する:meth:`assert_called_once_with メソッドを使うことができます。

```
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
>>> mock.foo_bar()
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
Traceback (most recent call last):
    ...
AssertionError: Expected to be called once. Called 2 times.
```

Both assert_called_with and assert_called_once_with make assertions about the *most recent* call. If your mock is going to be called several times, and you want to make assertions about *all* those calls you can use *call_args_list*:

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, 3)
>>> mock(4, 5, 6)
>>> mock()
>>> mock.call_args_list
[call(1, 2, 3), call(4, 5, 6), call()]
```

call ヘルパーを使えば、これらの呼び出しについて簡単にアサートすることができます。予想される呼び出しのリストを作成し、それを “call_args_list“と比較することができます。これは “call_args_list“の repr と驚くほど似ています:

```
>>> expected = [call(1, 2, 3), call(4, 5, 6), call()]
>>> mock.call_args_list == expected
True
```

ミュータブルな引数に対処する

稀ではありますが、問題となることがあるのはミュータブルな引数でモックが呼び出された場合です。`call_args` や “call_args_list“ は引数への*参照*を保存します。もしテスト対象のコードによって引数が変更されてしまうと、モックが呼び出された時の値をアサートすることはできなくなります。

この問題を示すサンプルコードを示します。

```
def frob(val):
    pass

def grob(val):
    "First frob and then clear val"
    frob(val)
    val.clear()
```

`grob` が “frob“を正しい引数で呼び出していることをテストしようとすると何が起こるか見てみましょう:

```
>>> with patch('mymodule.frob') as mock_frob:
...     val = {6}
...     mymodule.grob(val)
...
>>> val
set()
>>> mock_frob.assert_called_with({6})
Traceback (most recent call last):
    ...
AssertionError: Expected: (({6},), {})
Called with: ((set(),), {})
```

一つの選択肢は、あなたが渡す引数をモックがコピーすることです。この方法は、オブジェクト id の同一性に依存したアサーションを行う場合に問題を引き起こす可能性があります。

Here's one solution that uses the `side_effect` functionality. If you provide a `side_effect` function for a mock then `side_effect` will be called with the same args as the mock. This gives us an opportunity to copy the arguments and store them for later assertions. In this example I'm using *another* mock to store the arguments so that I can use the mock methods for doing the assertion. Again a helper function sets this up for me.

```
>>> from copy import deepcopy
>>> from unittest.mock import Mock, patch, DEFAULT
>>> def copy_call_args(mock):
...     new_mock = Mock()
...     def side_effect(*args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         new_mock(*args, **kwargs)
...         return DEFAULT
...     mock.side_effect = side_effect
...     return new_mock
...
```

(次のページに続く)

(前のページからの続き)

```
>>> with patch('mymodule.frob') as mock_frob:
...     new_mock = copy_call_args(mock_frob)
...     val = {6}
...     mymodule.grob(val)
...
>>> new_mock.assert_called_with({6})
>>> new_mock.call_args
call({6})
```

copy_call_args は、テストで呼び出されるモックとともに呼び出されます。この関数はアサーションで使用するための新しいモックを返します。side_effect 関数は引数のコピーを作成し、そのコピーを使って new_mock を呼び出します。

注釈: もしモックが一度しか使われないのであれば、呼び出された時点で引数をチェックする簡単な方法があります。単純に “side_effect“関数の中でチェックを行えばいいのです。

```
>>> def side_effect(arg):
...     assert arg == {6}
...
>>> mock = Mock(side_effect=side_effect)
>>> mock({6})
>>> mock(set())
Traceback (most recent call last):
    ...
AssertionError
```

別の方法としては、(*copy.deepcopy()* を使用して) 引数をコピーする、*Mock* や *MagicMock* のサブクラスを作成する方法があります。以下に実装例を示します。

```
>>> from copy import deepcopy
>>> class CopyingMock(MagicMock):
...     def __call__(self, /, *args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         return super().__call__(*args, **kwargs)
...
>>> c = CopyingMock(return_value=None)
>>> arg = set()
>>> c(arg)
>>> arg.add(1)
>>> c.assert_called_with(set())
>>> c.assert_called_with(arg)
Traceback (most recent call last):
    ...
AssertionError: Expected call: mock({1})
Actual call: mock(set())
>>> c.foo
<CopyingMock name='mock.foo' id='...'>
```

When you subclass `Mock` or `MagicMock` all dynamically created attributes, and the `return_value` will use your subclass automatically. That means all children of a `CopyingMock` will also have the type `CopyingMock`.

patch をネストする

Using patch as a context manager is nice, but if you do multiple patches you can end up with nested with statements indenting further and further to the right:

```
>>> class MyTest(unittest.TestCase):
...
...     def test_foo(self):
...         with patch('mymodule.Foo') as mock_foo:
...             with patch('mymodule.Bar') as mock_bar:
...                 with patch('mymodule.Spam') as mock_spam:
...                     assert mymodule.Foo is mock_foo
...                     assert mymodule.Bar is mock_bar
...                     assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').test_foo()
>>> assert mymodule.Foo is original
```

With unittest `cleanup` functions and the *patch のメソッド: start と stop* we can achieve the same effect without the nested indentation. A simple helper method, `create_patch`, puts the patch in place and returns the created mock for us:

```
>>> class MyTest(unittest.TestCase):
...
...     def create_patch(self, name):
...         patcher = patch(name)
...         thing = patcher.start()
...         self.addCleanup(patcher.stop)
...         return thing
...
...     def test_foo(self):
...         mock_foo = self.create_patch('mymodule.Foo')
...         mock_bar = self.create_patch('mymodule.Bar')
...         mock_spam = self.create_patch('mymodule.Spam')
...
...         assert mymodule.Foo is mock_foo
...         assert mymodule.Bar is mock_bar
...         assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').run()
>>> assert mymodule.Foo is original
```

MagicMock で辞書をモックする

You may want to mock a dictionary, or other container object, recording all access to it whilst having it still behave like a dictionary.

We can do this with *MagicMock*, which will behave like a dictionary, and using *side_effect* to delegate dictionary access to a real underlying dictionary that is under our control.

When the `__getitem__()` and `__setitem__()` methods of our `MagicMock` are called (normal dictionary access) then `side_effect` is called with the key (and in the case of `__setitem__` the value too). We can also control what is returned.

After the `MagicMock` has been used we can use attributes like *call_args_list* to assert about how the dictionary was used:

```
>>> my_dict = {'a': 1, 'b': 2, 'c': 3}
>>> def getitem(name):
...      return my_dict[name]
...
>>> def setitem(name, val):
...     my_dict[name] = val
...
>>> mock = MagicMock()
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

注釈: An alternative to using `MagicMock` is to use `Mock` and *only* provide the magic methods you specifically want:

```
>>> mock = Mock()
>>> mock.__getitem__ = Mock(side_effect=getitem)
>>> mock.__setitem__ = Mock(side_effect=setitem)
```

A *third* option is to use `MagicMock` but passing in `dict` as the *spec* (or *spec_set*) argument so that the `MagicMock` created only has dictionary magic methods available:

```
>>> mock = MagicMock(spec_set=dict)
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

With these side effect functions in place, the `mock` will behave like a normal dictionary but recording the access. It even raises a *KeyError* if you try to access a key that doesn't exist.

```
>>> mock['a']
1
>>> mock['c']
3
>>> mock['d']
```

(次のページに続く)

(前のページからの続き)

```
Traceback (most recent call last):
    ...
KeyError: 'd'
>>> mock['b'] = 'fish'
>>> mock['d'] = 'eggs'
>>> mock['b']
'fish'
>>> mock['d']
'eggs'
```

After it has been used you can make assertions about the access using the normal mock methods and attributes:

```
>>> mock.__getitem__.call_args_list
[call('a'), call('c'), call('d'), call('b'), call('d')]
>>> mock.__setitem__.call_args_list
[call('b', 'fish'), call('d', 'eggs')]
>>> my_dict
{'a': 1, 'b': 'fish', 'c': 3, 'd': 'eggs'}
```

Mock のサブクラスと属性

There are various reasons why you might want to subclass *Mock*. One reason might be to add helper methods. Here's a silly example:

```
>>> class MyMock(MagicMock):
...     def has_been_called(self):
...         return self.called
...
>>> mymock = MyMock(return_value=None)
>>> mymock
<MyMock id='...'>
>>> mymock.has_been_called()
False
>>> mymock()
>>> mymock.has_been_called()
True
```

The standard behaviour for `Mock` instances is that attributes and the return value mocks are of the same type as the mock they are accessed on. This ensures that `Mock` attributes are `Mocks` and `MagicMock` attributes are `MagicMocks`[*2]. So if you're subclassing to add helper methods then they'll also be available on the attributes and return value mock of instances of your subclass.

```
>>> mymock.foo
<MyMock name='mock.foo' id='...'>
```

(次のページに続く)

[*2] An exception to this rule are the non-callable mocks. Attributes use the callable variant because otherwise non-callable mocks couldn't have callable methods.

(前のページからの続き)

```
>>> mymock.foo.has_been_called()
False
>>> mymock.foo()
<MyMock name='mock.foo()' id='...'>
>>> mymock.foo.has_been_called()
True
```

Sometimes this is inconvenient. For example, one user is subclassing mock to created a Twisted adaptor. Having this applied to attributes too actually causes errors.

`Mock` (in all its flavours) uses a method called `_get_child_mock` to create these "sub-mocks" for attributes and return values. You can prevent your subclass being used for attributes by overriding this method. The signature is that it takes arbitrary keyword arguments (`**kwargs`) which are then passed onto the mock constructor:

```
>>> class Subclass(MagicMock):
...     def _get_child_mock(self, /, **kwargs):
...         return MagicMock(**kwargs)
...
>>> mymock = Subclass()
>>> mymock.foo
<MagicMock name='mock.foo' id='...'>
>>> assert isinstance(mymock, Subclass)
>>> assert not isinstance(mymock.foo, Subclass)
>>> assert not isinstance(mymock(), Subclass)
```

patch.dict で import をモックする

One situation where mocking can be hard is where you have a local import inside a function. These are harder to mock because they aren't using an object from the module namespace that we can patch out.

Generally local imports are to be avoided. They are sometimes done to prevent circular dependencies, for which there is *usually* a much better way to solve the problem (refactor the code) or to prevent "up front costs" by delaying the import. This can also be solved in better ways than an unconditional local import (store the module as a class or module attribute and only do the import on first use).

That aside there is a way to use `mock` to affect the results of an import. Importing fetches an *object* from the *sys.modules* dictionary. Note that it fetches an *object*, which need not be a module. Importing a module for the first time results in a module object being put in *sys.modules*, so usually when you import something you get a module back. This need not be the case however.

This means you can use *patch.dict()* to *temporarily* put a mock in place in *sys.modules*. Any imports whilst this patch is active will fetch the mock. When the patch is complete (the decorated function exits, the with statement body is complete or `patcher.stop()` is called) then whatever was there previously will be restored safely.

Here's an example that mocks out the 'fooble' module.

```
>>> import sys
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...    import fooble
...    fooble.blob()
...
<Mock name='mock.blob()' id='...'>
>>> assert 'fooble' not in sys.modules
>>> mock.blob.assert_called_once_with()
```

As you can see the `import fooble` succeeds, but on exit there is no 'fooble' left in *sys.modules*.

This also works for the `from module import name` form:

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...    from fooble import blob
...    blob.blip()
...
<Mock name='mock.blob.blip()' id='...'>
>>> mock.blob.blip.assert_called_once_with()
```

With slightly more work you can also mock package imports:

```
>>> mock = Mock()
>>> modules = {'package': mock, 'package.module': mock.module}
>>> with patch.dict('sys.modules', modules):
...    from package.module import fooble
...    fooble()
...
<Mock name='mock.module.fooble()' id='...'>
>>> mock.module.fooble.assert_called_once_with()
```

Tracking order of calls and less verbose call assertions

The *Mock* class allows you to track the *order* of method calls on your mock objects through the *method_calls* attribute. This doesn't allow you to track the order of calls between separate mock objects, however we can use *mock_calls* to achieve the same effect.

Because mocks track calls to child mocks in `mock_calls`, and accessing an arbitrary attribute of a mock creates a child mock, we can create our separate mocks from a parent one. Calls to those child mock will then all be recorded, in order, in the `mock_calls` of the parent:

```
>>> manager = Mock()
>>> mock_foo = manager.foo
>>> mock_bar = manager.bar
```

```
>>> mock_foo.something()
<Mock name='mock.foo.something()' id='...'>
```

(次のページに続く)

(前のページからの続き)

```
>>> mock_bar.other.thing()
<Mock name='mock.bar.other.thing()' id='...'>
```

```
>>> manager.mock_calls
[call.foo.something(), call.bar.other.thing()]
```

We can then assert about the calls, including the order, by comparing with the `mock_calls` attribute on the manager mock:

```
>>> expected_calls = [call.foo.something(), call.bar.other.thing()]
>>> manager.mock_calls == expected_calls
True
```

If `patch` is creating, and putting in place, your mocks then you can attach them to a manager mock using the *attach_mock()* method. After attaching calls will be recorded in `mock_calls` of the manager.

```
>>> manager = MagicMock()
>>> with patch('mymodule.Class1') as MockClass1:
...     with patch('mymodule.Class2') as MockClass2:
...         manager.attach_mock(MockClass1, 'MockClass1')
...         manager.attach_mock(MockClass2, 'MockClass2')
...         MockClass1().foo()
...         MockClass2().bar()
<MagicMock name='mock.MockClass1().foo()' id='...'>
<MagicMock name='mock.MockClass2().bar()' id='...'>
>>> manager.mock_calls
[call.MockClass1(),
call.MockClass1().foo(),
call.MockClass2(),
call.MockClass2().bar()]
```

If many calls have been made, but you're only interested in a particular sequence of them then an alternative is to use the *assert_has_calls()* method. This takes a list of calls (constructed with the *call* object). If that sequence of calls are in *mock_calls* then the assert succeeds.

```
>>> m = MagicMock()
>>> m().foo().bar().baz()
<MagicMock name='mock().foo().bar().baz()' id='...'>
>>> m.one().two().three()
<MagicMock name='mock.one().two().three()' id='...'>
>>> calls = call.one().two().three().call_list()
>>> m.assert_has_calls(calls)
```

Even though the chained call `m.one().two().three()` aren't the only calls that have been made to the mock, the assert still succeeds.

Sometimes a mock may have several calls made to it, and you are only interested in asserting about *some* of those calls. You may not even care about the order. In this case you can pass `any_order=True` to `assert_has_calls`:

```
>>> m = MagicMock()
>>> m(1), m.two(2, 3), m.seven(7), m.fifty('50')
(...)
>>> calls = [call.fifty('50'), call(1), call.seven(7)]
>>> m.assert_has_calls(calls, any_order=True)
```

More complex argument matching

Using the same basic concept as *ANY* we can implement matchers to do more complex assertions on objects used as arguments to mocks.

Suppose we expect some object to be passed to a mock that by default compares equal based on object identity (which is the Python default for user defined classes). To use *assert_called_with()* we would need to pass in the exact same object. If we are only interested in some of the attributes of this object then we can create a matcher that will check these attributes for us.

You can see in this example how a 'standard' call to `assert_called_with` isn't sufficient:

```
>>> class Foo:
...     def __init__(self, a, b):
...         self.a, self.b = a, b
...
>>> mock = Mock(return_value=None)
>>> mock(Foo(1, 2))
>>> mock.assert_called_with(Foo(1, 2))
Traceback (most recent call last):
    ...
AssertionError: Expected: call(<__main__.Foo object at 0x...>)
Actual call: call(<__main__.Foo object at 0x...>)
```

A comparison function for our `Foo` class might look something like this:

```
>>> def compare(self, other):
...     if not type(self) == type(other):
...         return False
...     if self.a != other.a:
...         return False
...     if self.b != other.b:
...         return False
...     return True
...
```

And a matcher object that can use comparison functions like this for its equality operation would look something like this:

```
>>> class Matcher:
...     def __init__(self, compare, some_obj):
...         self.compare = compare
...         self.some_obj = some_obj
```

(次のページに続く)

(前のページからの続き)

```
...     def __eq__(self, other):
...         return self.compare(self.some_obj, other)
...
```

全てをつなぎ合わせて:

```
>>> match_foo = Matcher(compare, Foo(1, 2))
>>> mock.assert_called_with(match_foo)
```

The `Matcher` is instantiated with our compare function and the `Foo` object we want to compare against. In `assert_called_with` the `Matcher` equality method will be called, which compares the object the mock was called with against the one we created our matcher with. If they match then `assert_called_with` passes, and if they don't an *AssertionError* is raised:

```
>>> match_wrong = Matcher(compare, Foo(3, 4))
>>> mock.assert_called_with(match_wrong)
Traceback (most recent call last):
    ...
AssertionError: Expected: ((<Matcher object at 0x...>,), {})
Called with: ((<Foo object at 0x...>,), {})
```

With a bit of tweaking you could have the comparison function raise the *AssertionError* directly and provide a more useful failure message.

As of version 1.5, the Python testing library PyHamcrest provides similar functionality, that may be useful here, in the form of its equality matcher (hamcrest.library.integration.match_equality).

26.7 2to3 - Python 2 から 3 への自動コード変換

2to3 は、Python 2.x のソースコードを読み込み、一連の **変換プログラム** を適用して Python 3.x のコードに変換するプログラムです。標準ライブラリはほとんど全てのコードを取り扱うのに十分な変換プログラムを含んでいます。ただし 2to3 を構成している *lib2to3* は柔軟かつ一般的なライブラリなので、2to3 のために自分で変換プログラムを書くこともできます。*lib2to3* は、Python コードを自動編集する必要がある場合にも適用することができます。

26.7.1 2to3 の使用

2to3 は大抵の場合、Python インタープリターと共に、スクリプトとしてインストールされます。場所は、Python のルートディレクトリにある、`Tools/scripts` ディレクトリです。

2to3 に与える基本の引数は、変換対象のファイル、もしくは、ディレクトリのリストです。ディレクトリの場合は、Python ソースコードを再帰的に探索します。

Python 2.x のサンプルコード、`example.py` を示します:

```
def greet(name):
    print "Hello, {0}!".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

これは、コマンドラインから 2to3 を呼び出すことで、Python 3.x コードに変換されます:

```
$ 2to3 example.py
```

オリジナルのソースファイルに対する差分が表示されます。2to3 は必要となる変更をソースファイルに書き込むこともできます (-n も与えたのでない限りオリジナルのバックアップも作成されます)。変更の書き戻しは -w フラグによって有効化されます:

```
$ 2to3 -w example.py
```

変換後、example.py は以下のようになります:

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
greet(name)
```

変換処理を通じて、コメントと、インデントは保存されます。

デフォルトでは、2to3 は一連の **事前定義された変換プログラム** を実行します。-l フラグは、利用可能な変換プログラムの一覧を表示します。-f フラグにより、実行する変換プログラムを明示的に与えることもできます。同様に、-x は変換プログラムを明示的に無効化します。下記の例では、imports と has_key 変換プログラムだけを実行します:

```
$ 2to3 -f imports -f has_key example.py
```

このコマンドは apply 以外のすべての変換プログラムを実行します:

```
$ 2to3 -x apply example.py
```

いくつかの変換プログラムは **明示的**、つまり、デフォルトでは実行されず、コマンドラインで実行するものとして列記する必要があります。デフォルトの変換プログラムに idioms 変換プログラムを追加して実行するには、下記のようにします:

```
$ 2to3 -f all -f idioms example.py
```

ここで、all を指定することで、全てのデフォルトの変換プログラムを有効化できることに注意して下さい。

2to3 がソースコードに修正すべき点を見つけても、自動的には修正できない場合もあります。この場合、2to3 はファイルの変更点の下に警告を表示します。3.x に準拠したコードにするために、あなたはこの警告に対処しなくてはなりません。

2to3 は doctest の修正もできます。このモードを有効化するには -d フラグを指定して下さい。doctest **だけ**が修正されることに注意して下さい。これは、モジュールが有効な Python コードであることを要求しないということでもあります。例えば、reST ドキュメント中の doctest に似たサンプルコードなども、このオプションで修正することができます。

-v は、変換処理のより詳しい情報の出力を有効化します。

いくつかの print 文は関数呼び出しとしても文としても解析できるので、2to3 は print 関数を含むファイルを常に読めるとは限りません。2to3 は `from __future__ import print_function` コンパイラディレクティブが存在することを検出すると、内部の文法を変更して *print()* を関数として解釈するようになります。-p フラグによって手動でこの変更を有効化することもできます。print 文を変換済みのコードに対して変換プログラムを適用するには -p を使用してください。

-o または --output-dir で処理結果の出力先ディレクトリを変更出来ます。入力ファイルを上書きしないならバックアップは意味をなさないので、オプション -n フラグが要ります。

バージョン 3.2.3 で追加: -o オプションが追加されました。

-W または --write-unchanged-files により、たとえファイルに変更が必要なくても常にファイルを出力するように 2to3 に指示することが出来ます。これは、-o とともに使って、Python ソースツリー全体を変換して別のディレクトリに書き出す際に最も有用です。

バージョン 3.2.3 で追加: -W フラグが追加されました。

オプション --add-suffix で、全ての出力ファイル名に与えた文字列を追加します。これを指定するのであれば別名で書き出されるためバックアップは必要ないので、オプション -n フラグが要ります。例えば:

```
$ 2to3 -n -W --add-suffix=3 example.py
```

とすれば変換後ファイルは example.py3 として書き出されます。

バージョン 3.2.3 で追加: --add-suffix オプションが追加されました。

ひとつのディレクトリツリーからプロジェクト全体を変換したければこのように使います:

```
$ 2to3 --output-dir=python3-version/mycode -W -n python2-version/mycode
```

26.7.2 変換プログラム

コード変形の各ステップは変換プログラムに隠蔽されます。2to3 -l コマンドは変換プログラムのリストを表示します。上記 の通り、それぞれの変換プログラムを個別に有効化したり無効化したりすることができます。ここではそれらをさらに詳細に説明します。

apply
: apply() の使用を削除します。例えば apply(function, *args, **kwargs) は function(*args, **kwargs) に変換されます。

asserts
: 廃止になった *unittest* メソッド名を新しい名前に置換します。

対象	変換先
failUnlessEqual(a, b)	*assertEqual(a, b)*
assertEquals(a, b)	*assertEqual(a, b)*
failIfEqual(a, b)	*assertNotEqual(a, b)*
assertNotEquals(a, b)	*assertNotEqual(a, b)*
failUnless(a)	*assertTrue(a)*
assert_(a)	*assertTrue(a)*
failIf(a)	*assertFalse(a)*
failUnlessRaises(exc, cal)	*assertRaises(exc, cal)*
failUnlessAlmostEqual(a, b)	*assertAlmostEqual(a, b)*
assertAlmostEquals(a, b)	*assertAlmostEqual(a, b)*
failIfAlmostEqual(a, b)	*assertNotAlmostEqual(a, b)*
assertNotAlmostEquals(a, b)	*assertNotAlmostEqual(a, b)*

basestring

basestring を *str* に変換します。

buffer

buffer を *memoryview* に変換します。*memoryview* API は buffer と似ているものの厳密に同じではないので、この変換プログラムはオプションです。

dict

辞書をイテレートするメソッドを修正します。dict.iteritems() は *dict.items()* に、dict.iterkeys() は *dict.keys()* に、dict.itervalues() は *dict.values()* に変換されます。同様に dict.viewitems(), dict.viewkeys() dict.viewvalues() はそれぞれ *dict.items()*, *dict.keys()*, *dict.values()* に変換されます。また、*list* の呼び出しの中で *dict.items()*, *dict.keys()*, *dict.values()* を使用している場合はそれをラップします。

except

except X, T を except X as T に変換します。

exec

exec 文を *exec()* 関数に変換します。

execfile

execfile() の使用を削除します。execfile() への引数は *open()*, *compile()*, *exec()* の呼び出しでラップされます。

exitfunc

sys.exitfunc への代入を *atexit* モジュールの使用に変更します。

filter

list 呼び出しの中で *filter()* を使用している部分をラップします。

funcattrs

名前が変更された関数の属性を修正します。例えば my_function.func_closure は my_function.

__closure__ に変換されます。

future

from __future__ import new_feature 文を削除します。

getcwdu

os.getcwdu() を *os.getcwd()* に置き換えます。

has_key

dict.has_key(key) を key in dict に変更します。

idioms

このオプションの変換プログラムは、Python コードをより Python らしい書き方にするいくつかの変形を行います。type(x) is SomeClass や type(x) == SomeClass のような型の比較は isinstance(x, SomeClass) に変換されます。while 1 は while True になります。また、適切な場所では *sorted()* が使われるようにします。例えば、このブロックは

```
L = list(some_iterable)
L.sort()
```

次のように変更されます

```
L = sorted(some_iterable)
```

import

暗黙の相対インポート (sibling imports) を検出して、明示的な相対インポート (relative imports) に変換します。

imports

標準ライブラリ中のモジュール名の変更を扱います。

imports2

標準ライブラリ中の別のモジュール名の変更を扱います。単に技術的な制約のために *imports* とは別になっています。

input

input(prompt) を eval(input(prompt)) に変換します。

intern

intern() を *sys.intern()* に変換します。

isinstance

isinstance() の第 2 引数の重複を修正します。例えば isinstance(x, (int, int)) は isinstance(x, (int)) に isinstance(x, (int, float, int)) は isinstance(x, (int, float)) に変換されます。

itertools_imports

itertools.ifilter(), itertools.izip(), itertools.imap() のインポートを削除します。また itertools.ifilterfalse() のインポートを *itertools.filterfalse()* に変換します。

itertools

itertools.ifilter(), itertools.izip(), itertools.imap() を使っている箇所を同等の組み込み関数で置き換えます。itertools.ifilterfalse() は *itertools.filterfalse()* に変換されます。

long

long を *int* に変更します。

map

list 呼び出しの中の *map()* をラップします。また、map(None, x) を list(x) に変換します。from future_builtins import map を使うと、この変換プログラムを無効にできます。

metaclass

古いメタクラス構文 (クラス定義中の __metaclass__ = Meta) を、新しい構文 (class X(metaclass=Meta)) に変換します。

methodattrs

古いメソッドの属性名を修正します。例えば meth.im_func は meth.__func__ に変換されます。

ne

古い不等号の構文 <> を != に変換します。

next

イテレータの next() メソッドの使用を *next()* 関数に変換します。また *next()* メソッドを *__next__()* に変更します。

nonzero

__nonzero__() を __bool__() に変更します。

numliterals

8 進数リテラルを新しい構文に変換します。

operator

operator モジュール内のさまざまな関数呼び出しを、他の、しかし機能的には同等の関数呼び出しに変換します。必要に応じて、import collections.abc などの適切な import ステートメントが追加されます。以下のマッピングが行われます。

対象	変換先
operator.isCallable(obj)	callable(obj)
operator.sequenceIncludes(obj)	operator.contains(obj)
operator.isSequenceType(obj)	isinstance(obj, collections.abc.Sequence)
operator.isMappingType(obj)	isinstance(obj, collections.abc.Mapping)
operator.isNumberType(obj)	isinstance(obj, numbers.Number)
operator.repeat(obj, n)	operator.mul(obj, n)
operator.irepeat(obj, n)	operator.imul(obj, n)

paren

リスト内包表記で必要になる括弧を追加します。例えば `[x for x in 1, 2]` は `[x for x in (1, 2)]` になります。

print

print 文を *print()* 関数に変換します。

raise

`raise E, V` を `raise E(V)` に、`raise E, V, T` を `raise E(V).with_traceback(T)` に変換します。例外の代わりにタプルを使用することは 3.0 で削除されたので、E がタプルならこの変換は不正確になります。

raw_input

`raw_input()` を *input()* に変換します。

reduce

`reduce()` が *functools.reduce()* に移動されたことを扱います。

reload

`reload()` を *importlib.reload()* に変換します。

renames

`sys.maxint` を *sys.maxsize* に変更します。

repr

バッククォートを使った repr を *repr()* 関数に置き換えます。

set_literal

set コンストラクタの使用を set リテラルに置換します。この変換プログラムはオプションです。

standarderror

StandardError を *Exception* に変更します。

sys_exc

廃止された `sys.exc_value`, `sys.exc_type`, `sys.exc_traceback` の代わりに *sys.exc_info()* を使うように変更します。

throw

ジェネレータの `throw()` メソッドの API 変更を修正します。

tuple_params

関数定義における暗黙的なタプルパラメータの展開を取り除きます。この変換プログラムによって一時変数が追加されます。

types

types モジュールのいくつかのメンバオブジェクトが削除されたことによって壊れたコードを修正します。

unicode

unicode を *str* に変更します。

urllib
: *urllib* と urllib2 が *urllib* パッケージに変更されたことを扱います。

ws_comma
: コンマ区切りの要素から余計な空白を取り除きます。この変換プログラムはオプションです。

xrange
: xrange() を *range()* に変更して、既存の *range()* 呼び出しを *list* でラップします。

xreadlines
: for x in file.xreadlines() を for x in file に変更します。

zip
: *list* 呼び出しの中で使われている *zip()* をラップします。これは from future_builtins import zip が見つかった場合は無効にされます。

26.7.3 lib2to3 - 2to3's library

ソースコード: Lib/lib2to3/

注釈: *lib2to3* API は安定しておらず、将来、劇的に変更されるかも知れないと考えるべきです。

26.8 test --- Python 用回帰テストパッケージ

注釈: *test* パッケージは Python の内部利用専用です。ドキュメント化されているのは Python のコア開発者のためです。ここで述べられているコードは Python のリリースで予告なく変更されたり、削除される可能性があるため、Python 標準ライブラリー外でこのパッケージを使用することは推奨されません。

---

test パッケージには、Python 用の全ての回帰テストの他に、*test.support* モジュールと test.regrtest モジュールが入っています。*test.support* はテストを充実させるために使い、test.regrtest はテストスイートを実行するのに使います。

test パッケージ内のモジュールのうち、名前が test_ で始まるものは、特定のモジュールや機能に対するテストスイートです。新しいテストはすべて *unittest* か *doctest* モジュールを使って書くようにしてください。古いテストのいくつかは、sys.stdout への出力を比較する「従来の」テスト形式になっていますが、この形式のテストは廃止予定です。

参考:

unittest **モジュール**　PyUnit 回帰テストを書く。

doctest **モジュール**　ドキュメンテーション文字列に埋め込まれたテスト。

26.8.1 test パッケージのためのユニットテストを書く

unittest モジュールを使ってテストを書く場合、幾つかのガイドラインに従うことが推奨されます。1 つは、テストモジュールの名前を、test_ で始め、テスト対象となるモジュール名で終えることです。テストモジュール中のテストメソッドは名前を test_ で始めて、そのメソッドが何をテストしているかという説明で終えます。これはテスト実行プログラムが、そのメソッドをテストメソッドとして認識するために必要です。また、テストメソッドにはドキュメンテーション文字列を入れるべきではありません。コメント（例えば **# True あるいは False だけを返すテスト関数** ）を使用して、テストメソッドのドキュメントを記述してください。これは、ドキュメンテーション文字列が存在する場合はその内容が出力されてしまうため、どのテストを実行しているのかをいちいち表示したくないからです。

以下のような決まり文句を使います:

```
import unittest
from test import support

class MyTestCase1(unittest.TestCase):

    # Only use setUp() and tearDown() if necessary

    def setUp(self):
        ... code to execute in preparation for tests ...

    def tearDown(self):
        ... code to execute to clean up after tests ...

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...

    def test_feature_two(self):
        # Test feature two.
        ... testing code ...

    ... more test methods ...

class MyTestCase2(unittest.TestCase):
    ... same structure as MyTestCase1 ...

... more test classes ...

if __name__ == '__main__':
    unittest.main()
```

このコードのパターンを使うと test.regrtest からテストスイートを実行でき、*unittest* のコマンドラインインターフェースをサポートしているスクリプトとして自分自身を起動したり、python -m unittest というコマンドラインインターフェースを通して起動したりできます。

回帰テストの目的はコードを解き明かすことです。そのためには以下のいくつかのガイドラインに従ってください:

- テストスイートから、すべてのクラス、関数および定数を実行するべきです。これには外部に公開される外部 API だけでなく「プライベートな」コードも含みます。
- ホワイトボックス・テスト（対象のコードの詳細を元にテストを書くこと）を推奨します。ブラックボックス・テスト（公開されるインタフェース仕様だけをテストすること）は、すべての境界条件を確実にテストするには完全ではありません。
- すべての取りうる値を、無効値も含めてテストするようにしてください。そのようなテストを書くことで、全ての有効値が通るだけでなく、不適切な値が正しく処理されることも確認できます。
- コード内のできる限り多くのパスを網羅してください。分岐するように入力を調整したテストを書くことで、コードの多くのパスをたどることができます。
- テスト対象のコードにバグが発見された場合は、明示的にテスト追加するようにしてください。そのようなテストを追加することで、将来コードを変更した際にエラーが再発することを防止できます。
- テストの後始末 (例えば一時ファイルをすべて閉じたり削除したりすること) を必ず行ってください。
- テストがオペレーティングシステムの特定の状況に依存する場合、テスト開始時に条件を満たしているかを検証してください。
- インポートするモジュールをできるかぎり少なくし、可能な限り早期にインポートを行ってください。そうすることで、テストの外部依存性を最小限にし、モジュールのインポートによる副作用から生じる変則的な動作を最小限にできます。
- できる限りテストコードを再利用するようにしましょう。時として、入力の違いだけを記述すれば良くなるくらい、テストコードを小さくすることができます。例えば以下のように、サブクラスで入力を指定することで、コードの重複を最小化することができます:

```
class TestFuncAcceptsSequencesMixin:

    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = [1, 2, 3]

class AcceptStrings(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = 'abc'

class AcceptTuples(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = (1, 2, 3)
```

  このパターンを使うときには、*unittest.TestCase* を継承した全てのクラスがテストとして実行されることを忘れないでください。上の例の `Mixin` クラスはテストデータを持っておらず、それ自身は実行できないので、*unittest.TestCase* を継承していません。

参考:

Test Driven Development　コードより前にテストを書く方法論に関する Kent Beck の著書。

26.8.2 コマンドラインインタフェースを利用してテストを実行する

test パッケージはスクリプトとして Python の回帰テストスイートを実行できます。-m オプションを利用して、**python -m test.regrtest** として実行します。この仕組みの内部では test.regrtest; を使っています; 古いバージョンの Python で使われている **python -m test.regrtest** という呼び出しは今でも上手く動きます。スクリプトを実行すると、自動的に *test* パッケージ内のすべての回帰テストを実行し始めます。パッケージ内の名前が test_ で始まる全モジュールを見つけ、それをインポートし、もしあるなら関数 test_main() を実行し、test_main が無い場合は unittest.TestLoader.loadTestsFromModule からテストをロードしてテストを実行します。実行するテストの名前もスクリプトに渡される可能性があります。単一の回帰テストを指定 (**python -m test test_spam**) すると、出力を最小限にし、テストが成功したかあるいは失敗したかだけを出力します。

直接 *test* を実行すると、テストに利用するリソースを設定できます。これを行うには、-u コマンドラインオプションを使います。-u のオプションに all を指定すると、すべてのリソースを有効にします: **python -m test -uall** 。(よくある場合ですが) 何か一つを除く全てが必要な場合、カンマで区切った不要なリソースのリストを all の後に並べます。コマンド **python -m test -uall,-audio,-largefile** とすると、audio と largefile リソースを除く全てのリソースを使って *test* を実行します。すべてのリソースのリストと追加のコマンドラインオプションを出力するには、**python -m test -h** を実行してください。

テストを実行しようとするプラットフォームによっては、回帰テストを実行する別の方法があります。Unix では、Python をビルドしたトップレベルディレクトリで **make test** を実行できます。Windows 上では、PCbuild ディレクトリから **rt.bat** を実行すると、すべての回帰テストを実行します。

26.9 test.support --- テストのためのユーティリティ関数

test.support モジュールでは、Python の回帰テストに対するサポートを提供しています。

注釈: *test.support* はパブリックなモジュールではありません。ここでドキュメント化されているのは Python 開発者がテストを書くのを助けるためです。このモジュールの API はリリース間で後方非互換な変更がなされる可能性があります。

このモジュールは次の例外を定義しています:

exception test.support.TestFailed

テストが失敗したとき送出される例外です。これは、*unittest* ベースのテストでは廃止予定で、*unittest.TestCase* の assertXXX メソッドが推奨されます。

exception test.support.ResourceDenied

unittest.SkipTest のサブクラスです。(ネットワーク接続のような) リソースが利用できないとき送出されます。*requires()* 関数によって送出されます。

test.support モジュールでは、以下の定数を定義しています:

test.support.verbose

冗長な出力が有効な場合は True です。実行中のテストについてのより詳細な情報が欲しいときに

チェックします。*verbose* は test.regrtest によって設定されます。

test.support.**is_jython**
実行中のインタプリタが Jython ならば True になります。

test.support.**is_android**
システムが Android の場合 True になります。

test.support.**unix_shell**
Windows 以外ではシェルのパスです; そうでない場合は None です。

test.support.**FS_NONASCII**
os.fsencode() でエンコードできる 非 ASCII 文字。

test.support.**TESTFN**
テンポラリファイルの名前として安全に利用できる名前に設定されます。作成した一時ファイルは全て閉じ、unlink (削除) しなければなりません。

test.support.**TESTFN_UNICODE**
非 ASCII 名を一時ファイルに設定します。

test.support.**TESTFN_ENCODING**
sys.getfilesystemencoding() を設定します。

test.support.**TESTFN_UNENCODABLE**
strict モードのファイルシステムエンコーディングでエンコードできないファイル名（str 型）に設定します。そのようなファイル名を生成できない場合は、'None'になる可能性があります。

test.support.**TESTFN_UNDECODABLE**
strict モードのファイルシステムエンコーディングでデコードできないファイル名（bytes 型）に設定します。そのようなファイル名を生成できない場合は、None になる可能性があります。

test.support.**TESTFN_NONASCII**
FS_NONASCII 文字を含むファイル名を設定します。

test.support.**IPV6_ENABLED**
このホストで IPV6 が有効化されている場合 True に、そうでない場合 False に設定されます。

test.support.**SAVEDCWD**
os.getcwd() に設定されます。

test.support.**PGO**
テストが PGO (Profile Guided Optimization) の役に立たないときにスキップできるなら設定します。

test.support.**PIPE_MAX_SIZE**
書き込みをブロックするための、基底にある OS のパイプバッファサイズより大きいであろう定数。

test.support.**SOCK_MAX_SIZE**
書き込みをブロックするための、基底にある OS のソケットバッファサイズより大きいであろう定数。

test.support.**TEST_SUPPORT_DIR**
test.support を含んだトップディレクトリを設定します。

test.support.**TEST_HOME_DIR**
テストパッケージのトップディレクトリを設定します。

test.support.**TEST_DATA_DIR**
テストパッケージ内の data ディレクトリを設定します。

test.support.**MAX_Py_ssize_t**
大量のメモリを使うテストのための *sys.maxsize* を設定します。

test.support.**max_memuse**
大量のメモリを使うテストのためのメモリ上限となる *set_memlimit()* を設定します。*MAX_Py_ssize_t* が設定上限です。

test.support.**real_max_memuse**
大量のメモリを使うテストのためのメモリ上限となる *set_memlimit()* を設定します。*MAX_Py_ssize_t* の設定上限はありません。

test.support.**MISSING_C_DOCSTRINGS**
CPython 上で実行されていて、Windows 上ではなく、WITH_DOC_STRINGS が設定されていない場合に True を返します。

test.support.**HAVE_DOCSTRINGS**
docstring があるかを確認します。

test.support.**TEST_HTTP_URL**
ネットワークテスト専用の HTTP サーバーの URL を定義します。

test.support.**ALWAYS_EQ**
Object that is equal to anything. Used to test mixed type comparison.

test.support.**LARGEST**
Object that is greater than anything (except itself). Used to test mixed type comparison.

test.support.**SMALLEST**
Object that is less than anything (except itself). Used to test mixed type comparison.

test.support モジュールでは、以下の関数を定義しています:

test.support.**forget**(*module_name*)
モジュール名 *module_name* を sys.modules から取り除き、モジュールのバイトコンパイル済みファイルを全て削除します。

test.support.**unload**(*name*)
sys.modules から *name* を削除します。

test.support.**unlink**(*filename*)
Call *os.unlink()* on *filename*. On Windows platforms, this is wrapped with a wait loop that checks for the existence fo the file.

test.support.**rmdir**(*filename*)

Call *os.rmdir()* on *filename*. On Windows platforms, this is wrapped with a wait loop that checks for the existence of the file.

test.support.**rmtree**(*path*)

Call *shutil.rmtree()* on *path* or call *os.lstat()* and *os.rmdir()* to remove a path and its contents. On Windows platforms, this is wrapped with a wait loop that checks for the existence of the files.

test.support.**make_legacy_pyc**(*source*)

Move a **PEP 3147**/**PEP 488** pyc file to its legacy pyc location and return the file system path to the legacy pyc file. The *source* value is the file system path to the source file. It does not need to exist, however the PEP 3147/488 pyc file must exist.

test.support.**is_resource_enabled**(*resource*)

resource が有効で利用可能ならば `True` を返します。利用可能なリソースのリストは、`test.regrtest` がテストを実行している間のみ設定されます。

test.support.**python_is_optimized**()

Return `True` if Python was not built with `-O0` or `-Og`.

test.support.**with_pymalloc**()

Return `_testcapi.WITH_PYMALLOC`.

test.support.**requires**(*resource*, *msg=None*)

resource が利用できなければ、*ResourceDenied* を送出します。その場合、*msg* は *ResourceDenied* の引数になります。`__name__` が `'__main__'` である関数にから呼び出された場合には常に `True` を返します。テストを `test.regrtest` から実行するときに使われます。

test.support.**system_must_validate_cert**(*f*)

Raise *unittest.SkipTest* on TLS certification validation failures.

test.support.**sortdict**(*dict*)

Return a repr of *dict* with keys sorted.

test.support.**findfile**(*filename*, *subdir=None*)

filename という名前のファイルへのパスを返します。一致するものが見つからなければ、*filename* 自体を返します。*filename* 自体もファイルへのパスでありえるので、*filename* が返っても失敗ではありません。

subdir を設定することで、パスのディレクトリを直接見に行くのではなく、相対パスを使って見付けにいくように指示できます。

test.support.**create_empty_file**(*filename*)

Create an empty file with *filename*. If it already exists, truncate it.

test.support.**fd_count**()

Count the number of open file descriptors.

test.support.**match_test**(*test*)

Match *test* to patterns set in *set_match_tests()*.

test.support.**set_match_tests**(*patterns*)

Define match test with regular expression *patterns*.

test.support.**run_unittest**(**classes*)

渡された *unittest.TestCase* サブクラスを実行します。この関数は名前が test_ で始まるメソッドを探して、テストを個別に実行します。

引数に文字列を渡すことも許可されています。その場合、文字列は sys.module のキーでなければなりません。指定された各モジュールは、unittest.TestLoader.loadTestsFromModule() でスキャンされます。この関数は、よく次のような test_main() 関数の形で利用されます。

```
def test_main():
    support.run_unittest(__name__)
```

この関数は、名前で指定されたモジュールの中の全ての定義されたテストを実行します。

test.support.**run_doctest**(*module*, *verbosity=None*, *optionflags=0*)

与えられた *module* の *doctest.testmod()* を実行します。(failure_count, test_count) を返します。

If *verbosity* is None, *doctest.testmod()* is run with verbosity set to *verbose*. Otherwise, it is run with verbosity set to None. *optionflags* is passed as optionflags to *doctest.testmod()*.

test.support.**setswitchinterval**(*interval*)

Set the *sys.setswitchinterval()* to the given *interval*. Defines a minimum interval for Android systems to prevent the system from hanging.

test.support.**check_impl_detail**(***guards*)

Use this check to guard CPython's implementation-specific tests or to run them only on the implementations guarded by the arguments:

```
check_impl_detail()               # Only on CPython (default).
check_impl_detail(jython=True)    # Only on Jython.
check_impl_detail(cpython=False)  # Everywhere except CPython.
```

test.support.**check_warnings**(**filters*, *quiet=True*)

warning が正しく発行されているかどうかチェックする、*warnings.catch_warnings()* を使いやすくするラッパーです。これは、*warnings.simplefilter()* を always に設定して、記録された結果を自動的に検証するオプションと共に warnings.catch_warnings(record=True) を呼ぶのとほぼ同じです。

check_warnings は ("message regexp", WarningCategory) の形をした 2 要素タプルを位置引数として受け取ります。1 つ以上の *filters* が与えられた場合や、オプションのキーワード引数 *quiet* が False の場合、警告が期待通りであるかどうかをチェックします。指定された各 filter は最低でも 1 回は囲われたコード内で発生した警告とマッチしなければテストが失敗しますし、指定されたどの filter

ともマッチしない警告が発生してもテストが失敗します。前者のチェックを無効にするには、*quiet* を True にします。

引数が 1 つもない場合、デフォルトでは次のようになります:

```
check_warnings(("", Warning), quiet=True)
```

この場合、全ての警告は補足され、エラーは発生しません。

コンテキストマネージャーに入る時、WarningRecorder インスタンスが返されます。このレコーダーオブジェクトの *warnings* 属性から、*catch_warnings()* から得られる警告のリストを取得することができます。便利さのために、レコーダーオブジェクトから直接、一番最近に発生した警告を表すオブジェクトの属性にアクセスできます (以下にある例を参照してください)。警告が 1 つも発生しなかった場合、それらの全ての属性は None を返します。

レコーダーオブジェクトの reset() メソッドは警告リストをクリアします。

コンテキストマネージャーは次のようにして使います:

```
with check_warnings(("assertion is always true", SyntaxWarning),
                    ("", UserWarning)):
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

この場合、どちらの警告も発生しなかった場合や、それ以外の警告が発生した場合は、*check_warnings()* はエラーを発生させます。

警告が発生したかどうかだけでなく、もっと詳しいチェックが必要な場合は、次のようなコードになります:

```
with check_warnings(quiet=True) as w:
    warnings.warn("foo")
    assert str(w.args[0]) == "foo"
    warnings.warn("bar")
    assert str(w.args[0]) == "bar"
    assert str(w.warnings[0].args[0]) == "foo"
    assert str(w.warnings[1].args[0]) == "bar"
    w.reset()
    assert len(w.warnings) == 0
```

全ての警告をキャプチャし、テストコードがその警告を直接テストします。

バージョン 3.2 で変更: 新しいオプション引数 *filters* と *quiet*

test.support.check_no_resource_warning(*testcase*)

Context manager to check that no *ResourceWarning* was raised. You must remove the object which may emit *ResourceWarning* before the end of the context manager.

test.support.set_memlimit(*limit*)

Set the values for *max_memuse* and *real_max_memuse* for big memory tests.

test.support.**record_original_stdout**(*stdout*)

Store the value from *stdout*. It is meant to hold the stdout at the time the regrtest began.

test.support.**get_original_stdout**()

Return the original stdout set by *record_original_stdout()* or `sys.stdout` if it's not set.

test.support.**strip_python_strerr**(*stderr*)

Strip the *stderr* of a Python process from potential debug output emitted by the interpreter. This will typically be run on the result of *subprocess.Popen.communicate()*.

test.support.**args_from_interpreter_flags**()

Return a list of command line arguments reproducing the current settings in `sys.flags` and `sys.warnoptions`.

test.support.**optim_args_from_interpreter_flags**()

Return a list of command line arguments reproducing the current optimization settings in `sys.flags`.

test.support.**captured_stdin**()

test.support.**captured_stdout**()

test.support.**captured_stderr**()

名前付きストリームを *io.StringIO* オブジェクトで一時的に置き換えるコンテキストマネージャです。

出力ストリームの使用例:

```
with captured_stdout() as stdout, captured_stderr() as stderr:
    print("hello")
    print("error", file=sys.stderr)
assert stdout.getvalue() == "hello\n"
assert stderr.getvalue() == "error\n"
```

入力ストリームの使用例:

```
with captured_stdin() as stdin:
    stdin.write('hello\n')
    stdin.seek(0)
    # call test code that consumes from sys.stdin
    captured = input()
self.assertEqual(captured, "hello")
```

test.support.**temp_dir**(*path=None*, *quiet=False*)

path に一時ディレクトリを作成し与えるコンテキストマネージャです。

If *path* is `None`, the temporary directory is created using *tempfile.mkdtemp()*. If *quiet* is `False`, the context manager raises an exception on error. Otherwise, if *path* is specified and cannot be created, only a warning is issued.

test.support.**change_cwd**(*path*, *quiet=False*)

カレントディレクトリを一時的に *path* に変更し与えるコンテキストマネージャです。

quiet が False の場合、コンテキストマネージャはエラーが起きると例外を送出します。それ以外の場合には、警告を出すだけでカレントディレクトリは同じままにしておきます。

test.support.**temp_cwd**(*name='tempcwd'*, *quiet=False*)

一時的に新しいディレクトリを作成し、カレントディレクトリ (current working directory, CWD) を変更するコンテキストマネージャです。

一時的にカレントディレクトリを変更する前に、カレントディレクトリに *name* という名前のディレクトリを作成します。*name* が None の場合は、一時ディレクトリは *tempfile.mkdtemp()* を使って作成されます。

quiet が False でカレントディレクトリの作成や変更ができない場合、例外を送出します。それ以外の場合には、警告を出すだけで元のカレントディレクトリが使われます。

test.support.**temp_umask**(*umask*)

一時的にプロセスの umask を設定するコンテキストマネージャ。

test.support.**transient_internet**(*resource_name*, ***, *timeout=30.0*, *errnos=()*)

A context manager that raises *ResourceDenied* when various issues with the internet connection manifest themselves as exceptions.

test.support.**disable_faulthandler**()

A context manager that replaces sys.stderr with sys.__stderr__.

test.support.**gc_collect**()

Force as many objects as possible to be collected. This is needed because timely deallocation is not guaranteed by the garbage collector. This means that __del__ methods may be called later than expected and weakrefs may remain alive for longer than expected.

test.support.**disable_gc**()

A context manager that disables the garbage collector upon entry and reenables it upon exit.

test.support.**swap_attr**(*obj*, *attr*, *new_val*)

Context manager to swap out an attribute with a new object.

使い方:

```
with swap_attr(obj, "attr", 5):
    ...
```

This will set obj.attr to 5 for the duration of the with block, restoring the old value at the end of the block. If attr doesn't exist on obj, it will be created and then deleted at the end of the block.

The old value (or None if it doesn't exist) will be assigned to the target of the "as" clause, if there is one.

test.support.**swap_item**(*obj*, *attr*, *new_val*)

Context manager to swap out an item with a new object.

使い方:

```
with swap_item(obj, "item", 5):
    ...
```

This will set `obj["item"]` to 5 for the duration of the `with` block, restoring the old value at the end of the block. If `item` doesn't exist on `obj`, it will be created and then deleted at the end of the block.

The old value (or `None` if it doesn't exist) will be assigned to the target of the "as" clause, if there is one.

test.support.wait_threads_exit(*timeout=60.0*)

Context manager to wait until all threads created in the `with` statement exit.

test.support.start_threads(*threads*, *unlock=None*)

Context manager to start *threads*. It attempts to join the threads upon exit.

test.support.calcobjsize(*fmt*)

Return *struct.calcsize()* for `nP{fmt}0n` or, if `gettotalrefcount` exists, `2PnP{fmt}0P`.

test.support.calcvobjsize(*fmt*)

Return *struct.calcsize()* for `nPn{fmt}0n` or, if `gettotalrefcount` exists, `2PnPn{fmt}0P`.

test.support.checksizeof(*test*, *o*, *size*)

For testcase *test*, assert that the `sys.getsizeof` for *o* plus the GC header size equals *size*.

test.support.can_symlink()

OS がシンボリックリンクをサポートする場合 `True` を返し、その他の場合は `False` を返します。

test.support.can_xattr()

Return `True` if the OS supports xattr, `False` otherwise.

@test.support.skip_unless_symlink

シンボリックリンクのサポートが必要なテストを実行することを表すデコレータ。

@test.support.skip_unless_xattr

A decorator for running tests that require support for xattr.

@test.support.skip_unless_bind_unix_socket

A decorator for running tests that require a functional bind() for Unix sockets.

@test.support.anticipate_failure(*condition*)

ある条件で *unittest.expectedFailure()* の印をテストに付けるデコレータ。このデコレータを使うときはいつも、関連する問題を指し示すコメントを付けておくべきです。

@test.support.run_with_locale(*catstr*, **locales*)

別のロケールで関数を実行し、完了したら適切に元の状態に戻すためのデコレータ。*catstr* は (例えば `"LC_ALL"` のような) ロケールカテゴリを文字列で表したものです。渡された *locales* が順々に試され、一番最初に出てきた妥当なロケールが使われます。

@test.support.**run_with_tz**(*tz*)

A decorator for running a function in a specific timezone, correctly resetting it after it has finished.

@test.support.**requires_freebsd_version**(**min_version*)

Decorator for the minimum version when running test on FreeBSD. If the FreeBSD version is less than the minimum, raise *unittest.SkipTest*.

@test.support.**requires_linux_version**(**min_version*)

Decorator for the minimum version when running test on Linux. If the Linux version is less than the minimum, raise *unittest.SkipTest*.

@test.support.**requires_mac_version**(**min_version*)

Decorator for the minimum version when running test on Mac OS X. If the MAC OS X version is less than the minimum, raise *unittest.SkipTest*.

@test.support.**requires_IEEE_754**

Decorator for skipping tests on non-IEEE 754 platforms.

@test.support.**requires_zlib**

Decorator for skipping tests if *zlib* doesn't exist.

@test.support.**requires_gzip**

Decorator for skipping tests if *gzip* doesn't exist.

@test.support.**requires_bz2**

Decorator for skipping tests if *bz2* doesn't exist.

@test.support.**requires_lzma**

Decorator for skipping tests if *lzma* doesn't exist.

@test.support.**requires_resource**(*resource*)

Decorator for skipping tests if *resource* is not available.

@test.support.**requires_docstrings**

Decorator for only running the test if *HAVE_DOCSTRINGS*.

@test.support.**cpython_only**(*test*)

Decorator for tests only applicable to CPython.

@test.support.**impl_detail**(*msg=None*, ***guards*)

Decorator for invoking *check_impl_detail()* on *guards*. If that returns `False`, then uses *msg* as the reason for skipping the test.

@test.support.**no_tracing**(*func*)

Decorator to temporarily turn off tracing for the duration of the test.

@test.support.**refcount_test**(*test*)

Decorator for tests which involve reference counting. The decorator does not run the test if it is not run by CPython. Any trace function is unset for the duration of the test to prevent unexpected refcounts caused by the trace function.

@test.support.reap_threads(*func*)

Decorator to ensure the threads are cleaned up even if the test fails.

@test.support.bigmemtest(*size*, *memuse*, *dry_run=True*)

Decorator for bigmem tests.

size is a requested size for the test (in arbitrary, test-interpreted units.) *memuse* is the number of bytes per unit for the test, or a good estimate of it. For example, a test that needs two byte buffers, of 4 GiB each, could be decorated with `@bigmemtest(size=_4G, memuse=2)`.

The *size* argument is normally passed to the decorated test method as an extra argument. If *dry_run* is `True`, the value passed to the test method may be less than the requested value. If *dry_run* is `False`, it means the test doesn't support dummy runs when `-M` is not specified.

@test.support.bigaddrspacetest(*f*)

Decorator for tests that fill the address space. *f* is the function to wrap.

test.support.make_bad_fd()

一時ファイルを開いた後に閉じ、そのファイル記述子を返すことで無効な記述子を作成します。

test.support.check_syntax_error(*testcase*, *statement*, *errtext=''*, ***, *lineno=None*, *offset=None*)

Test for syntax errors in *statement* by attempting to compile *statement*. *testcase* is the *unittest* instance for the test. *errtext* is the regular expression which should match the string representation of the raised *SyntaxError*. If *lineno* is not `None`, compares to the line of the exception. If *offset* is not `None`, compares to the offset of the exception.

test.support.check_syntax_warning(*testcase*, *statement*, *errtext=''*, ***, *lineno=1*, *offset=None*)

Test for syntax warning in *statement* by attempting to compile *statement*. Test also that the *SyntaxWarning* is emitted only once, and that it will be converted to a *SyntaxError* when turned into error. *testcase* is the *unittest* instance for the test. *errtext* is the regular expression which should match the string representation of the emitted *SyntaxWarning* and raised *SyntaxError*. If *lineno* is not `None`, compares to the line of the warning and exception. If *offset* is not `None`, compares to the offset of the exception.

バージョン 3.8 で追加.

test.support.open_urlresource(*url*, **args*, ***kw*)

Open *url*. If open fails, raises *TestFailed*.

test.support.import_module(*name*, *deprecated=False*, ***, *required_on()*)

この関数は *name* で指定されたモジュールをインポートして返します。通常のインポートと異なり、この関数はモジュールをインポートできなかった場合に *unittest.SkipTest* 例外を発生させます。

Module and package deprecation messages are suppressed during this import if *deprecated* is `True`. If a module is required on a platform but optional for others, set *required_on* to an iterable of platform prefixes which will be compared against *sys.platform*.

バージョン 3.1 で追加.

test.support.**import_fresh_module**(*name*, *fresh=()*, *blocked=()*, *deprecated=False*)

この関数は、*name* で指定された Python モジュールを、インポート前に sys.modules から削除することで新規にインポートしてそのコピーを返します。reload() 関数と違い、もとのモジュールはこの操作によって影響をうけません。

fresh は、同じようにインポート前に sys.modules から削除されるモジュール名の iterable です。

blocked もモジュール名のイテラブルで、インポート中にモジュールキャッシュ内でその名前を None に置き換えることで、そのモジュールをインポートしようとすると *ImportError* を発生させます。

指定されたモジュールと *fresh* や *blocked* 引数内のモジュール名はインポート前に保存され、フレッシュなインポートが完了したら sys.modules に戻されます。

deprecated が True の場合、インポート中はモジュールとパッケージの廃止メッセージが抑制されます。

指定したモジュールがインポートできなかった場合に、この関数は *ImportError* を送出します。

使用例:

```
# Get copies of the warnings module for testing without affecting the
# version being used by the rest of the test suite. One copy uses the
# C implementation, the other is forced to use the pure Python fallback
# implementation
py_warnings = import_fresh_module('warnings', blocked=['_warnings'])
c_warnings = import_fresh_module('warnings', fresh=['_warnings'])
```

バージョン 3.1 で追加.

test.support.**modules_setup**()

Return a copy of *sys.modules*.

test.support.**modules_cleanup**(*oldmodules*)

Remove modules except for *oldmodules* and encodings in order to preserve internal cache.

test.support.**threading_setup**()

Return current thread count and copy of dangling threads.

test.support.**threading_cleanup**(**original_values*)

Cleanup up threads not specified in *original_values*. Designed to emit a warning if a test leaves running threads in the background.

test.support.**join_thread**(*thread*, *timeout=30.0*)

Join a *thread* within *timeout*. Raise an *AssertionError* if thread is still alive after *timeout* seconds.

test.support.**reap_children**()

Use this at the end of test_main whenever sub-processes are started. This will help ensure that no extra children (zombies) stick around to hog resources and create problems when looking for refleaks.

test.support.**get_attribute**(*obj*, *name*)

Get an attribute, raising *unittest.SkipTest* if *AttributeError* is raised.

test.support.**bind_port**(*sock*, *host=HOST*)

Bind the socket to a free port and return the port number. Relies on ephemeral ports in order to ensure we are using an unbound port. This is important as many tests may be running simultaneously, especially in a buildbot environment. This method raises an exception if the `sock.family` is *AF_INET* and `sock.type` is *SOCK_STREAM*, and the socket has `SO_REUSEADDR` or `SO_REUSEPORT` set on it. Tests should never set these socket options for TCP/IP sockets. The only case for setting these options is testing multicasting via multiple UDP sockets.

Additionally, if the `SO_EXCLUSIVEADDRUSE` socket option is available (i.e. on Windows), it will be set on the socket. This will prevent anyone else from binding to our host/port for the duration of the test.

test.support.**bind_unix_socket**(*sock*, *addr*)

Bind a unix socket, raising *unittest.SkipTest* if *PermissionError* is raised.

test.support.**catch_threading_exception**()

Context manager catching *threading.Thread* exception using *threading.excepthook()*.

Attributes set when an exception is catched:

- `exc_type`
- `exc_value`
- `exc_traceback`
- `thread`

See *threading.excepthook()* documentation.

These attributes are deleted at the context manager exit.

使い方:

```
with support.catch_threading_exception() as cm:
    # code spawning a thread which raises an exception
    ...

    # check the thread exception, use cm attributes:
    # exc_type, exc_value, exc_traceback, thread
    ...

# exc_type, exc_value, exc_traceback, thread attributes of cm no longer
# exists at this point
# (to avoid reference cycles)
```

バージョン 3.8 で追加.

test.support.catch_unraisable_exception()

Context manager catching unraisable exception using *sys.unraisablehook()*.

Storing the exception value (cm.unraisable.exc_value) creates a reference cycle. The reference cycle is broken explicitly when the context manager exits.

Storing the object (cm.unraisable.object) can resurrect it if it is set to an object which is being finalized. Exiting the context manager clears the stored object.

使い方:

```
with support.catch_unraisable_exception() as cm:
    # code creating an "unraisable exception"
    ...

    # check the unraisable exception: use cm.unraisable
    ...

# cm.unraisable attribute no longer exists at this point
# (to break a reference cycle)
```

バージョン 3.8 で追加.

test.support.find_unused_port(*family=socket.AF_INET*, *socktype=socket.SOCK_STREAM*)

Returns an unused port that should be suitable for binding. This is achieved by creating a temporary socket with the same family and type as the sock parameter (default is *AF_INET*, *SOCK_STREAM*), and binding it to the specified host address (defaults to 0.0.0.0) with the port set to 0, eliciting an unused ephemeral port from the OS. The temporary socket is then closed and deleted, and the ephemeral port is returned.

Either this method or *bind_port()* should be used for any tests where a server socket needs to be bound to a particular port for the duration of the test. Which one to use depends on whether the calling code is creating a Python socket, or if an unused port needs to be provided in a constructor or passed to an external program (i.e. the -accept argument to openssl's s_server mode). Always prefer *bind_port()* over *find_unused_port()* where possible. Using a hard coded port is discouraged since it can make multiple instances of the test impossible to run simultaneously, which is a problem for buildbots.

test.support.load_package_tests(*pkg_dir*, *loader*, *standard_tests*, *pattern*)

Generic implementation of the *unittest* load_tests protocol for use in test packages. *pkg_dir* is the root directory of the package; *loader*, *standard_tests*, and *pattern* are the arguments expected by load_tests. In simple cases, the test package's __init__.py can be the following:

```
import os
from test.support import load_package_tests

def load_tests(*args):
    return load_package_tests(os.path.dirname(__file__), *args)
```

test.support.**fs_is_case_insensitive**(*directory*)

Return `True` if the file system for *directory* is case-insensitive.

test.support.**detect_api_mismatch**(*ref_api*, *other_api*, *, *ignore=()*)

Returns the set of attributes, functions or methods of *ref_api* not found on *other_api*, except for a defined list of items to be ignored in this check specified in *ignore*.

By default this skips private attributes beginning with '_' but includes all magic methods, i.e. those starting and ending in '___'.

バージョン 3.5 で追加.

test.support.**patch**(*test_instance*, *object_to_patch*, *attr_name*, *new_value*)

Override *object_to_patch.attr_name* with *new_value*. Also add cleanup procedure to *test_instance* to restore *object_to_patch* for *attr_name*. The *attr_name* should be a valid attribute for *object_to_patch*.

test.support.**run_in_subinterp**(*code*)

Run *code* in subinterpreter. Raise *unittest.SkipTest* if *tracemalloc* is enabled.

test.support.**check_free_after_iterating**(*test*, *iter*, *cls*, *args=()*)

Assert that *iter* is deallocated after iterating.

test.support.**missing_compiler_executable**(*cmd_names=[]*)

Check for the existence of the compiler executables whose names are listed in *cmd_names* or all the compiler executables when *cmd_names* is empty and return the first missing executable or `None` when none is found missing.

test.support.**check__all__**(*test_case*, *module*, *name_of_module=None*, *extra=()*, *blacklist=()*)

Assert that the `__all__` variable of *module* contains all public names.

The module's public names (its API) are detected automatically based on whether they match the public name convention and were defined in *module*.

The *name_of_module* argument can specify (as a string or tuple thereof) what module(s) an API could be defined in order to be detected as a public API. One case for this is when *module* imports part of its public API from other modules, possibly a C backend (like `csv` and its `_csv`).

The *extra* argument can be a set of names that wouldn't otherwise be automatically detected as "public", like objects without a proper `__module__` attribute. If provided, it will be added to the automatically detected ones.

The *blacklist* argument can be a set of names that must not be treated as part of the public API even though their names indicate otherwise.

使用例:

```
import bar
import foo
```

(次のページに続く)

(前のページからの続き)

```
import unittest
from test import support

class MiscTestCase(unittest.TestCase):
    def test__all__(self):
        support.check__all__(self, foo)

class OtherTestCase(unittest.TestCase):
    def test__all__(self):
        extra = {'BAR_CONST', 'FOO_CONST'}
        blacklist = {'baz'}  # Undocumented name.
        # bar imports part of its API from _bar.
        support.check__all__(self, bar, ('bar', '_bar'),
                             extra=extra, blacklist=blacklist)
```

バージョン 3.6 で追加.

test.support.adjust_int_max_str_digits(*max_digits*)

This function returns a context manager that will change the global *sys.set_int_max_str_digits()* setting for the duration of the context to allow execution of test code that needs a different limit on the number of digits when converting between an integer and string.

バージョン 3.8.14 で追加.

test.support モジュールでは、以下のクラスを定義しています:

class test.support.TransientResource(*exc*, ***kwargs*)

このクラスのインスタンスはコンテキストマネージャーで、指定された型の例外が発生した場合に *ResourceDenied* 例外を発生させます。キーワード引数は全て、with 文の中で発生した全ての例外の属性名/属性値と比較されます。全てのキーワード引数が例外の属性に一致した場合に、*ResourceDenied* 例外が発生します。

class test.support.EnvironmentVarGuard

一時的に環境変数をセット・アンセットするためのクラスです。このクラスのインスタンスはコンテキストマネージャーとして利用されます。また、os.environ に対する参照・更新を行う完全な辞書のインタフェースを持ちます。コンテキストマネージャーが終了した時、このインスタンス経由で環境変数へ行った全ての変更はロールバックされます。

バージョン 3.1 で変更: 辞書のインタフェースを追加しました。

EnvironmentVarGuard.set(*envvar*, *value*)

一時的に、envvar を value にセットします。

EnvironmentVarGuard.unset(*envvar*)

一時的に envvar をアンセットします。

class test.support.SuppressCrashReport

A context manager used to try to prevent crash dialog popups on tests that are expected to crash

a subprocess.

On Windows, it disables Windows Error Reporting dialogs using SetErrorMode.

On UNIX, *resource.setrlimit()* is used to set *resource.RLIMIT_CORE*'s soft limit to 0 to prevent coredump file creation.

On both platforms, the old value is restored by `__exit__()`.

class test.support.CleanImport(**module_names*)

A context manager to force import to return a new module reference. This is useful for testing module-level behaviors, such as the emission of a DeprecationWarning on import. Example usage:

```
with CleanImport('foo'):
    importlib.import_module('foo')  # New reference.
```

class test.support.DirsOnSysPath(**paths*)

A context manager to temporarily add directories to sys.path.

This makes a copy of *sys.path*, appends any directories given as positional arguments, then reverts *sys.path* to the copied settings when the context ends.

Note that *all* *sys.path* modifications in the body of the context manager, including replacement of the object, will be reverted at the end of the block.

class test.support.SaveSignals

Class to save and restore signal handlers registered by the Python signal handler.

class test.support.Matcher

matches(*self*, *d*, ***kwargs*)

Try to match a single dict with the supplied arguments.

match_value(*self*, *k*, *dv*, *v*)

Try to match a single stored value (*dv*) with a supplied value (*v*).

class test.support.WarningsRecorder

ユニットテスト時に warning を記録するためのクラスです。上の、*check_warnings()* のドキュメントを参照してください。

class test.support.BasicTestRunner

run(*test*)

Run *test* and return the result.

class test.support.TestHandler(*logging.handlers.BufferingHandler*)

Class for logging support.

class test.support.FakePath(*path*)

Simple *path-like object*. It implements the `__fspath__()` method which just returns the *path*

argument. If *path* is an exception, it will be raised in `__fspath__()`.

26.10 `test.support.script_helper` --- Utilities for the Python execution tests

The *test.support.script_helper* module provides support for Python's script execution tests.

`test.support.script_helper.interpreter_requires_environment()`

Return `True` if `sys.executable interpreter` requires environment variables in order to be able to run at all.

This is designed to be used with `@unittest.skipIf()` to annotate tests that need to use an `assert_python*()` function to launch an isolated mode (`-I`) or no environment mode (`-E`) sub-interpreter process.

A normal build & test does not run into this situation but it can happen when trying to run the standard library test suite from an interpreter that doesn't have an obvious home with Python's current home finding logic.

Setting `PYTHONHOME` is one way to get most of the testsuite to run in that situation. `PYTHONPATH` or `PYTHONUSERSITE` are other common environment variables that might impact whether or not the interpreter can start.

`test.support.script_helper.run_python_until_end(*args, **env_vars)`

Set up the environment based on *env_vars* for running the interpreter in a subprocess. The values can include `__isolated`, `__cleanenv`, `__cwd`, and `TERM`.

`test.support.script_helper.assert_python_ok(*args, **env_vars)`

Assert that running the interpreter with *args* and optional environment variables *env_vars* succeeds (`rc == 0`) and return a `(return code, stdout, stderr)` tuple.

If the `__cleanenv` keyword is set, *env_vars* is used as a fresh environment.

Python is started in isolated mode (command line option `-I`), except if the `__isolated` keyword is set to `False`.

`test.support.script_helper.assert_python_failure(*args, **env_vars)`

Assert that running the interpreter with *args* and optional environment variables *env_vars* fails (`rc != 0`) and return a `(return code, stdout, stderr)` tuple.

See *assert_python_ok()* for more options.

`test.support.script_helper.spawn_python(*args, stdout=subprocess.PIPE, stderr=subprocess.STDOUT, **kw)`

Run a Python subprocess with the given arguments.

kw is extra keyword args to pass to *subprocess.Popen()*. Returns a *subprocess.Popen* object.

test.support.script_helper.**kill_python**(*p*)

Run the given *subprocess.Popen* process until completion and return stdout.

test.support.script_helper.**make_script**(*script_dir*, *script_basename*, *source*, *omit_suffix=False*)

Create script containing *source* in path *script_dir* and *script_basename*. If *omit_suffix* is `False`, append `.py` to the name. Return the full script path.

test.support.script_helper.**make_zip_script**(*zip_dir*, *zip_basename*, *script_name*, *name_in_zip=None*)

Create zip file at *zip_dir* and *zip_basename* with extension `zip` which contains the files in *script_name*. *name_in_zip* is the archive name. Return a tuple containing `(full path, full path of archive name)`.

test.support.script_helper.**make_pkg**(*pkg_dir*, *init_source=''*)

Create a directory named *pkg_dir* containing an `__init__` file with *init_source* as its contents.

test.support.script_helper.**make_zip_pkg**(*zip_dir*, *zip_basename*, *pkg_name*, *script_basename*, *source*, *depth=1*, *compiled=False*)

Create a zip package directory with a path of *zip_dir* and *zip_basename* containing an empty `__init__` file and a file *script_basename* containing the *source*. If *compiled* is `True`, both source files will be compiled and added to the zip package. Return a tuple of the full zip path and the archive name for the zip file.

Python の開発モードも参照してください: `-X dev` オプションおよび `PYTHONDEVMODE` 環境変数。

第

TWENTYSEVEN

デバッグとプロファイル

ここに含まれるライブラリは Python での開発を手助けするものです。デバッガを使うと、コードのステップ実行や、スタックフレームの解析、ブレークポイントの設定などができます。プロファイラはコードを実行して実行時間の詳細を提供し、プログラムのボトルネックを特定できるようにします。

27.1 Audit events table

This table contains all events raised by *sys.audit()* or `PySys_Audit()` calls throughout the CPython runtime and the standard library. These calls were added in 3.8.0 or later (see **PEP 578**).

See *sys.addaudithook()* and `PySys_AddAuditHook()` for information on handling these events.

CPython implementation detail: This table is generated from the CPython documentation, and may not represent events raised by other implementations. See your runtime specific documentation for actual events raised.

Audit event	Arguments
array.___new___	`typecode`, `initializer`
builtins.breakpoint	`breakpointhook`
builtins.id	`id`
builtins.input	`prompt`
builtins.input/result	`result`
code.___new___	`code`, `filename`, `name`, `argcount`, `posonlyargcount`, `kwonlyargcount`, `nlocals`
compile	`source`, `filename`
cpython.PyInterpreterState_Clear	
cpython.PyInterpreterState_New	
cpython._PySys_ClearAuditHooks	
cpython.run_command	`command`
cpython.run_file	`filename`
cpython.run_interactivehook	`hook`
cpython.run_module	`module-name`
cpython.run_startup	`filename`

表 1 – 前のページからの続き

Audit event	Arguments
cpython.run_stdin	
ctypes.addressof	obj
ctypes.call_function	func_pointer, arguments
ctypes.cdata	address
ctypes.cdata/buffer	pointer, size, offset
ctypes.create_string_buffer	init, size
ctypes.create_unicode_buffer	init, size
ctypes.dlopen	name
ctypes.dlsym	library, name
ctypes.dlsym/handle	handle, name
ctypes.get_errno	
ctypes.get_last_error	
ctypes.seh_exception	code
ctypes.set_errno	errno
ctypes.set_last_error	error
ctypes.string_at	address, size
ctypes.wstring_at	address, size
ensurepip.bootstrap	root
exec	code_object
fcntl.fcntl	fd, cmd, arg
fcntl.flock	fd, operation
fcntl.ioctl	fd, request, arg
fcntl.lockf	fd, cmd, len, start, whence
ftplib.connect	self, host, port
ftplib.sendcmd	self, cmd
function.__new__	code
gc.get_objects	generation
gc.get_referents	objs
gc.get_referrers	objs
glob.glob	pathname, recursive
imaplib.open	self, host, port
imaplib.send	self, data
import	module, filename, sys.path, sys.meta_path, sys.path_hooks
mmap.__new__	fileno, length, access, offset
msvcrt.get_osfhandle	fd
msvcrt.locking	fd, mode, nbytes
msvcrt.open_osfhandle	handle, flags
nntplib.connect	self, host, port
nntplib.putline	self, line
object.__delattr__	obj, name

表 1 – 前のページからの続き

Audit event	Arguments
object.__getattr__	obj, name
object.__setattr__	obj, name, value
open	file, mode, flags
os.add_dll_directory	path
os.chdir	path
os.chflags	path, flags
os.chmod	path, mode, dir_fd
os.chown	path, uid, gid, dir_fd
os.exec	path, args, env
os.fork	
os.forkpty	
os.getxattr	path, attribute
os.kill	pid, sig
os.killpg	pgid, sig
os.link	src, dst, src_dir_fd, dst_dir_fd
os.listdir	path
os.listxattr	path
os.lockf	fd, cmd, len
os.mkdir	path, mode, dir_fd
os.posix_spawn	path, argv, env
os.putenv	key, value
os.remove	path, dir_fd
os.removexattr	path, attribute
os.rename	src, dst, src_dir_fd, dst_dir_fd
os.rmdir	path, dir_fd
os.scandir	path
os.setxattr	path, attribute, value, flags
os.spawn	mode, path, args, env
os.startfile	path, operation
os.symlink	src, dst, dir_fd
os.system	command
os.truncate	fd, length
os.unsetenv	key
os.utime	path, times, ns, dir_fd
pdb.Pdb	
pickle.find_class	module, name
poplib.connect	self, host, port
poplib.putline	self, line
pty.spawn	argv
resource.prlimit	pid, resource, limits

表 1 – 前のページからの続き

Audit event	Arguments
resource.setrlimit	resource, limits
setopencodehook	
shutil.chown	path, user, group
shutil.copyfile	src, dst
shutil.copymode	src, dst
shutil.copystat	src, dst
shutil.copytree	src, dst
shutil.make_archive	base_name, format, root_dir, base_dir
shutil.move	src, dst
shutil.rmtree	path
shutil.unpack_archive	filename, extract_dir, format
signal.pthread_kill	thread_id, signalnum
smtplib.connect	self, host, port
smtplib.send	self, data
socket.___new___	self, family, type, protocol
socket.bind	self, address
socket.connect	self, address
socket.getaddrinfo	host, port, family, type, protocol
socket.gethostbyaddr	ip_address
socket.gethostbyname	hostname
socket.gethostname	
socket.getnameinfo	sockaddr
socket.getservbyname	servicename, protocolname
socket.getservbyport	port, protocolname
socket.sendmsg	self, address
socket.sendto	self, address
socket.sethostname	name
sqlite3.connect	database
subprocess.Popen	executable, args, cwd, env
sys._current_frames	
sys._getframe	
sys.addaudithook	
sys.excepthook	hook, type, value, traceback
sys.set_asyncgen_hooks_finalizer	
sys.set_asyncgen_hooks_firstiter	
sys.setprofile	
sys.settrace	
sys.unraisablehook	hook, unraisable
syslog.closelog	
syslog.openlog	ident, logoption, facility

表 1 – 前のページからの続き

Audit event	Arguments
syslog.setlogmask	maskpri
syslog.syslog	priority, message
telnetlib.Telnet.open	self, host, port
telnetlib.Telnet.write	self, buffer
tempfile.mkdtemp	fullpath
tempfile.mkstemp	fullpath
urllib.Request	fullurl, data, headers, method
webbrowser.open	url
winreg.ConnectRegistry	computer_name, key
winreg.CreateKey	key, sub_key, access
winreg.DeleteKey	key, sub_key, access
winreg.DeleteValue	key, value
winreg.DisableReflectionKey	key
winreg.EnableReflectionKey	key
winreg.EnumKey	key, index
winreg.EnumValue	key, index
winreg.ExpandEnvironmentStrings	str
winreg.LoadKey	key, sub_key, file_name
winreg.OpenKey	key, sub_key, access
winreg.OpenKey/result	key
winreg.PyHKEY.Detach	key
winreg.QueryInfoKey	key
winreg.QueryReflectionKey	key
winreg.QueryValue	key, sub_key, value_name
winreg.SaveKey	key, file_name
winreg.SetValue	key, sub_key, type, value

The following events are raised internally and do not correspond to any public API of CPython:

Audit event	引数
_winapi.CreateFile	file_name, desired_access, share_mode, creation_disposition, flags_and_attributes
_winapi.CreateJunction	src_path, dst_path
_winapi.CreateNamedPipe	name, open_mode, pipe_mode
_winapi.CreatePipe	
_winapi.CreateProcess	application_name, command_line, current_directory
_winapi.OpenProcess	process_id, desired_access
_winapi.TerminateProcess	handle, exit_code
ctypes.PyObj_FromPtr	obj

27.2 bdb --- デバッガーフレームワーク

ソースコード: Lib/bdb.py

---

bdb モジュールは、ブレークポイントを設定したり、デバッガー経由で実行を管理するような、基本的なデバッガー機能を提供します。

以下の例外が定義されています:

exception bdb.BdbQuit

Bdb クラスが、デバッガーを終了させるために投げる例外。

bdb モジュールは 2 つのクラスを定義しています:

class bdb.Breakpoint(*self*, *file*, *line*, *temporary=0*, *cond=None*, *funcname=None*)

このクラスはテンポラリブレークポイント、無視するカウント、無効化と再有効化、条件付きブレークポイントを実装しています。

ブレークポイントは bpbynumber という名前のリストで番号によりインデックスされ、bplist により (file, line) の形でインデックスされます。bpbynumber は *Breakpoint* クラスのインスタンスを指しています。一方 bplist は、同じ行に複数のブレークポイントが設定される場合があるので、インスタンスのリストを指しています。

ブレークポイントを作るとき、設定されるファイル名は正規化されていなければなりません。*funcname* が設定されたとき、ブレークポイントはその関数の最初の行が実行されたときにヒットカウントにカウントされます。条件付ブレークポイントは毎回カウントされます。

Breakpoint インスタンスは以下のメソッドを持ちます:

deleteMe()
このブレークポイントをファイル/行に関連付けられたリストから削除します。このブレークポイントがその行に設定された最後のブレークポイントだった場合、そのファイル/行に対するエントリ自体を削除します。

enable()
このブレークポイントを有効にします。

disable()
このブレークポイントを無効にします。

bpformat()
ブレークポイントに関する情報を持つ文字列をフォーマットして返します:

- ブレークポイント番号。
- テンポラリブレークポイントかどうか。
- ファイル/行の位置。
- ブレークする条件。
- 次の N 回無視されるか。
- ヒットカウント。

バージョン 3.2 で追加.

bpprint(*out=None*)
ファイル *out* に、またはそれが None の場合は標準出力に、*bpformat()* の出力を表示する。

class bdb.Bdb(*skip=None*)
Bdb クラスは一般的な Python デバッガーの基本クラスとして振舞います。

このクラスはトレース機能の詳細を扱います。ユーザーとのインタラクションは、派生クラスが実装するべきです。標準ライブラリのデバッガクラス (*pdb.Pdb*) がその利用例です。

skip 引数は、もし与えられたならグロブ形式のモジュール名パターンの iterable でなければなりません。デバッガはこれらのパターンのどれかにマッチするモジュールで発生したフレームにステップインしなくなります。フレームが特定のモジュールで発生したかどうかは、フレームのグローバル変数の __name__ によって決定されます。

バージョン 3.1 で追加: *skip* 引数が追加されました。

以下の *Bdb* のメソッドは、通常オーバーライドする必要はありません。

canonic(*filename*)
標準化されたファイル名を取得するための補助関数。標準化されたファイル名とは、(大文字小文字を区別しないファイルシステムにおいて) 大文字小文字を正規化し、絶対パスにしたものです。ファイル名が ”<” と ”>” で囲まれていた場合はそれを取り除いたものです。

reset()

botframe, stopframe, returnframe, quitting 属性を、デバッグを始められる状態に設定します。

trace_dispatch(*frame*, *event*, *arg*)

この関数は、デバッグされているフレームのトレース関数としてインストールされます。戻り値は新しいトレース関数 (殆どの場合はこの関数自身) です。

デフォルトの実装は、実行しようとしている *event* (文字列として渡されます) の種類に基づいてフレームのディスパッチ方法を決定します。*event* は次のうちのどれかです:

- "line": 新しい行を実行しようとしています。
- "call": 関数が呼び出されているか、別のコードブロックに入ります。
- "return": 関数か別のコードブロックから return しようとしています。
- "exception": 例外が発生しました。
- "c_call": C 関数を呼び出そうとしています。
- "c_return": C 関数から return しました。
- "c_exception": C 関数が例外を発生させました。

Python のイベントに対しては、以下の専用の関数群が呼ばれます。C のイベントに対しては何もしません。

arg 引数は以前のイベントに依存します。

トレース関数についてのより詳しい情報は、*sys.settrace()* のドキュメントを参照してください。コードとフレームオブジェクトについてのより詳しい情報は、types を参照してください。

dispatch_line(*frame*)

デバッガーが現在の行で止まるべきであれば、*user_line()* メソッド (サブクラスでオーバーライドされる) を呼び出します。Bdb.quitting フラグ (*user_line()* から設定できます) が設定されていた場合、*BdbQuit* 例外を発生させます。このスコープのこれからのトレースのために、*trace_dispatch()* メソッドの参照を返します。

dispatch_call(*frame*, *arg*)

デバッガーがこの関数呼び出しで止まるべきであれば、*user_call()* メソッド (サブクラスでオーバーライドされる) を呼び出します。Bdb.quitting フラグ (*user_call()* から設定できます) が設定されていた場合、*BdbQuit* 例外を発生させます。このスコープのこれからのトレースのために、*trace_dispatch()* メソッドの参照を返します。

dispatch_return(*frame*, *arg*)

デバッガーがこの関数からのリターンで止まるべきであれば、*user_return()* メソッド (サブクラスでオーバーライドされる) を呼び出します。Bdb.quitting フラグ (*user_return()* から設定できます) が設定されていた場合、*BdbQuit* 例外を発生させます。このスコープのこれからのトレースのために、*trace_dispatch()* メソッドの参照を返します。

dispatch_exception(*frame*, *arg*)

デバッガーがこの例外発生で止まるべきであれば、*user_exception()* メソッド (サブクラスでオーバーライドされる) を呼び出します。Bdb.quitting フラグ (*user_exception()* から設定できます) が設定されていた場合、*BdbQuit* 例外を発生させます。このスコープのこれからのトレースのために、*trace_dispatch()* メソッドの参照を返します。

通常、継承クラスは以下のメソッド群をオーバーライドしません。しかし、停止やブレークポイント機能を再定義したい場合には、オーバーライドすることもあります。

stop_here(*frame*)

このメソッドは *frame* がコールスタック中で botframe よりも下にあるかチェックします。botframe はデバッグを開始したフレームです。

break_here(*frame*)

このメソッドは、*frame* に属するファイル名と行に、あるいは、少なくとも現在の関数にブレークポイントがあるかどうかをチェックします。ブレークポイントがテンポラリブレークポイントだった場合、このメソッドはそのブレークポイントを削除します。

break_anywhere(*frame*)

このメソッドは、現在のフレームのファイル名の中にブレークポイントが存在するかどうかをチェックします。

継承クラスはデバッガー操作をするために以下のメソッド群をオーバーライドするべきです。

user_call(*frame*, *argument_list*)

このメソッドは、呼ばれた関数の中でブレークする必要がある可能性がある場合に、*dispatch_call()* から呼び出されます。

user_line(*frame*)

このメソッドは、*stop_here()* か *break_here()* が True を返したときに、*dispatch_line()* から呼び出されます。

user_return(*frame*, *return_value*)

このメソッドは、*stop_here()* が True を返したときに、*dispatch_return()* から呼び出されます。

user_exception(*frame*, *exc_info*)

このメソッドは、*stop_here()* が True を返したときに、*dispatch_exception()* から呼び出されます。

do_clear(*arg*)

ブレークポイントがテンポラリブレークポイントだったときに、それをどう削除するかを決定します。

継承クラスはこのメソッドを実装しなければなりません。

継承クラスとクライアントは、ステップ状態に影響を及ぼすために以下のメソッドを呼び出すことができます。

set_step()
コードの次の行でストップします。

set_next(*frame*)
与えられたフレームかそれより下 (のフレーム) にある、次の行でストップします。

set_return(*frame*)
指定されたフレームから抜けるときにストップします。

set_until(*frame*)
現在の行番号よりも大きい行番号に到達したとき、あるいは、現在のフレームから戻るときにストップします。

set_trace([*frame*])
frame からデバッグを開始します。*frame* が指定されなかった場合、デバッグは呼び出し元のフレームから開始します。

set_continue()
ブレークポイントに到達するか終了したときにストップします。もしブレークポイントが 1 つも無い場合、システムのトレース関数を None に設定します。

set_quit()
quitting 属性を True に設定します。これにより、次回の dispatch_*() メソッドのどれかの呼び出しで、*BdbQuit* 例外を発生させます。

継承クラスとクライアントは以下のメソッドをブレークポイント操作に利用できます。これらのメソッドは、何か悪いことがあればエラーメッセージを含む文字列を返し、すべてが順調であれば None を返します。

set_break(*filename*, *lineno*, *temporary=0*, *cond*, *funcname*)
新しいブレークポイントを設定します。引数の *lineno* 行が *filename* に存在しない場合、エラーメッセージを返します。*filename* は、*canonic()* メソッドで説明されているような、標準形である必要があります。

clear_break(*filename*, *lineno*)
filename の *lineno* 行にあるブレークポイントを削除します。もしブレークポイントが無かった場合、エラーメッセージを返します。

clear_bpbynumber(*arg*)
Breakpoint.bpbynumber の中で *arg* のインデックスを持つブレークポイントを削除します。*arg* が数値でないか範囲外の場合、エラーメッセージを返します。

clear_all_file_breaks(*filename*)
filename に含まれるすべてのブレークポイントを削除します。もしブレークポイントが無い場合、エラーメッセージを返します。

clear_all_breaks()
すべてのブレークポイントを削除します。

get_bpbynumber(*arg*)

与えられた数値によって指定されるブレークポイントを返します。*arg* が文字列なら数値に変換されます。*arg* が非数値の文字列である場合、指定されたブレークポイントが存在しないか削除された場合、*ValueError* が上げられます。

バージョン 3.2 で追加.

get_break(*filename*, *lineno*)
filename の *lineno* にブレークポイントが存在するかどうかをチェックします。

get_breaks(*filename*, *lineno*)
filename の *lineno* にあるすべてのブレークポイントを返します。ブレークポイントが存在しない場合は空のリストを返します。

get_file_breaks(*filename*)
filename の中のすべてのブレークポイントを返します。ブレークポイントが存在しない場合は空のリストを返します。

get_all_breaks()
セットされているすべてのブレークポイントを返します。

継承クラスとクライアントは以下のメソッドを呼んでスタックトレースを表現するデータ構造を取得することができます。

get_stack(*f*, *t*)
与えられたフレームおよび上位 (呼び出し側) と下位のすべてのフレームに対するレコードのリストと、上位フレームのサイズを得ます。

format_stack_entry(*frame_lineno*, *lprefix=': '*)
`(frame, lineno)` で指定されたスタックエントリに関する次のような情報を持つ文字列を返します:

- そのフレームを含むファイル名の標準形。
- 関数名、もしくは `"<lambda>"`。
- 入力された引数。
- 戻り値。
- (あれば) その行のコード。

以下の 2 つのメソッドは、文字列として渡された **文** をデバッグするもので、クライアントから利用されます。

run(*cmd*, *globals=None*, *locals=None*)
exec() 関数を利用して文を実行しデバッグします。*globals* はデフォルトでは `__main__.__dict__` で、*locals* はデフォルトでは *globals* です。

runeval(*expr*, *globals=None*, *locals=None*)
eval() 関数を利用して式を実行しデバッグします。*globals* と *locals* は *run()* と同じ意味です。

runctx(*cmd*, *globals*, *locals*)
後方互換性のためのメソッドです。*run()* を使ってください。

runcall(*func*, **args*, ***kwds*)
1 つの関数呼び出しをデバッグし、その結果を返します。

最後に、このモジュールは以下の関数を提供しています:

bdb.checkfuncname(*b*, *frame*)
この場所でブレークする必要があるかどうかを、ブレークポイント *b* が設定された方法に依存する方法でチェックします。

ブレークポイントが行番号で設定されていた場合、この関数は `b.line` が、同じく引数として与えられた *frame* の中の行に一致するかどうかをチェックします。ブレークポイントが関数名で設定されていた場合、この関数は *frame* が指定された関数のものであるかどうかと、その関数の最初の行であるかどうかをチェックします。

bdb.effective(*file*, *line*, *frame*)
指定されたソースコード中の行に (有効な) ブレークポイントがあるかどうかを判断します。ブレークポイントと、テンポラリブレークポイントを削除して良いかどうかを示すフラグからなるタプルを返します。マッチするブレークポイントが存在しない場合は `(None, None)` を返します。

bdb.set_trace()
Bdb クラスのインスタンスを使って、呼び出し元のフレームからデバッグを開始します。

27.3 faulthandler --- Python traceback のダンプ

バージョン 3.3 で追加.

このモジュールは、例外発生時、タイムアウト時、ユーザシグナルの発生時などのタイミングで python traceback を明示的にダンプするための関数を含んでいます。これらのシグナル、SIGSEGV、SIGFPE、SIGABRT、SIGBUS、SIGILL に対するフォールトハンドラをインストールするには *faulthandler.enable()* を実行してください。python 起動時に有効にするには環境変数 PYTHONFAULTHANDLER を設定するか、コマンドライン引数に `-X faulthandler` を指定してください。

Python のフォールトハンドラは、apport や Windows のフォールトハンドラのようなシステムフォールトハンドラと互換性があります。このモジュールは `sigaltstack()` 関数が使用可能であればシグナルハンドラ用に代替スタックを利用します。これによってスタックオーバーフロー時にもスタックトレースを出力することができます。

フォールトハンドラは絶望的なケースで呼び出されます。そのためシグナルセーフな関数しか使うことができません (例: ヒープメモリ上にメモリ確保はできません)。この制限により、traceback のダンプ機能は通常の Python の traceback と比べてごく僅かなものです:

- ASCII のみサポートされます。エンコード時には `backslashreplace` エラーハンドラを使用します。
- すべての文字列は 500 文字以内に制限されています。

- ファイル名、関数名、行数のみ表示します。(ソースコードの表示はありません)
- 100 フレーム、100 スレッドに制限されています。
- 順番は保持されます: 最新の呼び出しが最初に表示されます。

デフォルトでは、Python の traceback は *sys.stderr* に書き出されます。traceback を見るには、対象アプリケーションはターミナル上で実行しなければなりません。*faulthandler.enable()* に渡す引数によってログファイルを指定することができます。

モジュールは C 言語で実装されているので、アプリのクラッシュ時でも Python がデッドロックした場合でもダンプができます。

27.3.1 traceback のダンプ

faulthandler.**dump_traceback**(*file=sys.stderr*, *all_threads=True*)

全スレッドの traceback を *file* へダンプします。もし *all_threads* が `False` であれば、現在のスレッドのみダンプします。

バージョン 3.5 で変更: Added support for passing file descriptor to this function.

27.3.2 フォールトハンドラの状態

faulthandler.**enable**(*file=sys.stderr*, *all_threads=True*)

フォールトハンドラを有効にします。`SIGSEGV`、`SIGFPE`、`SIGABRT`、`SIGBUS`、`SIGILL` シグナルに対して Python の traceback をダンプするハンドラをインストールします。もし *all_threads* が `True` であれば、すべての実行中のスレッドについて traceback をダンプします。そうでなければ現在のスレッドのみダンプします。

The *file* must be kept open until the fault handler is disabled: see *issue with file descriptors*.

バージョン 3.5 で変更: Added support for passing file descriptor to this function.

バージョン 3.6 で変更: On Windows, a handler for Windows exception is also installed.

faulthandler.**disable**()

フォールトハンドラを無効にします: *enable()* によってインストールされたシグナルハンドラをアンインストールします。

faulthandler.**is_enabled**()

フォールトハンドラが有効かどうかチェックします。

27.3.3 タイムアウト後に traceback をダンプする

faulthandler.**dump_traceback_later**(*timeout*, *repeat=False*, *file=sys.stderr*, *exit=False*)

timeout 秒経過後か、*repeat* が True の場合は *timeout* 秒おきに全スレッドの traceback をダンプします。もし *exit* が True であれば traceback をダンプした後、status=1 で _exit() を呼び出します。(注: _exit() を呼び出すとプロセスを即座に終了します。つまりファイルバッファのクリアといった終了処理を行いません。) 関数が 2 回呼ばれた場合、最新の呼び出しが前回の呼び出しパラメータを引き継いでタイムアウト時間をリセットします。タイマーの分解能は 1 秒未満です。

The *file* must be kept open until the traceback is dumped or *cancel_dump_traceback_later()* is called: see *issue with file descriptors*.

This function is implemented using a watchdog thread.

バージョン 3.7 で変更: This function is now always available.

バージョン 3.5 で変更: Added support for passing file descriptor to this function.

faulthandler.**cancel_dump_traceback_later**()

dump_traceback_later() の最新の呼び出しをキャンセルします。

27.3.4 ユーザシグナルに対して traceback をダンプする

faulthandler.**register**(*signum*, *file=sys.stderr*, *all_threads=True*, *chain=False*)

ユーザシグナルを登録します: すべてのスレッドで traceback をダンプするために *signum* シグナルをインストールします。ただし *all_threads* が False であれば現在のスレッドのみ *file* にダンプします。もし chain が True であれば以前のハンドラも呼び出します。

The *file* must be kept open until the signal is unregistered by *unregister()*: see *issue with file descriptors*.

Windows では利用不可です。

バージョン 3.5 で変更: Added support for passing file descriptor to this function.

faulthandler.**unregister**(*signum*)

ユーザシグナルを登録解除します: *register()* でインストールした *signum* シグナルハンドラをアンインストールします。シグナルが登録された場合は True を返し、そうでなければ False を返します。

Windows では利用不可です。

27.3.5 ファイル記述子の問題

enable()、*dump_traceback_later()* ならびに *register()* は引数 *file* に渡されたファイル記述子を保持します。ファイルが閉じられファイル記述子が新しいファイルで再利用された場合や、*os.dup2()* の使用でファイル記述子が置き換えた場合、traceback の結果は別のファイルへ書き込まれます。ファイルが置き換えられた場合は、毎回これらの関数を呼び出しなおしてください。

27.3.6 使用例

フォールトハンドラを有効化・無効化したときの Linux でのセグメンテーションフォールトの例:

```
$ python3 -c "import ctypes; ctypes.string_at(0)"
Segmentation fault

$ python3 -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault

Current thread 0x00007fb899f39700 (most recent call first):
  File "/home/python/cpython/Lib/ctypes/__init__.py", line 486 in string_at
  File "<stdin>", line 1 in <module>
Segmentation fault
```

27.4 pdb --- Python デバッガ

ソースコード: Lib/pdb.py

モジュール *pdb* は Python プログラム用の対話型ソースコードデバッガを定義します。(条件付き) ブレークポイントの設定やソース行レベルでのシングルステップ実行、スタックフレームのインスペクション、ソースコードリスティングおよびあらゆるスタックフレームのコンテキストにおける任意の Python コードの評価をサポートしています。事後解析デバッギングもサポートし、プログラムの制御下で呼び出すことができます。

デバッガーは拡張可能です -- 実際にはクラス *Pdb* として定義されています。現在これについてのドキュメントはありませんが、ソースを読めば簡単に理解できます。拡張インターフェースはモジュール *bdb* と *cmd* を使っています。

デバッガのプロンプトは (Pdb) です。デバッガに制御された状態でプログラムを実行するための典型的な使い方は以下のようになります:

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
```

(次のページに続く)

(前のページからの続き)

```
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

バージョン 3.3 で変更: *readline* モジュールによるコマンドおよびコマンド引数のタブ補完が利用できます。たとえば、p コマンドの引数では現在のグローバルおよびローカル名が候補として表示されます。

他のスクリプトをデバッグするために、pdb.py をスクリプトとして呼び出すこともできます。例えば:

```
python3 -m pdb myscript.py
```

スクリプトとして pdb を起動すると、デバッグ中のプログラムが異常終了したときに pdb が自動的に事後デバッグモードに入ります。事後デバッグ後 (またはプログラムの正常終了後)、pdb はプログラムを再起動します。自動再起動を行った場合、pdb の状態 (ブレークポイントなど) はそのまま維持されるので、たいていの場合、プログラム終了時にデバッガーも終了させるよりも便利なはずです。

バージョン 3.2 で追加: pdb.py は -c オプションを受け付けるようになりました。このオプションで .pdbrc ファイルが与えられると、ファイル内のコマンドを実行します。**デバッガコマンド** を参照してください。

バージョン 3.7 で追加: pdb.py now accepts a -m option that execute modules similar to the way python3 -m does. As with a script, the debugger will pause execution just before the first line of the module.

実行するプログラムをデバッガで分析する典型的な使い方は:

```
import pdb; pdb.set_trace()
```

これでコードの中のその後に続く文をステップ実行できます。そして *continue* コマンドでデバッガーを停止し処理を続行できます。

バージョン 3.7 で追加: The built-in *breakpoint()*, when called with defaults, can be used instead of import pdb; pdb.set_trace().

クラッシュしたプログラムを調べるための典型的な使い方は以下のようになります:

```
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./mymodule.py", line 4, in test
    test2()
  File "./mymodule.py", line 3, in test2
    print(spam)
NameError: spam
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print(spam)
(Pdb)
```

このモジュールは以下の関数を定義しています。それぞれが少しづつ違った方法でデバッガに入ります:

pdb.**run**(*statement*, *globals=None*, *locals=None*)
デバッガーに制御された状態で (文字列またはコードオブジェクトとして与えられた) *statement* を実行します。あらゆるコードが実行される前にデバッガープロンプトが現れます。ブレークポイントを設定し、*continue* とタイプできます。あるいは、文を *step* や *next* を使って一つづつ実行することができます (これらのコマンドはすべて下で説明します)。オプションの *globals* と *locals* 引数はコードを実行する環境を指定します。デフォルトでは、モジュール *__main__* の辞書が使われます。(組み込み関数 *exec()* または *eval()* の説明を参照してください。)

pdb.**runeval**(*expression*, *globals=None*, *locals=None*)
デバッガーに制御された状態で (文字列またはコードオブジェクトとして与えられる) *expression* を評価します。*runeval()* から復帰するとき、式の値を返します。その他の点では、この関数は *run()* と同様です。

pdb.**runcall**(*function*, **args*, ***kwds*)
function (関数またはメソッドオブジェクト、文字列ではありません) を与えられた引数とともに呼び出します。*runcall()* から復帰するとき、関数呼び出しが返したものはなんでも返します。関数に入るとすぐにデバッガプロンプトが現れます。

pdb.**set_trace**(***, *header=None*)
Enter the debugger at the calling stack frame. This is useful to hard-code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails). If given, *header* is printed to the console just before debugging begins.

バージョン 3.7 で変更: *header* キーワード専用引数。

pdb.**post_mortem**(*traceback=None*)
与えられた *traceback* オブジェクトの事後解析デバッギングに入ります。もし *traceback* が与えられなければ、その時点で取り扱っている例外のうちのひとつを使います。(デフォルト動作をさせるには、例外を取り扱っている最中である必要があります。)

pdb.**pm**()
sys.last_traceback のトレースバックの事後解析デバッギングに入ります。

run* 関数と *set_trace()* は、*Pdb* クラスをインスタンス化して同名のメソッドを実行することのエイリアス関数です。それ以上の機能を利用したい場合は、インスタンス化を自分で行わなければなりません:

class pdb.**Pdb**(*completekey='tab'*, *stdin=None*, *stdout=None*, *skip=None*, *nosigint=False*, *readrc=True*)
Pdb はデバッガクラスです。

completekey, *stdin*, *stdout* 引数は、基底にある *cmd.Cmd* クラスに渡されます。そちらの解説を参照してください。

skip 引数が指定された場合、glob スタイルのモジュール名パターンの iterable (イテレート可能オブジェクト) でなければなりません。デバッガはこのパターンのどれかにマッチするモジュールに属するフレームにはステップインしません。[*1]

[*1] フレームが属するモジュールは、そのフレームのグローバルの `__name__` によって決定されます。

デフォルトでは、Pdb は continue コマンドが投入されると、(ユーザーがコンソールから Ctrl-C を押したときに送られる) SIGINT シグナル用ハンドラーを設定します。これにより Ctrl-C を押すことで再度デバッガーを起動することができます。Pdb に SIGINT ハンドラーを変更させたくない場合は *nosigint* を true に設定してください。

readrc 引数はデフォルトでは真で、Pdb が .pdbrc ファイルをファイルシステムから読み込むかどうかを制御します。

skip を使ってトレースする呼び出しの例:

```
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

引数無しで **監査イベント** pdb.Pdb を送出します。

バージョン 3.1 で追加: *skip* 引数が追加されました。

バージョン 3.2 で追加: *nosigint* 引数が追加されました。以前は Pdb が SIGINT ハンドラーを設定することはありませんでした。

バージョン 3.6 で変更: *readrc* 引数。

run(*statement*, *globals=None*, *locals=None*)
runeval(*expression*, *globals=None*, *locals=None*)
runcall(*function*, **args*, ***kwds*)
set_trace()
前述のこれら関数のドキュメントを参照してください。

27.4.1 デバッガコマンド

デバッガーに認識されるコマンドは以下に一覧されています。たいていのコマンドは以下のように 1、2 文字に省略できます。例えば h(elp) は h か help がで help コマンドを呼び出すことを意味します (ただし he, hel, H, Help, HELP は使用できません)。コマンドの引数はホワイトスペース (スペースかタブ) で区切ってください。コマンド構文として任意の引数は大括弧 (`[]`) で括られています (実際に大括弧はタイプしないでください)。いくつかから選択できる引数は縦線 (`|`) で分割されて記述されています。

空行を入力すると入力された直前のコマンドを繰り返します。例外: 直前のコマンドが *list* コマンドならば、次の 11 行がリストされます。

デバッガが認識しないコマンドは Python 文とみなして、デバッグしているプログラムのコンテキストおいて実行されます。先頭にに感嘆符 (`!`) を付けることで Python 文であると明示することもできます。これはデバッグ中のプログラムを調査する強力な方法です。変数を変更したり関数を呼び出したりすることも可能です。このような文で例外が発生した場合には例外名が出力されますが、デバッガの状態は変化しません。

デバッガーは **エイリアス** をサポートしています。エイリアスはデバッグ中のコンテキストに適用可能な一定レベルのパラメータを保持することができます。

複数のコマンドを `;;` で区切って一行で入力することができます。(一つだけの `;` では使用できません。なぜなら、Python パーサーへ渡される行内の複数のコマンドのための分離記号だからです。) コマンドを分割す

るために何も知的なことはしていません。たとえ引用文字列の途中であっても、入力は最初の ;; で分割されます。

ファイル .pdbrc がユーザーのホームディレクトリかカレントディレクトリにあれば、それが読み込まれ、デバッガーのプロンプトでタイプしたように実行されます。これは特にエイリアスを使用するときに役立ちます。両方にファイルが存在する場合、ホームディレクトリのものが最初に読まれ、次にカレントディレクトリにあるものが読み込まれます。後者により前者の定義は上書きされます。

バージョン 3.2 で変更: .pdbrc に *continue* や *next* のようなデバッグを続行するコマンドが使用できるようになりました。以前はこのようなコマンドは無視されていました。

h(elp) [command]

引数を指定しない場合、利用できるコマンドの一覧が表示されます。引数として *command* が与えられた場合、そのコマンドのヘルプが表示されます。help pdb で完全なドキュメント (*pdb* モジュールの doctring) が表示されます。*command* 引数は識別子でなければならないため、! コマンドのヘルプを表示するには help exec と入力します。

w(here)

スタックの底にある最も新しいフレームと一緒にスタックトレースをプリントします。矢印はカレントフレームを指し、それがほとんどのコマンドのコンテキストを決定します。

d(own) [count]

スタックフレーム内で現在のフレームを *count* レベル (デフォルトは 1) 新しいフレーム方向に移動します。

u(p) [count]

スタックフレーム内で現在のフレームを *count* レベル (デフォルトは 1) 古いフレーム方向に移動します。

b(reak) [([filename:]lineno | function) [, condition]]

lineno 引数を指定した場合、現在のファイルのその行番号の場所にブレークポイントを設定します。*function* 引数を指定した場合、その関数の中の最初の実行可能文にブレークポイントを設定します。別のファイル (まだロードされていないかもしれないもの) のブレークポイントを指定するには、行番号の前にファイル名とコロンを付けます。ファイルは *sys.path* にそって検索されます。各ブレークポイントには番号が割り当てられ、その番号を他のすべてのブレークポイントコマンドで参照されることに注意してください。

第二引数を指定する場合、その値は式で、その評価値が真でなければブレークポイントは有効になりません。

引数なしの場合は、それぞれのブレークポイントに対して、そのブレークポイントに行き当たった回数、現在の通過カウント (ignore count) と、もしあれば関連条件を含めてすべてのブレークポイントをリストします。

tbreak [([filename:]lineno | function) [, condition]]

一時的なブレークポイントで、最初にそこに達したときに自動的に取り除かれます。引数は *break* と同じです。

cl(ear) [filename:lineno | bpnumber [bpnumber ...]]

filename:lineno 引数を与えると、その行にある全てのブレークポイントを解除します。スペースで区切られたブレークポイントナンバーのリストを与えると、それらのブレークポイントを解除します。引数なしの場合は、すべてのブレークポイントを解除します (が、はじめに確認します) 。

`disable [bpnumber [bpnumber ...]]`
ブレークポイント番号 *bpnumber* のブレークポイントを無効にします。ブレークポイントを無効にすると、プログラムの実行を止めることができなくなりますが、ブレークポイントの解除と違いブレークポイントのリストに残っており、(再び) 有効にできます。

`enable [bpnumber [bpnumber ...]]`
指定したブレークポイントを有効にします。

`ignore bpnumber [count]`
与えられたブレークポイントナンバーに通過カウントを設定します。count が省略されると、通過カウントは 0 に設定されます。通過カウントがゼロになったとき、ブレークポイントが機能する状態になります。ゼロでないときは、そのブレークポイントが無効にされず、どんな関連条件も真に評価されていて、ブレークポイントに来るたびに count が減らされます。

`condition bpnumber [condition]`
ブレークポイントに新しい *condition* を設定します。*condition* はブレークポイントを制御する条件式で、この式が真を返す場合のみブレークポイントが有効になります。*condition* を指定しないと既存の条件が除去されます; ブレークポイントは常に有効になります。

`commands [bpnumber]`
ブレークポイントナンバー *bpnumber* にコマンドのリストを指定します。コマンドそのものはその後の行に続けます。`end` だけからなる行を入力することでコマンド群の終わりを示します。例を挙げます:

```
(Pdb) commands 1
(com) p some_variable
(com) end
(Pdb)
```

ブレークポイントからコマンドを取り除くには、`commands` のあとに `end` だけを続けます。つまり、コマンドを一つも指定しないようにします。

bpnumber 引数を指定しない場合、`commands` は最後にセットしたブレークポイントを参照します。

ブレークポイントコマンドはプログラムを走らせ直すのに使えます。単に *continue* コマンドや **:pdb:'step'**、その他実行を再開するコマンドを使えば良いのです。

実行を再開するコマンド (現在のところ **:pdb:'continue'**, **:pdb:'step'**, **:pdb:'next'**, **:pdb:'return'**, **:pdb:'jump'**, **:pdb:'quit'** とそれらの省略形) によって、コマンドリストは終了するものと見なされます (コマンドにすぐ end が続いているかのように)。というのも実行を再開すれば (それが単純な next や step であっても) 別のブレークポイントに到達するかもしれないからです。そのブレークポイントにさらにコマンドリストがあれば、どちらのリストを実行すべきか状況が曖昧になります。

コマンドリストの中で 'silent' コマンドを使うと、ブレークポイントで停止したという通常のメッセージはプリントされません。この振る舞いは特定のメッセージを出して実行を続けるようなブレークポイ

ントでは望ましいものでしょう。他のコマンドが何も画面出力をしなければ、そのブレークポイントに到達したというサインを見ないことになります。

s(tep)
現在の行を実行し、最初に実行可能なものがあらわれたときに (呼び出された関数の中か、現在の関数の次の行で) 停止します。

n(ext)
現在の関数の次の行に達するか、あるいは関数が返るまで実行を継続します。(*next* と *step* の差は *step* が呼び出された関数の内部で停止するのに対し、*next* は呼び出された関数を (ほぼ) 全速力で実行し、現在の関数内の次の行で停止するだけです。)

unt(il) [lineno]
引数なしだと、現在の行から 1 行先まで実行します。

行数を指定すると、指定行数かそれ以上に達するまで実行します。どちらにしても現在のフレームが返ってきた時点で停止します。

バージョン 3.2 で変更: 明示的に行数指定ができるようになりました。

r(eturn)
現在の関数が返るまで実行を継続します。

c(ont(inue))
ブレークポイントに出会うまで、実行を継続します。

j(ump) lineno
次に実行する行を指定します。最も底のフレーム中でのみ実行可能です。前に戻って実行したり、不要な部分をスキップして先の処理を実行する場合に使用します。

ジャンプには制限があり、例えば `for` ループの中には飛び込めませんし、`finally` 節の外にも飛ぶ事ができません。

l(ist) [first[, last]]
現在のファイルのソースコードを表示します。引数を指定しないと、現在の行の前後 11 行分を表示するか、直前の表示を続行します。引数に `.` を指定すると、現在の行の前後 11 行分を表示します。数値を 1 個指定すると、その行番号の前後 11 行分を表示します。数値を 2 個指定すると、開始行と最終行として表示します; 2 個めの引数が 1 個め未満だった場合、1 個目を開始行、2 個めを開始行からの行数とみなします。

現在のフレーム内の現在の行は `->` で表示されます。例外をデバッグ中の場合、例外が発生または伝搬した行は、それが現在の行とは異なるとき `>>` で表示されます。

バージョン 3.2 で追加: `>>` マーカー。

ll | longlist
現在の関数またはフレームの全ソースコードを表示します。注目する行は *list* と同じようにマーカーがつきます。

バージョン 3.2 で追加.

`a(rgs)`
現在の関数の引数リストをプリントします。

`p expression`
現在のコンテキストにおいて *expression* を評価し、その値をプリントします。

注釈: `print()` も使えますが、これはデバッガーコマンドではありません --- これは Python の関数 *print()* が実行されます。

`pp expression`
p コマンドに似ていますが、式の値以外は *pprint* モジュールを使用して "pretty-print" されます。

`whatis expression`
式の型を表示します。

`source expression`
与えられたオブジェクトのソースコードの取得を試み、可能であれば表示します。

バージョン 3.2 で追加.

`display [expression]`
式の値が変更されていれば表示します。毎回実行は現在のフレームで停止します。

式を指定しない場合、現在のフレームのすべての式を表示します。

バージョン 3.2 で追加.

`undisplay [expression]`
現在のフレーム内で式をこれ以上表示しないようにします。式を指定しない場合、現在のフレームで display 指定されている式を全てクリアします。

バージョン 3.2 で追加.

`interact`
対話型インタープリターを (*code* モジュールを使って) 開始します。グローバル名前空間には現在のスコープで見つかったすべてのグローバルおよびローカル名が含まれます。

バージョン 3.2 で追加.

`alias [name [command]]`
name という名前の *command* を実行するエイリアスを作成します。コマンドは引用符で囲まれていては **いけません**。入れ替え可能なパラメータは %1、%2 などで指し示され、さらに %* は全パラメータに置き換えられます。コマンドが与えられなければ、*name* に対する現在のエイリアスを表示します。引数が与えられなければ、すべてのエイリアスがリストされます。

エイリアスは入れ子になってもよく、pdb プロンプトで合法的にタイプできるどんなものでも含めることができます。内部 pdb コマンドをエイリアスによって上書きすることが **できます**。そのとき、このようなコマンドはエイリアスが取り除かれるまで隠されます。エイリアス化はコマンド行の最初の語へ再帰的に適用されます。行の他のすべての語はそのままです。

例として、二つの便利なエイリアスがあります (特に .pdbrc ファイルに置かれたときに):

```
# Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print("%1.",k,"=",%1.__dict__[k])
# Print instance variables in self
alias ps pi self
```

unalias name
指定したエイリアスを削除します。

! statement
現在のスタックフレームのコンテキストにおいて (一行の) *statement* を実行します。文の最初の語がデバッガーコマンドと共通でない場合は、感嘆符を省略することができます。グローバル変数を設定するために、同じ行に **global** 文とともに代入コマンドの前に付けることができます。:

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

run [args ...]
restart [args ...]
デバッグ中のプログラムを再実行します。引数が与えられると、*shlex* で分割され、結果が新しい *sys.argv* として使われます。ヒストリー、ブレークポイント、アクション、そして、デバッガーオプションは引き継がれます。*restart* は *run* の別名です。

q(uit)
デバッガーを終了します。実行しているプログラムは中断されます。

debug code
Enter a recursive debugger that steps through the code argument (which is an arbitrary expression or statement to be executed in the current environment).

retval
Print the return value for the last return of a function.

脚注

27.5 Python プロファイラ

ソースコード: Lib/profile.py と Lib/pstats.py

27.5.1 プロファイラとは

cProfile と *profile* は **決定論的プロファイリング** *(deterministic profiling)* を行います。**プロファイル** *(profile)* とは、プログラムの各部分がどれだけ頻繁に呼ばれたか、そして実行にどれだけ時間がかかったかという統計情報です。*pstats* モジュールを使ってこの統計情報をフォーマットし表示することができます。

Python 標準ライブラリは同じインターフェイスを提供するプロファイラの実装を 2 つ提供しています:

1. *cProfile* はほとんどのユーザーに推奨されるモジュールです。C 言語で書かれた拡張モジュールで、オーバーヘッドが少ないため長時間実行されるプログラムのプロファイルに適しています。Brett Rosen と Ted Czotter によって提供された `lsprof` に基づいています。

2. *profile* はピュア Python モジュールで、*cProfile* モジュールはこのモジュールのインタフェースを真似ています。対象プログラムに相当のオーバーヘッドが生じます。もしプロファイラに何らかの拡張をしたいのであれば、こちらのモジュールを拡張する方が簡単でしょう。このモジュールはもともと Jim Roskind により設計、実装されました。

注釈: 2 つのプロファイラモジュールは、与えられたプログラムの実行時プロファイルを取得するために設計されており、ベンチマークのためのものではありません (ベンチマーク用途には *timeit* のほうが正確な計測結果を求められます)。これは Python コードと C で書かれたコードをベンチマークするときに特に大きく影響します。プロファイラは Python コードに対してオーバーヘッドを発生しますが、C 言語レベルの関数に対してはオーバーヘッドを生じません。なので C 言語で書かれたコードが、実際の速度と関係なく、Python で書かれたコードより速く見えるでしょう。

27.5.2 かんたんユーザマニュアル

この節は ” マニュアルなんか読みたくない人” のために書かれています。ここではきわめて簡単な概要説明とアプリケーションのプロファイリングを手っ取り早く行う方法だけを解説します。

1 つの引数を取る関数をプロファイルしたい場合、次のようにできます:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

(お使いのシステムで *cProfile* が使えないときは代わりに *profile* を使って下さい)

上のコードは *re.compile()* を実行して、プロファイル結果を次のように表示します:

```
         197 function calls (192 primitive calls) in 0.002 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.001    0.001 <string>:1(<module>)
        1    0.000    0.000    0.001    0.001 re.py:212(compile)
```

(次のページに続く)

(前のページからの続き)

```
    1    0.000    0.000    0.001    0.001 re.py:268(_compile)
    1    0.000    0.000    0.000    0.000 sre_compile.py:172(_compile_charset)
    1    0.000    0.000    0.000    0.000 sre_compile.py:201(_optimize_charset)
    4    0.000    0.000    0.000    0.000 sre_compile.py:25(_identityfunction)
  3/1    0.000    0.000    0.000    0.000 sre_compile.py:33(_compile)
```

最初の行は 197 回の呼び出しを測定したことを示しています。その中で 192 回は **プリミティブ** です。すなわち呼び出しが再帰によってなされてはいません。次の行は `Ordered by: standard name` は一番右の列の文字列が出力のソートに用いられたことを示します。列の見出しは以下を含みます:

ncalls　呼び出し回数

tottime　与えられた関数に消費された合計時間 (sub-function の呼び出しで消費された時間は除外されています)

percall　`tottime` を `ncalls` で割った値

cumtime　この関数と全ての subfunction に消費された累積時間 (起動から終了まで)。この数字は再帰関数に **ついても** 正確です。

percall　`cumtime` をプリミティブな呼び出し回数で割った値

filename:lineno(function)　その関数のファイル名、行番号、関数名

最初の行に 2 つの数字がある場合 (たとえば `3/1`) は、関数が再帰的に呼び出されたということです。2 つ目の数字はプリミティブな呼び出しの回数で、1 つ目の数字は総呼び出し回数です。関数が再帰的に呼び出されなかった場合、それらは同じ値で数字は 1 つしか表示されないことに留意してください。

`run()` 関数でファイル名を指定することで、プロファイル実行の終了時に出力を表示する代わりに、ファイルに保存することが出来ます。

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

pstats.Stats クラスはファイルからプロファイルの結果を読み込んで様々な書式に整えます。

ファイル *cProfile* と *profile* は他のスクリプトをプロファイルするためのスクリプトとして起動することも出来ます。例えば:

```
python -m cProfile [-o output_file] [-s sort_order] (-m module | myscript.py)
```

`-o` はプロファイルの結果を標準出力の代わりにファイルに書き出します。

`-s` は出力を *sort_stats()* で出力をソートする値を指定します。`-o` がない場合に有効です。

`-m` specifies that a module is being profiled instead of a script.

バージョン 3.7 で追加: Added the `-m` option to *cProfile*.

バージョン 3.8 で追加: Added the `-m` option to *profile*.

pstats モジュールの *Stats* クラスにはプロファイル結果のファイルに保存されているデータを処理して出力するための様々なメソッドがあります。

```
import pstats
from pstats import SortKey
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

strip_dirs() メソッドによって全モジュール名から無関係なパスが取り除かれました。*sort_stats()* メソッドにより、出力される標準的な モジュール/行/名前 文字列にしたがって全項目がソートされました。*print_stats()* メソッドによって全統計が出力されました。以下のようなソート呼び出しを試すことができます:

```
p.sort_stats(SortKey.NAME)
p.print_stats()
```

最初の行ではリストを関数名でソートしています。2 行目で情報を出力しています。さらに次の内容も試してください:

```
p.sort_stats(SortKey.CUMULATIVE).print_stats(10)
```

このようにすると、関数が消費した累計時間でソートして、さらにその上位 10 件だけを表示します。どのアルゴリズムが時間を多く消費しているのか知りたいときは、この方法が役に立つはずです。

ループで多くの時間を消費している関数はどれか調べたいときは、次のようにします:

```
p.sort_stats(SortKey.TIME).print_stats(10)
```

上記はそれぞれの関数で消費された時間でソートして、上位 10 件の関数の情報が表示されます。

次の内容も試してください:

```
p.sort_stats(SortKey.FILENAME).print_stats('__init__')
```

このようにするとファイル名でソートされ、そのうちクラスの初期化メソッド (メソッド名 `__init__`) に関する統計情報だけが表示されます:

```
p.sort_stats(SortKey.TIME, SortKey.CUMULATIVE).print_stats(.5, 'init')
```

上記は時間 (time) をプライマリキー、累計時間 (cumulative time) をセカンダリキーにしてソートした後でさらに条件を絞って統計情報を出力します。`.5` は上位 50% だけを選択することを意味し、さらにその中から文字列 `init` を含むものだけが表示されます。

どの関数がどの関数を呼び出しているのかを知りたければ、次のようにします (`p` は最後に実行したときの状態でソートされています):

```
p.print_callers(.5, 'init')
```

このようにすると、関数ごとの呼び出し側関数の一覧が得られます。

さらに詳しい機能を知りたければマニュアルを読むか、次の関数の実行結果から内容を推察してください:

```
p.print_callees()
p.add('restats')
```

スクリプトとして起動した場合、*pstats* モジュールはプロファイルのダンプを読み込み、分析するための統計ブラウザとして動きます。シンプルな行指向のインタフェース (*cmd* を使って実装) とヘルプ機能を備えています。

27.5.3 リファレンスマニュアル -- profile と cProfile

profile および *cProfile* モジュールは以下の関数を提供します:

profile.run(*command*, *filename=None*, *sort=-1*)

この関数は *exec()* 関数に渡せる一つの引数と、オプション引数としてファイル名を指定できます。全ての場合でこのルーチンは以下を実行します:

```
exec(command, __main__.__dict__, __main__.__dict__)
```

そして実行結果からプロファイル情報を収集します。ファイル名が指定されていない場合は、*Stats* のインスタンスが自動的に作られ、簡単なプロファイルレポートが表示されます。*sort* が指定されている場合は、*Stats* インスタンスに渡され、結果をどのように並び替えるかを制御します。

profile.runctx(*command*, *globals*, *locals*, *filename=None*, *sort=-1*)

この関数は *run()* に似ていますが、globals、locals 辞書を *command* のために与えるための追加引数を取ります。このルーチンは以下を実行します:

```
exec(command, globals, locals)
```

そしてプロファイル統計情報を *run()* 同様に収集します。

class profile.Profile(*timer=None*, *timeunit=0.0*, *subcalls=True*, *builtins=True*)

このクラスは普通、プロファイリングを `cProfile.run()` 関数が提供するもの以上に正確に制御したい場合にのみ使われます。

timer 引数に、コードの実行時間を計測するためのカスタムな時刻取得用関数を渡せます。これは現在時刻を表す単一の数値を返す関数でなければなりません。もしこれが整数を返す関数ならば、*timeunit* には単位時間当たりの実際の持続時間を指定します。たとえば関数が 1000 分の 1 秒単位で計測した時間を返すとすると、*timeunit* は `.001` でしょう。

Profile クラスを直接使うと、プロファイルデータをファイルに書き出すことなしに結果をフォーマット出来ます:

```
import cProfile, pstats, io
from pstats import SortKey
pr = cProfile.Profile()
pr.enable()
```

(次のページに続く)

(前のページからの続き)

```
# ... do something ...
pr.disable()
s = io.StringIO()
sortby = SortKey.CUMULATIVE
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print(s.getvalue())
```

The *Profile* class can also be used as a context manager (supported only in *cProfile* module. see **コンテキストマネージャ型**):

```
import cProfile

with cProfile.Profile() as pr:
    # ... do something ...

pr.print_stats()
```

バージョン 3.8 で変更: コンテキストマネージャーサポートが追加されました。

enable()
Start collecting profiling data. Only in *cProfile*.

disable()
Stop collecting profiling data. Only in *cProfile*.

create_stats()
プロファイリングデータの収集を停止し、現在のプロファイルとして結果を内部で記録します。

print_stats(*sort=-1*)
現在のプロファイルに基づいて *Stats* オブジェクトを作成し、結果を標準出力に出力します。

dump_stats(*filename*)
現在のプロファイルの結果を *filename* に書き出します。

run(*cmd*)
exec() を用いて cmd をプロファイルします。

runctx(*cmd*, *globals*, *locals*)
exec() を用いて指定されたグローバルおよびローカルな環境で cmd をプロファイルします。

runcall(*func*, **args*, ***kwargs*)
`func(*args, **kwargs)` をプロファイルします。

Note that profiling will only work if the called command/function actually returns. If the interpreter is terminated (e.g. via a *sys.exit()* call during the called command/function execution) no profiling results will be printed.

27.5.4 Stats クラス

Stats クラスを使用してプロファイラデータを解析します。

class pstats.Stats(**filenames or profile, stream=sys.stdout*)

このコンストラクタは *filename* で指定した (単一または複数の) ファイルもしくは Profile のインスタンスから ” 統計情報オブジェクト” のインスタンスを生成します。出力は *stream* で指定したストリームに出力されます。

上記コンストラクタで指定するファイルは、使用する *Stats* に対応したバージョンの *profile* または *cProfile* で作成されたものでなければなりません。将来のバージョンのプロファイラとの互換性は **保証されておらず** 、他のプロファイラで生成されたファイルや異なるオペレーティングシステムで実行される同じプロファイルとの互換性もないことに注意してください。複数のファイルを指定した場合、同一の関数の統計情報はすべて合算され、複数のプロセスで構成される全体をひとつのレポートで検証することが可能になります。既存の *Stats* オブジェクトに別のファイルの情報を追加するときは、*add()* メソッドを使用します。

プロファイルデータをファイルから読み込む代わりに、cProfile.Profile または *profile.Profile* オブジェクトをプロファイルデータのソースとして使うことができます。

Stats には次のメソッドがあります:

strip_dirs()

Stats クラスのこのメソッドは、ファイル名の前に付いているすべてのパス情報を取り除くためのものです。出力の幅を 80 文字以内に収めたいときに重宝します。このメソッドはオブジェクトを変更するため、取り除いたパス情報は失われます。パス情報除去の操作後、オブジェクトが保持するデータエントリは、オブジェクトの初期化、ロード直後と同じように ” ランダムに” 並んでいます。*strip_dirs()* を実行した結果、2 つの関数名が区別できない (両者が同じファイルの同じ行番号で同じ関数名となった) 場合、一つのエントリに合算されされます。

add(**filenames*)

Stats クラスのこのメソッドは、既存のプロファイリングオブジェクトに情報を追加します。引数は対応するバージョンの *profile.run()* または cProfile.run() によって生成されたファイルの名前でなくてはなりません。関数の名前が区別できない (ファイル名、行番号、関数名が同じ) 場合、一つの関数の統計情報として合算されます。

dump_stats(*filename*)

Stats オブジェクトに読み込まれたデータを、ファイル名 *filename* のファイルに保存します。ファイルが存在しない場合は新たに作成され、すでに存在する場合には上書きされます。このメソッドは *profile.Profile* クラスおよび cProfile.Profile クラスの同名のメソッドと等価です。

sort_stats(**keys*)

This method modifies the *Stats* object by sorting it according to the supplied criteria. The argument can be either a string or a SortKey enum identifying the basis of a sort (example: 'time', 'name', SortKey.TIME or SortKey.NAME). The SortKey enums argument have advantage over the string argument in that it is more robust and less error prone.

2 つ以上のキーが指定された場合、2 つ目以降のキーは、それ以前のキーで等価となったデータエントリの再ソートに使われます。たとえば `sort_stats(SortKey.NAME, SortKey.FILE)` とした場合、まずすべてのエントリが関数名でソートされた後、同じ関数名で複数のエントリがあればファイル名でソートされます。

For the string argument, abbreviations can be used for any key names, as long as the abbreviation is unambiguous.

The following are the valid string and SortKey:

Valid String Arg	Valid enum Arg	意味
`'calls'`	SortKey.CALLS	呼び出し数
`'cumulative'`	SortKey.CUMULATIVE	累積時間
`'cumtime'`	N/A	累積時間
`'file'`	N/A	ファイル名
`'filename'`	SortKey.FILENAME	ファイル名
`'module'`	N/A	ファイル名
`'ncalls'`	N/A	呼び出し数
`'pcalls'`	SortKey.PCALLS	プリミティブな呼び出し回数
`'line'`	SortKey.LINE	行番号
`'name'`	SortKey.NAME	関数名
`'nfl'`	SortKey.NFL	関数名/ファイル名/行番号
`'stdname'`	SortKey.STDNAME	標準名
`'time'`	SortKey.TIME	内部時間
`'tottime'`	N/A	内部時間

すべての統計情報のソート結果は降順 (最も多く時間を消費したものが一番上に来る) となることに注意してください。ただし、関数名、ファイル名、行数に関しては昇順 (アルファベット順) になります。`SortKey.NFL` と `SortKey.STDNAME` には微妙な違いがあります。標準名 (standard name) とは表示された名前によるソートで、埋め込まれた行番号のソート順が特殊です。たとえば、(ファイル名が同じで) 3、20、40 という行番号のエントリがあった場合、20、3、40 の順に表示されます。一方 `SortKey.NFL` は行番号を数値として比較します。要するに、`sort_stats(SortKey.NFL)` は `sort_stats(SortKey.NAME, SortKey.FILENAME, SortKey.LINE)` と指定した場合と同じになります。

後方互換性のため、数値を引数に使った `-1`, `0`, `1`, `2` の形式もサポートしています。それぞれ `'stdname'`, `'calls'`, `'time'`, `'cumulative'` として処理されます。引数をこの旧スタイルで指定した場合、最初のキー (数値キー) だけが使われ、複数のキーを指定しても 2 番目以降は無視されます。

バージョン 3.7 で追加: Added the SortKey enum.

reverse_order()

Stats クラスのこのメソッドは、オブジェクト内の情報のリストを逆順にソートします。デフォルトでは選択したキーに応じて昇順、降順が適切に選ばれることに注意してください。

print_stats(**restrictions*)

Stats クラスのこのメソッドは、*profile.run()* の項で述べたプロファイルのレポートを出力します。

出力するデータの順序はオブジェクトに対し最後に行った *sort_stats()* による操作に基づきます (*add()* と *strip_dirs()* による制限にも支配されます)。

引数は (もし与えられると) リストを重要なエントリのみに制限するために使われます。初期段階でリストはプロファイルした関数の完全な情報を持っています。制限の指定は、(行数を指定する) 整数、(行のパーセンテージを指定する) 0.0 から 1.0 までの割合を指定する小数、(出力する standard name にマッチする) 正規表現として解釈される文字列のいずれかを使って行います。複数の制限が指定された場合、指定の順に適用されます。たとえば次のようになります:

```
print_stats(.1, 'foo:')
```

上記の場合まず出力するリストは全体の 10% に制限され、さらにファイル名の一部に文字列 `.*foo:` を持つ関数だけが出力されます:

```
print_stats('foo:', .1)
```

こちらの例の場合、リストはまずファイル名に `.*foo:` を持つ関数だけに制限され、その中の最初の 10% だけが出力されます。

print_callers(**restrictions*)

Stats クラスのこのメソッドは、プロファイルのデータベースの中から何らかの関数呼び出しを行った関数をすべて出力します。出力の順序は *print_stats()* によって与えられるものと同じです。出力を制限する引数も同じです。各呼び出し側関数についてそれぞれ一行ずつ表示されます。フォーマットは統計を作り出したプロファイラごとに微妙に異なります:

- *profile* の場合、呼び出し側関数の後に括弧で囲まれて表示される数値はその呼び出しが何回行われたかを示しています。利便性のため、2 番目の括弧なしで表示される数値によって、関数が消費した累積時間を表しています。
- *cProfile* の場合、各呼び出し側関数の後に 3 つの数字が付きます。呼び出しが何回行われたかと、この呼び出し側関数からの呼び出しによって現在の関数内で消費された合計時間および累積時間です。

print_callees(**restrictions*)

Stats クラスのこのメソッドは、指定した関数から呼び出された関数のリストを出力します。呼び出し側、呼び出される側の方向は逆ですが、引数と出力の順序に関しては *print_callers()* と同じです。

27.5.5 決定論的プロファイリングとは

決定論的プロファイリング とは、すべての **関数呼び出し**, **関数からのリターン**, **例外発生** をモニターし、正確なタイミングを記録することで、イベント間の時間、つまりどの時間にユーザコードが実行されているのかを計測するやり方です。もう一方の **統計的プロファイリング** (このモジュールでこの方法は採用していません) とは、有効なインストラクションポインタからランダムにサンプリングを行い、プログラムのどこで時間が使われているかを推定する方法です。後者の方法は、オーバヘッドが少ないものの、プログラムのどこで多くの時間が使われているか、その相対的な示唆に留まります。

Python の場合、実行中は必ずインタプリタが動作しているため、決定論的プロファイリングを行うために計測用のコードを追加する必要はありません。Python は自動的に各イベントに **フック** (オプションのコールバック) を提供します。加えて Python のインタプリタという性質によって、実行時に大きなオーバーヘッドを伴う傾向がありますが、それに比べると一般的なアプリケーションでは決定論的プロファイリングで追加される処理のオーバーヘッドは少ない傾向にあります。結果的に、決定論的プロファイリングは少ないコストで Python プログラムの実行時間に関する詳細な統計を得られる方法となっているのです。

呼び出し回数はコード中のバグ発見にも使用できます (とんでもない数の呼び出しが行われている部分)。インライン拡張の対象とすべき部分を見つけるためにも使えます (呼び出し頻度の高い部分)。内部時間の統計は、注意深く最適化すべき" ホットループ" の発見にも役立ちます。累積時間の統計は、アルゴリズム選択に関連した高レベルのエラー検知に役立ちます。なお、このプロファイラは再帰的なアルゴリズム実装の累計時間を計ることが可能で、通常のループを使った実装と直接比較することもできるようになっています。

27.5.6 制限事項

一つの制限はタイミング情報の正確さに関するものです。決定論的プロファイラには正確さに関する根本的問題があります。最も明白な制限は、(一般に) " クロック" は .001 秒の精度しかないということです。これ以上の精度で計測することはできません。仮に充分な精度が得られたとしても、" 誤差" が計測の平均値に影響を及ぼすことがあります。この最初の誤差を取り除いたとしても、それがまた別の誤差を引き起こす原因となります。

もう一つの問題として、イベントを検知してからプロファイラがその時刻を実際に **取得** するまでに " いくらかの時間がかかる" ことです。同様に、イベントハンドラが終了する時にも、時刻を取得して (そしてその値を保存して) から、ユーザコードが処理を再開するまでの間に遅延が発生します。結果的に多く呼び出される関数または多数の関数から呼び出される関数の情報にはこの種の誤差が蓄積する傾向にあります。このようにして蓄積される誤差は、典型的にはクロックの精度を下回ります (1 クロック以下) が、一方でこの時間が累計して非常に大きな値になることも **あり得ます** 。

この問題はオーバーヘッドの小さい *cProfile* よりも *profile* においてより重要です。そのため、*profile* は誤差が確率的に (平均値で) 減少するようにプラットフォームごとに補正する機能を備えています。プロファイラに補正を施すと (最小二乗の意味で) 正確さが増しますが、ときには数値が負の値になってしまうこともあります (呼び出し回数が極めて少なく、確率の神があなたに意地悪をしたとき :-))。プロファイルの結果に負の値が出力されても **驚かないでください** 。これは補正を行った場合にのみ生じることで、補正を行わない場合に比べて計測結果は実際にはより正確になっているはずだからです。

27.5.7 キャリブレーション (補正)

profile のプロファイラは time 関数呼び出しおよびその値を保存するためのオーバーヘッドを補正するために、各イベントの処理時間から定数を引きます。デフォルトでこの定数の値は 0 です。以下の手順で、プラットフォームに合った、より適切な定数が得られます (**制限事項** 参照)。

```
import profile
pr = profile.Profile()
for i in range(5):
    print(pr.calibrate(10000))
```

calibrate メソッドは引数として与えられた数だけ Python の呼び出しを行います。直接呼び出す場合と、プロファイラを使って呼び出す場合の両方が実施され、それぞれの時間が計測されます。その結果、プロファイラのイベントに隠されたオーバーヘッドが計算され、その値は浮動小数として返されます。たとえば、1.8 GHz の Intel Core i5 で Mac OS X を使用、Python の time.process_time() をタイマとして使った場合、値はおよそ 4.04e-6 となります。

この手順で使用しているオブジェクトはほぼ一定の結果を返します。**非常に** 早いコンピュータを使う場合、もしくはタイマの性能が貧弱な場合は、一定の結果を得るために引数に 100000 や 1000000 といった大きな値を指定する必要があるかもしれません。

一定の結果が得られたら、それを使う方法には 3 通りあります:

```
import profile

# 1. Apply computed bias to all Profile instances created hereafter.
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

選択肢がある場合は、補正値は小さめに設定した方が良いでしょう。プロファイルの結果に負の値が表われる” 頻度が低く” なるはずです。

27.5.8 カスタムな時刻取得用関数を使う

プロファイラが時刻を取得する方法を変更したいなら (たとえば、実測時間やプロセスの経過時間を使いたい場合)、時刻取得用の関数を `Profile` クラスのコンストラクタに指定することができます:

```
pr = profile.Profile(your_time_func)
```

この結果生成されるプロファイラは時刻取得に `your_time_func()` を呼び出すようになります。*profile.Profile* と `cProfile.Profile` のどちらを使っているかにより、`your_time_func` の戻り値は異なって解釈されます:

profile.Profile your_time_func() は単一の数値、あるいは (*os.times()* と同じように) その合計が累計時間を示すリストを返すようになっていなければなりません。関数が 1 つの数値、あるいは長さ 2 の数値のリストを返すようになっていれば、ディスパッチルーチンには特別な高速化バージョンが使われます。

あなたが選択した時刻取得関数のために、プロファイラのクラスを補正すべきであることを警告しておきます (**キャリブレーション** *(補正)* 参照)。ほとんどのマシンでは、プロファイル時のオーバヘッドの小ささという意味においては、時刻取得関数が長整数値を返すのが最も良い結果を生みます。(*os.times()* は浮動小数点数のタプルを返すので **かなり悪い** です。) 綺麗な手段でより良い時刻取得関数に置き換えたいと望むなら、クラスを派生して、ディスパッチメソッドを置き換えてあなたの関数を一番いいように処理するように固定化してしまってください。補正定数の調整もお忘れなく。

cProfile.Profile your_time_func() は単一の数値を返すようになっていなければなりません。単一の整数を返すのであれば、クラスコンストラクタの第二引数に単位時間当たりの実際の持続時間を渡せます。例えば your_integer_time_func が 1000 分の 1 秒単位で計測した時間を返すとすると、Profile インスタンスを以下のように構築出来ます:

```
pr = cProfile.Profile(your_integer_time_func, 0.001)
```

cProfile.Profile クラスはキャリブレーションができないので、自前のタイマ関数は注意を払って使う必要があり、またそれは可能な限り速くなければなりません。自前のタイマ関数で最高の結果を得るには、_lsprof 内部モジュールの C ソースファイルにハードコードする必要があるかもしれません。

Python 3.3 で *time* に、プロセス時間や実時間の精密な計測に使える新しい関数が追加されました。例えば、*time.perf_counter()* を参照してください。

27.6 timeit --- 小さなコード断片の実行時間計測

ソースコード: Lib/timeit.py

このモジュールは小さい Python コードをの時間を計測するシンプルな手段を提供しています。**コマンドラインインターフェイス** の他 **呼び出しも可能** です。このモジュールは実行時間を計測するときに共通するいくつかの罠を回避します。O'Reilly 出版の *Python Cookbook* にある、Tim Peter による "Algorithms" 章も参照してください。

27.6.1 基本的な例

次の例は **コマンドラインインターフェイス** を使って 3 つの異なる式の時間を測定する方法を示しています。

```
$ python3 -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 5: 30.2 usec per loop
$ python3 -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 5: 27.5 usec per loop
$ python3 -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 5: 23.2 usec per loop
```

同じ事を *Python* インターフェイス を使って実現することもできます:

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.3018611848820001
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.2727368790656328
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.23702679807320237
```

呼び出し可能オブジェクトは *Python* インターフェイス から渡すこともできます:

```
>>> timeit.timeit(lambda: "-".join(map(str, range(100))), number=10000)
0.19665591977536678
```

ただし、*timeit()* はコマンドラインインターフェイスを使った時だけ繰り返し回数を自動で決定する事に注意してください。使用例 節でより高度な例を説明しています。

27.6.2 Python インターフェイス

このモジュールは 3 つの有用な関数と 1 つの公開クラスを持っています:

timeit.timeit(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *number=1000000*, *globals=None*)

与えられた文、*setup* コードおよび *timer* 関数で *Timer* インスタンスを作成し、その *timeit()* メソッドを *number* 回実行します。任意の *globals* 引数はコードを実行する名前空間を指定します。

バージョン 3.5 で変更: 任意の *globals* 引数が追加されました。

timeit.repeat(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *repeat=5*, *number=1000000*, *globals=None*)

与えられた文、*setup* コードおよび *timer* 関数で *Timer* インスタンスを作成し、その *repeat()* メソッドを *repeat* 回繰り返すのを *number* 回実行します。任意の *globals* 引数はコードを実行する名前空間を指定します。

バージョン 3.5 で変更: 任意の *globals* 引数が追加されました。

バージョン 3.7 で変更: *repeat* のデフォルト値が 3 から 5 へ変更されました。

timeit.default_timer()

デフォルトのタイマーです。常に *time.perf_counter()* です。

バージョン 3.3 で変更: デフォルトのタイマーが *time.perf_counter()* になりました。

class timeit.Timer(*stmt='pass'*, *setup='pass'*, *timer=<timer function>*, *globals=None*)

小さなコード片の実行時間を計測するためのクラスです。

コンストラクターは計測する文、セットアップのための追加の文、ならびにタイマー関数を取ります。両文のデフォルトは `'pass'` です。タイマー関数はプラットフォーム依存です (モジュールの doctring を参照)。*stmt* および *setup* には、複数行の文字列リテラルを含まない限り、; や改行で区切られた複

数の文でも構いません。デフォルトでは文は timeit の名前空間内で実行されます。この挙動は名前空間を *globals* に渡すことで制御出来ます。

最初の命令文の実行時間を計測するには、*timeit()* メソッドを使用します。*repeat()* と:meth:*autorange*‘**メソッドは** *:meth:*.timeit‘ を複数回呼び出したい時に使用します。

setup の実行時間は全実行時間から除外されています。

stmt および *setup* パラメータは、引数なしの呼び出し可能オブジェクトをとることもできます。呼び出し可能オブジェクトを指定すると、そのオブジェクトの呼び出しがタイマー関数に埋め込まれ、その関数が *timeit()* メソッドによって実行されます。この場合、関数呼び出しが増えるためにタイミングのオーバーヘッドが少し増える点に注意してください。

バージョン 3.5 で変更: 任意の *globals* 引数が追加されました。

timeit(*number=1000000*)

メイン文を *number* 回実行した時間を計測します。このメソッドはセットアップ文を 1 回だけ実行し、メイン文を指定回数実行するのにかかった秒数を浮動小数で返します。引数はループを何回実行するかの指定で、デフォルト値は 100 万回です。メイン文、セットアップ文、タイマー関数はコンストラクターで指定されたものを使用します。

注釈: デフォルトでは、*timeit()* は計測中、一時的に **ガベージコレクション** を停止します。この手法の利点は個々の計測結果がより比較しやすくなることです。欠点は、ガベージコレクションが計測される関数の性能の重要な要素である場合があることです。その場合、*setup* 文字列の最初の文でガベージコレクションを有効にできます。以下に例を示します:

```
timeit.Timer('for i in range(10): oct(i)', 'gc.enable()').timeit()
```

---

autorange(*callback=None*)

timeit() を呼び出す回数を自動的に決定します。

これは総時間が 0.2 秒以上になるように *timeit()* を繰り返し呼び出す便利な関数で、最終的な結果（ループ回数、ループ回数に要した時間）を返します。要した時間が少なくとも 0.2 秒になるまで、シーケンス 1, 2, 5, 10, 20, 50, ... から増加する回数で *timeit()* を呼び出します。

callback が与えられ、None でない場合は、callback(number, time_taken) という 2 つの引数を指定して試行された後に呼び出されます。

バージョン 3.6 で追加.

repeat(*repeat=5*, *number=1000000*)

timeit() を複数回繰り返します。

これは *timeit()* を繰り返し呼び出したい時に有用で、結果をリストにして返します。最初の引数で何回 *timeit()* を呼ぶか指定します。第 2 引数で *timeit()* の引数 *number* を指定します。

注釈: 結果のベクトルから平均値や標準偏差を計算して出力させたいと思うかもしれませんが、

それはあまり意味がありません。多くの場合、最も低い値がそのマシンが与えられたコード断片を実行する場合の下限値です。結果のうち高めの値は、Python のスピードが一定しないために生じたものではなく、その他の計測精度に影響を及ぼすプロセスによるものです。したがって、結果のうち *min()* だけが見るべき値となるでしょう。この点を押さえた上で、統計的な分析よりも常識的な判断で結果を見るようにしてください。

バージョン 3.7 で変更: *repeat* のデフォルト値が 3 から 5 へ変更されました。

print_exc(*file=None*)

計測対象コードのトレースバックを出力するためのヘルパーです。

利用例:

```
t = Timer(...)       # outside the try/except
try:
    t.timeit(...)    # or t.repeat(...)
except Exception:
    t.print_exc()
```

これの標準のトレースバックより優れた点は、コンパイルしたテンプレートのソース行が表示されることです。オプションの引数 *file* にはトレースバックの出力先を指定します。デフォルトは *sys.stderr* になります。

27.6.3 コマンドラインインターフェイス

コマンドラインからプログラムとして呼び出す場合は、次の書式を使います:

```
python -m timeit [-n N] [-r N] [-u U] [-s S] [-h] [statement ...]
```

以下のオプションが使用できます:

-n N, --number=N

'statement' を実行する回数

-r N, --repeat=N

timer を繰り返す回数（デフォルトは 5）

-s S, --setup=S

最初に 1 回だけ実行する文 (デフォルトは pass)

-p, --process

デフォルトの *time.perf_counter()* の代わり *time.process_time()* を利用して、実時間ではなくプロセス時間を計測します

バージョン 3.3 で追加.

-u, --unit=U

timer 出力の時間の単位を指定します。nsec、usec、msec、sec を選択できます。

バージョン 3.5 で追加.

-v, --verbose
時間計測の結果をそのまま詳細な数値でくり返し表示する

-h, --help
簡単な使い方を表示して終了する

文は複数行指定することもできます。その場合、各行は独立した文として引数に指定されたものとして処理します。クォートと行頭のスペースを使って、インデントした文を使うことも可能です。この複数行のオプションは *-s* においても同じ形式で指定可能です。

オプション *-n* でループの回数が指定されていない場合、1, 2, 5, 10, 20, 50, ... という数列で、少なくとも総時間が 0.2 秒になるまで回数を増やしていって、適切なループ回数が計算されます。

default_timer() の測定値は、同じマシン上で実行されている他のプログラムの影響を受ける可能性があるため、正確な時間計測が必要な場合は、計測を数回繰り返し、最適な時間を使用することをおすすめします。*-r* オプションはこれに適しています。ほとんどの場合、デフォルトの 5 回の繰り返しで十分でしょう。*time.process_time()* を使用すれば CPU 時間を測定できます。

注釈: pass 文の実行には基本的なオーバーヘッドが存在します。ここにあるコードはこの事実を隠そうとはしていませんが、注意する必要があります。基本的なオーバーヘッドは引数なしでプログラムを起動することにより計測でき、それは Python のバージョンによって異なるでしょう。

27.6.4 使用例

最初に 1 回だけ実行されるセットアップ文を指定することが可能です:

```
$ python -m timeit -s 'text = "sample string"; char = "g"'  'char in text'
5000000 loops, best of 5: 0.0877 usec per loop
$ python -m timeit -s 'text = "sample string"; char = "g"'  'text.find(char)'
1000000 loops, best of 5: 0.342 usec per loop
```

```
>>> import timeit
>>> timeit.timeit('char in text', setup='text = "sample string"; char = "g"')
0.41440500499993504
>>> timeit.timeit('text.find(char)', setup='text = "sample string"; char = "g"')
1.7246671520006203
```

同じことは *Timer* クラスとそのメソッドを使用して行うこともできます:

```
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample string"; char = "g"')
>>> t.timeit()
0.3955516149999312
```

(次のページに続く)

(前のページからの続き)

```
>>> t.repeat()
[0.40183617287970225, 0.37027556854118704, 0.38344867356679524, 0.3712595970846668, 0.
↪37866875250654886]
```

以下の例は、複数行を含んだ式を計測する方法を示しています。ここでは、オブジェクトの存在する属性と存在しない属性に対してテストするために *hasattr()* と try/except を使用した場合のコストを比較しています:

```
$ python -m timeit 'try:' '  str.__bool__' 'except AttributeError:' '  pass'
20000 loops, best of 5: 15.7 usec per loop
$ python -m timeit 'if hasattr(str, "__bool__"): pass'
50000 loops, best of 5: 4.26 usec per loop

$ python -m timeit 'try:' '  int.__bool__' 'except AttributeError:' '  pass'
200000 loops, best of 5: 1.43 usec per loop
$ python -m timeit 'if hasattr(int, "__bool__"): pass'
100000 loops, best of 5: 2.23 usec per loop
```

```
>>> import timeit
>>> # attribute is missing
>>> s = """\
... try:
...     str.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.9138244460009446
>>> s = "if hasattr(str, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.5829014980008651
>>>
>>> # attribute is present
>>> s = """\
... try:
...     int.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

定義した関数に *timeit* モジュールがアクセスできるようにするために、import 文の入った *setup* パラメータを渡すことができます:

```
def test():
```

(次のページに続く)

(前のページからの続き)

```
    """Stupid test function"""
    L = [i for i in range(100)]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

他のオプションは *globals()* を *globals* 引数に渡すことです。これによりコードはあなたのグローバル名前空間内で実行されます。これはそれぞれでインポートするより 便利です:

```
def f(x):
    return x**2
def g(x):
    return x**4
def h(x):
    return x**8

import timeit
print(timeit.timeit('[func(42) for func in (f,g,h)]', globals=globals()))
```

27.7 trace --- Python 文実行のトレースと追跡

ソースコード: Lib/trace.py

---

trace モジュールにより、プログラム実行のトレース、注釈された文のカバレッジの一覧の作成、呼び出し元/呼び出し先の関係の出力、プログラム実行中に実行された関数の表の作成を行うことができます。これは別のプログラム中やコマンドラインから利用することができます。

参考:

Coverage.py　サードパーティの人気なカバレッジツールで、HTML 出力に加えて分岐カバレッジのような高度な機能も提供しています。

27.7.1 コマンドラインからの使用

trace モジュールはコマンドラインから起動することができます。これは次のように簡単です

```
python -m trace --count -C . somefile.py ...
```

これで、somefile.py の実行中に import された Python モジュールの注釈付きリストがカレントディレクトリに生成されます。

--help

使い方を表示して終了します。

--version
モジュールのバージョンを表示して終了します。

バージョン 3.8 で追加: 実行可能なモジュールを走らせられる --module オプションが追加されました。

主要なオプション

trace を実行するときには、以下のオプションの少なくとも 1 つを指定しなければいけません。*--listfuncs* オプションは、*--trace* オプション、*--count* オプションと相互排他的です。*--listfuncs* が与えられたとき、*--trace* オプション、*--count* オプションの両方とも受け付けず、他の場合も同様です。

-c, --count
プログラム完了時に、それぞれの文が何回実行されたかを示す注釈付きリストのファイルを生成します。下記の *--coverdir*, *--file*, *--no-report* も参照してください。

-t, --trace
実行された通りに行を表示します。

-l, --listfuncs
プログラム実行の際に実行された関数を表示します。

-r, --report
--count と *--file* 引数を使った過去のプログラム実行結果から、注釈付きリストのファイルを生成します。コードを実行するわけではありません。

-T, --trackcalls
プログラム実行によって明らかになった呼び出しの関係を表示します。

修飾的オプション

-f, --file=<file>
複数回にわたるトレース実行についてカウント (count) を累積するファイルに名前をつけます。*--count* オプションと一緒に使って下さい。

-C, --coverdir=<dir>
レポートファイルを保存するディレクトリを指定します。package.module についてのカバレッジレポートは *dir/package/module*.cover に書き込まれます。

-m, --missing
注釈付きリストの生成時に、実行されなかった行に >>>>>> の印を付けます。

-s, --summary
--count または *--report* の利用時に、処理されたファイルそれぞれの簡潔な概要を標準出力 (stdout) に書き出します。

-R, --no-report
注釈付きリストを生成しません。これは *--count* を何度か走らせてから最後に単一の注釈付きリストを生成するような場合に便利です。

-g, --timing
各行の先頭にプログラム開始からの時間を付けます。トレース中にだけ使われます。

フィルターオプション

これらのオプションは複数回指定できます。

--ignore-module=<mod>
指定されたモジュールと（パッケージだった場合は）そのサブモジュールを無視します。引数はカンマ区切りのモジュール名リストです。

--ignore-dir=<dir>
指定されたディレクトリとサブディレクトリ中のモジュールとパッケージを全て無視します。引数は *os.pathsep* で区切られたディレクトリのリストです。

27.7.2 プログラミングインターフェース

class trace.Trace(*count=1*, *trace=1*, *countfuncs=0*, *countcallers=0*, *ignoremods=()*, *ignoredirs=()*, *infile=None*, *outfile=None*, *timing=False*)
文 (statement) や式 (expression) の実行をトレースするオブジェクトを作成します。引数は全てオプションです。*count* は行数を数えます。*trace* は行実行のトレースを行います。*countfuncs* は実行中に呼ばれた関数を列挙します。*countcallers* は呼び出しの関係を追跡します。*ignoremods* は無視するモジュールやパッケージのリストです。*ignoredirs* は無視するパッケージやモジュールを含むディレクトリのリストです。*infile* は保存された集計 (count) 情報を読むファイルの名前です。*outfile* は更新された集計 (count) 情報を書き出すファイルの名前です。*timing* は、タイムスタンプをトレース開始時点からの相対秒数で表示します。

run(*cmd*)
コマンドを実行して、現在のトレースパラメータに基づいてその実行から統計情報を集めます。*cmd* は、*exec()* に渡せるような文字列か code オブジェクトです。

runctx(*cmd*, *globals=None*, *locals=None*)
指定された globals と locals 環境下で、コマンドを実行して、現在のトレースパラメータに基づいてその実行から統計情報を集めます。*cmd* は、*exec()* に渡せるような文字列か code オブジェクトです。定義しない場合、*globals* と *locals* はデフォルトで空の辞書となります。

runfunc(*func*, **args*, ***kwds*)
与えられた引数の *func* を、*Trace* オブジェクトのコントロール下で現在のトレースパラメタのもとに呼び出します。

results()
与えられた *Trace* インスタンスの run, runctx, runfunc の以前の呼び出しについて集計した結果を納めた *CoverageResults* オブジェクトを返します。蓄積されたトレース結果はリセットしません。

class trace.CoverageResults
カバレッジ結果のコンテナで、*Trace.results()* で生成されるものです。ユーザーが直接生成するも

のではありません。

update(*other*)
別の *CoverageResults* オブジェクトのデータを統合します。

write_results(*show_missing=True*, *summary=False*, *coverdir=None*)
カバレッジ結果を書き出します。ヒットしなかった行も出力するには *show_missing* を指定します。モジュールごとの概要を出力に含めるには *summary* を指定します。*coverdir* に指定するのは結果ファイルを出力するディレクトリです。None の場合は各ソースファイルごとの結果がそれぞれのディレクトリに置かれます。

簡単な例でプログラミングインターフェースの使い方を見てみましょう:

```
import sys
import trace

# create a Trace object, telling it what to ignore, and whether to
# do tracing or line-counting or both.
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)

# run the new command using the given tracer
tracer.run('main()')

# make a report, placing output in the current directory
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")
```

27.8 tracemalloc --- メモリ割り当ての追跡

バージョン 3.4 で追加.

ソースコード: Lib/tracemalloc.py

---

tracemalloc モジュールは、Python が割り当てたメモリブロックをトレースするためのデバッグツールです。このモジュールは以下の情報を提供します。

- オブジェクトが割り当てられた場所のトレースバック
- ファイル名ごと、及び行ごとに割り当てられたメモリブロックの以下の統計を取ります：総サイズ、ブロック数、割り当てられたブロックの平均サイズ
- メモリリークを検出するために 2 つのスナップショットの差を計算します。

Python が割り当てたメモリブロックの大半をトレースするには、PYTHONTRACEMALLOC 環境変数を 1 に設定して可能な限り早くモジュールを開始させるか、-X tracemalloc コマンドラインオプションを使用してくだ

さい。実行時に *tracemalloc.start()* を呼んで Python のメモリ割り当てのトレースを開始することが出来ます。

デフォルトでは、割り当てられたメモリブロック 1 つのトレースは最新 1 フレームを保存します。開始時に 25 フレームを保存するには、PYTHONTRACEMALLOC 環境変数を 25 に設定するか、-X tracemalloc=25 コマンドラインオプションを使用してください。

27.8.1 使用例

上位 10 を表示する

最も多くのメモリを割り当てているファイル 10 を表示します:

```
import tracemalloc

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("[ Top 10 ]")
for stat in top_stats[:10]:
    print(stat)
```

Python テストスイートの出力例です:

```
[ Top 10 ]
<frozen importlib._bootstrap>:716: size=4855 KiB, count=39328, average=126 B
<frozen importlib._bootstrap>:284: size=521 KiB, count=3199, average=167 B
/usr/lib/python3.4/collections/__init__.py:368: size=244 KiB, count=2315, average=108 B
/usr/lib/python3.4/unittest/case.py:381: size=185 KiB, count=779, average=243 B
/usr/lib/python3.4/unittest/case.py:402: size=154 KiB, count=378, average=416 B
/usr/lib/python3.4/abc.py:133: size=88.7 KiB, count=347, average=262 B
<frozen importlib._bootstrap>:1446: size=70.4 KiB, count=911, average=79 B
<frozen importlib._bootstrap>:1454: size=52.0 KiB, count=25, average=2131 B
<string>:5: size=49.7 KiB, count=148, average=344 B
/usr/lib/python3.4/sysconfig.py:411: size=48.0 KiB, count=1, average=48.0 KiB
```

Python がモジュールから 4855 KiB のデータ (バイトコードで定数) を読み込んでいることと、*collections* モジュールが *namedtuple* 型をビルドするのに 244 KiB を割り当てていることが分かります。

オプションの詳細については *Snapshot.statistics()* を参照してください。

差を計算する

スナップショットを 2 つ取り、差を表示します:

```
import tracemalloc
tracemalloc.start()
# ... start your application ...

snapshot1 = tracemalloc.take_snapshot()
# ... call the function leaking memory ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
for stat in top_stats[:10]:
    print(stat)
```

Python テストスイートのテストを実行する前後の出力例です:

```
[ Top 10 differences ]
<frozen importlib._bootstrap>:716: size=8173 KiB (+4428 KiB), count=71332 (+39369),␣
↪average=117 B
/usr/lib/python3.4/linecache.py:127: size=940 KiB (+940 KiB), count=8106 (+8106), average=119 B
/usr/lib/python3.4/unittest/case.py:571: size=298 KiB (+298 KiB), count=589 (+589),␣
↪average=519 B
<frozen importlib._bootstrap>:284: size=1005 KiB (+166 KiB), count=7423 (+1526), average=139 B
/usr/lib/python3.4/mimetypes.py:217: size=112 KiB (+112 KiB), count=1334 (+1334), average=86 B
/usr/lib/python3.4/http/server.py:848: size=96.0 KiB (+96.0 KiB), count=1 (+1), average=96.0␣
↪KiB
/usr/lib/python3.4/inspect.py:1465: size=83.5 KiB (+83.5 KiB), count=109 (+109), average=784 B
/usr/lib/python3.4/unittest/mock.py:491: size=77.7 KiB (+77.7 KiB), count=143 (+143),␣
↪average=557 B
/usr/lib/python3.4/urllib/parse.py:476: size=71.8 KiB (+71.8 KiB), count=969 (+969),␣
↪average=76 B
/usr/lib/python3.4/contextlib.py:38: size=67.2 KiB (+67.2 KiB), count=126 (+126), average=546 B
```

Python がモジュールデータ 8173 KiB (バイトコードと定数) を読み込み、前回スナップショットを取ったとき、すなわちテストの前に読み込んだ量より 4428 KiB 多いということが分かります。同様に、*linecache* モジュールはトレースバックの書式化に Python ソースコード 940 KiB をキャッシュし、その全ては前回のスナップショットの後に行われたことが分かります。

システムに空きメモリがほとんどない場合、スナップショットをオフラインで解析するための *Snapshot.dump()* メソッドを使用して、スナップショットをディスクに書き込むことが出来ます。そして *Snapshot.load()* メソッドを使用してスナップショットを再読み込みします。

メモリブロックのトレースバックを取得する

最大のメモリブロックのトレースバックを表示するコードです:

```
import tracemalloc

# Store 25 frames
tracemalloc.start(25)

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')

# pick the biggest memory block
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))
for line in stat.traceback.format():
    print(line)
```

Python テストスイートの出力例です (トレースバックは 25 フレームに制限されています):

```
903 memory blocks: 870.1 KiB
  File "<frozen importlib._bootstrap>", line 716
  File "<frozen importlib._bootstrap>", line 1036
  File "<frozen importlib._bootstrap>", line 934
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/doctest.py", line 101
    import pdb
  File "<frozen importlib._bootstrap>", line 284
  File "<frozen importlib._bootstrap>", line 938
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/test/support/__init__.py", line 1728
    import doctest
  File "/usr/lib/python3.4/test/test_pickletools.py", line 21
    support.run_doctest(pickletools)
  File "/usr/lib/python3.4/test/regrtest.py", line 1276
    test_runner()
  File "/usr/lib/python3.4/test/regrtest.py", line 976
    display_failure=not verbose)
  File "/usr/lib/python3.4/test/regrtest.py", line 761
    match_tests=ns.match_tests)
  File "/usr/lib/python3.4/test/regrtest.py", line 1563
    main()
  File "/usr/lib/python3.4/test/__main__.py", line 3
    regrtest.main_in_temp_cwd()
```

(次のページに続く)

(前のページからの続き)

```
  File "/usr/lib/python3.4/runpy.py", line 73
    exec(code, run_globals)
  File "/usr/lib/python3.4/runpy.py", line 160
    "__main__", fname, loader, pkg_name)
```

We can see that the most memory was allocated in the *importlib* module to load data (bytecode and constants) from modules: `870.1 KiB`. The traceback is where the *importlib* loaded data most recently: on the `import pdb` line of the *doctest* module. The traceback may change if a new module is loaded.

top を整形化する

`<frozen importlib._bootstrap>` および `<unknown>` ファイルを無視して、メモリ割り当て量の上位 10 を整形化して表示するコードです:

```
import linecache
import os
import tracemalloc

def display_top(snapshot, key_type='lineno', limit=10):
    snapshot = snapshot.filter_traces((
        tracemalloc.Filter(False, "<frozen importlib._bootstrap>"),
        tracemalloc.Filter(False, "<unknown>"),
    ))
    top_stats = snapshot.statistics(key_type)

    print("Top %s lines" % limit)
    for index, stat in enumerate(top_stats[:limit], 1):
        frame = stat.traceback[0]
        print("#%s: %s:%s: %.1f KiB"
              % (index, frame.filename, frame.lineno, stat.size / 1024))
        line = linecache.getline(frame.filename, frame.lineno).strip()
        if line:
            print('    %s' % line)

    other = top_stats[limit:]
    if other:
        size = sum(stat.size for stat in other)
        print("%s other: %.1f KiB" % (len(other), size / 1024))
    total = sum(stat.size for stat in top_stats)
    print("Total allocated size: %.1f KiB" % (total / 1024))

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
display_top(snapshot)
```

Python テストスイートの出力例です:

```
Top 10 lines
#1: Lib/base64.py:414: 419.8 KiB
    _b85chars2 = [(a + b) for a in _b85chars for b in _b85chars]
#2: Lib/base64.py:306: 419.8 KiB
    _a85chars2 = [(a + b) for a in _a85chars for b in _a85chars]
#3: collections/__init__.py:368: 293.6 KiB
    exec(class_definition, namespace)
#4: Lib/abc.py:133: 115.2 KiB
    cls = super().__new__(mcls, name, bases, namespace)
#5: unittest/case.py:574: 103.1 KiB
    testMethod()
#6: Lib/linecache.py:127: 95.4 KiB
    lines = fp.readlines()
#7: urllib/parse.py:476: 71.8 KiB
    for a in _hexdig for b in _hexdig}
#8: <string>:5: 62.0 KiB
#9: Lib/_weakrefset.py:37: 60.0 KiB
    self.data = set()
#10: Lib/base64.py:142: 59.8 KiB
    _b32tab2 = [a + b for a in _b32tab for b in _b32tab]
6220 other: 3602.8 KiB
Total allocated size: 5303.1 KiB
```

オプションの詳細については *Snapshot.statistics()* を参照してください。

27.8.2 API

関数

tracemalloc.**clear_traces**()

Python が割り当てたメモリブロックのトレースを消去します。

stop() を参照してください。

tracemalloc.**get_object_traceback**(*obj*)

Python オブジェクト *obj* が割り当てられたトレースバックを取得します。*Traceback* インスタンスか、*tracemalloc* モジュールがメモリ割り当てをトレースしていない場合かオブジェクトの割り当てをトレースしていない場合は、None を返します。

gc.get_referrers() や *sys.getsizeof()* 関数も参照してください。

tracemalloc.**get_traceback_limit**()

トレースのトレースバック内に格納されている最大フレーム数を取得します。

tracemalloc モジュールは上限を取得するためにメモリ割り当てをトレースしていなければなりません。そうでなければ例外が送出されます。

start() 関数で上限を設定します。

tracemalloc.**get_traced_memory**()

tracemalloc モジュールがトレースするメモリブロックの現在のサイズと最大時のサイズをタプルとして取得します: (current: int, peak: int)。

tracemalloc.**get_tracemalloc_memory**()

tracemalloc モジュールがメモリブロックのトレースを保存するのに使用しているメモリ使用量をバイト単位で取得します。*int* を返します。

tracemalloc.**is_tracing**()

tracemalloc モジュールが Python のメモリ割り当てをトレースしていれば True を、そうでなければ False を返します。

start() ならびに *stop()* 関数も参照してください。

tracemalloc.**start**(*nframe: int=1*)

Python のメモリ割り当てのトレースを開始します: Python メモリアロケータにフックします。トレースの収集されたトレースバックは *nframe* フレームに制限されます。デフォルトでは、あるブロックのトレースは最新のフレームのみを保存します、つまり上限は 1 です。*nframe* は 1 以上でなければなりません。

1 より多くのフレームを保存するのは 'traceback' でグループ化された統計や累積的な統計を計算する場合にのみ有用です。*Snapshot.compare_to()* および *Snapshot.statistics()* メソッドを参照してください。

保存するフレーム数を増やすと *tracemalloc* モジュールのメモリと CPU のオーバーヘッドは増加します。*tracemalloc* モジュールが使用しているメモリ量を調べるには *get_tracemalloc_memory()* 関数を使用してください。

PYTHONTRACEMALLOC 環境変数 (PYTHONTRACEMALLOC=NFRAME) と -X tracemalloc=NFRAME コマンドラインオプションを使って実行開始時にトレースを始めることが出来ます。

stop()、*is_tracing()*、*get_traceback_limit()* 関数を参照してください。

tracemalloc.**stop**()

Python のメモリ割り当てのトレースを停止します。つまり、Python のメモリ割り当てへのフックをアンインストールします。Python が割り当てたメモリブロックについてこれまで集めたトレースも全てクリアします。

トレースが全部クリアされる前にスナップショットを取りたい場合は *take_snapshot()* 関数を呼んでください。

start()、*is_tracing()*、*clear_traces()* 関数も参照してください。

tracemalloc.**take_snapshot**()

Python が割り当てたメモリブロックのトレースのスナップショットを取ります。新しい *Snapshot* インスタンスを返します。

スナップショットは *tracemalloc* モジュールがメモリ割り当てのトレースを始める前に割り当てられたメモリブロックを含みません。

トレースのトレースバックは *get_traceback_limit()* フレームに制限されています。より多くのフ

レームを保存するには *start()* 関数の *nframe* 引数を使用してください。

スナップショットを取るには *tracemalloc* モジュールはメモリ割り当てをトレースしていなければなりません。*start()* 関数を参照してください。

get_object_traceback() 関数を参照してください。

DomainFilter

class tracemalloc.**DomainFilter**(*inclusive: bool*, *domain: int*)

Filter traces of memory blocks by their address space (domain).

バージョン 3.6 で追加.

inclusive

If *inclusive* is `True` (include), match memory blocks allocated in the address space *domain*.

If *inclusive* is `False` (exclude), match memory blocks not allocated in the address space *domain*.

domain

Address space of a memory block (`int`). Read-only property.

Filter

class tracemalloc.**Filter**(*inclusive: bool*, *filename_pattern: str*, *lineno: int=None*, *all_frames: bool=False*, *domain: int=None*)

メモリブロックのトレースをフィルターします。

filename_pattern のシンタックスについては *fnmatch.fnmatch()* 関数を参照してください。`'.pyc'` 拡張子は `'.py'` に置換されます。

例:

- `Filter(True, subprocess.__file__)` は *subprocess* モジュールのみを含みます
- `Filter(False, tracemalloc.__file__)` は *tracemalloc* モジュールのトレースを除外します
- `Filter(False, "<unknown>")` は空のトレースバックを除外します

バージョン 3.5 で変更: `'.pyo'` ファイル拡張子が `'.py'` に置換されることはもうありません。

バージョン 3.6 で変更: *domain* 属性が追加されました。

domain

Address space of a memory block (`int` or `None`).

tracemalloc uses the domain `0` to trace memory allocations made by Python. C extensions can use other domains to trace other resources.

inclusive
If *inclusive* is `True` (include), only match memory blocks allocated in a file with a name matching *filename_pattern* at line number *lineno*.

If *inclusive* is `False` (exclude), ignore memory blocks allocated in a file with a name matching *filename_pattern* at line number *lineno*.

lineno
フィルタの行番号です (`int`)。*lineno* が `None` の場合フィルタはあらゆる行番号にマッチします。

filename_pattern
フィルタのファイル名のパターンです (`str`)。読み出し専用のプロパティです。

all_frames
all_frames が `True` の場合トレースバックの全てのフレームをチェックします。*all_frames* が `False` の場合最新のフレームをチェックします。

トレースバックの上限が `1` の場合この属性の影響はありません。*get_traceback_limit()* 関数と *Snapshot.traceback_limit* 属性を参照してください。

Frame

class tracemalloc.Frame
トレースバックのフレームです。

Traceback クラスは *Frame* インスタンスのシークエンスです。

filename
ファイル名 (`str`)。

lineno
行番号 (`int`)。

Snapshot

class tracemalloc.Snapshot
Python が割り当てたメモリブロックのトレースのスナップショットです。

take_snapshot() 関数はスナップショットのインスタンスを作ります。

compare_to(*old_snapshot: Snapshot*, *key_type: str*, *cumulative: bool=False*)
古いスナップショットとの差を計算します。*key_type* でグループ化された *StatisticDiff* インスタンスのソート済みリストとして統計を取得します。

key_type および *cumulative* 引数については *Snapshot.statistics()* メソッドを参照してください。

結果は降順でソートされます: キーは *StatisticDiff.size_diff* の絶対値、*StatisticDiff.size*、*StatisticDiff.count_diff* の絶対置、*Statistic.count*、そして *StatisticDiff.traceback* です。

dump(*filename*)

スナップショットをファイルに書き込みます。

スナップショットをリロードするには *load()* を使用します。

filter_traces(*filters*)

Create a new *Snapshot* instance with a filtered *traces* sequence, *filters* is a list of *DomainFilter* and *Filter* instances. If *filters* is an empty list, return a new *Snapshot* instance with a copy of the traces.

All inclusive filters are applied at once, a trace is ignored if no inclusive filters match it. A trace is ignored if at least one exclusive filter matches it.

バージョン 3.6 で変更: *DomainFilter* instances are now also accepted in *filters*.

classmethod load(*filename*)

スナップショットをファイルからロードします。

dump() を参照してください。

statistics(*key_type: str*, *cumulative: bool=False*)

key_type でグループ化された *Statistic* インスタンスのソート済みリストとして統計を取得します:

key_type	description
`'filename'`	ファイル名
`'lineno'`	ファイル名と行番号
`'traceback'`	traceback

cumulative が `True` の場合、最新のフレームだけでなく、トレースのトレースバックの全フレームのメモリーブロックについて大きさと数を累積します。累積モードは *key_type* が `'filename'` および `'lineno'` と等しい場合にのみ使用することが出来ます。

結果は降順でソートされます: キーは *Statistic.size*, *Statistic.count*, *Statistic.traceback* です。

traceback_limit

traces のトレースバック内に保存されるフレーム数の最大値です。スナップショットが取られたときの *get_traceback_limit()* の結果です。

traces

Python が割り当てた全メモリブロックのトレースで、*Trace* インスタンスのシークェンスです。

シークェンスの順序は未定義です。統計のソート済みリストを取得するには *Snapshot.statistics()* を使用してください。

Statistic

class tracemalloc.Statistic
メモリ割り当ての統計です。

Snapshot.statistics() は *Statistic* インスタンスの一覧を返します。

StatisticDiff クラスも参照してください。

count
メモリブロック数 (int)。

size
メモリブロックのバイト単位の総サイズ (int)。

traceback
メモリブロックが割り当てられているトレースバック。*Traceback* インスタンス。

StatisticDiff

class tracemalloc.StatisticDiff
新旧 *Snapshot* インスタンスのメモリ割り当ての統計差です。

Snapshot.compare_to() は *StatisticDiff* インスタンスのリストを返します。*Statistic* クラスも参照してください。

count
新しいスナップショット内のメモリブロックの数 (int) です。新しいスナップショット内でメモリブロックが解放された場合は 0 です。

count_diff
新旧スナップショットのメモリブロック数の差 (int) です。メモリブロックが新しいスナップショット内で割り当てられた場合は 0 です。

size
新しいスナップショット内のメモリブロックのバイト単位での総サイズ (int) です。新しいスナップショット内でメモリブロックが解放された場合は 0 です。

size_diff
新旧スナップショットのバイト単位での総サイズの差 (int) です。メモリブロックが新しいスナップショット内で割り当てられた場合は 0 です。

traceback
メモリブロックが割り当てられたトレースバックで、*Traceback* のインスタンスです。

Trace

class tracemalloc.Trace
メモリブロックをトレースします。

Snapshot.traces 属性は *Trace* インスタンスのシークエンスです。

バージョン 3.6 で変更: *domain* 属性が追加されました。

domain
Address space of a memory block (int). Read-only property.

tracemalloc uses the domain 0 to trace memory allocations made by Python. C extensions can use other domains to trace other resources.

size
メモリブロックのバイト単位のサイズ (int)。

traceback
メモリブロックが割り当てられているトレースバック。*Traceback* インスタンス。

Traceback

class tracemalloc.Traceback
Sequence of *Frame* instances sorted from the oldest frame to the most recent frame.

A traceback contains at least 1 frame. If the tracemalloc module failed to get a frame, the filename "<unknown>" at line number 0 is used.

When a snapshot is taken, tracebacks of traces are limited to *get_traceback_limit()* frames. See the *take_snapshot()* function.

Trace.traceback 属性は TTraceback のインスタンスです。

バージョン 3.7 で変更: Frames are now sorted from the oldest to the most recent, instead of most recent to oldest.

format(*limit=None*, *most_recent_first=False*)
Format the traceback as a list of lines with newlines. Use the *linecache* module to retrieve lines from the source code. If *limit* is set, format the *limit* most recent frames if *limit* is positive. Otherwise, format the abs(limit) oldest frames. If *most_recent_first* is True, the order of the formatted frames is reversed, returning the most recent frame first instead of last.

Similar to the *traceback.format_tb()* function, except that *format()* does not include newlines.

以下はプログラム例です:

```
print("Traceback (most recent call first):")
for line in traceback:
    print(line)
```

出力:

```
Traceback (most recent call first):
  File "test.py", line 9
    obj = Object()
  File "test.py", line 12
    tb = tracemalloc.get_object_traceback(f())
```

第

TWENTYEIGHT

ソフトウェア・パッケージと配布

以下のライブラリは Python で書かれたソフトウェアを配布、インストールするためのものです。これらは Python Package Index に対して動作するように設計されていますが、ローカルのインデックスサーバーや、インデックスサーバーなしに使うこともできます。

28.1 distutils --- Python モジュールの構築とインストール

distutils パッケージは、現在インストールされている Python に追加するためのモジュール構築、および実際のインストールを支援します。新規のモジュールは 100%-pure Python でも、C で書かれた拡張モジュールでも、あるいは Python と C 両方のコードが入っているモジュールからなる Python パッケージでもかまいません。

Python ユーザの大半はこのパッケージを直接使い **たくはない** でしょうが、代わりに Python Packaging Authority が保守しているクロスバージョンツールを使うでしょう。特に、setuptools は *distutils* の改良された代替品で、以下を提供しています:

- プロジェクトの依存性の宣言のサポート
- ソースのリリースの際どのファイルを含めるか指定する追加の機構 (バージョン管理システムとの統合のためのプラグインも含む)
- プロジェクトの ”エントリーポイント” を宣言する機能、アプリケーションプラグインシステムとして使うことができます
- インストール時に事前にビルドすることなく、Windows コマンドライン実行ファイルを自動的に生成する機能
- サポートしている Python の全バージョンで一貫性のある挙動

たとえスクリプト自身が `distutils` のみをインポートしていても、推奨される pip インストーラは `setuptools` で全 `setup.py` スクリプトを実行します。詳細は Python Packaging User Guide を参照してください。

現在のパッケージと配布システムへの理解を深めようとしている著者やユーザのために、レガシーな *distutils* に基づくユーザドキュメントと API のリファレンスは利用可能なままになっています。

- install-index
- distutils-index

28.2 ensurepip --- pip インストーラのブートストラップ

バージョン 3.4 で追加.

ensurepip パッケージは pip インストーラを既にインストールされている Python 環境や仮想環境にブートストラップする助けになります。このブートストラップのアプローチは pip が独立したリリースサイクルを持ち、最新の利用可能な安定版が CPython リファレンスインタープリタのメンテナンスリリースや feature リリースにバンドルされていることを反映しています。

ほとんどの場合、Python のエンドユーザーがこのモジュールを直接呼び出す必要はないでしょう (pip はデフォルトでブートストラップされるからです)。しかし、もし Python のインストール時に pip のインストールをスキップしたり、仮想環境を構築したり、明示的に pip をアンインストールした場合、直接呼び出す必要があるかもしれません。

注釈: このモジュールはインターネットに **アクセスしません**。pip のブートストラップに必要な全てはこのパッケージの一部として含まれています。

参考:

installing-index エンドユーザーが Python パッケージをインストールする際のガイドです。

PEP 453: Python **インストールの際の明示的な** pip **のブートストラッピング** このモジュールのもともとの論拠と仕様。

28.2.1 コマンドラインインターフェイス

コマンドラインインターフェースを起動するには -m スイッチをつけてインタープリターを使用します。

最も簡単な起動方法は:

```
python -m ensurepip
```

この起動方法は pip をインストールします。既にインストールされていた場合は何もしません。インストールされた pip のバージョンを ensurepip にバンドルされているもののうち、できるだけ新しいものにするためには、--upgrade オプションを追加して:

```
python -m ensurepip --upgrade
```

デフォルトでは、pip は現在の仮想環境 (もしアクティブなら) か、システムのサイトパッケージ (もしアクティブな仮想環境がなければ) にインストールされます。インストール先は 2 つの追加コマンドラインオプションで制御できます:

- `--root <dir>`: 現在のアクティブな仮想環境 (もしあれば) の root や現在インストールされている Python の root ディレクトリに入れる代わりに、与えられたディレクトリを root として `pip` をインストールします。

- `--user`: は、現在インストールされている Python にグローバルにインストールされる代わりに、ユーザーの site packages ディレクトリに `pip` をインストールします (このオプションはアクティブな仮想環境のもとでは許可されません)。

デフォルトでは `pipX` と `pipX.Y` がインストールされます (X.Y は `ensurepip` を起動した Python のバージョン)。インストールされるスクリプトは 2 つの追加コマンドラインオプションで制御できます:

- `--altinstall`: alternate インストール。X.Y でバージョン付けされたものだけがインストールされます。

- `--default-pip`: "default pip" のインストールが要求されると、通常の二つのスクリプトに加えて `pip` スクリプトがインストールされます。

2 つのスクリプト選択オプションを指定すると例外が発生します。

28.2.2 モジュール API

ensurepip はプログラムから利用出来る 2 つの関数を公開しています:

`ensurepip.version()`
環境にブートストラップする際にインストールされることになる pip のバンドルバージョンを示す文字列を返します。

`ensurepip.bootstrap`(*root=None*, *upgrade=False*, *user=False*, *altinstall=False*, *default_pip=False*, *verbosity=0*)
現在の環境あるいは指示された環境へ `pip` をブートストラップします。

root で、インストールの root ディレクトリを変更します。*root* が `None` の場合は、インストールは現在の環境でのデフォルトの場所を使います。

upgrade で、`pip` のバンドルのバージョンとして、インストール済みの以前のバージョンをアップグレードするかどうかを指定します。

user で、グローバルなインストールではなく user スキームを使うかどうかを指定します。

デフォルトではスクリプト `pipX` と `pipX.Y` はインストールされます (X.Y は Python の現在のバージョンです)。

altinstall が設定されていた場合は `pipX` はインストール **されません**。

default_pip がセットされていれば、`pip` スクリプトが 2 つの標準スクリプトと共にインストールされます。

altinstall と *default_pip* の両方を指定すると、*ValueError* を起こします。

verbosity でブートストラップ操作からの *sys.stdout* への出力の冗長レベルをコントロールします。

引数 root 付きで 監査イベント ensurepip.bootstrap を送出します。

注釈: ブートストラップ処理は sys.path, os.environ の両方に対して副作用を持ちます。代わりに、サブプロセスとしてコマンドラインインターフェイスを使うことで、これら副作用を避けることが出来ます。

注釈: ブートストラップ処理は pip によって必要とされるモジュールを追加インストールするかもしれませんが、ほかのソフトウェアはそれら依存物がいつもデフォルトで存在していることを仮定すべきではありません (将来のバージョンの pip ではその依存はなくなるかもしれませんので)。

28.3 venv --- 仮想環境の作成

バージョン 3.3 で追加.

ソースコード: Lib/venv/

---

venv モジュールは、軽量な ”仮想環境” の作成のサポートを提供します。仮想環境には、仮想環境ごとの site ディレクトリがあり、これはシステムの site ディレクトリから分離させることができます。それぞれの仮想環境には、それ自身に (この仮想環境を作成するのに使ったバイナリのバージョンに合った) Python バイナリがあり、仮想環境ごとの site ディレクトリに独立した Python パッケージ群をインストールできます。

Python 仮想環境に関してより詳しくは **PEP 405** を参照してください。

参考:

Python Packaging User Guide: Creating and using virtual environments

28.3.1 仮想環境の作成

仮想環境 を作成するには venv コマンドを実行します:

```
python3 -m venv /path/to/new/virtual/environment
```

このコマンドを実行すると、ターゲットディレクトリ (および必要なだけの親ディレクトリ) が作成され、その中に pyvenv.cfg ファイルが置かれます。そのファイルの home キーはこのコマンドを呼び出した Python のインストール場所を指します (よく使われるターゲットディレクトリの名前は .venv です)。このコマンドはまた、Python バイナリのコピーまたはシンボリックリンク (のプラットフォームあるいは仮想環境作成時に使われた引数に対して適切な方) を含む bin (Windows では Scripts) サブディレクトリを作成します。さらに、lib/pythonX.Y/site-packages (Windows では Lib\site-packages) サブディレクトリも (最初は空の状態で) 作成します。指定したディレクトリが存在している場合は、それが再利用されます。

バージョン 3.6 で非推奨: Python 3.3 と 3.4 では、仮想環境の作成に推奨していたツールは pyvenv でしたが、Python 3.6 では非推奨です。

バージョン 3.5 で変更: 仮想環境の作成には、venv の使用をお勧めします。

Windows では、venv コマンドは次のように実行します:

```
c:\>c:\Python35\python -m venv c:\path\to\myenv
```

あるいは、インストールされている Python のために PATH 変数や PATHEXT 変数が設定してある場合は次のコマンドでも実行できます:

```
c:\>python -m venv c:\path\to\myenv
```

このコマンドを -h をつけて実行すると利用できるオプションが表示されます:

```
usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear]
            [--upgrade] [--without-pip] [--prompt PROMPT]
            ENV_DIR [ENV_DIR ...]

Creates virtual Python environments in one or more target directories.

positional arguments:
  ENV_DIR               A directory to create the environment in.

optional arguments:
  -h, --help            show this help message and exit
  --system-site-packages
                        Give the virtual environment access to the system
                        site-packages dir.
  --symlinks            Try to use symlinks rather than copies, when symlinks
                        are not the default for the platform.
  --copies              Try to use copies rather than symlinks, even when
                        symlinks are the default for the platform.
  --clear               Delete the contents of the environment directory if it
                        already exists, before environment creation.
  --upgrade             Upgrade the environment directory to use this version
                        of Python, assuming Python has been upgraded in-place.
  --without-pip         Skips installing or upgrading pip in the virtual
                        environment (pip is bootstrapped by default)
  --prompt PROMPT       Provides an alternative prompt prefix for this
                        environment.

Once an environment has been created, you may wish to activate it, e.g. by
sourcing an activate script in its bin directory.
```

バージョン 3.4 で変更: デフォルトで pip をインストールします。--without-pip と --copies オプションを追加しました。

バージョン 3.4 で変更: 以前のバージョンでは、対象となるディレクトリが既に存在していた場合は、--clear オプションや --upgrade オプションを付けない限りはエラーを送出していました。

注釈: Windows でもシンボリックリンクはサポートされていますが、シンボリックリンクを使うのは推奨されません。特に注目すべきなのは、ファイルエクスプローラ上で `python.exe` をダブルクリックすると、シンボリックリンクを貪欲に解決し仮想環境を無視するということです。

注釈: Microsoft Windows では、ユーザー向けの実行ポリシーを設定して `Activate.ps1` スクリプトが使えるようにする必要があるかもしれません。この設定は次の PowerShell コマンドでできます:

PS C:> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser

さらに詳しくは About Execution Policies を参照してください。

作成された `pyvenv.cfg` ファイルには、`include-system-site-packages` キーも含まれます。これは `venv` が `--system-site-packages` オプションをつけて実行されたなら `true` で、そうでなければ `false` です。

`--without-pip` オプションが与えられない限り、`pip` を仮想環境でブートするために *`ensurepip`* が呼ばれます。

`venv` には複数のパスを渡すことができ、その場合はそれぞれのパスに同一の仮想環境が作成されます。

仮想環境の作成が完了すれば、その仮想環境のバイナリディレクトリにあるスクリプトで " 有効化" できます。スクリプトの呼び出し方はプラットフォーム固有です (*<venv>* の部分は、仮想環境があるディレクトリのパスに置き換える必要があります):

プラットフォーム	シェル	仮想環境を有効化するためのコマンド
POSIX	bash/zsh	$ source <venv>/bin/activate
	fish	$. <venv>/bin/activate.fish
	csh/tcsh	$ source <venv>/bin/activate.csh
	PowerShell Core	$ <venv>/bin/Activate.ps1
Windows	cmd.exe	C:\> <venv>\Scripts\activate.bat
	PowerShell	PS C:\> <venv>\Scripts\Activate.ps1

When a virtual environment is active, the `VIRTUAL_ENV` environment variable is set to the path of the virtual environment. This can be used to check if one is running inside a virtual environment.

環境を有効化するのに特別な指定は **必要** ありません; 有効化は仮想環境のバイナリディレクトリをパスの最初に加えて、"python" で仮想環境の Python インタプリタが呼び出されるようにし、フルパスを入力せずにインストールされたスクリプトを実行できるようにするだけです。しかし、インストールされたすべてのスクリプトは有効化しなくても実行可能で、仮想環境の Python で自動的に実行されなければなりません。

シェルで "deactivate" と入力することで仮想環境を無効化できます。厳密な仕組みはプラットフォーム固有であり、内部の実装詳細です (たいていはスクリプトかシェル関数が使われます)。

バージョン 3.4 で追加: `fish` および `csh` の有効化スクリプト。

バージョン 3.8 で追加: PowerShell Core のサポートのために POSIX 環境にインストールされた、PowerShell 有効化スクリプト。

注釈: 仮想環境とは、その中にインストールされた Python インタープリタ、ライブラリ、そしてスクリプトが、他の仮想環境にインストールされたものから隔離されている Python 環境です。そして (デフォルトでは) 仮想環境は ” システム” の Python、すなわち OS の一部としてインストールされた Python にインストールされている全てのライブラリからも隔離されています。

仮想環境は、Python 実行ファイルと、それが仮想環境であることを示す幾つかのファイルを含んだディレクトリツリーです。

setuptools や pip などの一般的なインストールツールは期待通りに仮想環境と連係します。言い換えると、仮想環境が有効なときには、Python パッケージを仮想環境へインストールするのに、インストールツールへの明示的な指示は必要ありません。

仮想環境が有効な場合 (すなわち、仮想環境の Python インタープリタを実行しているとき)、*sys.prefix* と *sys.exec_prefix* は仮想環境のベースディレクトリを示します。代わりに *sys.base_prefix* と *sys.base_exec_prefix* が仮想環境を作るときに使った、仮想環境ではない環境の Python がインストールされている場所を示します。仮想環境が無効の時は、*sys.prefix* は *sys.base_prefix* と、*sys.exec_prefix* は *sys.base_exec_prefix* と同じになります (全て仮想環境ではない環境の Python のインストール場所を示します)。

仮想環境が有効なときに、不注意で仮想環境の外にプロジェクトがインストールされることを避けるために、インストールパスを変更するあらゆるオプションは *distutils* の全ての設定ファイルによって無視されます。

コマンドシェルで作業をしているとき、仮想環境の実行可能ファイルディレクトリにある `activate` スクリプト (正確なファイル名とそのファイルを使うコマンドはシェル依存です) を実行して、仮想環境を有効化できます。このスクリプトは仮想環境の実行可能ファイルのディレクトリを、実行中のシェルの `PATH` 環境変数の先頭に差し込みます。それ以外の状況では、仮想環境を有効化するための作業は必要無いはずです。仮想環境にインストールされたスクリプトには、仮想環境の Python インタプリタを指す ”shebang” 行があります。これはつまり、スクリプトは `PATH` の値に関係無く、仮想環境のインタプリタで実行されるということです。Windows では、Windows 用の Python ランチャがインストールされている場合は ”shebang” 行の処理がサポートされています (この機能はバージョン 3.3 で Python に追加されました - 詳細は **PEP 397** を参照してください)。従って、インストールされたスクリプトを Windows のエクスプローラのウィンドウからダブルクリックすると、`PATH` にその仮想環境への参照を持たせる必要無く、そのスクリプトが正しいインタープリタで実行されるはずです。

28.3.2 API

上述の高水準のメソッドは、サードパーティの仮想環境の作成者が環境の作成を必要に応じてカスタマイズするための機構を提供する簡素な API を利用します。それが *EnvBuilder* クラスです。

class venv.EnvBuilder(*system_site_packages=False*, *clear=False*, *symlinks=False*, *upgrade=False*, *with_pip=False*, *prompt=None*)

EnvBuilder クラスを実体化するときに、以下のキーワード引数を受け取ります:

- `system_site_packages` -- 真偽値で、システムの Python の site-packages を仮想環境から利用できるかどうかを示します (デフォルト: `False`)。
- `clear` -- 真偽値で、真の場合環境を作成する前に既存の対象ディレクトリの中身を削除します。
- `symlinks` -- 真偽値で、Python のバイナリをコピーせずにシンボリックの作成を試みるかどうかを示します。
- `upgrade` -- 真偽値で、真の場合実行中の Python で既存の環境をアップグレードします。その Python がインプレースでアップグレードされたときに用います。デフォルトは `False` です。
- `with_pip` -- 真偽値で、真の場合仮想環境に pip がインストールされていることを保証します。`--default-pip` オプションで *ensurepip* を使用します。
- `prompt` -- 仮想環境が有効になっているときに使われる文字列 (デフォルトは `None` で、その環境のディレクトリ名が使われることになります)。

バージョン 3.4 で変更: `with_pip` 引数が追加されました。

バージョン 3.6 で追加: `prompt` 引数が追加されました。

サードパーティーの仮想環境ツールの作成者は、*EnvBuilder* を継承して使うことができます。

返される env-builder オブジェクトには `create` というメソッドがあります:

create(*env_dir*)

仮想環境を持つことになるターゲットディレクトリ (絶対パスあるいは現在のディレクトリからの相対パス) を指定し、仮想環境を作成します。`create` メソッドは、指定されたディレクトリに仮想環境を構築するか、適切な例外を送出します。

EnvBuilder クラスの `create` メソッドは、サブクラスのカスタマイズに使えるフックを説明します:

```
def create(self, env_dir):
    """
    Create a virtualized Python environment in a directory.
    env_dir is the target directory to create an environment in.
    """
    env_dir = os.path.abspath(env_dir)
    context = self.ensure_directories(env_dir)
    self.create_configuration(context)
    self.setup_python(context)
```

(次のページに続く)

(前のページからの続き)

```
    self.setup_scripts(context)
    self.post_setup(context)
```

メソッド *ensure_directories()*, *create_configuration()*, *setup_python()*, *setup_scripts()*, *post_setup()* はそれぞれオーバーライドできます。

ensure_directories(*env_dir*)

仮想環境のディレクトリと全ての必要なディレクトリを作成し、コンテキストオブジェクトを返します。これは (パスなどの) 属性を保持しているだけのオブジェクトで、他のメソッドから使うためのものです。既存の仮想環境ディレクトリ上での操作を許可するために `clear` や `upgrade` が指定されている場合に限り、作成されるディレクトリは既に存在していても構いません。

create_configuration(*context*)

仮想環境に `pyvenv.cfg` 設定ファイルを作成します。

setup_python(*context*)

Python 実行ファイルのコピーまたはシンボリックリンクを仮想環境に作成します。POSIX システムで、特定の `python3.x` 実行ファイルが使われている場合、同じ名前のファイルが既に存在していない限り、`python` および `python3` へのシンボリックリンクがその実行ファイルを指すように作成されます。

setup_scripts(*context*)

プラットフォームに対応した有効化スクリプトを仮想環境にインストールします。

post_setup(*context*)

サードパーティーライブラリがオーバーライドするための空のメソッドです。このメソッドをオーバーライドして、仮想環境構築後にパッケージのプリインストールなどのステップを実装できます。

バージョン 3.7.2 で変更: Windows では、バイナリそのものをコピーするのではなく、`python[w].exe` にリダイレクトを行うスクリプトを使うようになりました。3.7.2 でのみ、ソース群でビルドを実行しているのでなければ *setup_python()* だけは何もしません。

バージョン 3.7.3 で変更: Windows では、*setup_scripts()* の代わりに *setup_python()* の一部としてリダイレクトを行うスクリプトをコピーします。これは 3.7.2 には当てはまりません。シンボリックリンクを使ったときは、オリジナルの実行ファイルをコピーします。

これらに加えて、*EnvBuilder* は *setup_scripts()* やサブクラスの *post_setup()* が仮想環境にスクリプトをインストールするためのユーティリティーメソッドを提供しています。

install_scripts(*context*, *path*)

path は "common", "posix", "nt" ディレクトリを格納したディレクトリへのパスです。各サブディレクトリには仮想環境の bin ディレクトリにインストールするスクリプトを格納します。"common" の中身と、*os.name* に一致するディレクトリの中身を、以下の置換処理を行いながらコピーします:

- `__VENV_DIR__` は仮想環境ディレクトリの絶対パスに置換されます。

- `__VENV_NAME__` は仮想環境の名前 (仮想環境ディレクトリのパスの最後の部分) に置換されます。
- `__VENV_PROMPT__` はプロンプトに置換されます (括弧で囲まれ空白が続く環境名)。
- `__VENV_BIN_NAME__` は bin ディレクトリ名 (`bin` か `Scripts`) に置換されます。
- `__VENV_PYTHON__` は仮想環境の Python 実行ファイルの絶対パスに置換されます。

(既存環境のアップグレード中は) ディレクトリは存在しても構いません。

モジュールレベルの簡易関数もあります:

venv.create(*env_dir*, *system_site_packages=False*, *clear=False*, *symlinks=False*, *with_pip=False*, *prompt=None*)

EnvBuilder を指定されたキーワード引数を使って作成し、その *create()* メソッドに *env_dir* 引数を渡して実行します。

バージョン 3.3 で追加.

バージョン 3.4 で変更: `with_pip` 引数が追加されました。

バージョン 3.6 で変更: `prompt` 引数が追加されました。

28.3.3 EnvBuilder を拡張する例

次のスクリプトで、作成された仮想環境に setuptools と pip をインストールするサブクラスを実装して *EnvBuilder* を拡張する方法を示します:

```
import os
import os.path
from subprocess import Popen, PIPE
import sys
from threading import Thread
from urllib.parse import urlparse
from urllib.request import urlretrieve
import venv

class ExtendedEnvBuilder(venv.EnvBuilder):
    """
    This builder installs setuptools and pip so that you can pip or
    easy_install other packages into the created virtual environment.

    :param nodist: If true, setuptools and pip are not installed into the
                   created virtual environment.
    :param nopip: If true, pip is not installed into the created
                  virtual environment.
    :param progress: If setuptools or pip are installed, the progress of the
                     installation can be monitored by passing a progress
                     callable. If specified, it is called with two
                     arguments: a string indicating some progress, and a
                     context indicating where the string is coming from.
```

(次のページに続く)

(前のページからの続き)

```
                    The context argument can have one of three values:
                    'main', indicating that it is called from virtualize()
                    itself, and 'stdout' and 'stderr', which are obtained
                    by reading lines from the output streams of a subprocess
                    which is used to install the app.

                    If a callable is not specified, default progress
                    information is output to sys.stderr.
    """

    def __init__(self, *args, **kwargs):
        self.nodist = kwargs.pop('nodist', False)
        self.nopip = kwargs.pop('nopip', False)
        self.progress = kwargs.pop('progress', None)
        self.verbose = kwargs.pop('verbose', False)
        super().__init__(*args, **kwargs)

    def post_setup(self, context):
        """
        Set up any packages which need to be pre-installed into the
        virtual environment being created.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        os.environ['VIRTUAL_ENV'] = context.env_dir
        if not self.nodist:
            self.install_setuptools(context)
        # Can't install pip without setuptools
        if not self.nopip and not self.nodist:
            self.install_pip(context)

    def reader(self, stream, context):
        """
        Read lines from a subprocess' output stream and either pass to a progress
        callable (if specified) or write progress information to sys.stderr.
        """
        progress = self.progress
        while True:
            s = stream.readline()
            if not s:
                break
            if progress is not None:
                progress(s, context)
            else:
                if not self.verbose:
                    sys.stderr.write('.')
                else:
                    sys.stderr.write(s.decode('utf-8'))
                sys.stderr.flush()
        stream.close()
```

(次のページに続く)

(前のページからの続き)

```
    def install_script(self, context, name, url):
        _, _, path, _, _, _ = urlparse(url)
        fn = os.path.split(path)[-1]
        binpath = context.bin_path
        distpath = os.path.join(binpath, fn)
        # Download script into the virtual environment's binaries folder
        urlretrieve(url, distpath)
        progress = self.progress
        if self.verbose:
            term = '\n'
        else:
            term = ''
        if progress is not None:
            progress('Installing %s ...%s' % (name, term), 'main')
        else:
            sys.stderr.write('Installing %s ...%s' % (name, term))
            sys.stderr.flush()
        # Install in the virtual environment
        args = [context.env_exe, fn]
        p = Popen(args, stdout=PIPE, stderr=PIPE, cwd=binpath)
        t1 = Thread(target=self.reader, args=(p.stdout, 'stdout'))
        t1.start()
        t2 = Thread(target=self.reader, args=(p.stderr, 'stderr'))
        t2.start()
        p.wait()
        t1.join()
        t2.join()
        if progress is not None:
            progress('done.', 'main')
        else:
            sys.stderr.write('done.\n')
        # Clean up - no longer needed
        os.unlink(distpath)

    def install_setuptools(self, context):
        """
        Install setuptools in the virtual environment.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        url = 'https://bitbucket.org/pypa/setuptools/downloads/ez_setup.py'
        self.install_script(context, 'setuptools', url)
        # clear up the setuptools archive which gets downloaded
        pred = lambda o: o.startswith('setuptools-') and o.endswith('.tar.gz')
        files = filter(pred, os.listdir(context.bin_path))
        for f in files:
            f = os.path.join(context.bin_path, f)
            os.unlink(f)
```

(次のページに続く)

(前のページからの続き)

```
    def install_pip(self, context):
        """
        Install pip in the virtual environment.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        url = 'https://bootstrap.pypa.io/get-pip.py'
        self.install_script(context, 'pip', url)

def main(args=None):
    compatible = True
    if sys.version_info < (3, 3):
        compatible = False
    elif not hasattr(sys, 'base_prefix'):
        compatible = False
    if not compatible:
        raise ValueError('This script is only for use with '
                         'Python 3.3 or later')
    else:
        import argparse

        parser = argparse.ArgumentParser(prog=__name__,
                                         description='Creates virtual Python '
                                                     'environments in one or '
                                                     'more target '
                                                     'directories.')
        parser.add_argument('dirs', metavar='ENV_DIR', nargs='+',
                            help='A directory in which to create the
                                 'virtual environment.')
        parser.add_argument('--no-setuptools', default=False,
                            action='store_true', dest='nodist',
                            help="Don't install setuptools or pip in the "
                                 "virtual environment.")
        parser.add_argument('--no-pip', default=False,
                            action='store_true', dest='nopip',
                            help="Don't install pip in the virtual "
                                 "environment.")
        parser.add_argument('--system-site-packages', default=False,
                            action='store_true', dest='system_site',
                            help='Give the virtual environment access to the '
                                 'system site-packages dir.')
        if os.name == 'nt':
            use_symlinks = False
        else:
            use_symlinks = True
        parser.add_argument('--symlinks', default=use_symlinks,
                            action='store_true', dest='symlinks',
                            help='Try to use symlinks rather than copies, '
                                 'when symlinks are not the default for '
                                 'the platform.')
```

(次のページに続く)

(前のページからの続き)

```
        parser.add_argument('--clear', default=False, action='store_true',
                            dest='clear', help='Delete the contents of the '
                                               'virtual environment '
                                               'directory if it already '
                                               'exists, before virtual '
                                               'environment creation.')
        parser.add_argument('--upgrade', default=False, action='store_true',
                            dest='upgrade', help='Upgrade the virtual '
                                                 'environment directory to '
                                                 'use this version of '
                                                 'Python, assuming Python '
                                                 'has been upgraded '
                                                 'in-place.')
        parser.add_argument('--verbose', default=False, action='store_true',
                            dest='verbose', help='Display the output '
                                               'from the scripts which '
                                               'install setuptools and pip.')
        options = parser.parse_args(args)
        if options.upgrade and options.clear:
            raise ValueError('you cannot supply --upgrade and --clear together.')
        builder = ExtendedEnvBuilder(system_site_packages=options.system_site,
                                     clear=options.clear,
                                     symlinks=options.symlinks,
                                     upgrade=options.upgrade,
                                     nodist=options.nodist,
                                     nopip=options.nopip,
                                     verbose=options.verbose)
        for d in options.dirs:
            builder.create(d)

if __name__ == '__main__':
    rc = 1
    try:
        main()
        rc = 0
    except Exception as e:
        print('Error: %s' % e, file=sys.stderr)
    sys.exit(rc)
```

このスクリプトは オンライン よりダウンロードすることも可能です。

28.4 zipapp --- 実行可能な Python zip 書庫を管理する

バージョン 3.5 で追加.

ソースコード: Lib/zipapp.py

このモジュールは Python コードを含む zip ファイルの作成を行うツールを提供します。zip ファイルは Python インタープリタで直接実行することが出来ます。このモジュールは コマンドラインインターフェイス

と *Python API* の両方を提供します。

28.4.1 基本的な例

実行可能なアーカイブを Python コードを含むディレクトリから作成する為に **コマンドラインインターフェイス** をどのように利用することができるかを以下に例示します。アーカイブは実行時にアーカイブ内の `myapp` モジュールから `main` 関数を実行します。

```
$ python -m zipapp myapp -m "myapp:main"
$ python myapp.pyz
<output from myapp>
```

28.4.2 コマンドラインインターフェイス

コマンドラインからプログラムとして呼び出す場合は、次の形式を使用います:

```
$ python -m zipapp source [options]
```

source がディレクトリである場合、*source* ディレクトリの内容からアーカイブを作成します。*source* がファイルである場合、*source* ファイル自身をアーカイブ化し、保存先アーカイブへコピーします。(または --info オプションが指定されている場合はファイルのシェバン行が表示されます。)

以下のオプションが解釈されます:

-o <output>, --output=<output>
: 出力を *output* に指定した名前のファイルへ書込みます。このオプションが指定されていない場合、出力先ファイル名は入力元 *source* と同じになり、`.pyz` 拡張子が付与されます。ファイル名が明示的に指定されている場合は、指定されたファイル名を使用します。(必要であれば `.pyz` 拡張子が含まれます。)

  source がアーカイブである場合は、出力先ファイル名を必ず指定しなければなりません。(*source* がアーカイブである場合は *output* を必ず *source* とは別の名前にしてください。)

-p <interpreter>, --python=<interpreter>
: 実行コマンドとしての *interpreter* を指定する `#!` 行を書庫に追加します。また、POSIX では書庫を実行可能にします。デフォルトでは `#!` 行を書かず、ファイルを実行可能にはしません。

-m <mainfn>, --main=<mainfn>
: *mainfn* を実行するアーカイブへ `__main__.py` ファイルを書込んでください。*mainfn* 引数は "pkg.mod:fn" の形式で指定します。"pkg.mod" の場所はアーカイブ内の package/module です。"fn" は指定した module から呼出すことのできる関数です。`__main__.py` ファイルが module から呼出すことのできる関数を実行します。

  書庫をコピーする際、*--main* を指定することは出来ません。

-c, --compress
: Compress files with the deflate method, reducing the size of the output file. By default, files are stored uncompressed in the archive.

書庫をコピーする際、*--compress* に効果はありません。

バージョン 3.7 で追加.

--info
診断するために書庫に埋め込まれたインタープリタを表示します。この場合、他の全てのオプションは無視され、SOURCE はディレクトリではなく書庫でなければなりません。

-h, --help
簡単な使用法を表示して終了します。

28.4.3 Python API

このモジュールは 2 つの簡便関数を定義しています:

zipapp.create_archive(*source*, *target=None*, *interpreter=None*, *main=None*, *filter=None*, *compressed=False*)
source からアプリケーション書庫を作成します。ソースは以下のいずれかです:

- ディレクトリ名、または新しいアプリケーションアーカイブがディレクトリのコンテンツから作成される場合に *path-like object* オブジェクトが参照するディレクトリ。
- 既存のアプリケーションアーカイブファイルの名前、または (*interpreter* 引数に指定した値を反映し、修正する) アーカイブへファイルがコピーされる場合に *path-like object* オブジェクトが参照するファイル。
- バイトモードの読込みで開くファイルオブジェクト。ファイルの内容がアプリケーションアーカイブとなり、ファイルオブジェクトがアーカイブの起点となります。

target 引数は作成される書庫が書き込まれる場所を決めます:

- ファイル名、または *path-like object* オブジェクトを指定した場合、アーカイブは指定したファイルへ書込まれます。
- 開いているファイルオブジェクトを指定した場合、アーカイブはそのファイルオブジェクトへ書込みを行ないます。ファイルオブジェクトは必ずバイトモードの書込みで開いてください。
- target を指定しないか `None` を渡した場合、source は必ずディレクトリでなければならず、target は source のファイル名に `.pyz` 拡張子を付与したファイル名となります。

interpreter 引数はアーカイブが実行時に使用する Python インタープリタの名前を指定します。インタープリタ名は ” シェバン” 行としてアーカイブの起点に書込まれます。POSIX では OS によってシェバンが解釈され、Windows では Python ランチャーによって扱われます。シェバン行が書込まれていない場合は *interpreter* の結果を無視します。interpreter が指定されており、target がファイル名である場合、target ファイルの実行可能ビットが設定されます。

main 引数はアーカイブのメインプログラムとして使用する callable の名前を指定します。*main* 引数は source がディレクトリであり、source が既に `__main__.py` ファイルを保持していない場合に限り、指定することができます。*main* 引数は ”pkg.module:callable” の形式を取り、アーカイブは ”pkg.module” をインポートして実行され、指定した callable を引数なしで実行します。source がディ

レクトリであり、__main__.py が含まれていない場合、*main* は無視すべきエラーとなり、作成されたアーカイブには実行可能ビットが設定されません。

The optional *filter* argument specifies a callback function that is passed a Path object representing the path to the file being added (relative to the source directory). It should return `True` if the file is to be added.

The optional *compressed* argument determines whether files are compressed. If set to `True`, files in the archive are compressed with the deflate method; otherwise, files are stored uncompressed. This argument has no effect when copying an existing archive.

source または *target* へファイルオブジェクトを指定した場合、caller が create_archive の呼出し後にオブジェクトを閉じます。

既存のアーカイブをコピーする際、ファイルオブジェクトは `read` , `readline` , `write` メソッドのみを提供します。アーカイブをディレクトリから作成する際、target がファイルオブジェクトである場合は、`zipfile.ZipFile` クラスへ渡されます。必ずクラスが必要とするメソッドを提供してください。

バージョン 3.7 で追加: *filter* と *compressed* 引数が追加されました。

`zipapp.get_interpreter`(*archive*)

アーカイブの最初の行の `#!` に指定されたインタープリタを返します。`#!` が無い場合は *None* を返します。*archive* 引数は、ファイル名またはバイトモードの読込みで開いたファイルに準じるオブジェクトを指定することができ、アーカイブの起点で決定されます。

28.4.4 使用例

ディレクトリを書庫に圧縮し、実行します。

```
$ python -m zipapp myapp
$ python myapp.pyz
<output from myapp>
```

同じことを *create_archive()* 関数を使用して行うことができます:

```
>>> import zipapp
>>> zipapp.create_archive('myapp', 'myapp.pyz')
```

POSIX でアプリケーションを直接実行可能にするには使用するインタープリタを指定します。

```
$ python -m zipapp myapp -p "/usr/bin/env python"
$ ./myapp.pyz
<output from myapp>
```

シバン行を既存の書庫で置換するには、*create_archive()* function: を使用して変更された書庫を作成します:

```
>>> import zipapp
>>> zipapp.create_archive('old_archive.pyz', 'new_archive.pyz', '/usr/bin/python3')
```

アーカイブ内のファイルを更新するには BytesIO オブジェクトを使用してメモリーへ格納し、元のファイルを上書きして置換してください。ファイルを上書きする際にエラーが発生し、元のファイルが失われる危険性があることに注意してください。このコードは上記のようなエラーからファイルを保護しませんが、プロダクションコードは保護するべきです。この方法はアーカイブがメモリーに収まる場合にのみ動作します:

```
>>> import zipapp
>>> import io
>>> temp = io.BytesIO()
>>> zipapp.create_archive('myapp.pyz', temp, '/usr/bin/python2')
>>> with open('myapp.pyz', 'wb') as f:
>>>     f.write(temp.getvalue())
```

28.4.5 インタープリタの指定

Note that if you specify an interpreter and then distribute your application archive, you need to ensure that the interpreter used is portable. The Python launcher for Windows supports most common forms of POSIX `#!` line, but there are other issues to consider:

- If you use "/usr/bin/env python" (or other forms of the "python" command, such as "/usr/bin/python"), you need to consider that your users may have either Python 2 or Python 3 as their default, and write your code to work under both versions.
- If you use an explicit version, for example "/usr/bin/env python3" your application will not work for users who do not have that version. (This may be what you want if you have not made your code Python 2 compatible).
- There is no way to say "python X.Y or later", so be careful of using an exact version like "/usr/bin/env python3.4" as you will need to change your shebang line for users of Python 3.5, for example.

Typically, you should use an "/usr/bin/env python2" or "/usr/bin/env python3", depending on whether your code is written for Python 2 or 3.

28.4.6 Creating Standalone Applications with zipapp

Using the *zipapp* module, it is possible to create self-contained Python programs, which can be distributed to end users who only need to have a suitable version of Python installed on their system. The key to doing this is to bundle all of the application's dependencies into the archive, along with the application code.

The steps to create a standalone archive are as follows:

1. Create your application in a directory as normal, so you have a `myapp` directory containing a `__main__.py` file, and any supporting application code.

2. Install all of your application's dependencies into the `myapp` directory, using pip:

   ```
   $ python -m pip install -r requirements.txt --target myapp
   ```

   (this assumes you have your project requirements in a `requirements.txt` file - if not, you can just list the dependencies manually on the pip command line).

3. Optionally, delete the `.dist-info` directories created by pip in the `myapp` directory. These hold metadata for pip to manage the packages, and as you won't be making any further use of pip they aren't required - although it won't do any harm if you leave them.

4. Package the application using:

   ```
   $ python -m zipapp -p "interpreter" myapp
   ```

This will produce a standalone executable, which can be run on any machine with the appropriate interpreter available. See **インタープリタの指定** for details. It can be shipped to users as a single file.

On Unix, the `myapp.pyz` file is executable as it stands. You can rename the file to remove the `.pyz` extension if you prefer a "plain" command name. On Windows, the `myapp.pyz[w]` file is executable by virtue of the fact that the Python interpreter registers the `.pyz` and `.pyzw` file extensions when installed.

Making a Windows executable

On Windows, registration of the `.pyz` extension is optional, and furthermore, there are certain places that don't recognise registered extensions "transparently" (the simplest example is that `subprocess.run(['myapp'])` won't find your application - you need to explicitly specify the extension).

On Windows, therefore, it is often preferable to create an executable from the zipapp. This is relatively easy, although it does require a C compiler. The basic approach relies on the fact that zipfiles can have arbitrary data prepended, and Windows exe files can have arbitrary data appended. So by creating a suitable launcher and tacking the `.pyz` file onto the end of it, you end up with a single-file executable that runs your application.

A suitable launcher can be as simple as the following:

```
#define Py_LIMITED_API 1
#include "Python.h"

#define WIN32_LEAN_AND_MEAN
#include <windows.h>

#ifdef WINDOWS
int WINAPI wWinMain(
    HINSTANCE hInstance,      /* handle to current instance */
    HINSTANCE hPrevInstance,  /* handle to previous instance */
```

(次のページに続く)

(前のページからの続き)

```
    LPWSTR lpCmdLine,         /* pointer to command line */
    int nCmdShow              /* show state of window */
)
#else
int wmain()
#endif
{
    wchar_t **myargv = _alloca((__argc + 1) * sizeof(wchar_t*));
    myargv[0] = __wargv[0];
    memcpy(myargv + 1, __wargv, __argc * sizeof(wchar_t *));
    return Py_Main(__argc+1, myargv);
}
```

If you define the `WINDOWS` preprocessor symbol, this will generate a GUI executable, and without it, a console executable.

To compile the executable, you can either just use the standard MSVC command line tools, or you can take advantage of the fact that distutils knows how to compile Python source:

```
>>> from distutils.ccompiler import new_compiler
>>> import distutils.sysconfig
>>> import sys
>>> import os
>>> from pathlib import Path

>>> def compile(src):
>>>     src = Path(src)
>>>     cc = new_compiler()
>>>     exe = src.stem
>>>     cc.add_include_dir(distutils.sysconfig.get_python_inc())
>>>     cc.add_library_dir(os.path.join(sys.base_exec_prefix, 'libs'))
>>>     # First the CLI executable
>>>     objs = cc.compile([str(src)])
>>>     cc.link_executable(objs, exe)
>>>     # Now the GUI executable
>>>     cc.define_macro('WINDOWS')
>>>     objs = cc.compile([str(src)])
>>>     cc.link_executable(objs, exe + 'w')

>>> if __name__ == "__main__":
>>>     compile("zastub.c")
```

The resulting launcher uses the "Limited ABI", so it will run unchanged with any version of Python 3.x. All it needs is for Python (`python3.dll`) to be on the user's `PATH`.

For a fully standalone distribution, you can distribute the launcher with your application appended, bundled with the Python "embedded" distribution. This will run on any PC with the appropriate architecture (32 bit or 64 bit).

Caveats

There are some limitations to the process of bundling your application into a single file. In most, if not all, cases they can be addressed without needing major changes to your application.

1. If your application depends on a package that includes a C extension, that package cannot be run from a zip file (this is an OS limitation, as executable code must be present in the filesystem for the OS loader to load it). In this case, you can exclude that dependency from the zipfile, and either require your users to have it installed, or ship it alongside your zipfile and add code to your `__main__.py` to include the directory containing the unzipped module in `sys.path`. In this case, you will need to make sure to ship appropriate binaries for your target architecture(s) (and potentially pick the correct version to add to `sys.path` at runtime, based on the user's machine).

2. If you are shipping a Windows executable as described above, you either need to ensure that your users have `python3.dll` on their PATH (which is not the default behaviour of the installer) or you should bundle your application with the embedded distribution.

3. The suggested launcher above uses the Python embedding API. This means that in your application, `sys.executable` will be your application, and *not* a conventional Python interpreter. Your code and its dependencies need to be prepared for this possibility. For example, if your application uses the *multiprocessing* module, it will need to call *multiprocessing.set_executable()* to let the module know where to find the standard Python interpreter.

28.4.7 The Python Zip Application Archive Format

Python has been able to execute zip files which contain a `__main__.py` file since version 2.6. In order to be executed by Python, an application archive simply has to be a standard zip file containing a `__main__.py` file which will be run as the entry point for the application. As usual for any Python script, the parent of the script (in this case the zip file) will be placed on *sys.path* and thus further modules can be imported from the zip file.

The zip file format allows arbitrary data to be prepended to a zip file. The zip application format uses this ability to prepend a standard POSIX "shebang" line to the file (`#!/path/to/interpreter`).

Formally, the Python zip application format is therefore:

1. An optional shebang line, containing the characters `b'#!'` followed by an interpreter name, and then a newline (`b'\n'`) character. The interpreter name can be anything acceptable to the OS "shebang" processing, or the Python launcher on Windows. The interpreter should be encoded in UTF-8 on Windows, and in *sys.getfilesystemencoding()* on POSIX.

2. Standard zipfile data, as generated by the *zipfile* module. The zipfile content *must* include a file called `__main__.py` (which must be in the "root" of the zipfile - i.e., it cannot be in a subdirectory). The zipfile data can be compressed or uncompressed.

If an application archive has a shebang line, it may have the executable bit set on POSIX systems, to allow it to be executed directly.

There is no requirement that the tools in this module are used to create application archives - the module is a convenience, but archives in the above format created by any means are acceptable to Python.

第

TWENTYNINE

PYTHON ランタイムサービス

この章では、Python インタープリタや Python 環境に深く関連する各種の機能を解説します。以下に一覧を示します:

29.1 sys --- システムパラメータと関数

このモジュールでは、インタプリタで使用・管理している変数や、インタプリタの動作に深く関連する関数を定義しています。このモジュールは常に利用可能です。

`sys.abiflags`

Python が標準的な `configure` でビルトされた POSIX システムにおいて、**PEP 3149** に記述された ABI フラグを含みます。

バージョン 3.8 で変更: デフォルトのフラグは空文字列になりました。(pymalloc のための `m` フラグが取り除かれました)

バージョン 3.2 で追加.

`sys.addaudithook(`*hook*`)`

呼び出し可能な *hook* を現在の (サブ) インタープリタのアクティブな監査用フックのリストに加えます。

When an auditing event is raised through the *`sys.audit()`* function, each hook will be called in the order it was added with the event name and the tuple of arguments. Native hooks added by `PySys_AddAuditHook()` are called first, followed by hooks added in the current (sub)interpreter. Hooks can then log the event, raise an exception to abort the operation, or terminate the process entirely.

引数無しで **監査イベント** `sys.addaudithook` を送出します。

See the *audit events table* for all events raised by CPython, and **PEP 578** for the original design discussion.

バージョン 3.8 で追加.

バージョン 3.8.1 で変更: *`RuntimeError`* ではない class:*Exception* は抑制されなくなりました。

CPython implementation detail: When tracing is enabled (see *settrace()*), Python hooks are only traced if the callable has a `__cantrace__` member that is set to a true value. Otherwise, trace functions will skip the hook.

`sys.argv`

Python スクリプトに渡されたコマンドライン引数のリスト。`argv[0]` はスクリプトの名前となりますが、フルパス名かどうかは、OS によって異なります。コマンドライン引数に `-c` を付けて Python を起動した場合、`argv[0]` は文字列 `'-c'` となります。スクリプト名なしで Python を起動した場合、`argv[0]` は空文字列になります。

標準入力もしくはコマンドライン引数で指定されたファイルのリストに渡ってループするには、*fileinput* モジュールを参照してください。

注釈: Unix では、コマンドライン引数は OS からバイト列で渡されます。Python はそれをファイルシステムエンコーディングと "surrogateescape" エラーハンドラを使ってデコードします。オリジナルのバイト列が必要なときには、`[os.fsencode(arg) for arg in sys.argv]` で取得できます。

`sys.audit`(*event*, **args*)

Raise an auditing event and trigger any active auditing hooks. *event* is a string identifying the event, and *args* may contain optional arguments with more information about the event. The number and types of arguments for a given event are considered a public and stable API and should not be modified between releases.

For example, one auditing event is named `os.chdir`. This event has one argument called *path* that will contain the requested new working directory.

sys.audit() will call the existing auditing hooks, passing the event name and arguments, and will re-raise the first exception from any hook. In general, if an exception is raised, it should not be handled and the process should be terminated as quickly as possible. This allows hook implementations to decide how to respond to particular events: they can merely log the event or abort the operation by raising an exception.

Hooks are added using the *sys.addaudithook()* or `PySys_AddAuditHook()` functions.

The native equivalent of this function is `PySys_Audit()`. Using the native function is preferred when possible.

See the *audit events table* for all events raised by CPython.

バージョン 3.8 で追加.

`sys.base_exec_prefix`

Python の起動中、`site.py` が実行される前、*exec_prefix* と同じ値が設定されます。**仮想環境**で実行されたのでなければ、この値は変化しません; `site.py` が仮想環境で使用されていると判断した場合、*prefix* および *exec_prefix* は仮想環境を示すよう変更され、*base_prefix* および *base_exec_prefix* は引き続き (仮想環境が作成された) ベースの Python インストール先を示します。

バージョン 3.3 で追加.

sys.base_prefix
: Python の起動中、site.py が実行される前、*prefix* と同じ値が設定されます。**仮想環境** で実行されたのでなければ、この値は変化しません; site.py が仮想環境で使用されていると検出した場合、*prefix* および *exec_prefix* は仮想環境を示すよう変更され、*base_prefix* および *base_exec_prefix* は引き続き (仮想環境が作成された) ベースの Python インストール先を示します。

バージョン 3.3 で追加.

sys.byteorder
: プラットフォームのバイト順を示します。ビッグエンディアン (最上位バイトが先頭) のプラットフォームでは 'big', リトルエンディアン (最下位バイトが先頭) では 'little' となります。

sys.builtin_module_names
: コンパイル時に Python インタプリタに組み込まれた、全てのモジュール名のタプル (この情報は、他の手段では取得することができません。modules.keys() は、インポートされたモジュールのみのリストを返します。)

sys.call_tracing(*func*, *args*)
: トレーシングが有効な間、func(*args) を呼び出します。トレーシングの状態は保存され、後で復元されます。これは、別のコードをチェックポイントから再帰的にデバッグするために、デバッガから呼び出されることを意図しています。

sys.copyright
: Python インタプリタの著作権を表示する文字列です。

sys._clear_type_cache()
: 内部の型キャッシュをクリアします。型キャッシュは属性とメソッドの検索を高速化するために利用されます。この関数は、参照リークをデバッグするときに不要な参照を削除するため **だけ** に利用してください。

この関数は、内部的かつ特殊な目的にのみ利用されるべきです。

sys._current_frames()
: 各スレッドの識別子を関数が呼ばれた時点のそのスレッドでアクティブになっている一番上のスタックフレームに結びつける辞書を返します。モジュール *traceback* の関数を使えばそのように与えられたフレームのコールスタックを構築できます。

この関数はデッドロックをデバッグするのに非常に有効です。デッドロック状態のスレッドの協調動作を必要としませんし、そういったスレッドのコールスタックはデッドロックである限りフリーズしたままです。デッドロックにないスレッドのフレームについては、そのフレームを調べるコードを呼んだ時にはそのスレッドの現在の実行状況とは関係ないところを指し示しているかもしれません。

この関数は、内部的かつ特殊な目的にのみ利用されるべきです。

引数無しで **監査イベント** sys._current_frames を送出します。

sys.breakpointhook()
: このフック関数は組み込みの *breakpoint()* から呼ばれます。デフォルトでは、この関数は *pdb* デ

バッガに処理を移行させますが、他の関数を設定することもでき、使用するデバッガを選べます。

この関数のシグネチャは何を呼び出すかに依存します。例えば、(pdb.set_trace() などの) デフォルトの結び付けでは引数は求められませんが、追加の引数 (位置引数およびキーワード引数) を求める関数に結び付けるかもしれません。組み込みの breakpoint() 関数は、自身の *args と **kws をそのまま渡します。breakpointhooks() の返り値が何であれ、そのまま breakpoint() から返されます。

デフォルトの実装では、最初に環境変数 PYTHONBREAKPOINT を調べます。それが "0" に設定されていた場合は、この関数はすぐに終了します。つまり何もしません。この環境変数が設定されていないか、空文字列に設定されていた場合は、pdb.set_trace() が呼ばれます。それ以外の場合は、この変数は実行する関数の名前である必要があります。関数名の指定では、例えば package.subpackage.module.function のようなドットでつながれた Python のインポートの命名体系を使ったを使います。この場合では、package.subpackage.module がインポートされ、そのインポートされたモジュールには function() という呼び出し可能オブジェクトがなくてはなりません。この関数が渡された *args と **kws で実行され、function() の返り値が何であれ sys.breakpointhook() は組み込みの *breakpoint()* 関数にその値をそのまま返します。

PYTHONBREAKPOINT で指名された呼び出し可能オブジェクトのインポートで何かしら問題が起きた場合、*RuntimeWarning* が報告されブレークポイントは無視されることに注意してください。

また、sys.breakpointhook() がプログラム上で上書きされていた場合は PYTHONBREAKPOINT は **調べられない** ことにも注意してください。

バージョン 3.7 で追加.

sys._debugmallocstats()
CPython のメモリアロケータの状態に関する低レベルの情報を標準エラー出力に出力します。

Python が **configure** 時に --with-pydebug オプションを指定してビルドされている場合、負荷のかかる一貫性チェックも行います。

バージョン 3.3 で追加.

CPython implementation detail: この関数は CPython 固有です。正確な出力形式は定義されていませんし、変更されるかもしれません。

sys.dllhandle
Python DLL のハンドルを示す整数です。

利用可能な環境: Windows 。

sys.displayhook(*value*)
value が None 以外の時、repr(value) を sys.stdout に出力し、builtins._ に保存します。repr(value) がエラーハンドラを sys.stdout.errors (おそらく 'strict') として sys.stdout.encoding にエンコードできない場合、エラーハンドラを 'backslashreplace' として sys.stdout.encoding にエンコードします。

sys.displayhook は、Python の対話セッションで入力された **式** が評価されたときに呼び出されます。対話セッションの出力をカスタマイズする場合、sys.displayhook に引数の数が一つの関数を指定します。

擬似コード:

```
def displayhook(value):
    if value is None:
        return
    # Set '_' to None to avoid recursion
    builtins._ = None
    text = repr(value)
    try:
        sys.stdout.write(text)
    except UnicodeEncodeError:
        bytes = text.encode(sys.stdout.encoding, 'backslashreplace')
        if hasattr(sys.stdout, 'buffer'):
            sys.stdout.buffer.write(bytes)
        else:
            text = bytes.decode(sys.stdout.encoding, 'strict')
            sys.stdout.write(text)
    sys.stdout.write("\n")
    builtins._ = value
```

バージョン 3.2 で変更: *UnicodeEncodeError* には 'backslashreplace' エラーハンドラを指定してください。

sys.dont_write_bytecode

この値が真の時、Python はソースモジュールをインポートする時に .pyc ファイルを生成しません。この値は -B コマンドラインオプションと PYTHONDONTWRITEBYTECODE 環境変数の値によって、起動時に True か False に設定されますが、実行時にこの変数を変更してバイトコード生成を制御することもできます。

sys.pycache_prefix

If this is set (not None), Python will write bytecode-cache .pyc files to (and read them from) a parallel directory tree rooted at this directory, rather than from __pycache__ directories in the source code tree. Any __pycache__ directories in the source code tree will be ignored and new *.pyc* files written within the pycache prefix. Thus if you use *compileall* as a pre-build step, you must ensure you run it with the same pycache prefix (if any) that you will use at runtime.

A relative path is interpreted relative to the current working directory.

This value is initially set based on the value of the -X pycache_prefix=PATH command-line option or the PYTHONPYCACHEPREFIX environment variable (command-line takes precedence). If neither are set, it is None.

バージョン 3.8 で追加.

sys.excepthook(*type*, *value*, *traceback*)

指定したトレースバックと例外を sys.stderr に出力します。

例外が発生し、その例外が捕捉されない場合、インタプリタは例外クラス・例外インスタンス・トレースバックオブジェクトを引数として sys.excepthook を呼び出します。対話セッション中に発生した場合はプロンプトに戻る直前に呼び出され、Python プログラムの実行中に発生した場合はプログラ

ムの終了直前に呼び出されます。このトップレベルでの例外情報出力処理をカスタマイズする場合、sys.excepthook に引数の数が三つの関数を指定します。

引数 hook, type, value, traceback を指定して 監査イベント sys.excepthook を送出します。

参考:

The *sys.unraisablehook()* function handles unraisable exceptions and the *threading.excepthook()* function handles exception raised by *threading.Thread.run()*.

sys.__breakpointhook__
sys.__displayhook__
sys.__excepthook__
sys.__unraisablehook__

これらのオブジェクトはそれぞれ、プログラム起動時の breakpointhook, displayhook, excepthook, unraisablehook の値を保存しています。この値は、breakpointhook, displayhook, excepthook, unraisablehook に不正なオブジェクトや別のオブジェクトが指定された場合に、元の値に復旧するために使用します。

バージョン 3.7 で追加: ___breakpointhook___

バージョン 3.8 で追加: ___unraisablehook___

sys.exc_info()

現在処理中の例外を示す 3 個の値のタプルを返します。この値は、現在のスレッドおよび現在のスタックフレームのものです。現在のスタックフレームが例外処理中でない場合、例外処理中のスタックフレームが見つかるまでその呼び出したスタックフレームや呼び出し元などを調べます。ここで、”例外処理中” とは ”except 節を実行中である” フレームを指します。どのスタックフレームでも、最後に処理した例外の情報のみを参照できます。

If no exception is being handled anywhere on the stack, a tuple containing three None values is returned. Otherwise, the values returned are (type, value, traceback). Their meaning is: *type* gets the type of the exception being handled (a subclass of *BaseException*); *value* gets the exception instance (an instance of the exception type); *traceback* gets a traceback object which encapsulates the call stack at the point where the exception originally occurred.

sys.exec_prefix

Python のプラットフォーム依存なファイルがインストールされているディレクトリ名を示す文字列です。この値はサイト固有であり、デフォルトは '/usr/local' ですが、ビルド時に configure の --exec-prefix 引数で指定することができます。すべての設定ファイル (pyconfig.h など) は *exec_prefix*/lib/python*X.Y*/config に、共有ライブラリは *exec_prefix*/lib/python*X.Y*/lib-dynload にインストールされます (*X.Y* は 3.2 のような Python のバージョン番号)。

注釈: 仮想環境 で起動されている場合、この値は site.py によって仮想環境を示すよう変更されます。実際の Python のインストール先は *base_exec_prefix* から取得できます。

---

sys.executable

Python インタプリタの実行ファイルの絶対パスを示す文字列です。このような名前が意味を持つシステムで利用可能です。Python が自身の実行ファイルの実際のパスを取得できない場合、*sys.executable* は空の文字列または None になります。

sys.exit([*arg*])

Python を終了します。*exit()* は *SystemExit* を送出するので、try ステートメントの finally 節に終了処理を記述したり、上位レベルで例外を捕捉して exit 処理を中断したりすることができます。

オプション引数 *arg* には、終了ステータスとして整数 (デフォルトは 0) や他の型のオブジェクトを指定することができます。整数を指定した場合、シェル等は 0 は " 正常終了"、0 以外の整数を " 異常終了" として扱います。多くのシステムでは、有効な終了ステータスは 0--127 で、これ以外の値を返した場合の動作は未定義です。システムによっては特定の終了コードに個別の意味を持たせている場合がありますが、このような定義は僅かしかありません。Unix プログラムでは構文エラーの場合には 2 を、それ以外のエラーならば 1 を返します。*arg* に None を指定した場合は、数値の 0 を指定した場合と同じです。それ以外の型のオブジェクトを指定すると、そのオブェクトが *stderr* に出力され、終了コードとして 1 を返します。エラー発生時には sys.exit("**エラーメッセージ**") と書くと、簡単にプログラムを終了することができます。

究極には、*exit()* は例外を送出する " だけ" なので、これがメインスレッドから呼び出されたときは、プロセスを終了するだけで、例外は遮断されません。

バージョン 3.6 で変更: Python インタープリタが (バッファされたデータを標準ストリームに吐き出すときのエラーなどの) *SystemExit* を捕捉した後の後始末でエラーが起きた場合、終了ステータスは 120 に変更されます。

sys.flags

コマンドラインフラグの状態を表す **名前付きタプル** です。各属性は読み出し専用になっています。

属性	フラグ
debug	-d
inspect	-i
interactive	-i
isolated	-I
optimize	-O または -OO
dont_write_bytecode	-B
no_user_site	-s
no_site	-S
ignore_environment	-E
verbose	-v
bytes_warning	-b
quiet	-q
hash_randomization	-R
dev_mode	-X dev
utf8_mode	-X utf8
int_max_str_digits	-X int_max_str_digits (*integer string conversion length limitation*)

バージョン 3.2 で変更: 新しい -q 向けに quiet 属性が追加されました。

バージョン 3.2.3 で追加: hash_randomization 属性が追加されました。

バージョン 3.3 で変更: 廃止済みの division_warning 属性を削除しました。

バージョン 3.4 で変更: -I isolated フラグ向けに isolated 属性が追加されました。

バージョン 3.7 で変更: 新しい -X dev フラグ向けに dev_mode 属性が追加され、新しい -X utf8 フラグ向けに utf8_mode 属性が追加されました。

バージョン 3.8.14 で変更: Added the int_max_str_digits attribute.

sys.float_info

float 型に関する情報を保持している 名前付きタプル です。精度と内部表現に関する低レベル情報を含んでいます。これらの値は 'C' 言語の標準ヘッダファイル float.h で定義されている様々な浮動小数点定数に対応しています。詳細は 1999 ISO/IEC C standard [C99] の 5.2.4.2.2 節 'Characteristics of floating types' を参照してください。

属性	float.h のマクロ	説明
epsilon	DBL_EPSILON	difference between 1.0 and the least value greater than 1.0 that is representable as a float
dig	DBL_DIG	浮動小数点数で正確に表示できる最大の 10 進数桁; 以下参照
mant_dig	DBL_MANT_DIG	浮動小数点精度: 浮動小数点数の主要部の桁 base-radix
max	DBL_MAX	float が表せる最大の正の (infinite ではない) 値
max_exp	DBL_MAX_EXP	float が radix**(e-1) で表現可能な、最大の整数 *e*
max_10_exp	DBL_MAX_10_EXP	float が 10**e で表現可能な、最大の整数 *e*
min	DBL_MIN	minimum representable positive *normalized* float
min_exp	DBL_MIN_EXP	radix**(e-1) が正規化 float であるような最小の整数 *e*
min_10_exp	DBL_MIN_10_EXP	10**e が正規化 float であるような最小の整数 *e*
radix	FLT_RADIX	指数部の基数
rounds	FLT_ROUNDS	算術演算で利用される丸めモードを表す整数定数。これはインタプリタ起動時のシステムの FLT_ROUNDS マクロの値を示します。取りうる値とその意味については、C99 標準の 5.2.4.2.2 節を参照してください。

sys.float_info.dig に対してはさらに説明が必要です。もし、文字列 s が表す 10 進数の有効桁数が多くても sys.float_info.dig の時には、s を浮動小数点数に変換して戻すと同じ 10 進数を表す文字列に復元されます:

```
>>> import sys
>>> sys.float_info.dig
15
>>> s = '3.14159265358979'    # decimal string with 15 significant digits
```

(次のページに続く)

(前のページからの続き)

```
>>> format(float(s), '.15g')  # convert to float and back -> same value
'3.14159265358979'
```

ただし、文字列が有効桁数 sys.float_info.dig より大きい場合には、常に復元されるとは限りません:

```
>>> s = '9876543211234567'    # 16 significant digits is too many!
>>> format(float(s), '.16g')  # conversion changes value
'9876543211234568'
```

sys.float_repr_style

repr() 関数が浮動小数点数に対してどう振る舞うかを示す文字列です。この文字列が値 'short' を持てば、有限の浮動小数点数 x に対して、repr(x) は float(repr(x)) == x を満たす短い文字列を返そうとします。これは、Python 3.1 以降での標準の振る舞いです。それ以外の場合、float_repr_style は値 'legacy' を持ち、repr(x) は 3.1 以前のバージョンの Python と同じように振る舞います。

バージョン 3.1 で追加.

sys.getallocatedblocks()

サイズに関わりなく、現在インタプリタから確保されたメモリブロックの数を返します。この関数は主にメモリリークを追跡しバグを見付けるのに便利です。インタプリタの内部的なキャッシュがあるために、呼び出しごとに結果が変わることがあります。より結果が予測しやすくなる *_clear_type_cache()* や *gc.collect()* を呼ぶ必要があるかもしれません。

Python のあるビルドや実装が適切にこの情報を計算できない場合は、その代わりに *getallocatedblocks()* は 0 を返すことが許されています。

バージョン 3.4 で追加.

sys.getandroidapilevel()

ビルト時の Android のバージョンを整数で返します。

利用可能な環境: Android 。

バージョン 3.7 で追加.

sys.getcheckinterval()

インタプリタの ” チェックインターバル (check interval)” を返します; *setcheckinterval()* を参照してください。

バージョン 3.2 で非推奨: 代わりに *getswitchinterval()* を使ってください。

sys.getdefaultencoding()

Unicode 実装で使用される現在のデフォルトエンコーディング名を返します。

sys.getdlopenflags()

dlopen() の呼び出しで使われるフラグの現在の値を返します。フラグの値のシンボル名は *os* から見付けられます (RTLD_xxx 定数、例えば *os.RTLD_LAZY*)。

利用可能な環境: Unix。

sys.getfilesystemencoding()

Unicode ファイル名と bytes ファイル名どうしを変換するのに使われるエンコーディングの名前を返します。最良の互換性のために、たとえファイル名の表現に bytes がサポートされていたとしても、全てのケースで str をファイル名に使うべきです。ファイル名を受け取ったり返したりする関数は str と bytes をサポートし、すぐにシステムにとって好ましい表現に変換すべきです。

このエンコーディングは常に ASCII 互換です。

os.fsencode() や *os.fsdecode()* を、正しいエンコーディングやエラーモードが使われていることを保証するために使うべきです。

- UTF-8 モードでは、どのプラットフォームでもエンコーディングは `utf-8` です。
- macOS では、エンコーディングは `'utf-8'` となります。
- Unix では、エンコーディングはロケールのエンコーディングです。
- Windows では、エンコーディングはユーザの設定によって `'utf-8'` または `'mbcs'` のどちらかになります。
- Android では、エンコーディングは `'utf-8'` となります。
- VxWorks では、エンコーディングは `'utf-8'` となります。

バージョン 3.2 で変更: *getfilesystemencoding()* の結果が `None` になることはなくなりました。

バージョン 3.6 で変更: Windows はもう `'mbcs'` を返す保証は無くなりました。より詳しいことは **PEP 529** および *_enablelegacywindowsfsencoding()* を参照してください。

バージョン 3.7 で変更: UTF-8 モードで 'utf-8' を返すようになりました。

sys.getfilesystemencodeerrors()

Unicode ファイルシステムと bytes ファイル名どうしの変換で使われるエラーモード名を返します。エンコーディング名は *getfilesystemencoding()* から返されます。

os.fsencode() や *os.fsdecode()* を、正しいエンコーディングやエラーモードが使われていることを保証するために使うべきです。

バージョン 3.6 で追加.

sys.get_int_max_str_digits()

Returns the current value for the *integer string conversion length limitation*. See also *set_int_max_str_digits()*.

バージョン 3.8.14 で追加.

sys.getrefcount(*object*)

object の参照数を返します。*object* は (一時的に) *getrefcount()* からも参照されるため、参照数は予想される数よりも 1 多くなります。

sys.getrecursionlimit()

現在の最大再帰数を返します。最大再帰数は、Python インタプリタスタックの最大の深さです。この

制限は Python プログラムが無限に再帰し、C スタックがオーバーフローしてクラッシュすることを防止するために設けられています。この値は *setrecursionlimit()* で指定することができます。

sys.getsizeof(*object*[, *default*])

object のサイズをバイト数で返します。*object* は任意の型のオブジェクトです。すべての組み込みオブジェクトは正しい値を返します。サードパーティー製の型については実装依存になります。

オブジェクトに直接起因するメモリ消費のみを表し、参照するオブジェクトは含みません。

オブジェクトがサイズを取得する手段を提供していない時は *default* が返されます。*default* が指定されていない場合は *TypeError* が送出されます。

getsizeof() は *object* の `__sizeof__` メソッドを呼び出し、そのオブジェクトがガベージコレクタに管理されていた場合はガベージコレクタのオーバーヘッドを増やします。

getsizeof() を再帰的に使い、コンテナとその中身のサイズを割り出す例は、再帰的な sizeof のレシピ を参照してください。

sys.getswitchinterval()

インタプリタの " スレッド切り替え間隔" を返します。*setswitchinterval()* を参照してください。

バージョン 3.2 で追加.

sys._getframe([*depth*])

コールスタックからフレームオブジェクトを返します。オプション引数 *depth* を指定すると、スタックのトップから *depth* だけ下のフレームオブジェクトを取得します。*depth* がコールスタックよりも深ければ、*ValueError* が発生します。*depth* のデフォルト値は 0 で、この場合はコールスタックのトップのフレームを返します。

引数無しで **監査イベント** `sys._getframe` を送出します。

CPython implementation detail: この関数は、内部的かつ特殊な目的にのみ利用されるべきです。全ての Python 実装で存在することが保証されているわけではありません。

sys.getprofile()

setprofile() 関数で設定したプロファイラ関数を取得します。

sys.gettrace()

settrace() 関数で設定したトレース関数を取得します。

CPython implementation detail: *gettrace()* 関数は、デバッガ、プロファイラ、カバレッジツールなどの実装に使うことのみを想定しています。この関数の振る舞いは言語定義ではなく実装プラットフォームの一部です。そのため、他の Python 実装では利用できないかもしれません。

sys.getwindowsversion()

実行中の Windows バージョンを示す、名前付きタプルを返します。名前の付いている要素は、*major*, *minor*, *build*, *platform*, *service_pack*, *service_pack_minor*, *service_pack_major*, *suite_mask*, *product_type*, *platform_version* です。*service_pack* は文字列を含み、*platform_version* は 3-タプル、それ以外は整数です。この構成要素には名前でもアクセスできるので、`sys.getwindowsversion()[0]`

は sys.getwindowsversion().major と等価です。先行のバージョンとの互換性のため、最初の 5 要素のみが添字の指定で取得できます。

platform は 2 (VER_PLATFORM_WIN32_NT) になっているでしょう。

product_type は、以下の値のいずれかになります。:

定数	意味
1 (VER_NT_WORKSTATION)	システムはワークステーションです。
2 (VER_NT_DOMAIN_CONTROLLER)	システムはドメインコントローラです。
3 (VER_NT_SERVER)	システムはサーバですが、ドメインコントローラではありません。

この関数は、Win32 GetVersionEx() 関数を呼び出します。これらのフィールドに関する詳細は OSVERSIONINFOEX() についてのマイクロソフトのドキュメントを参照してください。

platform_version は、プロセスでエミュレートされているバージョンではなく、現在のオペレーティングシステムのメジャーバージョン、マイナーバージョン、ビルドナンバーを返します。この関数は機能の検知ではなくロギングで使うためのものです。

注釈: *platform_version* derives the version from kernel32.dll which can be of a different version than the OS version. Please use *platform* module for achieving accurate OS version.

利用可能な環境: Windows 。

バージョン 3.2 で変更: 名前付きタプルに変更され、*service_pack_minor*, *service_pack_major*, *suite_mask*, および *product_type* が追加されました。

バージョン 3.6 で変更: *platform_version* の追加

sys.get_asyncgen_hooks()

asyncgen_hooks オブジェクトを返します。このオブジェクトは *(firstiter, finalizer)* という形の *namedtuple* に似た形をしています。*firstiter* と *finalizer* は両方とも None か、*asynchronous generator iterator* を引数に取る関数と、イベントループが非同期ジェネレータの終了処理をスケジューリングするのに使う関数であることが求められます。

バージョン 3.6 で追加: より詳しくは **PEP 525** を参照してください。

注釈: This function has been added on a provisional basis (see **PEP 411** for details.)

---

sys.get_coroutine_origin_tracking_depth()

Get the current coroutine origin tracking depth, as set by *set_coroutine_origin_tracking_depth()*.

バージョン 3.7 で追加.

注釈: This function has been added on a provisional basis (see **PEP 411** for details.) Use it only for debugging purposes.

sys.hash_info

数値のハッシュ実装のパラメータを与える *named tuple* です。数値型のハッシュ化についての詳細は **数値型のハッシュ化** を参照してください。

属性	説明
width	ハッシュ値に利用されるビット幅
modulus	数値ハッシュ計算に利用される素数の法
inf	正の無限大に対して返されるハッシュ値
nan	nan に対して返されるハッシュ値
imag	複素数の虚部を表すための乗数
algorithm	*str*、*bytes*、*memoryview* オブジェクトのハッシュアルゴリズムの名前
hash_bits	ハッシュアルゴリズムの内部出力サイズ
seed_bits	ハッシュアルゴリズムのシードキーのサイズ

バージョン 3.2 で追加.

バージョン 3.4 で変更: *algorithm*、*hash_bits*、*seed_bits* を追加

sys.hexversion

単精度整数にエンコードされたバージョン番号。この値は新バージョン (正規リリース以外であっても) ごとにかならず増加します。例えば、Python 1.5.2 以降でのみ動作するプログラムでは、以下のようなチェックを行います。

```
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

hexversion は *hex()* 関数を使って 16 進数に変換しなければ値の意味を持ちません。より読みやすいバージョン番号が必要な場合には *named tuple* *sys.version_info* を使用してください。

hexversion のより詳しい情報については apiabiversion を参照してください。

sys.implementation

現在起動している Python インタプリタに関する情報が格納されたオブジェクトです。すべての Python 実装において、以下の属性が存在することが要求されています。

name は実装の識別子です (例: 'cpython')。実際の文字列は Python 実装によって定義されますが、小文字であることが保証されています。

version は *sys.version_info* と同じ形式の、Python **実装** のバージョンを表す名前付きタプルです。

これは現在起動しているインタプリタ固有のバージョンを意味します。sys.version_info は Python **言語** のバージョンであり、これはそのバージョンに準拠した **実装** のバージョンです。例えば、PyPy 1.8 の sys.implementation.version は sys.version_info(1, 8, 0, 'final', 0) になるのに対し、sys.version_info は sys.version_info(2, 7, 2, 'final', 0) になります。CPython は、それがリファレンス実装であるため、両者は同じ値になります。

hexversion は *sys.hexversion* と同形式の、16 進数での実装のバージョンです。

cache_tag は、キャッシュされたモジュールのファイル名に使用されるタグです。慣例により、これは実装名とバージョンを組み合わせた文字列になります (例: 'cpython-33') が、Python 実装はその他の適切な値を使う場合があります。cache_tag に None が設定された場合、モジュールのキャッシングが無効になっていることを示します。

sys.implementation には Python 実装固有の属性を追加することができます。それら非標準属性の名前はアンダースコアから始めなくてはならず、そしてここでは説明されません。その内容に関係なく、*sys.implementation* はインタプリタ起動中や実装のバージョン間で変更することはありません (ただし Python 言語のバージョン間についてはその限りではありません)。詳細は **PEP 421** を参照してください。

バージョン 3.3 で追加.

注釈: The addition of new required attributes must go through the normal PEP process. See **PEP 421** for more information.

---

sys.int_info

Python における整数の内部表現に関する情報を保持する、*named tuple* です。この属性は読み出し専用です。

属性	説明
bits_per_digit	各桁に保持されるビットの数です。Python の整数は、内部的に 2**int_info.bits_per_digit を基数として保存されます
sizeof_digit	桁を表すために使われる、C 型の大きさ (バイト) です
default_max_str_digits	default value for *sys.get_int_max_str_digits()* when it is not otherwise explicitly configured.
str_digits_check_threshold	minimum non-zero value for *sys.set_int_max_str_digits()*, PYTHONINTMAXSTRDIGITS, or -X int_max_str_digits.

バージョン 3.1 で追加.

バージョン 3.8.14 で変更: Added default_max_str_digits and str_digits_check_threshold.

sys.__interactivehook__

この属性が存在する場合、インタープリタが 対話モード で起動したときに値は (引数なしで) 自動的

に呼ばれます。これは PYTHONSTARTUP ファイルの読み込み後に行われるため、それにこのフックを設定することが出来ます。*site* モジュールで **設定します**。

引数 hook を指定して **監査イベント** cpython.run_interactivehook を送出します。

バージョン 3.4 で追加.

sys.**intern**(*string*)

string を ” 隔離” された文字列のテーブルに入力し、隔離された文字列を返します -- この文字列は *string* 自体かコピーです。隔離された文字列は辞書検索のパフォーマンスを少しだけ向上させるのに有効です -- 辞書中のキーが隔離されており、検索するキーが隔離されている場合、(ハッシュ化後の) キーの比較は文字列の比較ではなくポインタの比較で行うことができるからです。通常、Python プログラム内で利用されている名前は自動的に隔離され、モジュール、クラス、またはインスタンス属性を保持するための辞書は隔離されたキーを持っています。

隔離された文字列はイモータルではありません。その恩恵を受けるためには *intern()* の返り値への参照を維持しなくてはなりません。

sys.**is_finalizing**()

Return *True* if the Python interpreter is *shutting down*, *False* otherwise.

バージョン 3.5 で追加.

sys.**last_type**

sys.**last_value**

sys.**last_traceback**

通常は定義されておらず、捕捉されない例外が発生してインタプリタがエラーメッセージとトレースバックを出力した場合にのみ設定されます。この値は、対話セッション中にエラーが発生した時、デバッグモジュールをロード (例: import pdb; pdb.pm() など。詳細は *pdb* モジュールを参照) して発生したエラーを調査する場合に利用します。デバッガをロードすると、プログラムを再実行せずに情報を取得することができます

変数の意味は、上記 *exc_info()* の返り値と同じです。

sys.**maxsize**

Py_ssize_t 型の変数が取りうる最大値を示す整数です。通常、32 ビットプラットフォームでは 2**31 - 1、64 ビットプラットフォームでは 2**63 - 1 になります。

sys.**maxunicode**

Unicode コードポイントの最大値を示す整数、すなわち 1114111 (16 進数で 0x10FFFF) です。

バージョン 3.3 で変更: **PEP 393** 以前は、オプション設定に Unicode 文字の保存形式が UCS-2 と UCS-4 のどちらを指定したかによって、sys.maxunicode の値は 0xFFFF が 0x10FFFF になっていました。

sys.**meta_path**

A list of *meta path finder* objects that have their *find_spec()* methods called to see if one of the objects can find the module to be imported. The *find_spec()* method is called with at least the absolute name of the module being imported. If the module to be imported is contained in

a package, then the parent package's `__path__` attribute is passed in as a second argument. The method returns a *module spec*, or `None` if the module cannot be found.

参考:

importlib.abc.MetaPathFinder The abstract base class defining the interface of finder objects on *meta_path*.

importlib.machinery.ModuleSpec The concrete class which *find_spec()* should return instances of.

バージョン 3.4 で変更: *Module specs* were introduced in Python 3.4, by **PEP 451**. Earlier versions of Python looked for a method called *find_module()*. This is still called as a fallback if a *meta_path* entry doesn't have a *find_spec()* method.

`sys.modules`

既に読み込まれているモジュールとモジュール名がマップされている辞書です。これを使用してモジュールの強制再読み込みやその他のトリック操作が行えます。ただし、この辞書の置き換えは想定された操作ではなく、必要不可欠なアイテムを削除することで Python が異常終了してしまうかもしれません。

`sys.path`

モジュールを検索するパスを示す文字列のリスト。`PYTHONPATH` 環境変数と、インストール時に指定したデフォルトパスで初期化されます。

起動時に初期化された後、リストの先頭 (`path[0]`) には Python インタプリタを起動したスクリプトのあるディレクトリが挿入されます。スクリプトのディレクトリがない (インタプリタが対話セッションで起動された時や、スクリプトを標準入力から読み込んだ場合など) 場合、`path[0]` は空文字列となり、Python はカレントディレクトリからモジュールの検索を開始します。スクリプトディレクトリは、`PYTHONPATH` で指定したディレクトリの **前** に挿入されますので注意が必要です。

プログラムはその目的のために、このリストを自由に修正できます。文字列とバイト列だけが *sys.path* に追加でき、ほかの全てのデータ型はインポート中に無視されます。

参考:

site モジュールのドキュメントで、.pth ファイルを使って *sys.path* を拡張する方法を解説しています。

`sys.path_hooks`

path を引数にとって、その path に対する *finder* の作成を試みる呼び出し可能オブジェクトのリスト。finder の作成に成功したら、その呼出可能オブジェクトのは finder を返します。失敗した場合は、*ImportError* を発生させます。

オリジナルの仕様は **PEP 302** を参照してください。

`sys.path_importer_cache`

finder オブジェクトのキャッシュ機能を果たす辞書です。キーは *sys.path_hooks* に渡されている

パスで、値は見つかった finder になります。パスが正常なファイルシステムパスであり、finder が *sys.path_hooks* 上に見つからない場合は None が格納されます。

オリジナルの仕様は **PEP 302** を参照してください。

バージョン 3.3 で変更: finder が見つからない時、*imp.NullImporter* ではなく None が格納されるようになりました。

sys.platform

プラットフォームを識別する文字列で、*sys.path* にプラットフォーム固有のサブディレクトリを追加する場合などに利用します。

Linux と AIX を除く Unix システムでは、uname -s が返す小文字の OS 名に uname -r が返すバージョン名の先頭を追加したものになります (例: 'sunos5'、'freebsd8')。この値は *Python* **がビルドされた時点のもの** です。システム固有のバージョンをテストする場合を除き、以下のような用法で使うことを推奨します:

```
if sys.platform.startswith('freebsd'):
    # FreeBSD-specific code here...
elif sys.platform.startswith('linux'):
    # Linux-specific code here...
elif sys.platform.startswith('aix'):
    # AIX-specific code here...
```

その他のシステムでは以下の値になります:

システム	platform の値
AIX	'aix'
Linux	'linux'
Windows	'win32'
Windows/Cygwin	'cygwin'
macOS	'darwin'

バージョン 3.3 で変更: Linux では、*sys.platform* はもはやメジャーバージョン番号を含みません。'linux2' や 'linux3' の代わりに 'linux' が格納されます。古い Python ではこの変数はバージョン番号を含むため、前述した startswith イディオムを使用することをお勧めします。

バージョン 3.8 で変更: AIX では、*sys.platform* はもはやメジャーバージョン番号を含みません。'aix5' や 'aix7' の代わりに 'aix' が格納されます。古い Python ではこの変数はバージョン番号を含むため、前述した startswith イディオムを使用することをお勧めします。

参考:

os.name が持つ情報はおおざっぱな括りであり、*os.uname()* はシステムに依存したバージョン情報になります。

platform モジュールはシステムの詳細な識別情報をチェックする機能を提供しています。

sys.prefix

Python のプラットフォーム非依存なファイルがインストールされているディレクトリの接頭辞を示す文字列です。この値はサイト固有であり、デフォルトでは '/usr/local' ですが、ビルド時に configure スクリプトの --prefix 引数で指定する事ができます。Python ライブラリの主要部分は *prefix*/lib/python*X.Y* にインストールされ、プラットフォーム非依存なヘッダファイル (pyconfig.h 以外すべて) は *prefix*/include/python*X.Y* に格納されます (*X.Y* は 3.2 のような Python のバージョン番号)。

注釈: 仮想環境 で起動されている場合、この値は site.py によって仮想環境を示すよう変更されます。実際の Python のインストール先は *base_prefix* から取得できます。

---

sys.ps1

sys.ps2

インタプリタの一次プロンプト、二次プロンプトを指定する文字列。対話モードで実行中のみ定義され、初期値は '>>> ' と '... ' です。文字列以外のオブジェクトを指定した場合、インタプリタが対話コマンドを読み込むごとにオブジェクトの *str()* を評価します。この機能は、動的に変化するプロンプトを実装する場合に利用します。

sys.setcheckinterval(*interval*)

インタプリタの " チェック間隔" を示す整数値を指定します。この値はスレッドスイッチやシグナルハンドラのチェックを行う周期を決定します。デフォルト値は 100 で、この場合 100 の Python 仮想命令を実行するとチェックを行います。この値を大きくすればスレッドを利用するプログラムのパフォーマンスが向上します。この値が 0 **以下** の場合、各仮想命令を実行するたびにチェックを行い、レスポンス速度が最大になりますがオーバヘッドもまた最大となります。

バージョン 3.2 で非推奨: スレッドスイッチングと非同期タスクの内部ロジックが変更されたため、この関数はもはや効果がありません 。代わりに *setswitchinterval()* を使用してください。

sys.setdlopenflags(*n*)

インタプリタが拡張モジュールをロードする時、dlopen() で使用するフラグを設定します。sys.setdlopenflags(0) とすれば、モジュールインポート時にシンボルの遅延解決を行う事ができます。シンボルを拡張モジュール間で共有する場合には、sys.setdlopenflags(os.RTLD_GLOBAL) と指定します。フラグ値の定義名は *os* モジュールで定義されています (RTLD_xxx 定数、例えば *os.RTLD_LAZY*)。

利用可能な環境: Unix。

sys.set_int_max_str_digits(*n*)

Set the *integer string conversion length limitation* used by this interpreter. See also *get_int_max_str_digits()*.

バージョン 3.8.14 で追加.

sys.setprofile(*profilefunc*)

Set the system's profile function, which allows you to implement a Python source code profiler in Python. See chapter *Python* プロファイラ for more information on the Python profiler. The

system's profile function is called similarly to the system's trace function (see *settrace()*), but it is called with different events, for example it isn't called for each executed line of code (only on call and return, but the return event is reported even when an exception has been set). The function is thread-specific, but there is no way for the profiler to know about context switches between threads, so it does not make sense to use this in the presence of multiple threads. Also, its return value is not used, so it can simply return `None`. Error in the profile function will cause itself unset.

Profile 関数は 3 個の引数、*frame*、*event*、および *arg* を受け取る必要があります。*frame* は現在のスタックフレームです。*event* は文字列で、`'call'`, `'return'`, `'c_call'`, `'c_return'`, `'c_exception'` のどれかが渡されます。*arg* はイベントの種類によって異なります。

引数無しで **監査イベント** `sys.setprofile` を送出します。

event には以下の意味があります。

`'call'` 関数が呼び出された (もしくは、何かのコードブロックに入った)。プロファイル関数が呼ばれる。*arg* は `None` が渡される。

`'return'` 関数 (あるいは別のコードブロック) から戻ろうとしている。プロファイル関数が呼ばれる。*arg* は返されようとしている値、または、このイベントが例外が送出されることによって起こったなら `None` 。

`'c_call'` C 関数 (拡張関数かビルトイン関数) が呼ばれようとしている。*arg* は C 関数オブジェクト。

`'c_return'` C 関数から戻った。*arg* は C の関数オブジェクト。

`'c_exception'` C 関数が例外を発生させた。*arg* は C の関数オブジェクト。

`sys.setrecursionlimit(`*limit*`)`

Python インタプリタの、スタックの最大の深さを *limit* に設定します。この制限は Python プログラムが無限に再帰し、C スタックがオーバーフローしてクラッシュすることを防止するために設けられています。

limit の最大値はプラットフォームによって異なります。深い再帰処理が必要な場合にはプラットフォームがサポートしている範囲内でより大きな値を指定することができますが、この値が大きすぎればクラッシュするので注意が必要です。

If the new limit is too low at the current recursion depth, a *RecursionError* exception is raised.

バージョン 3.5.1 で変更: A *RecursionError* exception is now raised if the new limit is too low at the current recursion depth.

`sys.setswitchinterval(`*interval*`)`

インタプリタのスレッド切り替え間隔を秒で指定します。この浮動小数点値が、並列に動作している Python スレッドに割り当てられたタイムスライスの理想的な処理時間です。実際の値はより高くなることがあり、特に長時間実行される内部関数やメソッドが使われた時や、他にスレッドが、その間隔の終了をオペレーティングシステムが決定するようスケジュールされた場合などが該当します。インタプリタは自身のスケジューラを持ちません。

バージョン 3.2 で追加.

sys.settrace(*tracefunc*)

Set the system's trace function, which allows you to implement a Python source code debugger in Python. The function is thread-specific; for a debugger to support multiple threads, it must register a trace function using *settrace()* for each thread being debugged or use *threading.settrace()*.

Trace 関数は 3 個の引数、*frame*、*event*、および *arg* を受け取る必要があります。*frame* は現在のスタックフレームです。*event* は文字列で、'call', 'line', 'return', 'exception', 'opcode' のどれかが渡されます。*arg* はイベントの種類によって異なります。

trace 関数は (*event* に 'call' を渡された状態で) 新しいローカルスコープに入るたびに呼ばれます。この場合、新しいスコープで利用するローカルの trace 関数への参照か、そのスコープを trace しないのであれば None を返します。

ローカル trace 関数は自身への参照 (もしくはそのスコープの以降の trace を行う別の関数) を返すべきです。もしくは、そのスコープの trace を止めるために None を返します。

If there is any error occurred in the trace function, it will be unset, just like settrace(None) is called.

event には以下の意味があります。

'call' 関数が呼び出された (もしくは、何かのコードブロックに入った)。グローバルの trace 関数が呼ばれる。*arg* は None が渡される。戻り値はローカルの trace 関数。

'line' The interpreter is about to execute a new line of code or re-execute the condition of a loop. The local trace function is called; *arg* is None; the return value specifies the new local trace function. See Objects/lnotab_notes.txt for a detailed explanation of how this works. Per-line events may be disabled for a frame by setting f_trace_lines to *False* on that frame.

'return' 関数 (あるいは別のコードブロック) から戻ろうとしている。ローカルの trace 関数が呼ばれる。*arg* は返されようとしている値、または、このイベントが例外が送出されることによって起こったなら None 。trace 関数の戻り値は無視される。

'exception' 例外が発生した。ローカルの trace 関数が呼ばれる。*arg* は (exception, value, traceback) のタプル。戻り値は新しいローカルの trace 関数。

'opcode' The interpreter is about to execute a new opcode (see *dis* for opcode details). The local trace function is called; *arg* is None; the return value specifies the new local trace function. Per-opcode events are not emitted by default: they must be explicitly requested by setting f_trace_opcodes to *True* on the frame.

例外が呼び出しチェインを辿って伝播していくことに注意してください。'exception' イベントは各レベルで発生します。

For more fine-grained usage, it's possible to set a trace function by assigning frame.f_trace = tracefunc explicitly, rather than relying on it being set indirectly via the return value from an already installed trace function. This is also required for activating the trace function on

the current frame, which *settrace()* doesn't do. Note that in order for this to work, a global tracing function must have been installed with *settrace()* in order to enable the runtime tracing machinery, but it doesn't need to be the same tracing function (e.g. it could be a low overhead tracing function that simply returns `None` to disable itself immediately on each frame).

code と frame オブジェクトについては、types を参照してください。

引数無しで **監査イベント** `sys.settrace` を送出します。

CPython implementation detail: *settrace()* 関数は、デバッガ、プロファイラ、カバレッジツール等で使うためだけのものです。この関数の挙動は言語定義よりも実装プラットフォームの分野の問題で、全ての Python 実装で利用できるとは限りません。

バージョン 3.7 で変更: `'opcode'` event type added; `f_trace_lines` and `f_trace_opcodes` attributes added to frames

`sys.set_asyncgen_hooks`(*firstiter*, *finalizer*)

Accepts two optional keyword arguments which are callables that accept an *asynchronous generator iterator* as an argument. The *firstiter* callable will be called when an asynchronous generator is iterated for the first time. The *finalizer* will be called when an asynchronous generator is about to be garbage collected.

引数無しで **監査イベント** `sys.set_asyncgen_hooks_firstiter` を送出します。

引数無しで **監査イベント** `sys.set_asyncgen_hooks_finalizer` を送出します。

Two auditing events are raised because the underlying API consists of two calls, each of which must raise its own event.

バージョン 3.6 で追加: See **PEP 525** for more details, and for a reference example of a *finalizer* method see the implementation of `asyncio.Loop.shutdown_asyncgens` in Lib/asyncio/base_events.py

---

注釈: This function has been added on a provisional basis (see **PEP 411** for details.)

`sys.set_coroutine_origin_tracking_depth`(*depth*)

Allows enabling or disabling coroutine origin tracking. When enabled, the `cr_origin` attribute on coroutine objects will contain a tuple of (filename, line number, function name) tuples describing the traceback where the coroutine object was created, with the most recent call first. When disabled, `cr_origin` will be None.

To enable, pass a *depth* value greater than zero; this sets the number of frames whose information will be captured. To disable, pass set *depth* to zero.

This setting is thread-specific.

バージョン 3.7 で追加.

注釈: This function has been added on a provisional basis (see **PEP 411** for details.) Use it only for debugging purposes.

sys._enablelegacywindowsfsencoding()

Changes the default filesystem encoding and errors mode to 'mbcs' and 'replace' respectively, for consistency with versions of Python prior to 3.6.

This is equivalent to defining the PYTHONLEGACYWINDOWSFSENCODING environment variable before launching Python.

利用可能な環境: Windows 。

バージョン 3.6 で追加: より詳しくは **PEP 529** を参照してください。

sys.stdin
sys.stdout
sys.stderr

インタプリタが使用する、それぞれ標準入力、標準出力、および標準エラー出力の **ファイルオブジェクト** です:

- stdin は (*input()* の呼び出しも含む) すべての対話型入力に使われます。
- stdout は *print()* および *expression* 文の出力と、*input()* のプロンプトに使用されます。
- インタプリタ自身のプロンプトおよびエラーメッセージは stderr に出力されます。

これらのストリームは *open()* が返すような通常の **テキストファイル** です。引数は以下のように選択されます:

- The character encoding is platform-dependent. Non-Windows platforms use the locale encoding (see *locale.getpreferredencoding()*).

  On Windows, UTF-8 is used for the console device. Non-character devices such as disk files and pipes use the system locale encoding (i.e. the ANSI codepage). Non-console character devices such as NUL (i.e. where isatty() returns True) use the value of the console input and output codepages at startup, respectively for stdin and stdout/stderr. This defaults to the system locale encoding if the process is not initially attached to a console.

  The special behaviour of the console can be overridden by setting the environment variable PYTHONLEGACYWINDOWSSTDIO before starting Python. In that case, the console codepages are used as for any other character device.

  Under all platforms, you can override the character encoding by setting the PYTHONIOENCODING environment variable before starting Python or by using the new -X utf8 command line option and PYTHONUTF8 environment variable. However, for the Windows console, this only applies when PYTHONLEGACYWINDOWSSTDIO is also set.

- When interactive, stdout and stderr streams are line-buffered. Otherwise, they are

block-buffered like regular text files. You can override this value with the `-u` command-line option.

注釈: 標準ストリームにバイナリーデータの読み書きを行うには下位のバイナリー *buffer* オブジェクトを使用してください。例えば *bytes* を *stdout* に書き出す場合は `sys.stdout.buffer.write(b'abc')` を使用してください。

However, if you are writing a library (and do not control in which context its code will be executed), be aware that the standard streams may be replaced with file-like objects like *io.StringIO* which do not support the `buffer` attribute.

---

sys.__stdin__

sys.__stdout__

sys.__stderr__

それぞれ起動時の `stdin`, `stderr`, `stdout` の値を保存しています。終了処理時に利用されます。また、`sys.std*` オブジェクトが (訳注:別のファイルライクオブジェクトに) リダイレクトされている場合でも、実際の標準ストリームへの出力に利用できます。

また、標準ストリームが壊れたオブジェクトに置き換えられた場合に、動作する実際のファイルを復元するために利用することもできます。しかし、明示的に置き換え前のストリームを保存しておき、そのオブジェクトを復元する事を推奨します。

注釈: 一部の条件下では、`stdin`、`stdout`、および `stderr` もオリジナルの `__stdin__`、`__stdout__`、および `__stderr__` も `None` になることがあります。これはコンソールに接続しない Windows GUI アプリケーションであり、かつ **pythonw** で起動された Python アプリケーションが該当します。

---

sys.thread_info

スレッドの実装に関する情報が格納された *named tuple* です。

属性	説明
`name`	スレッド実装の名前: • `'nt'`: Windows スレッド • `'pthread'`: POSIX スレッド • `'solaris'`: Solaris スレッド
`lock`	ロック実装の名前: • `'semaphore'`: セマフォを使用するロック • `'mutex+cond'`: mutex と条件変数を使用するロック • `None` この情報が不明の場合
version	Name and version of the thread library. It is a string, or `None` if this information is unknown.

バージョン 3.3 で追加.

sys.tracebacklimit
: 捕捉されない例外が発生した時、出力されるトレースバック情報の最大レベル数を指定する整数値 (デフォルト値は 1000)。0 以下の値が設定された場合、トレースバック情報は出力されず例外型と例外値のみが出力されます。

sys.unraisablehook(*unraisable*, /)
: Handle an unraisable exception.

  Called when an exception has occurred but there is no way for Python to handle it. For example, when a destructor raises an exception or during garbage collection (*gc.collect()*).

  The *unraisable* argument has the following attributes:

  - *exc_type*: 例外の型
  - *exc_value*: 例外の値、"None"の可能性がある。
  - *exc_traceback*: Exception traceback, can be None.
  - *err_msg*: Error message, can be None.
  - *object*: Object causing the exception, can be None.

  The default hook formats *err_msg* and *object* as: `f'{err_msg}: {object!r}'`; use "Exception ignored in" error message if *err_msg* is None.

  sys.unraisablehook() can be overridden to control how unraisable exceptions are handled.

  Storing *exc_value* using a custom hook can create a reference cycle. It should be cleared explicitly to break the reference cycle when the exception is no longer needed.

  Storing *object* using a custom hook can resurrect it if it is set to an object which is being finalized. Avoid storing *object* after the custom hook completes to avoid resurrecting objects.

  See also *excepthook()* which handles uncaught exceptions.

  引数 hook, unraisable を指定して 監査イベント sys.unraisablehook を送出します。

  バージョン 3.8 で追加.

sys.version
: Python インタプリタのバージョン番号の他、ビルド番号や使用コンパイラなどの情報を示す文字列です。この文字列は Python 対話型インタプリタが起動した時に表示されます。バージョン情報はここから抜き出さずに、*version_info* および *platform* が提供する関数を使って下さい。

sys.api_version
: 使用中のインタプリタの C API バージョン。Python と拡張モジュール間の不整合をデバッグする場合などに利用できます。

sys.version_info
: バージョン番号を示す 5 要素タプル:*major*, *minor*, *micro*, *releaselevel*, *serial* 。*releaselevel* 以外は全

て整数です。*releaselevel* の値は、'alpha', 'beta', 'candidate', 'final' の何れかです。Python 2.0 の version_info は、(2, 0, 0, 'final', 0) となります。構成要素には名前でもアクセスできるので、sys.version_info[0] は sys.version_info.major と等価、などになります。

バージョン 3.1 で変更: 名前を使った要素アクセスがサポートされました。

sys.warnoptions
: この値は、warnings フレームワーク内部のみ使用され、変更することはできません。詳細は *warnings* を参照してください。

sys.winver
: Windows プラットフォームで、レジストリのキーとなるバージョン番号。Python DLL の文字列リソース 1000 に設定されています。通常、この値は *version* の先頭三文字となります。この値は参照専用で、別の値を設定しても Python が使用するレジストリキーを変更することはできません。

利用可能な環境: Windows 。

sys._xoptions
: コマンドラインオプション -X で渡された様々な実装固有のフラグの辞書です。オプション名にはその値が明示されていればそれが、それ以外の場合は *True* がマップされます。例えば:

```
$ ./python -Xa=b -Xc
Python 3.2a3+ (py3k, Oct 16 2010, 20:14:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys._xoptions
{'a': 'b', 'c': True}
```

CPython implementation detail: この -X によって渡されたオプションにアクセスする方法は CPython 固有です。他の実装ではそれらは他の意味でエクスポートされるか、あるいは何もしません。

バージョン 3.2 で追加.

出典

29.2 sysconfig --- Python の構成情報にアクセスする

バージョン 3.2 で追加.

ソースコード: Lib/sysconfig.py

---

sysconfig モジュールは、インストールパスのリストや、現在のプラットフォームに関連した構成などの、Python の構成情報 (configuration information) へのアクセスを提供します。

29.2.1 構成変数

Python の配布物は、Python 自体のバイナリや、*distutils* によってコンパイルされる外部の C 拡張をビルドするために必要な、Makefile と pyconfig.h ヘッダーファイルを含んでいます。

sysconfig はこれらのファイルに含まれるすべての変数を辞書に格納し、*get_config_vars()* や *get_config_var()* でアクセスできるようにします。

Windows では構成変数はだいぶ少なくなります。

sysconfig.**get_config_vars**(**args*)

引数がない場合、現在のプラットフォームに関するすべての構成変数の辞書を返します。

引数がある場合、各引数を構成変数辞書から検索した結果の変数のリストを返します。

各引数において、変数が見つからなかった場合は None が返されます。

sysconfig.**get_config_var**(*name*)

1 つの変数 *name* を返します。get_config_vars().get(name) と同じです。

name が見つからない場合、None を返します。

Example of usage:

```
>>> import sysconfig
>>> sysconfig.get_config_var('Py_ENABLE_SHARED')
0
>>> sysconfig.get_config_var('LIBDIR')
'/usr/local/lib'
>>> sysconfig.get_config_vars('AR', 'CXX')
['ar', 'g++']
```

29.2.2 インストールパス

Python はプラットフォームとインストールオプションによって、異なるインストールスキームを利用します。このスキームは、*os.name* の値に基づいてユニークな識別子で *sysconfig* に格納されます。

distutils やそれに基づいたシステムによって新しいコンポーネントをインストールするときは、同じスキームに従ってファイルを正しい場所にコピーします。

Python は現在 7 つのスキームをサポートしています:

- *posix_prefix*: Linux や Mac OS X などの POSIX プラットフォーム用のスキームです。これは Python やコンポーネントをインストールするときに使われるデフォルトのスキームです。
- *posix_home*: インストール時に *home* オプションが利用された場合における、POSIX プラットフォーム用のスキームです。このスキームはコンポーネントが Distutils に特定の home prefix を指定してインストールされたときに利用されます。

- *posix_user*: Distutils に *user* オプションを指定してコンポーネントをインストールするときに使われる、POSIX プラットフォーム用のスキームです。このスキームはユーザーのホームディレクトリ以下に配置されたパスヲ定義します。
- *nt*: Windows などの NT プラットフォーム用のスキームです。
- *nt_user*: *user* オプションが利用された場合の、NT プラットフォーム用のスキームです。

各スキームは、ユニークな識別子を持ったいくつかのパスの集合から成っています。現在 Python は 8 つのパスを利用します:

- *stdlib*: プラットフォーム非依存の、標準 Python ライブラリファイルを格納するディレクトリ.
- *platstdlib*: プラットフォーム依存の、標準 Python ライブラリファイルを格納するディレクトリ.
- *platlib*: プラットフォーム依存の、site ごとのファイルを格納するディレクトリ。
- *purelib*: プラットフォーム非依存の、site ごとのファイルを格納するディレクトリ。
- *include*: プラットフォーム非依存のヘッダーファイルを格納するディレクトリ。
- *platinclude*: プラットフォーム依存の、ヘッダーファイルを格納するディレクトリ。
- *scripts*: スクリプトファイルのためのディレクトリ。
- *data*: データファイルのためのディレクトリ。

sysconfig はこれらのパスを決定するためのいくつかの関数を提供しています。

sysconfig.**get_scheme_names**()
現在 *sysconfig* でサポートされているすべてのスキームを格納したタプルを返します。

sysconfig.**get_path_names**()
現在 *sysconfig* でサポートされているすべてのパス名を格納したタプルを返します。

sysconfig.**get_path**(*name*[, *scheme*[, *vars*[, *expand*]]])
scheme で指定されたインストールスキームから、パス *name* に従ってインストールパスを返します。

name は *get_path_names()* が返すリストに含まれる値でなければなりません。

sysconfig はインストールパスを、パス名、プラットフォーム、展開される変数に従って格納します。例えば、*nt* スキームでの *stdlib* パスは `{base}/Lib` になります。

get_path() はパスを展開するのに *get_config_vars()* が返す変数を利用します。すべての変数は各プラットフォームにおいてデフォルト値を持っていて、この関数を呼び出したときにデフォルト値を取得する場合があります。

scheme が指定された場合、*get_scheme_names()* が返すリストに含まれる値でなければなりません。指定されなかった場合は、現在のプラットフォームでのデフォルトスキームが利用されます。

vars が指定された場合、*get_config_vars()* が返す辞書をアップデートする変数辞書でなければなりません。

expand が `False` に設定された場合、パスは変数を使って展開されません。

name が見つからない場合、None を返します。

sysconfig.get_paths([*scheme*[, *vars*[, *expand*]]])
インストールスキームに基づいたすべてのインストールパスを格納した辞書を返します。詳しい情報は *get_path()* を参照してください。

scheme が指定されない場合、現在のプラットフォームでのデフォルトスキーマが使用されます。

vars を指定する場合、パスを展開するために使用される辞書を更新する値の辞書でなければなりません。

expand に偽を指定すると、パスは展開されません。

scheme が実在するスキームでなかった場合、*get_paths()* は *KeyError* を発生させます。

29.2.3 その他の関数

sysconfig.get_python_version()
MAJOR.MINOR の型の Python バージョン番号文字列を返します。'%d.%d' % sys.version_info[:2] に似ています。

sysconfig.get_platform()
現在のプラットフォームを識別するための文字列を返します。

これはプラットフォーム依存のビルドディレクトリやプラットフォーム依存の配布物を区別するために使われます。典型的には、('os.uname()' のように) OS の名前とバージョン、アーキテクチャを含みますが、厳密には OS に依存します。例えば Linux ではカーネルのバージョンはそれほど重要ではありません。

返される値の例:

- linux-i586
- linux-alpha (?)
- solaris-2.6-sun4u

Windows では以下のどれかを返します:

- win-amd64 (64bit Windows on AMD64, 別名 x86_64, Intel64, EM64T)
- win32 (その他すべて - 具体的には sys.platform が返す値)

Mac OS X では以下のどれかを返します:

- macosx-10.6-ppc
- macosx-10.4-ppc64
- macosx-10.3-i386
- macosx-10.4-fat

その他の非 POSIX プラットフォームでは、現在のところ単に *sys.platform* を返します。

sysconfig.**is_python_build**()
実行中の Python インタプリタがソースからビルドされ、かつそのビルドされた場所から実行されている場合に True を返します。make install を実行した場所からの実行やバイナリインストーラによるインストールではいけません。

sysconfig.**parse_config_h**(*fp*[, *vars*])
config.h スタイルのファイルを解析します。

fp は config.h スタイルのファイルを指すファイルライクオブジェクトです。

name/value ペアを格納した辞書を返します。第二引数にオプションの辞書が渡された場合、新しい辞書ではなくその辞書を利用し、ファイルから読み込んだ値で更新します。

sysconfig.**get_config_h_filename**()
pyconfig.h のパスを返します。

sysconfig.**get_makefile_filename**()
Makefile のパスを返します。

29.2.4 sysconfig をスクリプトとして使う

Python の *-m* オプションを使えば、*sysconfig* をスクリプトとして使用できます:

```
$ python -m sysconfig
Platform: "macosx-10.4-i386"
Python version: "3.2"
Current installation scheme: "posix_prefix"

Paths:
        data = "/usr/local"
        include = "/Users/tarek/Dev/svn.python.org/py3k/Include"
        platinclude = "."
        platlib = "/usr/local/lib/python3.2/site-packages"
        platstdlib = "/usr/local/lib/python3.2"
        purelib = "/usr/local/lib/python3.2/site-packages"
        scripts = "/usr/local/bin"
        stdlib = "/usr/local/lib/python3.2"

Variables:
        AC_APPLE_UNIVERSAL_BUILD = "0"
        AIX_GENUINE_CPLUSPLUS = "0"
        AR = "ar"
        ARFLAGS = "rc"
        ...
```

これは、*get_platform()*、*get_python_version()*、*get_path()* および *get_config_vars()* が返す情報を標準出力に出力します。

29.3 builtins --- 組み込みオブジェクト

このモジュールは Python の全ての「組み込み」識別子に直接アクセスするためのものです。例えば builtins.open は組み込み関数 *open()* の完全な名前です。ドキュメントは **組み込み関数** と **組み込み定数** を参照してください。

通常このモジュールはほとんどのアプリケーションで明示的にアクセスされることはありませんが、組み込みの値と同じ名前のオブジェクトを提供するモジュールが同時にその名前の組み込みオブジェクトも必要とするような場合には有用です。たとえば、組み込みの *open()* をラップした *open()* という関数を実装したいモジュールがあったとすると、このモジュールは次のように直接的に使われます:

```
import builtins

def open(path):
    f = builtins.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''Wrapper around a file that converts output to upper-case.'''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()

    # ...
```

ほとんどのモジュールではグローバル変数の一部として __builtins__ が利用できるようになっています。__builtins__ の内容は通常このモジュールそのものか、あるいはこのモジュールの *__dict__* 属性です。これは実装の詳細部分なので、異なる Python の実装では __builtins__ は使われていないこともあります。

29.4 __main__ --- トップレベルのスクリプト環境

'__main__' はトップレベルのコードが実行されるスコープの名前です。モジュールが、標準入力から読み込まれたとき、スクリプトとして実行されたとき、あるいはインタラクティブプロンプトのとき、___name___ には '__main__' が設定されます。

モジュールは、自身の __name__ をチェックすることでメインスコープで実行されているかどうかを確認できます。これはモジュールがスクリプトとして、あるいはインポートでなく python -m で起動されたときに実行するコードの条件として使用できる一般的なイディオムです:

```
if __name__ == "__main__":
    # execute only if run as a script
    main()
```

パッケージについては、__main__.py モジュールを用意することで同じ効果を得られます。__main__.py にモジュールが -m オプションで呼びだされたときに実行したいコードを書くことができます。

29.5 warnings --- 警告の制御

ソースコード: Lib/warnings.py

警告メッセージは一般に、ユーザに警告しておいた方がよいような状況下にプログラムが置かれているが、その状況は (通常は) 例外を送出したりそのプログラムを終了させるほどの正当な理由がないといった状況で発されます。例えば、プログラムが古いモジュールを使っている場合には警告を発したくなるかもしれません。

Python プログラマは、このモジュールの *warn()* 関数を使って警告を発することができます。(C 言語のプログラマは PyErr_WarnEx() を使います; 詳細は exceptionhandling を参照してください)。

警告メッセージは通常 *sys.stderr* に出力されますが、すべての警告を無視したり、警告を例外にしたりと、その処理を柔軟に変更することができます。警告の処理は warning category 、警告メッセージのテキスト、警告が発行されたソースの位置に基づいて変化します。同じソースの位置で特定の警告が繰り返された場合、通常は抑制されます。

警告制御には 2 つの段階 (stage) があります: 第一に、警告が発されるたびに、メッセージを出力すべきかどうかの決定が行われます; 次に、メッセージを出力するなら、メッセージはユーザによって設定が可能なフックを使って書式化され印字されます。

警告メッセージを出力するかどうかの決定は、警告フィルタ <warning-filter>‘によって制御されます。警告フィルタは一致規則 (matching rule) と動作からなるシーケンスです。:func:‘filterwarnings を呼び出して一致規則をフィルタに追加することができ、*resetwarnings()* を呼び出してフィルタを標準設定の状態にリセットすることができます。

警告メッセージの印字は *showwarning()* を呼び出して行うことができ、この関数は上書きすることができます; この関数の標準の実装では、*formatwarning()* を呼び出して警告メッセージを書式化しますが、この関数についても自作の実装を使うことができます。

参考:

logging.captureWarnings() を使うことで、標準ロギング基盤ですべての警告を扱うことができます。

29.5.1 警告カテゴリ

警告カテゴリを表現する組み込み例外は数多くあります。このカテゴリ化は警告をグループごとフィルタする上で便利です。

これらは厳密に言えば **組み込み例外** ですが、概念的には警告メカニズムに属しているのでここで記述されています。

標準の警告カテゴリをユーザの作成したコード上でサブクラス化することで、さらに別の警告カテゴリを定義することができます。警告カテゴリは常に *Warning* クラスのサブクラスでなければなりません。

現在以下の警告カテゴリクラスが定義されています:

Class	説明
Warning	すべての警告カテゴリクラスの基底クラスです。*Exception* のサブクラスです。
UserWarning	*warn()* の標準のカテゴリです。
DeprecationWarning	他の Python 開発者へ向けて警告を発するときの、廃止予定の機能についての警告の基底カテゴリです。(`__main__` によって引き起こされない限り通常は無視されます)
SyntaxWarning	その文法機能があいまいであることを示す警告カテゴリの基底クラスです。
RuntimeWarning	そのランタイム機能があいまいであることを示す警告カテゴリの基底クラスです。
FutureWarning	Python で書かれたアプリケーションのエンドユーザーへ向けて警告を発するときの、非推奨の機能についての警告の基底カテゴリです。
PendingDeprecationWarning	将来その機能が廃止されることを示す警告カテゴリの基底クラスです (デフォルトでは無視されます)。
ImportWarning	モジュールのインポート処理中に引き起こされる警告カテゴリの基底クラスです (デフォルトでは無視されます)。
UnicodeWarning	Unicode に関係した警告カテゴリの基底クラスです。
BytesWarning	*bytes* や *bytearray* に関連した警告カテゴリの基底クラスです。
ResourceWarning	リソースの使用に関連した警告カテゴリの基底クラスです。

バージョン 3.7 で変更: Previously *DeprecationWarning* and *FutureWarning* were distinguished based on whether a feature was being removed entirely or changing its behaviour. They are now distinguished based on their intended audience and the way they're handled by the default warnings filters.

29.5.2 警告フィルタ

警告フィルタは、ある警告を無視すべきか、表示すべきか、あるいは (例外を送出する) エラーにするべきかを制御します。

概念的には、警告フィルタは複数のフィルタ仕様からなる順番リストを維持しています; 何らかの特定の警告が生じると、一致するものが見つかるまでリスト中の各フィルタとの照合が行われます; 一致したフィルタ仕様がその警告の処理方法を決定します。フィルタの各エントリは (*action*, *message*, *category*, *module*, *lineno*) からなるタプルです。ここで:

- *action* は以下の文字列のうちの一つです:

値	処理方法
"default"	print the first occurrence of matching warnings for each location (module + line number) where the warning is issued
"error"	一致した警告を例外に変えます
"ignore"	一致した警告を出力しません
"always"	一致した警告を常に出力します
"module"	print the first occurrence of matching warnings for each module where the warning is issued (regardless of line number)
"once"	一致した警告のうち、警告の原因になった場所にかかわらず最初の警告のみ出力します

- *message* は正規表現を含む文字列で、警告メッセージの先頭はこのパターンに一致しなければなりません。正規表現は常に大小文字の区別をしないようにコンパイルされます。
- *category* はクラス (*Warning* のサブクラス) です。警告クラスはこのクラスのサブクラスに一致しなければなりません。
- *module* は正規表現を含む文字列で、モジュール名はこのパターンに一致しなければなりません。正規表現は常に大小文字の区別をしないようにコンパイルされます。
- *lineno* は整数で、警告が発生した場所の行番号に一致しなければなりません。0 の場合はすべての行と一致します。

Warning クラスは組み込みの *Exception* クラスから派生しているので、警告をエラーに変換するには単に `category(message)` を `raise` します。

If a warning is reported and doesn't match any registered filter then the "default" action is applied (hence its name).

Describing Warning Filters

The warnings filter is initialized by `-W` options passed to the Python interpreter command line and the `PYTHONWARNINGS` environment variable. The interpreter saves the arguments for all supplied entries without interpretation in *sys.warnoptions*; the *warnings* module parses these when it is first imported (invalid options are ignored, after printing a message to *sys.stderr*).

Individual warnings filters are specified as a sequence of fields separated by colons:

```
action:message:category:module:line
```

The meaning of each of these fields is as described in 警告フィルタ. When listing multiple filters on a single line (as for `PYTHONWARNINGS`), the individual filters are separated by commas and the filters listed later take precedence over those listed before them (as they're applied left-to-right, and the most recently applied filters take precedence over earlier ones).

Commonly used warning filters apply to either all warnings, warnings in a particular category, or warnings raised by particular modules or packages. Some examples:

```
default                      # Show all warnings (even those ignored by default)
ignore                       # Ignore all warnings
error                        # Convert all warnings to errors
error::ResourceWarning       # Treat ResourceWarning messages as errors
default::DeprecationWarning  # Show DeprecationWarning messages
ignore,default:::mymodule    # Only report warnings triggered by "mymodule"
error:::mymodule[.*]         # Convert warnings to errors in "mymodule"
                             # and any subpackages of "mymodule"
```

デフォルトの警告フィルタ

デフォルトで、Python はいくつかの警告フィルタをインストールします。これは `-W` コマンドラインオプション、`PYTHONWARNINGS` 環境変数または *filterwarnings()* の呼び出しでオーバーライドできます。

In regular release builds, the default warning filter has the following entries (in order of precedence):

```
default::DeprecationWarning:__main__
ignore::DeprecationWarning
ignore::PendingDeprecationWarning
ignore::ImportWarning
ignore::ResourceWarning
```

In debug builds, the list of default warning filters is empty.

バージョン 3.2 で変更: *PendingDeprecationWarning* に加えて、*DeprecationWarning* もデフォルトで無視されるようになりました。

バージョン 3.7 で変更: *DeprecationWarning* is once again shown by default when triggered directly by code in `__main__`.

バージョン 3.7 で変更: *BytesWarning* no longer appears in the default filter list and is instead configured via *sys.warnoptions* when `-b` is specified twice.

Overriding the default filter

Developers of applications written in Python may wish to hide *all* Python level warnings from their users by default, and only display them when running tests or otherwise working on the application. The *sys.warnoptions* attribute used to pass filter configurations to the interpreter can be used as a marker to indicate whether or not warnings should be disabled:

```
import sys

if not sys.warnoptions:
    import warnings
    warnings.simplefilter("ignore")
```

Developers of test runners for Python code are advised to instead ensure that *all* warnings are displayed by default for the code under test, using code like:

```
import sys

if not sys.warnoptions:
    import os, warnings
    warnings.simplefilter("default") # Change the filter in this process
    os.environ["PYTHONWARNINGS"] = "default" # Also affect subprocesses
```

Finally, developers of interactive shells that run user code in a namespace other than `__main__` are advised to ensure that *DeprecationWarning* messages are made visible by default, using code like the following (where `user_ns` is the module used to execute code entered interactively):

```
import warnings
warnings.filterwarnings("default", category=DeprecationWarning,
                                   module=user_ns.get("__name__"))
```

29.5.3 一時的に警告を抑制する

If you are using code that you know will raise a warning, such as a deprecated function, but do not want to see the warning (even when warnings have been explicitly configured via the command line), then it is possible to suppress the warning using the *catch_warnings* context manager:

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
```

このサンプルのコンテキストマネージャーの中では、すべての警告が無視されています。これで、他の廃止予定のコードを含まない (つもりの) 部分まで警告を抑止せずに、廃止予定だと分かっているコードだけ警告を表示させないようにすることができます。注意: これが保証できるのはシングルスレッドのアプリケーションだけです。2 つ以上のスレッドが同時に *catch_warnings* コンテキストマネージャーを使用した場合、動作は未定義です。

29.5.4 警告のテスト

コードが警告を発生させることをテストするには、*catch_warnings* コンテキストマネージャーを利用します。このクラスを使うと、一時的に警告フィルタを操作してテストに利用できます。例えば、次のコードでは、発生したすべての警告を取得してチェックしています:

```
import warnings

def fxn():
```

(次のページに続く)

(前のページからの続き)

```
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
    # Cause all warnings to always be triggered.
    warnings.simplefilter("always")
    # Trigger a warning.
    fxn()
    # Verify some things
    assert len(w) == 1
    assert issubclass(w[-1].category, DeprecationWarning)
    assert "deprecated" in str(w[-1].message)
```

always の代わりに error を利用することで、すべての警告で例外を発生させることができます。1 つ気をつけないといけないのは、一度 once/default ルールによって発生した警告は、フィルタに何をセットしているかにかかわらず、警告レジストリをクリアしない限りは 2 度と発生しません。

コンテキストマネージャーが終了したら、警告フィルタはコンテキストマネージャーに入る前のものに戻されます。これは、テスト中に予期しない方法で警告フィルタが変更され、テスト結果が中途半端になる事を予防します。このモジュールの *showwarning()* 関数も元の値に戻されます。注意: これが保証できるのはシングルスレッドのアプリケーションだけです。2 つ以上のスレッドが同時に *catch_warnings* コンテキストマネージャを使用した場合、動作は未定義です。

同じ種類の警告を発生させる複数の操作をテストする場合、各操作が新しい警告を発生させている事を確認するのは大切な事です (例えば、警告を例外として発生させて各操作が例外を発生させることを確認したり、警告リストの長さが各操作で増加していることを確認したり、警告リストを各操作の前に毎回クリアする事ができます)。

29.5.5 Updating Code For New Versions of Dependencies

Warning categories that are primarily of interest to Python developers (rather than end users of applications written in Python) are ignored by default.

Notably, this "ignored by default" list includes *DeprecationWarning* (for every module except `__main__`), which means developers should make sure to test their code with typically ignored warnings made visible in order to receive timely notifications of future breaking API changes (whether in the standard library or third party packages).

In the ideal case, the code will have a suitable test suite, and the test runner will take care of implicitly enabling all warnings when running tests (the test runner provided by the *unittest* module does this).

In less ideal cases, applications can be checked for use of deprecated interfaces by passing `-Wd` to the Python interpreter (this is shorthand for `-W default`) or setting `PYTHONWARNINGS=default` in the environment. This enables default handling for all warnings, including those that are ignored by default. To change what action is taken for encountered warnings you can change what argument is passed to `-W` (e.g. `-W error`). See the `-W` flag for more details on what is possible.

29.5.6 利用可能な関数

warnings.warn(*message*, *category=None*, *stacklevel=1*, *source=None*)

警告を発するか、無視するか、あるいは例外を送出します。*category* 引数が与えられた場合、警告カテゴリクラス <warning-categories>‘でなければなりません; 標準の値は :exc:‘UserWarning です。*message* を *Warning* インスタンスで代用することもできますが、この場合 *category* は無視され、`message.__class__` が使われ、メッセージ文は `str(message)` になります。発された例外が前述した **警告フィルタ** によってエラーに変更された場合、この関数は例外を送出します。引数 *stacklevel* は Python でラッパー関数を書く際に利用することができます。例えば:

```
def deprecation(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

こうすることで、警告が参照するソースコード部分を、`deprecation()` 自身ではなく `deprecation()` を呼び出した側にできます (というのも、前者の場合は警告メッセージの目的を台無しにしてしまうからです)。

source 引数が与えられた場合、これは *ResourceWarning* を発生させた破壊されたオブジェクトです。

バージョン 3.6 で変更: *source* 引数を追加しました。

warnings.warn_explicit(*message*, *category*, *filename*, *lineno*, *module=None*, *registry=None*, *module_globals=None*, *source=None*)

warn() の機能に対する低レベルのインタフェースで、メッセージ、警告カテゴリ、ファイル名および行番号、そしてオプションのモジュール名およびレジストリ情報 (モジュールの `__warningregistry__` 辞書) を明示的に渡します。モジュール名は標準で `.py` が取り去られたファイル名になります; レジストリが渡されなかった場合、警告が抑制されることはありません。*message* が文字列のとき、*category* は *Warning* のサブクラスでなければなりません。また *message* は *Warning* のインスタンスであってもよく、この場合 *category* は無視されます。

module_globals は、もし与えられるならば、警告が発せられるコードが使っているグローバル名前空間でなければなりません (この引数は zipfile やその他の非ファイルシステムのインポート元の中にあるモジュールのソースを表示することをサポートするためのものです)。

source 引数が与えられた場合、これは *ResourceWarning* を発生させた破壊されたオブジェクトです。

バージョン 3.6 で変更: *source* 引数を追加しました。

warnings.showwarning(*message*, *category*, *filename*, *lineno*, *file=None*, *line=None*)

警告をファイルに書き込みます。標準の実装では、`formatwarning(message, category, filename, lineno, line)` を呼び出し、返された文字列を *file* に書き込みます。*file* は標準では *sys.stderr* です。この関数は `warnings.showwarning` に任意の呼び出し可能オブジェクトを代入して置き換えることができます。*line* は警告メッセージに含めるソースコードの 1 行です。*line* が与えられない場合、*showwarning()* は *filename* と *lineno* から行を取得することを試みます。

warnings.formatwarning(*message*, *category*, *filename*, *lineno*, *line=None*)

警告を通常の方法で書式化します。返される文字列内には改行が埋め込まれている可能性があり、かつ文字列は改行で終端されています。*line* は警告メッセージに含まれるソースコードの 1 行です。*line* が

渡されない場合、*formatwarning()* は *filename* と *lineno* から行の取得を試みます。

warnings.filterwarnings(*action*, *message=''*, *category=Warning*, *module=''*, *lineno=0*, *append=False*)

警告フィルタ仕様 のリストにエントリを一つ挿入します。標準ではエントリは先頭に挿入されます; *append* が真ならば、末尾に挿入されます。この関数は引数の型をチェックし、*message* および *module* の正規表現をコンパイルしてから、これらをタプルにして警告フィルタのリストに挿入します。二つのエントリが特定の警告に合致した場合、リストの先頭に近い方のエントリが後方にあるエントリに優先します。引数が省略されると、標準ではすべてにマッチする値に設定されます。

warnings.simplefilter(*action*, *category=Warning*, *lineno=0*, *append=False*)

単純なエントリを **警告フィルタ仕様** のリストに挿入します。引数の意味は *filterwarnings()* と同じですが、この関数により挿入されるフィルタはカテゴリと行番号が一致していればすべてのモジュールのすべてのメッセージに合致しますので、正規表現は必要ありません。

warnings.resetwarnings()

警告フィルタをリセットします。これにより、`-W` コマンドラインオプションによるもの *simplefilter()* 呼び出しによるものを含め、*filterwarnings()* の呼び出しによる影響はすべて無効化されます。

29.5.7 利用可能なコンテキストマネージャー

class warnings.catch_warnings(***, *record=False*, *module=None*)

警告フィルタと *showwarning()* 関数をコピーし、終了時に復元するコンテキストマネージャーです。*record* 引数が *False* (デフォルト値) だった場合、コンテキスト開始時には *None* を返します。もし *record* が *True* だった場合、リストを返します。このリストにはカスタムの *showwarning()* 関数 (この関数は同時に `sys.stdout` への出力を抑制します) によってオブジェクトが継続的に追加されます。リストの中の各オブジェクトは、*showwarning()* 関数の引数と同じ名前の属性を持っています。

module 引数は、保護したいフィルタを持つモジュールを取ります。*warnings* を import して得られるモジュールの代わりに利用されます。この引数は、主に *warnings* モジュール自体をテストする目的で追加されました。

注釈: *catch_warnings* マネージャーは、モジュールの *showwarning()* 関数と内部のフィルタ仕様のリストを置き換え、その後復元することによって動作しています。これは、コンテキストマネージャーがグローバルな状態を変更していることを意味していて、したがってスレッドセーフではありません。

29.6 dataclasses --- データクラス

ソースコード: Lib/dataclasses.py

このモジュールは、`__init__()` や `__repr__()` のような *special method* を生成し、ユーザー定義のクラスに自動的に追加するデコレータや関数を提供します。このモジュールは **PEP 557** に記載されました。

これらの生成されたメソッドで利用されるメンバー変数は **PEP 526** 型アノテーションを用いて定義されます。例えば、このコードでは:

```
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """Class for keeping track of an item in inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

とりわけ、以下のような `__init__()` が追加されます:

```
def __init__(self, name: str, unit_price: float, quantity_on_hand: int = 0):
    self.name = name
    self.unit_price = unit_price
    self.quantity_on_hand = quantity_on_hand
```

このメソッドは自動的にクラスに追加される点に留意して下さい。上記の `InventoryItem` クラスの定義中にこのメソッドが直接明記されるわけではありません。

バージョン 3.7 で追加.

29.6.1 モジュールレベルのデコレータ、クラス、関数

@dataclasses.dataclass(**, init=True, repr=True, eq=True, order=False, unsafe_hash=False, frozen=False*)

この関数は、後述する *special method* を生成し、クラスに追加する *decorator* です。

dataclass() デコレータは、**フィールド** を探すためにクラスを検査します。**フィールド** は **型アノテーション** を持つクラス変数として定義されます。後述する 2 つの例外を除き、*dataclass()* は変数アノテーションで指定した型を検査しません。

生成されるすべてのメソッドの中でのフィールドの順序は、それらのフィールドがクラス定義に現れた順序です。

dataclass() デコレータは、後述する様々な " ダンダー" メソッド (訳注:dunder は double underscore の略で、メソッド名の前後にアンダースコアが 2 つ付いているメソッド) をクラスに追加します。クラ

スに既にこれらのメソッドが存在する場合の動作は、後述する引数によって異なります。デコレータは呼び出した際に指定したクラスと同じクラスを返します。新しいクラスは生成されません。

dataclass() が引数を指定しない単純なデコレータとして使用された場合、ドキュメントに記載されているシグネチャのデフォルト値のとおりに動作します。つまり、以下の 3 つの *dataclass()* の用例は同等です:

```
@dataclass
class C:
    ...

@dataclass()
class C:
    ...

@dataclass(init=True, repr=True, eq=True, order=False, unsafe_hash=False, frozen=False)
class C:
   ...
```

dataclass() の引数は以下の通りです:

- `init`: (デフォルトの) 真の場合、`__init__()` メソッドが生成されます。

  もしクラスに `__init__()` が既に定義されていた場合は、この引数は無視されます。

- `repr`: (デフォルトの) 真の場合、`__repr__()` メソッドが生成されます。生成された repr 文字列には、クラス名、各フィールドの名前および repr 文字列が、クラス上での定義された順序で並びます。repr から除外するように印が付けられたフィールドは、repr 文字列には含まれません。例えば、このようになります: `InventoryItem(name='widget', unit_price=3.0, quantity_on_hand=10)`。

  もしクラスに `__repr__()` が既に定義されていた場合は、この引数は無視されます。

- `eq`: (デフォルトの) 真の場合、`__eq__()` メソッドが生成されます。このメソッドはクラスの比較を、そのクラスのフィールドからなるタプルを比較するように行います。比較する 2 つのインスタンスのクラスは同一でなければなりません。

  もしクラスに `__eq__()` が既に定義されていた場合は、この引数は無視されます。

- `order`: 真 (デフォルト値は `False`) の場合、`__lt__()`、`__le__()`、`__gt__()`、`__ge__()` メソッドが生成されます。これらの比較は、クラスをそのフィールドからなるタプルであるかのように取り扱います。比較される 2 つのインスタンスは、同一の型でなければなりません。もし `order` が true で、`eq` に false を指定すすると、*ValueError* が送出されます。

  もし、クラスで既に `__lt__()`, `__le__()`, `__gt__()`, `__ge__()` のうちいずれかが定義されていると *TypeError* が送出されます。

- `unsafe_hash`: (デフォルトの) `False` の場合、`eq` と `frozen` がどう設定されているかに従って `__hash__()` メソッドが生成されます。

  `__hash__()` は、組み込みの *hash()* から使われたり、dict や set のようなハッシュ化されたコレ

クションにオブジェクトを追加するときに使われます。__hash__() があるということはそのクラスのインスタンスが不変 (イミュータブル) であることを意味します。可変性というのは複雑な性質で、プログラマの意図、__eq__() が存在しているかどうかとその振る舞い、*dataclass()* デコレータの eq フラグと frozen フラグの値に依存します。

デフォルトでは、*dataclass()* は追加しても安全でない限り __hash__() メソッドを暗黙的には追加しません。また、明示的に定義され存在している __hash__() メソッドに追加したり変更したりはしません。クラスの属性の __hash__ = None という設定は、Python にとって __hash__() のドキュメントにあるような特別な意味があります。

__hash__() が明示的に定義されていなかったり、None に設定されていた場合は、*dataclass()* は暗黙的に __hash__() メソッドを追加する **かもしれません** 。推奨はできませんが、unsafe_hash=True とすることで *dataclass()* に __hash__() メソッドを作成させられます。こうしてしまうと、クラスが論理的には不変だがそれにもかかわらず変更できてしまう場合、問題になり得ます。こうするのは特別なユースケースで、慎重に検討するべきです。

__hash__() メソッドが暗黙的に作られるかどうかを決定する規則は次の通りです。データクラスに明示的な __hash__() メソッドを持たせた上で、unsafe_hash=True と設定することはできません; こうすると *TypeError* になります。

eq と frozen が両方とも真だった場合、デフォルトでは *dataclass()* は __hash__() メソッドを生成します。eq が真で frozen が偽の場合、__hash__() は None に設定され、(可変なので) ハッシュ化不可能とされます。eq が偽の場合は、__hash__() は手を付けないまま、つまりスーパークラスの __hash__() メソッドが使われることになります (スーパークラスが *object* だった場合は、id に基づいたハッシュ化にフォールバックするということになります)。

- frozen: 真 (デフォルト値は False) の場合、フィールドへの代入は例外を生成します。これにより読み出し専用の凍結されたインスタンスを模倣します。__setattr__() あるいは __delattr__() がクラスに定義されていた場合は、*TypeError* が送出されます。後にある議論を参照してください。

フィールド には、通常の Python の文法でデフォルト値を指定できます。

```
@dataclass
class C:
    a: int       # 'a' has no default value
    b: int = 0   # assign a default value for 'b'
```

この例では、生成された __init__() メソッドには a と b の両方が含まれ、以下のように定義されます:

```
def __init__(self, a: int, b: int = 0):
```

デフォルト値を指定しないフィールドを、デフォルト値を指定したフィールドの後ろに定義すると、*TypeError* が送出されます。これは、単一のクラスであっても、クラス継承の結果でも起きえます。

dataclasses.**field**(**, default=MISSING, default_factory=MISSING, repr=True, hash=None, init=True, compare=True, metadata=None*)

通常の単純なユースケースでは、この他の機能は必要ありません。しかし、データクラスには、フィール

ドごとの情報を必要とする機能もあります。追加の情報の必要性に応えるために、デフォルトのフィールドの値をモジュールから提供されている *field()* 関数の呼び出しに置き換えられます。例えば次のようになります:

```
@dataclass
class C:
    mylist: List[int] = field(default_factory=list)

c = C()
c.mylist += [1, 2, 3]
```

上にあるように、MISSING 値は default パラメータと default_factory パラメータが提供されたかどうかを検出するのに使われる番兵オブジェクトです。この番兵が使われるのは、None が default の有効な値だからです。どんなコードでも MISSING 値を直接使うべきではありません。

field() の引数は次の通りです:

- default: 与えられた場合、このフィールドのデフォルト値になります。これが必要なのは、*field()* の呼び出しそのものが通常ではデフォルト値がいる位置を横取りしているからです。
- default_factory: 提供されていた場合、0 引数の呼び出し可能オブジェクトでなければならず、このフィールドの初期値が必要になったときに呼び出されます。他の目的も含めて、下で議論されているように、フィールドに可変なデフォルト値を指定するのに使えます。default と default_factory の両方を指定するとエラーになります。
- init: (デフォルトの) 真の場合、生成される __init__() メソッドの引数にこのフィールドを含めます。
- repr: (デフォルトの) 真の場合、生成される __repr__() メソッドによって返される文字列に、このフィールドを含めます。
- compare: (デフォルトの) 真の場合、生成される等価関数と比較関数 (__eq__() 、__gt__() など) にこのフィールドを含めます。
- hash: これは真偽値あるいは None に設定できます。真の場合、このフィールドは、生成された __hash__() メソッドに含まれます。(デフォルトの) None の場合、compare の値を使います: こうすることは普通は期待通りの振る舞いになります。比較で使われるフィールドはハッシュに含まれるものと考えるべきです。この値を None 以外に設定することは推奨されません。

  フィールドのハッシュ値を計算するコストが高い場合に、hash=False だが compare=True と設定する理由が 1 つあるとすれば、フィールドが等価検査に必要かつ、その型のハッシュ値を計算するのに他のフィールドも使われることです。フィールドがハッシュから除外されていたとしても、比較には使えます。
- metadata: これはマッピングあるいは None に設定できます。None は空の辞書として扱われます。この値は *MappingProxyType()* でラップされ、読み出し専用になり、*Field* オブジェクトに公開されます。これはデータクラスから使われることはなく、サードパーティーの拡張機構として提供されます。複数のサードパーティーが各々のキーを持て、メタデータの名前空間として使えます。

field() の呼び出しでフィールドのデフォルト値が指定されている場合は、このフィールドのクラス

属性は、その指定された default 値で置き換えられます。default が提供されていない場合は、そのクラス属性は削除されます。こうする意図は、*dataclass()* デコレータが実行された後には、ちょうどデフォルト値そのものが指定されたかのように、クラス属性がデフォルト値を全て持っているようにすることです。例えば、次のような場合:

```
@dataclass
class C:
    x: int
    y: int = field(repr=False)
    z: int = field(repr=False, default=10)
    t: int = 20
```

クラス属性 C.z は 10 、クラス属性 C.t は 20 になり、クラス属性 C.x と C.y には値が設定されません。

class dataclasses.**Field**

Field オブジェクトはそれぞれの定義されたフィールドを記述します。このオブジェクトは内部で作られ、モジュールレベル関数の *fields()* によって返されます (下の解説を見てください)。ユーザーは絶対に *Field* オブジェクトを直接インスタンス化すべきではありません。ドキュメント化されている属性は次の通りです:

- name: フィールド名
- type: フィールドの型
- default, default_factory, init, repr, hash, compare, metadata は *field()* の宣言と同じ意味と値を持ちます。

他の属性があることもありますが、それらはプライベートであり、調べたり、依存したりしてはなりません。

dataclasses.**fields**(*class_or_instance*)

このデータクラスのフィールドを定義する *Field* オブジェクトをタプルで返します。データクラスあるいはデータクラスのインスタンスを受け付けます。データクラスやデータクラスのインスタンスが渡されなかった場合は、*TypeError* を送出します。ClassVar や InitVar といった疑似フィールドは返しません。

dataclasses.**asdict**(*instance*, ***, *dict_factory=dict*)

データクラスの instance を (ファクトリ関数 dict_factory を使い) 辞書に変換します。それぞれのデータクラスは、name: value という組になっている、フィールドの辞書に変換されます。データクラス、辞書、リスト、タプルは再帰的に処理されます。例えば、次のようになります:

```
@dataclass
class Point:
     x: int
     y: int

@dataclass
class C:
```

(次のページに続く)

(前のページからの続き)

```
    mylist: List[Point]

p = Point(10, 20)
assert asdict(p) == {'x': 10, 'y': 20}

c = C([Point(0, 0), Point(10, 4)])
assert asdict(c) == {'mylist': [{'x': 0, 'y': 0}, {'x': 10, 'y': 4}]}
```

instance がデータクラスのインスタンスでなかった場合、*TypeError* を送出します。

dataclasses.**astuple**(*instance*, ***, *tuple_factory=tuple*)

データクラスの instance を (ファクトリ関数 tuple_factory を使い) タプルに変換します。それぞれのデータクラスは、フィールドの値のタプルに変換されます。データクラス、辞書、リスト、タプルは再帰的に処理されます。

1 つ前の例の続きです:

```
assert astuple(p) == (10, 20)
assert astuple(c) == ([(0, 0), (10, 4)],)
```

instance がデータクラスのインスタンスでなかった場合、*TypeError* を送出します。

dataclasses.**make_dataclass**(*cls_name*, *fields*, ***, *bases=()*, *namespace=None*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*)

cls_name という名前、fields で定義されるフィールド、bases で与えられた基底クラス、namespace で与えられた名前空間付きで初期化されたデータクラスを作成します。fields はイテラブルで、要素が name, (name, type), (name, type, Field) のうちのどれかです。単に name だけが与えられた場合は、typing.Any が type として使われます。init, repr, eq, order, unsafe_hash, frozen の値は、*dataclass()* のときと同じ意味を持ちます。

厳密にはこの関数は必須ではありません。というのは、__annotations__ 付きのクラスを新しく作成するどの Python の機構でも、*dataclass()* 関数を適用してそのクラスをデータクラスに変換できるからです。この関数は便利さのために提供されています。例えば次のように使います:

```
C = make_dataclass('C',
                   [('x', int),
                     'y',
                    ('z', int, field(default=5))],
                   namespace={'add_one': lambda self: self.x + 1})
```

は、次のコードと等しいです:

```
@dataclass
class C:
    x: int
    y: 'typing.Any'
    z: int = 5
```

(次のページに続く)

(前のページからの続き)

```
    def add_one(self):
        return self.x + 1
```

dataclasses.**replace**(*instance*, ***changes*)

instance と同じ型のオブジェクトを新しく作成し、フィールドを changes にある値で置き換えます。instance がデータクラスではなかった場合、*TypeError* を送出します。changes にある値がフィールドを指定していなかった場合も、*TypeError* を送出します。

新しく返されるオブジェクトは、データクラスの __init__() メソッドを呼び出して作成されます。これにより、もしあれば __post_init__() も呼び出されることが保証されます。

初期化限定変数でデフォルト値を持たないものがもしあれば、*replace()* の呼び出し時に初期値が指定され、__init__() と __post_init__() に渡せるようにしなければなりません。

changes に、init=False と定義されたフィールドが含まれているとエラーになります。この場合 *ValueError* が送出されます。

replace() を呼び出しているときに init=False であるフィールドがどのように働くかに気を付けてください。そのフィールドは元のオブジェクトからコピーされるのではなく、仮に初期化されたとしても結局は __post_init__() で初期化されます。init=False であるフィールドは滅多に使いませんし、使うとしたら注意深く使用します。そのようなフィールドが使われている場合は、代わりのクラスコンストラクタ、あるいは、インスタンスのコピー処理をする独自実装の replace() (もしくは似た名前の) メソッドを持たせるのが賢明でしょう。

dataclasses.**is_dataclass**(*class_or_instance*)

引数がデータクラスかデータクラスのインスタンスだった場合に True を返します。それ以外の場合は False を返します。

引数がデータクラスのインスタンスである (そして、データクラスそのものではない) かどうかを知る必要がある場合は、not isinstance(obj, type) で追加のチェックをしてください:

```
def is_dataclass_instance(obj):
    return is_dataclass(obj) and not isinstance(obj, type)
```

29.6.2 初期化後の処理

生成された __init__() のコードは、__post_init__() という名前のメソッドがクラスに定義されていたら、それを呼び出します。通常は self.__post_init__() のように呼び出されます。しかし InitVar フィールドが定義されていた場合、それらもクラスに定義された順序で __post_init__() に渡されます。__init__() メソッドが生成されなかった場合は、__post_init__() は自動的には呼び出されません。

他の機能と組み合わせることで、他の 1 つ以上のフィールドに依存しているフィールドが初期化できます。例えば次のようにできます:

```
@dataclass
class C:
    a: float
    b: float
    c: float = field(init=False)

    def __post_init__(self):
        self.c = self.a + self.b
```

下にある初期化限定変数についての節で、__post_init__() にパラメータを渡す方法を参照してください。*replace()* が init=False であるフィールドをどう取り扱うかについての警告も参照してください。

29.6.3 クラス変数

dataclass() が実際にフィールドの型の検査を行う 2 箇所のうち 1 つは、フィールドが **PEP 526** で定義されたクラス変数かどうかの判定です。その判定はフィールドの型が typing.ClassVar かどうかで行います。フィールドが ClassVar の場合、フィールドとは見なされなくなり、データクラスの機構からは無視されます。そのような ClassVar 疑似フィールドは、モジュールレベル関数 *fields()* の返り値には含まれません。

29.6.4 初期化限定変数

dataclass() が型アノテーションの検査を行うもう 1 つの箇所は、フィールドが初期化限定変数かどうかの判定です。その判定はフィールドの型が dataclasses.InitVar 型であるかどうかで行います。フィールドが InitVar の場合、初期化限定フィールドと呼ばれる疑似フィールドと見なされます。これは本物のフィールドではないので、モジュールレベル関数 *fields()* の返り値には含まれません。初期化限定フィールドは生成された __init__() メソッドに引数として追加され、オプションの __post_init__() メソッドにも渡されます。初期化限定フィールドは、データクラスからはそれ以外では使われません。

例えば、あるフィールドがデータベースから初期化されると仮定して、クラスを作成するときには値が与えられない次の場合を考えます:

```
@dataclass
class C:
    i: int
    j: int = None
    database: InitVar[DatabaseType] = None

    def __post_init__(self, database):
        if self.j is None and database is not None:
            self.j = database.lookup('j')

c = C(10, database=my_database)
```

このケースでは、*fields()* は i と j の *Field* オブジェクトは返しますが、database の *Field* オブジェクトは返しません。

29.6.5 凍結されたインスタンス

真に不変な Python のオブジェクトを作成するのは不可能です。しかし、frozen=True を *dataclass()* デコレータに渡すことで、不変性の模倣はできます。このケースでは、データクラスは __setattr__() メソッドと __delattr__() メソッドをクラスに追加します。これらのメソッドは起動すると *FrozenInstanceError* を送出します。

frozen=True を使うとき、実行する上でのわずかな代償があります: __init__() でフィールドを初期化するのに単純に割り当てることはできず、object.__setattr__() を使わなくてはなりません。

29.6.6 継承

データクラスが *dataclass()* デコレータで作成されるとき、MRO を逆向きに (すなわち、*object* を出発点として) 全ての基底クラスを調べていき、見付かったデータクラスそれぞれについて、その基底クラスが持っているフィールドを順序付きマッピングオブジェクトに追加します。全ての基底クラスのフィールドが追加し終わったら、自分自身のフィールドを順序付きマッピングオブジェクトに追加します。生成された全てのメソッドは、このフィールドが集められ整列された順序付きのマッピングオブジェクトを利用します。フィールドは挿入順序で並んでいるので、派生クラスは基底クラスをオーバーライドします。例えば次のようになります:

```
@dataclass
class Base:
    x: Any = 15.0
    y: int = 0

@dataclass
class C(Base):
    z: int = 10
    x: int = 15
```

最終的に出来上がるフィールドのリストは x, y, z の順番になります。最終的な x の型は、クラス C で指定されている通り int です。

C の生成された __init__() メソッドは次のようになります:

```
def __init__(self, x: int = 15, y: int = 0, z: int = 10):
```

29.6.7 デフォルトファクトリ関数

field() に default_factory を指定した場合、そのフィールドのデフォルト値が必要とされたときに、引数無しで呼び出されます。これは例えば、リストの新しいインスタンスを作成するために使います:

```
mylist: list = field(default_factory=list)
```

あるフィールドが (init=False を使って) __init__() から除外され、かつ、default_factory が

指定されていた場合、デフォルトファクトリ関数は生成された `__init__()` 関数から常に呼び出されます。フィールドに初期値を与える方法が他に無いので、このような動きになります。

29.6.8 可変なデフォルト値

Python はメンバ変数のデフォルト値をクラス属性に保持します。データクラスを使っていない、この例を考えてみましょう:

```
class C:
    x = []
    def add(self, element):
        self.x.append(element)

o1 = C()
o2 = C()
o1.add(1)
o2.add(2)
assert o1.x == [1, 2]
assert o1.x is o2.x
```

クラス `C` の 2 つのインスタンスが、予想通り同じクラス変数 `x` を共有していることに注意してください。

データクラスを使っているこのコードが **もし仮に** 有効なものだとしたら:

```
@dataclass
class D:
    x: List = []
    def add(self, element):
        self.x += element
```

データクラスは次のようなコードを生成するでしょう:

```
class D:
    x = []
    def __init__(self, x=x):
        self.x = x
    def add(self, element):
        self.x += element

assert D().x is D().x
```

これには、クラス `C` を使ったさっきの例と同じ問題があります。すなわち、クラス `D` の 2 つのインスタンスは、クラスインスタンスを作成するときに `x` の具体的な値を指定しておらず、同じ `x` のコピーを共有します。データクラスは Python の通常のクラス作成の仕組みを使っているだけなので、この同じ問題を抱えています。データクラスがこの問題を検出する一般的な方法を持たない代わりに、データクラスは型が `list` や `dict` や `set` のデフォルトパラメーターを検出した場合、*TypeError* を送出します。これは完全ではない解決法ですが、よくあるエラーの多くを防げます。

デフォルトファクトリ関数を使うのが、フィールドのデフォルト値として可変な型の新しいインスタンスを作成する手段です:

```
@dataclass
class D:
    x: list = field(default_factory=list)

assert D().x is not D().x
```

29.6.9 例外

exception dataclasses.FrozenInstanceError

frozen=True 付きで定義されたデータクラスで、暗黙的に定義された __setattr__() または __delattr__() が呼び出されたときに送出されます。これは *AttributeError* のサブクラスです。

29.7 contextlib --- with 文コンテキスト用ユーティリティ

ソースコード: Lib/contextlib.py

このモジュールは with 文に関わる一般的なタスクのためのユーティリティを提供します。詳しい情報は、コンテキストマネージャ型 と context-managers を参照してください。

29.7.1 ユーティリティ

以下の関数とクラスを提供しています:

class contextlib.AbstractContextManager

object.__enter__() と object.__exit__() の 2 つのメソッドを実装した抽象基底クラス (*abstract base class*) です。object.__enter__() は self を返すデフォルトの実装が提供されるいっぽう、object.__exit__() はデフォルトで None を返す抽象メソッドです。コンテキストマネージャ型 の定義も参照してください。

バージョン 3.6 で追加.

class contextlib.AbstractAsyncContextManager

object.__aenter__() と object.__aexit__() の 2 つのメソッドを実装するクラスのための抽象基底クラス (*abstract base class*) です。object.__aenter__() は self を返すデフォルト実装が提供されるいっぽう、object.__aexit__() はデフォルトで None を返す抽象メソッドです。async-context-managers の定義も参照してください。

バージョン 3.7 で追加.

@contextlib.contextmanager

この関数は with 文コンテキストマネージャのファクトリ関数を定義するために利用できる デコレー

タ です。新しいクラスや `__enter__()` と `__exit__()` メソッドを別々に定義しなくても、ファクトリ関数を定義することができます。

多くのオブジェクトが with 文の仕様を固有にサポートしていますが、コンテキストマネージャの権限に属さず、`close()` メソッドを実装していないために `contextlib.closing` の利用もできないリソースを管理する必要があることがあります。

リソースを正しく管理するよう保証する抽象的な例は以下のようなものでしょう:

```
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwds):
    # Code to acquire resource, e.g.:
    resource = acquire_resource(*args, **kwds)
    try:
        yield resource
    finally:
        # Code to release resource, e.g.:
        release_resource(resource)

>>> with managed_resource(timeout=3600) as resource:
...     # Resource is released at the end of this block,
...     # even if code in the block raises an exception
```

デコレート対象の関数は呼び出されたときに **ジェネレータ**-イテレータを返す必要があります。このイテレータは必ず値を 1 つ yield しなければなりません。`with` 文の `as` 節が存在するなら、その値は as 節のターゲットへ束縛されることになります。

ジェネレータが yield を実行した箇所で `with` 文のネストされたブロックが実行されます。ブロックから抜けた後でジェネレータは再開されます。ブロック内で処理されない例外が発生した場合は、ジェネレータ内部の yield を実行した箇所で例外が再送出されます。なので、(もしあれば) エラーを捕捉したり、クリーンアップ処理を確実に実行したりするために、`try...except...finally` 構文を使用できます。例外を捕捉する目的が、(完全に例外を抑制してしまうのではなく) 単に例外のログをとるため、もしくはあるアクションを実行するためなら、ジェネレータはその例外を再送出しなければなりません。例外を再送出しない場合、ジェネレータのコンテキストマネージャは `with` 文に対して例外が処理されたことを示し、`with` 文の直後の文から実行を再開します。

contextmanager() は *ContextDecorator* を使っているので、*contextmanager()* で作ったコンテキストマネージャは `with` 文だけでなくデコレータとしても利用できます。デコレーターとして利用された場合、新しい generator インスタンスが関数呼び出しのたびに暗黙に生成されます (このことによって、*contextmanager()* によって作られたなにがしか「単発」コンテキストマネージャを、コンテキストマネージャがデコレータとして使われるためには多重に呼び出されることをサポートする必要がある、という要件に合致させることが出来ます。)

バージョン 3.2 で変更: *ContextDecorator* の使用。

@contextlib.asynccontextmanager

contextmanager() と似ていますが、非同期コンテキストマネージャ (asynchronous context manager) を生成します。

この関数は async with 文のための非同期コンテキストマネージャのファクトリ関数を定義するために利用できるデコレータ (*decorator*) です。新しいクラスや __aenter__() と __aexit__() メソッドを個別に定義する必要はありません。このデコレータは非同期ジェネレータ (*asynchronous generator*) 関数に適用しなければなりません。

簡単な例:

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def get_connection():
    conn = await acquire_db_connection()
    try:
        yield conn
    finally:
        await release_db_connection(conn)

async def get_all_users():
    async with get_connection() as conn:
        return conn.query('SELECT ...')
```

バージョン 3.7 で追加.

contextlib.**closing**(*thing*)

ブロックの完了時に *thing* を close するコンテキストマネージャを返します。これは基本的に以下と等価です:

```
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

そして、明示的に page を close する必要なしに、次のように書くことができます:

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('http://www.python.org')) as page:
    for line in page:
        print(line)
```

page を明示的に close する必要は無く、エラーが発生した場合でも、with ブロックを出るときに page.close() が呼ばれます。

contextlib.**nullcontext**(*enter_result=None*)

enter_result を __enter__ メソッドが返すだけで、その他は何もしないコンテキストマネージャを返します。たとえば以下のように、ある機能を持った別のコンテキストマネージャに対する、選択可能な

代役として使われることが意図されています:

```
def myfunction(arg, ignore_exceptions=False):
    if ignore_exceptions:
        # Use suppress to ignore all exceptions.
        cm = contextlib.suppress(Exception)
    else:
        # Do not ignore any exceptions, cm has no effect.
        cm = contextlib.nullcontext()
    with cm:
        # Do something
```

enter_result を使った例です:

```
def process_file(file_or_path):
    if isinstance(file_or_path, str):
        # If string, open file
        cm = open(file_or_path)
    else:
        # Caller is responsible for closing file
        cm = nullcontext(file_or_path)

    with cm as file:
        # Perform processing on the file
```

バージョン 3.7 で追加.

contextlib.**suppress**(**exceptions*)

任意の例外リストを受け取り、with ブロック内でいずれかが起こると with ブロックの直後から黙って実行を再開するコンテキストマネージャを返します。

ほかの完全に例外を抑制するメカニズム同様、このコンテキストマネージャは、黙ってプログラム実行を続けることが正しいことであるとわかっている、非常に限定的なエラーをカバーする以上の使い方はしてはいけません。

例えば:

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove('somefile.tmp')

with suppress(FileNotFoundError):
    os.remove('someotherfile.tmp')
```

これは以下と等価です:

```
try:
    os.remove('somefile.tmp')
except FileNotFoundError:
    pass
```

(次のページに続く)

(前のページからの続き)

```
try:
    os.remove('someotherfile.tmp')
except FileNotFoundError:
    pass
```

このコンテキストマネージャは **再入可能** (*リエントラント*) です。

バージョン 3.4 で追加.

contextlib.**redirect_stdout**(*new_target*)

sys.stdout を一時的に別のファイルまたは file-like オブジェクトにリダイレクトするコンテキストマネージャです。

このツールは、出力先が標準出力 (stdout) に固定されている既存の関数やクラスに出力先の柔軟性を追加します。

For example, the output of *help()* normally is sent to *sys.stdout*. You can capture that output in a string by redirecting the output to an *io.StringIO* object:

```
f = io.StringIO()
with redirect_stdout(f):
    help(pow)
s = f.getvalue()
```

help() の出力をディスク上のファイルに送るためには、出力を通常のファイルにリダイレクトします:

```
with open('help.txt', 'w') as f:
    with redirect_stdout(f):
        help(pow)
```

help() の出力を標準エラー出力 (*sys.stderr*) に送るには以下のようにします:

```
with redirect_stdout(sys.stderr):
    help(pow)
```

sys.stdout のシステム全体にわたる副作用により、このコンテキストマネージャはライブラリコードやマルチスレッドアプリケーションでの使用には適していません。また、サブプロセスの出力に対しても効果がありません。そのような制限はありますが、それでも多くのユーティリティスクリプトに対して有用なアプローチです。

このコンテキストマネージャは **再入可能** (*リエントラント*) です。

バージョン 3.4 で追加.

contextlib.**redirect_stderr**(*new_target*)

redirect_stdout() と同じですが、標準エラー出力 (*sys.stderr*) を別のファイルや file-like オブジェクトにリダイレクトします。

このコンテキストマネージャは **再入可能** (*リエントラント*) です。

バージョン 3.5 で追加.

class contextlib.ContextDecorator

コンテキストマネージャをデコレータとしても使用できるようにする基底クラスです。

ContextDecorator から継承したコンテキストマネージャは、通常のコンテキストマネージャーと同じく __enter__ および __exit__ を実装する必要があります。__exit__ はデコレータとして使用された場合でも例外をオプションの引数として受け取ります。

contextmanager() は ContextDecorator を利用しているので、自動的にデコレーターとしても利用できるようになります。

ContextDecorator の例:

```
from contextlib import ContextDecorator

class mycontext(ContextDecorator):
    def __enter__(self):
        print('Starting')
        return self

    def __exit__(self, *exc):
        print('Finishing')
        return False

>>> @mycontext()
... def function():
...     print('The bit in the middle')
...
>>> function()
Starting
The bit in the middle
Finishing

>>> with mycontext():
...     print('The bit in the middle')
...
Starting
The bit in the middle
Finishing
```

これは次のような形のコードに対するシンタックスシュガーになります:

```
def f():
    with cm():
        # Do stuff
```

ContextDecorator を使うと代わりに次のように書けます:

```
@cm()
def f():
    # Do stuff
```

デコレーターを使うと、cm が関数の一部ではなく全体に適用されていることが明確になります (インデントレベルを 1 つ節約できるのもメリットです)。

すでに基底クラスを持っているコンテキストマネージャーも、ContextDecorator を mixin クラスとして利用することで拡張できます:

```
from contextlib import ContextDecorator

class mycontext(ContextBaseClass, ContextDecorator):
    def __enter__(self):
        return self

    def __exit__(self, *exc):
        return False
```

注釈: デコレートされた関数が複数回呼び出せるように、内部のコンテキストマネージャーは複数の with 文に対応する必要があります。そうでないなら、明示的な with 文を関数内で利用するべきです。

バージョン 3.2 で追加.

class contextlib.**ExitStack**

他の、特にオプションであったり入力に依存するようなコンテキストマネージャーやクリーンアップ関数を動的に組み合わせるためのコンテキストマネージャーです。

例えば、複数のファイルを 1 つの with 文で簡単に扱うことができます:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # All opened files will automatically be closed at the end of
    # the with statement, even if attempts to open files later
    # in the list raise an exception
```

各インスタンスは登録されたコールバックのスタックを管理し、インスタンスが (明示的に、あるいは with 文の終わりに暗黙的に) close されるときに逆順でそれを呼び出します。コンテキストスタックのインスタンスが暗黙的にガベージコレクトされたときには callback は呼び出され **ません** 。

このスタックモデルは、(file オブジェクトのように) __init__ メソッドでリソースを確保するコンテキストマネージャーを正しく扱うためのものです。

登録されたコールバックが登録の逆順で実行されるので、複数のネストされた with 文を利用するのと同じ振る舞いをします。これは例外処理にも適用されます。内側のコールバックが例外を抑制したり置き換えたりした場合、外側のコールバックには更新された状態に応じた引数が渡されます。

これは正しく exit callback の stack を巻き戻すための、比較的低レベルな API です。アプリケーション独自のより高レベルなコンテキストマネージャーを作るための基板として使うのに適しています。

バージョン 3.3 で追加.

enter_context(*cm*)

新しいコンテキストマネージャーに enter し、その `__exit__()` method をコールバックスタックに追加します。渡されたコンテキストマネージャーの `__enter__()` メソッドの戻り値を返します。

コンテキストマネージャーは、普段 `with` 文で利用された時と同じように、例外を抑制することができます。

`push`(*exit*)

コンテキストマネージャーの `__exit__()` メソッドをコールバックスタックに追加します。

このメソッドは `__enter__` を **呼び出さない** ので、コンテキストマネージャーを実装するときに、`__enter__()` の実装の一部を自身の `__exit__()` メソッドでカバーするために利用できます。

コンテキストマネージャーではないオブジェクトが渡された場合、このメソッドはそのオブジェクトをコンテキストマネージャーの `__exit__()` メソッドと同じシグネチャを持つコールバック関数だと仮定して、直接コールバックスタックに追加します。

それらのコールバック関数も、コンテキストマネージャーの `__exit__()` と同じく、true 値を返すことで例外を抑制することができます。

この関数はデコレータとしても使えるように、受け取ったオブジェクトをそのまま返します。

`callback`(*callback*, **args*, ***kwds*)

任意の関数と引数を受け取り、コールバックスタックに追加します。

他のメソッドと異なり、このメソッドで追加されたコールバックは例外を抑制しません (例外の詳細も渡されません)。

この関数はデコレータとしても使えるように、受け取った callback をそのまま返します。

`pop_all`()

コールバックスタックを新しい *ExitStack* インスタンスに移して、それを返します。このメソッドは callback を実行しません。代わりに、新しい stack が (明示的に、あるいは `with` 文の終わりに暗黙的に) close されるときに実行されます。

例えば、複数のファイルを "all or nothing" に開く処理を次のように書けます:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # Hold onto the close method, but don't call it yet.
    close_files = stack.pop_all().close
    # If opening any file fails, all previously opened files will be
    # closed automatically. If all files are opened successfully,
    # they will remain open even after the with statement ends.
    # close_files() can then be invoked explicitly to close them all.
```

`close`()

すぐにコールバックスタックを巻き戻し、コールバック関数を登録の逆順に呼び出します。登録されたすべてのコンテキストマネージャーと終了 callback に、例外が起こらなかった場合の引数が渡されます。

class contextlib.AsyncExitStack

ExitStack に似た 非同期コンテキストマネージャ です。スタック上で同期と非同期の両方のコンテキストマネージャの組み合わせをサポートします。また、後処理のためのコルーチンも持っています。

close() メソッドは実装されていません。代わりに *aclose()* を使ってください。

enter_async_context(*cm*)

enter_context() と同様のメソッドですが、非同期コンテキストマネージャを受け取ります。

push_async_exit(*exit*)

push() と同様のメソッドですが、非同期コンテキストマネージャかコルーチン関数を受け取ります。

push_async_callback(*callback*, **args*, ***kwds*)

callback() と同様のメソッドですが、コルーチン関数を受け取ります。

aclose()

close() と同様のメソッドですが、待ち受け可能オブジェクト (awaitables) を適切に処理します。

asynccontextmanager() の使用例の続きです:

```
async with AsyncExitStack() as stack:
    connections = [await stack.enter_async_context(get_connection())
        for i in range(5)]
    # All opened connections will automatically be released at the end of
    # the async with statement, even if attempts to open a connection
    # later in the list raise an exception.
```

バージョン 3.7 で追加.

29.7.2 例とレシピ

このセクションでは、*contextlib* が提供するツールの効果的な使い方を示す例とレシピを紹介します。

可変数個のコンテキストマネージャーをサポートする

ExitStack の第一のユースケースは、クラスのドキュメントにかかれている通り、一つの with 文で可変数個のコンテキストマネージャーや他のクリーンアップ関数をサポートすることです。ユーザーの入力 (指定された複数個のファイルを開く場合など) に応じて複数個のコンテキストマネージャーが必要となる場合や、いくつかのコンテキストマネージャーがオプションとなる場合に、可変数個のコンテキストマネージャーが必要になります:

```
with ExitStack() as stack:
    for resource in resources:
        stack.enter_context(resource)
    if need_special_resource():
        special = acquire_special_resource()
        stack.callback(release_special_resource, special)
    # Perform operations that use the acquired resources
```

上の例にあるように、*ExitStack* はコンテキストマネージャープロトコルをサポートしていないリソースの管理を with 文を使って簡単に行えるようにします。

__enter__ メソッドからの例外をキャッチする

稀に、__enter__ メソッドからの例外を、with 文の body やコンテキストマネージャーの __exit__ メソッドからの例外は間違えて捕まえないように、catch したい場合があります。*ExitStack* を使って、コンテキストマネージャープロトコル内のステップを分離することができます:

```
stack = ExitStack()
try:
    x = stack.enter_context(cm)
except Exception:
    # handle __enter__ exception
else:
    with stack:
        # Handle normal case
```

実際のところ、このようなコードが必要になるのならば、利用している API 側で try/except/finally 文を使った直接的なリソース管理インタフェースを提供するべきです。しかし、すべての API がそのようによく設計されているとは限りません。もしコンテキストマネージャーが提供されている唯一のリソース管理 API であるなら、*ExitStack* を使って with 文を使って処理することができない様々なシチュエーションの処理をすることができます。

__enter__ 実装内のクリーンアップ

ExitStack.push() のドキュメントで言及したとおり、このメソッドはすでに獲得したリソースを、__enter__() メソッドの残りのステップが失敗した時にクリーンアップするために利用することができます。

次の例では、リソースの確保と開放の関数に加えて、オプションのバリデーション関数を受け取るコンテキストマネージャーで、この方法を使ってコンテキストマネージャープロトコルを提供しています:

```
from contextlib import contextmanager, AbstractContextManager, ExitStack

class ResourceManager(AbstractContextManager):

    def __init__(self, acquire_resource, release_resource, check_resource_ok=None):
        self.acquire_resource = acquire_resource
        self.release_resource = release_resource
        if check_resource_ok is None:
            def check_resource_ok(resource):
                return True
        self.check_resource_ok = check_resource_ok

    @contextmanager
    def _cleanup_on_error(self):
        with ExitStack() as stack:
```

(次のページに続く)

(前のページからの続き)

```
            stack.push(self)
            yield
            # The validation check passed and didn't raise an exception
            # Accordingly, we want to keep the resource, and pass it
            # back to our caller
            stack.pop_all()

    def __enter__(self):
        resource = self.acquire_resource()
        with self._cleanup_on_error():
            if not self.check_resource_ok(resource):
                msg = "Failed validation for {!r}"
                raise RuntimeError(msg.format(resource))
        return resource

    def __exit__(self, *exc_details):
        # We don't need to duplicate any of our resource release logic
        self.release_resource()
```

try-finally + flag 変数パターンを置き換える

try-finally 文に、finally 句の内容を実行するかどうかを示すフラグ変数を組み合わせたパターンを目にすることがあるかもしれません。一番シンプルな (単に except 句を使うだけでは処理できない) ケースでは次のようなコードになります:

```
cleanup_needed = True
try:
    result = perform_operation()
    if result:
        cleanup_needed = False
finally:
    if cleanup_needed:
        cleanup_resources()
```

try 文を使ったコードでは、セットアップとクリーンアップのコードが任意の長さのコードで分離してしまうので、開発者やレビューアにとって問題になりえます。

ExitStack を使えば、代わりに with 文の終わりに実行されるコールバックを登録し、後でそのコールバックをスキップするかどうかを決定できます:

```
from contextlib import ExitStack

with ExitStack() as stack:
    stack.callback(cleanup_resources)
    result = perform_operation()
    if result:
        stack.pop_all()
```

これにより、別のフラグ変数を使う代わりに、必要なクリーンアップ処理を手前に明示しておくことができ

ます。

もしあるアプリケーションがこのパターンを多用するのであれば、小さいヘルパークラスを導入してよりシンプルにすることができます:

```
from contextlib import ExitStack

class Callback(ExitStack):
    def __init__(self, callback, /, *args, **kwds):
        super().__init__()
        self.callback(callback, *args, **kwds)

    def cancel(self):
        self.pop_all()

with Callback(cleanup_resources) as cb:
    result = perform_operation()
    if result:
        cb.cancel()
```

もしリソースのクリーンアップが単体の関数にまとまってない場合でも、*ExitStack.callback()* のデコレーター形式を利用してリソース開放処理を宣言することができます:

```
from contextlib import ExitStack

with ExitStack() as stack:
    @stack.callback
    def cleanup_resources():
        ...
    result = perform_operation()
    if result:
        stack.pop_all()
```

デコレータープロトコルの使用上、このように宣言されたコールバック関数は引数を取ることができません。その代わりに、リリースするリソースをクロージャー変数としてアクセスできる必要があります。

コンテキストマネージャーを関数デコレーターとして使う

ContextDecorator はコンテキストマネージャーを通常の `with` 文に加えて関数デコレーターとしても利用できるようにします。

例えば、関数やまとまった文を、そこに入った時と出た時の時間をトラックするロガーでラップしたい場合があります。そのために関数デコレーターとコンテキストマネージャーを別々に書く代わりに、*ContextDecorator* を継承すると 1 つの定義で両方の機能を提供できます:

```
from contextlib import ContextDecorator
import logging

logging.basicConfig(level=logging.INFO)
```

(次のページに続く)

(前のページからの続き)

```
class track_entry_and_exit(ContextDecorator):
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        logging.info('Entering: %s', self.name)

    def __exit__(self, exc_type, exc, exc_tb):
        logging.info('Exiting: %s', self.name)
```

このクラスのインスタンスはコンテキストマネージャーとしても利用でき:

```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

また関数デコレーターとしても利用できます:

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

コンテキストマネージャーを関数デコレーターとして使う場合、`__enter__()` メソッドの戻り値にアクセスする手段がないという制限があることに注意してください。もしその値が必要であれば、明示的な `with` 文を使う必要があります。

参考:

PEP 343 - "with" ステートメント Python の `with` 文の仕様、背景、および例が記載されています。

29.7.3 単回使用、再利用可能、およびリエントラントなコンテキストマネージャ

ほとんどのコンテキストマネージャは、`with` 文の中で一度だけ使われるような場合に効果的になるように書かれています。これら単回使用のコンテキストマネージャは毎回新規に生成されなければなりません - それらを再利用しようとすると、例外を引き起こすか、正しく動作しません。

この共通の制限が意味することは、コンテキストマネージャは (上記すべての使用例に示すとおり) 一般に `with` 文のヘッダ部分で直接生成することが推奨されるということです。

ファイルオブジェクトは単回使用のコンテキストマネージャ有効に利用した例です。最初の `with` 文によりファイルがクローズされ、それ以降そのファイルオブジェクトに対するすべての IO 操作を防止します。

contextmanager() により生成されたコンテキストマネージャも単回使用のコンテキスト マネージャです。二回目に使おうとした場合、内部にあるジェネレータが値の生成に失敗したと訴えるでしょう:

```
>>> from contextlib import contextmanager
>>> @contextmanager
```

(次のページに続く)

(前のページからの続き)

```
... def singleuse():
...     print("Before")
...     yield
...     print("After")
...
>>> cm = singleuse()
>>> with cm:
...     pass
...
Before
After
>>> with cm:
...     pass
...
Traceback (most recent call last):
    ...
RuntimeError: generator didn't yield
```

リエントラントなコンテキストマネージャ

より洗練されたコンテキストマネージャには" リエントラント" なものがあります。そのようなコンテキストマネージャは、複数の `with` 文で使えるだけでなく、同じコンテキストマネージャをすでに使っている `with` 文の **内部** でも使うことができます。

threading.RLock is an example of a reentrant context manager, as are *suppress()* and *redirect_stdout()*. Here's a very simple example of reentrant use:

```
>>> from contextlib import redirect_stdout
>>> from io import StringIO
>>> stream = StringIO()
>>> write_to_stream = redirect_stdout(stream)
>>> with write_to_stream:
...     print("This is written to the stream rather than stdout")
...     with write_to_stream:
...         print("This is also written to the stream")
...
>>> print("This is written directly to stdout")
This is written directly to stdout
>>> print(stream.getvalue())
This is written to the stream rather than stdout
This is also written to the stream
```

リエントラントな性質の実例はお互いを呼び出しあう複数の関数を含んでいる可能性が高く、したがってこの例よりもはるかに複雑です。

リエントラントであることはスレッドセーフであることと同じ **ではない** ことには注意が必要です。たとえば *redirect_stdout()* は、*sys.stdout* を異なるストリームに束縛することによりシステムの状態に対してグローバルな変更を行うことから、明らかにスレッドセーフではありません。

再利用可能なコンテキストマネージャ

単回使用のコンテキストマネージャとリエントラントなコンテキストマネージャのいずれとも異なるタイプに ” 再利用可能” なコンテキストマネージャがあります (あるいは、より明確には、” 再利用可能だがリエントラントでない” コンテキストマネージャです。リエントラントなコンテキストマネージャもまた再利用可能だからです)。再利用可能なコンテキストマネージャは複数回利用をサポートしますが、同じコンテキストマネージャのインスタンスがすでに with 文で使われている場合には失敗します (もしくは正しく動作しません)。

threading.Lock は再利用可能だがリエントラントでないコンテキストマネージャの例です (リエントラントなロックのためには *threading.RLock* を代わりに使う必要があります)。

再利用可能だがリエントラントでないコンテキストマネージャのもうひとつの例は *ExitStack* です。これは現在登録されている **全ての** コールバック関数を、どこで登録されたかにかかわらず、呼び出します:

```
>>> from contextlib import ExitStack
>>> stack = ExitStack()
>>> with stack:
...     stack.callback(print, "Callback: from first context")
...     print("Leaving first context")
...
Leaving first context
Callback: from first context
>>> with stack:
...     stack.callback(print, "Callback: from second context")
...     print("Leaving second context")
...
Leaving second context
Callback: from second context
>>> with stack:
...     stack.callback(print, "Callback: from outer context")
...     with stack:
...         stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Callback: from outer context
Leaving outer context
```

例における出力が示すように、ひとつのスタックオブジェクトを複数の with 文で再利用しても正しく動作します。しかし入れ子にして使った場合は、一番内側の with 文を抜ける際にスタックが空になります。これは望ましい動作とは思えません。

ひとつの *ExitStack* インスタンスを再利用する代わりに複数のインスタンスを使うことにより、この問題は回避することができます:

```
>>> from contextlib import ExitStack
>>> with ExitStack() as outer_stack:
...     outer_stack.callback(print, "Callback: from outer context")
...     with ExitStack() as inner_stack:
```

(次のページに続く)

(前のページからの続き)

```
...         inner_stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Leaving outer context
Callback: from outer context
```

29.8 abc --- 抽象基底クラス

ソースコード: Lib/abc.py

このモジュールは Python に **PEP 3119** で概要が示された **抽象基底クラス** (ABC) を定義する基盤を提供します。なぜこれが Python に付け加えられたかについてはその PEP を参照してください。(ABC に基づいた数の型階層を扱った **PEP 3141** と *numbers* モジュールも参照してください。)

collections.abc サブモジュールには ABC から派生した具象クラスがいくつかあります。もちろん、このクラスから、さらに派生させることもできます。また collections.abs サブモジュールにはいくつかの ABC もあって、あるクラスやインスタンスが特定のインタフェースを提供しているかどうか、たとえば、ハッシュ可能なのかやマッピングなのか、をテストできます。

このモジュールは、抽象基底クラスを定義するためのメタクラス *ABCMeta* と、継承を利用して抽象基底クラスを代替的に定義するヘルパークラス *ABC* を提供します。

class abc.ABC

ABCMeta をメタクラスとするヘルパークラスです。このクラスを使うと、混乱しがちなメタクラスを使わずに、単に *ABC* を継承するだけで抽象基底クラスを作成できます。例:

```
from abc import ABC

class MyABC(ABC):
    pass
```

ABC の型はやはり *ABCMeta* であり、そのため *ABC* から継承するときは、メタクラスの衝突を引き起こし得る多重継承のような、メタクラスを使う上でのいつもの用心が求められることに注意してください。metaclass キーワードを渡し、*ABCMeta* を直接利用することで、抽象基底クラスを直接定義することもできます。例:

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass
```

バージョン 3.4 で追加.

class abc.ABCMeta

抽象基底クラス (ABC) を定義するためのメタクラス。

ABC を作るときにこのメタクラスを使います。ABC は直接的にサブクラス化することができ、ミックスイン (mix-in) クラスのように振る舞います。また、無関係な具象クラス (組み込み型でも構いません) と無関係な ABC を ” 仮想的サブクラス” として登録できます -- これらとその子孫は組み込み関数 *issubclass()* によって登録した ABC のサブクラスと判定されますが、登録した ABC は MRO (Method Resolution Order, メソッド解決順) には現れませんし、この ABC のメソッド実装が (*super()* を通してだけでなく) 呼び出し可能になるわけでもありません。[*1]

メタクラス *ABCMeta* を使って作られたクラスには以下のメソッドがあります:

register(*subclass*)

subclass を ” 仮想的サブクラス” としてこの ABC に登録します。たとえば:

```
from abc import ABC

class MyABC(ABC):
    pass

MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

バージョン 3.3 で変更: クラスデコレータとして使うことができるように、登録されたサブクラスを返します。

バージョン 3.4 で変更: *register()* の呼び出しを検出するために、*get_cache_token()* 関数を使うことができます。

また、次のメソッドを抽象基底クラスの中でオーバーライドできます:

__subclasshook__(*subclass*)

(クラスメソッドとして定義しなければなりません。)

subclass がこの ABC のサブクラスと見なせるかどうかチェックします。これによって ABC のサブクラスと見なしたい全てのクラスについて *register()* を呼び出すことなく issubclass の振る舞いをさらにカスタマイズできます。(このクラスメソッドは ABC の __subclasscheck__() メソッドから呼び出されます。)

このメソッドは True, False または NotImplemented を返さなければなりません。True を返す場合は、*subclass* はこの ABC のサブクラスと見なされます。False を返す場合は、*subclass* はたとえ通常の意味でサブクラスであっても ABC のサブクラスではないと見なされます。NotImplemented の場合、サブクラスチェックは通常のメカニズムに戻ります。

この概念のデモとして、次の ABC 定義の例を見てください:

[*1] C++ プログラマは Python の仮想的基底クラスの概念は C++ のものと同じではないということを銘記すべきです。

```
class Foo:
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
        return iter(self)

class MyIterable(ABC):

    @abstractmethod
    def __iter__(self):
        while False:
            yield None

    def get_iterator(self):
        return self.__iter__()

    @classmethod
    def __subclasshook__(cls, C):
        if cls is MyIterable:
            if any("__iter__" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented

MyIterable.register(Foo)
```

ABC MyIterable は標準的なイテラブルのメソッド *__iter__()* を抽象メソッドとして定義します。ここで与えられている抽象クラスの実装は、サブクラスから呼び出すことができます。get_iterator() メソッドも MyIterable 抽象基底クラスの一部ですが、抽象的でない派生クラスはこれをオーバーライドする必要はありません。

ここで定義されるクラスメソッド *__subclasshook__()* の意味は、*__iter__()* メソッドがクラスの (または *__mro__* でアクセスされる基底クラスの一つの) *__dict__* にある場合にもそのクラスが MyIterable だと見なされるということです。

最後に、一番下の行は Foo を *__iter__()* メソッドを定義しないにもかかわらず MyIterable の仮想的サブクラスにします (Foo は古い様式の __len__() と __getitem__() を用いたイテレータプロトコルを使っています。)。これによって Foo のメソッドとして get_iterator が手に入るわけではないことに注意してください。それは別に提供されています。

abc モジュールは以下のデコレータも提供しています:

@abc.abstractmethod

抽象メソッドを示すデコレータです。

このデコレータを使うには、クラスのメタクラスが *ABCMeta* かそれを継承したものである必要があります。*ABCMeta* の派生クラスをメタクラスに持つクラスは、全ての抽象メソッドとプロパティをオーバーロードしない限りインスタンス化することができません。抽象メソッドは通常の 'super' 呼び出し方法で呼ぶことができます。*abstractmethod()* はプロパティやデスクリプタのために抽象メソッド

を定義することもできます。

クラスに動的に抽象メソッドを追加する、あるいはメソッドやクラスが作られた後から抽象的かどうかの状態を変更しようと試みることは、サポートされません。*abstractmethod()* が影響を与えるのは正規の継承により派生したサブクラスのみで、ABC の `register()` メソッドで登録された ” 仮想的サブクラス” は影響されません。

abstractmethod() が他のメソッドデスクリプタと組み合わされる場合、次の例のように、一番内側のデコレータとして適用しなければなりません:

```
class C(ABC):
    @abstractmethod
    def my_abstract_method(self, ...):
        ...
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, ...):
        ...
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(...):
        ...

    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
    @my_abstract_property.setter
    @abstractmethod
    def my_abstract_property(self, val):
        ...

    @abstractmethod
    def _get_x(self):
        ...
    @abstractmethod
    def _set_x(self, val):
        ...
    x = property(_get_x, _set_x)
```

デスクリプタを ABC 機構と協調させるために、デスクリプタは `__isabstractmethod__` を使って自身が抽象であることを示さなければなりません。一般的に、この属性は、そのデスクリプタを構成するのに使われたメソッドのいずれかが抽象である場合に `True` になります。例えば、Python 組み込みの *property* は、次のコードと透過に振る舞います:

```
class Descriptor:
    ...
    @property
    def __isabstractmethod__(self):
        return any(getattr(f, '__isabstractmethod__', False) for
                   f in (self._fget, self._fset, self._fdel))
```

注釈: Java の抽象メソッドと違い、これらの抽象メソッドは実装を持ち得ます。この実装は *super()* メカニズムを通してそれをオーバーライドしたクラスから呼び出すことができます。これは協調的多重継承を使ったフレームワークにおいて super 呼び出しの終点として有効です。

abc モジュールは以下のレガシーなデコレータも提供しています:

@abc.abstractclassmethod

バージョン 3.2 で追加.

バージョン 3.3 で非推奨: *classmethod* を *abstractmethod()* と一緒に使えるようになったため、このデコレータは冗長になりました。

組み込みの *classmethod()* のサブクラスで、抽象クラスメソッドであることを示します。それ以外は *abstractmethod()* と同じです。

この特殊ケースは *classmethod()* デコレータが抽象メソッドに適用された場合に抽象的だと正しく認識されるようになったため撤廃されます:

```
class C(ABC):
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, ...):
        ...
```

@abc.abstractstaticmethod

バージョン 3.2 で追加.

バージョン 3.3 で非推奨: *staticmethod* を *abstractmethod()* と一緒に使えるようになったため、このデコレータは冗長になりました。

組み込みの *staticmethod()* のサブクラスで、抽象静的メソッドであることを示します。それ以外は *abstractmethod()* と同じです。

この特殊ケースは *staticmethod()* デコレータが抽象メソッドに適用された場合に抽象的だと正しく認識されるようになったため撤廃されます:

```
class C(ABC):
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(...):
        ...
```

@abc.abstractproperty

バージョン 3.3 で非推奨: *property*、property.getter()、property.setter() および property.deleter() を *abstractmethod()* と一緒に使えるようになったため、このデコレータは冗長になりました。

組み込みの *property()* のサブクラスで、抽象プロパティであることを示します。

この特殊ケースは *property()* デコレータが抽象メソッドに適用された場合に抽象的だと正しく認識されるようになったため撤廃されます:

```
class C(ABC):
    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
```

この例は読み出し専用のプロパティを定義しています。プロパティを構成するメソッドの 1 つ以上を abstract にすることで、読み書きできる抽象プロパティを定義することができます:

```
class C(ABC):
    @property
    def x(self):
        ...

    @x.setter
    @abstractmethod
    def x(self, val):
        ...
```

構成するメソッド全てが abstract でない場合、abstract と定義されたメソッドのみが、具象サブクラスによってオーバーライドする必要があります:

```
class D(C):
    @C.x.setter
    def x(self, val):
        ...
```

abc モジュールは以下の関数も提供しています:

abc.**get_cache_token**()
現在の抽象基底クラスのキャッシュトークンを返します。

このトークンは、仮想サブクラスのための抽象基底クラスの現在のバージョンを特定する (等価性検査をサポートしている) 不透明なオブジェクトです。任意の ABC での *ABCMeta.register()* の呼び出しごとに、トークンは変更されます。

バージョン 3.4 で追加.

脚注

29.9 atexit --- 終了ハンドラ

atexit モジュールは、クリーンアップ関数の登録およびその解除を行う関数を定義します。登録された関数はインタプリタの通常終了時に自動的に実行されます。*atexit* はそれら関数を登録した順と **逆に** 実行します; A、B、C の順に登録した場合、インタプリタ終了時に C、B、A の順に実行されます。

注意: このモジュールを使用して登録された関数は、プログラムが Python が扱わないシグナルによって kill された場合、Python 内部で致命的なエラーが検出された場合、あるいは *os._exit()* が呼び出された場合は実行されません。

バージョン 3.7 で変更: C-API のサブインタープリタで使われているとき、登録された関数は登録先のインタープリタのローカルな関数になります。

atexit.**register**(*func*, **args*, ***kwargs*)

func を終了時に実行する関数として登録します。*func* に渡す引数は *register()* の引数として指定しなければなりません。同じ関数を同じ引数で複数回登録できます。

通常のプログラムの終了時、例えば *sys.exit()* が呼び出されるとき、あるいは、メインモジュールの実行が完了したときに、登録された全ての関数を、最後に登録されたものから順に呼び出します。通常、より低レベルのモジュールはより高レベルのモジュールより前に import されるので、後で後始末が行われるという仮定に基づいています。

終了ハンドラの実行中に例外が発生すると、(*SystemExit* 以外の場合は) トレースバックを表示して、例外の情報を保存します。全ての終了ハンドラに動作するチャンスを与えた後に、最後に送出された例外を再送出します。

この関数は *func* を返し、これをデコレータとして利用できます。

atexit.**unregister**(*func*)

func をインタプリタ終了時に実行される関数のリストから削除します。*unregister()* で削除されると、*func* は、たとえ複数個登録されていてもインタプリタ終了時に呼び出されないことが保証されます。*func* が登録されていない場合、*unregister()* は何もせず、何も通知しません。

参考:

readline **モジュール** *readline* ヒストリファイルを読み書きするための *atexit* の有用な例です。

29.9.1 atexit の例

次の簡単な例では、あるモジュールを import した時にカウンタを初期化しておき、プログラムが終了するときにアプリケーションがこのモジュールを明示的に呼び出さなくてもカウンタが更新されるようにする方法を示しています。

```
try:
    with open("counterfile") as infile:
        _count = int(infile.read())
except FileNotFoundError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    with open("counterfile", "w") as outfile:
```

(次のページに続く)

(前のページからの続き)

```
        outfile.write("%d" % _count)

import atexit
atexit.register(savecounter)
```

register() に指定した位置引数とキーワード引数は登録した関数を呼び出す際に渡されます:

```
def goodbye(name, adjective):
    print('Goodbye, %s, it was %s to meet you.' % (name, adjective))

import atexit
atexit.register(goodbye, 'Donny', 'nice')

# or:
atexit.register(goodbye, adjective='nice', name='Donny')
```

デコレータ として利用する例:

```
import atexit

@atexit.register
def goodbye():
    print("You are now leaving the Python sector.")
```

デコレータとして利用できるのは、その関数が引数なしで呼び出された場合に限られます。

29.10 traceback --- スタックトレースの表示または取得

ソースコード: Lib/traceback.py

このモジュールは Python プログラムのスタックトレースを抽出し、書式を整え、表示するための標準インターフェースを提供します。モジュールがスタックトレースを表示するとき、Python インタープリタの動作を正確に模倣します。インタープリタの" ラッパー" の場合のように、プログラムの制御の元でスタックトレースを表示したいと思ったときに役に立ちます。

モジュールはトレースバックオブジェクトを使用します。このオブジェクトの型は *sys.last_traceback* 変数に格納され、*sys.exc_info()* の三番目に返されます。

このモジュールには、以下の関数が定義されています:

traceback.print_tb(*tb*, *limit=None*, *file=None*)

limit が正の場合、トレースバックオブジェクト *tb* から *limit* までのスタックトレース項目 (呼び出し側のフレームから開始) を出力します。そうでない場合、最新 `abs(limit)` 項目を出力します。*limit* が省略されるか `None` の場合、すべての項目が出力されます。*file* が省略されるか `None` の場合は、`sys.stderr` へ出力されます。そうでない場合、*file* は出力を受け取る開いたファイルまたは file-like オブジェクトでなければなりません。

バージョン 3.5 で変更: 負の *limit* がサポートされました。

traceback.**print_exception**(*etype*, *value*, *tb*, *limit=None*, *file=None*, *chain=True*)

トレースバックオブジェクト *tb* から例外の情報とスタックトレースの項目を *file* に出力します。この関数は以下の点で *print_tb()* と異なります:

- *tb* が None でない場合ヘッダ Traceback (most recent call last): を出力します
- スタックトレースの後に例外 *etype* と *value* を出力します
- *type(value)* が *SyntaxError* であり、*value* が適切な形式を持っていれば、そのエラーのおおよその位置を示すマークと共にシンタックスエラーが発生した行が表示されます。

オプション引数 *limit* の意味は *print_tb()* のものと同じです。*chain* が真の場合 (デフォルト)、連鎖した例外 (例外の __cause__ か __context__ 属性) も出力されます。これはインタープリタ自身が処理されていない例外を出力するときと同じです。

バージョン 3.5 で変更: 引数 *etype* は無視され、*value* の型から推論されます。

traceback.**print_exc**(*limit=None*, *file=None*, *chain=True*)

print_exception(*sys.exc_info(), limit, file, chain) の省略表現です。

traceback.**print_last**(*limit=None*, *file=None*, *chain=True*)

print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit, file, chain) の省略表現です。一般に、例外が対話的なプロンプトに達した後にのみ機能します (*sys.last_type* 参照)。

traceback.**print_stack**(*f=None*, *limit=None*, *file=None*)

limit が正の場合、最大 *limit* 個のスタックトレース項目 (呼び出し地点から開始) を出力します。そうでない場合、最新の abs(limit) 個を出力します。*limit* が省略されるか None の場合、すべての項目が出力されます。オプションの *f* 引数は、開始するスタックフレームを指定するために用いることができます。オプションの *file* 引数は *print_tb()* の *file* 引数と同様の意味を持っています。

バージョン 3.5 で変更: 負の *limit* がサポートされました。

traceback.**extract_tb**(*tb*, *limit=None*)

トレースバックオブジェクト *tb* から取り出された ” 前処理済み” スタックトレース項目のリストを表す *StackSummary* オブジェクトを返します。これはスタックトレースの異なる書式化を行うために役に立ちます。オプションの *limit* 引数は *print_tb()* の *limit* 引数と同様の意味を持っています。” 前処理済み” スタックトレース項目は 属性 filename, lineno, name, および line を含む *FrameSummary* オブジェクトで、スタックトレースに対して通常出力される情報を表しています。line は前後の空白を取り除いた文字列です。ソースが利用できない場合は None です。

traceback.**extract_stack**(*f=None*, *limit=None*)

現在のスタックフレームから生のトレースバックを取り出します。戻り値は *extract_tb()* と同じ形式です。オプションの *f* と *limit* 引数は *print_stack()* と同じ意味を持ちます。

traceback.**format_list**(*extracted_list*)

extract_tb() または *extract_stack()* が返すタプルのリストまたは *FrameSummary* オブジェクト

が与えられると、出力の準備を整えた文字列のリストを返します。結果として生じるリストの中の各文字列は、引数リストの中の同じインデックスの要素に対応します。各文字列は末尾に改行が付いています。さらに、ソーステキスト行が None でないそれらの要素に対しては、文字列は内部に改行を含んでいるかもしれません。

traceback.**format_exception_only**(*etype*, *value*)

トレースバックの例外部分を書式化します。引数は sys.last_type と sys.last_value のような例外の型と値です。戻り値はそれぞれが改行で終わっている文字列のリストです。通常、リストは一つの文字列を含んでいます。しかし、*SyntaxError* 例外に対しては、(出力されるときに) 構文エラーが起きた場所についての詳細な情報を示す行をいくつか含んでいます。どの例外が起きたのかを示すメッセージは、常にリストの最後の文字列です。

traceback.**format_exception**(*etype*, *value*, *tb*, *limit=None*, *chain=True*)

スタックトレースと例外情報を書式化します。引数は *print_exception()* の対応する引数と同じ意味を持ちます。戻り値は文字列のリストで、それぞれの文字列は改行で終わり、そのいくつかは内部に改行を含みます。これらの行が連結されて出力される場合は、厳密に *print_exception()* と同じテキストが出力されます。

バージョン 3.5 で変更: 引数 *etype* は無視され、*value* の型から推論されます。

traceback.**format_exc**(*limit=None*, *chain=True*)

これは、print_exc(limit) に似ていますが、ファイルに出力する代わりに文字列を返します。

traceback.**format_tb**(*tb*, *limit=None*)

format_list(extract_tb(tb, limit)) の省略表現です。

traceback.**format_stack**(*f=None*, *limit=None*)

format_list(extract_stack(f, limit)) の省略表現です。

traceback.**clear_frames**(*tb*)

各フレームオブジェクトの clear() メソッドを呼んでトレースバック *tb* 内の全スタックフレームの局所変数を消去します。

バージョン 3.4 で追加.

traceback.**walk_stack**(*f*)

Walk a stack following f.f_back from the given frame, yielding the frame and line number for each frame. If *f* is None, the current stack is used. This helper is used with *StackSummary.extract()*.

バージョン 3.5 で追加.

traceback.**walk_tb**(*tb*)

Walk a traceback following tb_next yielding the frame and line number for each frame. This helper is used with *StackSummary.extract()*.

バージョン 3.5 で追加.

このモジュールは以下のクラスも定義しています:

29.10.1 TracebackException オブジェクト

バージョン 3.5 で追加.

TracebackException オブジェクトは実際の例外から作られ、後の出力のために軽量な方法でデータをキャプチャします。

class traceback.**TracebackException**(*exc_type*, *exc_value*, *exc_traceback*, ***, *limit=None*, *lookup_lines=True*, *capture_locals=False*)

後のレンダリングのために例外をキャプチャします。*limit*、*lookup_lines*、*capture_locals* は *StackSummary* class のものです。

局所変数がキャプチャされたとき、それらはトレースバックに表示されることに注意してください。

__cause__

元々の `__cause__` の *TracebackException*。

__context__

元々の `__context__` の *TracebackException*。

__suppress_context__

元々の例外からの `__suppress_context__`。

stack

トレースバックを表す *StackSummary*。

exc_type

元々のトレースバックのクラス。

filename

構文エラー用 - エラーが発生したファイルの名前。

lineno

構文エラー用 - エラーが発生した行番号。

text

構文エラー用 - エラーが発生したテクスト。

offset

構文エラー用 - エラーが発生したテクストへのオフセット。

msg

構文エラー用 - コンパイラのエラーメッセージ。

classmethod from_exception(*exc*, ***, *limit=None*, *lookup_lines=True*, *capture_locals=False*)

後のレンダリングのために例外をキャプチャします。*limit*、*lookup_lines*、*capture_locals* は *StackSummary* class のものです。

局所変数がキャプチャされたとき、それらはトレースバックに表示されることに注意してください。

format(*, *chain=True*)
例外を書式化します。

chain が `True` でない場合 `__cause__` と `__context__` は書式化されません。

返り値は文字列のジェネレータで、それぞれ改行で終わりますが、内部に改行を持つものもあります。*print_exception()* はこのメソッドのラッパで、単にファイルに出力します。

例外発生を指すメッセージは常に出力の最後の文字列です。

format_exception_only()
トレースバックの例外部を書式化します。

返り値は文字列のジェネレータで、それぞれ改行で終わります。

通常、ジェネレータが出す文字列は一つです。しかしながら *SyntaxError* 例外の場合、構文例外が発生した箇所の詳細な情報を表す行を複数出します。

例外発生を指すメッセージは常に出力の最後の文字列です。

29.10.2 StackSummary オブジェクト

バージョン 3.5 で追加.

StackSummary オブジェクトは書式化の準備ができているコールスタックを表します。

class traceback.StackSummary

classmethod extract(*frame_gen*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*)
ジェネレータ (例えば *walk_stack()* や *walk_tb()* が返すもの) から *StackSummary* を構築します。

If *limit* is supplied, only this many frames are taken from *frame_gen*. If *lookup_lines* is `False`, the returned *FrameSummary* objects will not have read their lines in yet, making the cost of creating the *StackSummary* cheaper (which may be valuable if it may not actually get formatted). If *capture_locals* is `True` the local variables in each *FrameSummary* are captured as object representations.

classmethod from_list(*a_list*)
Construct a *StackSummary* object from a supplied list of *FrameSummary* objects or old-style list of tuples. Each tuple should be a 4-tuple with filename, lineno, name, line as the elements.

format()
Returns a list of strings ready for printing. Each string in the resulting list corresponds to a single frame from the stack. Each string ends in a newline; the strings may contain internal newlines as well, for those items with source text lines.

For long sequences of the same frame and line, the first few repetitions are shown, followed by a summary line stating the exact number of further repetitions.

バージョン 3.6 で変更: Long sequences of repeated frames are now abbreviated.

29.10.3 FrameSummary オブジェクト

バージョン 3.5 で追加.

FrameSummary オブジェクトトレースバック内の単一のフレームを表します。

class traceback.**FrameSummary**(*filename*, *lineno*, *name*, *lookup_line=True*, *locals=None*, *line=None*)

Represent a single frame in the traceback or stack that is being formatted or printed. It may optionally have a stringified version of the frames locals included in it. If *lookup_line* is `False`, the source code is not looked up until the *FrameSummary* has the `line` attribute accessed (which also happens when casting it to a tuple). `line` may be directly provided, and will prevent line lookups happening at all. *locals* is an optional local variable dictionary, and if supplied the variable representations are stored in the summary for later display.

29.10.4 トレースバックの例

この簡単な例では基本的な read-eval-print ループを実装しています。標準的な Python の対話インタープリタループに似ていますが、Python のものより便利ではありません。インタープリタループのより完全な実装については、*code* モジュールを参照してください。

```
import sys, traceback

def run_user_code(envdir):
    source = input(">>> ")
    try:
        exec(source, envdir)
    except Exception:
        print("Exception in user code:")
        print("-"*60)
        traceback.print_exc(file=sys.stdout)
        print("-"*60)

envdir = {}
while True:
    run_user_code(envdir)
```

次の例は例外とトレースバックの出力並びに形式が異なることを示します:

```
import sys, traceback

def lumberjack():
    bright_side_of_death()
```

(次のページに続く)

(前のページからの続き)

```
def bright_side_of_death():
    return tuple()[0]

try:
    lumberjack()
except IndexError:
    exc_type, exc_value, exc_traceback = sys.exc_info()
    print("*** print_tb:")
    traceback.print_tb(exc_traceback, limit=1, file=sys.stdout)
    print("*** print_exception:")
    # exc_type below is ignored on 3.5 and later
    traceback.print_exception(exc_type, exc_value, exc_traceback,
                              limit=2, file=sys.stdout)
    print("*** print_exc:")
    traceback.print_exc(limit=2, file=sys.stdout)
    print("*** format_exc, first and last line:")
    formatted_lines = traceback.format_exc().splitlines()
    print(formatted_lines[0])
    print(formatted_lines[-1])
    print("*** format_exception:")
    # exc_type below is ignored on 3.5 and later
    print(repr(traceback.format_exception(exc_type, exc_value,
                                          exc_traceback)))
    print("*** extract_tb:")
    print(repr(traceback.extract_tb(exc_traceback)))
    print("*** format_tb:")
    print(repr(traceback.format_tb(exc_traceback)))
    print("*** tb_lineno:", exc_traceback.tb_lineno)
```

この例の出力は次のようになります:

```
*** print_tb:
  File "<doctest...>", line 10, in <module>
    lumberjack()
*** print_exception:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** print_exc:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** format_exc, first and last line:
Traceback (most recent call last):
```

(次のページに続く)

(前のページからの続き)

```
IndexError: tuple index out of range
*** format_exception:
['Traceback (most recent call last):\n',
 '  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n',
 'IndexError: tuple index out of range\n']
*** extract_tb:
[<FrameSummary file <doctest...>, line 10 in <module>>,
 <FrameSummary file <doctest...>, line 4 in lumberjack>,
 <FrameSummary file <doctest...>, line 7 in bright_side_of_death>]
*** format_tb:
['  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n']
*** tb_lineno: 10
```

次の例は、スタックの print と format の違いを示しています:

```
>>> import traceback
>>> def another_function():
...     lumberstack()
...
>>> def lumberstack():
...     traceback.print_stack()
...     print(repr(traceback.extract_stack()))
...     print(repr(traceback.format_stack()))
...
>>> another_function()
  File "<doctest>", line 10, in <module>
    another_function()
  File "<doctest>", line 3, in another_function
    lumberstack()
  File "<doctest>", line 6, in lumberstack
    traceback.print_stack()
[('<doctest>', 10, '<module>', 'another_function()'),
 ('<doctest>', 3, 'another_function', 'lumberstack()'),
 ('<doctest>', 7, 'lumberstack', 'print(repr(traceback.extract_stack()))')]
['  File "<doctest>", line 10, in <module>\n    another_function()\n',
 '  File "<doctest>", line 3, in another_function\n    lumberstack()\n',
 '  File "<doctest>", line 8, in lumberstack\n    print(repr(traceback.format_stack()))\n']
```

最後の例は、残りの幾つかの関数のデモをします:

```
>>> import traceback
>>> traceback.format_list([('spam.py', 3, '<module>', 'spam.eggs()'),
...                        ('eggs.py', 42, 'eggs', 'return "bacon"')])
['  File "spam.py", line 3, in <module>\n    spam.eggs()\n',
 '  File "eggs.py", line 42, in eggs\n    return "bacon"\n']
>>> an_error = IndexError('tuple index out of range')
>>> traceback.format_exception_only(type(an_error), an_error)
```

(次のページに続く)

(前のページからの続き)

```
['IndexError: tuple index out of range\n']
```

29.11 __future__ --- future 文の定義

ソースコード: Lib/__future__.py

__future__ は実際にモジュールであり、次の 3 つの役割があります:

- import 文を解析する既存のツールを混乱させることを避け、インポートしようとしているモジュールを見つけられるようにするため。
- 2.1 以前のリリースで future 文 が実行された場合に、最低でもランタイム例外を投げるようにするため (*__future__* のインポートは失敗します。なぜなら、2.1 以前にはそういう名前のモジュールはなかったからです)。
- 互換性のない変化がいつ言語に導入され、いつ言語の一部になる --- あるいは、なった --- のかを文書化するため。これは実行できる形式で書かれたドキュメントなので、*__future__* をインポートしてその中身を調べることでプログラムから調査することができます。

__future__.py のそれぞれの文は次のような形式をしています:

```
FeatureName = _Feature(OptionalRelease, MandatoryRelease,
                       CompilerFlag)
```

ここで、普通は、*OptionalRelease* は *MandatoryRelease* より小さく、2 つとも *sys.version_info* と同じフォーマットの 5 つのタプルからなります:

```
(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int
 PY_MINOR_VERSION, # the 1; an int
 PY_MICRO_VERSION, # the 0; an int
 PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; string
 PY_RELEASE_SERIAL # the 3; an int
)
```

OptionalRelease はその機能が導入された最初のリリースを記録します。

まだ時期が来ていない *MandatoryRelease* の場合、*MandatoryRelease* はその機能が言語の一部となるリリースを記します。

その他の場合、*MandatoryRelease* はその機能がいつ言語の一部になったのかを記録します。そのリリースから、あるいはそれ以降のリリースでは、この機能を使う際に future 文は必要ではありませんが、future 文を使い続けても構いません。

MandatoryRelease は None になるかもしれません。つまり、予定された機能が破棄されたということです。

_Feature クラスのインスタンスには対応する 2 つのメソッド、getOptionalRelease() と getMandatoryRelease() があります。

CompilerFlag は、動的にコンパイルされるコードでその機能を有効にするために、組み込み関数 *compile()* の第 4 引数に渡す (ビットフィールド) フラグです。このフラグは `_Feature` インスタンスの `compiler_flag` 属性に保存されています。

機能の記述が *__future__* から削除されたことはまだありません。Python 2.1 で future 文が導入されて以来、この仕組みを使って以下の機能が言語に導入されてきました:

feature	optional in	mandatory in	effect
nested_scopes	2.1.0b1	2.2	**PEP 227**: *Statically Nested Scopes*
generators	2.2.0a1	2.3	**PEP 255**: *Simple Generators*
division	2.2.0a2	3.0	**PEP 238**: *Changing the Division Operator*
absolute_import	2.5.0a1	3.0	**PEP 328**: *Imports: Multi-Line and Absolute/Relative*
with_statement	2.5.0a1	2.6	**PEP 343**: *The "with" Statement*
print_function	2.6.0a2	3.0	**PEP 3105**: *Make print a function*
unicode_literals	2.6.0a2	3.0	**PEP 3112**: *Bytes literals in Python 3000*
generator_stop	3.5.0b1	3.7	**PEP 479**: *StopIteration handling inside generators*
annotations	3.7.0b1	3.10	**PEP 563**: *Postponed evaluation of annotations*

参考:

future　コンパイラがどのように future インポートを扱うか。

29.12 gc --- ガベージコレクタインターフェース

このモジュールは、循環ガベージコレクタの無効化・検出頻度の調整・デバッグオプションの設定などを行うインターフェースを提供します。また、検出した到達不能オブジェクトのうち、解放する事ができないオブジェクトを参照する事もできます。循環ガベージコレクタは Pyhon の参照カウントを補うためのものなので、もしプログラム中で循環参照が発生しない事が明らかな場合には検出をする必要はありません。自動検出は、`gc.disable()` で停止する事ができます。メモリリークをデバッグするときには、`gc.set_debug(gc.DEBUG_LEAK)` とします。これは `gc.DEBUG_SAVEALL` を含んでいることに注意しましょう。ガベージとして検出されたオブジェクトは、インスペクション用に gc.garbage に保存されます。

gc モジュールは、以下の関数を提供しています:

`gc.enable()`
　自動ガベージコレクションを有効にします。

gc.disable()
自動ガベージコレクションを無効にします。

gc.isenabled()
自動ガベージコレクションが有効なら True を返します。

gc.collect(*generation=2*)
引数を指定しない場合は、全ての検出を行います。オプション引数 *generation* は、どの世代を検出するかを (0 から 2 までの) 整数値で指定します。無効な世代番号を指定した場合は *ValueError* が発生します。検出した到達不可オブジェクトの数を返します。

ビルトイン型が持っている free list は、フル GC か最高世代 (2) の GC の時にクリアされます。*float* など、実装によって幾つかの free list では全ての要素が解放されるわけではありません。

gc.set_debug(*flags*)
ガベージコレクションのデバッグフラグを設定します。デバッグ情報は sys.stderr に出力されます。デバッグフラグは、下の値の組み合わせを指定する事ができます。

gc.get_debug()
現在のデバッグフラグを返します。

gc.get_objects(*generation=None*)
Returns a list of all objects tracked by the collector, excluding the list returned. If *generation* is not None, return only the objects tracked by the collector that are in that generation.

バージョン 3.8 で変更: 新しい *generation* パラメータ。

引数 generation で 監査イベント gc.get_objects を送出します。

gc.get_stats()
インタプリタが開始してからの、世代ごと回収統計を持つ辞書の、3 世代ぶんのリストを返します。キーの数は将来変わるかもしれませんが、現在のところそれぞれの辞書には以下の項目が含まれています:

- collections は、この世代が検出を行った回数です;
- collected は、この世代内で回収されたオブジェクトの総数です;
- uncollectable は、この世代内で回収不能であることがわかった (そしてそれゆえに *garbage* リストに移動した) オブジェクトの総数です。

バージョン 3.4 で追加.

gc.set_threshold(*threshold0*[, *threshold1*[, *threshold2*]])
ガベージコレクションの閾値（検出頻度）を指定します。*threshold0* を 0 にすると、検出は行われません。

The GC classifies objects into three generations depending on how many collection sweeps they have survived. New objects are placed in the youngest generation (generation 0). If an object survives a collection it is moved into the next older generation. Since generation 2 is the oldest generation, objects in that generation remain there after a collection. In order to decide when to run, the collector keeps track of the number object allocations and deallocations since the last

collection. When the number of allocations minus the number of deallocations exceeds *threshold0*, collection starts. Initially only generation 0 is examined. If generation 0 has been examined more than *threshold1* times since generation 1 has been examined, then generation 1 is examined as well. With the third generation, things are a bit more complicated, see Collecting the oldest generation for more information.

gc.get_count()
現在の検出数を、(count0, count1, count2) のタプルで返します。

gc.get_threshold()
現在の検出閾値を、(threshold0, threshold1, threshold2) のタプルで返します。

gc.get_referrers(*objs)
objs で指定したオブジェクトのいずれかを参照しているオブジェクトのリストを返します。この関数では、ガベージコレクションをサポートしているコンテナのみを返します。他のオブジェクトを参照していても、ガベージコレクションをサポートしていない拡張型は含まれません。

尚、戻り値のリストには、すでに参照されなくなっているが、循環参照の一部でまだガベージコレクションで回収されていないオブジェクトも含まれるので注意が必要です。有効なオブジェクトのみを取得する場合、*get_referrers()* の前に *collect()* を呼び出してください。

警告: *get_referrers()* から返されるオブジェクトは作りかけや利用できない状態である場合があるので、利用する際には注意が必要です。*get_referrers()* をデバッグ以外の目的で利用するのは避けてください。

引数 objs を指定して **監査イベント** gc.get_referrers を送出します。

gc.get_referents(*objs)
引数で指定したオブジェクトのいずれかから参照されている、全てのオブジェクトのリストを返します。参照先のオブジェクトは、引数で指定したオブジェクトの C レベルメソッド tp_traverse で取得し、全てのオブジェクトが直接到達可能な全てのオブジェクトを返すわけではありません。tp_traverse はガベージコレクションをサポートするオブジェクトのみが実装しており、ここで取得できるオブジェクトは循環参照の一部となる可能性のあるオブジェクトのみです。従って、例えば整数オブジェクトが直接到達可能であっても、このオブジェクトは戻り値には含まれません。

引数 objs を指定して **監査イベント** gc.get_referents を送出します。

gc.is_tracked(*obj*)
obj がガベージコレクタに管理されていたら True を返し、それ以外の場合は False を返します。一般的なルールとして、アトミックな (訳注: 他のオブジェクトを参照しないで単一で値を表す型。int や str など) 型のインスタンスは管理されず、それ以外の型 (コンテナ型、ユーザー定義型など) のインスタンスは管理されます。しかし、いくつかの型では専用の最適化を行い、シンプルなインスタンスの場合に GC のオーバーヘッドを減らしています。(例: 全ての key と value がアトミック型の値である dict)

```
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
False
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
False
>>> gc.is_tracked({"a": []})
True
```

バージョン 3.1 で追加.

gc.**freeze**()

Freeze all the objects tracked by gc - move them to a permanent generation and ignore all the future collections. This can be used before a POSIX fork() call to make the gc copy-on-write friendly or to speed up collection. Also collection before a POSIX fork() call may free pages for future allocation which can cause copy-on-write too so it's advised to disable gc in parent process and freeze before fork and enable gc in child process.

バージョン 3.7 で追加.

gc.**unfreeze**()

Permanent 世代領域にあるオブジェクトを解凍します。つまり、最も古い世代へ戻します。

バージョン 3.7 で追加.

gc.**get_freeze_count**()

Permanent 世代領域にあるオブジェクトの数を返します。

バージョン 3.7 で追加.

以下の変数は読み出し専用アクセスのために提供されています。(この変数を操作することはできますが、その値は記憶されません):

gc.**garbage**

到達不能であることが検出されたが、解放する事ができないオブジェクトのリスト（回収不能オブジェクト)。Python 3.4 からは、NULL でない tp_del スロットを持つ C 拡張型のインスタンスを使っている場合を除き、このリストはほとんど常に空であるはずです。

DEBUG_SAVEALL が設定されている場合、全ての到達不能オブジェクトは解放されずにこのリストに格納されます。

バージョン 3.2 で変更: **インタプリタシャットダウン** 時にこのリストが空でない場合、(デフォルトでは黙りますが) *ResourceWarning* が発行されます。*DEBUG_UNCOLLECTABLE* がセットされていると、加えて回収不能オブジェクトを出力します。

バージョン 3.4 で変更: **PEP 442** に従い、__del__() メソッドを持つオブジェクトはもう *gc.garbage* に行き着くことはありません。

gc.callbacks
: ガベージコレクタの起動前と終了後に呼び出されるコールバック関数のリスト。コールバックは *phase* と *info* の 2 つの引数で呼び出されます。

  phase は以下 2 つのいずれかです:

  "start": ガベージコレクションを始めます。

  "stop": ガベージコレクションが終了しました。

  info はコールバックに付加情報を提供する辞書です。現在のところ以下のキーが定義されています:

  "generation": 回収される最も古い世代。

  "collected": *phase* が "stop" のとき、正常に回収されたオブジェクトの数。

  "uncollectable": *phase* が "stop" のとき、回収出来ずに *garbage* リストに入れられたオブジェクトの数。

  アプリケーションは自身のコールバックをこのリストに追加出来ます。基本的なユースケースは以下のようなものでしょう:

  世代が回収される頻度やガベージコレクションにかかった時間の長さといった、ガベージコレクションの統計情報を集めます。

  garbage に回収できない独自の型のオブジェクトが現れたとき、アプリケーションがそれらを特定し消去できるようにする。

  バージョン 3.3 で追加.

以下は *set_debug()* に指定することのできる定数です:

gc.DEBUG_STATS
: 検出中に統計情報を出力します。この情報は、検出頻度を最適化する際に有益です。

gc.DEBUG_COLLECTABLE
: 見つかった回収可能オブジェクトの情報を出力します。

gc.DEBUG_UNCOLLECTABLE
: 見つかった回収不能オブジェクト（到達不能だが、ガベージコレクションで解放する事ができないオブジェクト）の情報を出力します。回収不能オブジェクトは、garbage リストに追加されます。

  バージョン 3.2 で変更: **インタプリタシャットダウン** 時に *garbage* リストが空でない場合に、その中身の出力も行います。

gc.DEBUG_SAVEALL
: 設定されている場合、全ての到達不能オブジェクトは解放されずに *garbage* に追加されます。これはプログラムのメモリリークをデバッグするときに便利です。

gc.DEBUG_LEAK
: プログラムのメモリリークをデバッグするときに指定します（ DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_SAVEALL と同じ）。

29.13 inspect --- 活動中のオブジェクトの情報を取得する

ソースコード: Lib/inspect.py

---

inspect は、活動中のオブジェクト (モジュール、クラス、メソッド、関数、トレースバック、フレームオブジェクト、コードオブジェクトなど) から情報を取得する関数を定義しており、クラスの内容を調べたり、メソッドのソースコードを取得したり、関数の引数リストを取り出して整形したり、詳細なトレースバックを表示するのに必要な情報を取得したりするために利用できます。

このモジュールの機能は 4 種類に分類することができます。型チェック、ソースコードの情報取得、クラスや関数からの情報取得、インタープリタのスタック情報の調査です。

29.13.1 型とメンバー

getmembers() は、クラスやモジュールなどのオブジェクトからメンバーを取得します。名前が "is" で始まる関数は、主に *getmembers()* の第 2 引数として利用するために提供されています。以下のような特殊属性を参照できるかどうか調べる時にも使えるでしょう:

型	属性	説明
モジュール	__doc__	ドキュメント文字列
	__file__	ファイル名 (組み込みモジュールには存在しません)
クラス	__doc__	ドキュメント文字列
	__name__	クラスの定義名
	__qualname__	qualified name
	__module__	クラスを定義しているモジュールの名前
メソッド	__doc__	ドキュメント文字列
	__name__	メソッドの定義名
	__qualname__	qualified name
	__func__	メソッドを実装している関数オブジェクト
	__self__	メソッドに結合しているインスタンス、または None
	__module__	name of module in which this method was defined
関数	__doc__	ドキュメント文字列
	__name__	関数の定義名
	__qualname__	qualified name
	__code__	関数をコンパイルした バイトコード を格納するコードオブジェクト
	__defaults__	位置またはキーワード引数の全ての既定値のタプル
	__kwdefaults__	キーワード専用引数の全ての既定値のマッピング
	__globals__	関数が定義されたグローバル名前空間
	__annotations__	仮引数名からアノテーションへのマッピング; "return" キーは return アノテーショ
	__module__	name of module in which this function was defined
traceback	tb_frame	このレベルのフレームオブジェクト

表 1 – 前のページからの続き

型	属性	説明
	tb_lasti	最後に実行しようとしたバイトコード中のインストラクションを示すインデックス
	tb_lineno	現在の Python ソースコードの行番号
	tb_next	このオブジェクトの内側 (このレベルから呼び出された) のトレースバックオブジェ
フレーム	f_back	外側 (このフレームを呼び出した) のフレームオブジェクト
	f_builtins	このフレームで参照している組み込み名前空間
	f_code	このフレームで実行しているコードオブジェクト
	f_globals	このフレームで参照しているグローバル名前空間
	f_lasti	最後に実行しようとしたバイトコード中のインストラクションを示すインデックス
	f_lineno	現在の Python ソースコードの行番号
	f_locals	このフレームで参照しているローカル名前空間
	f_trace	このフレームのトレース関数、または None
コード	co_argcount	number of arguments (not including keyword only arguments, * or ** args)
	co_code	コンパイルされたバイトコードそのままの文字列
	co_cellvars	tuple of names of cell variables (referenced by containing scopes)
	co_consts	バイトコード中で使用している定数のタプル
	co_filename	コードオブジェクトを生成したファイルのファイル名
	co_firstlineno	Python ソースコードの先頭行
	co_flags	bitmap of CO_* flags, read more *here*
	co_lnotab	行番号からバイトコードインデックスへの変換表を文字列にエンコードしたもの
	co_freevars	tuple of names of free variables (referenced via a function's closure)
	co_posonlyargcount	number of positional only arguments
	co_kwonlyargcount	number of keyword only arguments (not including ** arg)
	co_name	コードオブジェクトが定義されたときの名前
	co_names	ローカル変数名のタプル
	co_nlocals	ローカル変数の数
	co_stacksize	必要とされる仮想マシンのスタックスペース
	co_varnames	引数名とローカル変数名のタプル
ジェネレータ	__name__	名前
	__qualname__	qualified name
	gi_frame	フレーム
	gi_running	ジェネレータが実行中かどうか
	gi_code	コード
	gi_yieldfrom	yield from でイテレートされているオブジェクト、または None
コルーチン	__name__	名前
	__qualname__	qualified name
	cr_await	待機されているオブジェクト、または None
	cr_frame	フレーム
	cr_running	コルーチンが実行中かどうか
	cr_code	コード
	cr_origin	where coroutine was created, or None. See *sys.set_coroutine_origin_tracki*

表 1 – 前のページからの続き

型	属性	説明
組み込み	___doc___	ドキュメント文字列
	___name___	関数、メソッドの元々の名前
	___qualname___	qualified name
	___self___	メソッドが結合しているインスタンス、または None

バージョン 3.5 で変更: ジェネレータに `__qualname__` と `gi_yieldfrom` 属性が追加されました。

ジェネレータの `__name__` 属性がコード名ではなく関数名から設定されるようになり、変更できるようになりました。

バージョン 3.7 で変更: Add `cr_origin` attribute to coroutines.

`inspect.getmembers`(*object*[, *predicate*])

Return all the members of an object in a list of `(name, value)` pairs sorted by name. If the optional *predicate* argument—which will be called with the `value` object of each member—is supplied, only members for which the predicate returns a true value are included.

注釈: 引数がクラスで属性がメタクラスのカスタム `__dir__()` に列挙されている場合、*getmembers()* はメタクラスで定義されたクラス属性のみを返します。

`inspect.getmodulename`(*path*)

Return the name of the module named by the file *path*, without including the names of enclosing packages. The file extension is checked against all of the entries in *importlib.machinery.all_suffixes()*. If it matches, the final path component is returned with the extension removed. Otherwise, `None` is returned.

Note that this function *only* returns a meaningful name for actual Python modules - paths that potentially refer to Python packages will still return `None`.

バージョン 3.3 で変更: The function is based directly on *importlib*.

`inspect.ismodule`(*object*)

オブジェクトがモジュールの場合 `True` を返します。

`inspect.isclass`(*object*)

オブジェクトが組み込みか Python が生成したクラスの場合に `True` を返します。

`inspect.ismethod`(*object*)

オブジェクトがメソッドの場合 `True` を返します。

`inspect.isfunction`(*object*)

オブジェクトが、*lambda* 式で生成されたものを含む Python 関数の場合に `True` を返します。

`inspect.isgeneratorfunction`(*object*)

オブジェクトが Python のジェネレータ関数の場合 `True` を返します。

バージョン 3.8 で変更: Functions wrapped in *functools.partial()* now return `True` if the wrapped function is a Python generator function.

`inspect.isgenerator`(*object*)
オブジェクトがジェネレータの場合 `True` を返します。

`inspect.iscoroutinefunction`(*object*)
オブジェクトが **コルーチン関数** (`async def` シンタックスで定義された関数) の場合 `True` を返します。

バージョン 3.5 で追加.

バージョン 3.8 で変更: Functions wrapped in *functools.partial()* now return `True` if the wrapped function is a *coroutine function*.

`inspect.iscoroutine`(*object*)
オブジェクトが `async def` で生成された **コルーチン** の場合 `True` を返します。

バージョン 3.5 で追加.

`inspect.isawaitable`(*object*)
オブジェクトを `await` 式内で使用できる場合 `True` を返します。

ジェネレータベースのコルーチンと通常のジェネレータを区別するのに使うこともできます。

```
def gen():
    yield
@types.coroutine
def gen_coro():
    yield

assert not isawaitable(gen())
assert isawaitable(gen_coro())
```

バージョン 3.5 で追加.

`inspect.isasyncgenfunction`(*object*)
オブジェクトが *asynchronous generator* 関数の場合に `True` を返します。例:

```
>>> async def agen():
...     yield 1
...
>>> inspect.isasyncgenfunction(agen)
True
```

バージョン 3.6 で追加.

バージョン 3.8 で変更: Functions wrapped in *functools.partial()* now return `True` if the wrapped function is a *asynchronous generator* function.

`inspect.isasyncgen`(*object*)
オブジェクトが *asynchronous generator* 関数によって生成された *asynchronous generator iterator*

である場合に True を返します。

バージョン 3.6 で追加.

inspect.**istraceback**(*object*)
オブジェクトがトレースバックの場合は True を返します。

inspect.**isframe**(*object*)
オブジェクトがフレームの場合は True を返します。

inspect.**iscode**(*object*)
オブジェクトがコードの場合は True を返します。

inspect.**isbuiltin**(*object*)
オブジェクトが組み込み関数か束縛済みの組み込みメソッドの場合に True を返します。

inspect.**isroutine**(*object*)
オブジェクトがユーザ定義か組み込みの関数またはメソッドの場合は True を返します。

inspect.**isabstract**(*object*)
オブジェクトが抽象基底クラスであるときに True を返します。

inspect.**ismethoddescriptor**(*object*)
オブジェクトがメソッドデスクリプタであり、*ismethod()*, *isclass()*, *isfunction()*, *isbuiltin()* でない場合に True を返します。

これは例えば int.__add__ で真になります。このテストをパスするオブジェクトは __get__() メソッドを持ちますが __set__() メソッドを持ちません。それ以外の属性を持っているかもしれません。通常 *__name__* 属性を持っていますし、たいていは __doc__ も持っています。

デスクリプタを使って実装されたメソッドで、上記のいずれかのテストもパスしているものは、*ismethoddescriptor()* では False を返します。これは単に他のテストの方がもっと確実だからです -- 例えば、*ismethod()* をパスしたオブジェクトは __func__ 属性 (など) を持っていると期待できます。

inspect.**isdatadescriptor**(*object*)
オブジェクトがデータデスクリプタの場合に True を返します。

データデスクリプタは __set__ または __delete__ 属性を持ちます。データデスクリプタの例は (Python 上で定義された) プロパティや getset やメンバーです。後者のふたつは C で定義されており、個々の型に特有のテストも行います。そのため、Python の実装よりもより確実です。通常、データデスクリプタは *__name__* や __doc__ 属性を持ちます (プロパティ、getset 、メンバーは両方の属性を持っています) が、保証されているわけではありません。

inspect.**isgetsetdescriptor**(*object*)
オブジェクトが getset デスクリプタの場合に True を返します。

CPython implementation detail: getset とは、拡張モジュールで PyGetSetDef 構造体を用いて定義された属性のことです。そのような型を持たない Python 実装の場合は、このメソッドは常に False を返します。

inspect.ismemberdescriptor(*object*)
オブジェクトがメンバーデスクリプタの場合に True を返します。

CPython implementation detail: メンバーデスクリプタとは、拡張モジュールで PyMemberDef 構造体を用いて定義された属性のことです。そのような型を持たない Python 実装の場合は、このメソッドは常に False を返します。

29.13.2 ソースコードの情報取得

inspect.getdoc(*object*)
Get the documentation string for an object, cleaned up with *cleandoc()*. If the documentation string for an object is not provided and the object is a class, a method, a property or a descriptor, retrieve the documentation string from the inheritance hierarchy.

バージョン 3.5 で変更: ドキュメンテーション文字列がオーバーライドされていない場合は継承されるようになりました。

inspect.getcomments(*object*)
Return in a single string any lines of comments immediately preceding the object's source code (for a class, function, or method), or at the top of the Python source file (if the object is a module). If the object's source code is unavailable, return None. This could happen if the object has been defined in C or the interactive shell.

inspect.getfile(*object*)
オブジェクトを定義している (テキストまたはバイナリの) ファイルの名前を返します。オブジェクトが組み込みモジュール、クラス、関数の場合は *TypeError* 例外が発生します。

inspect.getmodule(*object*)
オブジェクトを定義しているモジュールを推測します。

inspect.getsourcefile(*object*)
オブジェクトを定義している Python ソースファイルの名前を返します。オブジェクトが組み込みのモジュール、クラス、関数の場合には、*TypeError* 例外が発生します。

inspect.getsourcelines(*object*)
オブジェクトのソース行のリストと開始行番号を返します。引数にはモジュール、クラス、メソッド、関数、トレースバック、フレーム、コードオブジェクトを指定することができます。戻り値は指定したオブジェクトに対応するソースコードのソース行リストと元のソースファイル上での開始行となります。ソースコードを取得できない場合は *OSError* が発生します。

バージョン 3.3 で変更: *IOError* の代わりに *OSError* を送出します。前者は後者のエイリアスです。

inspect.getsource(*object*)
オブジェクトのソースコードを返します。引数にはモジュール、クラス、メソッド、関数、トレースバック、フレーム、コードオブジェクトを指定することができます。ソースコードは単一の文字列で返します。ソースコードを取得できない場合は *OSError* が発生します。

バージョン 3.3 で変更: *IOError* の代わりに *OSError* を送出します。前者は後者のエイリアスです。

inspect.cleandoc(*doc*)

コードブロックと位置を合わせるためのインデントを docstring から削除します。

先頭行の行頭の空白文字は全て削除されます。2 行目以降では全行で同じ数の行頭の空白文字が、削除できるだけ削除されます。その後、先頭と末尾の空白行が削除され、全てのタブが空白に展開されます。

29.13.3 Signature オブジェクトで呼び出し可能オブジェクトを内省する

バージョン 3.3 で追加.

The Signature object represents the call signature of a callable object and its return annotation. To retrieve a Signature object, use the *signature()* function.

inspect.signature(*callable*, *, *follow_wrapped=True*)

与えられた callable の *Signature* オブジェクトを返します:

```
>>> from inspect import signature
>>> def foo(a, *, b:int, **kwargs):
...     pass

>>> sig = signature(foo)

>>> str(sig)
'(a, *, b:int, **kwargs)'

>>> str(sig.parameters['b'])
'b:int'

>>> sig.parameters['b'].annotation
<class 'int'>
```

単純な関数やクラスから、*functools.partial()* オブジェクトまで、幅広い Python の呼び出し可能なオブジェクトを受け付けます。

シグネチャが提供できない場合は、*ValueError* を送出し、オブジェクトの型がサポートされない場合は、*TypeError* を送出します。

A slash(/) in the signature of a function denotes that the parameters prior to it are positional-only. For more info, see the FAQ entry on positional-only parameters.

バージョン 3.5 で追加: follow_wrapped parameter. Pass False to get a signature of callable specifically (callable.__wrapped__ will not be used to unwrap decorated callables.)

注釈: Some callables may not be introspectable in certain implementations of Python. For example, in CPython, some built-in functions defined in C provide no metadata about their arguments.

---

class inspect.Signature(*parameters=None*, *, *return_annotation=Signature.empty*)

A Signature object represents the call signature of a function and its return annotation. For each

parameter accepted by the function it stores a *Parameter* object in its *parameters* collection.

The optional *parameters* argument is a sequence of *Parameter* objects, which is validated to check that there are no parameters with duplicate names, and that the parameters are in the right order, i.e. positional-only first, then positional-or-keyword, and that parameters with defaults follow parameters without defaults.

The optional *return_annotation* argument, can be an arbitrary Python object, is the "return" annotation of the callable.

Signature オブジェクトは **イミュータブル** です。変更されたコピーを作成するには *Signature.replace()* を使用してください。

バージョン 3.5 で変更: Signature オブジェクトがピックル並びにハッシュ可能になりました。

empty

return アノテーションがないことを指すクラスレベルの特殊マーカです。

parameters

An ordered mapping of parameters' names to the corresponding *Parameter* objects. Parameters appear in strict definition order, including keyword-only parameters.

バージョン 3.7 で変更: Python only explicitly guaranteed that it preserved the declaration order of keyword-only parameters as of version 3.7, although in practice this order had always been preserved in Python 3.

return_annotation

呼び出し可能オブジェクトの "return" アノテーションです。呼び出し可能オブジェクトに "return" アノテーションがない場合、この属性は *Signature.empty* に設定されます。

bind(**args*, ***kwargs*)

Create a mapping from positional and keyword arguments to parameters. Returns *BoundArguments* if `*args` and `**kwargs` match the signature, or raises a *TypeError*.

bind_partial(**args*, ***kwargs*)

Works the same way as *Signature.bind()*, but allows the omission of some required arguments (mimics *functools.partial()* behavior.) Returns *BoundArguments*, or raises a *TypeError* if the passed arguments do not match the signature.

replace(**[, parameters][, return_annotation]*)

Create a new Signature instance based on the instance replace was invoked on. It is possible to pass different `parameters` and/or `return_annotation` to override the corresponding properties of the base signature. To remove return_annotation from the copied Signature, pass in *Signature.empty*.

```
>>> def test(a, b):
...     pass
>>> sig = signature(test)
>>> new_sig = sig.replace(return_annotation="new return anno")
```

(次のページに続く)

(前のページからの続き)

```
>>> str(new_sig)
"(a, b) -> 'new return anno'"
```

classmethod from_callable(*obj*, ***, *follow_wrapped=True*)

Return a *Signature* (or its subclass) object for a given callable `obj`. Pass `follow_wrapped=False` to get a signature of `obj` without unwrapping its `__wrapped__` chain.

このメソッドは *Signature* のサブクラス化を単純化します:

```
class MySignature(Signature):
    pass
sig = MySignature.from_callable(min)
assert isinstance(sig, MySignature)
```

バージョン 3.5 で追加.

class inspect.Parameter(*name*, *kind*, ***, *default=Parameter.empty*, *annotation=Parameter.empty*)

Parameter オブジェクトは **イミュータブル** です。変更されたコピーを作成するには *Parameter.replace()* を使用してください。

バージョン 3.5 で変更: Parameter オブジェクトがピックル並びにハッシュ可能になりました。

empty

デフォルト値とアノテーションがないことを指すクラスレベルの特殊マーカです。

name

仮引数名 (文字列) です。名前は有効な Python 識別子でなければなりません。

CPython implementation detail: CPython generates implicit parameter names of the form `.0` on the code objects used to implement comprehensions and generator expressions.

バージョン 3.6 で変更: These parameter names are exposed by this module as names like `implicit0`.

default

引数のデフォルト値です。引数にデフォルト値がない場合、この属性は *Parameter.empty* に設定されます。

annotation

引数のアノテーションです。引数にアノテーションがない場合、この属性は *Parameter.empty* に設定されます。

kind

実引数値がどのように仮引数に束縛されるかを記述します。有効な値 (*Parameter* を通じてアクセスできます、たとえば `Parameter.KEYWORD_ONLY`) は:

名前	意味
POSITIONAL_ONLY	Value must be supplied as a positional argument. Positional only parameters are those which appear before a / entry (if present) in a Python function definition.
POSITIONAL_OR_KEYWORD	値をキーワードまたは位置引数として渡すことができます (これは Python で実装された関数の標準的な束縛動作です)。
VAR_POSITIONAL	その他の仮引数に束縛されていない位置引数のタプルです。Python の関数定義における *args 仮引数に対応します。
KEYWORD_ONLY	値をキーワード引数として渡さなければなりません。キーワード専用引数は Python の関数定義において * や *args の後に現れる引数です。
VAR_KEYWORD	その他の仮引数に束縛されていないキーワード引数の辞書です。Python の関数定義における **kwargs 仮引数に対応します。

例: デフォルト値のない全てのキーワード専用引数を出力します:

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     if (param.kind == param.KEYWORD_ONLY and
...                         param.default is param.empty):
...         print('Parameter:', param)
Parameter: c
```

kind.description

Describes a enum value of Parameter.kind.

バージョン 3.8 で追加.

Example: print all descriptions of arguments:

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     print(param.kind.description)
positional or keyword
positional or keyword
keyword-only
keyword-only
```

replace(**[, name][, kind][, default][, annotation]*)

Create a new Parameter instance based on the instance replaced was invoked on. To override a *Parameter* attribute, pass the corresponding argument. To remove a default value or/and an annotation from a Parameter, pass *Parameter.empty*.

```
>>> from inspect import Parameter
>>> param = Parameter('foo', Parameter.KEYWORD_ONLY, default=42)
>>> str(param)
'foo=42'

>>> str(param.replace()) # Will create a shallow copy of 'param'
'foo=42'

>>> str(param.replace(default=Parameter.empty, annotation='spam'))
"foo:'spam'"
```

バージョン 3.4 で変更: Python 3.3 では、Parameter オブジェクトは `kind` が `POSITIONAL_ONLY` の場合 `None` に設定された `name` を持つことができました。これはもう許可されません。

class inspect.BoundArguments

Result of a *Signature.bind()* or *Signature.bind_partial()* call. Holds the mapping of arguments to the function's parameters.

arguments

An ordered, mutable mapping (*collections.OrderedDict*) of parameters' names to arguments' values. Contains only explicitly bound arguments. Changes in *arguments* will reflect in *args* and *kwargs*.

Should be used in conjunction with *Signature.parameters* for any argument processing purposes.

注釈: Arguments for which *Signature.bind()* or *Signature.bind_partial()* relied on a default value are skipped. However, if needed, use *BoundArguments.apply_defaults()* to add them.

---

args

位置引数の値のタプルです。*arguments* 属性から動的に計算されます。

kwargs

キーワード引数の値の辞書です。*arguments* 属性から動的に計算されます。

signature

親の *Signature* オブジェクトへの参照です。

apply_defaults()

存在しない引数のデフォルト値を設定します。

For variable-positional arguments (`*args`) the default is an empty tuple.

For variable-keyword arguments (**kwargs) the default is an empty dict.

```
>>> def foo(a, b='ham', *args): pass
>>> ba = inspect.signature(foo).bind('spam')
>>> ba.apply_defaults()
>>> ba.arguments
OrderedDict([('a', 'spam'), ('b', 'ham'), ('args', ())])
```

バージョン 3.5 で追加.

args および *kwargs* 属性を使用して関数を呼び出すことができます:

```
def test(a, *, b):
    ...

sig = signature(test)
ba = sig.bind(10, b=20)
test(*ba.args, **ba.kwargs)
```

参考:

PEP 362: - **関数シグニチャオブジェクト** The detailed specification, implementation details and examples.

29.13.4 クラスと関数

inspect.**getclasstree**(*classes*, *unique=False*)

リストで指定したクラスの継承関係から、ネストしたリストを作成します。ネストしたリストには、直前の要素から派生したクラスが格納されます。各要素は長さ 2 のタプルで、クラスと基底クラスのタプルを格納しています。*unique* が真の場合、各クラスは戻り値のリスト内に一つだけしか格納されません。真でなければ、多重継承を利用したクラスとその派生クラスは複数回格納される場合があります。

inspect.**getargspec**(*func*)

Get the names and default values of a Python function's parameters. A *named tuple* `ArgSpec(args, varargs, keywords, defaults)` is returned. *args* is a list of the parameter names. *varargs* and *keywords* are the names of the `*` and `**` parameters or `None`. *defaults* is a tuple of default argument values or `None` if there are no default arguments; if this tuple has *n* elements, they correspond to the last *n* elements listed in *args*.

バージョン 3.0 で非推奨: Use *getfullargspec()* for an updated API that is usually a drop-in replacement, but also correctly handles function annotations and keyword-only parameters.

Alternatively, use *signature()* and *Signature Object*, which provide a more structured introspection API for callables.

inspect.**getfullargspec**(*func*)

Get the names and default values of a Python function's parameters. A *named tuple* is returned:

FullArgSpec(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults, annotations)

args is a list of the positional parameter names. *varargs* is the name of the * parameter or None if arbitrary positional arguments are not accepted. *varkw* is the name of the ** parameter or None if arbitrary keyword arguments are not accepted. *defaults* is an *n*-tuple of default argument values corresponding to the last *n* positional parameters, or None if there are no such defaults defined. *kwonlyargs* is a list of keyword-only parameter names in declaration order. *kwonlydefaults* is a dictionary mapping parameter names from *kwonlyargs* to the default values used if no argument is supplied. *annotations* is a dictionary mapping parameter names to annotations. The special key "return" is used to report the function return value annotation (if any).

Note that *signature()* and *Signature Object* provide the recommended API for callable introspection, and support additional behaviours (like positional-only arguments) that are sometimes encountered in extension module APIs. This function is retained primarily for use in code that needs to maintain compatibility with the Python 2 inspect module API.

バージョン 3.4 で変更: This function is now based on *signature()*, but still ignores __wrapped__ attributes and includes the already bound first parameter in the signature output for bound methods.

バージョン 3.6 で変更: This method was previously documented as deprecated in favour of *signature()* in Python 3.5, but that decision has been reversed in order to restore a clearly supported standard interface for single-source Python 2/3 code migrating away from the legacy *getargspec()* API.

バージョン 3.7 で変更: Python only explicitly guaranteed that it preserved the declaration order of keyword-only parameters as of version 3.7, although in practice this order had always been preserved in Python 3.

inspect.**getargvalues**(*frame*)

指定したフレームに渡された引数の情報を取得します。戻り値は **名前付きタプル** ArgInfo(args, varargs, keywords, locals) です。*args* は引数名のリストです。*varargs* と *keywords* は * 引数と ** 引数の名前で、引数がなければ None となります。*locals* は指定したフレームのローカル変数の辞書です。

注釈: This function was inadvertently marked as deprecated in Python 3.5.

---

inspect.**formatargspec**(*args*[, *varargs*, *varkw*, *defaults*, *kwonlyargs*, *kwonlydefaults*, *annotations*[, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*, *formatreturns*, *formatannotations*]])

Format a pretty argument spec from the values returned by *getfullargspec()*.

The first seven arguments are (args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults, annotations).

The other six arguments are functions that are called to turn argument names, * argument name,

** argument name, default values, return annotation and individual annotations into strings, respectively.

例えば:

```
>>> from inspect import formatargspec, getfullargspec
>>> def f(a: int, b: float):
...     pass
...
>>> formatargspec(*getfullargspec(f))
'(a: int, b: float)'
```

バージョン 3.5 で非推奨: Use *signature()* and *Signature Object*, which provide a better introspecting API for callables.

inspect.**formatargvalues**(*args*[, *varargs*, *varkw*, *locals*, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*])

getargvalues() で取得した 4 つの値を読みやすく整形します。format* 引数はオプションで、名前と値を文字列に変換する整形関数を指定することができます。

注釈: This function was inadvertently marked as deprecated in Python 3.5.

---

inspect.**getmro**(*cls*)

cls クラスの基底クラス (*cls* 自身も含む) を、メソッドの優先順位順に並べたタプルを返します。結果のリスト内で各クラスは一度だけ格納されます。メソッドの優先順位はクラスの型によって異なります。非常に特殊なユーザ定義のメタクラスを使用していない限り、*cls* が戻り値の先頭要素となります。

inspect.**getcallargs**(*func*, */*, **args*, ***kwds*)

args と *kwds* を、Python の関数もしくはメソッド *func* を呼び出した場合と同じように引数名に束縛します。束縛済みメソッド (bound method) の場合、最初の引数 (慣習的に `self` という名前が付けられます) にも、関連づけられたインスタンスを束縛します。引数名 (`*` や `**` 引数が存在すればその名前も) に *args* と *kwds* からの値をマップした辞書を返します。*func* を正しく呼び出せない場合、つまり `func(*args, **kwds)` がシグネチャの不一致のために例外を投げるような場合には、それと同じ型で同じか似ているメッセージの例外を発生させます。例:

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
>>> getcallargs(f, 1, 2, 3) == {'a': 1, 'named': {}, 'b': 2, 'pos': (3,)}
True
>>> getcallargs(f, a=2, x=4) == {'a': 2, 'named': {'x': 4}, 'b': 1, 'pos': ()}
True
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() missing 1 required positional argument: 'a'
```

バージョン 3.2 で追加.

バージョン 3.5 で非推奨: Use *Signature.bind()* and *Signature.bind_partial()* instead.

inspect.**getclosurevars**(*func*)

Get the mapping of external name references in a Python function or method *func* to their current values. A *named tuple* `ClosureVars(nonlocals, globals, builtins, unbound)` is returned. *nonlocals* maps referenced names to lexical closure variables, *globals* to the function's module globals and *builtins* to the builtins visible from the function body. *unbound* is the set of names referenced in the function that could not be resolved at all given the current module globals and builtins.

func が Python の関数やメソッドでない場合 *TypeError* が送出されます。

バージョン 3.3 で追加.

inspect.**unwrap**(*func*, **, stop=None*)

Get the object wrapped by *func*. It follows the chain of `__wrapped__` attributes returning the last object in the chain.

stop is an optional callback accepting an object in the wrapper chain as its sole argument that allows the unwrapping to be terminated early if the callback returns a true value. If the callback never returns a true value, the last object in the chain is returned as usual. For example, *signature()* uses this to stop unwrapping if any object in the chain has a `__signature__` attribute defined.

ValueError is raised if a cycle is encountered.

バージョン 3.4 で追加.

29.13.5 インタープリタスタック

When the following functions return "frame records," each record is a *named tuple* `FrameInfo(frame, filename, lineno, function, code_context, index)`. The tuple contains the frame object, the filename, the line number of the current line, the function name, a list of lines of context from the source code, and the index of the current line within that list.

バージョン 3.5 で変更: タプルではなく名前付きタプルを返します。

注釈: フレームレコードの最初の要素などのフレームオブジェクトへの参照を保存すると、循環参照になってしまう場合があります。循環参照ができると、Python の循環参照検出機能を有効にしていたとしても関連するオブジェクトが参照しているすべてのオブジェクトが解放されにくくなり、明示的に参照を削除しないとメモリ消費量が増大する恐れがあります。

参照の削除を Python の循環参照検出機能にまかせることもできますが、`finally` 節で循環参照を解除すれば確実にフレーム (とそのローカル変数) は削除されます。また、循環参照検出機能は Python のコンパイルオプションや *gc.disable()* で無効とされている場合がありますので注意が必要です。例:

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
```

(次のページに続く)

(前のページからの続き)

```
    try:
        # do something with the frame
    finally:
        del frame
```

If you want to keep the frame around (for example to print a traceback later), you can also break reference cycles by using the `frame.clear()` method.

以下の関数でオプション引数 *context* には、戻り値のソース行リストに何行分のソースを含めるかを指定します。ソース行リストには、実行中の行を中心として指定された行数分のリストを返します。

`inspect.getframeinfo`(*frame*, *context=1*)

フレームまたはトレースバックオブジェクトの情報を取得します。名前付きタプル `Traceback(filename, lineno, function, code_context, index)` が返されます。

`inspect.getouterframes`(*frame*, *context=1*)

指定したフレームと、その外側の全フレームのフレームレコードを返します。外側のフレームとは *frame* が生成されるまでのすべての関数呼び出しを示します。戻り値のリストの先頭は *frame* のフレームレコードで、末尾の要素は *frame* のスタックにある最も外側のフレームのフレームレコードとなります。

バージョン 3.5 で変更: 名前付きタプル `FrameInfo(frame, filename, lineno, function, code_context, index)` のリストが返されます。

`inspect.getinnerframes`(*traceback*, *context=1*)

指定したフレームと、その内側の全フレームのフレームレコードを返します。内のフレームとは *frame* から続く一連の関数呼び出しを示します。戻り値のリストの先頭は *traceback* のフレームレコードで、末尾の要素は例外が発生した位置を示します。

バージョン 3.5 で変更: 名前付きタプル `FrameInfo(frame, filename, lineno, function, code_context, index)` のリストが返されます。

`inspect.currentframe`()

呼び出し元のフレームオブジェクトを返します。

CPython implementation detail: この関数はインタプリタの Python スタックフレームサポートに依存します。これは Python のすべての実装に存在している保証はありません。Python スタックフレームサポートのない環境では、この関数は `None` を返します。

`inspect.stack`(*context=1*)

呼び出し元スタックのフレームレコードのリストを返します。最初の要素は呼び出し元のフレームレコードで、末尾の要素はスタックにある最も外側のフレームのフレームレコードとなります。

バージョン 3.5 で変更: 名前付きタプル `FrameInfo(frame, filename, lineno, function, code_context, index)` のリストが返されます。

`inspect.trace`(*context=1*)

実行中のフレームと処理中の例外が発生したフレームの間のフレームレコードのリストを返します。最

初の要素は呼び出し元のフレームレコードで、末尾の要素は例外が発生した位置を示します。

バージョン 3.5 で変更: 名前付きタプル `FrameInfo(frame, filename, lineno, function, code_context, index)` のリストが返されます。

29.13.6 属性の静的なフェッチ

Both *getattr()* and *hasattr()* can trigger code execution when fetching or checking for the existence of attributes. Descriptors, like properties, will be invoked and `__getattr__()` and `__getattribute__()` may be called.

For cases where you want passive introspection, like documentation tools, this can be inconvenient. *getattr_static()* has the same signature as *getattr()* but avoids executing code when it fetches attributes.

`inspect.getattr_static(obj, attr, default=None)`

Retrieve attributes without triggering dynamic lookup via the descriptor protocol, `__getattr__()` or `__getattribute__()`.

Note: this function may not be able to retrieve all attributes that getattr can fetch (like dynamically created attributes) and may find attributes that getattr can't (like descriptors that raise AttributeError). It can also return descriptors objects instead of instance members.

If the instance *__dict__* is shadowed by another member (for example a property) then this function will be unable to find instance members.

バージョン 3.2 で追加.

getattr_static() does not resolve descriptors, for example slot descriptors or getset descriptors on objects implemented in C. The descriptor object is returned instead of the underlying attribute.

You can handle these with code like the following. Note that for arbitrary getset descriptors invoking these may trigger code execution:

```
# example code for resolving the builtin descriptor types
class _foo:
    __slots__ = ['foo']

slot_descriptor = type(_foo.foo)
getset_descriptor = type(type(open(__file__)).name)
wrapper_descriptor = type(str.__dict__['__add__'])
descriptor_types = (slot_descriptor, getset_descriptor, wrapper_descriptor)

result = getattr_static(some_object, 'foo')
if type(result) in descriptor_types:
    try:
        result = result.__get__()
    except AttributeError:
        # descriptors can raise AttributeError to
```

(次のページに続く)

(前のページからの続き)

```
        # indicate there is no underlying value
        # in which case the descriptor itself will
        # have to do
        pass
```

29.13.7 ジェネレータおよびコルーチンの現在の状態

When implementing coroutine schedulers and for other advanced uses of generators, it is useful to determine whether a generator is currently executing, is waiting to start or resume or execution, or has already terminated. *getgeneratorstate()* allows the current state of a generator to be determined easily.

`inspect.getgeneratorstate`(*generator*)

ジェネレータイテレータの現在の状態を取得します。

取り得る状態は:

- GEN_CREATED: 実行開始を待機しています。
- GEN_RUNNING: インタープリタによって現在実行されています。
- GEN_SUSPENDED: yield 式で現在サスペンドされています。
- GEN_CLOSED: 実行が完了しました。

バージョン 3.2 で追加.

`inspect.getcoroutinestate`(*coroutine*)

Get current state of a coroutine object. The function is intended to be used with coroutine objects created by `async def` functions, but will accept any coroutine-like object that has `cr_running` and `cr_frame` attributes.

取り得る状態は:

- CORO_CREATED: 実行開始を待機しています。
- CORO_RUNNING: インタープリタにより現在実行中です。
- CORO_SUSPENDED: await 式により現在停止中です。
- CORO_CLOSED: 実行が完了しました。

バージョン 3.5 で追加.

ジェネレータの現在の内部状態を問い合わせることも出来ます。これは主に内部状態が期待通り更新されているかどうかを確認するためのテストの目的に有用です。

`inspect.getgeneratorlocals`(*generator*)

Get the mapping of live local variables in *generator* to their current values. A dictionary is returned

that maps from variable names to values. This is the equivalent of calling *locals()* in the body of the generator, and all the same caveats apply.

If *generator* is a *generator* with no currently associated frame, then an empty dictionary is returned. *TypeError* is raised if *generator* is not a Python generator object.

CPython implementation detail: This function relies on the generator exposing a Python stack frame for introspection, which isn't guaranteed to be the case in all implementations of Python. In such cases, this function will always return an empty dictionary.

バージョン 3.3 で追加.

`inspect.getcoroutinelocals(`*coroutine*`)`

This function is analogous to *getgeneratorlocals()*, but works for coroutine objects created by `async def` functions.

バージョン 3.5 で追加.

29.13.8 Code Objects Bit Flags

Python code objects have a `co_flags` attribute, which is a bitmap of the following flags:

`inspect.CO_OPTIMIZED`

The code object is optimized, using fast locals.

`inspect.CO_NEWLOCALS`

If set, a new dict will be created for the frame's `f_locals` when the code object is executed.

`inspect.CO_VARARGS`

The code object has a variable positional parameter (`*args`-like).

`inspect.CO_VARKEYWORDS`

The code object has a variable keyword parameter (`**kwargs`-like).

`inspect.CO_NESTED`

The flag is set when the code object is a nested function.

`inspect.CO_GENERATOR`

The flag is set when the code object is a generator function, i.e. a generator object is returned when the code object is executed.

`inspect.CO_NOFREE`

The flag is set if there are no free or cell variables.

`inspect.CO_COROUTINE`

The flag is set when the code object is a coroutine function. When the code object is executed it returns a coroutine object. See **PEP 492** for more details.

バージョン 3.5 で追加.

`inspect.CO_ITERABLE_COROUTINE`
: The flag is used to transform generators into generator-based coroutines. Generator objects with this flag can be used in `await` expression, and can `yield from` coroutine objects. See **PEP 492** for more details.

  バージョン 3.5 で追加.

`inspect.CO_ASYNC_GENERATOR`
: The flag is set when the code object is an asynchronous generator function. When the code object is executed it returns an asynchronous generator object. See **PEP 525** for more details.

  バージョン 3.6 で追加.

注釈: The flags are specific to CPython, and may not be defined in other Python implementations. Furthermore, the flags are an implementation detail, and can be removed or deprecated in future Python releases. It's recommended to use public APIs from the *inspect* module for any introspection needs.

29.13.9 コマンドラインインターフェイス

The *inspect* module also provides a basic introspection capability from the command line.

By default, accepts the name of a module and prints the source of that module. A class or function within the module can be printed instead by appended a colon and the qualified name of the target object.

`--details`
: Print information about the specified object rather than the source code

29.14 site --- サイト固有の設定フック

ソースコード: Lib/site.py

このモジュールは初期化中に自動的にインポートされます。自動インポートはインタプリタの `-S` オプションで禁止できます。

このモジュールをインポートすると、`-S` オプションを使わない限り、サイト固有のパスをモジュール検索パスに追加し、いくつかの組み込み関数を追加します。`-S` オプションを使った場合、このモジュールはモジュール検索パスの変更や組み込み関数の追加を自動で行うことはなく、安全にインポートできます。通常のサイト固有の追加処理を明示的に起動するには、*site.main()* 関数を呼んでください。

バージョン 3.3 で変更: 以前は `-S` を使っているときでも、モジュールをインポートするとパス変更が起動されていました。

site.main() 関数の処理は、前部と後部からなる最大で四つまでのディレクトリを構築するところから始まります。前部では sys.prefix と sys.exec_prefix を使用します; 空の前部は使われません。後部では、1 つ目は空文字列を使い、2 つ目は lib/site-packages (Windows) または lib/python*X.Y*/site-packages (Unix と Macintosh) を使います。前部-後部の異なる組み合わせごとに、それが存在しているディレクトリを参照しているかどうかを調べ、存在している場合は sys.path へ追加します。そして、新しく追加されたパスからパス設定ファイルを検索します。

バージョン 3.5 で変更: "site-python" ディレクトリのサポートは削除されました。

"pyvenv.cfg" という名前のファイルが上で挙げたディレクトリの 1 つに存在していた場合、sys.executable, sys.prefix, sys.exec_prefix にはそのディレクトリが設定され、site-packages もチェックします (sys.base_prefix と sys.base_exec_prefix は常にインストールされている Python の " 実際の" プレフィックスです)。(ブートストラップの設定ファイルである) "pyvenv.cfg" で、キー "include-system-site-packages" に "true" (大文字小文字は区別しない) 以外が設定されている場合は、site-packages を探しにシステムレベルのプレフィックスも見に行きません; そうでない場合は見に行きます。

パス設定ファイルは *name*.pth という形式の名前をもつファイルで、上の 4 つのディレクトリのひとつにあります。その内容は sys.path に追加される追加項目 (一行に一つ) です。存在しない項目は sys.path へは決して追加されませんが、項目がファイルではなくディレクトリを参照しているかどうかはチェックされません。項目が sys.path へ二回以上追加されることはありません。空行と # で始まる行は読み飛ばされます。import で始まる (そしてその後ろにスペースかタブが続く) 行は実行されます。

注釈: .pth ファイル内の実行可能な行は、特定のモジュールが実際に使用されるかどうかに関係なく、Python の起動時に毎回実行されます。したがって、その影響は最小限に抑えられるべきです。実行可能な行の主な目的は、対応するモジュールをインポート可能にすることです (サードパーティ製インポートフックのロード、PATH の調整など)。その他の初期化は、モジュールが実際にインポートされたときに行われます。コードのチャンクを 1 行に制限することは、より複雑なものをここに入れないようにするための意図的な手段です。

例えば、sys.prefix と sys.exec_prefix が /usr/local に設定されていると仮定します。そのとき Python X.Y ライブラリは /usr/local/lib/python*X.Y* にインストールされています。ここにはサブディレクトリ /usr/local/lib/python*X.Y*/site-packages があり、その中に三つのサブディレクトリ foo, bar および spam と二つのパス設定ファイル foo.pth と bar.pth をもつと仮定します。foo.pth には以下のものが記載されていると想定してください:

```
# foo package configuration

foo
bar
bletch
```

また、bar.pth には:

```
# bar package configuration

bar
```

が記載されているとします。そのとき、次のバージョンごとのディレクトリが sys.path へこの順番で追加されます:

```
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

bletch は存在しないため省略されるということに注意してください。bar ディレクトリは foo ディレクトリの前に来ます。なぜなら、bar.pth がアルファベット順で foo.pth の前に来るからです。また、spam はどちらのパス設定ファイルにも記載されていないため、省略されます。

これらのパス操作の後に、sitecustomize という名前のモジュールをインポートしようします。そのモジュールは任意のサイト固有のカスタマイゼーションを行うことができます。典型的にはこれはシステム管理者によって site-packages ディレクトリに作成されます。このインポートが *ImportError* またはそのサブクラス例外で失敗し、例外の name 属性が 'sitecustomize' に等しい場合は、何も表示せずに無視されます。Python が出力ストリームを利用できない状態で起動された場合、Windows では pythonw.exe (IDLE を開始するとデフォルトで使われます) のような、mod:*sitecustomize* から試みられた出力は無視されます。その他の例外は黙殺され、そしてそれはおそらく不可思議な失敗にみえるでしょう。

このあとで、*ENABLE_USER_SITE* が真であれば、任意のユーザ固有のカスタマイズを行うことが出来る usercustomize と名付けられたモジュールのインポートが試みられます。このファイルはユーザの site-packages ディレクトリ (下記参照) に作られることを意図していて、その場所はオプション -s によって無効にされない限りは sys.path に含まれます。このインポートが *ImportError* またはそのサブクラス例外で失敗し、例外の name 属性が 'usercustomize' と等しい場合、それは黙って無視されます。

いくつかの非 Unix システムでは、sys.prefix と sys.exec_prefix は空で、パス操作は省略されます。しかし、sitecustomize と usercustomize のインポートはそのときでも試みられます。

29.14.1 readline の設定

readline をサポートするシステムではこのモジュールは、Python を 対話モード で -S オプションなしで開始した場合に *rlcompleter* モジュールをインポートして設定します。デフォルトの振る舞いでは、タブ補完を有効にし、履歴保存ファイルに ~/.python_history を使います。これを無効にするには、あなたの sitecustomize または usercustomize あるいは PYTHONSTARTUP ファイル内で *sys.__interactivehook__* 属性を削除 (または上書き) してください。(---訳注: site モジュールは *__interactivehook__* に readline 設定関数を設定後に sitecustomize 等のユーザ設定を実行します。*__interactivehook__* のチェックが行われるのは site モジュール実行の後です。---)

バージョン 3.4 で変更: rlcompleter と history のアクティベーションが自動で行われるようになりました。

29.14.2 モジュールの内容

site.PREFIXES
: site パッケージディレクトリの prefix のリスト。

site.ENABLE_USER_SITE
: ユーザサイトディレクトリのステータスを示すフラグ。True の場合、ユーザサイトディレクトリが有効で sys.path に追加されていることを意味しています。False の場合、ユーザによるリクエスト (オプション -s か PYTHONNOUSERSITE) によって、None の場合セキュリティ上の理由 (ユーザまたはグループ ID と実効 ID の間のミスマッチ) あるいは管理者によって、ユーザサイトディレクトリが無効になっていることを示しています。ユーザサイトディレクトリのステータスを示すフラグ。True の場合、ユーザサイトディレクトリが有効で sys.path に追加されていることを意味しています。False の場合、ユーザによるリクエスト (オプション -s か PYTHONNOUSERSITE) によって、None の場合セキュリティ上の理由 (ユーザまたはグループ ID と実効 ID の間のミスマッチ) あるいは管理者によって、ユーザサイトディレクトリが無効になっていることを示しています。

site.USER_SITE
: Python 実行時のユーザの site-packages へのパスです。*getusersitepackages()* がまだ呼び出されていなければ None かもしれません。デフォルト値は UNIX と framework なしの Mac OS X ビルドでは ~/.local/lib/python*X.Y*/site-packages 、Mac framework ビルドでは ~/Library/Python/*X.Y*/lib/python/site-packages、Windows では *%APPDATA%*\Python\Python*XY*\site-packages です。このディレクトリは site ディレクトリなので、ここにいる .pth ファイルが処理されます。

site.USER_BASE
: ユーザの site-packages のベースとなるディレクトリへのパスです。*getuserbase()* がまだ呼び出されていなければ None かもしれません。デフォルト値は UNIX と framework なしの Mac OS X ビルドでは ~/.local 、Mac framework ビルドでは ~/Library/Python/*X.Y* 、Windows では *%APPDATA%*\Python です。この値は Distutils が、スクリプト、データファイル、Python モジュールなどのインストール先のディレクトリを user installation scheme で計算するのに使われます。PYTHONUSERBASE も参照してください。

site.main()
: モジュール検索パスに標準のサイト固有ディレクトリを追加します。この関数は、Python インタプリタが -S で開始されていない限り、このモジュールインポート時に自動的に呼び出されます。

  バージョン 3.3 で変更: この関数は以前は無条件に呼び出されていました。

site.addsitedir(*sitedir*, *known_paths=None*)
: sys.path にディレクトリを追加し、その .pth ファイル群を処理します。典型的には sitecustomize か usercustomize 内で使われます (上述)。

site.getsitepackages()
: 全てのグローバルな site-packages ディレクトリのリストを返します。

  バージョン 3.2 で追加.

site.getuserbase()
: ユーザのベースディレクトリへのパス *USER_BASE* を返します。未初期化であればこの関数は

PYTHONUSERBASE を参考にして、設定もします。

バージョン 3.2 で追加.

site.getusersitepackages()
ユーザ固有の site パッケージのディレクトリへのパス *USER_SITE* を返します。未初期化であればこの関数は *USER_BASE* を参考にして、設定もします。ユーザ固有の site パッケージが sys.path に追加されたかどうかを確認するには *ENABLE_USER_SITE* を使ってください。

バージョン 3.2 で追加.

29.14.3 コマンドラインインターフェイス

site モジュールはユーザディレクトリをコマンドラインから得る手段も提供しています:

```
$ python3 -m site --user-site
/home/user/.local/lib/python3.3/site-packages
```

引数なしで呼び出された場合、*sys.path* の中身を表示し、続けて *USER_BASE* とそのディレクトリが存在するかどうか、*USER_SITE* とそのディレクトリが存在するかどうか、最後に *ENABLE_USER_SITE* の値を、標準出力に出力します。

--user-base
ユーザのベースディレクトリを表示します。

--user-site
ユーザの site-packages ディレクトリを表示します。

両方のオプションが指定された場合、ユーザのベースとユーザの site が (常にこの順序で) *os.pathsep* 区切りで表示されます。

いずれかのオプションが与えられた場合に、このスクリプトは次のいずれかの終了コードで終了します: ユーザの site-packages が有効ならば 0 、ユーザにより無効にされていれば 1 、セキュリティ的な理由あるいは管理者によって無効にされている場合 2 、そして何かエラーがあった場合は 2 より大きな値。

参考:

PEP 370 -- ユーザごとの site-packages ディレクトリ

第

THIRTY

カスタム PYTHON インタプリタ

この章で解説されるモジュールで Python の対話インタプリタに似たインタフェースを書くことができます。もし Python そのもの以外に何か特殊な機能をサポートした Python インタプリタを作りたければ、*code* モジュールを参照してください。(*codeop* モジュールはより低レベルで、不完全 (かもしれない) Python コード断片のコンパイルをサポートするために使われます。)

この章で解説されるモジュールの完全な一覧は:

30.1 code --- インタプリタ基底クラス

ソースコード: Lib/code.py

---

code モジュールは read-eval-print (読み込み-評価-表示) ループを Python で実装するための機能を提供します。対話的なインタプリタプロンプトを提供するアプリケーションを作るために使える二つのクラスと便利な関数が含まれています。

class code.**InteractiveInterpreter**(*locals=None*)

このクラスは構文解析とインタプリタ状態 (ユーザの名前空間) を取り扱います。入力バッファリングやプロンプト出力、または入力ファイル指定を扱いません (ファイル名は常に明示的に渡されます)。オプションの *locals* 引数はその中でコードが実行される辞書を指定します。その初期値は、キー `'__name__'` が `'__console__'` に設定され、キー `'__doc__'` が `None` に設定された新しく作られた辞書です。

class code.**InteractiveConsole**(*locals=None*, *filename="<console>"*)

対話的な Python インタプリタの振る舞いを厳密にエミュレートします。このクラスは *InteractiveInterpreter* を元に作られていて、通常の `sys.ps1` と `sys.ps2` をつかったプロンプト出力と入力バッファリングが追加されています。

code.**interact**(*banner=None*, *readfunc=None*, *local=None*, *exitmsg=None*)

read-eval-print ループを実行するための便利な関数。これは *InteractiveConsole* の新しいインスタンスを作り、*readfunc* が与えられた場合は *InteractiveConsole.raw_input()* メソッドとして使われるように設定します。*local* が与えられた場合は、インタプリタループのデフォルト名前空間として使うために *InteractiveConsole* コンストラクタへ渡されます。そして、インスタンスの *interact()*

メソッドは（もし提供されていれば）見出しと終了メッセージして使うために *banner* と *exitmsg* を受け取り実行されます。コンソールオブジェクトは使われた後捨てられます。

バージョン 3.6 で変更: *exitmsg* 引数が追加されました.

code.**compile_command**(*source*, *filename="<input>"*, *symbol="single"*)

この関数は Python のインタプリタメインループ (別名、read-eval-print ループ) をエミュレートしようとするプログラムにとって役に立ちます。扱いにくい部分は、ユーザが (完全なコマンドや構文エラーではなく) さらにテキストを入力すれば完全になりうる不完全なコマンドを入力したときを決定することです。この関数は **ほとんど** の場合に実際のインタプリタメインループと同じ決定を行います。

source はソース文字列です。*filename* はオプションのソースが読み出されたファイル名で、デフォルトで `'<input>'` です。*symbol* はオプションの文法の開始記号で、`'single'` (デフォルト) または `'eval'` か `'exec'` にすべきです。

コマンドが完全で有効ならば、コードオブジェクトを返します (`compile(source, filename, symbol)` と同じ)。コマンドが完全でないならば、`None` を返します。コマンドが完全で構文エラーを含む場合は、*SyntaxError* を発生させます。または、コマンドが無効なリテラルを含む場合は、*OverflowError* もしくは *ValueError* を発生させます。

30.1.1 対話的なインタプリタオブジェクト

InteractiveInterpreter.**runsource**(*source*, *filename="<input>"*, *symbol="single"*)

インタプリタ内のあるソースをコンパイルし実行します。引数は *compile_command()* のものと同じです。*filename* のデフォルトは `'<input>'` で、*symbol* は `'single'` です。あるいくつかのことが起きる可能性があります:

- 入力が不正。*compile_command()* が例外 (*SyntaxError* か *OverflowError*) を起こした場合。*showsyntaxerror()* メソッドの呼び出によって、構文トレースバックが表示されるでしょう。*runsource()* は `False` を返します。
- 入力が完全でなく、さらに入力が必要。*compile_command()* が `None` を返した場合。*runsource()* は `True` を返します。
- 入力が完全。*compile_command()* がコードオブジェクトを返した場合。(*SystemExit* を除く実行時例外も処理する) *runcode()* を呼び出すことによって、コードは実行されます。*runsource()* は `False` を返します。

戻り値は、次の行のプロンプトに `sys.ps1` か `sys.ps2` のどちらを使うのか判断するために使えます。

InteractiveInterpreter.**runcode**(*code*)

コードオブジェクトを実行します。例外が生じたときは、トレースバックを表示するために *showtraceback()* が呼び出されます。伝搬することが許されている *SystemExit* を除くすべての例外が捉えられます。

KeyboardInterrupt についての注意。このコードの他の場所でこの例外が生じる可能性がありますし、常に捕らえることができるとは限りません。呼び出し側はそれを処理するために準備しておくべきです。

InteractiveInterpreter.**showsyntaxerror**(*filename=None*)

起きたばかりの構文エラーを表示します。複数の構文エラーに対して一つあるのではないため、これはスタックトレースを表示しません。*filename* が与えられた場合は、Python のパーサが与えるデフォルトのファイル名の代わりに例外の中へ入れられます。なぜなら、文字列から読み込んでいるときはパーサは常に `'<string>'` を使うからです。出力は *write()* メソッドによって書き込まれます。

InteractiveInterpreter.**showtraceback**()

起きたばかりの例外を表示します。スタックの最初の項目を取り除きます。なぜなら、それはインタプリタオブジェクトの実装の内部にあるからです。出力は *write()* メソッドによって書き込まれます。

バージョン 3.5 で変更: 最初のトレースバックではなく、完全なトレースバックの連鎖が表示されます。

InteractiveInterpreter.**write**(*data*)

文字列を標準エラーストリーム (`sys.stderr`) へ書き込みます。必要に応じて適切な出力処理を提供するために、派生クラスはこれをオーバーライドすべきです。

30.1.2 対話的なコンソールオブジェクト

InteractiveConsole クラスは *InteractiveInterpreter* のサブクラスです。以下の追加メソッドだけでなく、インタプリタオブジェクトのすべてのメソッドも提供します。

InteractiveConsole.**interact**(*banner=None*, *exitmsg=None*)

対話的な Python コンソールをそっくりにエミュレートします。オプションの *banner* 引数は最初のやりとりの前に表示するバナーを指定します。デフォルトでは、標準 Python インタプリタが表示するものと同じようなバナーを表示します。それに続けて、実際のインタプリタと混乱しないように (とても似ているから!) 括弧の中にコンソールオブジェクトのクラス名を表示します。

オプション引数の *exitmsg* は、終了時に出力される終了メッセージを指定します。空文字列を渡すと、出力メッセージを抑止します。もし、*exitmsg* が与えられないか、`None` の場合は、デフォルトのメッセージが出力されます。

バージョン 3.4 で変更: バナーの表示を抑制するには、空の文字列を渡してください。

バージョン 3.6 で変更: 終了時に、終了メッセージを表示します。

InteractiveConsole.**push**(*line*)

ソーステキストの一行をインタプリタへ送ります。その行の末尾に改行がついていてはいけません。内部に改行を持っているかもしれません。その行はバッファへ追加され、ソースとして連結された内容が渡されインタプリタの `runsource()` メソッドが呼び出されます。コマンドが実行されたか、有効であることをこれが示している場合は、バッファはリセットされます。そうでなければ、コマンドが不完全で、その行が付加された後のままバッファは残されます。さらに入力が必要ならば、戻り値は `True` です。その行がある方法で処理されたならば、`False` です (これは `runsource()` と同じです)。

InteractiveConsole.**resetbuffer**()

入力バッファから処理されていないソーステキストを取り除きます。

InteractiveConsole.**raw_input**(*prompt=""*)

プロンプトを書き込み、一行を読み込みます。返る行は末尾に改行を含みません。ユーザが EOF キー

シーケンスを入力したときは、*EOFError* を発生させます。基本実装では、sys.stdin から読み込みます。サブクラスはこれを異なる実装と置き換えるかもしれません。

30.2 codeop --- Python コードをコンパイルする

ソースコード: Lib/codeop.py

---

codeop モジュールは、*code* モジュールで行われているような Python の read-eval-print ループをエミュレートするユーティリティを提供します。そのため、このモジュールを直接利用する場面はあまり無いでしょう。プログラムにこのようなループを含めたい場合は、*code* モジュールの方が便利です。

この仕事には二つの部分があります:

1. 入力の一行が Python の文として完全であるかどうかを見分けられること: 簡単に言えば、次が '>>>' か、あるいは '...' かどうかを見分けます。
2. どの future 文をユーザが入力したのかを覚えていること。したがって、実質的にそれに続く入力をこれらとともにコンパイルすることができます。

codeop モジュールはこうしたことのそれぞれを行う方法とそれら両方を行う方法を提供します。

前者は実行するには:

codeop.**compile_command**(*source*, *filename="<input>"*, *symbol="single"*)

Python コードの文字列であるべき *source* をコンパイルしてみて、*source* が有効な Python コードの場合はコードオブジェクトを返します。このような場合、コードオブジェクトのファイル名属性は、デフォルトで '<input>' である *filename* でしょう。*source* が有効な Python コードでは **ない** が、有効な Python コードの接頭語である場合には、None を返します。

source に問題がある場合は、例外を発生させます。無効な Python 構文がある場合は、*SyntaxError* を発生させます。また、無効なリテラルがある場合は、*OverflowError* または *ValueError* を発生させます。

symbol 引数は *source* が文としてコンパイルされるか ('single' 、デフォルト) 、文の並びとしてか ('exec') 、または **式** としてコンパイルされるかを決定します ('eval')。他のどんな値も *ValueError* を発生させる原因となります。

注釈: ソースの終わりに達する前に、成功した結果をもってパーサは構文解析を止めることがあります。このような場合、後ろに続く記号はエラーとならずに無視されます。例えば、バックスラッシュの後ろに改行が 2 つあって、その後ろにゴミがあるかもしれません。パーサの API がより良くなればすぐに、この挙動は修正されるでしょう。

---

class codeop.**Compile**

このクラスのインスタンスは組み込み関数 *compile()* とシグネチャが一致する __call__() メソッドを持っていますが、インスタンスが *__future__* 文を含むプログラムテキストをコンパイルする場

合は、インスタンスは有効なその文とともに続くすべてのプログラムテキストを' 覚えていて' コンパイルするという違いがあります。

class codeop.**CommandCompiler**

このクラスのインスタンスは *compile_command()* とシグネチャが一致する `__call__()` メソッドを持っています。インスタンスが `__future__` 文を含むプログラムテキストをコンパイルする場合に、インスタンスは有効なその文とともにそれに続くすべてのプログラムテキストを' 覚えていて' コンパイルするという違いがあります。

第

THIRTYONE

モジュールのインポート

この章で解説されるモジュールは他の Python モジュールをインポートする新しい方法と、インポート処理をカスタマイズするためのフックを提供します。

この章で解説されるモジュールの完全な一覧は:

31.1 zipimport --- Zip アーカイブからモジュールを import する

ソースコード: Lib/zipimport.py

このモジュールは、Python モジュール (*.py, *.pyc) やパッケージを ZIP 形式のアーカイブから import できるようにします。通常、*zipimport* を明示的に使う必要はありません; 組み込みの import は、*sys.path* の要素が ZIP アーカイブへのパスを指している場合にこのモジュールを自動的に使います。

普通、*sys.path* はディレクトリ名の文字列からなるリストです。このモジュールを使うと、*sys.path* の要素に ZIP ファイルアーカイブを示す文字列を使えるようになります。ZIP アーカイブにはサブディレクトリ構造を含めることができ、パッケージの import をサポートさせたり、アーカイブ内のパスを指定してサブディレクトリ下から import を行わせたりできます。例えば、example.zip/lib/ のように指定すると、アーカイブ中の lib/ サブディレクトリ下だけから import を行います。

ZIP アーカイブ内にはどんなファイルを置いてもかまいませんが、インポートできるのは .py および .pyc ファイルだけです。動的モジュール (.pyd, .so) の ZIP インポートは行えません。アーカイブ内に .py ファイルしかない場合、Python は対応する .pyc ファイルを追加してアーカイブを変更しようとはしません。つまり、ZIP アーカイブ内に .pyc がない場合は、インポートが多少遅くなるかもしれないので注意してください。

バージョン 3.8 で変更: 以前は、アーカイブコメント付きの ZIP アーカイブはサポートされていませんでした。

参考:

PKZIP Application Note　ZIP ファイルフォーマットおよびアルゴリズムを作成した Phil Katz によるドキュメント。

PEP 273 - Zip **アーカイブからモジュールをインポートする**　このモジュールの実装も行った、James C. Ahlstrom による PEP です。Python 2.3 は **PEP 273** の仕様に従っていますが、Just van Rossum

の書いた import フックによる実装を使っています。インポートフックは **PEP 302** で解説されています。

PEP 302 - 新たなインポートフック　このモジュールを動作させる助けになっている import フックの追加を提案している PEP です。

このモジュールでは例外を一つ定義しています:

exception zipimport.ZipImportError
zipimporter オブジェクトが送出する例外です。*ImportError* のサブクラスなので、*ImportError* としても捕捉できます。

31.1.1 zipimporter オブジェクト

zipimporter は ZIP ファイルを import するためのクラスです。

class zipimport.zipimporter(*archivepath*)
新たな zipimporter インスタンスを生成します。*archivepath* は ZIP ファイルへのパスまたは ZIP ファイル中の特定のパスへのパスでなければなりません。たとえば、`foo/bar.zip/lib` という *archivepath* の場合、`foo/bar.zip` という ZIP ファイルの中の `lib` ディレクトリにあるモジュールを (存在するものとして) 検索します。

archivepath が有効な ZIP アーカイブを指していない場合、*ZipImportError* を送出します。

find_module(*fullname*[, *path*])
fullname で指定されたモジュールを検索します。*fullname* は完全に修飾された (ドット表記の) モジュール名でなければなりません。モジュールが見つかった場合には zipimporter インスタンス自体を返し、そうでない場合には *None* を返します。オプションの *path* 引数は無視されます --- この引数は importer プロトコルとの互換性を保つためのものです。

get_code(*fullname*)
fullname に指定したモジュールのコードオブジェクトを返します。モジュールがない場合には *ZipImportError* を送出します。

get_data(*pathname*)
pathname に関連付けられたデータを返します。該当するファイルが見つからなかった場合には *OSError* を送出します。

バージョン 3.3 で変更: 以前は *OSError* の代わりに *IOError* が送出されていました。

get_filename(*fullname*)
指定されたモジュールが import された場合、そのモジュールに設定した `__file__` の値を返します。モジュールが見つからない場合、*ZipImportError* を送出します。

バージョン 3.1 で追加.

get_source(*fullname*)
fullname で指定されたモジュールのソースコードを返します。モジュールが見つからない場合、

ZipImportError を送出します。アーカイブにはモジュールがあるもののソースコードがない場合、*None* を返します。

is_package(*fullname*)
: *fullname* で指定されたモジュールがパッケージの場合 True を返します。モジュールを見つけられない場合 *ZipImportError* を送出します。

load_module(*fullname*)
: *fullname* で指定されたモジュールをロードします。*fullname* は完全修飾された (ドット表記の) モジュール名でなければなりません。import 済みのモジュールを返します。モジュールがない場合には *ZipImportError* を送出します。

archive
: importer に関連付けられた ZIP ファイルのファイル名です。サブパスは含まれません。

prefix
: モジュールを検索する ZIP ファイル中のサブパスです。この文字列は ZIP ファイルのルートを指している zipimporter オブジェクトでは空です。

スラッシュでつなげると、*archive* と *prefix* 属性は *zipimporter* コンストラクタに渡された元々の *archivepath* 引数と等しくなります。

31.1.2 使用例

モジュールを ZIP アーカイブから import する例を以下に示します - *zipimport* モジュールが明示的に使われていないことに注意してください。

```
$ unzip -l example.zip
Archive:  example.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
     8467  11-26-02 22:30   jwzthreading.py
 --------                   -------
     8467                   1 file
$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, 'example.zip')  # Add .zip file to front of path
>>> import jwzthreading
>>> jwzthreading.__file__
'example.zip/jwzthreading.py'
```

31.2 pkgutil --- パッケージ拡張ユーティリティ

ソースコード: Lib/pkgutil.py

このモジュールはインポートシステムの、特にパッケージサポートに関するユーティリティです。

class pkgutil.**ModuleInfo**(*module_finder*, *name*, *ispkg*)
モジュールの概要情報を格納する namedtuple

バージョン 3.6 で追加.

pkgutil.**extend_path**(*path*, *name*)
パッケージを構成するモジュールの検索パスを拡張します。パッケージの `__init__.py` で次のように書くことを意図したものです:

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

上記はパッケージの `__path__` に `sys.path` の全ディレクトリのサブディレクトリとしてパッケージ名と同じ名前を追加します。これは 1 つの論理的なパッケージの異なる部品を複数のディレクトリに分けて配布したいときに役立ちます。

同時に `*.pkg` の `*` の部分が *name* 引数に指定された文字列に一致するファイルの検索もおこないます。この機能は `import` で始まる特別な行がないことを除き `*.pth` ファイルに似ています (*site* の項を参照)。`*.pkg` は重複のチェックを除き、信頼できるものとして扱われます。`*.pkg` ファイルの中に見つかったエントリはファイルシステム上に実在するか否かを問わず、そのまますべてパスに追加されます。(このような仕様です。)

入力パスがリストでない場合 (フリーズされたパッケージのとき) は何もせずにリターンします。入力パスが変更されていなければ、アイテムを末尾に追加しただけのコピーを返します。

sys.path はシーケンスであることが前提になっています。*sys.path* の要素の内、実在するディレクトリを指す文字列となっていないものは無視されます。ファイル名として使ったときにエラーが発生する *sys.path* の Unicode 要素がある場合、この関数 (*os.path.isdir()* を実行している行) で例外が発生する可能性があります。

class pkgutil.**ImpImporter**(*dirname=None*)
Python の「旧式の」インポートアルゴリズムをラップする、**PEP 302** に基づく Finder です。

dirname が文字列の場合、そのディレクトリを検索する **PEP 302** finder を作成します。*dirname* が `None` のとき、現在の *sys.path* とフリーズされた、あるいはビルトインの全てのモジュールを検索する **PEP 302** finder を作成します。

ImpImporter は現在のところ *sys.meta_path* に配置しての利用をサポートしていないことに注意してください。

バージョン 3.3 で非推奨: 標準インポートメカニズムが完全に **PEP 302** 準拠になり、これが *importlib* で利用可能であるため、このエミュレーションはもはや必要ありません。

class pkgutil.ImpLoader(*fullname*, *file*, *filename*, *etc*)
Python の ” クラシック” インポートアルゴリズムをラップする *loader*

バージョン 3.3 で非推奨: 標準インポートメカニズムが完全に **PEP 302** 準拠になり、これが *importlib* で利用可能であるため、このエミュレーションはもはや必要ありません。

pkgutil.find_loader(*fullname*)
fullname に対するモジュール *loader* オブジェクトを取得します。

これは後方互換性のために提供している *importlib.util.find_spec()* へのラッパーで、そこでのほどんどの失敗を *ImportError* に変換し、完全な ModuleSpec を返す代わりにローダのみを返しています。

バージョン 3.3 で変更: パッケージ内部の **PEP 302** エミュレーションに依存するのではなく直接的に *importlib* に基くように更新されました。

バージョン 3.4 で変更: **PEP 451** ベースに更新されました。

pkgutil.get_importer(*path_item*)
指定された *path_item* に対する *finder* を取得します。

path hook により新しい finder が作成された場合は、それは *sys.path_importer_cache* にキャッシュされます。

キャッシュ (やその一部) は、*sys.path_hooks* のリスキャンが必要になった場合は手動でクリアすることができます。

バージョン 3.3 で変更: パッケージ内部の **PEP 302** エミュレーションに依存するのではなく直接的に *importlib* に基くように更新されました。

pkgutil.get_loader(*module_or_name*)
module_or_name に対する *loader* オブジェクトを取得します。

module か package が通常の import 機構によってアクセスできる場合、その機構の該当部分に対するラッパーを返します。モジュールが見つからなかったり import できない場合は None を返します。その名前のモジュールがまだ import されていない場合、そのモジュールを含むパッケージが (あれば) そのパッケージの __path__ を確立するために import されます。

バージョン 3.3 で変更: パッケージ内部の **PEP 302** エミュレーションに依存するのではなく直接的に *importlib* に基くように更新されました。

バージョン 3.4 で変更: **PEP 451** ベースに更新されました。

pkgutil.iter_importers(*fullname=”*)
Yield *finder* objects for the given module name.

If fullname contains a ”.”, the finders will be for the package containing fullname, otherwise they will be all registered top level finders (i.e. those on both sys.meta_path and sys.path_hooks).

その名前のついたモジュールがパッケージ内に含まれている場合、この関数を実行した副作用としてそのパッケージが import されます。

モジュール名が指定されない場合は全てのトップレベルの finder が生成されます。

バージョン 3.3 で変更: パッケージ内部の **PEP 302** エミュレーションに依存するのではなく直接的に *importlib* に基くように更新されました。

pkgutil.**iter_modules**(*path=None*, *prefix=''*)

Yields *ModuleInfo* for all submodules on *path*, or, if *path* is `None`, all top-level modules on `sys.path`.

path は `None` か、モジュールを検索する path のリストのどちらかでなければなりません。

prefix は出力の全てのモジュール名の頭に出力する文字列です。

注釈: これは `iter_modules()` メソッドを定義している *finder* に対してのみ動作します。このインターフェイスは非標準なので、モジュールは *importlib.machinery.FileFinder* と *zipimport.zipimporter* の実装も提供します。

バージョン 3.3 で変更: パッケージ内部の **PEP 302** エミュレーションに依存するのではなく直接的に *importlib* に基くように更新されました。

pkgutil.**walk_packages**(*path=None*, *prefix=''*, *onerror=None*)

path を指定すれば再帰的にその中のモジュールすべてに対して、*path* が `None` ならばアクセスできるすべてのモジュールに対して、*ModuleInfo* を yield します。

path は `None` か、モジュールを検索する path のリストのどちらかでなければなりません。

prefix は出力の全てのモジュール名の頭に出力する文字列です。

この関数は与えられた *path* 上の全ての **パッケージ** (全てのモジュール **ではない**) を、サブモジュールを検索するのに必要な `__path__` 属性にアクセスするために import します。

onerror は、パッケージを import しようとしたときに何かの例外が発生した場合に、1 つの引数 (import しようとしていたパッケージの名前) で呼び出される関数です。*onerror* 関数が提供されない場合、*ImportError* は補足され無視されます。それ以外の全ての例外は伝播し、検索を停止させます。

例:

```
# list all modules python can access
walk_packages()

# list all submodules of ctypes
walk_packages(ctypes.__path__, ctypes.__name__ + '.')
```

注釈: これは `iter_modules()` メソッドを定義している *finder* に対してのみ動作します。このインターフェイスは非標準なので、モジュールは *importlib.machinery.FileFinder* と *zipimport.zipimporter* の実装も提供します。

バージョン 3.3 で変更: パッケージ内部の **PEP 302** エミュレーションに依存するのではなく直接的に *importlib* に基くように更新されました。

pkgutil.**get_data**(*package*, *resource*)

パッケージからリソースを取得します。

この関数は *loader* *get_data* API のラッパーです。*package* 引数は標準的なモジュール形式 (`foo.bar`) のパッケージ名でなければなりません。*resource* 引数は `/` をパス区切りに使った相対ファイル名の形式です。親ディレクトリを `..` としたり、ルートからの (`/` で始まる) 名前を使うことはできません。

この関数が返すのは指定されたリソースの内容であるバイナリ文字列です。

ファイルシステム中に位置するパッケージで既にインポートされているものに対しては、次と大体同じです:

```
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()
```

If the package cannot be located or loaded, or it uses a *loader* which does not support *get_data*, then `None` is returned. In particular, the *loader* for *namespace packages* does not support *get_data*.

31.3 modulefinder --- スクリプト中で使われているモジュールを検索する

ソースコード: Lib/modulefinder.py

このモジュールでは、スクリプト中で import されているモジュールセットを調べるために使える *ModuleFinder* クラスを提供しています。`modulefinder.py` はまた、Python スクリプトのファイル名を引数に指定してスクリプトとして実行し、import されているモジュールのレポートを出力させることもできます。

modulefinder.**AddPackagePath**(*pkg_name*, *path*)

pkg_name という名前のパッケージの在り処が *path* であることを記録します。

modulefinder.**ReplacePackage**(*oldname*, *newname*)

実際にはパッケージ内で *oldname* という名前になっているモジュールを *newname* という名前で指定できるようにします。

class modulefinder.**ModuleFinder**(*path=None*, *debug=0*, *excludes=[]*, *replace_paths=[]*)

このクラスでは *run_script()* および *report()* メソッドを提供しています。これらのメソッドは何らかのスクリプト中で import されているモジュールの集合を調べます。*path* はモジュールを検索する先のディレクトリ名からなるリストです。*path* を指定しない場合、`sys.path` を使います。*debug* にはデバッグレベルを設定します; 値を大きくすると、実行している内容を表すデバッグメッセージを出力します。*excludes* は検索から除外するモジュール名です。*replace_paths* には、モジュールパス内で置き換えられるパスをタプル (`oldpath, newpath`) からなるリストで指定します。

report()
スクリプトで import しているモジュールと、そのパスからなるリストを列挙したレポートを標準出力に出力します。モジュールを見つけられなかったり、モジュールがないように見える場合にも報告します。

run_script(*pathname*)
pathname に指定したファイルの内容を解析します。ファイルには Python コードが入っていなければなりません。

modules
モジュール名をモジュールに結びつける辞書。*ModuleFinder* の使用例 を参照して下さい。

31.3.1 ModuleFinder の使用例

解析対象のスクリプトはこれ (bacon.py) です:

```
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```

bacon.py のレポートを出力するスクリプトです:

```
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print('Loaded modules:')
for name, mod in finder.modules.items():
    print('%s: ' % name, end='')
    print(','.join(list(mod.globalnames.keys())[:3]))

print('-'*50)
print('Modules not imported:')
print('\n'.join(finder.badmodules.keys()))
```

出力例です (アーキテクチャに依って違ってくるかもしれません):

```
Loaded modules:
_types:
copyreg:  _inverted_registry,_slotnames,__all__
```

(次のページに続く)

(前のページからの続き)

```
sre_compile:  isstring,_sre,_optimize_unicode
_sre:
sre_constants:  REPEAT_ONE,makedict,AT_END_LINE
sys:
re:  __module__,finditer,_expand
itertools:
__main__:  re,itertools,baconhameggs
sre_parse:  _PATTERNENDERS,SRE_FLAG_UNICODE
array:
types:  __module__,IntType,TypeType
-----------------------------------------------------
Modules not imported:
guido.python.ham
baconhameggs
```

31.4 runpy --- Python モジュールの位置特定と実行

ソースコード: Lib/runpy.py

---

runpy モジュールは Python のモジュールをインポートせずにその位置を特定したり実行したりするのに使われます。その主な目的はファイルシステムではなく Python のモジュール名前空間を使って位置を特定したスクリプトの実行を可能にする `-m` コマンドラインスイッチを実装することです。

これはサンドボックスモジュール **ではない** ことに注意してください。すべてのコードは現在のプロセスで実行され、あらゆる副作用 (たとえば他のモジュールのキャッシュされたインポート等) は関数から戻った後にそのまま残ります。

さらに、*runpy* 関数から戻った後で、実行されたコードによって定義された任意の関数およびクラスが正常に動作することは保証されません。この制限が受け入れられないユースケースでは、*importlib* がこのモジュールより適切な選択となるでしょう。

runpy モジュールは 2 つの関数を提供しています:

runpy.**run_module**(*mod_name*, *init_globals=None*, *run_name=None*, *alter_sys=False*)

指定されたモジュールのコードを実行し、実行後のモジュールグローバル辞書を返します。モジュールのコードはまず標準インポート機構 (詳細は **PEP 302** を参照) を使ってモジュールの位置を特定され、まっさらなモジュール名前空間で実行されます。

mod_name 引数は絶対モジュール名でなければなりません。モジュール名が通常のモジュールではなくパッケージを参照していた場合、そのパッケージが import された後その中の `__main__` モジュールが実行され、実行後のモジュールグローバル辞書を返します。

オプションの辞書型引数 *init_globals* はコードを実行する前にモジュールグローバル辞書に前もって必要な設定しておくのに使われます。与えられた辞書は変更されません。その辞書の中に以下に挙げる特別なグローバル変数が定義されていたとしても、それらの定義は *run_module()* 関数によってオーバーライドされます。

特別なグローバル変数 __name__, __spec__, __file__, __cached__, __loader__, __package__ はモジュールコードが実行される前にグローバル辞書にセットされます。(この変数群は修正される最小セットです。これ以外の変数もインタプリタの実装の詳細として暗黙的に設定されるかもしれません)。

__name__ は、オプション引数 *run_name* が *None* でない場合、指定されたモジュールがパッケージであれば mod_name + '.__main__' に、そうでなければ *mod_name* 引数の値がセットされます。

__spec__ will be set appropriately for the *actually* imported module (that is, __spec__.name will always be *mod_name* or mod_name + '.__main__', never *run_name*).

__file__, __cached__, __loader__ and __package__ are set as normal based on the module spec.

引数 *alter_sys* が与えられて *True* に評価されるならば、sys.argv[0] は __file__ の値で更新され sys.modules[__name__] は実行されるモジュールの一時的モジュールオブジェクトで更新されます。sys.argv[0] と sys.modules[__name__] はどちらも関数が処理を戻す前にもとの値に復旧します。

この *sys* に対する操作はスレッドセーフではないということに注意してください。他のスレッドは部分的に初期化されたモジュールを見たり、入れ替えられた引数リストを見たりするかもしれません。この関数をスレッド化されたコードから起動するときは *sys* モジュールには手を触れないことが推奨されます。

参考:

コマンドラインから、-m オプションを与えることで同じ機能を実現出来ます。

バージョン 3.1 で変更: __main__ サブモジュールを検索することによってパッケージを実行する機能が追加されました。

バージョン 3.2 で変更: __cached__ グローバル変数が追加されました (**PEP 3147** を参照)。

バージョン 3.4 で変更: Updated to take advantage of the module spec feature added by **PEP 451**. This allows __cached__ to be set correctly for modules run this way, as well as ensuring the real module name is always accessible as __spec__.name.

runpy.**run_path**(*file_path*, *init_globals=None*, *run_name=None*)

指定されたファイルシステム上の場所にあるコードを実行して、結果としてそのモジュールグローバル辞書を返します。通常の CPython 実行時にコマンドラインで指定するスクリプト名と同じく、指定できるパスは Python ソースファイル、コンパイルされたバイトコードファイル、もしくは __main__ モジュールを含む有効な sys.path エントリ (例: トップレベルに __main__.py ファイルを持つ zip ファイル) です。

シンプルなスクリプトを指定した場合は、指定されたコードはシンプルに新しいモジュール名前空間で実行されます。有効な sys.path エントリ (一般的には zip ファイルかディレクトリ) を指定した場合は、最初にそのエントリが sys.path の先頭に追加されます。次に更新した sys.path を元に *__main__* モジュールを検索して実行します。指定された場所に *__main__* モジュールが無かった時、sys.path のどこか他のエントリに存在する別の *__main__* を実行してしまう可能性があることに注意してください。

オプションの辞書型引数 *init_globals* はコードを実行する前にモジュールグローバル辞書に前もって

必要な設定しておくのに使われます。与えられた辞書は変更されません。その辞書の中に以下に挙げる特別なグローバル変数が定義されていたとしても、それらの定義は *run_path()* 関数によってオーバーライドされます。

特別なグローバル変数 `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__`, `__package__` はモジュールコードが実行される前にグローバル辞書にセットされます。(この変数群は修正される最小セットです。これ以外の変数もインタプリタの実装の詳細として暗黙的に設定されるかもしれません)。

`__name__` は、オプション引数 *run_name* が *None* でない場合、*run_name* に設定され、それ以外の場合は `'<run_path>'` に設定されます。

If the supplied path directly references a script file (whether as source or as precompiled byte code), then `__file__` will be set to the supplied path, and `__spec__`, `__cached__`, `__loader__` and `__package__` will all be set to *None*.

If the supplied path is a reference to a valid sys.path entry, then `__spec__` will be set appropriately for the imported `__main__` module (that is, `__spec__.name` will always be `__main__`). `__file__`, `__cached__`, `__loader__` and `__package__` will be set as normal based on the module spec.

sys モジュールに対していくつかの変更操作が行われます。まず、`sys.path` が上記のように修正されます。`sys.argv[0]` は `file_path` に更新され、`sys.modules[__name__]` は実行されるモジュールのための一時モジュールオブジェクトに更新されます。*sys* の要素に対する全ての変更は、この関数から戻る前に元に戻されます。

run_module() と違い、*sys* に対する変更はオプションではありません。これらの変更は sys.path エントリの実行に必要不可欠だからです。スレッドセーフ性に関する制限はこの関数にも存在します。この関数をマルチスレッドプログラムから実行する場合は、import lock によりシリアライズして実行するか、別プロセスに委譲してください。

参考:

コマンドラインから using-on-interface-options で同じ機能を使えます (`python path/to/script`)。

バージョン 3.2 で追加.

バージョン 3.4 で変更: Updated to take advantage of the module spec feature added by **PEP 451**. This allows `__cached__` to be set correctly in the case where `__main__` is imported from a valid sys.path entry rather than being executed directly.

参考:

PEP 338 - **モジュールをスクリプトとして実行する**　Nick Coghlan によって書かれ実装された PEP。

PEP 366 - **main モジュールの明示的な相対インポート**　Nick Coghlan によって書かれ実装された PEP。

PEP 451 -- **インポートシステムのための ModuleSpec 型**　PEP written and implemented by Eric Snow

using-on-general - CPython コマンドライン詳細

importlib.import_module() 関数

31.5 importlib --- import の実装

バージョン 3.1 で追加.

ソースコード: Lib/importlib/__init__.py

31.5.1 はじめに

importlib パッケージの目的は 2 つあります。1 つ目は Python ソースコード中にある `import` 文の (そして、拡張として、*__import__()* 関数の) 実装を提供することです。このパッケージは `import` 文の、どの Python インタープリターでも動作する実装を提供します。また、Python 以外の言語で実装されたどの実装よりも把握しやすい実装を提供します。

2 つ目の目的は、このパッケージが公開している `import` を実装するための要素を利用して、(**インポーター**として知られる) インポートプロセスで動作するカスタムのオブジェクトを実装しやすくすることです。

参考:

import　`import` 文の言語リファレンス。

Packages specification　パッケージの元の仕様。幾つかの動作はこの仕様が書かれた頃から変更されています (例: *sys.modules* で `None` に基づくリダイレクト)。

__import__() **関数**　`import` 文はこの関数のシンタックスシュガーです。

PEP 235　大文字小文字を区別しないプラットフォームでのインポート

PEP 263　Python のソースコードのエンコーディング

PEP 302　新しいインポートフック

PEP 328　複数行のインポートと、絶対/相対インポート

PEP 366　main モジュールの明示的な相対インポート

PEP 420　暗黙的な名前空間パッケージ

PEP 451　インポートシステムのための ModuleSpec 型

PEP 488　PYO ファイルの撤廃

PEP 489　複数フェーズでの拡張モジュールの初期化

PEP 552　決定論的 pyc

PEP 3120　デフォルトのソースエンコーディングとして UTF-8 を使用

PEP 3147　PYC リポジトリディレクトリ

31.5.2 関数

importlib.__import__(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)
組み込みの *__import__()* 関数の実装です。

注釈: プログラムからモジュールをインポートする場合はこの関数の代わりに *import_module()* を使ってください。

---

importlib.import_module(*name*, *package=None*)
モジュールをインポートします。*name* 引数は、インポートするモジュールを絶対または相対表現 (例えば pkg.mod または ..mod) で指定します。name が相対表現で与えられたら、*package* 引数を、パッケージ名を解決するためのアンカーとなるパッケージの名前に設定する必要があります (例えば import_module('..mod', 'pkg.subpkg') は pkg.mod をインポートします)。

import_module() 関数は *importlib.__import__()* を単純化するラッパーとして働きます。つまり、この関数のすべての意味は *importlib.__import__()* から受け継いでいます。これらの 2 つの関数の最も重要な違いは、*import_module()* が指定されたパッケージやモジュール (例えば pkg.mod) を返すのに対し、*__import__()* はトップレベルのパッケージやモジュール (例えば pkg) を返すことです。

もしモジュールを動的にインポートしていて、インタープリタの実行開始後にモジュールが作成された (例えば、Python ソースファイルを作成した) 場合、インポートシステムが新しいモジュールを見つけられるように、*invalidate_caches()* を呼ぶ必要があるでしょう。

バージョン 3.3 で変更: 親パッケージは自動的にインポートされます。

importlib.find_loader(*name*, *path=None*)
モジュールのローダーを、オプションで指定された *path* 内から、検索します。モジュールが *sys.modules* にあれば、sys.modules[name].__loader__ が返されます (ただしローダーが None であるか設定されていなければ *ValueError* が送出されます)。なければ、*sys.meta_path* を使った検索がなされます。ローダーが見つからなければ None が返ります。

ドットのついた名前表記は、親モジュールのロードが必要なときに暗黙にインポートしないので、望ましくありません。サブモジュールを適切にインポートするには、そのサブモジュールの全ての親パッケージをインポートし、*path* に正しい引数を使ってください。

バージョン 3.3 で追加.

バージョン 3.4 で変更: __loader__ が set でない場合、None に設定されているときと同様に *ValueError* を送出します。

バージョン 3.4 で非推奨: 代わりに *importlib.util.find_spec()* を使用してください。

importlib.invalidate_caches()
sys.meta_path に保存されたファインダーの内部キャッシュを無効にします。ファインダーが invalidate_caches() を実装していれば、無効化を行うためにそれが呼び出されます。すべてのファ

インダーが新しいモジュールの存在に気づくことを保証しているプログラムの実行中に、モジュールが作成またはインストールされたなら、この関数が呼び出されるべきです。

バージョン 3.3 で追加.

importlib.**reload**(*module*)

以前にインポートされた *module* をリロードします。引数はモジュールオブジェクトでなければならず、したがってそれ以前に必ずインポートに成功していなければなりません。この関数は、モジュールのソースファイルを外部エディタで編集していて Python インタープリタから離れることなく新しいバージョンを試したい際に便利です。戻り値はモジュールオブジェクトです。(もし再インポートが異なるオブジェクトを *sys.modules* に配置したら、元の *module* とは異なるかもしれません。)

reload() が実行された場合:

- Python モジュールのコードは再コンパイルされ、モジュールレベルのコードが再度実行されます。モジュールの辞書中にある何らかの名前に結び付けられたオブジェクトは、そのモジュールを最初にロードしたときの **ローダー** を再利用して新たに定義されます。拡張モジュールの `init` 関数が二度呼び出されることはありません。
- Python における他のオブジェクトと同様、以前のオブジェクトのメモリ領域は、参照カウントがゼロにならないかぎり再利用されません。
- モジュール名前空間内の名前は新しいオブジェクト (または更新されたオブジェクト) を指すよう更新されます。
- 以前のオブジェクトが (外部の他のモジュールなどからの) 参照を受けている場合、それらを新たなオブジェクトに再束縛し直すことはないので、必要なら自分で名前空間を更新しなければなりません。

いくつか補足説明があります:

モジュールが再ロードされた際、その辞書 (モジュールのグローバル変数を含みます) はそのまま残ります。名前の再定義を行うと、以前の定義を上書きするので、一般的には問題はありません。新たなバージョンのモジュールが古いバージョンで定義された名前を定義していない場合、古い定義がそのまま残ります。辞書がグローバルテーブルやオブジェクトのキャッシュを維持していれば、この機能をモジュールを有効性を引き出すために使うことができます --- つまり、`try` 文を使えば、必要に応じてテーブルがあるかどうかをテストし、その初期化を飛ばすことができます:

```
try:
    cache
except NameError:
    cache = {}
```

組み込みモジュールや動的にロードされるモジュールを再ロードすることは、一般的にそれほど便利ではありません。*sys*, *__main__*, *builtins* やその他重要なモジュールの再ロードはお勧め出来ません。多くの場合、拡張モジュールは 1 度以上初期化されるようには設計されておらず、再ロードされた場合には何らかの理由で失敗するかもしれません。

一方のモジュールが `from ... import ...` を使って、オブジェクトを他方のモジュールからインポートしているなら、他方のモジュールを *reload()* で呼び出しても、そのモジュールからインポートされた

オブジェクトを再定義することはできません --- この問題を回避する一つの方法は、from 文を再度実行することで、もう一つの方法は from 文の代わりに import と限定的な名前 (*module.name*) を使うことです。

あるモジュールがクラスのインスタンスを生成している場合、そのクラスを定義しているモジュールの再ロードはそれらインスタンスのメソッド定義に影響しません --- それらは古いクラス定義を使い続けます。これは派生クラスの場合でも同じです。

バージョン 3.4 で追加.

バージョン 3.7 で変更: リロードされたモジュールの ModuleSpec が欠けていたときは *ModuleNotFoundError* が送出されます。

31.5.3 importlib.abc -- インポートに関連する抽象基底クラス

ソースコード: Lib/importlib/abc.py

---

importlib.abc モジュールは、import に使われるすべてのコア抽象基底クラス含みます。コア抽象基底クラスの実装を助けるために、コア抽象基底クラスのサブクラスもいくつか提供されています。

抽象基底クラス階層:

```
object
 +-- Finder (deprecated)
 |    +-- MetaPathFinder
 |    +-- PathEntryFinder
 +-- Loader
      +-- ResourceLoader --------+
      +-- InspectLoader          |
           +-- ExecutionLoader --+
                                 +-- FileLoader
                                 +-- SourceLoader
```

class importlib.abc.**Finder**

finder を表す抽象基底クラスです。

バージョン 3.3 で非推奨: 代わりに *MetaPathFinder* または *PathEntryFinder* を使ってください。

abstractmethod **find_module**(*fullname*, *path=None*)

指定されたモジュールの **ローダー** を検索するための抽象メソッドです。もとは **PEP 302** で仕様が定められ、このメソッドは *sys.meta_path* の中およびパスに基づくインポートサブシステムの中で使用することを意図されています。

バージョン 3.4 で変更: 呼び出されたときに *NotImplementedError* を送出する代わりに None を返します。

class importlib.abc.**MetaPathFinder**

meta path finder を表す抽象基底クラスです。互換性のため、これは *Finder* のサブクラスです。

バージョン 3.3 で追加.

find_spec(*fullname*, *path*, *target=None*)

指定されたモジュールに対応する **スペック** を検索する抽象メソッド。もしこれがトップレベルのインポートなら、*path* は None です。そうでなければ、これはサブパッケージまたはモジュールのための検索で、*path* は親パッケージの __path__ の値です。スペックが見つからなければ None が返されます。target は、渡されてきたならモジュールオブジェクトです。これはファインダーがどのようなスペックを返せばよいか推測するために使用します。具体的な MetaPathFinders を実装するためには *importlib.util.spec_from_loader()* が便利かもしれません。

バージョン 3.4 で追加.

find_module(*fullname*, *path*)

指定されたモジュールの **ローダー** を検索するためのレガシーなメソッドです。これがトップレベルのインポートなら、*path* は None になります。そうでなければ、これはサブパッケージまたはモジュールの検索で、*path* は親パッケージの __path__ の値になります。ローダーが見つからなければ、None が返されます。

find_spec() が定義された場合、後方互換な機能が提供されます。

バージョン 3.4 で変更: このメソッドが呼ばれた場合 *NotImplementedError* を投げる代わりに None を返します。この機能を提供するのに *find_spec()* を使用できます。

バージョン 3.4 で非推奨: 代わりに *find_spec()* を使用してください。

invalidate_caches()

このファインダーで使われている内部キャッシュがあれば無効にするオプションのメソッドです。*sys.meta_path* 上のすべてのファインダーのキャッシュを無効化する際、*importlib.invalidate_caches()* によって使われます。

バージョン 3.4 で変更: 呼び出されたときに NotImplemented を送出する代わりに None を返します。

class importlib.abc.**PathEntryFinder**

パスエントリ・ファインダー を表す抽象基底クラスです。*MetaPathFinder* と似ているところがありますが、PathEntryFinder は PathFinder の与えるパスに基づくインポートサブシステムの中でのみ使うことが意図されています。この抽象基底クラスは互換性の理由だけのために、*Finder* のサブクラスにしてあります。

バージョン 3.3 で追加.

find_spec(*fullname*, *target=None*)

指定されたモジュールに対応する **スペック** を検索する抽象メソッド。ファインダーは、割り当てられている **パス・エントリー** 内のモジュールだけを検索します。スペックが見つからなければ None が返されます。target は、渡されてきたならモジュールオブジェクトです。これはファインダーがどのようなスペックを返せばよいか推測するために使用します。具体的な PathEntryFinders を実装するためには *importlib.util.spec_from_loader()* が便利かもしれません。

バージョン 3.4 で追加.

find_loader(*fullname*)

指定されたモジュールの **ローダー** を検索する抽象メソッドです。(loader, portion) の 2-タプルを返します。ただし portion は名前空間パッケージの部分に寄与するファイルシステム上の場所のシーケンスです。loader は名前空間パッケージへのファイルシステム上の場所の寄与を表す portion を明記するとき None にできます。loader が名前空間パッケージの一部ではないことを明記するとき portion に空のリストが使えます。loader が None で portion が空のリストなら、名前空間パッケージのローダーや場所が見つかりませんでした (すなわち、モジュールの何も見つかりませんでした)。

find_spec() が定義された場合、後方互換な機能が提供されます。

バージョン 3.4 で変更: *NotImplementedError* を送出する代わりに (None, []) を返します。機能を提供できる場合 *find_spec()* を使用します。

バージョン 3.4 で非推奨: 代わりに *find_spec()* を使用してください。

find_module(*fullname*)

Finder.find_module() の具象実装で、self.find_loader(fullname)[0] と等価です。

バージョン 3.4 で非推奨: 代わりに *find_spec()* を使用してください。

invalidate_caches()

このファインダーで使われている内部キャッシュがあれば無効にするオプションのメソッドです。キャッシュされたすべてのファインダーの無効化する際、PathFinder.invalidate_caches() によって使われます。

class importlib.abc.Loader

loader の抽象基底クラスです。ローダーの厳密な定義は **PEP 302** を参照してください。

リソースの読み出しをサポートさせたいローダーには、*importlib.abc.ResourceReader* で指定されている get_resource_reader(fullname) を実装してください。

バージョン 3.7 で変更: オプションの get_resource_reader() メソッドが導入されました。

create_module(*spec*)

モジュールをインポートする際に使用されるモジュールオブジェクトを返すメソッド。このメソッドは None を戻すことができ、その場合はデフォルトのモジュール作成のセマンティクスが適用されることを示します。

バージョン 3.4 で追加.

バージョン 3.5 で変更: Python 3.6 からは、*exec_module()* が定義されている場合は、このメソッドはオプションではなくなります。

exec_module(*module*)

モジュールがインポートまたはリロードされる際に、そのモジュールをモジュール自身の名前空間の中で実行する抽象的なメソッド。exec_module() が呼ばれる時点で、モジュールはすでに初期設定されている必要があります。このメソッドが存在するときは、*create_module()* の定義が必須です。

バージョン 3.4 で追加.

バージョン 3.6 で変更: *create_module()* の定義が必須となりました。

load_module(*fullname*)

モジュールをロードするためのレガシーなメソッドです。モジュールがロードできなければ *ImportError* を送出し、ロードできればロードされたモジュールを返します。

要求されたモジュールが既に *sys.modules* に存在したなら、そのモジュールが使われリロードされる必要があります。存在しなければ、インポートからの再帰を防ぐため、ローダーはロードが始まる前に新しいモジュールを作成して *sys.modules* に挿入する必要があります。ローダーがモジュールを挿入した後にロードが失敗したなら、ローダーはそのモジュールを *sys.modules* から削除する必要があります。ローダーが実行を始める前に既に *sys.modules* にあったモジュールは、そのままにします (*importlib.util.module_for_loader()* を参照してください)。

ローダーはモジュールにいくつかの属性を設定する必要があります。(なお、これらの属性には、モジュールがリロードされた際に変化するものがあります):

- **__name__** モジュールの名前です。
- **__file__** モジュールのデータが保存されている場所へのパスです (組み込みモジュールには設定されません)。
- **__cached__** モジュールのコンパイルされた版が保存されている (べき) 場所へのパスです (この属性が適切でないときには設定されません)。
- **__path__** パッケージ内の検索パスを指定する文字列のリストです。この属性はモジュールには設定されません。
- **__package__** The fully-qualified name of the package under which the module was loaded as a submodule (or the empty string for top-level modules). For packages, it is the same as **__name__**. The *importlib.util.module_for_loader()* decorator can handle the details for **__package__**.
- **__loader__** モジュールをロードするのに使われたローダーです。*importlib.util.module_for_loader()* デコレータで、**__package__** の詳細を扱えます。

exec_module() が利用可能な場合、後方互換な機能が提供されます。

バージョン 3.4 で変更: このメソッドが呼ばれた時に、*NotImplementedError* の代わりに *ImportError* を送出します。*exec_module()* が利用可能な時は、この機能は提供されます。

バージョン 3.4 で非推奨: モジュールをロードするための推奨される API は、*exec_module()* (および *create_module()*) です。ローダーは load_module() の代わりにそれを実装するべきです。exec_module() が実装されている場合、インポート機構は load_module() の他のすべての責任を肩代わりします。

module_repr(*module*)

実装されていた場合、与えられたモジュールの repr を計算して文字列として返すためのレガシー

なメソッドです。モジュール型のデフォルトの repr() は、必要に応じてこのメソッドの結果を使います。

バージョン 3.3 で追加.

バージョン 3.4 で変更: 抽象メソッドではなくオプショナルになりました。

バージョン 3.4 で非推奨: インポート機構はこれを自動的に考慮するようになりました。

class importlib.abc.**ResourceReader**

resources の読み出し機能を提供する **抽象基底クラス** *(abstract base class, ABC)* です。

From the perspective of this ABC, a *resource* is a binary artifact that is shipped within a package. Typically this is something like a data file that lives next to the `__init__.py` file of the package. The purpose of this class is to help abstract out the accessing of such data files so that it does not matter if the package and its data file(s) are stored in a e.g. zip file versus on the file system.

For any of methods of this class, a *resource* argument is expected to be a *path-like object* which represents conceptually just a file name. This means that no subdirectory paths should be included in the *resource* argument. This is because the location of the package the reader is for, acts as the "directory". Hence the metaphor for directories and file names is packages and resources, respectively. This is also why instances of this class are expected to directly correlate to a specific package (instead of potentially representing multiple packages or a module).

Loaders that wish to support resource reading are expected to provide a method called `get_resource_reader(fullname)` which returns an object implementing this ABC's interface. If the module specified by fullname is not a package, this method should return *None*. An object compatible with this ABC should only be returned when the specified module is a package.

バージョン 3.7 で追加.

abstractmethod **open_resource**(*resource*)

Returns an opened, *file-like object* for binary reading of the *resource*.

リソースが見付からない場合は、*FileNotFoundError* が送出されます。

abstractmethod **resource_path**(*resource*)

resource へのファイルシステムパスを返します。

リソースの実体がファイルシステムに存在しない場合、*FileNotFoundError* が送出されます。

abstractmethod **is_resource**(*name*)

name という名前がリソースだと見なせるなら `True` を返します。*name* が存在しない場合は *FileNotFoundError* が送出されます。

abstractmethod **contents**()

Returns an *iterable* of strings over the contents of the package. Do note that it is not required that all names returned by the iterator be actual resources, e.g. it is acceptable to return names for which *is_resource()* would be false.

Allowing non-resource names to be returned is to allow for situations where how a package and its resources are stored are known a priori and the non-resource names would be useful. For instance, returning subdirectory names is allowed so that when it is known that the package and resources are stored on the file system then those subdirectory names can be used directly.

The abstract method returns an iterable of no items.

class importlib.abc.ResourceLoader

loader の抽象基底クラスで、ストレージバックエンドから任意のリソースをロードするオプションの **PEP 302** プロトコルを実装します。

バージョン 3.7 で非推奨: This ABC is deprecated in favour of supporting resource loading through *importlib.abc.ResourceReader*.

abstractmethod get_data(*path*)

path に割り当てられたデータのバイト列を返す抽象メソッドです。任意のデータを保管できるファイル的なストレージバックエンドをもつローダーは、この抽象メソッドを実装して、保管されたデータに直接アクセスさせるようにできます。*path* が見つからなければ *OSError* を送出する必要があります。*path* は、モジュールの `__file__` 属性を使って、またはパッケージの `__path__` の要素を使って、構成されることが期待されます。

バージョン 3.4 で変更: *NotImplementedError* の代わりに *OSError* を送出します。

class importlib.abc.InspectLoader

loader の抽象基底クラスで、ローダーがモジュールを検査するためのオプションの **PEP 302** プロトコルを実装します。

get_code(*fullname*)

モジュールの *code* オブジェクトを返すか、(例えば組み込みモジュールの場合に) モジュールがコードオブジェクトを持たなければ `None` を返します。要求されたモジュールをローダーが見つけられなかった場合は *ImportError* を送出します。

注釈: このメソッドにはデフォルト実装がありますが、とはいえパフォーマンスのために、可能ならばオーバライドしたほうが良いです。

バージョン 3.4 で変更: このメソッドはもはや抽象メソッドではなく、具象実装が提供されます。

abstractmethod get_source(*fullname*)

モジュールのソースを返す抽象メソッドです。これは認識されたすべての行セパレータを `'\n'` 文字に変換し、*universal newlines* を使ったテキスト文字列として返されます。利用できるソースがなければ (例えば組み込みモジュール)、`None` を返します。指定されたモジュールが見つからなければ、*ImportError* を送出します。

バージョン 3.4 で変更: *NotImplementedError* の代わりに *ImportError* を送出します。

is_package(*fullname*)

モジュールがパッケージであれば True を返し、そうでなければ False を返す抽象メソッドです。

ローダー がモジュールを見つけられなかったなら *ImportError* が送出されます。

バージョン 3.4 で変更: *NotImplementedError* の代わりに *ImportError* を送出します。

static source_to_code(*data*, *path='<string>'*)

Python のソースからコードオブジェクトを作ります。

data 引数は *compile()* 関数がサポートするもの (すなわち文字列かバイト) なら何でも構いません。*path* 引数はソースコードの元々の場所への " パス" でなければなりませんが、抽象概念 (例えば zip ファイル内の場所) でも構いません。

結果のコードオブジェクトを使って、`exec(code, module.__dict__)` を呼ぶことでモジュール内でコードを実行できます。

バージョン 3.4 で追加.

バージョン 3.5 で変更: スタティックメソッドになりました。

exec_module(*module*)

Loader.exec_module() の実装です。

バージョン 3.4 で追加.

load_module(*fullname*)

Loader.load_module() の実装です。

バージョン 3.4 で非推奨: 代わりに *exec_module()* を使用してください。

class importlib.abc.**ExecutionLoader**

InspectLoader から継承された抽象基底クラスで、実装されていれば、モジュールをスクリプトとして実行する助けになります。この抽象基底クラスはオプションの **PEP 302** プロトコルを表します。

abstractmethod get_filename(*fullname*)

指定されたモジュールの `__file__` の値を返す抽象メソッドです。利用できるパスがなければ、*ImportError* が送出されます。

ソースコードが利用できるなら、そのモジュールのロードにバイトコードが使われたかにかかわらず、このメソッドはそのソースファイルへのパスを返す必要があります。

バージョン 3.4 で変更: *NotImplementedError* の代わりに *ImportError* を送出します。

class importlib.abc.**FileLoader**(*fullname*, *path*)

ResourceLoader と *ExecutionLoader* から継承された抽象基底クラスで、*ResourceLoader.get_data()* および *ExecutionLoader.get_filename()* の具象実装を提供します。

fullname 引数は、ローダーが解決しようとするモジュールの、完全に解決された名前です。*path* 引数は、モジュールのファイルへのパスです。

バージョン 3.3 で追加.

name

ローダーが扱えるモジュールの名前です。

path
モジュールのファイルへのパスです。

load_module(*fullname*)
親クラスの load_module() を呼び出します。

バージョン 3.4 で非推奨: 代わりに *Loader.exec_module()* を使用してください。

abstractmethod get_filename(*fullname*)
path を返します。

abstractmethod get_data(*path*)
path をバイナリファイルとして読み込み、そのバイト列を返します。

class importlib.abc.SourceLoader
ソース (オプションでバイトコード) ファイルのロードを実装する抽象基底クラスです。このクラスは、*ResourceLoader* と *ExecutionLoader* の両方を継承し、以下の実装が必要です:

- *ResourceLoader.get_data()*
- *ExecutionLoader.get_filename()* ソースファイルへのパスのみを返す必要があります。ソースなしのロードはサポートされていません。

このクラスでこれらの抽象メソッドを定義することで、バイトコードファイルを追加でサポートします。これらのメソッドを定義しなければ (またはそのモジュールが *NotImplementedError* を送出すれば)、このローダーはソースコードに対してのみ働きます。これらのメソッドを実装することで、ローダーはソースとバイトコードファイル **の組み合わせ** に対して働きます。バイトコードのみを与えた **ソースのない** ロードは認められません。バイトコードファイルは、Python コンパイラによる解析の工程をなくして速度を上げる最適化です。ですから、バイトコード特有の API は公開されていません。

path_stats(*path*)
指定されたパスについてのメタデータを含む *dict* を返す、オプションの抽象メソッドです。サポートされる辞書のキーは:

- 'mtime' (必須): ソースコードの更新時刻を表す整数または浮動小数点数です。
- 'size' (任意): バイト数で表したソースコードのサイズです。

未来の拡張のため、辞書内の他のキーは無視されます。パスが扱えなければ、*OSError* が送出されます。

バージョン 3.3 で追加.

バージョン 3.4 で変更: *NotImplementedError* の代わりに *OSError* を送出します。

path_mtime(*path*)
指定されたパスの更新時刻を返す、オプションの抽象メソッドです。

バージョン 3.3 で非推奨: このメソッドは廃止され、*path_stats()* が推奨されます。このモジュールを実装する必要はありませんが、互換性のため現在も利用できます。パスが扱えなければ、*OSError* が送出されます。

バージョン 3.4 で変更: *NotImplementedError* の代わりに *OSError* を送出します。

set_data(*path*, *data*)
ファイルパスに指定されたバイト列を書き込むオプションの抽象メソッドです。存在しない中間ディレクトリがあれば、自動で作成されます。

パスへの書き込みが読み出し専用のために失敗したとき (*errno.EACCES*/*PermissionError*) 、その例外を伝播させません。

バージョン 3.4 で変更: 呼ばれたときに *NotImplementedError* を送出することは最早ありません。

get_code(*fullname*)
InspectLoader.get_code() の具象実装です。

exec_module(*module*)
Loader.exec_module() の具象実装です。

バージョン 3.4 で追加.

load_module(*fullname*)
Loader.load_module() の具象実装です。

バージョン 3.4 で非推奨: 代わりに *exec_module()* を使用してください。

get_source(*fullname*)
InspectLoader.get_source() の具象実装です。

is_package(*fullname*)
InspectLoader.is_package() の具象実装です。モジュールは、次の **両方** を満たすならパッケージであると決定されます。モジュールの (*ExecutionLoader.get_filename()* で与えられる) ファイルパスが、ファイル拡張子を除くと `__init__` という名のファイルであること。モジュール名自体が `__init__` で終わらないこと。

31.5.4 importlib.resources -- リソース

ソースコード: Lib/importlib/resources.py

バージョン 3.7 で追加.

このモジュールは、Python のインポートシステムを利用して、**パッケージ** 内の **リソース** へのアクセスを提供します。パッケージをインポートすることができれば、そのパッケージ内のリソースにアクセスすることができます。リソースは、バイナリモードでもテキストモードでも、開いたり読んだりすることができます。

Resources are roughly akin to files inside directories, though it's important to keep in mind that this is just a metaphor. Resources and packages **do not** have to exist as physical files and directories on the file system.

注釈: This module provides functionality similar to pkg_resources Basic Resource Access without the performance overhead of that package. This makes reading resources included in packages easier, with more stable and consistent semantics.

The standalone backport of this module provides more information on using importlib.resources and migrating from pkg_resources to importlib.resources.

リソースの読み出しをサポートさせたいローダーには、*importlib.abc.ResourceReader* で指定されている `get_resource_reader(fullname)` を実装してください。

次の型が定義されています。

`importlib.resources.Package`
The `Package` type is defined as `Union[str, ModuleType]`. This means that where the function describes accepting a `Package`, you can pass in either a string or a module. Module objects must have a resolvable `__spec__.submodule_search_locations` that is not `None`.

`importlib.resources.Resource`
This type describes the resource names passed into the various functions in this package. This is defined as `Union[str, os.PathLike]`.

次の関数が利用可能です。

`importlib.resources.open_binary`(*package*, *resource*)
パッケージ 内の **リソース** をバイナリ読み取り用に開きます。

package は `Package` の要件に従った名前またはモジュールオブジェクトです。*resource* は *package* 内で開くリソースの名前です。パス区切り文字を含むことはできず、サブリソースを持つことはできません（つまり、ディレクトリにはなれません）。この関数は、バイナリ I/O ストリームを読み込むために開いている `typing.BinaryIO` のインスタンスを返します。

`importlib.resources.open_text`(*package*, *resource*, *encoding='utf-8'*, *errors='strict'*)
package 内の *resource* をテキスト読み取り用に開きます。デフォルトでは、リソースは UTF-8 として読み取り用に開かれます。

package は `Package` の要件に従った名前またはモジュールオブジェクトです。*resource* は *package* 内で開くリソースの名前です。パス区切り文字を含むことはできず、サブリソースを持つことはできません（つまり、ディレクトリにはなれません）。*encoding* と *errors* は、組み込みの *open()* と同じ意味を持ちます。

この関数は、テキスト I/O ストリームを読み込むために開いている `typing.TextIO` のインスタンスを返します。

`importlib.resources.read_binary`(*package*, *resource*)
package 内の *resource* の内容を読み取り、`bytes` として返します。

package は `Package` の要件に従った名前またはモジュールオブジェクトです。*resource* は *package* 内で開くリソースの名前です。パス区切り文字を含むことはできず、サブリソースを持つことはできませ

ん（つまり、ディレクトリにはなれません）。この関数は、リソースの内容を *bytes* として返します。

importlib.resources.**read_text**(*package*, *resource*, *encoding='utf-8'*, *errors='strict'*)
package 内の *resource* の内容を読み込んで `str` として返します。デフォルトでは、内容は厳密な UTF-8 として読み込まれます。

package は `Package` の要件に従った名前またはモジュールオブジェクトです。*resource* は *package* 内で開くリソースの名前です。パス区切り文字を含むことはできず、サブリソースを持つことはできません（つまり、ディレクトリにはなれません）。*encoding* と *errors* は、組み込みの *open()* と同じ意味を持ちます。この関数は、リソースの内容を *str* として返します。

importlib.resources.**path**(*package*, *resource*)
resource へのパスを実際のファイルシステムのパスとして返します。この関数は、`with` 文で使用するためのコンテキストマネージャを返します。コンテキストマネージャは *pathlib.Path* オブジェクトを提供します。

コンテキストマネージャーを終了すると、例えば zip ファイルからリソースを抽出する必要がある場合に作成される一時ファイルを削除します。

package は `Package` の要件に従った名前またはモジュールオブジェクトです。*resource* は *package* 内で開くリソースの名前です。パス区切り文字を含むことはできず、サブリソースを持つことはできません（つまり、ディレクトリにはなれません）。

importlib.resources.**is_resource**(*package*, *name*)
Return `True` if there is a resource named *name* in the package, otherwise `False`. Remember that directories are *not* resources! *package* is either a name or a module object which conforms to the `Package` requirements.

importlib.resources.**contents**(*package*)
パッケージ内の名前付きアイテムに対するイテラブルを返します。イテラブルは *str* リソース (ファイルなど) と非リソース (ディレクトリなど) を返します。イテラブルは、サブディレクトリへの再帰は行いません。

package は、名前または `Package` の要件に適合するモジュールオブジェクトのいずれかです。

31.5.5 importlib.machinery -- インポータおよびパスフック

ソースコード: Lib/importlib/machinery.py

このモジュールには、`import` がモジュールを検索してロードするのに役立つ様々なオブジェクトがあります。

importlib.machinery.**SOURCE_SUFFIXES**
認識されているソースモジュールのファイル接尾辞を表す文字列のリストです。

バージョン 3.3 で追加.

importlib.machinery.**DEBUG_BYTECODE_SUFFIXES**
最適化されていないバイトコードモジュールのファイル接尾辞を表す文字列のリストです。

バージョン 3.3 で追加.

バージョン 3.5 で非推奨: 代わりに *BYTECODE_SUFFIXES* を使ってください。

importlib.machinery.**OPTIMIZED_BYTECODE_SUFFIXES**
最適化されたバイトコードモジュールのファイル接尾辞を表す文字列のリストです。

バージョン 3.3 で追加.

バージョン 3.5 で非推奨: 代わりに *BYTECODE_SUFFIXES* を使ってください。

importlib.machinery.**BYTECODE_SUFFIXES**
認識されているバイトコードモジュールのファイル接尾辞を表す文字列のリストです (先頭のドットを含みます)。

バージョン 3.3 で追加.

バージョン 3.5 で変更: この値は `__debug__` に依存しなくなりました。

importlib.machinery.**EXTENSION_SUFFIXES**
認識されている最適化された拡張モジュールのファイル接尾辞を表す文字列のリストです。

バージョン 3.3 で追加.

importlib.machinery.**all_suffixes**()
標準のインポート機構によって認識されているすべてのファイル接尾辞を表す文字列の組み合わせられたリストを返します。これが役立つのは、あるファイルシステムパスがモジュールを参照する可能性があるかだけを知りたくて、そのモジュールの種類を詳しく知る必要はないコード (例えば *inspect.getmodulename()*) です。

バージョン 3.3 で追加.

class importlib.machinery.**BuiltinImporter**
組み込みモジュールの *importer* です。すべての既知のモジュールは *sys.builtin_module_names* に列挙されています。このクラスは *importlib.abc.MetaPathFinder* および *importlib.abc.InspectLoader* 抽象基底クラスを実装します。

インスタンス化の必要性を軽減するため、このクラスにはクラスメソッドだけが定義されています。

バージョン 3.5 で変更: **PEP 489** の一環として、ビルトインインポーターは `Loader.create_module()` と `Loader.exec_module()` を実装しています。

class importlib.machinery.**FrozenImporter**
フリーズされたモジュールの **インポーター** です。このクラスは *importlib.abc.MetaPathFinder* および *importlib.abc.InspectLoader* 抽象基底クラスを実装します。

インスタンス化の必要性を軽減するため、このクラスにはクラスメソッドだけが定義されています。

バージョン 3.4 で変更: Gained `create_module()` and `exec_module()` methods.

class importlib.machinery.**WindowsRegistryFinder**

Windows レジストリで宣言されたモジュールの *finder* です。このクラスは *importlib.abc.MetaPathFinder* 抽象基底クラスを実装します。

インスタンス化の必要性を軽減するため、このクラスにはクラスメソッドだけが定義されています。

バージョン 3.3 で追加.

バージョン 3.6 で非推奨: 代わりに *site* の設定を使ってください。Python の将来のバージョンでは、デフォルトでこのファインダーが使えなくなるかもしれません。

class importlib.machinery.**PathFinder**

sys.path およびパッケージの `__path__` 属性の *Finder* です。このクラスは *importlib.abc.MetaPathFinder* 抽象基底クラスを実装します。

インスタンス化の必要性を軽減するため、このクラスにはクラスメソッドだけが定義されています。

classmethod find_spec(*fullname*, *path=None*, *target=None*)

sys.path または定義されていれば *path* から、*fullname* で指定されたモジュールの **スペック** の検索を試みるクラスメソッドです。検索されるそれぞれのパスエントリに対して *sys.path_importer_cache* が検査されます。偽でないオブジェクトが見つかれば、それが目的のモジュールを検索するための **パスエントリ・ファインダー** として使われます。*sys.path_importer_cache* に目的のエントリが見つからなければ、パスエントリに対するファインダーが *sys.path_hooks* から検索され、見つかれば、それが *sys.path_importer_cache* に保管されるとともに、モジュールについて問い合わせられます。それでもファインダーが見つからなければ `None` が保管され、また返されます。

バージョン 3.4 で追加.

バージョン 3.5 で変更: もしカレントワーキングディレクトリ -- 空の文字列によって表されている -- がすでに有効でなければ、`None` が返されますが値は *sys.path_importer_cache* にキャッシュされません。

classmethod find_module(*fullname*, *path=None*)

find_spec() まわりのレガシーなラッパです。

バージョン 3.4 で非推奨: 代わりに *find_spec()* を使用してください。

classmethod invalidate_caches()

Calls *importlib.abc.PathEntryFinder.invalidate_caches()* on all finders stored in *sys.path_importer_cache* that define the method. Otherwise entries in *sys.path_importer_cache* set to `None` are deleted.

バージョン 3.7 で変更: Entries of `None` in *sys.path_importer_cache* are deleted.

バージョン 3.4 で変更: `''` (すなわち空の文字列) に対してはカレントワーキングディレクトリとともに *sys.path_hooks* のオブジェクトを呼び出します。

class importlib.machinery.**FileFinder**(*path*, **loader_details*)

ファイルシステムからの結果をキャッシュする *importlib.abc.PathEntryFinder* の具象実装です。

path 引数は検索を担当するファインダーのディレクトリです。

loader_details 引数は、可変個の 2 要素タプルで、それぞれがローダーとローダーが認識するファイル接尾辞のシーケンスとを含みます。ローダーは、呼び出し可能でモジュール名と見つかったファイルのパスとの 2 引数を受け付けることを期待されます。

ファインダーはモジュール検索のたびに stat を呼び出し、必要に応じてディレクトリの内容をキャッシュすることで、コードキャッシュが古くなっていないことを確かめます。キャッシュの古さはオペレーティングシステムのファイルシステムのステート情報の粒度に依存しますから、モジュールを検索し、新しいファイルを作成し、その後に新しいファイルが表すモジュールを検索する、という競合状態の可能性があります。この操作が stat の呼び出しの粒度に収まるほど速く起こると、モジュールの検索が失敗します。これを防ぐためには、モジュールを動的に作成する際に、必ず *importlib.invalidate_caches()* を呼び出してください。

バージョン 3.3 で追加.

path
ファインダーが検索されるパスです。

find_spec(*fullname*, *target=None*)
path 内で *fullname* を扱うスペックの探索を試みます。

バージョン 3.4 で追加.

find_loader(*fullname*)
path 内で *fullname* を扱うローダーの検索を試みます。

invalidate_caches()
内部キャッシュを完全に消去します。

classmethod path_hook(**loader_details*)
sys.path_hooks で使用するクロージャを返すクラスメソッドです。クロージャに直接渡された path 引数を直接的に、*loader_details* を間接的に使って、*FileFinder* のインスタンスが返されます。

クロージャへの引数が存在するディレクトリでなければ、*ImportError* が送出されます。

class importlib.machinery.**SourceFileLoader**(*fullname*, *path*)
importlib.abc.FileLoader を継承し、その他いくつかのメソッドの具象実装を提供する、*importlib.abc.SourceLoader* の具象実装です。

バージョン 3.3 で追加.

name
このローダーが扱うモジュールの名前です。

path
ソースファイルへのパスです。

is_package(*fullname*)
path がパッケージを表すとき True を返します。

path_stats(*path*)

importlib.abc.SourceLoader.path_stats() の具象実装です。

set_data(*path*, *data*)

importlib.abc.SourceLoader.set_data() の具象実装です。

load_module(*name=None*)

ロードするモジュールの名前指定がオプションの、*importlib.abc.Loader.load_module()* の具象実装です。

バージョン 3.6 で非推奨: 代わりに *importlib.abc.Loader.exec_module()* を使用してください。

class importlib.machinery.**SourcelessFileLoader**(*fullname*, *path*)

バイトコードファイル (すなわちソースコードファイルが存在しない) をインポートできる *importlib.abc.FileLoader* の具象実装です。

注意として、バイトコードを直接使う (つまりソースコードファイルがない) と、そのモジュールはすべての Python 実装では使用できないし、新しいバージョンの Python ではバイトコードフォーマットが変更されていたら使用できません。

バージョン 3.3 で追加.

name

ローダーが扱うモジュールの名前です。

path

バイトコードファイルへのパスです。

is_package(*fullname*)

そのモジュールがパッケージであるかを *path* に基づいて決定します。

get_code(*fullname*)

path から作成された *name* のコードオブジェクトを返します。

get_source(*fullname*)

このローダーが使われたとき、バイトコードファイルのソースがなければ None を返します。

load_module(*name=None*)

ロードするモジュールの名前指定がオプションの、*importlib.abc.Loader.load_module()* の具象実装です。

バージョン 3.6 で非推奨: 代わりに *importlib.abc.Loader.exec_module()* を使用してください。

class importlib.machinery.**ExtensionFileLoader**(*fullname*, *path*)

拡張モジュールのための *importlib.abc.ExecutionLoader* の具象実装です。

fullname 引数はローダーがサポートするモジュールの名前を指定します。*path* 引数は拡張モジュールのファイルへのパスです。

バージョン 3.3 で追加.

name
ローダーがサポートするモジュールの名前です。

path
拡張モジュールへのパスです。

create_module(*spec*)
与えられたスペックから **PEP 489** に従ってモジュールオブジェクトを作成します。

バージョン 3.5 で追加.

exec_module(*module*)
与えられたモジュールオブジェクトを **PEP 489** に従って初期化します。

バージョン 3.5 で追加.

is_package(*fullname*)
EXTENSION_SUFFIXES に基づいて、ファイルパスがパッケージの `__init__` モジュールを指していれば `True` を返します。

get_code(*fullname*)
拡張モジュールにコードオブジェクトがなければ `None` を返します。

get_source(*fullname*)
拡張モジュールにソースコードがなければ `None` を返します。

get_filename(*fullname*)
path を返します。

バージョン 3.4 で追加.

class importlib.machinery.**ModuleSpec**(*name*, *loader*, ***, *origin=None*, *loader_state=None*, *is_package=None*)
モジュールのインポートシステムに関する状態の仕様。これは通常はモジュールの `__spec__` 属性として公開されています。この後の解説では、モジュールオブジェクトから直接利用できる属性で、それぞれの仕様に対応しているものの名前が括弧書きで書かれています。例えば、`module.__spec__.origin == module.__file__` です。ただし、属性の **値** はたいていは同一ですが、2 つのオブジェクトどうしは同期されないため、異なっている可能性があることに注意してください。例えば、モジュールの `__path__` を実行時に更新できますが、`__spec__.submodule_search_locations` に自動的には反映されません。

バージョン 3.4 で追加.

name

(`__name__`)

モジュールの完全修飾名を表す文字列です。

loader

(`__loader__`)

The *Loader* that should be used when loading the module. *Finders* should always set this.

origin

(__file__)

Name of the place from which the module is loaded, e.g. "builtin" for built-in modules and the filename for modules loaded from source. Normally "origin" should be set, but it may be None (the default) which indicates it is unspecified (e.g. for namespace packages).

submodule_search_locations

(__path__)

パッケージの場合サブモジュールを見付けるべき場所を表す文字列のリスト (そうでない場合は None) です。

loader_state

ロード中に使う拡張モジュール指定のデータのコンテナ (または None) です。

cached

(__cached__)

コンパイルされたモジュールを保存すべき場所を表す文字列 (または None) です。

parent

(__package__)

(Read-only) The fully-qualified name of the package under which the module should be loaded as a submodule (or the empty string for top-level modules). For packages, it is the same as __name__.

has_location

モジュールの "origin" 属性がロード可能な場所を参照しているかどうかを示すブール値です。

31.5.6 importlib.util -- インポータのためのユーティリティコード

ソースコード: Lib/importlib/util.py

このモジュールには、**インポーター** の構築を助ける様々なオブジェクトがあります。

importlib.util.MAGIC_NUMBER
: バイトコードバージョン番号を表しているバイト列。バイトコードのロード／書き込みについてヘルプが必要なら *importlib.abc.SourceLoader* を参照してください。

  バージョン 3.4 で追加.

importlib.util.cache_from_source(*path*, *debug_override=None*, ***, *optimization=None*)

ソース *path* に関連付けられたバイトコンパイルされたファイルの **PEP 3147**/**PEP 488** パスを返します。例えば、*path* が /foo/bar/baz.py なら、Python 3.2 の場合返り値は /foo/bar/__pycache__/baz.cpython-32.pyc になります。cpython-32 という文字列は、現在のマジックタグから得られます (マジックタグについては get_tag() を参照; sys.implementation.cache_tag が未定義なら *NotImplementedError* が送出されます。)

optimization パラメータは、バイトコードファイルの最適化レベルを指定するために使われます。空文字列は最適化しないことを表します。したがって、*optimization* が '' のとき /foo/bar/baz.py に対して /foo/bar/__pycache__/baz.cpython-32.pyc というバイトコードパスが返ります。None にするとインタープリタの最適化レベルが使われます。それ以外では値の文字列表現が使われます。したがって、*optimization* が 2 のとき /foo/bar/baz.py に対して /foo/bar/__pycache__/baz.cpython-32.opt-2.pyc というバイトコードパスが返ります。*optimization* の文字列表現は英数字だけが可能で、そうでなければ *ValueError* が上げられます。

debug_override パラメータは deprecated で、システムの __debug__ 値をオーバーライドするために使用できます。True 値は *optimization* を空文字列に設定するのと等価です。False 値は *optimization* を 1 に設定するのと同等です。もし *debug_override* と *optimization* のどちらも None 以外であれば *TypeError* が上げられます。

バージョン 3.4 で追加.

バージョン 3.5 で変更: *optimization* パラメータが追加され、*debug_override* パラメータは deprecated になりました。

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

importlib.util.source_from_cache(*path*)

PEP 3147 ファイル名への *path* が与えられると、関連するソースコードのファイルパスを返します。例えば、*path* が /foo/bar/__pycache__/baz.cpython-32.pyc なら、返されるパスは /foo/bar/baz.py になります。*path* は存在する必要はありませんが、**PEP 3147** または **PEP 488** フォーマットに一致しない場合は *ValueError* が送出されます。sys.implementation.cache_tag が定義されていない場合、*NotImplementedError* が送出されます。

バージョン 3.4 で追加.

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

importlib.util.decode_source(*source_bytes*)

与えられたソースコードを表すバイト列をデコードして、文字列としてそれを一般的な改行形式 (universal newlines) で返します (*importlib.abc.InspectLoader.get_source()* で要求されるように)。

バージョン 3.4 で追加.

importlib.util.resolve_name(*name*, *package*)

相対的なモジュール名を解決して絶対的なものにします。

name の先頭にドットがなければ、単に **name** が返されます。これにより、例えば importlib.util.

resolve_name('sys', __spec__.parent) を使うときに **package** 変数が必要かどうかを確認する必要がなくなります。

name が相対的なモジュール名であるにもかかわらず package が偽値 (例えば None や空文字列) ならば、*ValueError* が送出されます。相対的な名前がそれを含むパッケージから抜け出る (例えば spam パッケージ内から ..bacon を要求する) 場合にも *ValueError* が送出されます。

バージョン 3.3 で追加.

importlib.util.**find_spec**(*name*, *package=None*)

モジュールの *spec* を、オプションで指定された **package** 名に対する相対で検索します。モジュールが *sys.modules* にあれば、sys.modules[name].__spec__ が返されます (ただしスペックが None であるか設定されていなければ *ValueError* が送出されます)。なければ、*sys.meta_path* を使った検索がなされます。スペックが見つからなければ None が返ります。

name がサブモジュールを示している (ドットを含む) 場合、親モジュールは自動的にインポートされます。

name と **package** は import_module() に対するものと同じように機能します。

バージョン 3.4 で追加.

バージョン 3.7 で変更: Raises *ModuleNotFoundError* instead of *AttributeError* if **package** is in fact not a package (i.e. lacks a __path__ attribute).

importlib.util.**module_from_spec**(*spec*)

spec と *spec.loader.create_module* に基づいて新しいモジュールを作ります。

spec.loader.create_module が None を返さない場合は、既に存在するどの属性もリセットされません。また、**spec** にアクセスしたり属性をモジュールに設定したりする際に *AttributeError* 例外が起きても例外は送出されません。

この関数は、新しいモジュールを作る方法として *types.ModuleType* よりも推奨されます。なぜなら、できるだけ多くのインポートコントロールされた属性をモジュールに設定するために **spec** が使用されるからです。

バージョン 3.5 で追加.

@importlib.util.**module_for_loader**

ロードに使う適切なモジュールオブジェクトの選択を扱うための、*importlib.abc.Loader.load_module()* への *decorator* です。このデコレータメソッドのシグニチャは、2 つの位置引数をとることを期待されます (例えば load_module(self, module)) 。第 2 引数はローダーによって使われるモジュール **object** になります。なお、このデコレータは 2 つの引数を想定するため、スタティックメソッドには働きません。

デコレートされたメソッドは、*loader* がロードしようとするモジュールの **name** を受け取ります。そのモジュールが *sys.modules* に見つからなければ新しいモジュールが構築されます。モジュールの出所に関わらず、__loader__ は **self** に設定され、(もし利用可能なら) __package__ は *importlib.abc.InspectLoader.is_package()* の戻り値に基づいて設定されます。これらの属性は、リロードをサポートするために無条件に設定されます。

デコレートされたメソッドによって例外が送出されたとき、モジュールが *sys.modules* に加えられていたら、部分的に初期化されたモジュールが *sys.modules* に残らないよう、そのモジュールは取り除かれます。モジュールが既に *sys.modules* にあったなら、それは残されます。

バージョン 3.3 で変更: `__loader__` および `__package__` は (可能なら) 自動的に設定されます。

バージョン 3.4 で変更: リロードをサポートするために `__name__` `__loader__` `__package__` は無条件に設定されます。

バージョン 3.4 で非推奨: インポート機構はこの関数が提供する全機能を直接実行するようになりました。

@importlib.util.set_loader

返されたモジュールの `__loader__` 属性を設定する、*importlib.abc.Loader.load_module()* への *decorator* です。属性が既に設定されていたら、このデコレータは何もしません。ラップされたメソッド (すなわち `self`) への 1 つ目の位置引数は `__loader__` に設定される値であると仮定されます。

バージョン 3.4 で変更: もし `__loader__` 属性が `None` に設定されていれば、属性が存在しないかのように `__loader__` を設定します。

バージョン 3.4 で非推奨: インポート機構はこれを自動的に考慮するようになりました。

@importlib.util.set_package

`__package__` 属性を戻り値のモジュールに設定するための、*importlib.abc.Loader.load_module()* への *decorator* です。もし `__package__` が設定されていて `None` 以外の値を持っているなら、それは変更されません。

バージョン 3.4 で非推奨: インポート機構はこれを自動的に考慮するようになりました。

importlib.util.spec_from_loader(*name*, *loader*, ***, *origin=None*, *is_package=None*)

この関数は、スペックに不足している情報を埋めるために `InspectLoader.is_package()` のような利用可能な *loader* API を使います。

バージョン 3.4 で追加.

importlib.util.spec_from_file_location(*name*, *location*, ***, *loader=None*, *submodule_search_locations=None*)

ファイルへのパスにもとづいて `ModuleSpec` インスタンスを生成するためのファクトリー関数。不足している情報は、ローダー API を利用してスペックから得られる情報と、モジュールがファイルベースであるという暗黙的な情報によって埋められます。

バージョン 3.4 で追加.

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

importlib.util.source_hash(*source_bytes*)

Return the hash of *source_bytes* as bytes. A hash-based `.pyc` file embeds the *source_hash()* of the corresponding source file's contents in its header.

バージョン 3.7 で追加.

class importlib.util.**LazyLoader**(*loader*)

モジュールが属性アクセスできるようになるまで、モジュールのローダーの実行を遅延するクラス。

This class **only** works with loaders that define *exec_module()* as control over what module type is used for the module is required. For those same reasons, the loader's *create_module()* method must return `None` or a type for which its `__class__` attribute can be mutated along with not using *slots*. Finally, modules which substitute the object placed into *sys.modules* will not work as there is no way to properly replace the module references throughout the interpreter safely; *ValueError* is raised if such a substitution is detected.

注釈: 起動時間が重要なプロジェクトでは、もし決して使われないモジュールがあれば、このクラスを使ってモジュールをロードするコストを最小化できるかもしれません。スタートアップ時間が重要でないプロジェクトでは、遅延されたロードの際に発生して文脈の外で起こるエラーメッセージのため、このクラスの使用は **著しく** 推奨されません。

バージョン 3.5 で追加.

バージョン 3.6 で変更: Began calling *create_module()*, removing the compatibility warning for *importlib.machinery.BuiltinImporter* and *importlib.machinery.ExtensionFileLoader*.

classmethod **factory**(*loader*)

遅延ローダを生成する callable を返すスタティックメソッド。これは、ローダーをインスタンスとしてではなくクラスとして渡すような状況において使われることを意図しています。

```
suffixes = importlib.machinery.SOURCE_SUFFIXES
loader = importlib.machinery.SourceFileLoader
lazy_loader = importlib.util.LazyLoader.factory(loader)
finder = importlib.machinery.FileFinder(path, (lazy_loader, suffixes))
```

31.5.7 使用例

プログラムからのインポート

プログラムからモジュールをインポートするには、*importlib.import_module()* を使ってください。

```
import importlib

itertools = importlib.import_module('itertools')
```

モジュールがインポートできるか確認する

インポートを実際に行わずに、あるモジュールがインポートできるかを知る必要がある場合は、*importlib.util.find_spec()* を使ってください。

```
import importlib.util
import sys

# For illustrative purposes.
name = 'itertools'

if name in sys.modules:
    print(f"{name!r} already in sys.modules")
elif (spec := importlib.util.find_spec(name)) is not None:
    # If you chose to perform the actual import ...
    module = importlib.util.module_from_spec(spec)
    sys.modules[name] = module
    spec.loader.exec_module(module)
    print(f"{name!r} has been imported")
else:
    print(f"can't find the {name!r} module")
```

ソースファイルから直接インポートする

To import a Python source file directly, use the following recipe (Python 3.5 and newer only):

```
import importlib.util
import sys

# For illustrative purposes.
import tokenize
file_path = tokenize.__file__
module_name = tokenize.__name__

spec = importlib.util.spec_from_file_location(module_name, file_path)
module = importlib.util.module_from_spec(spec)
sys.modules[module_name] = module
spec.loader.exec_module(module)
```

インポーターのセットアップ

For deep customizations of import, you typically want to implement an *importer*. This means managing both the *finder* and *loader* side of things. For finders there are two flavours to choose from depending on your needs: a *meta path finder* or a *path entry finder*. The former is what you would put on *sys.meta_path* while the latter is what you create using a *path entry hook* on *sys.path_hooks* which works with *sys.path* entries to potentially create a finder. This example will show you how to register your own importers so that import will use them (for creating an importer for yourself, read the documentation for the appropriate classes defined within this package):

```
import importlib.machinery
import sys

# For illustrative purposes only.
SpamMetaPathFinder = importlib.machinery.PathFinder
SpamPathEntryFinder = importlib.machinery.FileFinder
loader_details = (importlib.machinery.SourceFileLoader,
                  importlib.machinery.SOURCE_SUFFIXES)

# Setting up a meta path finder.
# Make sure to put the finder in the proper location in the list in terms of
# priority.
sys.meta_path.append(SpamMetaPathFinder)

# Setting up a path entry finder.
# Make sure to put the path hook in the proper location in the list in terms
# of priority.
sys.path_hooks.append(SpamPathEntryFinder.path_hook(loader_details))
```

Approximating `importlib.import_module()`

Import itself is implemented in Python code, making it possible to expose most of the import machinery through importlib. The following helps illustrate the various APIs that importlib exposes by providing an approximate implementation of *importlib.import_module()* (Python 3.4 and newer for the importlib usage, Python 3.6 and newer for other parts of the code).

```
import importlib.util
import sys

def import_module(name, package=None):
    """An approximate implementation of import."""
    absolute_name = importlib.util.resolve_name(name, package)
    try:
        return sys.modules[absolute_name]
    except KeyError:
        pass

    path = None
    if '.' in absolute_name:
        parent_name, _, child_name = absolute_name.rpartition('.')
        parent_module = import_module(parent_name)
        path = parent_module.__spec__.submodule_search_locations
    for finder in sys.meta_path:
        spec = finder.find_spec(absolute_name, path)
        if spec is not None:
            break
    else:
        msg = f'No module named {absolute_name!r}'
        raise ModuleNotFoundError(msg, name=absolute_name)
    module = importlib.util.module_from_spec(spec)
```

(次のページに続く)

(前のページからの続き)

```
    sys.modules[absolute_name] = module
    spec.loader.exec_module(module)
    if path is not None:
        setattr(parent_module, child_name, module)
    return module
```

31.6 importlib.metadata を使う

バージョン 3.8 で追加.

Source code: Lib/importlib/metadata.py

注釈: This functionality is provisional and may deviate from the usual version semantics of the standard library.

`importlib.metadata` is a library that provides for access to installed package metadata. Built in part on Python's import system, this library intends to replace similar functionality in the entry point API and metadata API of `pkg_resources`. Along with *`importlib.resources`* in Python 3.7 and newer (backported as importlib_resources for older versions of Python), this can eliminate the need to use the older and less efficient `pkg_resources` package.

By "installed package" we generally mean a third-party package installed into Python's `site-packages` directory via tools such as pip. Specifically, it means a package with either a discoverable `dist-info` or `egg-info` directory, and metadata defined by **PEP 566** or its older specifications. By default, package metadata can live on the file system or in zip archives on *`sys.path`*. Through an extension mechanism, the metadata can live almost anywhere.

31.6.1 概要

Let's say you wanted to get the version string for a package you've installed using `pip`. We start by creating a virtual environment and installing something into it:

```
$ python3 -m venv example
$ source example/bin/activate
(example) $ pip install wheel
```

以下のように実行することで、`wheel` のバージョン文字列を取得することができます：

```
(example) $ python
>>> from importlib.metadata import version
>>> version('wheel')
'0.32.3'
```

You can also get the set of entry points keyed by group, such as `console_scripts`, `distutils.commands` and others. Each group contains a sequence of *EntryPoint* objects.

ディストリビューションのメタデータ:: を取得することができます。

```
>>> list(metadata('wheel'))
['Metadata-Version', 'Name', 'Version', 'Summary', 'Home-page', 'Author', 'Author-email',
↪'Maintainer', 'Maintainer-email', 'License', 'Project-URL', 'Project-URL', 'Project-URL',
↪'Keywords', 'Platform', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier',
↪ 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier',
↪'Classifier', 'Requires-Python', 'Provides-Extra', 'Requires-Dist', 'Requires-Dist']
```

また、**配布物のバージョン番号** を取得し、**構成ファイル** をリストアップし、配布物の **配布物の要件** のリストを取得することができます。

31.6.2 機能 API

本パッケージは、公開 API を通じて以下の機能を提供します。

エントリポイント

The `entry_points()` function returns a dictionary of all entry points, keyed by group. Entry points are represented by `EntryPoint` instances; each `EntryPoint` has a `.name`, `.group`, and `.value` attributes and a `.load()` method to resolve the value.

```
>>> eps = entry_points()
>>> list(eps)
['console_scripts', 'distutils.commands', 'distutils.setup_keywords', 'egg_info.writers',
↪'setuptools.installation']
>>> scripts = eps['console_scripts']
>>> wheel = [ep for ep in scripts if ep.name == 'wheel'][0]
>>> wheel
EntryPoint(name='wheel', value='wheel.cli:main', group='console_scripts')
>>> main = wheel.load()
>>> main
<function main at 0x103528488>
```

The `group` and `name` are arbitrary values defined by the package author and usually a client will wish to resolve all entry points for a particular group. Read the setuptools docs for more information on entrypoints, their definition, and usage.

配布物メタデータ

Every distribution includes some metadata, which you can extract using the `metadata()` function:

```
>>> wheel_metadata = metadata('wheel')
```

The keys of the returned data structure[*1] name the metadata keywords, and their values are returned unparsed from the distribution metadata:

```
>>> wheel_metadata['Requires-Python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
```

配布物バージョン

The `version()` function is the quickest way to get a distribution's version number, as a string:

```
>>> version('wheel')
'0.32.3'
```

配布物ファイル

You can also get the full set of files contained within a distribution. The `files()` function takes a distribution package name and returns all of the files installed by this distribution. Each file object returned is a `PackagePath`, a *pathlib.Path* derived object with additional `dist`, `size`, and `hash` properties as indicated by the metadata. For example:

```
>>> util = [p for p in files('wheel') if 'util.py' in str(p)][0]
>>> util
PackagePath('wheel/util.py')
>>> util.size
859
>>> util.dist
<importlib.metadata._hooks.PathDistribution object at 0x101e0cef0>
>>> util.hash
<FileHash mode: sha256 value: bYkw5oMccfazVCoYQwKkkemoVyMAFoR34mmKBx8R1NI>
```

ファイルを取得したら、その内容を読むこともできます:

```
>>> print(util.read_text())
import base64
import sys
...
def as_bytes(s):
    if isinstance(s, text_type):
```

(次のページに続く)

[*1] Technically, the returned distribution metadata object is an *email.message.EmailMessage* instance, but this is an implementation detail, and not part of the stable API. You should only use dictionary-like methods and syntax to access the metadata contents.

(前のページからの続き)

```
        return s.encode('utf-8')
    return s
```

In the case where the metadata file listing files (RECORD or SOURCES.txt) is missing, `files()` will return `None`. The caller may wish to wrap calls to `files()` in always_iterable or otherwise guard against this condition if the target distribution is not known to have the metadata present.

配布物の要件

To get the full set of requirements for a distribution, use the `requires()` function:

```
>>> requires('wheel')
["pytest (>=3.0.0) ; extra == 'test'", "pytest-cov ; extra == 'test'"]
```

31.6.3 Distributions

While the above API is the most common and convenient usage, you can get all of that information from the `Distribution` class. A `Distribution` is an abstract object that represents the metadata for a Python package. You can get the `Distribution` instance:

```
>>> from importlib.metadata import distribution
>>> dist = distribution('wheel')
```

したがって、バージョン情報を取得する別の方法として、`Distribution` インスタンスを使用します:

```
>>> dist.version
'0.32.3'
```

`Distribution` インスタンスには、あらゆる種類の追加メタデータが用意されています:

```
>>> dist.metadata['Requires-Python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
>>> dist.metadata['License']
'MIT'
```

The full set of available metadata is not described here. See **PEP 566** for additional details.

31.6.4 検索アルゴリズムの拡張

Because package metadata is not available through *sys.path* searches, or package loaders directly, the metadata for a package is found through import system finders. To find a distribution package's metadata, `importlib.metadata` queries the list of *meta path finders* on *sys.meta_path*.

The default `PathFinder` for Python includes a hook that calls into `importlib.metadata.MetadataPathFinder` for finding distributions loaded from typical file-system-based paths.

抽象クラス *importlib.abc.MetaPathFinder* は Python の import システムによってファインダーに期待されるインターフェイスを定義しています。`importlib.metadata` はこのプロトコルを拡張し、*sys.meta_path* からファインダーにオプションの `find_distributions` を呼び出すことができるようにし、この拡張インターフェースを `DistributionFinder` 抽象基底クラスとして提示し、この抽象メソッドを定義しています:

```
@abc.abstractmethod
def find_distributions(context=DistributionFinder.Context()):
    """Return an iterable of all Distribution instances capable of
    loading the metadata for packages for the indicated ``context``.
    """
```

The `DistributionFinder.Context` object provides `.path` and `.name` properties indicating the path to search and names to match and may supply other relevant context.

つまり、ファイルシステム以外の場所にある配布パッケージのメタデータを見つけるには、`Distribution` をサブクラス化して抽象メソッドを実装します。そして、カスタムファインダーから `find_distributions()` メソッドで、派生した `Distribution` のインスタンスを返します。

脚注

第

THIRTYTWO

PYTHON 言語サービス

Python には Python 言語を使って作業するときに役に立つモジュールがたくさん提供されています。これらのモジュールはトークンの切り出し、パース、構文解析、バイトコードのディスアセンブリおよびその他のさまざまな機能をサポートしています。

これらのモジュールには、次のものが含まれています:

32.1 parser --- Python 解析木にアクセスする

parser モジュールは Python の内部パーサとバイトコード・コンパイラへのインターフェイスを提供します。このインターフェイスの第一の目的は、Python コードから Python の式の解析木を編集したり、これから実行可能なコードを作成したりできるようにすることです。これは任意の Python コードの断片を文字列として構文解析や変更を行うより良い方法です。なぜなら、構文解析がアプリケーションを作成するコードと同じ方法で実行されるからです。その上、高速です。

注釈: Python 2.5 以降、抽象構文木 (AST) の生成・コンパイルの段階に割り込むには *ast* モジュールを使うのがずっとお手軽です。

このモジュールについて注意すべきことが少しあります。それは作成したデータ構造を利用するために重要なことです。この文書は Python コードの解析木を編集するためのチュートリアルではありませんが、*parser* モジュールを使った例をいくつか示しています。

もっとも重要なことは、内部パーサが処理する Python の文法についてよく理解しておく必要があるということです。言語の文法に関する完全な情報については、reference-index を参照してください。標準の Python ディストリビューションに含まれるファイル `Grammar/Grammar` の中で定義されている文法仕様から、パーサ自身は作成されています。このモジュールが作成する ST オブジェクトの中に格納される解析木は、下で説明する *expr()* または *suite()* 関数によって作られるときに内部パーサから実際に出力されるものです。*sequence2st()* が作る ST オブジェクトは忠実にこれらの構造をシミュレートしています。言語の形式文法が改訂されるために、" 正しい" と考えられるシーケンスの値が Python のあるバージョンから別のバージョンで変化することがあるということに注意してください。しかし、Python のあるバージョンから別のバージョンへテキストのソースのままコードを移せば、目的のバージョンで正しい解析木を常に作成できます。た

だし、インタープリタの古いバージョンへ移行する際に、最近の言語コンストラクトをサポートしていないことがあるという制限だけがあります。同一のメジャーバージョンの範囲内では、通常はソースコードに前方互換性がありますが、一般的に解析木はあるバージョンから別のバージョンへの互換性がありません。

st2list() または *st2tuple()* から返されるシーケンスのそれぞれの要素は単純な形式です。文法の非終端要素を表すシーケンスは常に一より大きい長さを持ちます。最初の要素は文法の生成規則を識別する整数です。これらの整数は C ヘッダファイル `Include/graminit.h` と Python モジュール *symbol* の中の特定のシンボル名です。シーケンスに付け加えられている各要素は、入力文字列の中で認識されたままの形で生成規則の構成要素を表しています: これらは常に親と同じ形式を持つシーケンスです。この構造の注意すべき重要な側面は、`if_stmt` の中のキーワード `if` のような親ノードの型を識別するために使われるキーワードがいかなる特別な扱いもなくノードツリーに含まれているということです。例えば、`if` キーワードはタプル `(1, 'if')` と表されます。ここで、`1` は、ユーザが定義した変数名と関数名を含むすべての `NAME` トークンに対応する数値です。行番号情報が必要なときに返される別の形式では、同じトークンが `(1, 'if', 12)` のように表されます。ここでは、`12` が終端記号の見つかった行番号を表しています。

終端要素は同じ方法で表現されますが、子の要素や識別されたソーステキストの追加は全くありません。上記の `if` キーワードの例が代表的なものです。終端記号のいろいろな型は、C ヘッダファイル `Include/token.h` と Python モジュール *token* で定義されています。

ST オブジェクトはこのモジュールの機能をサポートするために必要ありませんが、三つの目的から提供されています: アプリケーションが複雑な解析木を処理するコストを償却するため、Python のリストやタプル表現に比べてメモリ空間を保全する解析木表現を提供するため、解析木を操作する追加モジュールを C で作ることを簡単にするため。ST オブジェクトを使っていることを隠すために、簡単な ” ラッパー” クラスを Python で作ることができます。

parser モジュールは二、三の別々の目的のために関数を定義しています。もっとも重要な目的は ST オブジェクトを作ることと、ST オブジェクトを解析木とコンパイルされたコードオブジェクトのような他の表現に変換することです。しかし、ST オブジェクトで表現された解析木の型を調べるために役に立つ関数もあります。

参考:

symbol **モジュール**　解析木の内部ノードを表す便利な定数。

token **モジュール**　便利な解析木の葉のノードを表す定数とノード値をテストするための関数。

32.1.1 ST オブジェクトを作成する

ST オブジェクトはソースコードあるいは解析木から作られます。ST オブジェクトをソースから作るときは、`'eval'` と `'exec'` 形式を作成するために別々の関数が使われます。

parser.expr(*source*)

まるで `compile(source, 'file.py', 'eval')` への入力であるかのように、*expr()* 関数はパラメータ *source* を構文解析します。解析が成功した場合は、ST オブジェクトは内部解析木表現を保持するために作成されます。そうでなければ、適切な例外を発生させます。

parser.suite(*source*)

まるで `compile(source, 'file.py', 'exec')` への入力であるかのように、*suite()* 関数はパラメータ *source* を構文解析します。解析が成功した場合は、ST オブジェクトは内部解析木表現を保持するために作成されます。そうでなければ、適切な例外を発生させます。

parser.sequence2st(*sequence*)

この関数はシーケンスとして表現された解析木を受け取り、可能ならば内部表現を作ります。木が Python の文法に合っていることと、すべてのノードが Python のホストバージョンで有効なノード型であることを確認した場合は、ST オブジェクトが内部表現から作成されて呼び出し側へ返されます。内部表現の作成に問題があるならば、あるいは木が正しいと確認できないならば、*ParserError* 例外を発生します。この方法で作られた ST オブジェクトが正しくコンパイルできると決めつけない方がよいでしょう。ST オブジェクトが *compilest()* へ渡されたとき、コンパイルによって送出された通常の例外がまだ発生するかもしれません。これは (*MemoryError* 例外のような) 構文に関係していない問題を示すのかもしれないし、`del f(0)` を解析した結果のようなコンストラクトが原因であるかもしれません。このようなコンストラクトは Python のパーサを逃れますが、バイトコードインタープリタによってチェックされます。

終端トークンを表すシーケンスは、`(1, 'name')` 形式の二つの要素のリストか、または `(1, 'name', 56)` 形式の三つの要素のリストです。三番目の要素が存在する場合は、有効な行番号だとみなされます。行番号が指定されるのは、入力木の終端記号の一部に対してです。

parser.tuple2st(*sequence*)

これは *sequence2st()* と同じ関数です。このエントリポイントは後方互換性のために維持されています。

32.1.2 ST オブジェクトを変換する

作成するために使われた入力に関係なく、ST オブジェクトはリスト木またはタプル木として表される解析木へ変換されるか、または実行可能なオブジェクトへコンパイルされます。解析木は行番号情報を持って、あるいは持たずに抽出されます。

parser.st2list(*st*, *line_info=False*, *col_info=False*)

この関数は呼び出し側から *st* に ST オブジェクトを受け取り、解析木と等価な Python のリストを返します。結果のリスト表現はインスペクションあるいはリスト形式の新しい解析木の作成に使うことができます。リスト表現を作るためにメモリが利用できる限り、この関数は失敗しません。解析木がインスペクションのためだけにつかわれるならば、メモリの消費量と断片化を減らすために *st2tuple()* を代わりに使うべきです。リスト表現が必要とされるとき、この関数はタプル表現を取り出して入れ子のリストに変換するよりかなり高速です。

line_info が真ならば、トークンを表すリストの三番目の要素として行番号情報がすべての終端トークンに含まれます。与えられた行番号はトークン **が終わる** 行を指定していることに注意してください。フラグが偽または省略された場合は、この情報は省かれます。

parser.st2tuple(*st*, *line_info=False*, *col_info=False*)

この関数は呼び出し側から *st* に ST オブジェクトを受け取り、解析木と等価な Python のタプルを返します。リストの代わりにタプルを返す以外は、この関数は *st2list()* と同じです。

line_info が真ならば、トークンを表すリストの三番目の要素として行番号情報がすべての終端トークンに含まれます。フラグが偽または省略された場合は、この情報は省かれます。

parser.compilest(*st*, *filename='<syntax-tree>'*)

組み込みの *exec()* や *eval()* 関数への呼び出しとして使えるコードオブジェクトを生成するために、Python バイトコードコンパイラを ST オブジェクトに対して呼び出すことができます。この関数はコンパイラへのインターフェイスを提供し、*filename* パラメータで指定されるソースファイル名を使って、*st* からパーサへ内部解析木を渡します。*filename* に与えられるデフォルト値は、ソースが ST オブジェクトだったことを示唆しています。

ST オブジェクトをコンパイルすることは、コンパイルに関する例外を引き起こすことになるかもしれません。例としては、`del f(0)` の解析木によって発生させられる *SyntaxError* があります: この文は Python の形式文法としては正しいと考えられますが、正しい言語コンストラクトではありません。この状況に対して発生する *SyntaxError* は、実際には Python バイトコンパイラによって通常作り出されます。これが *parser* モジュールがこの時点で例外を発生できる理由です。解析木のインスペクションを行うことで、コンパイルが失敗するほとんどの原因をプログラムによって診断することができます。

32.1.3 ST オブジェクトに対する問い合わせ

ST が式または suite として作成されたかどうかをアプリケーションが決定できるようにする二つの関数が提供されています。これらの関数のどちらも、ST が *expr()* または *suite()* を通してソースコードから作られたかどうか、あるいは、*sequence2st()* を通して解析木から作られたかどうかを決定できません。

parser.isexpr(*st*)

st が `'eval'` 形式を表している場合に、この関数は `True` を返します。そうでなければ、`False` を返します。これは役に立ちます。なぜならば、通常は既存の組み込み関数を使ってもコードオブジェクトに対してこの情報を問い合わせることができないからです。このどちらのようにも *compilest()* によって作成されたコードオブジェクトに問い合わせることはできませんし、そのコードオブジェクトは組み込み *compile()* 関数によって作成されたコードオブジェクトと同じであることに注意してください。

parser.issuite(*st*)

ST オブジェクトが (通常 "suite" として知られる) `'exec'` 形式を表しているかどうかを報告するという点で、この関数は *isexpr()* に酷似しています。追加の構文が将来サポートされるかもしれないので、この関数が `not isexpr(st)` と等価であるとみなすのは安全ではありません。

32.1.4 例外とエラー処理

parser モジュールは例外を一つ定義していますが、Python ランタイム環境の他の部分が提供する別の組み込み例外を発生させることもあります。各関数が発生させる例外の情報については、それぞれ関数を参照してください。

exception parser.ParserError

parser モジュール内部で障害が起きたときに発生する例外。普通の構文解析中に発生する組み込みの *SyntaxError* ではなく、一般的に妥当性確認が失敗した場合に引き起こされます。例外の引数として

は、障害の理由を説明する文字列である場合と、*sequence2st()* へ渡される解析木の中の障害を引き起こすシーケンスを含むタプルと説明用の文字列である場合があります。モジュール内の他の関数の呼び出しは単純な文字列値を検出すればよいだけですが、*sequence2st()* の呼び出しはどちらの例外の型も処理できる必要があります。

普通は構文解析とコンパイル処理によって発生する例外を、関数 *compilest()* 、*expr()* および *suite()* が発生させることに注意してください。このような例外には組み込み例外 *MemoryError* 、*OverflowError* 、*SyntaxError* および *SystemError* が含まれます。こうした場合には、これらの例外が通常その例外に関係する全ての意味を伝えます。詳細については、各関数の説明を参照してください。

32.1.5 ST オブジェクト

ST オブジェクト間の順序と等値性の比較がサポートされています。(*pickle* モジュールを使った) ST オブジェクトのピクルス化もサポートされています。

parser.**STType**
: *expr()* 、*suite()* と *sequence2st()* が返すオブジェクトの型。

ST オブジェクトは次のメソッドを持っています:

ST.**compile**(*filename='<syntax-tree>'*)
: `compilest(st, filename)` と同じ。

ST.**isexpr**()
: `isexpr(st)` と同じ。

ST.**issuite**()
: `issuite(st)` と同じ。

ST.**tolist**(*line_info=False*, *col_info=False*)
: `st2list(st, line_info, col_info)` と同じ。

ST.**totuple**(*line_info=False*, *col_info=False*)
: `st2tuple(st, line_info, col_info)` と同じ。

32.1.6 例: `compile()` のエミュレーション

たくさんの有用な演算を構文解析とバイトコード生成の間に行うことができますが、もっとも単純な演算は何もしないことです。このため、*parser* モジュールを使って中間データ構造を作ることは次のコードと等価です

```
>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
>>> eval(code)
10
```

parser モジュールを使った等価な演算はやや長くなりますが、ST オブジェクトとして中間内部解析木が維持されるようにします:

```
>>> import parser
>>> st = parser.expr('a + 5')
>>> code = st.compile('file.py')
>>> a = 5
>>> eval(code)
10
```

ST とコードオブジェクトの両方が必要なアプリケーションでは、このコードを簡単に利用できる関数にまとめることができます:

```
import parser

def load_suite(source_string):
    st = parser.suite(source_string)
    return st, st.compile()

def load_expression(source_string):
    st = parser.expr(source_string)
    return st, st.compile()
```

32.2 ast --- 抽象構文木

ソースコード: Lib/ast.py

---

ast モジュールは、Python アプリケーションで Python の抽象構文木を処理しやすくするものです。抽象構文そのものは、Python のリリースごとに変化する可能性があります。このモジュールを使用すると、現在の文法をプログラム上で知る助けになるでしょう。

抽象構文木を作成するには、ast.PyCF_ONLY_AST を組み込み関数 *compile()* のフラグとして渡すか、あるいはこのモジュールで提供されているヘルパー関数 *parse()* を使います。その結果は、*ast.AST* を継承したクラスのオブジェクトのツリーとなります。抽象構文木は組み込み関数 *compile()* を使って Python コード・オブジェクトにコンパイルすることができます。

32.2.1 Node クラス

class ast.AST

このクラスは全ての AST ノード・クラスの基底です。実際のノード・クラスは 後ほど 示す Parser/Python.asdl ファイルから派生したものです。これらのクラスは _ast C モジュールで定義され、*ast* にもエクスポートし直されています。

抽象文法の左辺のシンボル一つずつにそれぞれ一つのクラスがあります (たとえば ast.stmt や ast.expr)。それに加えて、右辺のコンストラクタ一つずつにそれぞれ一つのクラスがあり、これらのクラスは左辺のツリーのクラスを継承しています。たとえば、ast.BinOp は ast.expr から継承していま

す。代替を伴った生成規則 (production rules with alternatives) (別名 "sums") の場合、左辺は抽象クラスとなり、特定のコンストラクタ・ノードのインスタンスのみが作成されます。

_fields

各具象クラスは属性 *_fields* を持っており、すべての子ノードの名前をそこに保持しています。

具象クラスのインスタンスは、各子ノードに対してそれぞれひとつの属性を持っています。この属性は、文法で定義された型となります。たとえば `ast.BinOp` のインスタンスは `left` という属性を持っており、その型は `ast.expr` です。

これらの属性が、文法上 (クエスチョンマークを用いて) オプションであるとマークされている場合は、その値が `None` となることもあります。属性が 0 個以上の複数の値をとりうる場合 (アスタリスクでマークされている場合) は、値は Python のリストで表されます。全ての属性は AST を *compile()* でコンパイルする際には存在しなければならず、そして妥当な値でなければなりません。

lineno

col_offset

end_lineno

end_col_offset

Instances of `ast.expr` and `ast.stmt` subclasses have *lineno*, *col_offset*, *lineno*, and *col_offset* attributes. The *lineno* and *end_lineno* are the first and last line numbers of source text span (1-indexed so the first line is line 1) and the *col_offset* and *end_col_offset* are the corresponding UTF-8 byte offsets of the first and last tokens that generated the node. The UTF-8 offset is recorded because the parser uses UTF-8 internally.

コンパイラは終了位置を必要としないことに注意してください。このため終了位置は省略可能です。終了位置を示すオフセットは最後のシンボルの **後の位置** になります。例えば一行で書かれた式のソースコードのセグメントは `source_line[node.col_offset : node.end_col_offset]` により取得できます。

クラス `ast.T` のコンストラクタは引数を次のように解析します:

- 位置引数があるとすれば、`T._fields` にあるのと同じだけの個数が無ければなりません。これらの引数はそこにある名前を持った属性として割り当てられます。
- キーワード引数があるとすれば、それらはその名前の属性にその値を割り当てられます。

たとえば、`ast.UnaryOp` ノードを生成して属性を埋めるには、次のようにすることができます

```
node = ast.UnaryOp()
node.op = ast.USub()
node.operand = ast.Constant()
node.operand.value = 5
node.operand.lineno = 0
node.operand.col_offset = 0
node.lineno = 0
node.col_offset = 0
```

もしくはよりコンパクトにも書けます

```
node = ast.UnaryOp(ast.USub(), ast.Constant(5, lineno=0, col_offset=0),
                   lineno=0, col_offset=0)
```

バージョン 3.8 で変更: ast.Constant が全ての定数に使われるようになりました。

バージョン 3.8 で非推奨: ast.Num、ast.Str、ast.Bytes、ast.NameConstant そして ast.Ellipsis は現バージョンまで使用可能ですが、将来の Python リリースで削除される予定です。削除されるまでの間、これらをインスタンス化すると、異なるクラスのインスタンスが返されます。

32.2.2 抽象文法 (Abstract Grammar)

抽象文法は、現在次のように定義されています:

```
-- ASDL's 5 builtin types are:
-- identifier, int, string, object, constant

module Python
{
    mod = Module(stmt* body, type_ignore *type_ignores)
        | Interactive(stmt* body)
        | Expression(expr body)
        | FunctionType(expr* argtypes, expr returns)

        -- not really an actual node but useful in Jython's typesystem.
        | Suite(stmt* body)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr? returns,
                       string? type_comment)
          | AsyncFunctionDef(identifier name, arguments args,
                             stmt* body, expr* decorator_list, expr? returns,
                             string? type_comment)

          | ClassDef(identifier name,
             expr* bases,
             keyword* keywords,
             stmt* body,
             expr* decorator_list)
          | Return(expr? value)

          | Delete(expr* targets)
          | Assign(expr* targets, expr value, string? type_comment)
          | AugAssign(expr target, operator op, expr value)
          -- 'simple' indicates that we annotate simple name without parens
          | AnnAssign(expr target, expr annotation, expr? value, int simple)

          -- use 'orelse' because else is a keyword in target languages
          | For(expr target, expr iter, stmt* body, stmt* orelse, string? type_comment)
          | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse, string? type_comment)
          | While(expr test, stmt* body, stmt* orelse)
```

(次のページに続く)

(前のページからの続き)

```
        | If(expr test, stmt* body, stmt* orelse)
        | With(withitem* items, stmt* body, string? type_comment)
        | AsyncWith(withitem* items, stmt* body, string? type_comment)

        | Raise(expr? exc, expr? cause)
        | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
        | Assert(expr test, expr? msg)

        | Import(alias* names)
        | ImportFrom(identifier? module, alias* names, int? level)

        | Global(identifier* names)
        | Nonlocal(identifier* names)
        | Expr(expr value)
        | Pass | Break | Continue

        -- XXX Jython will be different
        -- col_offset is the byte offset in the utf8 string the parser uses
        attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

        -- BoolOp() can use left & right?
    expr = BoolOp(boolop op, expr* values)
         | NamedExpr(expr target, expr value)
         | BinOp(expr left, operator op, expr right)
         | UnaryOp(unaryop op, expr operand)
         | Lambda(arguments args, expr body)
         | IfExp(expr test, expr body, expr orelse)
         | Dict(expr* keys, expr* values)
         | Set(expr* elts)
         | ListComp(expr elt, comprehension* generators)
         | SetComp(expr elt, comprehension* generators)
         | DictComp(expr key, expr value, comprehension* generators)
         | GeneratorExp(expr elt, comprehension* generators)
         -- the grammar constrains where yield expressions can occur
         | Await(expr value)
         | Yield(expr? value)
         | YieldFrom(expr value)
         -- need sequences for compare to distinguish between
         -- x < 4 < 3 and (x < 4) < 3
         | Compare(expr left, cmpop* ops, expr* comparators)
         | Call(expr func, expr* args, keyword* keywords)
         | FormattedValue(expr value, int? conversion, expr? format_spec)
         | JoinedStr(expr* values)
         | Constant(constant value, string? kind)

         -- the following expression can appear in assignment context
         | Attribute(expr value, identifier attr, expr_context ctx)
         | Subscript(expr value, slice slice, expr_context ctx)
         | Starred(expr value, expr_context ctx)
         | Name(identifier id, expr_context ctx)
         | List(expr* elts, expr_context ctx)
```

(次のページに続く)

(前のページからの続き)

```
        | Tuple(expr* elts, expr_context ctx)

         -- col_offset is the byte offset in the utf8 string the parser uses
         attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

    expr_context = Load | Store | Del | AugLoad | AugStore | Param

    slice = Slice(expr? lower, expr? upper, expr? step)
          | ExtSlice(slice* dims)
          | Index(expr value)

    boolop = And | Or

    operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
                 | RShift | BitOr | BitXor | BitAnd | FloorDiv

    unaryop = Invert | Not | UAdd | USub

    cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

    comprehension = (expr target, expr iter, expr* ifs, int is_async)

    excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
                    attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

    arguments = (arg* posonlyargs, arg* args, arg? vararg, arg* kwonlyargs,
                 expr* kw_defaults, arg? kwarg, expr* defaults)

    arg = (identifier arg, expr? annotation, string? type_comment)
           attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

    -- keyword arguments supplied to call (NULL identifier for **kwargs)
    keyword = (identifier? arg, expr value)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)

    withitem = (expr context_expr, expr? optional_vars)

    type_ignore = TypeIgnore(int lineno, string tag)
}
```

32.2.3 ast ヘルパー

ノード・クラスの他に、*ast* モジュールは以下のような抽象構文木をトラバースするためのユーティリティ関数やクラスも定義しています:

ast.parse(*source*, *filename='<unknown>'*, *mode='exec'*, ***, *type_comments=False*, *feature_version=None*)

source を解析して AST ノードにします。compile(source, filename, mode, ast.PyCF_ONLY_AST) と等価です。

type_comments=True が与えられると、パーサは **PEP 484** および **PEP 526** で規定された型コメントをチェックし、返すように修正されます。これは ast.PyCF_TYPE_COMMENTS を追加したフラグを *compile()* に渡すことと等価です。パーサは不適切な場所に配置された型コメントに対してシンタックスエラーをレポートします。このフラグがない場合、型コメントは無視されて AST ノードの type_comment フィールドは常に None になります。さらに、# type: ignore コメントの位置は Module の type_ignores 属性として返されます (それ以外の場合は常に空のリストになります)。

さらに mode が 'func_type' の場合、入力構文は、たとえば (str, int) -> List[str] のような **PEP 484** の ” シグネチャ型コメント (signature type comments)” に対応するように修正されます。

また、feature_version を (major, minor) のタプルに設定すると、パーサは指定された Python バージョンの文法で構文解析を試みます。今のところ major は 3 でなければなりません。たとえば、feature_version=(3, 4) と設定すると async と await を変数名として使うことが可能になります。サポートされている最低のバージョンは (3, 4); 最高のバージョンは sys.version_info[0:2] です。

> **警告:** 十分に大きい文字列や複雑な文字列によって Python の抽象構文木コンパイラのスタックの深さの限界を越えることで、Python インタプリタをクラッシュさせることができます。

バージョン 3.8 で変更: type_comments、mode='func_type'、“feature_version“が追加されました。

ast.literal_eval(*node_or_string*)

式ノードまたは Python のリテラルまたはコンテナのディスプレイ表現を表す文字列を安全に評価します。与えられる文字列またはノードは次のリテラルのみからなるものに限られます: 文字列、バイト列、数、タプル、リスト、辞書、集合、ブール値、None 。

この関数は Python の式を含んだ信頼出来ない出どころからの文字列を、値自身を解析することなしに安全に評価するのに使えます。この関数は、例えば演算や添え字を含んだ任意の複雑な表現を評価するのには使えません。

> **警告:** 十分に大きい文字列や複雑な文字列によって Python の抽象構文木コンパイラのスタックの深さの限界を越えることで、Python インタプリタをクラッシュさせることができます。

バージョン 3.2 で変更: バイト列リテラルと集合リテラルが受け取れるようになりました。

ast.**get_docstring**(*node*, *clean=True*)

与えられた *node* (これは FunctionDef, AsyncFunctionDef, ClassDef, Module のいずれかのノードでなければなりません) のドキュメント文字列を返します。もしドキュメント文字列が無ければ None を返します。*clean* が真ならば、ドキュメント文字列のインデントを *inspect.cleandoc()* を用いて一掃します。

バージョン 3.5 で変更: AsyncFunctionDef がサポートされました。

ast.**get_source_segment**(*source*, *node*, ***, *padded=False*)

source のうちで *node* を生成したソースコードのセグメントを取得します。位置情報 (lineno, end_lineno, col_offset, または end_col_offset) が欠けている場合 None を返します。

padded が True の場合、複数行にわたる文の最初の行が元の位置に一致するように空白文字でパディングされます。

バージョン 3.8 で追加.

ast.**fix_missing_locations**(*node*)

compile() はノード・ツリーをコンパイルする際、lineno と col_offset 両属性をサポートする全てのノードに対しそれが存在するものと想定します。生成されたノードに対しこれらを埋めて回るのはどちらかというと退屈な作業なので、このヘルパーが再帰的に二つの属性がセットされていないものに親ノードと同じ値をセットしていきます。再帰の出発点が *node* です。

ast.**increment_lineno**(*node*, *n=1*)

node で始まるツリー内の各ノードの行番号と終了行番号を *n* ずつ増やします。これはファイルの中で別の場所に ” コードを移動する” ときに便利です。

ast.**copy_location**(*new_node*, *old_node*)

ソースの場所 (lineno, col_offset, end_lineno, および end_col_offset) を *old_node* から *new_node* に可能ならばコピーし、*new_node* を返します。

ast.**iter_fields**(*node*)

node にある node._fields のそれぞれのフィールドを (**フィールド名，値**) のタプルとして yield します。

ast.**iter_child_nodes**(*node*)

node の直接の子ノード全てを yield します。すなわち、yield されるのは、ノードであるような全てのフィールドおよびノードのリストであるようなフィールドの全てのアイテムです。

ast.**walk**(*node*)

node の全ての子孫ノード (*node* 自体を含む) を再帰的に yield します。順番は決められていません。この関数はノードをその場で変更するだけで文脈を気にしないような場合に便利です。

class ast.**NodeVisitor**

抽象構文木を渡り歩いてビジター関数を見つけたノードごとに呼び出すノード・ビジターの基底クラスです。この関数は *visit()* メソッドに送られる値を返してもかまいません。

このクラスはビジター・メソッドを付け加えたサブクラスを派生させることを意図しています。

visit(*node*)

ノードを訪れます。デフォルトの実装では `self.visit_classname` というメソッド (ここで *classname* はノードのクラス名です) を呼び出すか、そのメソッドがなければ *generic_visit()* を呼び出します。

generic_visit(*node*)
このビジターはノードの全ての子について *visit()* を呼び出します。

注意して欲しいのは、専用のビジター・メソッドを具えたノードの子ノードは、このビジターが *generic_visit()* を呼び出すかそれ自身で子ノードを訪れない限り訪れられないということです。

トラバースの途中でノードを変化させたいならば *NodeVisitor* を使ってはいけません。そうした目的のために変更を許す特別なビジター (*NodeTransformer*) があります。

バージョン 3.8 で非推奨: `visit_Num()`, `visit_Str()`, `visit_Bytes()`, `visit_NameConstant()` および `visit_Ellipsis()` の各メソッドは非推奨です。また将来の Python バージョンでは呼び出されなくなります。全ての定数ノードを扱うには `visit_Constant()` を追加してください。

class ast.NodeTransformer
NodeVisitor のサブクラスで抽象構文木を渡り歩きながらノードを変更することを許すものです。

NodeTransformer は抽象構文木 (AST) を渡り歩き、ビジター・メソッドの戻り値を使って古いノードを置き換えたり削除したりします。ビジター・メソッドの戻り値が `None` ならば、ノードはその場から取り去られ、そうでなければ戻り値で置き換えられます。置き換えない場合は戻り値が元のノードそのものであってもかまいません。

それでは例を示しましょう。Name (たとえば `foo`) を見つけるたび全て `data['foo']` に書き換える変換器 (transformer) です:

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Index(value=Constant(value=node.id)),
            ctx=node.ctx
        )
```

操作しようとしているノードが子ノードを持つならば、その子ノードの変形も自分で行うか、またはそのノードに対し最初に `generic_visit()` メソッドを呼び出すか、それを行うのはあなたの責任だということを肝に銘じましょう。

文のコレクションであるようなノード (全ての文のノードが当てはまります) に対して、このビジターは単独のノードではなくノードのリストを返すかもしれません。

NodeTransformer が (たとえば、`lineno` のような) 位置情報を与えずに (元の木の一部ではなく) 新しいノードを導入する場合、*fix_missing_locations()* を新しいサブツリーで呼び出して、位置情報を再計算する必要があります。

```
tree = ast.parse('foo', mode='eval')
new_tree = fix_missing_locations(RewriteName().visit(tree))
```

たいてい、変換器の使い方は次のようになります:

```
node = YourTransformer().visit(node)
```

ast.**dump**(*node*, *annotate_fields=True*, *include_attributes=False*)

node 内のツリーのフォーマットされたダンプを返します。主な使い道はデバッグです。*annotate_fields* が (デフォルトで)true の場合、返される文字列はフィールドの名前と値を示します。*annotate_fields* が false の場合、あいまいさのないフィールド名を省略することにより、結果文字列はよりコンパクトになります。行番号や列オフセットのような属性はデフォルトではダンプされません。これがほ欲しければ、*include_attributes* を true にセットすることができます。

参考:

外部ドキュメント Green Tree Snakes には Python AST についての詳細が書かれています。

ASTTokens は Python AST を、生成元のソースコードのトークン位置やテキストで注解します。これはソースコード変換を行うツールで有用です。

leoAst.py unifies the token-based and parse-tree-based views of python programs by inserting two-way links between tokens and ast nodes.

LibCST はコードを ast ツリーに似た構文木 (Concrete Syntax Tree) にパースし、かつ全ての書式設定の詳細を保持します。これは自動リファクタリングアプリケーション (codemod) やリンタを作成する際に有用です。

Parso はエラーリカバリや異なる Python バージョン (複数の Python バージョン) での復元可能なパース (round-trip parsing) をサポートします。また、Parso は Python ファイル内の複数の文法エラーをリストすることもできます。

32.3 symtable --- コンパイラの記号表へのアクセス

ソースコード: Lib/symtable.py

記号表 (symbol table) は、コンパイラが AST からバイトコードを生成する直前に作られます。記号表はコード中の全ての識別子のスコープの算出に責任を持ちます。*symtable* はこうした記号表を調べるインターフェイスを提供します。

32.3.1 記号表の生成

symtable.**symtable**(*code*, *filename*, *compile_type*)

Python ソース *code* に対するトップレベルの *SymbolTable* を返します。*filename* はコードを収めてあるファイルの名前です。*compile_type* は *compile()* の *mode* 引数のようなものです。

32.3.2 記号表の検査

class symtable.**SymbolTable**

ブロックに対する名前空間の記号表。コンストラクタはパブリックではありません。

get_type()

記号表の型を返します。有り得る値は `'class'`, `'module'`, `'function'` です。

get_id()

記号表の識別子を返します。

get_name()

記号表の名前を返します。この名前は記号表がクラスに対するものであればクラス名であり、関数に対するものであれば関数名であり、グローバルな (*get_type()* が `'module'` を返す) 記号表であれば `'top'` です。

get_lineno()

この記号表に対応するブロックの一行目の行番号を返します。

is_optimized()

この記号表に含まれるローカル変数が最適化できるならば `True` を返します。

is_nested()

ブロックが入れ子のクラスまたは関数のとき `True` を返します。

has_children()

ブロックが入れ子の名前空間を含んでいるならば `True` を返します。入れ子の名前空間は *get_children()* で得られます。

has_exec()

ブロックの中で `exec` が使われているならば `True` を返します。

get_identifiers()

この記号表にある記号の名前のリストを返します。

lookup(*name*)

記号表から名前 *name* を見つけ出して *Symbol* インスタンスとして返します。

get_symbols()

記号表中の名前を表す *Symbol* インスタンスのリストを返します。

get_children()

入れ子になった記号表のリストを返します。

class symtable.Function
関数またはメソッドの名前空間。このクラスは *SymbolTable* を継承しています。

get_parameters()
この関数の引数名からなるタプルを返します。

get_locals()
この関数のローカル変数の名前からなるタプルを返します。

get_globals()
この関数のグローバル変数の名前からなるタプルを返します。

get_nonlocals()
この関数の非ローカル変数の名前からなるタプルを返します。

get_frees()
この関数の自由変数の名前からなるタプルを返します。

class symtable.Class
クラスの名前空間。このクラスは *SymbolTable* を継承しています。

get_methods()
このクラスで宣言されているメソッド名からなるタプルを返します。

class symtable.Symbol
SymbolTable のエントリーでソースの識別子に対応するものです。コンストラクタはパブリックではありません。

get_name()
記号の名前を返します。

is_referenced()
記号がそのブロックの中で使われていれば True を返します。

is_imported()
記号が import 文で作られたものならば True を返します。

is_parameter()
記号がパラメータならば True を返します。

is_global()
記号がグローバルならば True を返します。

is_nonlocal()
記号が非ローカルならば True を返します。

is_declared_global()
記号が global 文によってグローバルであると宣言されているなら True を返します。

is_local()
記号がそのブロックに対してローカルならば True を返します。

is_annotated()
記号が注釈付きならば True を返します。

バージョン 3.6 で追加.

is_free()
記号がそのブロックの中で参照されていて、しかし代入は行われないならば True を返します。

is_assigned()
記号がそのブロックの中で代入されているならば True を返します。

is_namespace()
名前の束縛が新たな名前空間を導入するならば True を返します。

名前が関数または class 文のターゲットとして使われるならば、真です。

例えば:

```
>>> table = symtable.symtable("def some_func(): pass", "string", "exec")
>>> table.lookup("some_func").is_namespace()
True
```

一つの名前が複数のオブジェクトに束縛されうることに注意しましょう。結果が True であったとしても、その名前は他のオブジェクトにも束縛されるかもしれず、それがたとえば整数やリストであれば、そこでは新たな名前空間は導入されません。

get_namespaces()
この名前に束縛された名前空間のリストを返します。

get_namespace()
この名前に束縛された唯一の名前空間を返します。束縛された名前空間が複数ある場合 *ValueError* が送出されます。

32.4 symbol --- Python 解析木と共に使われる定数

ソースコード: Lib/symbol.py

このモジュールは解析木の内部ノードの数値を表す定数を提供します。ほとんどの Python 定数とは違い、これらは小文字の名前を使います。言語の文法のコンテキストにおける名前の定義については、Python ディストリビューションのファイル Grammar/Grammar を参照してください。名前がマップする特定の数値は Python のバージョン間で変わります。

このモジュールには、データオブジェクトも一つ付け加えられています:

symbol.sym_name
ディクショナリはこのモジュールで定義されている定数の数値を名前の文字列へマップし、より人が読みやすいように解析木を表現します。

32.5 token --- Python 解析木と共に使われる定数

ソースコード: Lib/token.py

このモジュールは解析木の葉ノード (終端記号) の数値を表す定数を提供します。言語の文法のコンテキストにおける名前の定義については、Python ディストリビューションのファイル Grammar/Grammar を参照してください。名前がマップする特定の数値は Python のバージョン間で変わります。

このモジュールは、数値コードから名前へのマッピングと、いくつかの関数も提供しています。関数は Python の C ヘッダファイルの定義を反映します。

token.**tok_name**
: ディクショナリはこのモジュールで定義されている定数の数値を名前の文字列へマップし、より人が読みやすいように解析木を表現します。

token.**ISTERMINAL**(*x*)
: 終端トークンの値に対して True を返します。

token.**ISNONTERMINAL**(*x*)
: 非終端トークンの値に対して True を返します。

token.**ISEOF**(*x*)
: *x* が入力の終わりを示すマーカーならば、True を返します。

token の定数一覧:

token.**ENDMARKER**

token.**NAME**

token.**NUMBER**

token.**STRING**

token.**NEWLINE**

token.**INDENT**

token.**DEDENT**

token.**LPAR**
: Token value for "(".

token.**RPAR**
: Token value for ")".

token.**LSQB**
: Token value for "[".

token.**RSQB**
: Token value for "]".

token.COLON
: Token value for ":".

token.COMMA
: Token value for ",".

token.SEMI
: Token value for ";".

token.PLUS
: Token value for "+".

token.MINUS
: Token value for "-".

token.STAR
: Token value for "*".

token.SLASH
: Token value for "/".

token.VBAR
: Token value for "|".

token.AMPER
: Token value for "&".

token.LESS
: Token value for "<".

token.GREATER
: Token value for ">".

token.EQUAL
: Token value for "=".

token.DOT
: Token value for ".".

token.PERCENT
: Token value for "%".

token.LBRACE
: Token value for "{".

token.RBRACE
: Token value for "}".

token.EQEQUAL
: Token value for "==".

token.NOTEQUAL
: Token value for "!=".

token.LESSEQUAL
: Token value for "<=".

token.GREATEREQUAL
: Token value for ">=".

token.TILDE
: Token value for "~".

token.CIRCUMFLEX
: Token value for "^".

token.LEFTSHIFT
: Token value for "<<".

token.RIGHTSHIFT
: Token value for ">>".

token.DOUBLESTAR
: Token value for "**".

token.PLUSEQUAL
: Token value for "+=".

token.MINEQUAL
: Token value for "-=".

token.STAREQUAL
: Token value for "*=".

token.SLASHEQUAL
: Token value for "/=".

token.PERCENTEQUAL
: Token value for "%=".

token.AMPEREQUAL
: Token value for "&=".

token.VBAREQUAL
: Token value for "|=".

token.CIRCUMFLEXEQUAL
: Token value for "^=".

token.LEFTSHIFTEQUAL
: Token value for "<<=".

token.RIGHTSHIFTEQUAL
: Token value for ">>=".

token.DOUBLESTAREQUAL
: Token value for "**=".

token.DOUBLESLASH
: Token value for "//".

token.DOUBLESLASHEQUAL
: Token value for "//=".

token.AT
: Token value for "@".

token.ATEQUAL
: Token value for "@=".

token.RARROW
: Token value for "->".

token.ELLIPSIS
: Token value for "...".

token.COLONEQUAL
: Token value for ":=".

token.OP

token.AWAIT

token.ASYNC

token.TYPE_IGNORE

token.TYPE_COMMENT

token.ERRORTOKEN

token.N_TOKENS

token.NT_OFFSET

以下のトークン値は、C のトークナイザでは利用されませんが、*tokenize* モジュールによって利用されます。

token.COMMENT
: コメントであることを表すために使われるトークン値です。

token.NL
: 終わりではない改行を表すために使われるトークン値。*NEWLINE* トークンは Python コードの論理行の終わりを表します。“NL“トークンはコードの論理行が複数の物理行にわたって続いているときに作られます。

token.ENCODING
: ソースの bytes をテキストにデコードするために使われるエンコーディングを示すトークン値。*tokenize.tokenize()* は常に最初に ENCODING トークンを返します。

token.TYPE_COMMENT
: Token value indicating that a type comment was recognized. Such tokens are only produced when *ast.parse()* is invoked with type_comments=True.

バージョン 3.5 で変更: *AWAIT* と *ASYNC* トークンが追加されました。

バージョン 3.7 で変更: *COMMENT* 、*NL*、*ENCODING* トークンが追加されました。

バージョン 3.7 で変更: *AWAIT* と *ASYNC* トークンが削除されました。”async” と ”await” は、*NAME* トークンとして扱われます。

バージョン 3.8 で変更: Added *TYPE_COMMENT*, *TYPE_IGNORE*, *COLONEQUAL*. Added *AWAIT* and *ASYNC* tokens back (they're needed to support parsing older Python versions for *ast.parse()* with feature_version set to 6 or lower).

32.6 keyword --- Python キーワードチェック

ソースコード: Lib/keyword.py

このモジュールでは、Python プログラムで文字列が :ref:‘キーワード <keywords>‘か否かをチェックする機能を提供します。

keyword.iskeyword(*s*)
: *s* が Python の キーワード であれば True を返します。

keyword.kwlist
: インタープリタで定義している全ての キーワード のシーケンス。特定の *__future__* 宣言がなければ有効ではないキーワードでもこのリストには含まれます。

32.7 tokenize --- Python ソースのためのトークナイザ

ソースコード: Lib/tokenize.py

---

tokenize モジュールでは、Python で実装された Python ソースコードの字句解析器を提供します。さらに、このモジュールの字句解析器はコメントもトークンとして返します。このため、このモジュールはスクリーン上で表示する際の色付け機能 (colorizers) を含む ” 清書出力器 (pretty-printer)” を実装する上で便利です。

トークン・ストリームの扱いをシンプルにするために、全ての operator と delimiter トークンおよび *Ellipsis* はジェネリックな *OP* トークンタイプとして返されます。正確な型は *tokenize.tokenize()* が返す *named tuple* の exact_type プロパティをチェックすれば解ります。

32.7.1 入力のトークナイズ

第一のエントリポイントは ジェネレータ です:

tokenize.tokenize(*readline*)

tokenize() ジェネレータは一つの引数 *readline* を必要とします。この引数は呼び出し可能オブジェクトで、ファイルオブジェクトの *io.IOBase.readline()* メソッドと同じインタフェースを提供している必要があります。この関数は呼び出しのたびに入力の一行を bytes で返さなければなりません。

このジェネレータは次の 5 要素のタプルを返します; トークンタイプ; トークン文字列; ソース内でそのトークンがどの行、列で開始するかを示す int の (srow, scol) タプル; どの行、列で終了するかを示す int の (erow, ecol) タプル; トークンが見つかった行。(タプルの最後の要素にある) 行は **物理** 行です。この 5 要素のタプルは *named tuple* として返され、フィールド名は type string start end line になります。

返される *named tuple* は追加のプロパティ exact_type を持ちます。このプロパティは *OP* トークンに対して正確な演算子のタイプを持ちます。それ以外のトークンタイプについては、exact_type は type フィールドと同じ値を持ちます。

バージョン 3.1 で変更: named tuple のサポートを追加。

バージョン 3.3 で変更: exact_type のサポートを追加。

tokenize() は **PEP 263** にしたがって、ソースのエンコーディングを UTF-8 BOM か encoding cookie を見つけて決定します。

tokenize.generate_tokens(*readline*)

Tokenize a source reading unicode strings instead of bytes.

Like *tokenize()*, the *readline* argument is a callable returning a single line of input. However, *generate_tokens()* expects *readline* to return a str object rather than bytes.

The result is an iterator yielding named tuples, exactly like *tokenize()*. It does not yield an *ENCODING* token.

token モジュールの全ての定数は *tokenize* モジュールからも公開されています。

もう一つの関数がトークン化プロセスを逆転するために提供されています。これは、スクリプトを字句解析し、トークンのストリームに変更を加え、変更されたスクリプトを書き戻すようなツールを作成する際に便利です。

tokenize.untokenize(*iterable*)

トークンの列を Python ソースコードに変換します。*iterable* は少なくとも二つの要素、トークンタイプおよびトークン文字列、からなるシーケンスを返さなければいけません。その他のシーケンスの要素は無視されます。

再構築されたスクリプトは一つの文字列として返されます。得られる結果はもう一度字句解析すると入力と一致することが保証されるので、変換がロスレスでありラウンドトリップできることは間違いありません。この保証はトークン型およびトークン文字列に対してのものでトークン間のスペース (コラム位置) のようなものは変わることがあり得ます。

It returns bytes, encoded using the *ENCODING* token, which is the first token sequence output by *tokenize()*. If there is no encoding token in the input, it returns a str instead.

tokenize() はトークナイズしようとしているソースファイルのエンコーディングを検出する必要があります。これを行うために使っている関数が公開されています:

tokenize.detect_encoding(*readline*)

detect_encoding() 関数は Python のソースファイルをデコードするのに使うエンコーディングを検出するために使われます。*tokenize()* ジェネレータと同じ readline を引数として取ります。

readline を最大 2 回呼び出し、利用するエンコーディング (文字列として) と、読み込んだ行を (bytes からデコードされないままの状態で) 返します。

UTF-8 BOM か **PEP 263** で定義されている encoding cookie からエンコーディングを検出します。BOM と cookie の両方が存在し、一致していない場合、*SyntaxError* が送出されます。BOM が見つかった場合はエンコーディングとして `'utf-8-sig'` が返されます。

エンコーディングが指定されていない場合、デフォルトの `'utf-8'` が返されます。

Python のソースファイルを開くには *open()* を使ってください。これは *detect_encoding()* を利用してファイルエンコーディングを検出します。

tokenize.open(*filename*)

detect_encoding() を使って検出したエンコーディングを利用して、ファイルを読み出し専用モードで開きます。

バージョン 3.2 で追加.

exception tokenize.TokenError

docstring や複数行にわたることが許される式がファイル内のどこかで終わっていない場合に送出されます。例えば:

```
"""Beginning of
docstring
```

もしくは:

```
[1,
 2,
 3
```

閉じていないシングルクォート文字列は送出されるべきエラーの原因とならないことに注意してください。それらは *ERRORTOKEN* とトークン化され、続いてその内容がトークン化されます。

32.7.2 コマンドラインからの使用

バージョン 3.3 で追加.

tokenize モジュールはコマンドラインからスクリプトとして実行することができます。次のようにシンプルに利用できます:

```
python -m tokenize [-e] [filename.py]
```

以下のオプションが使用できます:

-h, --help
　このヘルプメッセージを出力して終了します

-e, --exact
　exact type を使ってトークン名を表示します

filename.py が指定された場合、その内容がトークナイズされ stdout に出力されます。指定されなかった場合は stdin からトークナイズします。

32.7.3 使用例

スクリプト書き換えの例で、浮動小数点数リテラルを Decimal オブジェクトに変換します:

```
from tokenize import tokenize, untokenize, NUMBER, STRING, NAME, OP
from io import BytesIO

def decistmt(s):
    """Substitute Decimals for floats in a string of statements.

    >>> from decimal import Decimal
    >>> s = 'print(+21.3e-5*-.1234/81.7)'
    >>> decistmt(s)
    "print (+Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7'))"

    The format of the exponent is inherited from the platform C library.
    Known cases are "e-007" (Windows) and "e-07" (not Windows).  Since
    we're only showing 12 digits, and the 13th isn't close to 5, the
    rest of the output should be platform-independent.

    >>> exec(s)  #doctest: +ELLIPSIS
    -3.21716034272e-0...7

    Output from calculations with Decimal should be identical across all
    platforms.

    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7
    """
    result = []
```

(次のページに続く)

(前のページからの続き)

```
    g = tokenize(BytesIO(s.encode('utf-8')).readline)  # tokenize the string
    for toknum, tokval, _, _, _ in g:
        if toknum == NUMBER and '.' in tokval:  # replace NUMBER tokens
            result.extend([
                (NAME, 'Decimal'),
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')
            ])
        else:
            result.append((toknum, tokval))
    return untokenize(result).decode('utf-8')
```

コマンドラインからトークナイズする例。次のスクリプトが:

```
def say_hello():
    print("Hello, World!")

say_hello()
```

トークナイズされて次のような出力になります。最初のカラムはトークンが現れた行／カラム、次のカラムはトークンの名前、最後のカラムは (あれば) トークンの値です

```
$ python -m tokenize hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          OP             '('
1,14-1,15:          OP             ')'
1,15-1,16:          OP             ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           OP             '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          OP             ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           OP             '('
4,10-4,11:          OP             ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

トークンの exact_type 名は *-e* オプションで表示できます:

```
$ python -m tokenize -e hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
```

(次のページに続く)

(前のページからの続き)

```
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          LPAR           '('
1,14-1,15:          RPAR           ')'
1,15-1,16:          COLON          ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           LPAR           '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          RPAR           ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           LPAR           '('
4,10-4,11:          RPAR           ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

Example of tokenizing a file programmatically, reading unicode strings instead of bytes with *generate_tokens()*:

```
import tokenize

with tokenize.open('hello.py') as f:
    tokens = tokenize.generate_tokens(f.readline)
    for token in tokens:
        print(token)
```

Or reading bytes directly with *tokenize()*:

```
import tokenize

with open('hello.py', 'rb') as f:
    tokens = tokenize.tokenize(f.readline)
    for token in tokens:
        print(token)
```

32.8 tabnanny --- あいまいなインデントの検出

ソースコード: Lib/tabnanny.py

差し当たり、このモジュールはスクリプトとして呼び出すことを意図しています。しかし、IDE 上にインポートして下で説明する関数 *check()* を使うことができます。

注釈: このモジュールが提供する API を将来のリリースで変更する確率が高いです。このような変更は後方

互換性がないかもしれません。

tabnanny.check(*file_or_dir*)
: *file_or_dir* がディレクトリであってシンボリックリンクでないときに、*file_or_dir* という名前のディレクトリツリーを再帰的に下って行き、この通り道に沿ってすべての .py ファイルをチェックします。*file_or_dir* が通常の Python ソースファイルの場合には、問題のある空白をチェックします。診断メッセージは *print()* 関数を使って標準出力に書き込まれます。

tabnanny.verbose
: 冗長なメッセージをプリントするかどうかを示すフラグ。スクリプトとして呼び出された場合は、-v オプションによって増加します。

tabnanny.filename_only
: 問題のある空白を含むファイルのファイル名のみをプリントするかどうかを示すフラグ。スクリプトとして呼び出された場合は、-q オプションによって真に設定されます。

exception tabnanny.NannyNag
: あいまいなインデントを検出した場合に *process_tokens()* が送出します。この例外は *check()* で捕捉され処理されます。

tabnanny.process_tokens(*tokens*)
: この関数は、*tokenize* モジュールによって生成されたトークンを *check()* が処理するために使われます。

参考:

tokenize **モジュール**　Python ソースコードの字句解析器。

32.9 pyclbr --- Python モジュールブラウザサポート

ソースコード: Lib/pyclbr.py

The *pyclbr* module provides limited information about the functions, classes, and methods defined in a Python-coded module. The information is sufficient to implement a module browser. The information is extracted from the Python source code rather than by importing the module, so this module is safe to use with untrusted code. This restriction makes it impossible to use this module with modules not implemented in Python, including all standard and optional extension modules.

pyclbr.readmodule(*module*, *path=None*)
: Return a dictionary mapping module-level class names to class descriptors. If possible, descriptors for imported base classes are included. Parameter *module* is a string with the name of the module to read; it may be the name of a module within a package. If given, *path* is a sequence of directory paths prepended to sys.path, which is used to locate the module source code.

  This function is the original interface and is only kept for back compatibility. It returns a filtered version of the following.

pyclbr.**readmodule_ex**(*module*, *path=None*)

Return a dictionary-based tree containing a function or class descriptors for each function and class defined in the module with a `def` or `class` statement. The returned dictionary maps module-level function and class names to their descriptors. Nested objects are entered into the children dictionary of their parent. As with readmodule, *module* names the module to be read and *path* is prepended to sys.path. If the module being read is a package, the returned dictionary has a key `'__path__'` whose value is a list containing the package search path.

バージョン 3.7 で追加: Descriptors for nested definitions. They are accessed through the new children attribute. Each has a new parent attribute.

The descriptors returned by these functions are instances of Function and Class classes. Users are not expected to create instances of these classes.

32.9.1 Function オブジェクト

Class `Function` instances describe functions defined by def statements. They have the following attributes:

Function.file

Name of the file in which the function is defined.

Function.module

The name of the module defining the function described.

Function.name

関数の名前です。

Function.lineno

The line number in the file where the definition starts.

Function.parent

For top-level functions, None. For nested functions, the parent.

バージョン 3.7 で追加.

Function.children

A dictionary mapping names to descriptors for nested functions and classes.

バージョン 3.7 で追加.

32.9.2 クラスオブジェクト

Class `Class` instances describe classes defined by class statements. They have the same attributes as Functions and two more.

`Class.file`
Name of the file in which the class is defined.

`Class.module`
The name of the module defining the class described.

`Class.name`
クラスの名前です。

`Class.lineno`
The line number in the file where the definition starts.

`Class.parent`
For top-level classes, None. For nested classes, the parent.

バージョン 3.7 で追加.

`Class.children`
A dictionary mapping names to descriptors for nested functions and classes.

バージョン 3.7 で追加.

`Class.super`
記述しようとしている対象クラスの、直接の基底クラス群について記述している `Class` オブジェクトのリストです。スーパクラスとして挙げられているが *`readmodule_ex()`* が見つけられなかったクラスは、`Class` オブジェクトではなくクラス名の文字列としてリストに挙げられます。

`Class.methods`
A dictionary mapping method names to line numbers. This can be derived from the newer children dictionary, but remains for back-compatibility.

32.10 `py_compile` --- Python ソースファイルのコンパイル

ソースコード: Lib/py_compile.py

`py_compile` モジュールには、ソースファイルからバイトコードファイルを作る関数と、モジュールのソースファイルがスクリプトとして呼び出される時に使用される関数が定義されています。

頻繁に必要となるわけではありませんが、共有ライブラリとしてモジュールをインストールする場合や、特にソースコードのあるディレクトリにバイトコードのキャッシュファイルを書き込む権限がないユーザがいるときには、この関数は役に立ちます。

exception py_compile.PyCompileError

ファイルをコンパイル中にエラーが発生すると送出される例外。

py_compile.compile(*file*, *cfile=None*, *dfile=None*, *doraise=False*, *optimize=-1*, *invalidation_mode=PycInvalidationMode.TIMESTAMP*, *quiet=0*)

Compile a source file to byte-code and write out the byte-code cache file. The source code is loaded from the file named *file*. The byte-code is written to *cfile*, which defaults to the **PEP 3147**/**PEP 488** path, ending in `.pyc`. For example, if *file* is `/foo/bar/baz.py` *cfile* will default to `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. If *dfile* is specified, it is used as the name of the source file in error messages instead of *file*. If *doraise* is true, a *PyCompileError* is raised when an error is encountered while compiling *file*. If *doraise* is false (the default), an error string is written to `sys.stderr`, but no exception is raised. This function returns the path to byte-compiled file, i.e. whatever *cfile* value was used.

The *doraise* and *quiet* arguments determine how errors are handled while compiling file. If *quiet* is 0 or 1, and *doraise* is false, the default behaviour is enabled: an error string is written to `sys.stderr`, and the function returns `None` instead of a path. If *doraise* is true, a *PyCompileError* is raised instead. However if *quiet* is 2, no message is written, and *doraise* has no effect.

(明示的に指定されたか計算された結果の) *cfile* のパスがシンボリックリンクだったり通常のファイルでなかったりした場合は、*FileExistsError* が送出されます。この動作は、それらのパスにバイトコンパイルされたファイルを書き込む権限がある場合、インポートのときにそのパスを通常のファイルに変えてしまうことを警告するためのものです。これはリネームを使ってバイトコンパイルされたファイルを配置するインポートで、同時にファイル書き込みをしてしまう問題を避けるための副作用です。

optimize は最適化レベルを制御するもので、組み込みの *compile()* 関数に渡されます。デフォルトは現在のインタープリタの最適化レベルを選ぶ `-1` です。

invalidation_mode should be a member of the *PycInvalidationMode* enum and controls how the generated bytecode cache is invalidated at runtime. The default is *PycInvalidationMode.CHECKED_HASH* if the `SOURCE_DATE_EPOCH` environment variable is set, otherwise the default is *PycInvalidationMode.TIMESTAMP*.

バージョン 3.2 で変更: **PEP 3147** に準拠し *cfile* のデフォルト値を変更しました。以前のデフォルトは *file* + `'c'` (最適化が有効な場合は `'o'`) でした。同時に *optimize* パラメータも追加されました。

バージョン 3.4 で変更: バイトコードをキャッシュするファイルへの書き込みに *importlib* を使うようにコードを変更しました。これにより、例えば権限や write-and-move セマンティクスなどの、ファイルの作成や書き込みの動作が *importlib* と一致するようになりました。*cfile* がシンボリックリンクであるか通常のファイルでない場合、*FileExistsError* を送出するという注意書きも追加されました。

バージョン 3.7 で変更: The *invalidation_mode* parameter was added as specified in **PEP 552**. If the `SOURCE_DATE_EPOCH` environment variable is set, *invalidation_mode* will be forced to *PycInvalidationMode.CHECKED_HASH*.

バージョン 3.7.2 で変更: The `SOURCE_DATE_EPOCH` environment variable no longer overrides the value of the *invalidation_mode* argument, and determines its default value instead.

バージョン 3.8 で変更: *quiet* パラメータが追加されました。

class py_compile.PycInvalidationMode

A enumeration of possible methods the interpreter can use to determine whether a bytecode file is up to date with a source file. The `.pyc` file indicates the desired invalidation mode in its header. See pyc-invalidation for more information on how Python invalidates `.pyc` files at runtime.

バージョン 3.7 で追加.

TIMESTAMP

The `.pyc` file includes the timestamp and size of the source file, which Python will compare against the metadata of the source file at runtime to determine if the `.pyc` file needs to be regenerated.

CHECKED_HASH

The `.pyc` file includes a hash of the source file content, which Python will compare against the source at runtime to determine if the `.pyc` file needs to be regenerated.

UNCHECKED_HASH

Like *CHECKED_HASH*, the `.pyc` file includes a hash of the source file content. However, Python will at runtime assume the `.pyc` file is up to date and not validate the `.pyc` against the source file at all.

This option is useful when the `.pycs` are kept up to date by some system external to Python like a build system.

py_compile.**main**(*args=None*)

いくつか複数のソースファイルをコンパイルします。*args* で (あるいは *args* が `None` であればコマンドラインで) 指定されたファイルをコンパイルし、できたバイトコードを通常の方法で保存します。この関数はソースファイルの存在するディレクトリを検索しません; 指定されたファイルをコンパイルするだけです。args が `'-'` 1 つだけだった場合、ファイルのリストは標準入力から取られます。

バージョン 3.2 で変更: `'-'` のサポートが追加されました。

このモジュールがスクリプトとして実行されると、*main()* がコマンドラインで指定されたファイルを全てコンパイルします。一つでもコンパイルできないファイルがあると終了ステータスが 0 でない値になります。

参考:

compileall **モジュール** ディレクトリツリー内の Python ソースファイルを全てコンパイルするライブラリ。

32.11 compileall --- Python ライブラリをバイトコンパイル

ソースコード: Lib/compileall.py

このモジュールは、Python ライブラリのインストールを助けるユーティリティ関数群を提供します。この関数群は、ディレクトリツリー内の Python ソースファイルをコンパイルします。このモジュールを使って、キャッシュされたバイトコードファイルをライブラリのインストール時に生成することで、ライブラリディレクトリに書き込み権限をもたないユーザでも、これらを利用できるようになります。

32.11.1 コマンドラインでの使用

このモジュールは、(**python -m compileall** を使って) Python ソースをコンパイルするスクリプトとして機能します。

directory ...
file ...
位置引数は、コンパイルするファイル群か、再帰的に横断されるディレクトリでソースファイル群を含むものです。引数が与えられなければ、`-l <directories from sys.path>` を渡したのと同じように動作します。

-l
サブディレクトリを再帰処理せず、指名または暗示されたディレクトリ群に含まれるソースコードファイル群だけをコンパイルします。

-f
タイムスタンプが最新であってもリビルドを強制します。

-q
コンパイルされたファイルのリストを出力しません。一つ渡された場合でもエラーメッセージは出力されます。二つ (`-qq`) の場合全ての出力は抑制されます。

-d destdir
コンパイルされるそれぞれのファイルへのパスの先頭に、ディレクトリを追加します。これはコンパイル時トレースバックに使われ、バイトコードファイルが実行される時点でソースファイルが存在しない場合に、トレースバックやその他のメッセージに使われるバイトコードファイルにもコンパイルされます。

-x regex
regex を使って、コンパイル候補のそれぞれのファイルのフルパスを検索し、regex がマッチしたファイルを除外します。

-i list
ファイル `list` を読み込み、そのファイルのそれぞれの行を、コンパイルするファイルとディレクトリのリストに加えます。`list` が `-` なら、`stdin` の行を読み込みます。

-b
バイトコードファイルを、他のバージョンの Python によって生成されたバイトコードファイルを上書

きするかもしれない、レガシーな場所に生成します。デフォルトでは **PEP 3147** で決められた場所と名前を使い、複数のバージョンの Python が共存できるようにします。

`-r`
: サブディレクトリの最大再起深度を制御します。このオプションが与えられた場合、`-l` オプションは無視されます。`python -m compileall <directory> -r 0` は `python -m compileall <directory> -l` と等価です。

`-j N`
: 与えられたディレクトリ内のファイルを *N* ワーカでコンパイルします。`0` の場合 *os.cpu_count()* の結果が使用されます。

`--invalidation-mode [timestamp|checked-hash|unchecked-hash]`
: 生成したバイトコードファイルを実行時に無効化する方法を制御します。`timestamp` を指定すると、生成した ".pyc" ファイルにソースファイルのタイムスタンプとサイズを埋め込みます。`checked-hash` と `unchecked-hash` を指定すると、ハッシュベースの pyc ファイルが生成されます。ハッシュベースの pyc ファイルは、ソースファイルにタイムスタンプではなくハッシュ値を埋め込みます。Python がバイトコードキャッシュファイルを実行時に検証する方法の詳細を知るには、pyc-invalidation を参照してください。デフォルト値は、`SOURCE_DATE_EPOCH` 環境変数が設定されていなければ `timestamp` で、`SOURCE_DATE_EPOCH` 環境変数が設定されていれば、`checked-hash` になります。

バージョン 3.2 で変更: `-i`, `-b`, `-h` オプションを追加。

バージョン 3.5 で変更: `-j`, `-r`, `-qq` オプションが追加されました。`-q` オプションが複数のレベルの値に変更されました。`-b` は常に拡張子 `.pyc` のバイトエンコーディングファイルを生成し、`.pyo` を作りません。

バージョン 3.7 で変更: `--invalidation-mode` オプションが追加されました。

compile() 関数で利用される最適化レベルを制御するコマンドラインオプションはありません。Python インタプリタ自体のオプションを使ってください: `python -O -m compileall`.

Similarly, the *compile()* function respects the *sys.pycache_prefix* setting. The generated bytecode cache will only be useful if *compile()* is run with the same *sys.pycache_prefix* (if any) that will be used at runtime.

32.11.2 パブリックな関数

`compileall.compile_dir(`*dir*, *maxlevels=10*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *workers=1*, *invalidation_mode=None*`)`
: *dir* という名前のディレクトリーツリーをたどり、途中で見つけた全ての `.py` をコンパイルします。全ファイルのコンパイルが成功した場合は真を、それ以外の場合は偽を返します。

maxlevels 引数で最大再帰深度を制限します。デフォルトは `10` です。

ddir が与えられた場合、コンパイルされるそれぞれのファイルへのパスの先頭に、そのディレクトリを追加します。これはコンパイル時トレースバックに使われ、バイトコードファイルが実行される時点でソースファイルが存在しない場合に、トレースバックやその他のメッセージに使われるバイトコードファイルにもコンパイルされます。

force が真の場合、タイムスタンプが最新のであってもモジュールは再コンパイルされます。

rx が与えられた場合、コンパイル候補のそれぞれのファイルのフルパスに対して検索メソッドが呼び出され、それが真値を返したら、そのファイルは除外されます。

quiet が `False` か `0` (デフォルト) の場合、ファイル名とその他の情報は標準出力に表示されます。`1` の場合エラーのみが表示されます。`2` の場合出力はすべて抑制されます。

legacy が真の時、バイトコードファイルは古い場所と名前に書かれ、他のバージョンの Python によって作られたバイトコードファイルを上書きする可能性があります。デフォルトは **PEP 3147** で決められた場所と名前を使い、複数のバージョンの Python のバイトコードファイルが共存できるようにします。

optimize でコンパイラの最適化レベルを指定します。これは組み込みの *compile()* 関数に渡されます。

The argument *workers* specifies how many workers are used to compile files in parallel. The default is to not use multiple workers. If the platform can't use multiple workers and *workers* argument is given, then sequential compilation will be used as a fallback. If *workers* is 0, the number of cores in the system is used. If *workers* is lower than `0`, a *ValueError* will be raised.

invalidation_mode は、*py_compile.PycInvalidationMode* のメンバーでなければならず、生成された pyc ファイルを実行時に無効化する方法を制御します。

バージョン 3.2 で変更: *legacy* と *optimize* 引数が追加されました。

バージョン 3.5 で変更: `workers` パラメータが追加されました。

バージョン 3.5 で変更: *quiet* 引数が複数のレベルの値に変更されました。

バージョン 3.5 で変更: *optimize* の値に関わらず、*legacy* 引数は `.pyc` ファイルのみを書き出し、`.pyo` ファイルを書き出さないようになりました。

バージョン 3.6 で変更: *path-like object* を受け取るようになりました。

バージョン 3.7 で変更: *invalidation_mode* 引数を追加しました。

バージョン 3.7.2 で変更: *invalidation_mode* 引数のデフォルト値が None に変更されました。

バージョン 3.8 で変更: Setting *workers* to 0 now chooses the optimal number of cores.

`compileall.compile_file`(*fullname*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=None*)

パス *fullname* のファイルをコンパイルします。コンパイルが成功すれば真を、そうでなければ偽を返します。

ddir が与えられた場合、コンパイルされるファイルのパスの先頭にそのディレクトリを追加します。これはコンパイル時トレースバックに使われ、バイトコードファイルが実行される時点でソースファイルが存在しない場合に、トレースバックやその他のメッセージに使われるバイトコードファイルにもコンパイルされます。

rx が与えられた場合、コンパイル候補のファイルのフルパスに対して検索メソッドが呼び出され、それが真値を返したら、ファイルはコンパイルされず、True が返されます。

quiet が False か 0 (デフォルト) の場合、ファイル名とその他の情報は標準出力に表示されます。1 の場合エラーのみが表示されます。2 の場合出力はすべて抑制されます。

legacy が真の時、バイトコードファイルは古い場所と名前に書かれ、他のバージョンの Python によって作られたバイトコードファイルを上書きする可能性があります。デフォルトは **PEP 3147** で決められた場所と名前を使い、複数のバージョンの Python のバイトコードファイルが共存できるようにします。

optimize でコンパイラの最適化レベルを指定します。これは組み込みの *compile()* 関数に渡されます。

invalidation_mode は、*py_compile.PycInvalidationMode* のメンバーでなければならず、生成された pyc ファイルを実行時に無効化する方法を制御します。

バージョン 3.2 で追加.

バージョン 3.5 で変更: *quiet* 引数が複数のレベルの値に変更されました。

バージョン 3.5 で変更: *optimize* の値に関わらず、*legacy* 引数は .pyc ファイルのみを書き出し、.pyo ファイルを書き出さないようになりました。

バージョン 3.7 で変更: *invalidation_mode* 引数を追加しました。

バージョン 3.7.2 で変更: *invalidation_mode* 引数のデフォルト値が None に変更されました。

compileall.**compile_path**(*skip_curdir=True*, *maxlevels=0*, *force=False*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=None*)

sys.path からたどって見つけたすべての .py ファイルをバイトコンパイルします。すべてのファイルを問題なくコンパイルできたときに真を、それ以外のときに偽を返します。

skip_curdir が真 (デフォルト) のとき、カレントディレクトリは走査から除外されます。それ以外のすべての引数は *compile_dir()* 関数に渡されます。その他の compile 関数群と異なり、maxlevels のデフォルトが 0 になっていることに注意してください。

バージョン 3.2 で変更: *legacy* と *optimize* 引数が追加されました。

バージョン 3.5 で変更: *quiet* 引数が複数のレベルの値に変更されました。

バージョン 3.5 で変更: *optimize* の値に関わらず、*legacy* 引数は .pyc ファイルのみを書き出し、.pyo ファイルを書き出さないようになりました。

バージョン 3.7 で変更: *invalidation_mode* 引数を追加しました。

バージョン 3.7.2 で変更: *invalidation_mode* 引数のデフォルト値が None に変更されました。

Lib/ ディレクトリ以下にある全ての .py ファイルを強制的に再コンパイルするには、以下のようにします:

```
import compileall

```

(次のページに続く)

(前のページからの続き)

```
compileall.compile_dir('Lib/', force=True)

# Perform same compilation, excluding files in .svn directories.
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\\][.]svn'), force=True)

# pathlib.Path objects can also be used.
import pathlib
compileall.compile_dir(pathlib.Path('Lib/'), force=True)
```

参考:

Module *py_compile* 一つのソースファイルをバイトコンパイルします。

32.12 dis --- Python バイトコードの逆アセンブラ

ソースコード: Lib/dis.py

---

dis モジュールは CPython バイトコード bytecode を逆アセンブルすることでバイトコードの解析をサポートします。このモジュールが入力として受け取る CPython バイトコードはファイル `Include/opcode.h` に定義されており、コンパイラとインタプリタが使用しています。

CPython implementation detail: バイトコードは CPython インタプリタの実装詳細です。Python のバージョン間でバイトコードの追加や、削除、変更がないという保証はありません。このモジュールを使用することによって Python の異なる VM または異なるリリースの間で動作すると考えるべきではありません。

バージョン 3.6 で変更: 従来は使用されるバイト数は命令ごとに異なりましたが、このモジュールでは各々一つの命令につき 2 バイト使用することとなっています。

例: 以下の関数 `myfunc()` を考えると

```
def myfunc(alist):
    return len(alist)
```

`myfunc()` の逆アセンブル結果を表示するために次のコマンドを使うことができます:

```
>>> dis.dis(myfunc)
  2           0 LOAD_GLOBAL              0 (len)
              2 LOAD_FAST                0 (alist)
              4 CALL_FUNCTION            1
              6 RETURN_VALUE
```

(”2” は行番号です)。

32.12.1 バイトコード解析

バージョン 3.4 で追加.

バイトコード解析の API を使うと、Python のコード片を *Bytecode* オブジェクトでラップでき、コンパイルされたコードの細かいところに簡単にアクセスできます。

class dis.Bytecode(*x*, ***, *first_line=None*, *current_offset=None*)

関数、ジェネレータ、非同期ジェネレータ、コルーチン、メソッド、ソースコード文字列、(*compile()* が返すような) コードオブジェクトに対応するバイトコードを解析します。

これは、下で並べられている関数の多くのものをまとめた便利なラッパーです。とりわけ目立つのは *get_instructions()* で、*Bytecode* インスタンスに対し反復処理をしながら、バイトコード命令を *Instruction* インスタンスとして返します。

first_line が None でない場合は、それを逆アセンブルしたコードのソースの最初の行に表示する行番号とします。そうでない場合は、ソースの行の情報 (もしあれば) を逆アセンブルされたコードオブジェクトから直接取得します。

current_offset が None でない場合は、逆アセンブルされたコードでのあるインストラクションのオフセット位置を示します。これを設定すると、*dis()* の出力において、指定された命令コード (opcode) に " 現在の命令 (instruction)" を表す印が表示されます。

classmethod from_traceback(*tb*)

与えられたトレースバックから *Bytecode* インスタンスを構築し、*current_offset* がその例外の原因となった命令となるよう設定します。

codeobj

コンパイルされたコードオブジェクト。

first_line

コードオブジェクトのソースの最初の行 (利用可能であれば)

dis()

バイトコード命令の整形された表示を返します (*dis.dis()* と同じ出力になりますが、複数行文字列として返されます)。

info()

code_info() のようなコードオブジェクトの詳細を含んだ整形された複数行文字列を返します。

バージョン 3.7 で変更: This can now handle coroutine and asynchronous generator objects.

以下はプログラム例です:

```
>>> bytecode = dis.Bytecode(myfunc)
>>> for instr in bytecode:
...     print(instr.opname)
...
LOAD_GLOBAL
LOAD_FAST
```

(次のページに続く)

(前のページからの続き)

```
CALL_FUNCTION
RETURN_VALUE
```

32.12.2 解析関数

dis モジュールには、以下に挙げる入力を直接欲しい出力に変換する解析関数も定義してあります。1 つの命令だけが実行されている場合は、解析オブジェクトをいったん作るよりはこちらの方が便利です:

dis.code_info(*x*)

渡された関数、ジェネレータ、非同期ジェネレータ、コルーチン、メソッド、ソースコード文字列、コードオブジェクトに対する、詳細なコードオブジェクトの情報を、整形された複数行の文字列として返します。

この結果は実装に強く依存しており、Python VM や Python のバージョンによって異なることがあります。

バージョン 3.2 で追加.

バージョン 3.7 で変更: This can now handle coroutine and asynchronous generator objects.

dis.show_code(*x*, ***, *file=None*)

渡された関数、メソッド、ソースコード文字列、コードオブジェクトに対する、詳細なコードオブジェクトの情報を、*file* (または *file* が指定されていなければ `sys.stdout`) に表示します。

これは、インタラクティブシェル上で使うことを想定した、`print(code_info(x), file=file)` の便利なショートカットです。

バージョン 3.2 で追加.

バージョン 3.4 で変更: *file* 引数が追加されました。

dis.dis(*x=None*, ***, *file=None*, *depth=None*)

Disassemble the *x* object. *x* can denote either a module, a class, a method, a function, a generator, an asynchronous generator, a coroutine, a code object, a string of source code or a byte sequence of raw bytecode. For a module, it disassembles all functions. For a class, it disassembles all methods (including class and static methods). For a code object or sequence of raw bytecode, it prints one line per bytecode instruction. It also recursively disassembles nested code objects (the code of comprehensions, generator expressions and nested functions, and the code used for building nested classes). Strings are first compiled to code objects with the *compile()* built-in function before being disassembled. If no object is provided, this function disassembles the last traceback.

file 引数が渡された場合は、アセンブリをそこに書き込みます。そうでない場合は `sys.stdout` に出力します。

The maximal depth of recursion is limited by *depth* unless it is `None`. `depth=0` means no recursion.

バージョン 3.4 で変更: *file* 引数が追加されました。

バージョン 3.7 で変更: Implemented recursive disassembling and added *depth* parameter.

バージョン 3.7 で変更: This can now handle coroutine and asynchronous generator objects.

dis.distb(*tb=None*, ***, *file=None*)

トレースバックのスタックの先頭の関数を逆アセンブルします。None が渡された場合は最後のトレースバックを使います。例外を引き起こした命令が表示されます。

file 引数が渡された場合は、アセンブリをそこに書き込みます。そうでない場合は `sys.stdout` に出力します。

バージョン 3.4 で変更: *file* 引数が追加されました。

dis.disassemble(*code*, *lasti=-1*, ***, *file=None*)

dis.disco(*code*, *lasti=-1*, ***, *file=None*)

コードオブジェクトを逆アセンブルします。*lasti* が与えられた場合は、最後の命令を示します。出力は次のようなカラムに分割されます:

1. 各行の最初の命令に対する行番号。
2. 現在の命令。`-->` として示されます。
3. ラベル付けされた命令。`>>` とともに表示されます。
4. 命令のアドレス。
5. 命令コード名。
6. 命令パラメタ。
7. パラメタの解釈を括弧で囲んだもの。

パラメタの解釈は、ローカル変数とグローバル変数の名前、定数の値、分岐先、比較命令を認識します。

file 引数が渡された場合は、アセンブリをそこに書き込みます。そうでない場合は `sys.stdout` に出力します。

バージョン 3.4 で変更: *file* 引数が追加されました。

dis.get_instructions(*x*, ***, *first_line=None*)

渡された関数、メソッド、ソースコード文字列、コードオブジェクトにある命令のイテレータを返します。

イテレータは、与えられたコードの各命令の詳細情報を保持する名前付きタプル *Instruction* からなる列を生成します。

first_line が `None` でない場合は、それを逆アセンブルしたコードのソースの最初の行に表示する行番号とします。そうでない場合は、ソースの行の情報 (もしあれば) を逆アセンブルされたコードオブジェクトから直接取得します。

バージョン 3.4 で追加.

dis.findlinestarts(*code*)

This generator function uses the co_firstlineno and co_lnotab attributes of the code object *code* to find the offsets which are starts of lines in the source code. They are generated as (offset, lineno) pairs. See Objects/lnotab_notes.txt for the co_lnotab format and how to decode it.

バージョン 3.6 で変更: Line numbers can be decreasing. Before, they were always increasing.

dis.findlabels(*code*)

Detect all offsets in the raw compiled bytecode string *code* which are jump targets, and return a list of these offsets.

dis.stack_effect(*opcode*, *oparg=None*, *, *jump=None*)

opcode と引数 *oparg* がスタックに与える影響を計算します。

If the code has a jump target and *jump* is True, *stack_effect()* will return the stack effect of jumping. If *jump* is False, it will return the stack effect of not jumping. And if *jump* is None (default), it will return the maximal stack effect of both cases.

バージョン 3.4 で追加.

バージョン 3.8 で変更: Added *jump* parameter.

32.12.3 Python バイトコード命令

get_instructions() 関数と *Bytecode* クラスはバイトコード命令の詳細を *Instruction* インスタンスの形で提供します:

class dis.Instruction

バイトコード命令の詳細

opcode

以下の命令コードの値と 命令コードコレクション のバイトコードの値に対応する、命令の数値コードです。

opname

人間が読むための命令名

arg

(ある場合は) 命令の数値引数、無ければ None

argval

(もし分かっていれば) 解決された引数の値、そうでない場合は arg と同じもの

argrepr

人間が読むための命令引数の説明

offset

バイトコード列の中での命令の開始位置

starts_line
(ある場合は) この命令コードが始まる行、無ければ None

is_jump_target
他のコードからここへジャンプする場合は True 、そうでない場合は False

バージョン 3.4 で追加.

現在 Python コンパイラは次のバイトコード命令を生成します。

一般的な命令

NOP
なにもしないコード。バイトコードオプティマイザでプレースホルダとして使われます。

POP_TOP
スタックの先頭 (TOS) の要素を取り除きます。

ROT_TWO
スタックの先頭の 2 つの要素を入れ替えます。

ROT_THREE
スタックの二番目と三番目の要素の位置を 1 つ上げ、先頭を三番目へ下げます。

ROT_FOUR
Lifts second, third and fourth stack items one position up, moves top down to position four.

バージョン 3.8 で追加.

DUP_TOP
スタックの先頭にある参照の複製を作ります。

バージョン 3.2 で追加.

DUP_TOP_TWO
スタックの先頭の 2 つの参照を、そのままの順番で複製します。

バージョン 3.2 で追加.

1 オペランド命令

1 オペランド命令はスタックの先頭を取り出して操作を適用し、結果をスタックへプッシュし戻します。

UNARY_POSITIVE
TOS = +TOS を実行します。

UNARY_NEGATIVE
TOS = -TOS を実行します。

UNARY_NOT
TOS = not TOS を実行します。

UNARY_INVERT
: TOS = ~TOS を実行します。

GET_ITER
: TOS = iter(TOS) を実行します。

GET_YIELD_FROM_ITER
: TOS が *generator iterator* もしくは *coroutine* オブジェクトの場合は、そのままにしておきます。そうでない場合は TOS = iter(TOS) を実行します。

  バージョン 3.5 で追加.

2 オペランド命令

二項命令はスタックの先頭 (TOS) と先頭から二番目の要素をスタックから取り除きます。命令を実行し、スタックへ結果をプッシュし戻します。

BINARY_POWER
: TOS = TOS1 ** TOS を実行します。

BINARY_MULTIPLY
: TOS = TOS1 * TOS を実行します。

BINARY_MATRIX_MULTIPLY
: TOS = TOS1 @ TOS を実行します。

  バージョン 3.5 で追加.

BINARY_FLOOR_DIVIDE
: TOS = TOS1 // TOS を実行します。

BINARY_TRUE_DIVIDE
: TOS = TOS1 / TOS を実行します。

BINARY_MODULO
: TOS = TOS1 % TOS を実行します。

BINARY_ADD
: TOS = TOS1 + TOS を実行します。

BINARY_SUBTRACT
: TOS = TOS1 - TOS を実行します。

BINARY_SUBSCR
: TOS = TOS1[TOS] を実行します。

BINARY_LSHIFT
: TOS = TOS1 << TOS を実行します。

BINARY_RSHIFT
: TOS = TOS1 >> TOS を実行します。

BINARY_AND
: TOS = TOS1 & TOS を実行します。

BINARY_XOR
: TOS = TOS1 ^ TOS を実行します。

BINARY_OR
: TOS = TOS1 | TOS を実行します。

インプレース (in-place) 命令

インプレース命令は TOS と TOS1 を取り除いて結果をスタックへプッシュするという点で二項命令と似ています。しかし、TOS1 がインプレース命令をサポートしている場合には操作が直接 TOS1 に行われます。また、操作結果の TOS は (常に同じというわけではありませんが) 元の TOS1 と同じオブジェクトになることが多いです。

INPLACE_POWER
: インプレースの TOS = TOS1 ** TOS を実行します。

INPLACE_MULTIPLY
: インプレースの TOS = TOS1 * TOS を実行します。

INPLACE_MATRIX_MULTIPLY
: インプレースの TOS = TOS1 @ TOS を実行します。

  バージョン 3.5 で追加.

INPLACE_FLOOR_DIVIDE
: インプレースの TOS = TOS1 // TOS を実行します。

INPLACE_TRUE_DIVIDE
: インプレースの TOS = TOS1 / TOS を実行します。

INPLACE_MODULO
: インプレースの TOS = TOS1 % TOS を実行します。

INPLACE_ADD
: インプレースの TOS = TOS1 + TOS を実行します。

INPLACE_SUBTRACT
: インプレースの TOS = TOS1 - TOS を実行します。

INPLACE_LSHIFT
: インプレースの TOS = TOS1 << TOS を実行します。

INPLACE_RSHIFT
: インプレースの TOS = TOS1 >> TOS を実行します。

INPLACE_AND
: インプレースの TOS = TOS1 & TOS を実行します。

INPLACE_XOR
: インプレースの TOS = TOS1 ^ TOS を実行します。

INPLACE_OR
: インプレースの TOS = TOS1 | TOS を実行します。

STORE_SUBSCR
: TOS1[TOS] = TOS2 を実行します。

DELETE_SUBSCR
: del TOS1[TOS] を実行します。

コルーチン命令コード

GET_AWAITABLE
: TOS = get_awaitable(TOS) を実行します。get_awaitable(o) は、o がコルーチンオブジェクトもしくは CO_ITERABLE_COROUTINE フラグの付いたジェネレータオブジェクトの場合に o を返し、そうでない場合は o.__await__ を解決します。

  バージョン 3.5 で追加.

GET_AITER
: Implements TOS = TOS.__aiter__().

  バージョン 3.5 で追加.

  バージョン 3.7 で変更: Returning awaitable objects from __aiter__ is no longer supported.

GET_ANEXT
: PUSH(get_awaitable(TOS.__anext__())) を実行します。get_awaitable の詳細については GET_AWAITABLE を参照してください。

  バージョン 3.5 で追加.

END_ASYNC_FOR
: Terminates an async for loop. Handles an exception raised when awaiting a next item. If TOS is *StopAsyncIteration* pop 7 values from the stack and restore the exception state using the second three of them. Otherwise re-raise the exception using the three values from the stack. An exception handler block is removed from the block stack.

  バージョン 3.8 で追加.

BEFORE_ASYNC_WITH
: スタックの先頭にあるオブジェクトの __aenter__ と __aexit__ を解決します。__aexit__ と __aenter__() の結果をスタックに積みます。

  バージョン 3.5 で追加.

SETUP_ASYNC_WITH
: 新たなフレームオブジェクトを作成します。

  バージョン 3.5 で追加.

その他の命令コード

PRINT_EXPR
: 対話モードのための式文を実行します。TOS はスタックから取り除かれ表示されます。非対話モードにおいては、式文は *POP_TOP* で終了しています。

SET_ADD(*i*)
: `set.add(TOS1[-i], TOS)` を呼び出します。集合内包表記の実装に使われます。

LIST_APPEND(*i*)
: Calls `list.append(TOS1[-i], TOS)`. Used to implement list comprehensions.

MAP_ADD(*i*)
: Calls `dict.__setitem__(TOS1[-i], TOS1, TOS)`. Used to implement dict comprehensions.

  バージョン 3.1 で追加.

  バージョン 3.8 で変更: Map value is TOS and map key is TOS1. Before, those were reversed.

SET_ADD, *LIST_APPEND*, *MAP_ADD* は、追加した値または key/value ペアをスタックから取り除きますが、コンテナオブジェクトはループの次のイテレーションで利用できるようにスタックに残しておきます。

RETURN_VALUE
: 関数の呼び出し元へ TOS を返します。

YIELD_VALUE
: TOS をポップし、それを **ジェネレータ** から yield します。

YIELD_FROM
: TOS をポップし、それを *generator* から取得したサブイテレーターとして delegate します。

  バージョン 3.3 で追加.

SETUP_ANNOTATIONS
: Checks whether `__annotations__` is defined in `locals()`, if not it is set up to an empty `dict`. This opcode is only emitted if a class or module body contains *variable annotations* statically.

  バージョン 3.6 で追加.

IMPORT_STAR
: `'_'` で始まっていないすべてのシンボルをモジュール TOS から直接ローカル名前空間へロードします。モジュールはすべての名前をロードした後にポップされます。この命令コードは `from module import *` を実行します。

POP_BLOCK
: Removes one block from the block stack. Per frame, there is a stack of blocks, denoting `try` statements, and such.

POP_EXCEPT
: ブロックスタックからブロックを 1 つ取り除きます。ポップされたブロックは、例外ハンドラに入ったときに暗黙的に生成された例外ハンドラのブロックでなければなりません。フレームスタックから本

質的でない値をポップするのに加えて、直前にポップした 3 つの値が例外状態を回復するのに使われます。

POP_FINALLY(*preserve_tos*)

Cleans up the value stack and the block stack. If *preserve_tos* is not `0` TOS first is popped from the stack and pushed on the stack after performing other stack operations:

- If TOS is `NULL` or an integer (pushed by *BEGIN_FINALLY* or *CALL_FINALLY*) it is popped from the stack.
- If TOS is an exception type (pushed when an exception has been raised) 6 values are popped from the stack, the last three popped values are used to restore the exception state. An exception handler block is removed from the block stack.

It is similar to *END_FINALLY*, but doesn't change the bytecode counter nor raise an exception. Used for implementing `break`, `continue` and `return` in the `finally` block.

バージョン 3.8 で追加.

BEGIN_FINALLY

Pushes `NULL` onto the stack for using it in *END_FINALLY*, *POP_FINALLY*, *WITH_CLEANUP_START* and *WITH_CLEANUP_FINISH*. Starts the `finally` block.

バージョン 3.8 で追加.

END_FINALLY

Terminates a `finally` clause. The interpreter recalls whether the exception has to be re-raised or execution has to be continued depending on the value of TOS.

- If TOS is `NULL` (pushed by *BEGIN_FINALLY*) continue from the next instruction. TOS is popped.
- If TOS is an integer (pushed by *CALL_FINALLY*), sets the bytecode counter to TOS. TOS is popped.
- If TOS is an exception type (pushed when an exception has been raised) 6 values are popped from the stack, the first three popped values are used to re-raise the exception and the last three popped values are used to restore the exception state. An exception handler block is removed from the block stack.

LOAD_BUILD_CLASS

`builtins.__build_class__()` をスタックにプッシュします。これはクラスを構築するために、後で *CALL_FUNCTION* に呼ばれます。

SETUP_WITH(*delta*)

This opcode performs several operations before a with block starts. First, it loads `__exit__()` from the context manager and pushes it onto the stack for later use by *WITH_CLEANUP_START*. Then, `__enter__()` is called, and a finally block pointing to *delta* is pushed. Finally, the result of calling the `__enter__()` method is pushed onto the stack. The next opcode will either ignore it (*POP_TOP*), or store it in (a) variable(s) (*STORE_FAST*, *STORE_NAME*, or *UNPACK_SEQUENCE*).

バージョン 3.2 で追加.

WITH_CLEANUP_START
: Starts cleaning up the stack when a `with` statement block exits.

  At the top of the stack are either `NULL` (pushed by *BEGIN_FINALLY*) or 6 values pushed if an exception has been raised in the with block. Below is the context manager's `__exit__()` or `__aexit__()` bound method.

  If TOS is `NULL`, calls `SECOND(None, None, None)`, removes the function from the stack, leaving TOS, and pushes `None` to the stack. Otherwise calls `SEVENTH(TOP, SECOND, THIRD)`, shifts the bottom 3 values of the stack down, replaces the empty spot with `NULL` and pushes TOS. Finally pushes the result of the call.

WITH_CLEANUP_FINISH
: Finishes cleaning up the stack when a `with` statement block exits.

  TOS is result of `__exit__()` or `__aexit__()` function call pushed by *WITH_CLEANUP_START*. SECOND is `None` or an exception type (pushed when an exception has been raised).

  Pops two values from the stack. If SECOND is not None and TOS is true unwinds the EXCEPT_HANDLER block which was created when the exception was caught and pushes `NULL` to the stack.

All of the following opcodes use their arguments.

STORE_NAME(*namei*)
: `name = TOS` を実行します。*namei* はコードオブジェクトの属性 `co_names` における *name* のインデックスです。コンパイラは可能ならば *STORE_FAST* または *STORE_GLOBAL* を使おうとします。

DELETE_NAME(*namei*)
: `del name` を実行します。*namei* はコードオブジェクトの `co_names` 属性へのインデックスです。

UNPACK_SEQUENCE(*count*)
: TOS を *count* 個の個別の値にアンパックして、右から左の順にスタックに積みます。

UNPACK_EX(*counts*)
: 星付きの対象ありの代入を実行します: TOS にあるイテラブルを個別の値にばらしますが、ばらした値の総数はイテラブルの要素数より小さくなることがあります: そのときは、値の 1 つはばらされずに残った要素からなるリストです。

  counts の下位バイトはそのリスト値より前にある値の個数で、*counts* の上位バイトはそれより後ろにある値の個数です。そうしてできた値は右から左の順でスタックに積まれます。

STORE_ATTR(*namei*)
: `TOS.name = TOS1` を実行します。*namei* は `co_names` における名前のインデックスです。

DELETE_ATTR(*namei*)
: `del TOS.name` を実行します。`co_names` へのインデックスとして *namei* を使います。

STORE_GLOBAL(*namei*)
STORE_NAME と同じように動作しますが、name をグローバルとして保存します。

DELETE_GLOBAL(*namei*)
DELETE_NAME と同じように動作しますが、グローバルの name を削除します。

LOAD_CONST(*consti*)
`co_consts[consti]` をスタックにプッシュします。

LOAD_NAME(*namei*)
`co_names[namei]` に関連付けられた値をスタックにプッシュします。

BUILD_TUPLE(*count*)
スタックから *count* 個の要素を消費してタプルを作り出し、できたタプルをスタックにプッシュします。

BUILD_LIST(*count*)
BUILD_TUPLE と同じように動作しますが、この命令はリストを作り出します。

BUILD_SET(*count*)
BUILD_TUPLE と同じように動作しますが、この命令は set を作り出します。

BUILD_MAP(*count*)
Pushes a new dictionary object onto the stack. Pops `2 * count` items so that the dictionary holds *count* entries: `{..., TOS3: TOS2, TOS1: TOS}`.

バージョン 3.5 で変更: The dictionary is created from stack items instead of creating an empty dictionary pre-sized to hold *count* items.

BUILD_CONST_KEY_MAP(*count*)
The version of *BUILD_MAP* specialized for constant keys. Pops the top element on the stack which contains a tuple of keys, then starting from `TOS1`, pops *count* values to form values in the built dictionary.

バージョン 3.6 で追加.

BUILD_STRING(*count*)
Concatenates *count* strings from the stack and pushes the resulting string onto the stack.

バージョン 3.6 で追加.

BUILD_TUPLE_UNPACK(*count*)
Pops *count* iterables from the stack, joins them in a single tuple, and pushes the result. Implements iterable unpacking in tuple displays `(*x, *y, *z)`.

バージョン 3.5 で追加.

BUILD_TUPLE_UNPACK_WITH_CALL(*count*)
This is similar to *BUILD_TUPLE_UNPACK*, but is used for `f(*x, *y, *z)` call syntax. The stack item at position `count + 1` should be the corresponding callable `f`.

バージョン 3.6 で追加.

BUILD_LIST_UNPACK(*count*)
This is similar to *BUILD_TUPLE_UNPACK*, but pushes a list instead of tuple. Implements iterable unpacking in list displays `[*x, *y, *z]`.

バージョン 3.5 で追加.

BUILD_SET_UNPACK(*count*)
This is similar to *BUILD_TUPLE_UNPACK*, but pushes a set instead of tuple. Implements iterable unpacking in set displays `{*x, *y, *z}`.

バージョン 3.5 で追加.

BUILD_MAP_UNPACK(*count*)
Pops *count* mappings from the stack, merges them into a single dictionary, and pushes the result. Implements dictionary unpacking in dictionary displays `{**x, **y, **z}`.

バージョン 3.5 で追加.

BUILD_MAP_UNPACK_WITH_CALL(*count*)
This is similar to *BUILD_MAP_UNPACK*, but is used for `f(**x, **y, **z)` call syntax. The stack item at position `count + 2` should be the corresponding callable `f`.

バージョン 3.5 で追加.

バージョン 3.6 で変更: The position of the callable is determined by adding 2 to the opcode argument instead of encoding it in the second byte of the argument.

LOAD_ATTR(*namei*)
TOS を `getattr(TOS, co_names[namei])` と入れ替えます。

COMPARE_OP(*opname*)
ブール命令を実行します。命令名は `cmp_op[opname]` にあります。

IMPORT_NAME(*namei*)
モジュール `co_names[namei]` をインポートします。TOS と TOS1 がポップされ、*__import__()* の *fromlist* と *level* 引数になります。モジュールオブジェクトはスタックへプッシュされます。現在の名前空間は影響されません: 適切な import 文のためには、後続の *STORE_FAST* 命令が名前空間を変更します。

IMPORT_FROM(*namei*)
TOS にあるモジュールから属性 `co_names[namei]` をロードします。作成されたオブジェクトはスタックにプッシュされ、後続の *STORE_FAST* 命令によって保存されます。

JUMP_FORWARD(*delta*)
バイトコードカウンタを *delta* だけ増加させます。

POP_JUMP_IF_TRUE(*target*)
TOS が真ならば、バイトコードカウンタを *target* に設定します。TOS はポップされます。

バージョン 3.1 で追加.

POP_JUMP_IF_FALSE(*target*)
TOS が偽ならば、バイトコードカウンタを *target* に設定します。TOS はポップされます。

バージョン 3.1 で追加.

JUMP_IF_TRUE_OR_POP(*target*)
TOS が真ならば、バイトコードカウンタを *target* に設定し、TOS は スタックに残されます。そうでない (TOS が偽) なら、TOS はポップされます。

バージョン 3.1 で追加.

JUMP_IF_FALSE_OR_POP(*target*)
TOS が偽ならば、バイトコードカウンタを *target* に設定し、TOS は スタックに残されます。そうでない (TOS が真) なら、TOS はポップされます。

バージョン 3.1 で追加.

JUMP_ABSOLUTE(*target*)
バイトコードカウンタを *target* に設定します。

FOR_ITER(*delta*)
TOS はイテレータです。その *__next__()* メソッドを呼び出します。新しい値が yield された場合は、それをスタックにプッシュします (イテレータはその下に残されます)。イテレータの呼び出しで要素が尽きたことが示された場合は、TOS がポップされ、バイトコードカウンタが *delta* だけ増やされます。

LOAD_GLOBAL(*namei*)
co_names[namei] という名前のグローバルをスタック上にロードします。

SETUP_FINALLY(*delta*)
Pushes a try block from a try-finally or try-except clause onto the block stack. *delta* points to the finally block or the first except block.

CALL_FINALLY(*delta*)
Pushes the address of the next instruction onto the stack and increments bytecode counter by *delta*. Used for calling the finally block as a ”subroutine”.

バージョン 3.8 で追加.

LOAD_FAST(*var_num*)
ローカルな co_varnames[var_num] への参照をスタックにプッシュします。

STORE_FAST(*var_num*)
TOS をローカルな co_varnames[var_num] の中に保存します。

DELETE_FAST(*var_num*)
ローカルな co_varnames[var_num] を削除します。

LOAD_CLOSURE(*i*)
セルと自由変数の記憶領域のスロット *i* に含まれるセルへの参照をプッシュします。*i* が *co_cellvars*

の長さより小さければ、変数の名前は `co_cellvars[i]` です。そうでなければ `co_freevars[i - len(co_cellvars)]` です。

`LOAD_DEREF`(*i*)
セルと自由変数の記憶領域のスロット *i* に含まれるセルをロードします。セルが持つオブジェクトへの参照をスタックにプッシュします。

`LOAD_CLASSDEREF`(*i*)
LOAD_DEREF とほぼ同じですが、セルを調べる前にまずローカルの辞書を確認します。これはクラス本体に自由変数を読み込むために使います。

バージョン 3.4 で追加.

`STORE_DEREF`(*i*)
セルと自由変数の記憶領域のスロット *i* に含まれるセルへ TOS を保存します。

`DELETE_DEREF`(*i*)
セルと自由変数の記憶領域のスロット *i* にあるセルを空にします。`del` 文で使われます。

バージョン 3.2 で追加.

`RAISE_VARARGS`(*argc*)
Raises an exception using one of the 3 forms of the `raise` statement, depending on the value of *argc*:

- 0: `raise` (re-raise previous exception)
- 1: `raise TOS` (raise exception instance or type at `TOS`)
- 2: `raise TOS1 from TOS` (raise exception instance or type at `TOS1` with `__cause__` set to `TOS`)

`CALL_FUNCTION`(*argc*)
Calls a callable object with positional arguments. *argc* indicates the number of positional arguments. The top of the stack contains positional arguments, with the right-most argument on top. Below the arguments is a callable object to call. `CALL_FUNCTION` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

バージョン 3.6 で変更: This opcode is used only for calls with positional arguments.

`CALL_FUNCTION_KW`(*argc*)
Calls a callable object with positional (if any) and keyword arguments. *argc* indicates the total number of positional and keyword arguments. The top element on the stack contains a tuple of keyword argument names. Below that are keyword arguments in the order corresponding to the tuple. Below that are positional arguments, with the right-most parameter on top. Below the arguments is a callable object to call. `CALL_FUNCTION_KW` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

バージョン 3.6 で変更: Keyword arguments are packed in a tuple instead of a dictionary, *argc* indicates the total number of arguments.

`CALL_FUNCTION_EX`(*flags*)

Calls a callable object with variable set of positional and keyword arguments. If the lowest bit of *flags* is set, the top of the stack contains a mapping object containing additional keyword arguments. Below that is an iterable object containing positional arguments and a callable object to call. *BUILD_MAP_UNPACK_WITH_CALL* and *BUILD_TUPLE_UNPACK_WITH_CALL* can be used for merging multiple mapping objects and iterables containing arguments. Before the callable is called, the mapping object and iterable object are each "unpacked" and their contents passed in as keyword and positional arguments respectively. `CALL_FUNCTION_EX` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

バージョン 3.6 で追加.

`LOAD_METHOD`(*namei*)

Loads a method named `co_names[namei]` from the TOS object. TOS is popped. This bytecode distinguishes two cases: if TOS has a method with the correct name, the bytecode pushes the unbound method and TOS. TOS will be used as the first argument (`self`) by *CALL_METHOD* when calling the unbound method. Otherwise, `NULL` and the object return by the attribute lookup are pushed.

バージョン 3.7 で追加.

`CALL_METHOD`(*argc*)

Calls a method. *argc* is the number of positional arguments. Keyword arguments are not supported. This opcode is designed to be used with *LOAD_METHOD*. Positional arguments are on top of the stack. Below them, the two items described in *LOAD_METHOD* are on the stack (either `self` and an unbound method object or `NULL` and an arbitrary callable). All of them are popped and the return value is pushed.

バージョン 3.7 で追加.

`MAKE_FUNCTION`(*flags*)

Pushes a new function object on the stack. From bottom to top, the consumed stack must consist of values if the argument carries a specified flag value

- `0x01` a tuple of default values for positional-only and positional-or-keyword parameters in positional order
- `0x02` a dictionary of keyword-only parameters' default values
- `0x04` an annotation dictionary
- `0x08` a tuple containing cells for free variables, making a closure
- 関数に関連付けられたコード (TOS1 の位置)
- 関数の *qualified name* (TOS の位置)

BUILD_SLICE(*argc*)
: スライスオブジェクトをスタックにプッシュします。*argc* は 2 あるいは 3 でなければなりません。2 ならば slice(TOS1, TOS) がプッシュされます。3 ならば slice(TOS2, TOS1, TOS) がプッシュされます。これ以上の情報については、*slice()* 組み込み関数を参照してください。

EXTENDED_ARG(*ext*)
: Prefixes any opcode which has an argument too big to fit into the default one byte. *ext* holds an additional byte which act as higher bits in the argument. For each opcode, at most three prefixal EXTENDED_ARG are allowed, forming an argument from two-byte to four-byte.

FORMAT_VALUE(*flags*)
: Used for implementing formatted literal strings (f-strings). Pops an optional *fmt_spec* from the stack, then a required *value*. *flags* is interpreted as follows:

  - (flags & 0x03) == 0x00: *value* is formatted as-is.
  - (flags & 0x03) == 0x01: call *str()* on *value* before formatting it.
  - (flags & 0x03) == 0x02: call *repr()* on *value* before formatting it.
  - (flags & 0x03) == 0x03: call *ascii()* on *value* before formatting it.
  - (flags & 0x04) == 0x04: pop *fmt_spec* from the stack and use it, else use an empty *fmt_spec*.

  Formatting is performed using PyObject_Format(). The result is pushed on the stack.

  バージョン 3.6 で追加.

HAVE_ARGUMENT
: This is not really an opcode. It identifies the dividing line between opcodes which don't use their argument and those that do (< HAVE_ARGUMENT and >= HAVE_ARGUMENT, respectively).

  バージョン 3.6 で変更: Now every instruction has an argument, but opcodes < HAVE_ARGUMENT ignore it. Before, only opcodes >= HAVE_ARGUMENT had an argument.

32.12.4 命令コードコレクション

これらのコレクションは、自動でバイトコード命令を解析するために提供されています:

dis.opname
: 命令コード名のリスト。バイトコードをインデックスに使って参照できます。

dis.opmap
: 命令コード名をバイトコードに対応づける辞書。

dis.cmp_op
: すべての比較命令の名前のリスト。

dis.hasconst
定数にアクセスするバイトコードのリスト。

dis.hasfree
自由変数にアクセスするバイトコードのリスト (この文脈での ' 自由' とは、現在のスコープにある名前で内側のスコープから参照されているもの、もしくは外側のスコープにある名前で現在のスコープから参照しているものを指します。グローバルスコープや組み込みのスコープへの参照は含み **ません**)。

dis.hasname
名前によって属性にアクセスするバイトコードのリスト。

dis.hasjrel
相対ジャンプ先を持つバイトコードのリスト。

dis.hasjabs
絶対ジャンプ先を持つバイトコードのリスト。

dis.haslocal
ローカル変数にアクセスするバイトコードのリスト。

dis.hascompare
ブール命令のバイトコードのリスト。

32.13 pickletools --- pickle 開発者のためのツール群

ソースコード: Lib/pickletools.py

このモジュールには、*pickle* モジュールの詳細に関わる様々な定数や実装に関する長大なコメント、そして pickle 化されたデータを解析する上で有用な関数をいくつか定義しています。このモジュールの内容は *pickle* の実装に関わっている Python コア開発者にとって有用なものです; 普通の *pickle* 利用者にとっては、*pickletools* モジュールはおそらく関係ないものでしょう。

32.13.1 コマンドラインの使い方

バージョン 3.2 で追加.

コマンドラインから実行するとき、`python -m pickletools` は 1 つもしくは複数の pickle ファイルの内容を逆アセンブルします。pickle 形式の詳細ではなく pickle に保存された Python オブジェクトを見たい場合は、そのコマンドではなく `-m pickle` を使いたいと思うかもしれません。しかし、調べたい pickle ファイルが信頼できないソースから来たものであるとき、`-m pickletools` は pickle のバイトコードを実行しないので、より安全な選択肢です。

例えば、`x.pickle` ファイルに pickle 化されているタプル `(1, 2)` に対して実行すると次のようになります:

```
$ python -m pickle x.pickle
(1, 2)

$ python -m pickletools x.pickle
    0: \x80 PROTO      3
    2: K    BININT1    1
    4: K    BININT1    2
    6: \x86 TUPLE2
    7: q    BINPUT     0
    9: .    STOP
highest protocol among opcodes = 2
```

コマンドラインオプション

-a, --annotate
注釈として短い命令コードの説明を各行に表示します。

-o, --output=<file>
出力結果を書き込むファイル名。

-l, --indentlevel=<num>
新しい MARK レベルのインデントに使われる空白の数。

-m, --memo
複数のオブジェクトが逆アセンブルされたとき、逆アセンブリ間でメモを保持します。

-p, --preamble=<preamble>
複数の pickle ファイルが指定されたとき、各逆アセンブリの前に与えられたプリアンブルを表示します。

32.13.2 プログラミングインターフェース

pickletools.dis(*pickle*, *out=None*, *memo=None*, *indentlevel=4*, *annotate=0*)
pickle の抽象的な逆アセンブリを file-like オブジェクト *out* (デフォルトは `sys.stdout`) に出力します。*pickle* は文字列または file-like オブジェクトです。*memo* は Python の辞書で、pickle のメモとして使われます; これは、pickle 処理を行う 1 つのオブジェクトが、複数の pickle にわたって逆アセンブルを行うために使われます。ストリーム上の `MARK` 命令コードが示す後続のレベルは、*indentlevel* 個の空白でインデントされます。*annotate* に非ゼロの値が与えられた場合、出力される各命令コードは短い命令コードに注釈が付けられます。*annotate* の値は、注釈の先頭の位置のヒントとして使われます。

バージョン 3.2 で追加: *annotate* 引数。

pickletools.genops(*pickle*)
pickle 内の全ての opcode を取り出す イテレータ を返します。このイテレータは `(opcode, arg, pos)` の三つ組みからなる配列を返します。*opcode* は `OpcodeInfo` クラスのインスタンスのクラスです。*arg* は *opcode* の引数としてデコードされた Python オブジェクトの値です。*pos* は *opcode* の場所を表す値です。*pickle* は文字列でもファイル類似オブジェクトでもかまいません。

pickletools.**optimize**(*picklestring*)

使われていない PUT 命令コードを除去した上で、その新しい pickle 文字列を返します。最適化された pickle は、長さがより短く、転送時間がより少なく、必要とするストレージ領域がより狭く、unpickle 化がより効率的になります。

第

THIRTYTHREE

各種サービス

この章では、Python のすべてのバージョンで利用可能な各種サービスについて説明します。以下に概要を示します:

33.1 formatter --- 汎用の出力書式化機構

バージョン 3.4 で非推奨: 使用法がないため formatter モジュールは非推奨になりました。

このモジュールでは、二つのインタフェース定義を提供しており、それらの各インタフェースについて複数の実装を提供しています。*formatter* インタフェースと、formatter インタフェースに必要なされる *writer* インタフェースです。

formatter オブジェクトはある抽象化された書式イベントの流れを writer オブジェクト上の特定の出力イベントに変換します。formatter はいくつかのスタック構造を管理することで、writer オブジェクトの様々な属性を変更したり復元したりできるようにしています; このため、writer は相対的な変更や ” 元に戻す” 操作を処理できなくてもかまいません。writer の特定のプロパティのうち、formatter オブジェクトを介して制御できるのは、水平方向の字揃え、フォント、そして左マージンの字下げです。任意の、非排他的なスタイル設定を writer に提供するためのメカニズムも提供されています。さらに、段落分割のように、可逆でない書式化イベントの機能を提供するインタフェースもあります。

writer オブジェクトはデバイスインタフェースをカプセル化します。ファイル形式のような抽象デバイスも物理デバイス同様にサポートされています。ここで提供されている実装内容はすべて抽象デバイス上で動作します。デバイスインタフェースは formatter オブジェクトが管理しているプロパティを設定し、データを出力端に書き込めるようにします。

33.1.1 formatter インタフェース

formatter を作成するためのインタフェースは、インスタンス化しようとする個々の formatter クラスに依存します。以下で解説するのは、インスタンス化された全ての formatter がサポートしなければならないインタフェースです。

モジュールレベルではデータ要素を一つ定義しています:

formatter.AS_IS
: 後に述べる push_font() メソッドでフォント指定をする時に使える値です。また、その他の push_property() メソッドの新しい値として使うことができます。AS_IS の値をスタックに置くと、どのプロパティが変更されたかの追跡を行わずに、対応する pop_property() メソッドが呼び出されるようになります。

formatter インスタンスオブジェクトには以下の属性が定義されています:

formatter.writer
: formatter とやり取りを行う writer インスタンスです。

formatter.end_paragraph(*blanklines*)
: 開かれている段落があれば閉じ、次の段落との間に少なくとも *blanklines* が挿入されるようにします。

formatter.add_line_break()
: 強制改行挿入します。既に強制改行がある場合は挿入しません。論理的な段落は中断しません。

formatter.add_hor_rule(**args*, ***kw*)
: 出力に水平罫線を挿入します。現在の段落に何らかのデータがある場合、強制改行が挿入されますが、論理的な段落は中断しません。引数とキーワードは writer の send_line_break() メソッドに渡されます。

formatter.add_flowing_data(*data*)
: 空白を折りたたんで書式化しなければならないデータを提供します。空白の折りたたみでは、直前や直後の *add_flowing_data()* 呼び出しに入っている空白も考慮されます。このメソッドに渡されたデータは出力デバイスで行末の折り返し (word-wrap) されるものと想定されています。出力デバイスでの要求やフォント情報に応じて、writer オブジェクトでも何らかの行末折り返しが行われなければならないので注意してください。

formatter.add_literal_data(*data*)
: 変更を加えずに writer に渡さなければならないデータを提供します。改行およびタブを含む空白を *data* の値にしても問題ありません。

formatter.add_label_data(*format*, *counter*)
: 現在の左マージン位置の左側に配置されるラベルを挿入します。このラベルは箇条書き、数字つき箇条書きの書式を構築する際に使われます。*format* の値が文字列の場合、整数の値 *counter* の書式指定として解釈されます。*format* の値が文字列の場合、整数の値をとる *counter* の書式化指定として解釈されます。書式化された文字列はラベルの値になります; *format* が文字列でない場合、ラベルの値として直接使われます。ラベルの値は writer の send_label_data() メソッドの唯一の引数として渡されます。非文字列のラベル値をどう解釈するかは関連付けられた writer に依存します。

  書式化指定は文字列からなり、counter の値と合わせてラベルの値を算出するために使われます。書式文字列の各文字はラベル値にコピーされます。このときいくつかの文字は counter 値を変換を指すものとして認識されます。特に、文字 '1' はアラビア数字の counter 値を表し、'A' と 'a' はそれぞれ大文字および小文字のアルファベットによる counter 値を表し、'I' と 'i' はそれぞれ大文字および小文字のローマ数字による counter 値を表します。アルファベットおよびローマ字数字への変換の際には、counter の値はゼロ以上である必要があるので注意してください。

formatter.flush_softspace()
: 以前の *add_flowing_data()* 呼び出しでバッファされている出力待ちの空白を、関連付けられている writer オブジェクトに送信します。このメソッドは writer オブジェクトに対するあらゆる直接操作の前に呼び出さなければなりません。

formatter.push_alignment(*align*)
: 新たな字揃え (alignment) 設定を字揃えスタックの上にプッシュします。変更を行いたくない場合には *AS_IS* にすることができます。字揃え設定値が以前の設定から変更された場合、writer の `new_alignment()` メソッドが *align* の値と共に呼び出されます。

formatter.pop_alignment()
: 以前の字揃え設定を復元します。

formatter.push_font(*(size, italic, bold, teletype)*)
: writer オブジェクトのフォントプロパティのうち、一部または全てを変更します。*AS_IS* に設定されていないプロパティは引数で渡された値に設定され、その他の値は現在の設定を維持します。writer の `new_font()` メソッドは完全に設定解決されたフォント指定で呼び出されます。

formatter.pop_font()
: 以前のフォント設定を復元します。

formatter.push_margin(*margin*)
: 左マージンのインデント数を一つ増やし、論理タグ *margin* を新たなインデントに関連付けます。マージンレベルの初期値は `0` です。変更された論理タグの値は真値とならなければなりません; *AS_IS* 以外の偽の値はマージンの変更としては不適切です。

formatter.pop_margin()
: 以前のマージン設定を復元します。

formatter.push_style(**styles*)
: 任意のスタイル指定をスタックにプッシュします。全てのスタイルはスタイルスタックに順番にプッシュされます。*AS_IS* 値を含み、スタック全体を表すタプルは writer の `new_styles()` メソッドに渡されます。

formatter.pop_style(*n=1*)
: *push_style()* に渡された最新 *n* 個のスタイル指定をポップします。*AS_IS* 値を含み、変更されたスタックを表すタプルは writer の `new_styles()` メソッドに渡されます。

formatter.set_spacing(*spacing*)
: writer の割り付けスタイル (spacing style) を設定します。

formatter.assert_line_data(*flag=1*)
: 現在の段落にデータが予期せず追加されたことを formatter に知らせます。このメソッドは writer を直接操作した際に使わなければなりません。writer 操作の結果、出力の末尾が強制改行となった場合、オプションの *flag* 引数を偽に設定することができます。

33.1.2 formatter 実装

このモジュールでは、formatter オブジェクトに関して二つの実装を提供しています。ほとんどのアプリケーションではこれらのクラスを変更したりサブクラス化することなく使うことができます。

class formatter.NullFormatter(*writer=None*)

何も行わない formatter です。*writer* を省略すると、*NullWriter* インスタンスが生成されます。*NullFormatter* インスタンスは、writer のメソッドを全く呼び出しません。writer へのインタフェースを実装する場合にはこのクラスのインタフェースを継承する必要がありますが、実装を継承する必要は全くありません。

class formatter.AbstractFormatter(*writer*)

標準の formatter です。この formatter 実装は広範な writer で適用できることが実証されており、ほとんどの状況で直接使うことができます。高機能の WWW ブラウザを実装するために使われたこともあります。

33.1.3 writer インタフェース

writer を作成するためのインタフェースは、インスタンス化しようとする個々の writer クラスに依存します。以下で解説するのは、インスタンス化された全ての writer がサポートしなければならないインタフェースです。ほとんどのアプリケーションでは *AbstractFormatter* クラスを formatter として使うことができますが、通常 writer はアプリケーション側で与えなければならないので注意してください。

writer.flush()

バッファに蓄積されている出力データやデバイス制御イベントをフラッシュします。

writer.new_alignment(*align*)

字揃えのスタイルを設定します。*align* の値は任意のオブジェクトを取りえますが、慣習的な値は文字列または `None` で、`None` は writer の ”好む” 字揃えを使うことを表します。慣習的な *align* の値は `'left'`、`'center'`、`'right'`、および `'justify'` です。

writer.new_font(*font*)

フォントスタイルを設定します。*font* は、デバイスの標準のフォントが使われることを示す `None` か、`(size, italic, bold, teletype)` の形式をとるタプルになります。size はフォントサイズを示す文字列になります; 特定の文字列やその解釈はアプリケーション側で定義します。*italic*、*bold*、および *teletype* といった値はブール値で、それらの属性を使うかどうかを指定します。

writer.new_margin(*margin*, *level*)

マージンレベルを整数値 *level* に設定し、論理タグ (logical tag) を *margin* に設定します。論理タグの解釈は writer の判断に任されます; 論理タグの値に対する唯一の制限は *level* が非ゼロの値の際に偽であってはならないということです。

writer.new_spacing(*spacing*)

割り付けスタイル (spacing style) を *spacing* に設定します。

writer.new_styles(*styles*)

追加のスタイルを設定します。*styles* の値は任意の値からなるタプルです; *AS_IS* 値は無視されます。

styles タプルはアプリケーションや writer の実装上の都合により、集合としても、スタックとしても解釈され得ます。

writer.send_line_break()
現在の行を改行します。

writer.send_paragraph(*blankline*)
少なくとも *blankline* 空行分の間隔か、空行そのもので段落を分割します。*blankline* の値は整数になります。writer の実装では、改行を行う必要がある場合、このメソッドの呼び出しに先立って *send_line_break()* の呼び出しを受ける必要あります; このメソッドには段落の最後の行を閉じる機能は含まれておらず、段落間に垂直スペースを空ける役割しかありません。

writer.send_hor_rule(**args*, ***kw*)
水平罫線を出力デバイスに表示します。このメソッドへの引数は全てアプリケーションおよび writer 特有のものなので、注意して解釈する必要があります。このメソッドの実装では、すでに改行が *send_line_break()* によってなされているものと仮定しています。

writer.send_flowing_data(*data*)
行端が折り返され、必要に応じて再割り付け解析を行った (re-flowed) 文字データを出力します。このメソッドを連続して呼び出す上では、writer は複数の空白文字は単一のスペース文字に縮約されていると仮定することがあります。

writer.send_literal_data(*data*)
すでに表示用に書式化された文字データを出力します。これは通常、改行文字で表された改行を保存し、新たに改行を持ち込まないことを意味します。`send_formatted_data()` インタフェースと違って、データには改行やタブ文字が埋め込まれていてもかまいません。

writer.send_label_data(*data*)
可能ならば、*data* を現在の左マージンの左側に設定します。*data* の値には制限がありません; 文字列でない値の扱い方はアプリケーションや writer に完全に依存します。このメソッドは行の先頭でのみ呼び出されます。

33.1.4 writer 実装

このモジュールでは、3 種類の writer オブジェクトインタフェース実装を提供しています。ほとんどのアプリケーションでは、*NullWriter* から新しい writer クラスを派生する必要があるでしょう。

class formatter.NullWriter
インタフェース定義だけを提供する writer クラスです; どのメソッドも何ら処理を行いません。このクラスは、メソッド実装をまったく継承する必要のない writer 全ての基底クラスになります。

class formatter.AbstractWriter
この writer は formatter をデバッグするのに利用できますが、それ以外に利用できるほどのものではありません。各メソッドを呼び出すと、メソッド名と引数を標準出力に印字して呼び出されたことを示します。

class formatter.DumbWriter(*file=None*, *maxcol=72*)

単純な writer クラスで *file* に渡された **ファイルオブジェクト** か *file* が省略された場合には標準出力に出力を書き込みます。出力は *maxcol* で指定されたカラム数で単純な行端折り返しが行われます。このクラスは連続した段落を再割り付けするのに適しています。

第

THIRTYFOUR

MS WINDOWS 固有のサービス

この章では、MS Windows プラットフォーム上でのみ利用可能なモジュール群について記述します。

34.1 msilib --- Microsoft インストーラーファイルの読み書き

ソースコード: Lib/msilib/__init__.py

---

msilib モジュールは Microsoft インストーラー (.msi) の作成を支援します。このファイルは大抵の場合埋め込まれた「キャビネット」ファイル (.cab) を含むので、CAB ファイル作成用の API も公開されています。.cab ファイルの読み出しは現時点では実装されていませんが、.msi データベースの読み出しは可能です。

このパッケージの目的は .msi ファイルにある全てのテーブルへの完全なアクセスの提供であり、提供されているものは非常に低レベルな API です。このパッケージの二つの主要な使用目的は *distutils* の bdist_msi コマンドと、Python インストーラーパッケージそれ自体 (と言いつつ現在は別バージョンの msilib を使っているのですが) です。

パッケージの内容は大きく四つのパートに分けられます。低レベル CAB ルーチン、低レベル MSI ルーチン、少し高レベルの MSI ルーチン、標準的なテーブル構造、の四つです。

msilib.**FCICreate**(*cabname*, *files*)

新しい CAB ファイルを *cabname* という名前で作ります。*files* はタプルのリストで、それぞれのタプルはディスク上のファイル名と CAB ファイルで付けられるファイル名で構成されなければなりません。

ファイルはリストに現れた順番で CAB ファイルに追加されます。全てのファイルは MSZIP 圧縮アルゴリズムを使って一つの CAB ファイルに追加されます。

MSI 作成の様々なステップに対する Python コールバックは現在公開されていません。

msilib.**UuidCreate**()

新しい一意な識別子の文字列表現を返します。この関数は Windows API の関数 UuidCreate() と UuidToString() をラップしたものです。

msilib.**OpenDatabase**(*path*, *persist*)

MsiOpenDatabase を呼び出して新しいデータベースオブジェクトを返します。*path* は MSI フ

ァイルのファイル名です。*persist* は五つの定数 MSIDBOPEN_CREATEDIRECT, MSIDBOPEN_CREATE, MSIDBOPEN_DIRECT, MSIDBOPEN_READONLY, MSIDBOPEN_TRANSACT のどれか一つで、フラグ MSIDBOPEN_PATCHFILE を含めても構いません。これらのフラグの意味は Microsoft のドキュメントを参照してください。フラグに従って、既存のデータベースを開いたり新しいデータベースを作成したりします。

msilib.**CreateRecord**(*count*)

MSICreateRecord() を呼び出して新しいレコードオブジェクトを返します。*count* はレコードのフィールドの数です。

msilib.**init_database**(*name*, *schema*, *ProductName*, *ProductCode*, *ProductVersion*, *Manufacturer*)

name という名前の新しいデータベースを作り、*schema* で初期化し、プロパティ *ProductName*, *ProductCode*, *ProductVersion*, *Manufacturer* をセットして、返します。

schema は tables と _Validation_records という属性をもったモジュールオブジェクトでなければなりません。大抵の場合、*msilib.schema* を使うべきです。

データベースはこの関数から返された時点でスキーマとバリデーションレコードだけが収められています。

msilib.**add_data**(*database*, *table*, *records*)

全ての *records* を *database* の *table* テーブルに追加します。

table 引数は MSI スキーマで事前に定義されたテーブルでなければなりません。例えば、'Feature', 'File', 'Component', 'Dialog', 'Control', などです。

records はタプルのリストで、それぞれのタプルにはテーブルのスキーマに従ったレコードの全てのフィールドを含んでいるものでなければなりません。オプションのフィールドには None を渡すことができます。

フィールドの値には、整数・文字列・Binary クラスのインスタンスが使えます。

class msilib.**Binary**(*filename*)

Binary テーブル中のエントリーを表わします。*add_data()* を使ってこのクラスのオブジェクトを挿入するときには *filename* という名前のファイルをテーブルに読み込みます。

msilib.**add_tables**(*database*, *module*)

module の全てのテーブルの内容を *database* に追加します。*module* は *tables* という内容が追加されるべき全てのテーブルのリストと、テーブルごとに一つある実際の内容を持っている属性とを含んでいなければなりません。

この関数は典型的にシーケンステーブルをインストールするために使われます。

msilib.**add_stream**(*database*, *name*, *path*)

database の _Stream テーブルに、ファイル *path* を *name* というストリーム名で追加します。

msilib.**gen_uuid**()

新しい UUID を、MSI が通常要求する形式 (つまり、中括弧で囲み、16 進数は大文字) で返します。

参考:

FCICreate UuidCreate UuidToString

34.1.1 データベースオブジェクト

Database.**OpenView**(*sql*)

MSIDatabaseOpenView() を呼び出して取得したビューオブジェクトを返します。*sql* は実行される SQL ステートメントです。

Database.**Commit**()

MSIDatabaseCommit() を呼び出して現在のトランザクションで保留されている変更をコミットします。

Database.**GetSummaryInformation**(*count*)

MsiGetSummaryInformation() を呼び出して新しいサマリー情報オブジェクトを返します。*count* は更新された値の最大数です。

Database.**Close**()

MsiCloseHandle() を通してデータベースオブジェクトを閉じます。

バージョン 3.7 で追加.

参考:

MSIDatabaseOpenView MSIDatabaseCommit MSIGetSummaryInformation MsiCloseHandle

34.1.2 ビューオブジェクト

View.**Execute**(*params*)

MSIViewExecute() を通してビューに対する SQL 問い合わせを実行します。*params* が None でない場合、クエリ中のパラメータトークンの実際の値を与えるものです。

View.**GetColumnInfo**(*kind*)

MsiViewGetColumnInfo() の呼び出しを通してビューのカラムを説明するレコードを返します。*kind* は MSICOLINFO_NAMES または MSICOLINFO_TYPES です。

View.**Fetch**()

MsiViewFetch() の呼び出しを通してクエリの結果レコードを返します。

View.**Modify**(*kind*, *data*)

MsiViewModify() を呼び出してビューを変更します。*kind* は MSIMODIFY_SEEK, MSIMODIFY_REFRESH, MSIMODIFY_INSERT, MSIMODIFY_UPDATE, MSIMODIFY_ASSIGN, MSIMODIFY_REPLACE, MSIMODIFY_MERGE, MSIMODIFY_DELETE, MSIMODIFY_INSERT_TEMPORARY, MSIMODIFY_VALIDATE, MSIMODIFY_VALIDATE_NEW, MSIMODIFY_VALIDATE_FIELD, MSIMODIFY_VALIDATE_DELETE のいずれかです。

data は新しいデータを表わすレコードでなければなりません。

View.Close()
: MsiViewClose() を通してビューを閉じます。

参考:

MsiViewExecute MSIViewGetColumnInfo MsiViewFetch MsiViewModify MsiViewClose

34.1.3 サマリー情報オブジェクト

SummaryInformation.GetProperty(*field*)
: MsiSummaryInfoGetProperty() を通してサマリーのプロパティを返します。*field* はプロパティ名で、定数 PID_CODEPAGE, PID_TITLE, PID_SUBJECT, PID_AUTHOR, PID_KEYWORDS, PID_COMMENTS, PID_TEMPLATE, PID_LASTAUTHOR, PID_REVNUMBER, PID_LASTPRINTED, PID_CREATE_DTM, PID_LASTSAVE_DTM, PID_PAGECOUNT, PID_WORDCOUNT, PID_CHARCOUNT, PID_APPNAME, PID_SECURITY のいずれかです。

SummaryInformation.GetPropertyCount()
: MsiSummaryInfoGetPropertyCount() を通してサマリープロパティの個数を返します。

SummaryInformation.SetProperty(*field*, *value*)
: MsiSummaryInfoSetProperty() を通してプロパティをセットします。*field* は *GetProperty()* におけるものと同じ値をとります。*value* はプロパティの新しい値です。許される値の型は整数と文字列です。

SummaryInformation.Persist()
: MsiSummaryInfoPersist() を使って変更されたプロパティをサマリー情報ストリームに書き込みます。

参考:

MsiSummaryInfoGetProperty MsiSummaryInfoGetPropertyCount MsiSummaryInfoSetProperty MsiSummaryInfoPersist

34.1.4 レコードオブジェクト

Record.GetFieldCount()
: MsiRecordGetFieldCount() を通してレコードのフィールド数を返します。

Record.GetInteger(*field*)
: *field* の値を可能なら整数として返します。*field* は整数でなければなりません。

Record.GetString(*field*)
: *field* の値を可能なら文字列として返します。*field* は整数でなければなりません。

Record.SetString(*field*, *value*)
: MsiRecordSetString() を通して *field* を *value* にセットします。*field* は整数、*value* は文字列でなければなりません。

Record.**SetStream**(*field*, *value*)

MsiRecordSetStream() を通して *field* を *value* という名のファイルの内容にセットします。*field* は整数、*value* は文字列でなければなりません。

Record.**SetInteger**(*field*, *value*)

MsiRecordSetInteger() を通して *field* を *value* にセットします。*field* も *value* も整数でなければなりません。

Record.**ClearData**()

MsiRecordClearData() を通してレコードの全てのフィールドを 0 にセットします。

参考:

MsiRecordGetFieldCount MsiRecordSetString MsiRecordSetStream MsiRecordSetInteger MsiRecordClearData

34.1.5 エラー

全ての MSI 関数のラッパーは MSIError を送出します。例外の内部の文字列がより詳細な情報を含んでいます。

34.1.6 CAB オブジェクト

class msilib.**CAB**(*name*)

CAB クラスは CAB ファイルを表わすものです。MSI 構築中、ファイルは Files テーブルと CAB ファイルとに同時に追加されます。そして、全てのファイルを追加し終えたら、CAB ファイルは書き込まれることが可能になり、MSI ファイルに追加されます。

name は MSI ファイル中の CAB ファイルの名前です。

append(*full*, *file*, *logical*)

パス名 *full* のファイルを CAB ファイルに *logical* という名で追加します。*logical* という名が既に存在したならば、新しいファイル名が作られます。

ファイルの CAB ファイル中のインデクスと新しいファイル名を返します。

commit(*database*)

CAB ファイルを作り、MSI ファイルにストリームとして追加し、Media テーブルに送り込み、作ったファイルはディスクから削除します。

34.1.7 ディレクトリオブジェクト

class msilib.Directory(*database*, *cab*, *basedir*, *physical*, *logical*, *default*[, *componentflags*])

新しいディレクトリを Directory テーブルに作成します。ディレクトリには各時点で現在のコンポーネントがあります。そのコンポーネントは *start_component()* を使って明示的に作成されたか、最初にファイルが追加された際に暗黙裡に作成されたものです。ファイルは現在のコンポーネントと cab ファイルの両方に追加されます。ディレクトリを作成するには親ディレクトリオブジェクト (None でも可)、物理的ディレクトリへのパス、論理的ディレクトリ名を指定する必要があります。*default* はディレクトリテーブルの DefaultDir スロットを指定します。*componentflags* は新しいコンポーネントが得るデフォルトのフラグを指定します。

start_component(*component=None*, *feature=None*, *flags=None*, *keyfile=None*, *uuid=None*)

エントリを Component テーブルに追加し、このコンポーネントをこのディレクトリの現在のコンポーネントにします。もしコンポーネント名が与えられなければディレクトリ名が使われます。*feature* が与えられなければ、ディレクトリのデフォルトフラグが使われます。*keyfile* が与えられなければ、Component テーブルの KeyPath は null のままになります。

add_file(*file*, *src=None*, *version=None*, *language=None*)

ファイルをディレクトリの現在のコンポーネントに追加します。現在のコンポーネントが存在しない場合、新しいコンポーネントを開始します。デフォルトではソースとファイルテーブルのファイル名は同じになります。*src* ファイルが指定された場合、それは現在のディレクトリから相対的に解釈されます。オプションで *version* と *language* を File テーブルのエントリ用に指定することができます。

glob(*pattern*, *exclude=None*)

現在のコンポーネントに glob パターンで指定されたファイルのリストを追加します。個々のファイルを *exclude* リストで除外することができます。

remove_pyc()

アンインストールの際に .pyc を削除します。

参考:

Directory Table File Table Component Table FeatureComponents Table

34.1.8 フィーチャー

class msilib.Feature(*db*, *id*, *title*, *desc*, *display*, *level=1*, *parent=None*, *directory=None*, *attributes=0*)

id, *parent.id*, *title*, *desc*, *display*, *level*, *directory*, *attributes* の値を使って、新しいレコードを Feature テーブルに追加します。出来上がったフィーチャーオブジェクトは *Directory* の start_component() メソッドに渡すことができます。

set_current()

このフィーチャーを *msilib* の現在のフィーチャーにします。フィーチャーを明示的に指定しない場合、新しいコンポーネントが自動的にデフォルトのフィーチャーに追加されます。

参考:

Feature Table

34.1.9 GUI クラス

msilib は MSI データベースにある GUI テーブルをラップしたいくつかのクラスを提供します。しかし、標準的なユーザーインタフェースは提供されません。ユーザーインタフェースを備えた Python パッケージをインストールする MSI を作成するには `bdist_msi` を使用してください。

class msilib.**Control**(*dlg*, *name*)

ダイアログコントロールの基底クラス。*dlg* はコントロールの属するダイアログオブジェクト、*name* はコントロールの名前です。

event(*event*, *argument*, *condition=1*, *ordering=None*)

このコントロールの `ControlEvent` テーブルにエントリを作ります。

mapping(*event*, *attribute*)

このコントロールの `EventMapping` テーブルにエントリを作ります。

condition(*action*, *condition*)

このコントロールの `ControlCondition` テーブルにエントリを作ります。

class msilib.**RadioButtonGroup**(*dlg*, *name*, *property*)

name という名前のラジオボタンコントロールを作成します。*property* はラジオボタンが選ばれたときにセットされるインストーラープロパティです。

add(*name*, *x*, *y*, *width*, *height*, *text*, *value=None*)

グループに *name* という名前で、座標 *x*, *y* に大きさが *width*, *height* で *text* というラベルの付いたラジオボタンを追加します。*value* が `None` なら、デフォルトは *name* になります。

class msilib.**Dialog**(*db*, *name*, *x*, *y*, *w*, *h*, *attr*, *title*, *first*, *default*, *cancel*)

新しい *Dialog* オブジェクトを返します。`Dialog` テーブルを以下の引数に渡された情報を元に作成します: 座標、ダイアログ属性、タイトル、first・default・cancel という三つのコントロールに対する名前。

control(*name*, *type*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *control_next*, *help*)

新しい *Control* オブジェクトを返します。`Control` テーブルに指定されたパラメータのエントリが作られます。

これは汎用のメソッドで、特定の型に対しては特化したメソッドが提供されています。

text(*name*, *x*, *y*, *width*, *height*, *attributes*, *text*)

`Text` コントロールを追加して返します。

bitmap(*name*, *x*, *y*, *width*, *height*, *text*)

`Bitmap` コントロールを追加して返します。

line(*name*, *x*, *y*, *width*, *height*)
Line コントロールを追加して返します。

pushbutton(*name*, *x*, *y*, *width*, *height*, *attributes*, *text*, *next_control*)
PushButton コントロールを追加して返します。

radiogroup(*name*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *next_control*)
RadioButtonGroup コントロールを追加して返します。

checkbox(*name*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *next_control*)
CheckBox コントロールを追加して返します。

参考:

Dialog Table Control Table Control Types ControlCondition Table ControlEvent Table EventMapping Table RadioButton Table

34.1.10 事前に計算されたテーブル

msilib はスキーマとテーブル定義だけから成るサブパッケージをいくつか提供しています。現在のところ、これらの定義は MSI バージョン 2.0 に基づいています。

msilib.schema
これは MSI 2.0 用の標準 MSI スキーマで、テーブル定義のリストを提供する *tables* 変数と、MSI バリデーション用のデータを提供する *_Validation_records* 変数があります。

msilib.sequence
このモジュールは標準シーケンステーブルのテーブル内容を含んでいます。*AdminExecuteSequence*, *AdminUISequence*, *AdvtExecuteSequence*, *InstallExecuteSequence*, *InstallUISequence* が含まれています。

msilib.text
このモジュールは標準的なインストーラーのアクションのための UIText および ActionText テーブルの定義を含んでいます。

34.2 msvcrt --- MS VC++ 実行時システムの有用なルーチン群

このモジュールの関数は、Windows プラットフォームの便利な機能のいくつかに対するアクセス機構を提供しています。高レベルモジュールのいくつかは、提供するサービスを Windows で実装するために、これらの関数を使っています。例えば、*getpass* モジュールは関数 *getpass()* を実装するためにこのモジュールの関数を使います。

ここに挙げた関数の詳細なドキュメントについては、プラットフォーム API ドキュメントで見つけることができます。

このモジュールは、通常版とワイド文字列版の両方のコンソール I/O API を実装しています。通常版の API は ASCII 文字列のためのもので、国際化アプリケーションでは利用が制限されます。可能な限りワイド文字列版 API を利用するべきです。

バージョン 3.3 で変更: このモジュールの操作で以前は *IOError* が送出されていたところで *OSError* が送出されるようになりました。

34.2.1 ファイル操作関連

msvcrt.locking(*fd*, *mode*, *nbytes*)

C 言語による実行時システムにおけるファイル記述子 *fd* に基づいて、ファイルの一部にロックをかけます。失敗すると *OSError* が送出されます。ロックされるファイルの領域は、現在のファイル位置から *nbytes* バイトで、ファイルの末端まで延長することができます。*mode* は以下に列挙する LK_* のいずれか一つでなければなりません。一つのファイルの複数の領域を同時にロックすることは可能ですが、領域が重複してはなりません。連接する領域をまとめて指定することはできません; それらの領域は個別にロック解除しなければなりません。

引数 fd, mode, nbytes を指定して **監査イベント** msvcrt.locking を送出します。

msvcrt.LK_LOCK

msvcrt.LK_RLCK

指定されたバイト列にロックをかけます。指定領域がロックできなかった場合、プログラムは 1 秒後に再度ロックを試みます。10 回再試行した後でもロックをかけられない場合、*OSError* が送出されます。

msvcrt.LK_NBLCK

msvcrt.LK_NBRLCK

指定されたバイト列にロックをかけます。指定領域がロックできなかった場合、*OSError* が送出されます。

msvcrt.LK_UNLCK

指定されたバイト列のロックを解除します。指定領域はあらかじめロックされていなければなりません。

msvcrt.setmode(*fd*, *flags*)

ファイル記述子 *fd* に対して、行末文字の変換モードを設定します。テキストモードに設定するには、*flags* を *os.O_TEXT* にします; バイナリモードにするには *os.O_BINARY* にします。

msvcrt.open_osfhandle(*handle*, *flags*)

C 言語による実行時システムにおけるファイル記述子をファイルハンドル *handle* から生成します。*flags* パラメタは *os.O_APPEND* 、*os.O_RDONLY* 、および *os.O_TEXT* をビット単位で OR したものになります。返されるファイル記述子は *os.fdopen()* でファイルオブジェクトを生成するために使うことができます。

引数 handle, flags を指定して **監査イベント** msvcrt.open_osfhandle を送出します。

msvcrt.get_osfhandle(*fd*)

ファイル記述子 *fd* のファイルハンドルを返します。*fd* が認識できない場合、*OSError* を送出します。

引数 fd を指定して 監査イベント msvcrt.get_osfhandle を送出します。

34.2.2 コンソール I/O 関連

msvcrt.kbhit()
: 読み出し待ちの打鍵イベントが存在する場合に True を返します。

msvcrt.getch()
: 打鍵を読み取り、読み出された文字を返します。コンソールには何もエコーバックされません。この関数呼び出しは読み出し可能な打鍵がない場合にはブロックしますが、文字を読み出せるようにするために Enter の打鍵を待つ必要はありません。打鍵されたキーが特殊機能キー (function key) である場合、この関数は '\000' または '\xe0' を返します; キーコードは次に関数を呼び出した際に返されます。この関数で Control-C の打鍵を読み出すことはできません。

msvcrt.getwch()
: *getch()* のワイド文字列版。Unicode の値を返します。

msvcrt.getche()
: *getch()* に似ていますが、打鍵した字が印字可能な文字の場合エコーバックされます。

msvcrt.getwche()
: *getche()* のワイド文字列版。Unicode の値を返します。

msvcrt.putch(*char*)
: バイト文字列 *char* をバッファリングを行わないでコンソールに出力します。

msvcrt.putwch(*unicode_char*)
: *putch()* のワイド文字列版。Unicode の値を引数に取ります。

msvcrt.ungetch(*char*)
: バイト文字列 *char* をコンソールバッファに "押し戻し (push back)" ます; これにより、押し戻された文字は *getch()* や *getche()* で次に読み出される文字になります。

msvcrt.ungetwch(*unicode_char*)
: *ungetch()* のワイド文字列版。Unicode の値を引数に取ります。

34.2.3 その多の関数

msvcrt.heapmin()
: 強制的に malloc() ヒープをクリーンさせ、未使用のブロックをオペレーティングシステムに返させます。失敗した場合、*OSError* を送出します。

34.3 winreg --- Windows レジストリへのアクセス

これらの関数は Windows レジストリ API を Python から使えるようにします。プログラマがレジストリハンドルを明示的にクローズするのを忘れた場合でも、確実にハンドルがクローズされるようにするために、レジストリハンドルとして整数値ではなく **ハンドルオブジェクト** が使われます。

バージョン 3.3 で変更: このモジュールのいくつかの関数は以前は *WindowsError* を送出していました。それは今では *OSError* の別名です。

34.3.1 関数

このモジュールでは以下の関数を提供します:

winreg.CloseKey(*hkey*)
以前開かれたレジストリキーを閉じます。*hkey* 引数は以前開かれたレジストリキーを指定します。

> **注釈:** このメソッドを使って (または *hkey.Close()* によって) *hkey* が閉じられなかった場合、Python が *hkey* オブジェクトを破壊する際に閉じられます。

winreg.ConnectRegistry(*computer_name*, *key*)
他のコンピュータ上にある事前に定義されたレジストリハンドルとの接続を確立し、**ハンドルオブジェクト** を返します。

computer_name はリモートコンピュータの名前で、`r"\\computername"` の形式をとります。`None` の場合、ローカルのコンピュータが使われます。

key は、事前に定義された接続先のハンドルです。

戻り値は開かれたキーのハンドルです。関数が失敗した場合、*OSError* 例外が送出されます。

引数 `computer_name`, `key` を指定して **監査イベント** `winreg.ConnectRegistry` を送出します。

バージョン 3.3 で変更: **上記** を参照。

winreg.CreateKey(*key*, *sub_key*)
指定されたキーを生成するか開き、**ハンドルオブジェクト** を返します。

key はすでに開かれたキーか、既定の *HKEY_** **定数** のうちの一つです。

sub_key はこのメソッドが開く、または新規作成するキーの名前です。

key が事前に定義されたキーのうちの一つなら、*sub_key* は `None` でかまいません。その場合、この関数に渡されるキーハンドルと同じハンドルが返されます。

キーがすでに存在する場合、この関数はその既存のキーを開きます。

戻り値は開かれたキーのハンドルです。関数が失敗した場合、*OSError* 例外が送出されます。

引数 key, sub_key, access を指定して 監査イベント winreg.CreateKey を送出します。

引数 key を指定して 監査イベント winreg.OpenKey/result を送出します。

バージョン 3.3 で変更: 上記 を参照。

winreg.CreateKeyEx(*key*, *sub_key*, *reserved=0*, *access=KEY_WRITE*)

指定されたキーを生成するか開き、ハンドルオブジェクト を返します。

key はすでに開かれたキーか、既定の *HKEY_** 定数 のうちの一つです。

sub_key はこのメソッドが開く、または新規作成するキーの名前です。

reserved は予約された整数で、0 でなくてはなりません。デフォルト値は 0 です。

access は、key に対して想定されるセキュリティアクセスを示すアクセスマスクを指定する整数です。デフォルトは *KEY_WRITE* です。その他の利用可能な値については アクセス権 を参照してください。

key が事前に定義されたキーのうちの一つなら、*sub_key* は None でかまいません。その場合、この関数に渡されるキーハンドルと同じハンドルが返されます。

キーがすでに存在する場合、この関数はその既存のキーを開きます。

戻り値は開かれたキーのハンドルです。関数が失敗した場合、*OSError* 例外が送出されます。

引数 key, sub_key, access を指定して 監査イベント winreg.CreateKey を送出します。

引数 key を指定して 監査イベント winreg.OpenKey/result を送出します。

バージョン 3.2 で追加.

バージョン 3.3 で変更: 上記 を参照。

winreg.DeleteKey(*key*, *sub_key*)

指定されたキーを削除します。

key はすでに開かれたキーか、既定の *HKEY_** 定数 のうちの一つです。

sub_key は文字列で、*key* 引数によって指定されたキーのサブキーでなければなりません。この値は None であってはならず、キーはサブキーを持っていなくてもかまいません。

このメソッドはサブキーをもつキーを削除することはできません。

このメソッドの実行が成功すると、キー全体が、その値すべてを含めて削除されます。このメソッドが失敗した場合、*OSError* 例外が送出されます。

Raises an *auditing event* winreg.DeleteKey with arguments key, sub_key, access.

バージョン 3.3 で変更: 上記 を参照。

winreg.DeleteKeyEx(*key*, *sub_key*, *access=KEY_WOW64_64KEY*, *reserved=0*)

指定されたキーを削除します。

注釈: *DeleteKeyEx()* 関数は、Windows の 64-bit バージョンに特有の RegDeleteKeyEx Windows API 関数を使用して実装されています。RegDeleteKeyEx documentation を参照してください。

key はすでに開かれたキーか、既定の *HKEY_** 定数 のうちの一つです。

sub_key は *key* 引数によって指定された key の subkey でなければなりません。この値は None であってはなりません。また、key は subkey を持たないかもしれません。

reserved は予約された整数で、0 でなくてはなりません。デフォルト値は 0 です。

access は、key に対して想定されるセキュリティアクセスを示すアクセスマスクを指定する整数です。デフォルトは *KEY_WOW64_64KEY* です。その他の利用可能な値については **アクセス権** を参照してください。

このメソッドはサブキーをもつキーを削除することはできません。

このメソッドの実行が成功すると、キー全体が、その値すべてを含めて削除されます。このメソッドが失敗した場合、*OSError* 例外が送出されます。

サポートされていない Windows バージョンでは、*NotImplementedError* 例外を発生させます。

Raises an *auditing event* `winreg.DeleteKey` with arguments `key`, `sub_key`, `access`.

バージョン 3.2 で追加.

バージョン 3.3 で変更: **上記** を参照。

`winreg.DeleteValue`(*key*, *value*)

レジストリキーから指定された名前つきの値を削除します。

key はすでに開かれたキーか、既定の *HKEY_** 定数 のうちの一つです。

value は削除したい値を指定するための文字列です。

引数 `key`, `value` を指定して **監査イベント** `winreg.DeleteValue` を送出します。

`winreg.EnumKey`(*key*, *index*)

開かれているレジストリキーのサブキーを列挙し、文字列で返します。

key はすでに開かれたキーか、既定の *HKEY_** 定数 のうちの一つです。

index は整数値で、取得するキーのインデクスを指定します。

この関数は、呼び出されるたびに一つのサブキーの名前を取得します。この関数は通常、これ以上値がないことを示す *OSError* 例外が送出されるまで繰り返し呼び出されます。

引数 `key`, `index` を指定して **監査イベント** `winreg.EnumKey` を送出します。

バージョン 3.3 で変更: **上記** を参照。

`winreg.EnumValue`(*key*, *index*)

開かれているレジストリキーの値を列挙し、タプルで返します。

key はすでに開かれたキーか、既定の *HKEY_** 定数 のうちの一つです。

index は整数値で、取得する値のインデクスを指定します。

この関数は、呼び出されるたびに一つのサブキーの名前を取得します。この関数は通常、これ以上値がないことを示す *OSError* 例外が送出されるまで繰り返し呼び出されます。

結果は 3 要素のタプルになります:

インデックス	意味
0	値の名前を指定する文字列
1	値のデータを保持するためのオブジェクトで、その型は背後のレジストリ型に依存します
2	値のデータ型を指定する整数です (*SetValueEx()* のドキュメント内のテーブルを参照してください)

引数 key, index を指定して 監査イベント winreg.EnumValue を送出します。

バージョン 3.3 で変更: 上記 を参照。

winreg.**ExpandEnvironmentStrings**(*str*)

REG_EXPAND_SZ のように、環境変数プレースホルダ %NAME% を文字列で展開します:

```
>>> ExpandEnvironmentStrings('%windir%')
'C:\\Windows'
```

引数 str を指定して 監査イベント winreg.ExpandEnvironmentStrings を送出します。

winreg.**FlushKey**(*key*)

キーのすべての属性をレジストリに書き込みます。

key はすでに開かれたキーか、既定の *HKEY_** 定数 のうちの一つです。

キーを変更するために *FlushKey()* を呼ぶ必要はありません。レジストリの変更は怠惰なフラッシュ機構 (lazy flusher) を使ってフラッシュされます。また、システムの遮断時にもディスクにフラッシュされます。*CloseKey()* と違って、*FlushKey()* メソッドはレジストリに全てのデータを書き終えたときにのみ返ります。アプリケーションは、レジストリへの変更を絶対に確実にディスク上に反映させる必要がある場合にのみ、*FlushKey()* を呼ぶべきです。

注釈: *FlushKey()* を呼び出す必要があるかどうか分からない場合、おそらくその必要はありません。

---

winreg.**LoadKey**(*key*, *sub_key*, *file_name*)

指定されたキーの下にサブキーを生成し、サブキーに指定されたファイルのレジストリ情報を記録します。

key は *ConnectRegistry()* が返したハンドルか、定数 *HKEY_USERS* と *HKEY_LOCAL_MACHINE* のどちらかです。

sub_key は記録先のサブキーを指定する文字列です。

file_name はレジストリデータを読み出すためのファイル名です。このファイルは *SaveKey()* 関数で生成されたファイルでなくてはなりません。ファイル割り当てテーブル (FAT) ファイルシステム下では、ファイル名は拡張子を持っていてはなりません。

この関数を呼び出しているプロセスが SE_RESTORE_PRIVILEGE 特権を持たない場合には *LoadKey()* の呼び出しは失敗します。この特権とは、許可とは違うので注意してください。詳細は RegLoadKey documentation を参照してください。

key が *ConnectRegistry()* によって返されたハンドルの場合、*file_name* に指定されたパスはリモートコンピュータに対する相対パス名になります。

引数 key, sub_key, file_name を指定して **監査イベント** winreg.LoadKey を送出します。

winreg.OpenKey(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)
winreg.OpenKeyEx(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)

指定されたキーを開き、**ハンドルオブジェクト** を返します。

key はすでに開かれたキーか、既定の *HKEY_** **定数** のうちの一つです。

sub_key は開くサブキーを指定する文字列です。

reserved は予約された整数で、0 でなくてはなりません。デフォルト値は 0 です。

access は、key に対して想定されるセキュリティアクセスを示すアクセスマスクを指定する整数です。デフォルトは *KEY_READ* です。その他の利用可能な値については **アクセス権** を参照してください。

指定されたキーへの新しいハンドルが返されます。

この関数が失敗すると、*OSError* が送出されます。

引数 key, sub_key, access を指定して **監査イベント** winreg.OpenKey を送出します。

引数 key を指定して **監査イベント** winreg.OpenKey/result を送出します。

バージョン 3.2 で変更: 名前付き引数が使用できるようになりました。

バージョン 3.3 で変更: **上記** を参照。

winreg.QueryInfoKey(*key*)

キーに関する情報をタプルとして返します。

key はすでに開かれたキーか、既定の *HKEY_** **定数** のうちの一つです。

結果は 3 要素のタプルになります:

インデックス	意味
0	このキーが持つサブキーの数を表す整数。
1	このキーが持つ値の数を表す整数。
2	最後のキーの変更が (あれば) いつだったかを表す整数で、1601 年 1 月 1 日からの 100 ナノ秒単位で数えたもの。

引数 key を指定して **監査イベント** winreg.QueryInfoKey を送出します。

winreg.**QueryValue**(*key*, *sub_key*)

キーに対する、名前付けられていない値を文字列で取得します。

key はすでに開かれたキーか、既定の *HKEY_** **定数** のうちの一つです。

sub_key は値が関連付けられているサブキーの名前を保持する文字列です。この引数が None または空文字列の場合、この関数は *key* で指定されたキーに対して *SetValue()* メソッドで設定された値を取得します。

レジストリ中の値は名前、型、およびデータから構成されています。このメソッドはあるキーのデータ中で、名前 NULL をもつ最初の値を取得します。しかし背後の API 呼び出しは型情報を返しません。なので、可能ならいつでも *QueryValueEx()* を使うべきです。

引数 key, sub_key, value_name を指定して **監査イベント** winreg.QueryValue を送出します。

winreg.**QueryValueEx**(*key*, *value_name*)

開かれたレジストリキーに関連付けられている、指定した名前の値に対して、型およびデータを取得します。

key はすでに開かれたキーか、既定の *HKEY_** **定数** のうちの一つです。

value_name は要求する値を指定する文字列です。

結果は 2 つの要素からなるタプルです:

インデックス	意味
0	レジストリ要素の値。
1	この値のレジストリ型を表す整数。(*SetValueEx()* のドキュメント内のテーブルを参照してください。)

引数 key, sub_key, value_name を指定して **監査イベント** winreg.QueryValue を送出します。

winreg.**SaveKey**(*key*, *file_name*)

指定されたキーと、そのサブキー全てを指定したファイルに保存します。

key はすでに開かれたキーか、既定の *HKEY_** **定数** のうちの一つです。

file_name はレジストリデータを保存するファイルの名前です、このファイルはすでに存在していてはいけません。このファイル名が拡張子を含んでいる場合、*LoadKey()* メソッドは、FAT ファイルシステムを使うことができません。

key がリモートコンピュータ上にあるキーを表す場合、*file_name* で記述されているパスはリモートコンピュータに対して相対的なパスになります。このメソッドの呼び出し側は SeBackupPrivilege セキュリティ特権を保有していなければなりません。この特権とは、パーミッションとは異なるものです。詳細は Conflicts Between User Rights and Permissions documentation を参照してください。

この関数は *security_attributes* を NULL にして API に渡します。

引数 key, file_name を指定して **監査イベント** winreg.SaveKey を送出します。

winreg.SetValue(*key*, *sub_key*, *type*, *value*)

値を指定したキーに関連付けます。

key はすでに開かれたキーか、既定の *HKEY_** **定数** のうちの一つです。

sub_key は値が関連付けられているサブキーの名前を表す文字列です。

type はデータの型を指定する整数です。現状では、この値は *REG_SZ* でなければならず、これは文字列だけがサポートされていることを示します。他のデータ型をサポートするには *SetValueEx()* を使ってください。

value は新たな値を指定する文字列です。

sub_key 引数で指定されたキーが存在しなければ、SetValue 関数で生成されます。

値の長さは利用可能なメモリによって制限されます。(2048 バイト以上の) 長い値はファイルに保存して、そのファイル名を設定レジストリに保存するべきです。そうすればレジストリを効率的に動作させる役に立ちます。

key 引数に指定されたキーは *KEY_SET_VALUE* アクセスで開かれていなければなりません。

引数 key, sub_key, type, value を指定して **監査イベント** winreg.SetValue を送出します。

winreg.SetValueEx(*key*, *value_name*, *reserved*, *type*, *value*)

開かれたレジストリキーの値フィールドにデータを記録します。

key はすでに開かれたキーか、既定の *HKEY_** **定数** のうちの一つです。

value_name は値が関連付けられているサブキーの名前を表す文字列です。

reserved は何もしません - API には常にゼロが渡されます。

type はデータの型を指定する整数です。利用できる型については **値の型** を参照してください。

value は新たな値を指定する文字列です。

このメソッドではまた、指定されたキーに対して、さらに別の値や型情報を設定することができます。*key* 引数で指定されたキーは *KEY_SET_VALUE* アクセスで開かれていなければなりません。

キーを開くには、*CreateKey()* または *OpenKey()* メソッドを使ってください。

値の長さは利用可能なメモリによって制限されます。(2048 バイト以上の) 長い値はファイルに保存して、そのファイル名を設定レジストリに保存するべきです。そうすればレジストリを効率的に動作させる役に立ちます。

引数 key, sub_key, type, value を指定して **監査イベント** winreg.SetValue を送出します。

winreg.DisableReflectionKey(*key*)

64 ビット OS 上で動作している 32bit プロセスに対するレジストリリフレクションを無効にします。

key はすでに開かれたキーか、既定の *HKEY_** **定数** のうちの一つです。

32bit OS 上では一般的に *NotImplementedError* 例外を発生させます。

key がリフレクションリストに無い場合は、この関数は成功しますが効果はありません。あるキーのリフレクションを無効にしても、そのキーのサブキーのリフレクションには全く影響しません。

引数 key を指定して **監査イベント** winreg.DisableReflectionKey を送出します。

winreg.EnableReflectionKey(*key*)

指定された、リフレクションが無効にされたキーのリフレクションを再び有効にします。

key はすでに開かれたキーか、既定の *HKEY_** **定数** のうちの一つです。

32bit OS 上では一般的に *NotImplementedError* 例外を発生させます。

あるキーのリフレクションを再開しても、その全てのサブキーには影響しません。

引数 key を指定して **監査イベント** winreg.EnableReflectionKey を送出します。

winreg.QueryReflectionKey(*key*)

指定されたキーのリフレクション状態を確認します。

key はすでに開かれたキーか、既定の *HKEY_** **定数** のうちの一つです。

リフレクションが無効になっている場合、True を返します。

32bit OS 上では一般的に *NotImplementedError* 例外を発生させます。

引数 key を指定して **監査イベント** winreg.QueryReflectionKey を送出します。

34.3.2 定数

_winreg の多くの関数で利用するために以下の定数が定義されています。

HKEY_* 定数

winreg.HKEY_CLASSES_ROOT
: このキー以下のレジストリエントリは、ドキュメントのタイプ（またはクラス）や、それに関連付けられたプロパティを定義しています。シェルと COM アプリケーションがこの情報を利用します。

winreg.HKEY_CURRENT_USER
: このキー以下のレジストリエントリは、現在のユーザーの設定を定義します。この設定には、環境変数、プログラムグループに関するデータ、カラー、プリンター、ネットワーク接続、アプリケーション設定などが含まれます。

winreg.HKEY_LOCAL_MACHINE
: このキー以下のレジストリエントリは、コンピュータの物理的な状態を定義します。これには、バスタイプ、システムメモリ、インストールされているソフトウェアやハードウェアが含まれます。

winreg.HKEY_USERS
: このキー以下のレジストリエントリは、ローカルコンピュータの新規ユーザーのためのデフォルト設定や、現在のユーザーの設定を定義しています。

winreg.HKEY_PERFORMANCE_DATA
: このキー以下のレジストリエントリは、パフォーマンスデータへのアクセスを可能にしています。実際にはデータはレジストリには格納されていません。レジストリ関数がシステムにソースからデータを集めさせます。

winreg.HKEY_CURRENT_CONFIG
: ローカルコンピュータシステムの現在のハードウェアプロファイルに関する情報を含みます。

winreg.HKEY_DYN_DATA
: このキーは Windows の 98 以降のバージョンでは利用されていません。

アクセス権限

より詳しい情報については、Registry Key Security and Access を参照してください。

winreg.KEY_ALL_ACCESS
: STANDARD_RIGHTS_REQUIRED (*KEY_QUERY_VALUE*, *KEY_SET_VALUE*, *KEY_CREATE_SUB_KEY*, *KEY_ENUMERATE_SUB_KEYS*, *KEY_NOTIFY*, *KEY_CREATE_LINK*) アクセス権限の組み合わせ。

winreg.KEY_WRITE
: STANDARD_RIGHTS_WRITE (*KEY_SET_VALUE*, *KEY_CREATE_SUB_KEY*) アクセス権限の組み合わせ。

winreg.KEY_READ
: STANDARD_RIGHTS_READ (*KEY_QUERY_VALUE*, *KEY_ENUMERATE_SUB_KEYS*, *KEY_NOTIFY*) アクセス権限の組み合わせ。

winreg.KEY_EXECUTE
: *KEY_READ* と同じ。

winreg.KEY_QUERY_VALUE
: レジストリキーの値を問い合わせるのに必要。

winreg.KEY_SET_VALUE
: レジストリの値を作成、削除、設定するのに必要。

winreg.KEY_CREATE_SUB_KEY
: レジストリキーのサブキーを作るのに必要。

winreg.KEY_ENUMERATE_SUB_KEYS
: レジストリキーのサブキーを列挙するのに必要。

winreg.KEY_NOTIFY
: レジストリキーやそのサブキーに対する変更通知を要求するのに必要。

winreg.KEY_CREATE_LINK
: システムでの利用のために予約されている。

64-bit 特有のアクセス権

より詳しい情報については、Accessing an Alternate Registry View を参照してください。

winreg.KEY_WOW64_64KEY
: 64 bit Windows 上のアプリケーションが、64 bit のレジストリビュー上で操作する事を示します。

winreg.KEY_WOW64_32KEY
: 64 bit Windows 上のアプリケーションが、32 bit のレジストリビュー上で操作する事を示します。

値の型

より詳しい情報については、Registry Value Types を参照してください。

winreg.REG_BINARY
: 何らかの形式のバイナリデータ。

winreg.REG_DWORD
: 32 ビットの数。

winreg.REG_DWORD_LITTLE_ENDIAN
: 32 ビットのリトルエンディアン形式の数。*REG_DWORD* と等価。

winreg.REG_DWORD_BIG_ENDIAN
: 32 ビットのビッグエンディアン形式の数。

winreg.REG_EXPAND_SZ
: 環境変数を参照している、ヌル文字で終端された文字列。(%PATH%)。

winreg.REG_LINK
: Unicode のシンボリックリンク。

winreg.REG_MULTI_SZ
: ヌル文字で終端された文字列からなり、二つのヌル文字で終端されている配列。(Python はこの終端の処理を自動的に行います。)

winreg.REG_NONE
: 定義されていない値の形式。

winreg.REG_QWORD
: 64 ビットの数。

  バージョン 3.6 で追加.

winreg.REG_QWORD_LITTLE_ENDIAN
: 64 ビットのリトルエンディアン形式の数。*REG_QWORD* と等価。

  バージョン 3.6 で追加.

winreg.REG_RESOURCE_LIST
: デバイスドライバリソースのリスト。

winreg.REG_FULL_RESOURCE_DESCRIPTOR
: ハードウェアセッティング。

winreg.REG_RESOURCE_REQUIREMENTS_LIST
: ハードウェアリソースリスト。

winreg.REG_SZ
: ヌル文字で終端された文字列。

34.3.3 レジストリハンドルオブジェクト

このオブジェクトは Windows の HKEY オブジェクトをラップし、オブジェクトが破壊されたときに自動的にハンドルを閉じます。オブジェクトの *Close()* メソッドと *CloseKey()* 関数のどちらも、後始末がきちんと行われることを保証するために呼び出すことができます。

このモジュールのレジストリ関数は全て、これらのハンドルオブジェクトの一つを返します。

このモジュールのレジストリ関数でハンドルオブジェクトを受理するものは全て整数も受理しますが、ハンドルオブジェクトを利用することを推奨します。

ハンドルオブジェクトは `__bool__()` の意味構成を持ちます - すなわち

```
if handle:
    print("Yes")
```

は、ハンドルが現在有効な (閉じられたり切り離されたりしていない) 場合には `Yes` となります。

ハンドルオブジェクトは、比較の意味構成もサポートしています。このため、複数のハンドルオブジェクトが参照している下層の Windows ハンドル値が同じ場合、それらのハンドルオブジェクト同士の比較は真になります。

ハンドルオブジェクトは (例えば組み込みの *int()* 関数を使って) 整数に変換することができます。この場合、背後の Windows ハンドル値が返されます、また、*Detach()* メソッドを使って整数のハンドル値を返させると同時に、ハンドルオブジェクトから Windows ハンドルを切り離すこともできます。

PyHKEY.Close()
背後の Windows ハンドルを閉じます。

ハンドルがすでに閉じられていてもエラーは送出されません。

PyHKEY.Detach()
ハンドルオブジェクトから Windows ハンドルを切り離します。

切り離される以前にそのハンドルを保持していた整数オブジェクトが返されます。ハンドルがすでに切り離されていたり閉じられていたりした場合、ゼロが返されます。

この関数を呼び出した後、ハンドルは確実に無効化されますが、閉じられるわけではありません。背後の Win32 ハンドルがハンドルオブジェクトよりも長く維持される必要がある場合にはこの関数を呼び出すとよいでしょう。

Raises an *auditing event* `winreg.PyHKEY.Detach` with argument `key`.

PyHKEY.__enter__()
PyHKEY.__exit__(*exc_info*)
HKEY オブジェクトは `__enter__()`, `__exit__()` メソッドを実装していて、`with` 文のためのコンテキストプロトコルをサポートしています:

```
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ...  # work with key
```

このコードは、`with` ブロックから抜けるときに自動的に *key* を閉じます。

34.4 winsound --- Windows 用の音声再生インタフェース

winsound モジュールは Windows プラットフォーム上で提供されている基本的な音声再生機構へのアクセス手段を提供します。このモジュールではいくつかの関数と定数が定義されています。

winsound.Beep(*frequency*, *duration*)
PC のスピーカを鳴らします。引数 *frequency* は鳴らす音の周波数の指定で、単位は Hz です。値は 37 から 32,767 でなくてはなりません。引数 *duration* は音を何ミリ秒鳴らすかの指定です。システムがスピーカを鳴らすことができない場合、例外 *RuntimeError* が送出されます。

winsound.PlaySound(*sound*, *flags*)
プラットフォームの API から関数 `PlaySound()` を呼び出します。引数 *sound* はファイル名、システム音エイリアス、音声データの *bytes-like* **オブジェクト**、または `None` をとり得ます。*sound* の解釈は *flags* の値に依存します。この値は以下に述べる定数をビット単位 OR して組み合わせたものになります。*sound* 引数が `None` だった場合、現在再生中の Wave 形式サウンドの再生を停止します。システムのエラーが発生した場合、例外 *RuntimeError* が送出されます。

winsound.MessageBeep(*type=MB_OK*)

根底にある MessageBeep() 関数をプラットフォームの API から呼び出します。この関数は音声をレジストリの指定に従って再生します。*type* 引数はどの音声を再生するかを指定します; とり得る値は -1, MB_ICONASTERISK, MB_ICONEXCLAMATION, MB_ICONHAND, MB_ICONQUESTION, および MB_OK で、全て以下に記述されています。値 -1 は "単純なビープ音" を再生します; この値は他の場合で音声を再生することができなかった際の最終的な代替音です。システムがエラーを示したら、*RuntimeError* が送出されます。

winsound.SND_FILENAME

sound パラメタが WAV ファイル名であることを示します。*SND_ALIAS* と同時に使ってはいけません。

winsound.SND_ALIAS

引数 *sound* はレジストリにある音声データに関連付けられた名前であることを示します。指定した名前がレジストリ上にない場合、定数 *SND_NODEFAULT* が同時に指定されていない限り、システム標準の音声データが再生されます。標準の音声データが登録されていない場合、例外 *RuntimeError* が送出されます。*SND_FILENAME* と同時に使ってはいけません。

全ての Win32 システムは少なくとも以下の名前をサポートします; ほとんどのシステムでは他に多数あります:

PlaySound() name	対応するコントロールパネルでの音声名
'SystemAsterisk'	Asterisk
'SystemExclamation'	Exclamation
'SystemExit'	Exit Windows
'SystemHand'	Critical Stop
'SystemQuestion'	Question

例えば:

```
import winsound
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered (because
# "*" probably isn't the registered name of any sound).
winsound.PlaySound("*", winsound.SND_ALIAS)
```

winsound.SND_LOOP

音声データを繰り返し再生します。システムがブロックしないようにするため、*SND_ASYNC* フラグを同時に使わなくてはなりません。*SND_MEMORY* と同時に使うことはできません。

winsound.SND_MEMORY

PlaySound() の引数 *sound* が *bytes-like* **オブジェクト** の形式をとった WAV ファイルのメモリ上のイメージであることを示します。

注釈: このモジュールはメモリ上のイメージを非同期に再生する機能をサポートしていません。従っ

て、このフラグと *SND_ASYNC* を組み合わせると例外 *RuntimeError* が送出されます。

winsound.SND_PURGE
指定した音声の全てのインスタンスについて再生処理を停止します。

注釈: このフラグは現代の Windows プラットフォームではサポートされていません。

winsound.SND_ASYNC
音声を非同期に再生するようにして、関数呼び出しを即座に返します。

winsound.SND_NODEFAULT
指定した音声が見つからなかった場合にシステム標準の音声を鳴らさないようにします。

winsound.SND_NOSTOP
現在鳴っている音声を中断させないようにします。

winsound.SND_NOWAIT
サウンドドライバがビジー状態にある場合、関数がすぐ返るようにします。

注釈: このフラグは現代の Windows プラットフォームではサポートされていません。

winsound.MB_ICONASTERISK
音声 SystemDefault を再生します。

winsound.MB_ICONEXCLAMATION
音声 SystemExclamation を再生します。

winsound.MB_ICONHAND
音声 SystemHand を再生します。

winsound.MB_ICONQUESTION
音声 SystemQuestion を再生します。

winsound.MB_OK
音声 SystemDefault を再生します。

第

THIRTYFIVE

UNIX 固有のサービス

本章に記述されたモジュールは、Unix オペレーティングシステム、あるいはそれから派生した多くのものに固有の機能のためのインタフェースを提供します。その概要を以下に述べます:

35.1 posix --- 最も一般的な POSIX システムコール群

このモジュールはオペレーティングシステムの機能のうち、C 言語標準および (Unix インタフェースをほんの少し隠蔽した) POSIX 標準で標準化されている機能に対するアクセス機構を提供します。

このモジュールを直接インポートしてはいけません。その代わりに、このインターフェースの **ポータブル** 版である *os* モジュールをインポートしてください。Unix では、*os* モジュールは *posix* インターフェースのスーパーセットを提供しています。非 Unix オペレーティングシステムでは、*posix* モジュールは利用できませんが、その一部分は *os* インターフェースを通して常に利用可能です。一度 *os* をインポートし *posix* の代わりにそれを使えば、パフォーマンス上の代償は **ありません** 。加えて *os* は、os.environ の要素が変更されたときに *putenv()* を呼び出すなどの追加の機能も提供します。

エラーは例外として報告されます; よくある例外は型エラーです。一方、システムコールから報告されたエラーは以下に述べるように *OSError* を送出します。

35.1.1 ラージファイルのサポート

いくつかのオペレーティングシステム (AIX, HP-UX, Irix および Solaris が含まれます) は、int および long を 32 ビット値とする C プログラムモデルで 2GB を超えるサイズのファイルのサポートを提供しています。このサポートは典型的には関連するサイズとオフセットの組合せを 64-bit 値として定義することで実現しています。このようなファイルは時にラージファイル (*large files*) と呼ばれます。

Python では、off_t のサイズが long より大きく、かつ long long が少なくとも off_t と同じくらい大きなサイズであるとき、ラージファイルのサポートが有効になります。この場合、ファイルのサイズ、オフセットおよび Python の通常整数型の範囲を超えるような値の表現には Python の長整数型が使われます。このモードを有効にするのに、**configure** で Python のコンパイルに特定のコンパイルフラグを必要とするかもしれません。例えば、ラージファイルのサポートは Irix の最近のバージョンでは標準で有効ですが、Solaris 2.6 および 2.7 では、以下のようにする必要があります:

```
CFLAGS="`getconf LFS_CFLAGS`" OPT="-g -O2 $CFLAGS" \
        ./configure
```

ラージファイル対応の Linux システムでは、以下のようにすれば良いでしょう:

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT="-g -O2 $CFLAGS" \
        ./configure
```

35.1.2 注目すべきモジュールの内容

os モジュールのドキュメントで説明されている多数の関数に加え、*posix* では以下のデータ項目を定義しています:

posix.**environ**

インタプリタが起動したときの環境を表す文字列辞書です。キーと値は Unix ではバイト列、Windows では文字列です。例えば、`environ[b'HOME']` (Windows では `environ['HOME']`) はホームディレクトリのパス名で、C の `getenv("HOME")` と同じ値です。

この辞書を編集しても、*execv()* 、*popen()* 、*system()* で渡された環境変数文字列には影響は与えません; 環境変数を変更したい場合は、`environ` を *execve()* に渡すか、変数への代入文と export 文を *system()* や *popen()* に渡すコマンド文字列に追加してください。

バージョン 3.2 で変更: Unix ではキーと値はバイト列になりました。

注釈: *os* モジュールでは、もう一つの *os.environ* 実装を提供しており、環境変数が変更された場合、その内容を更新するようになっています。`environ` を更新した場合、この辞書は古い内容を表していることになってしまうので、このことにも注意してください。*posix* モジュール版を直接アクセスするよりも、*os* モジュール版を使う方が推奨されています。

35.2 pwd --- パスワードデータベースへのアクセスを提供する

このモジュールは Unix のユーザアカウントとパスワードのデータベースへのアクセスを提供します。全ての Unix 系 OS で利用できます。

パスワードデータベースの各エントリはタプルのようなオブジェクトで提供され、それぞれの属性は `passwd` 構造体のメンバに対応しています (下の属性欄については、`<pwd.h>` を見てください):

インデックス	属性	意味
0	pw_name	ログイン名
1	pw_passwd	暗号化されたパスワード (optional))
2	pw_uid	ユーザ ID(UID)
3	pw_gid	グループ ID(GID)
4	pw_gecos	実名またはコメント
5	pw_dir	ホームディレクトリ
6	pw_shell	シェル

UID と GID は整数で、それ以外は全て文字列です。検索したエントリが見つからないと *KeyError* が発生します。

注釈: 伝統的な Unix では、pw_passwd フィールドは DES 由来のアルゴリズムで暗号化されたパスワード (*crypt* モジュールをごらんください) が含まれています。しかし、近代的な UNIX 系 OS では **シャドウパスワード** とよばれる仕組みを利用しています。この場合には *pw_passwd* フィールドにはアスタリスク ('*') か、'x' という一文字だけが含まれており、暗号化されたパスワードは、一般には見えない /etc/shadow というファイルに入っています。*pw_passwd* フィールドに有用な値が入っているかはシステムに依存します。利用可能なら、暗号化されたパスワードへのアクセスが必要なときには *spwd* モジュールを利用してください。

このモジュールでは以下の内容を定義しています:

pwd.**getpwuid**(*uid*)
: 与えられた UID に対応するパスワードデータベースのエントリを返します。

pwd.**getpwnam**(*name*)
: 与えられたユーザ名に対応するパスワードデータベースのエントリを返します。

pwd.**getpwall**()
: パスワードデータベースの全てのエントリを、任意の順番で並べたリストを返します。

参考:

grp **モジュール** このモジュールに似た、グループデータベースへのアクセスを提供するモジュール。

spwd **モジュール** このモジュールと類似の、シャドウパスワードデータベースへのインタフェース。

35.3 spwd --- シャドウパスワードデータベース

このモジュールは Unix のシャドウパスワードデータベースへのアクセスを提供します。様々な Unix 環境で利用できます。

シャドウパスワードデータベースへアクセスできる権限が必要 (大抵の場合 root である必要があります) です。

シャドウパスワードデータベースのエントリはタプル状のオブジェクトで提供され、その属性は spwd 構造のメンバーに対応しています（以下を参照してください。<shadow.h> を参照):

インデックス	属性	意味
0	sp_namp	ログイン名
1	sp_pwdp	暗号化されたパスワード
2	sp_lstchg	最終更新日
3	sp_min	パスワード変更が出来るようになるまでの最小日数
4	sp_max	パスワードを変更しなくても良い最大日数
5	sp_warn	パスワードが期限切れになる前に、期限切れが近づいている旨の警告をユーザに出しはじめる日数
6	sp_inact	パスワードが期限切れになってからアカウントが無効になるまでの日数
7	sp_expire	アカウントが期限切れになる日の 1970-01-01 からの日数
8	sp_flag	将来のために予約

sp_namp と *sp_pwdp* は文字列で、他は全て整数です。エントリが見つからなかった時は *KeyError* が起きます。

以下の関数が定義されています:

spwd.getspnam(*name*)
: 与えられたユーザ名に対応するシャドーパスワードデータベースのエントリを返します。

  バージョン 3.6 で変更: ユーザが権限を持っていない場合、*KeyError* の代わりに *PermissionError* を送出します。

spwd.getspall()
: このモジュールでは以下を定義しています。

参考:

grp **モジュール** このモジュールに似た、グループデータベースへのアクセスを提供するモジュール。

pwd **モジュール** このモジュールに似た通常のパスワードデータベースへのインタフェース。

35.4 grp --- グループデータベースへのアクセス

このモジュールでは Unix グループ (group) データベースへのアクセス機構を提供します。全ての Unix バージョンで利用可能です。

このモジュールはグループデータベースのエントリをタプルに似たオブジェクトとして報告されます。このオブジェクトの属性は group 構造体の各メンバ (以下の属性フィールド、<pwd.h> を参照) に対応します:

インデックス	属性	意味
0	gr_name	グループ名
1	gr_passwd	(暗号化された) グループパスワード; しばしば空文字列になります
2	gr_gid	数字のグループ ID
3	gr_mem	グループメンバの全てのユーザ名

gid は整数、名前およびパスワードは文字列、そしてメンバリストは文字列からなるリストです。(ほとんどのユーザは、パスワードデータベースで自分が入れられているグループのメンバとしてグループデータベース内では明示的に列挙されていないので注意してください。完全なメンバ情報を取得するには両方のデータベースを調べてください。また、+ や - で始まる gr_name は YP/NIS 参照である可能性があり、*getgrnam()* や *getgrgid()* でアクセスできないかもしれないことにも注意してください。)

このモジュールでは以下の内容を定義しています:

grp.**getgrgid**(*gid*)
: 与えられたグループ ID に対するグループデータベースエントリを返します。要求したエントリが見つからなかった場合、*KeyError* が送出されます。

  バージョン 3.6 で非推奨: Python 3.6 から、浮動小数点数や文字列のような非整数値の引数のサポートが撤廃されました。

grp.**getgrnam**(*name*)
: 与えられたグループ名に対するグループデータベースエントリを返します。要求したエントリが見つからなかった場合、*KeyError* が送出されます。

grp.**getgrall**()
: 全ての入手可能なグループエントリを返します。順番は決まっていません。

参考:

pwd **モジュール**　このモジュールと類似の、ユーザデータベースへのインタフェース。

spwd **モジュール**　このモジュールと類似の、シャドウパスワードデータベースへのインタフェース。

35.5 crypt --- Unix パスワードをチェックするための関数

ソースコード: Lib/crypt.py

このモジュールは修正 DES アルゴリズムに基づいた一方向ハッシュ関数である *crypt(3)* ルーチンを実装しています。詳細については Unix マニュアルページを参照してください。このモジュールは、実際に入力されたパスワードを記録することなくチェック出来るようにするためのハッシュ化パスワードを記録したり、Unix パスワードに (脆弱性検査のための) 辞書攻撃を試みるのに使えます。

このモジュールは実行環境の *crypt(3)* の実装に依存しています。そのため、現在の実装で利用可能な拡張を、このモジュールでもそのまま利用できます。

利用可能環境: Unix。VxWorks では使えません。

35.5.1 ハッシュ化方式

バージョン 3.3 で追加.

crypt モジュールはハッシュ化方式の一覧を定義しています (すべての方式がすべてのプラットフォームで使えるわけではありません):

crypt.METHOD_SHA512
: A Modular Crypt Format method with 16 character salt and 86 character hash based on the SHA-512 hash function. This is the strongest method.

crypt.METHOD_SHA256
: Another Modular Crypt Format method with 16 character salt and 43 character hash based on the SHA-256 hash function.

crypt.METHOD_BLOWFISH
: Another Modular Crypt Format method with 22 character salt and 31 character hash based on the Blowfish cipher.

  バージョン 3.7 で追加.

crypt.METHOD_MD5
: Another Modular Crypt Format method with 8 character salt and 22 character hash based on the MD5 hash function.

crypt.METHOD_CRYPT
: 2 文字のソルトと 13 文字のハッシュ値を持つモジュラー暗号形式です。これが最も弱い方式です。

35.5.2 モジュール属性

バージョン 3.3 で追加.

crypt.methods
: 利用可能なパスワードのハッシュアルゴリズムのリストを、crypt.METHOD_* オブジェクトとして返します。このリストは最も強いものから弱いものの順で並べられています。

35.5.3 モジュール関数

crypt モジュールは以下の関数を定義しています:

crypt.crypt(*word*, *salt=None*)
: *word* will usually be a user's password as typed at a prompt or in a graphical interface. The optional *salt* is either a string as returned from *mksalt()*, one of the crypt.METHOD_* values (though not all may be available on all platforms), or a full encrypted password including salt, as returned by this function. If *salt* is not provided, the strongest method will be used (as returned by *methods()*).

  通常は、生の文字列のパスワードを *word* として渡し、前回の *crypt()* を呼び出した結果と今回の呼び出しの結果が同じになることで、パスワードの確認を行います。

  salt (2 文字から 16 文字のランダムな文字列で、方式を示す $digit$ が先頭に付いているかもしれません) は、暗号化アルゴリズムにぶれを生じさせるために使われます。*salt* に含まれる文字は、モジュラー暗号形式の先頭にある $digit$ を除いて、集合 [./a-zA-Z0-9] に含まれていなければいけません。

  ハッシュ化されたパスワードを文字列として返します。それは salt と同じアルファベット文字から構成されます。

  いくつかの拡張された *crypt(3)* は異なる値と *salt* の長さを許しているので、パスワードをチェックする際には crypt されたパスワード文字列全体を *salt* として渡すよう勧めます。

  バージョン 3.3 で変更: 文字列に加え、*salt* が crypt.METHOD_* 値も受け取るようになりました。

crypt.mksalt(*method=None*, ***, *rounds=None*)
: 指定された方式のランダムに生成されたソルトを返します。*method* が与えられなかった場合は、*methods()* で返される方式のうち最も強いものが使われます。

  The return value is a string suitable for passing as the *salt* argument to *crypt()*.

  rounds specifies the number of rounds for METHOD_SHA256, METHOD_SHA512 and METHOD_BLOWFISH. For METHOD_SHA256 and METHOD_SHA512 it must be an integer between 1000 and 999_999_999, the default is 5000. For METHOD_BLOWFISH it must be a power of two between 16 (2^4) and 2_147_483_648 (2^{31}), the default is 4096 (2^{12}).

  バージョン 3.3 で追加.

  バージョン 3.7 で変更: Added the *rounds* parameter.

35.5.4 使用例

典型的な使い方を簡単な例で示します (タイミング攻撃に晒されないように、一定時間の比較演算子を使う必要があり、*hmac.compare_digest()* がこの目的にちょうど良いです):

```
import pwd
import crypt
import getpass
from hmac import compare_digest as compare_hash

def login():
    username = input('Python login: ')
    cryptedpasswd = pwd.getpwnam(username)[1]
    if cryptedpasswd:
        if cryptedpasswd == 'x' or cryptedpasswd == '*':
            raise ValueError('no support for shadow passwords')
        cleartext = getpass.getpass()
        return compare_hash(crypt.crypt(cleartext, cryptedpasswd), cryptedpasswd)
    else:
        return True
```

利用可能な方式のうち最も強い方式を使いパスワードのハッシュ値を生成し、元のパスワードと比較してチェックします:

```
import crypt
from hmac import compare_digest as compare_hash

hashed = crypt.crypt(plaintext)
if not compare_hash(hashed, crypt.crypt(plaintext, hashed)):
    raise ValueError("hashed version doesn't validate against original")
```

35.6 termios --- POSIX スタイルの端末制御

このモジュールでは端末 I/O 制御のための POSIX 準拠の関数呼び出しインタフェースを提供します。これら呼び出しのための完全な記述については、Unix マニュアルページの *termios(3)* を参照してください。これは POSIX *termios* 形式の端末制御をサポートしていてインストール時に有効にした Unix のバージョンでのみ利用可能です。

このモジュールの関数は全て、ファイル記述子 *fd* を最初の引数としてとります。この値は、sys.stdin.fileno() が返すような整数のファイル記述子でも、sys.stdin 自体のような *file object* でもかまいません。

このモジュールではまた、モジュールで提供されている関数を使う上で必要となる全ての定数を定義しています; これらの定数は C の対応する関数と同じ名前を持っています。これらの端末制御インタフェースを利用する上でのさらなる情報については、あなたのシステムのドキュメンテーションを参考にしてください。

このモジュールには、以下の関数が定義されています:

termios.tcgetattr(*fd*)

ファイル記述子 *fd* の端末属性を含むリストを返します。その形式は: [iflag, oflag, cflag, lflag, ispeed, ospeed, cc] です。*cc* は端末特殊文字のリストです (それぞれ長さ 1 の文字列です。ただしインデクス VMIN および VTIME の内容は、それらのフィールドが定義されていた場合整数の値となります)。端末設定フラグおよび端末速度の解釈、および配列 *cc* のインデクス検索は、*termios* で定義されているシンボル定数を使って行わなければなりません。

termios.tcsetattr(*fd*, *when*, *attributes*)

ファイル記述子 *fd* の端末属性を *attributes* から取り出して設定します。*attributes* は *tcgetattr()* が返すようなリストです。引数 *when* は属性がいつ変更されるかを決定します: TCSANOW は即時変更を行い、TCSADRAIN は現在キューされている出力を全て転送した後に変更を行い、TCSAFLUSH は現在キューされている出力を全て転送し、全てのキューされている入力を無視した後に変更を行います。

termios.tcsendbreak(*fd*, *duration*)

ファイル記述子 *fd* にブレークを送信します。*duration* をゼロにすると、0.25〜0.5 秒間のブレークを送信します; *duration* の値がゼロでない場合、その意味はシステム依存です。

termios.tcdrain(*fd*)

ファイル記述子 *fd* に書き込まれた全ての出力が転送されるまで待ちます。

termios.tcflush(*fd*, *queue*)

ファイル記述子 *fd* にキューされたデータを無視します。どのキューかは *queue* セレクタで指定します: TCIFLUSH は入力キュー、TCOFLUSH は出力キュー、TCIOFLUSH は両方のキューです。

termios.tcflow(*fd*, *action*)

ファイル記述子 *fd* の入力または出力をサスペンドしたりレジュームしたりします。引数 *action* は出力をサスペンドする TCOOFF 、出力をレジュームする TCOON 、入力をサスペンドする TCIOFF 、入力をレジュームする TCION をとることができます。

参考:

tty **モジュール** 一般的な端末制御操作のための便利な関数。

35.6.1 使用例

以下はエコーバックを切った状態でパスワード入力を促す関数です。ユーザの入力に関わらず以前の端末属性を正確に回復するために、二つの *tcgetattr()* と try ... finally 文によるテクニックが使われています:

```
def getpass(prompt="Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] = new[3] & ~termios.ECHO          # lflags
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        passwd = input(prompt)
    finally:
```

(次のページに続く)

(前のページからの続き)

```
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```

35.7 tty --- 端末制御のための関数群

ソースコード: Lib/tty.py

---

tty モジュールは端末を cbreak および raw モードにするための関数を定義しています。

このモジュールは *termios* モジュールを必要とするため、Unix でしか動作しません。

tty モジュールでは、以下の関数を定義しています:

tty.**setraw**(*fd*, *when=termios.TCSAFLUSH*)
: ファイル記述子 *fd* のモードを raw モードに変えます。*when* を省略すると標準の値は termios.TCSAFLUSH になり、*termios.tcsetattr()* に渡されます。

tty.**setcbreak**(*fd*, *when=termios.TCSAFLUSH*)
: ファイル記述子 *fd* のモードを cbreak モードに変えます。*when* を省略すると標準の値は termios.TCSAFLUSH になり、*termios.tcsetattr()* に渡されます。

参考:

モジュール *termios* 低水準端末制御インタフェース。

35.8 pty --- 擬似端末ユーティリティ

ソースコード: Lib/pty.py

---

pty モジュールは擬似端末 (他のプロセスを実行してその制御をしている端末をプログラムで読み書きする) を制御する操作を定義しています。

擬似端末の制御はプラットフォームに強く依存するので、Linux 用のコードしか存在していません。(Linux 用のコードは他のプラットフォームでも動作するように作られていますがテストされていません。)

pty モジュールでは以下の関数を定義しています:

pty.**fork**()
: fork します。子プロセスの制御端末を擬似端末に接続します。返り値は (pid, fd) です。子プロセスは *pid* として 0、*fd* として *invalid* をそれぞれ受けとります。親プロセスは *pid* として子プロセスの PID、*fd* として子プロセスの制御端末 (子プロセスの標準入出力に接続されている) のファイル記述子を受けとります。

pty.openpty()

新しい擬似端末のペアを開きます。利用できるなら *os.openpty()* を使い、利用できなければ一般的な Unix システム用のエミュレーションコードを使います。マスター、スレーブそれぞれのためのファイル記述子、(master, slave) のタプルを返します。

pty.spawn(*argv*[, *master_read*[, *stdin_read*]])

Spawn a process, and connect its controlling terminal with the current process's standard io. This is often used to baffle programs which insist on reading from the controlling terminal. It is expected that the process spawned behind the pty will eventually terminate, and when it does *spawn* will return.

The functions *master_read* and *stdin_read* are passed a file descriptor which they should read from, and they should always return a byte string. In order to force spawn to return before the child process exits an *OSError* should be thrown.

The default implementation for both functions will read and return up to 1024 bytes each time the function is called. The *master_read* callback is passed the pseudoterminal's master file descriptor to read output from the child process, and *stdin_read* is passed file descriptor 0, to read from the parent process's standard input.

Returning an empty byte string from either callback is interpreted as an end-of-file (EOF) condition, and that callback will not be called after that. If *stdin_read* signals EOF the controlling terminal can no longer communicate with the parent process OR the child process. Unless the child process will quit without any input, *spawn* will then loop forever. If *master_read* signals EOF the same behavior results (on linux at least).

If both callbacks signal EOF then *spawn* will probably never return, unless *select* throws an error on your platform when passed three empty lists. This is a bug, documented in issue 26228.

引数 argv を指定して **監査イベント** pty.spawn を送出します。

バージョン 3.4 で変更: *spawn()* が *os.waitpid()* が返す子プロセスのステータス値を返すようになりました。

35.8.1 使用例

下記のプログラムは Unix コマンド *script(1)* のように動作します。疑似端末を使用して、端末セッションのすべての入出力を "typescript" に記録します。

```
import argparse
import os
import pty
import sys
import time

parser = argparse.ArgumentParser()
parser.add_argument('-a', dest='append', action='store_true')
parser.add_argument('-p', dest='use_python', action='store_true')
```

(次のページに続く)

(前のページからの続き)

```
parser.add_argument('filename', nargs='?', default='typescript')
options = parser.parse_args()

shell = sys.executable if options.use_python else os.environ.get('SHELL', 'sh')
filename = options.filename
mode = 'ab' if options.append else 'wb'

with open(filename, mode) as script:
    def read(fd):
        data = os.read(fd, 1024)
        script.write(data)
        return data

    print('Script started, file is', filename)
    script.write(('Script started on %s\n' % time.asctime()).encode())

    pty.spawn(shell, read)

    script.write(('Script done on %s\n' % time.asctime()).encode())
    print('Script done, file is', filename)
```

35.9 fcntl --- fcntl および ioctl システムコール

このモジュールでは、ファイル記述子 (file descriptor) に基づいたファイル制御および I/O 制御を実現します。このモジュールは、Unix のルーチンである fcntl() および ioctl() へのインタフェースです。これらのシステムコールの完全な説明は、*fcntl(2)* と *ioctl(2)* の Unix マニュアルページを参照してください。

このモジュール内の全ての関数はファイル記述子 *fd* を最初の引数に取ります。この値は sys.stdin.fileno() が返すような整数のファイル記述子でも、sys.stdin 自体のような、純粋にファイル記述子だけを返す *fileno()* メソッドを提供している *io.IOBase* オブジェクトでもかまいません。

バージョン 3.3 で変更: 以前は *IOError* を送出していたこのモジュールの操作が、*OSError* を送出するようになりました。

バージョン 3.8 で変更: The fcntl module now contains F_ADD_SEALS, F_GET_SEALS, and F_SEAL_* constants for sealing of *os.memfd_create()* file descriptors.

このモジュールには、以下の関数が定義されています:

fcntl.**fcntl**(*fd*, *cmd*, *arg=0*)

操作 *cmd* をファイル記述子 *fd* (または *fileno()* メソッドを提供しているファイルオブジェクト) に対して実行します。*cmd* として用いられる値はオペレーティングシステム依存で、*fcntl* モジュール内に関連する C ヘッダファイルと同じ名前が使われている定数の形で利用出来ます。引数 *arg* は整数値か *bytes* オブジェクトをとります。引数が整数値の場合、この関数の戻り値は C 言語の fcntl() を呼び出した際の整数の戻り値になります。引数が bytes の場合には、*struct.pack()* で作られるようなバイナリの構造体を表します。バイナリデータはバッファにコピーされ、そのアドレスが C 言語

の fcntl() 呼び出しに渡されます。呼び出しが成功した後に戻される値はバッファの内容で、*bytes* オブジェクトに変換されています。返されるオブジェクトは *arg* 引数と同じ長さになります。この値は 1024 バイトに制限されています。オペレーティングシステムからバッファに返される情報の長さが 1024 バイトよりも大きい場合、大抵はセグメンテーション違反となるか、より不可思議なデータの破損を引き起こします。

fcntl() が失敗した場合、*OSError* が送出されます。

引数 fd, cmd, arg を指定して **監査イベント** fcntl.fcntl を送出します。

fcntl.ioctl(*fd*, *request*, *arg=0*, *mutate_flag=True*)

この関数は *fcntl()* 関数と同じですが、引数の扱いがより複雑であるところが異なります。

パラメータ *request* は 32 ビットに収まる値に制限されます。*request* 引数として使うのに関係のある追加の定数は *termios* モジュールにあり、関連する C ヘッダファイルで使われているのと同じ名前が付けられています。

パラメタ *arg* は、整数、(*bytes* のような) 読み出し専用のバッファインタフェースをサポートするオブジェクト、読み書きバッファインタフェースをサポートするオブジェクトのどれかです。

最後の型のオブジェクトを除き、動作は *fcntl()* 関数と同じです。

可変なバッファが渡された場合、動作は *mutate_flag* 引数の値で決定されます。

この値が偽の場合、バッファの可変性は無視され、読み出し専用バッファの場合と同じ動作になりますが、上で述べた 1024 バイトの制限は回避されます -- 従って、オペレーティングシステムが希望するバッファ長までであれば正しく動作します。

mutate_flag が真 (デフォルト) の場合、バッファは (実際には) 根底にある *ioctl()* システムコールに渡され、後者の戻り値が呼び出し側の Python に引き渡され、バッファの新たな内容は *ioctl()* の動作を反映します。この説明はやや単純化されています。というのは、与えられたバッファが 1024 バイト長よりも短い場合、バッファはまず 1024 バイト長の静的なバッファにコピーされてから *ioctl()* に渡され、その後引数で与えたバッファに戻しコピーされるからです。

ioctl() が失敗すると、*OSError* 例外が送出されます。

以下に例を示します:

```
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPGRP, "  "))[0]
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPGRP, buf, 1)
0
>>> buf
array('h', [13341])
```

引数 fd, request, arg を指定して **監査イベント** fcntl.ioctl を送出します。

fcntl.flock(*fd*, *operation*)

ファイル記述子 *fd* (*fileno()* メソッドを提供しているファイルオブジェクトも含む) に対してロック操作 *operation* を実行します。詳細は Unix マニュアルの *flock(2)* を参照してください (システムによっては、この関数は fcntl() を使ってエミュレーションされています)。

flock() が失敗すると、*OSError* 例外が送出されます。

引数 fd, operation を指定して **監査イベント** fcntl.flock を送出します。

fcntl.lockf(*fd*, *cmd*, *len=0*, *start=0*, *whence=0*)

本質的に *fcntl()* によるロッキングの呼び出しをラップしたものです。*fd* はロックまたはアンロックするファイルのファイル記述子 (*fileno()* メソッドを提供するファイルオブジェクトも受け付けられます) で、*cmd* は以下の値のうちいずれかになります:

- LOCK_UN -- アンロック
- LOCK_SH -- 共有ロックを取得
- LOCK_EX -- 排他的ロックを取得

cmd が LOCK_SH または LOCK_EX の場合、LOCK_NB とビット OR にすることでロック取得時にブロックしないようにすることができます。LOCK_NB が使われ、ロックが取得できなかった場合、*OSError* が送出され、例外は *errno* 属性を持ち、その値は EACCES または EAGAIN になります (オペレーティングシステムに依存します; 可搬性のため、両方の値をチェックしてください)。少なくともいくつかのシステムでは、ファイル記述子が参照しているファイルが書き込みのために開かれている場合、LOCK_EX だけしか使うことができません。

len はロックを行いたいバイト数、*start* はロック領域先頭の *whence* からの相対的なバイトオフセット、*whence* は *io.IOBase.seek()* と同じで、具体的には:

- 0 -- ファイル先頭からの相対位置 (*os.SEEK_SET*)
- 1 -- 現在のバッファ位置からの相対位置 (*os.SEEK_CUR*)
- 2 -- ファイルの末尾からの相対位置 (*os.SEEK_END*)

start の標準の値は 0 で、ファイルの先頭から開始することを意味します。*len* の標準の値は 0 で、ファイルの終了までロックすることを表します。*whence* の標準の値も 0 です。

引数 fd, cmd, len, start, whence を指定して **監査イベント** fcntl.lockf を送出します。

以下に (全ての SVR4 互換システムでの) 例を示します:

```
import struct, fcntl, os

f = open(...)
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

最初の例では、戻り値 *rv* は整数値を保持しています; 二つ目の例では *bytes* オブジェクトを保持しています。*lockdata* 変数の構造体レイアウトはシステム依存です --- 従って *flock()* を呼ぶ方が良いでしょう。

参考:

os **モジュール** もし *os* モジュールに *os.O_SHLOCK* と *os.O_EXLOCK* が 存在する場合 (BSD のみ)、*os.open()* 関数は *lockf()* や *flock()* 関数を代替できます。

35.10 pipes --- シェルパイプラインへのインタフェース

ソースコード: Lib/pipes.py

---

pipes モジュールでは、*pipeline* の概念 --- あるファイルを別のファイルに変換する機構の直列接続 --- を抽象化するためのクラスを定義しています。

このモジュールは **/bin/sh** コマンドラインを利用するため、*os.system()* および *os.popen()* のための POSIX 準拠のシェル、または互換のシェルが必要です。

pipes モジュールでは、以下のクラスを定義しています:

class pipes.Template
パイプラインを抽象化したクラス。

以下はプログラム例です:

```
>>> import pipes
>>> t = pipes.Template()
>>> t.append('tr a-z A-Z', '--')
>>> f = t.open('pipefile', 'w')
>>> f.write('hello world')
>>> f.close()
>>> open('pipefile').read()
'HELLO WORLD'
```

35.10.1 テンプレートオブジェクト

テンプレートオブジェクトは以下のメソッドを持っています:

Template.reset()
パイプラインテンプレートを初期状態に戻します。

Template.clone()
元のパイプラインテンプレートと等価の新しいオブジェクトを返します。

Template.debug(*flag***)**
flag が真の場合、デバッグをオンにします。そうでない場合、デバッグをオフにします。デバッグがオンの時には、実行されるコマンドが印字され、より多くのメッセージを出力するようにするために、シェルに `set -x` 命令を与えます。

Template.append(*cmd*, *kind*)
新たなアクションをパイプラインの末尾に追加します。*cmd* 変数は有効な bourne shell 命令でなければなりません。*kind* 変数は二つの文字からなります。

最初の文字は `'-'` (コマンドが標準入力からデータを読み出すことを意味します)、`'f'` (コマンドがコマンドライン上で与えたファイルからデータを読み出すことを意味します)、あるいは `'.'` (コマンドは入力を読まないことを意味します、従ってパイプラインの先頭になります)、のいずれかになります。

同様に、二つ目の文字は `'-'` (コマンドが標準出力に結果を書き込むことを意味します)、`'f'` (コマンドがコマンドライン上で指定したファイルに結果を書き込むことを意味します)、あるいは `'.'` (コマンドはファイルを書き込まないことを意味し、パイプラインの末尾になります)、のいずれかになります。

Template.prepend(*cmd*, *kind*)
パイプラインの先頭に新しいアクションを追加します。引数の説明については *append()* を参照してください。

Template.open(*file*, *mode*)
ファイル類似のオブジェクトを返します。このオブジェクトは *file* を開いていますが、パイプラインを通して読み書きするようになっています。*mode* には `'r'` または `'w'` のいずれか一つしか与えることができないので注意してください。

Template.copy(*infile*, *outfile*)
パイプを通して *infile* を *outfile* にコピーします。

35.11 resource --- リソース使用状態の情報

このモジュールでは、プログラムによって使用されているシステムリソースを計測したり制御するための基本的なメカニズムを提供します。

特定のシステムリソースを指定したり、現在のプロセスやその子プロセスのリソース使用情報を要求するためにシンボル定数が使われます。

システムコールが失敗した場合 *OSError* を送出します。

exception resource.error
OSError の非推奨のエイリアスです。

バージョン 3.3 で変更: **PEP 3151** に基づき、このクラスは *OSError* のエイリアスになりました。

35.11.1 リソースの制限

リソースの使用は下に述べる *setrlimit()* 関数を使って制限することができます。各リソースは二つ組の制限値: ソフトリミット (soft limit) 、およびハードリミット (hard limit) 、で制御されます。ソフトリミットは現在の制限値で、時間とともにプロセスによって下げたり上げたりできます。ソフトリミットはハードリミットを超えることはできません。ハードリミットはソフトリミットよりも高い任意の値まで下げることができますが、上げることはできません。(スーパユーザの有効な UID を持つプロセスのみがハードリミットを上げることができます。)

制限をかけるべく指定できるリソースはシステムに依存します。指定できるリソースは *getrlimit(2)* マニュアルページで解説されています。以下に列挙するリソースは背後のオペレーティングシステムがサポートする場合にサポートされています; オペレーティングシステム側で値を調べたり制御したりできないリソースは、そのプラットフォーム向けのこのモジュール内では定義されていません。

resource.RLIM_INFINITY

無制限のリソースの上限を示すための定数です。

resource.getrlimit(*resource*)

resource の現在のソフトおよびハードリミットを表すタプル (`soft`, `hard`) を返します。無効なリソースが指定された場合には *ValueError* が、背後のシステムコールが予期せず失敗した場合には *error* が送出されます。

resource.setrlimit(*resource*, *limits*)

resource の新たな消費制限を設定します。*limits* 引数には、タプル (`soft`, `hard`) による二つの整数で、新たな制限を記述しなければなりません。*RLIM_INFINITY* を指定することで、無制限を要求することが出来ます。

無効なリソースが指定された場合、ソフトリミットの値がハードリミットの値を超えている場合、プロセスがハードリミットを引き上げようとした場合には *ValueError* が送出されます。リソースのハードリミットやシステムリミットが無制限でないのに *RLIM_INFINITY* を指定した場合も、*ValueError* になります。スーパーユーザの実効 UID を持ったプロセスは無制限を含めあらゆる妥当な制限値を要求出来ますが、システムが課している制限を超過した要求ではやはり *ValueError* となります。

`setrlimit` は背後のシステムコールが予期せず失敗した場合に、*error* を送出する場合があります。

VxWorks only supports setting *RLIMIT_NOFILE*.

引数 `src`, `dst`, ``limits`` を指定して **監査イベント** `resource.setrlimit` を送出します。

resource.prlimit(*pid*, *resource*[, *limits*])

1 つの関数の中で *setrlimit()* と *getrlimit()* を組み合わせ、任意のプロセスのリソースの制限値を取得したり設定したりします。*pid* が 0 の場合は、現在のプロセスに適用されます。*resource* および *limits* は、*limits* がオプションであることを除けば、*setrlimit()* と同じ意味です。

limits が与えられないときは、関数はプロセス *pid* の *resource* の制限値を返します。*limits* が与えられたときは、プロセスの *resource* の制限値が設定され、設定が変更される前のリソースの制限値が返されます。

pid が見付からないときは *ProcessLookupError* を、ユーザがプロセスの CAP_SYS_RESOURCE を持ってないときは *PermissionError* を送出します。

引数 pid, dst`, ``limits を指定して **監査イベント** resource.prlimit を送出します。

Availability: Linux 2.6.36 or later with glibc 2.13 or later.

バージョン 3.4 で追加.

以下のシンボルは、後に述べる関数 *setrlimit()* および *getrlimit()* を使って消費量を制御することができるリソースを定義しています。これらのシンボルの値は、C プログラムで使われているシンボルと全く同じです。

getrlimit(2) の Unix マニュアルページには、指定可能なリソースが列挙されています。全てのシステムで同じシンボルが使われているわけではなく、また同じリソースを表すために同じ値が使われているとも限らないので注意してください。このモジュールはプラットフォーム間の相違を隠蔽しようとはしていません --- あるプラットフォームで定義されていないシンボルは、そのプラットフォーム向けの本モジュールでは利用することができません。

resource.RLIMIT_CORE
: 現在のプロセスが生成できるコアファイルの最大 (バイト) サイズです。プロセスの全体イメージを入れるためにこの値より大きなサイズのコアファイルが要求された結果、部分的なコアファイルが生成される可能性があります。

resource.RLIMIT_CPU
: プロセッサが利用することができる最大プロセッサ時間 (秒) です。この制限を超えた場合、SIGXCPU シグナルがプロセスに送られます。(どのようにしてシグナルを捕捉したり、例えば開かれているファイルをディスクにフラッシュするといった有用な処理を行うかについての情報は、*signal* モジュールのドキュメントを参照してください)

resource.RLIMIT_FSIZE
: プロセスが作成するファイルの最大サイズです。

resource.RLIMIT_DATA
: プロセスのヒープの最大 (バイト) サイズです。

resource.RLIMIT_STACK
: 現在のプロセスのコールスタックの最大サイズ (バイト単位) です。これはマルチスレッドプロセスのメインスレッドのスタックのみに影響します。

resource.RLIMIT_RSS
: プロセスが取りうる最大 RAM 常駐ページサイズ (resident set size) です。

resource.RLIMIT_NPROC
: 現在のプロセスが生成できるプロセスの上限です。

resource.RLIMIT_NOFILE
: 現在のプロセスが開けるファイル記述子の上限です。

resource.RLIMIT_OFILE
: *RLIMIT_NOFILE* の BSD での名称です。

resource.RLIMIT_MEMLOCK
: メモリ中でロックできる最大アドレス空間です。

resource.RLIMIT_VMEM
: プロセスが占有できるマップメモリの最大領域です。

resource.RLIMIT_AS
: アドレス空間でプロセスが占有できる最大領域 (バイト単位) です。

resource.RLIMIT_MSGQUEUE
: POSIX メッセージキューに割り当てることの出来るバイト数です。

  Availability: Linux 2.6.8 or later.

  バージョン 3.4 で追加.

resource.RLIMIT_NICE
: プロセスの nice の上限です (20 - rlim_cur)。

  Availability: Linux 2.6.12 or later.

  バージョン 3.4 で追加.

resource.RLIMIT_RTPRIO
: リアルタイム優先順位の上限です。

  Availability: Linux 2.6.12 or later.

  バージョン 3.4 で追加.

resource.RLIMIT_RTTIME
: リアルタイムスケジューリングにおいて、プロセスがブロッキングシステムコールを行わずに使用できる CPU 時間の制限値（マイクロ秒単位)。

  Availability: Linux 2.6.25 or later.

  バージョン 3.4 で追加.

resource.RLIMIT_SIGPENDING
: プロセスがキュー出来るシグナルの数です。

  Availability: Linux 2.6.8 or later.

  バージョン 3.4 で追加.

resource.RLIMIT_SBSIZE
: このユーザが使用するソケットバッファの最大サイズ (バイト単位)。これは、このユーザが常に保持できるネットワークメモリの量、つまり mbuf の量を制限します。

  Availability: FreeBSD 9 or later.

バージョン 3.4 で追加.

resource.RLIMIT_SWAP
: The maximum size (in bytes) of the swap space that may be reserved or used by all of this user id's processes. This limit is enforced only if bit 1 of the vm.overcommit sysctl is set. Please see tuning(7) for a complete description of this sysctl.

Availability: FreeBSD 9 or later.

バージョン 3.4 で追加.

resource.RLIMIT_NPTS
: このユーザ ID が作成する擬似端末の数の上限です。

Availability: FreeBSD 9 or later.

バージョン 3.4 で追加.

35.11.2 リソースの使用状態

以下の関数はリソース使用情報を取得するために使われます:

resource.getrusage(*who*)
: この関数は、*who* 引数で指定される、現プロセスおよびその子プロセスによって消費されているリソースを記述するオブジェクトを返します。*who* 引数は以下に記述される RUSAGE_* 定数のいずれかを使って指定します。

簡単な例:

```
from resource import *
import time

# a non CPU-bound task
time.sleep(3)
print(getrusage(RUSAGE_SELF))

# a CPU-bound task
for i in range(10 ** 8):
    _ = 1 + 1
print(getrusage(RUSAGE_SELF))
```

返される値の各フィールドはそれぞれ、個々のシステムリソースがどれくらい使用されているか、例えばユーザモードでの実行に費やされた時間やプロセスが主記憶からスワップアウトされた回数、を示しています。幾つかの値、例えばプロセスが使用しているメモリ量は、内部時計の最小単位に依存します。

以前のバージョンとの互換性のため、返される値は 16 要素からなるタプルとしてアクセスすることもできます。

戻り値のフィールド ru_utime および ru_stime は浮動小数点数で、それぞれユーザモードでの実行に費やされた時間、およびシステムモードでの実行に費やされた時間を表します。それ以外の値は整数

です。これらの値に関する詳しい情報は *getrusage(2)* を調べてください。以下に簡単な概要を示します:

インデックス	フィールド	リソース
0	ru_utime	time in user mode (float seconds)
1	ru_stime	time in system mode (float seconds)
2	ru_maxrss	最大常駐ページサイズ
3	ru_ixrss	共有メモリサイズ
4	ru_idrss	非共有メモリサイズ
5	ru_isrss	非共有スタックサイズ
6	ru_minflt	I/O を必要としないページフォールト数
7	ru_majflt	I/O を必要とするページフォールト数
8	ru_nswap	スワップアウト回数
9	ru_inblock	ブロック入力操作数
10	ru_oublock	ブロック出力操作数
11	ru_msgsnd	送信メッセージ数
12	ru_msgrcv	受信メッセージ数
13	ru_nsignals	受信シグナル数
14	ru_nvcsw	自発的な実行コンテキスト切り替え数
15	ru_nivcsw	非自発的な実行コンテキスト切り替え数

この関数は無効な *who* 引数を指定した場合には *ValueError* を送出します。また、異常が発生した場合には *error* 例外が送出される可能性があります。

resource.**getpagesize**()

システムページ内のバイト数を返します。(ハードウェアページサイズと同じとは限りません。)

以下の RUSAGE_* シンボルはどのプロセスの情報を提供させるかを指定するために関数 *getrusage()* に渡されます。

resource.**RUSAGE_SELF**

getrusage() に渡すと呼び出し中のプロセスが消費しているリソースを要求します。そのプロセスの全スレッドが使用するリソースの合計です。

resource.**RUSAGE_CHILDREN**

呼び出し元のプロセスの子プロセスが消費するリソースを要求するために、*getrusage()* に渡して終了させ、待機させることができます。

resource.**RUSAGE_BOTH**

getrusage() に渡すと現在のプロセスおよび子プロセスの両方が消費しているリソースを要求します。全てのシステムで利用可能なわけではありません。

resource.**RUSAGE_THREAD**

getrusage() に渡すと現在のスレッドが消費しているリソースを要求します。全てのシステムで利用可能なわけではありません。

バージョン 3.2 で追加.

35.12 nis --- Sun の NIS (Yellow Pages) へのインタフェース

nis モジュールは複数のホストを集中管理する上で便利な NIS ライブラリを薄くラップします。

NIS は Unix システム上にしかないので、このモジュールは Unix でしか利用できません。

nis モジュールでは以下の関数を定義しています:

nis.match(*key*, *mapname*, *domain=default_domain*)

mapname 中で *key* に一致するものを返すか、見つからない場合にはエラー (*nis.error*) を送出します。両方の引数とも文字列で、*key* は 8 ビットクリーンです。返される値は (NULL その他を含む可能性のある) 任意のバイト列です。

mapname は他の名前の別名になっていないか最初にチェックされます。

domain 引数は検索に使う NIS ドメインを上書きできます。指定されなければ、デフォルト NIS ドメイン内が検索されます。

nis.cat(*mapname*, *domain=default_domain*)

`match(key, mapname)==value` となる *key* を *value* に対応付ける辞書を返します。辞書内のキーと値は共に任意のバイト列なので注意してください。

mapname は他の名前の別名になっていないか最初にチェックされます。

domain 引数は検索に使う NIS ドメインを上書きできます。指定されなければ、デフォルト NIS ドメイン内が検索されます。

nis.maps(*domain=default_domain*)

有効なマップのリストを返します。

domain 引数は検索に使う NIS ドメインを上書きできます。指定されなければ、デフォルト NIS ドメイン内が検索されます。

nis.get_default_domain()

システムのデフォルト NIS ドメインを返します。

nis モジュールは以下の例外を定義しています:

exception nis.error

NIS 関数がエラーコードを返した場合に送出されます。

35.13 syslog --- Unix syslog ライブラリルーチン群

このモジュールでは Unix syslog ライブラリルーチン群へのインタフェースを提供します。syslog の便宜レベルに関する詳細な記述は Unix マニュアルページを参照してください。

このモジュールはシステムの syslog ファミリのルーチンをラップしています。syslog サーバーと通信できる pure Python のライブラリが、*logging.handlers* モジュールの SysLogHandler にあります。

このモジュールには、以下の関数が定義されています:

syslog.**syslog**(*message*)
syslog.**syslog**(*priority*, *message*)

文字列 *message* をシステムログ機構に送信します。末尾の改行文字は必要に応じて追加されます。各メッセージは *facility* および *level* からなる優先度でタグ付けされます。オプションの *priority* 引数はメッセージの優先度を定義します。標準の値は LOG_INFO です。*priority* 中に、便宜レベルが (LOG_INFO | LOG_USER のように) 論理和を使ってコード化されていない場合、*openlog()* を呼び出した際の値が使われます。

syslog() が呼び出される前に *openlog()* が呼び出されなかった場合、openlog() が引数なしで呼び出されます。

引数 priority, message を指定して **監査イベント** syslog.syslog を送出します。

syslog.**openlog**([*ident*[, *logoption*[, *facility*]]])

openlog() 関数を呼び出すことで以降の *syslog()* の呼び出しに対するログオプションを設定することができます。ログがまだ開かれていない状態で *syslog()* を呼び出すと *openlog()* が引数なしで呼び出されます。

オプションの *ident* キーワード引数は全てのメッセージの先頭に付く文字列で、デフォルトでは sys.argv[0] から前方のパス部分を取り除いたものです。オプションの *logoption* キーワード引数 (デフォルトは 0) はビットフィールドです。組み合わせられる値については下記を参照してください。オプションの *facility* キーワード引数 (デフォルトは LOG_USER) は明示的に facility が encode されていないメッセージに設定される facility です。

引数 ident, *logoptiona‘*, facility を指定して **監査イベント** syslog.openlog を送出します。

バージョン 3.2 で変更: 前のバージョンでは、キーワード引数が使えず、*ident* が必須でした。*ident* のデフォルトはシステムライブラリに依存しており、Python プログラムのファイル名ではなく python となっていることが多かった。

syslog.**closelog**()

syslog モジュールの値をリセットし、システムライブラリの closelog() を呼び出します。

この関数を呼ぶと、モジュールが最初に import されたときと同じようにふるまいます。例えば、(*openlog()* を呼び出さないで) *syslog()* を最初に呼び出したときに、*openlog()* が呼び出され、*ident* やその他の *openlog()* の引数はデフォルト値にリセットされます。

引数無しで **監査イベント** syslog.closelog を送出します。

syslog.**setlogmask**(*maskpri*)

優先度マスクを *maskpri* に設定し、以前のマスク値を返します。*maskpri* に設定されていない優先度レベルを持った *syslog()* の呼び出しは無視されます。標準では全ての優先度をログ出力します。関数 LOG_MASK(pri) は個々の優先度 *pri* に対する優先度マスクを計算します。関数 LOG_UPTO(pri) は優先度 *pri* までの全ての優先度を含むようなマスクを計算します。

引数 maskpri を指定して 監査イベント socket.setlogmask を送出します。

このモジュールでは以下の定数を定義しています:

優先度 (降順): LOG_EMERG、LOG_ALERT、LOG_CRIT、LOG_ERR、LOG_WARNING、LOG_NOTICE、LOG_INFO、LOG_DEBUG。

機能: LOG_KERN、LOG_USER、LOG_MAIL、LOG_DAEMON、LOG_AUTH、LOG_LPR、LOG_NEWS、LOG_UUCP、LOG_CRON、LOG_SYSLOG、LOG_LOCAL0 から LOG_LOCAL7、および <syslog.h> で定義されていれば、LOG_AUTHPRIV。

ログオプション: LOG_PID、LOG_CONS、LOG_NDELAY、また <syslog.h> で定義されている場合、LOG_ODELAY、LOG_NOWAIT、および LOG_PERROR。

35.13.1 使用例

シンプルな例

1 つ目のシンプルな例:

```
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

いくつかのログオプションを設定する例。ログメッセージにプロセス ID を含み、メッセージをメールのログ用の facility にメッセージを書きます:

```
syslog.openlog(logoption=syslog.LOG_PID, facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```

第

THIRTYSIX

取って代わられたモジュール群

この章に記述されたモジュール群は非推奨で、後方互換性のためにのみ保存されています。これらは他のモジュール群に取って代わられました。

36.1 optparse --- コマンドラインオプション解析器

ソースコード: Lib/optparse.py

バージョン 3.2 で非推奨: *optparse* モジュールは廃止予定であり、これ以上の開発は行われません。*argparse* モジュールを使用してください。

---

optparse モジュールは、昔からある *getopt* よりも簡便で、柔軟性に富み、かつ強力なコマンドライン解析ライブラリです。*optparse* では、より宣言的なスタイルのコマンドライン解析手法、すなわち *OptionParser* のインスタンスを作成してオプションを追加してゆき、そのインスタンスでコマンドラインを解析するという手法をとっています。*optparse* を使うと、GNU/POSIX 構文でオプションを指定できるだけでなく、使用法やヘルプメッセージの生成も行えます。

optparse を使った簡単なスクリプトの例を以下に示します:

```
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```

このようにわずかな行数のコードによって、スクリプトのユーザはコマンドライン上で例えば以下のような「よくある使い方」を実行できるようになります:

```
<yourscript> --file=outfile -q
```

コマンドライン解析の中で、*optparse* はユーザの指定したコマンドライン引数値に応じて parse_args() の返す options の属性値を設定してゆきます。parse_args() がコマンドライン解析から処理を戻したとき、options.filename は "outfile" に、options.verbose は False になっているはずです。*optparse* は長い形式と短い形式の両方のオプション表記をサポートしており、短い形式は結合して指定できます。また、様々な形でオプションに引数値を関連付けられます。従って、以下のコマンドラインは全て上の例と同じ意味になります:

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

さらに、ユーザが以下のいずれかを実行すると

```
<yourscript> -h
<yourscript> --help
```

optparse はスクリプトのオプションについて簡単にまとめた内容を出力します:

```
Usage: <yourscript> [options]

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  write report to FILE
  -q, --quiet           don't print status messages to stdout
```

yourscript の中身は実行時に決まります (通常は sys.argv[0] になります)。

36.1.1 背景

optparse は、素直で慣習に則ったコマンドラインインタフェースを備えたプログラムの作成を援助する目的で設計されました。その結果、Unix で慣習的に使われているコマンドラインの構文や機能だけをサポートするに留まっています。こうした慣習に詳しくなければ、よく知っておくためにもこの節を読んでおきましょう。

用語集

引数 (argument) コマンドラインでユーザが入力するテキストの塊で、シェルが execl や execv に引き渡すものです。Python では、引数は sys.argv[1:] の要素となります。(sys.argv[0] は実行しようとしているプログラムの名前です。引数解析に関しては、この要素はあまり重要ではありません。) Unix シェルでは、「語 (word)」という用語も使います。

場合によっては sys.argv[1:] 以外の引数リストを代入する方が望ましいことがあるので、「引数」は「 sys.argv[1:] または sys.argv[1:] の代替として提供される別のリストの要素」と読むべきでしょう。

オプション (option) 追加的な情報を与えるための引数で、プログラムの実行に対する教示やカスタマイズを行います。オプションには多様な文法が存在します。伝統的な Unix における書法はハイフン ("-") の

後ろに一文字が続くもので、例えば `-x` や `-F` です。また、伝統的な Unix における書法では、複数のオプションを一つの引数にまとめられます。例えば `-x -F` は `-xF` と等価です。GNU プロジェクトでは `--` の後ろにハイフンで区切りの語を指定する方法、例えば `--file` や `--dry-run` も提供しています。*optparse* は、これら二種類のオプション書法だけをサポートしています。

他に見られる他のオプション書法には以下のようなものがあります:

- ハイフンの後ろに数個の文字が続くもので、例えば `-pf` (このオプションは複数のオプションを一つにまとめたものとは **違います**)
- ハイフンの後ろに語が続くもので、例えば `-file` (これは技術的には上の書式と同じですが、通常同じプログラム上で一緒に使うことはありません)
- プラス記号の後ろに一文字、数個の文字、または語を続けたもので、例えば `+f`, `+rgb`
- スラッシュ記号の後ろに一文字、数個の文字、または語を続けたもので、例えば `/f`, `/file`

上記のオプション書法は *optparse* ではサポートしておらず、今後もサポートする予定はありません。これは故意によるものです: 最初の三つはどの環境の標準でもなく、最後の一つは VMS や MS-DOS, そして Windows を対象にしているときにしか意味をなさないからです。

オプション引数 (option argument) あるオプションの後ろに続く引数で、そのオプションに密接な関連をもち、オプションと同時に引数リストから取り出されます。*optparse* では、オプション引数は以下のように別々の引数にできます:

```
-f foo
--file foo
```

また、一つの引数中にも入れられます:

```
-ffoo
--file=foo
```

通常、オプションは引数をとることもとらないこともあります。あるオプションは引数をとることがなく、またあるオプションは常に引数をとります。多くの人々が「オプションのオプション引数」機能を欲しています。これは、あるオプションが引数が指定されている場合には引数をとり、そうでない場合には引数をもたないようにするという機能です。この機能は引数解析をあいまいにするため、議論の的となっています: 例えば、もし `-a` がオプション引数をとり、`-b` がまったく別のオプションだとしたら、`-ab` をどうやって解析すればいいのでしょうか？ こうした曖昧さが存在するため、*optparse* は今のところこの機能をサポートしていません。

位置引数 (positional argument) 他のオプションが解析される、すなわち他のオプションとその引数が解析されて引数リストから除去された後に引数リストに置かれているものです。

必須のオプション (required option) コマンドラインで与えなければならないオプションです;「必須のオプション (required option)」という語は、英語では矛盾した言葉です。*optparse* では必須オプションの実装を妨げてはいませんが、とりたてて実装上役立つこともしていません。

例えば、下記のような架空のコマンドラインを考えてみましょう:

```
prog -v --report report.txt foo bar
```

-v と --report はどちらもオプションです。--report オプションが引数をとるとすれば、report.txt はオプションの引数です。foo と bar は位置引数になります。

オプションとは何か

オプションはプログラムの実行を調整したり、カスタマイズしたりするための補助的な情報を与えるために使います。もっとはっきりいうと、オプションはあくまでもオプション (省略可能) であるということです。本来、プログラムはともかくもオプションなしでうまく実行できてしかるべきです。(Unix や GNU ツールセットのプログラムをランダムにピックアップしてみてください。オプションを全く指定しなくてもちゃんと動くでしょう？ 例外は find, tar, dd くらいです---これらの例外は、オプション文法が標準的でなく、インタフェースが混乱を招くと酷評されてきた変種のはみ出しものなのです)

多くの人が自分のプログラムに「必須のオプション」を持たせたいと考えます。しかしよく考えてください。必須なら、それは **オプション** *(省略可能)* **ではないのです！** プログラムを正しく動作させるのに絶対的に必要な情報があるとすれば、そこには位置引数を割り当てるべきなのです。

良くできたコマンドラインインタフェース設計として、ファイルのコピーに使われる cp ユーティリティのことを考えてみましょう。ファイルのコピーでは、コピー先を指定せずにファイルをコピーするのは無意味な操作ですし、少なくとも一つのコピー元が必要です。従って、cp は引数無しで実行すると失敗します。とはいえ、cp はオプションを全く必要としない柔軟で便利なコマンドライン文法を備えています:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

まだあります。ほとんどの cp の実装では、ファイルモードや変更時刻を変えずにコピーする、シンボリックリンクの追跡を行わない、すでにあるファイルを上書きする前にユーザに尋ねる、など、ファイルをコピーする方法をいじるための一連のオプションを実装しています。しかし、こうしたオプションは、一つのファイルを別の場所にコピーする、または複数のファイルを別のディレクトリにコピーするという、cp の中心的な処理を乱すことはないのです。

位置引数とは何か

位置引数とは、プログラムを動作させる上で絶対的に必要な情報となる引数です。

よいユーザインタフェースとは、絶対に必要だとされるものが可能な限り少ないものです。プログラムを正しく動作させるために 17 個もの別個の情報が必要だとしたら、その **方法** はさして問題にはなりません ---ユーザはプログラムを正しく動作させられないうちに諦め、立ち去ってしまうからです。ユーザインタフェースがコマンドラインでも、設定ファイルでも、GUI やその他の何であっても同じです: 多くの要求をユーザに押し付ければ、ほとんどのユーザはただ音をあげてしまうだけなのです。

要するに、ユーザが絶対に提供しなければならない情報だけに制限する --- そして可能な限りよく練られたデフォルト設定を使うよう試みてください。もちろん、プログラムには適度な柔軟性を持たせたいとも望むはずですが、それこそがオプションの果たす役割です。繰り返しますが、設定ファイルのエントリであろうが、GUI でできた「環境設定」ダイアログ上のウィジェットであろうが、コマンドラインオプションであろうが

関係ありません --- より多くのオプションを実装すればプログラムはより柔軟性を持ちますが、実装はより難解になるのです。高すぎる柔軟性はユーザを閉口させ、コードの維持をより難しくするのです。

36.1.2 チュートリアル

optparse はとても柔軟で強力でありながら、ほとんどの場合には簡単に利用できます。この節では、*optparse* ベースのプログラムで広く使われているコードパターンについて述べます。

まず、*OptionParser* クラスを import しておかなければなりません。次に、プログラムの冒頭で *OptionParser* インスタンスを生成しておきます:

```
from optparse import OptionParser
...
parser = OptionParser()
```

これでオプションを定義できるようになりました。基本的な構文は以下の通りです:

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

各オプションには、`-f` や `--file` のような一つまたは複数のオプション文字列と、パーザがコマンドライン上のオプションを見つけた際に、何を準備し、何を行うべきかを *optparse* に教えるためのオプション属性 (option attribute) がいくつか入ります。

通常、各オプションには短いオプション文字列と長いオプション文字列があります。例えば:

```
parser.add_option("-f", "--file", ...)
```

オプション文字列は、(ゼロ文字の場合も含め) いくらでも短く、またいくらでも長くできます。ただしオプション文字列は少なくとも一つなければなりません。

OptionParser.add_option() に渡されたオプション文字列は、実際にはこの関数で定義したオプションに対するラベルになります。簡単のため、以後ではコマンドライン上で **オプションを見つける** という表現をしばしば使いますが、これは実際には *optparse* がコマンドライン上の **オプション文字列** を見つけ、対応づけされているオプションを探し出す、という処理に相当します。

オプションを全て定義したら、*optparse* にコマンドラインを解析するように指示します:

```
(options, args) = parser.parse_args()
```

(お望みなら、`parse_args()` に自作の引数リストを渡してもかまいません。とはいえ、実際にはそうした必要はほとんどないでしょう: `optionparser` はデフォルトで `sys.argv[1:]` を使うからです。)

`parse_args()` は二つの値を返します:

- 全てのオプションに対する値の入ったオブジェクト `options` --- 例えば、`--file` が単一の文字列引数をとる場合、`options.file` はユーザが指定したファイル名になります。オプションを指定しなかった場合には `None` になります

- オプションの解析後に残った位置引数からなるリスト args

このチュートリアルの節では、最も重要な四つのオプション属性: *action*, *type*, *dest* (destination), *help* についてしか触れません。このうち最も重要なのは *action* です。

オプション・アクションを理解する

アクション (action) は *optparse* がコマンドライン上にあるオプションを見つけたときに何をすべきかを指示します。*optparse* には押し着せのアクションのセットがハードコードされています。新たなアクションの追加は上級者向けの話題であり、*optparse* **の拡張** で触れます。ほとんどのアクションは、値を何らかの変数に記憶するよう *optparse* に指示します --- 例えば、文字列をコマンドラインから取り出して、options の属性の中に入れる、といった具合にです。

オプション・アクションを指定しない場合、*optparse* のデフォルトの動作は store になります。

store アクション

もっとも良く使われるアクションは store です。このアクションは次の引数 (あるいは現在の引数の残りの部分) を取り出し、正しい型の値か確かめ、指定した保存先に保存するよう *optparse* に指示します。

例えば:

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

例えば、以下のように指定しておき、偽のコマンドラインを作成して *optparse* に解析させてみましょう:

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

オプション文字列 -f を見つけると、*optparse* は次の引数である foo.txt を消費し、その値を options.filename に保存します。従って、この parse_args() 呼び出し後には options.filename は "foo.txt" になっています。

オプションの型として、*optparse* は他にも int や float をサポートしています:

```
parser.add_option("-n", type="int", dest="num")
```

このオプションには長い形式のオプション文字列がないため、設定に問題がないということに注意してください。また、デフォルトのアクションは store なので、ここでは action を明示的に指定していません。

架空のコマンドラインをもう一つ解析してみましょう。今度は、オプション引数をオプションの右側にぴったりくっつけて一緒くたにします: -n42 (一つの引数のみ) は -n 42 (二つの引数からなる) と等価になるので

```
(options, args) = parser.parse_args(["-n42"])
print(options.num)
```

は 42 を出力します。

型を指定しない場合、*optparse* は引数を string であると仮定します。デフォルトのアクションが store であることも併せて考えると、最初の例はもっと短くなります:

```
parser.add_option("-f", "--file", dest="filename")
```

保存先 (destination) を指定しない場合、*optparse* はデフォルト値としてオプション文字列から気のきいた名前を設定します: 最初に指定した長い形式のオプション文字列が --foo-bar であれば、デフォルトの保存先は foo_bar になります。長い形式のオプション文字列がなければ、*optparse* は最初に指定した短い形式のオプション文字列を探します: 例えば、-f に対する保存先は f になります。

optparse にはビルトインの complex 型も含まれています。型の追加については *optparse* **の拡張** で触れています。

ブール値 (フラグ) オプションの処理

フラグオプション---特定のオプションに対して真または偽の値の値を設定するオプション--- はよく使われます。*optparse* では、二つのアクション、store_true および store_false をサポートしています。例えば、verbose というフラグを -v で有効にして、-q で無効にしたいとします:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

ここでは二つのオプションに同じ保存先を指定していますが、全く問題ありません。(下記のように、デフォルト値の設定を少し注意深く行わなければならないだけです。)

-v をコマンドライン上に見つけると、*optparse* は options.verbose を True に設定します。-q を見つければ、options.verbose は False にセットされます。

その他のアクション

この他にも、*optparse* は以下のようなアクションをサポートしています:

"store_const"　定数値を保存します

"append"　オプションの引数を指定のリストに追加します

"count"　指定のカウンタを 1 増やします

"callback"　指定の関数を呼び出します

これらのアクションについては、**リファレンスガイド** 節および **オプション処理コールバック** 節で触れます。

デフォルト値

上記の例は全て、何らかのコマンドラインオプションが見つかった時に何らかの変数 (保存先: destination) に値を設定していました。では、該当するオプションが見つからなかった場合には何が起きるのでしょうか？デフォルトは全く与えていないため、これらの値は全て None になります。たいていはこれで十分ですが、もっときちんと制御したい場合もあります。*optparse* では各保存先に対してデフォルト値を指定し、コマンドラインの解析前にデフォルト値が設定されるようにできます。

まず、verbose/quiet の例について考えてみましょう。*optparse* に対して、-q がない限り verbose を True に設定させたいなら、以下のようにします:

```
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

デフォルトの値は特定のオプションではなく **保存先** に対して適用されます。また、これら二つのオプションはたまたま同じ保存先を持っているにすぎないため、上のコードは下のコードと全く等価になります:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

下のような場合を考えてみましょう:

```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

やはり verbose のデフォルト値は True になります; 特定の目的変数に対するデフォルト値として有効なのは、最後に指定した値だからです。

デフォルト値をすっきりと指定するには、*OptionParser* の set_defaults() メソッドを使います。このメソッドは parse_args() を呼び出す前ならいつでも使えます:

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

前の例と同様、あるオプションの値の保存先に対するデフォルトの値は最後に指定した値になります。コードを読みやすくするため、デフォルト値を設定するときには両方のやり方を混ぜるのではなく、片方だけを使うようにしましょう。

ヘルプの生成

optparse にはヘルプと使い方の説明 (usage text) を生成する機能があり、ユーザに優しいコマンドラインインタフェースを作成する上で役立ちます。やらなければならないのは、各オプションに対する *help* の値と、必要ならプログラム全体の使用法を説明する短いメッセージを与えることだけです。ユーザフレンドリな (ドキュメント付きの) オプションを追加した *OptionParser* を以下に示します:

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
```

(次のページに続く)

(前のページからの続き)

```
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose", default=True,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output to FILE")
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, intermediate, "
                       "or expert [default: %default]")
```

optparse がコマンドライン上で -h や --help を見つけた場合や、parser.print_help() を呼び出した場合、この *OptionParser* は以下のようなメッセージを標準出力に出力します:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]
```

(help オプションでヘルプを出力した場合、*optparse* は出力後にプログラムを終了します。)

optparse ができるだけうまくメッセージを生成するよう手助けするには、他にもまだまだやるべきことがあります:

- スクリプト自体の利用法を表すメッセージを定義します:

  ```
  usage = "usage: %prog [options] arg1 arg2"
  ```

  optparse は %prog を現在のプログラム名、すなわち os.path.basename(sys.argv[0]) と置き換えます。この文字列は詳細なオプションヘルプの前に展開され出力されます。

  usage の文字列を指定しない場合、*optparse* は型どおりとはいえ気の利いたデフォルト値、"Usage: %prog [options]" を使います。位置引数をとらないスクリプトの場合はこれで十分でしょう。

- 全てのオプションにヘルプ文字列を定義します。行の折り返しは気にしなくてかまいません---*optparse* は行の折り返しに気を配り、見栄えのよいヘルプ出力を生成します。

- オプションが値をとるということは自動的に生成されるヘルプメッセージの中で分かります。例えば、”mode” option の場合にはこのようになります:

  ```
  -m MODE, --mode=MODE
  ```

ここで "MODE" はメタ変数 (meta-variable) と呼ばれます: メタ変数は、ユーザが -m/--mode に対して指定するはずの引数を表します。デフォルトでは、*optparse* は保存先の変数名を大文字だけにしたものをメタ変数に使います。これは時として期待通りの結果になりません --- 例えば、上の例の --filename オプションでは明示的に metavar="FILE" を設定しており、その結果自動生成されたオプション説明テキストは:

```
-f FILE, --filename=FILE
```

この機能の重要さは、単に表示スペースを節約するといった理由にとどまりません: 上の例では、手作業で書いたヘルプテキストの中でメタ変数として FILE を使っています。その結果、ユーザに対してやや堅苦しい表現の書法 -f FILE と、より平易に意味付けを説明した "write output to FILE" との間に対応があるというヒントを与えています。これは、エンドユーザにとってより明解で便利なヘルプテキストを作成する単純でありながら効果的な手法なのです。

- デフォルト値を持つオプションはヘルプ文字列に %default を含むことができます---*optparse* はそれをオプションのデフォルト値に *str()* を適用したもので置き換えます。オプションがデフォルト値を持たない (もしくはデフォルト値が None である) 場合、%default は none に展開されます。

オプションをグループ化する

たくさんのオプションを扱う場合、オプションをグループ分けするとヘルプ出力が見やすくなります。*OptionParser* は、複数のオプションをまとめたオプショングループを複数持つことができます。

オプションのグループは、*OptionGroup* を使って作成します:

class optparse.**OptionGroup**(*parser*, *title*, *description=None*)

ここでは:

- *parser* は、このグループが属する *OptionParser* のインスタンスです
- *title* はグループのタイトルです
- *description* はオプションで、グループの長い説明です

OptionGroup は (*OptionParser* のように) OptionContainer を継承していて、オプションをグループに追加するために add_option() メソッドを利用できます。

全てのオプションを定義したら、*OptionParser* の add_option_group() メソッドを使ってグループを定義済みのパーサーに追加します。

前のセクションで定義したパーサーに、続けて *OptionGroup* を追加します:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)
```

この結果のヘルプ出力は次のようになります:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.
```

さらにサンプルを拡張して、複数のグループを使うようにしてみます:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true",
                 help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
                 help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every action done")
parser.add_option_group(group)
```

出力結果は次のようになります:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or expert
                        [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.
```

(次のページに続く)

(前のページからの続き)

```
  Debug Options:
    -d, --debug         Print debug information
    -s, --sql           Print all SQL statements executed
    -e                  Print every action done
```

もう 1 つの、特にオプショングループをプログラムから操作するときに利用できるメソッドがあります:

OptionParser.**get_option_group**(*opt_str*)

短いオプション文字列もしくは長いオプション文字列 *opt_str* (例。'-o' 、'--option') が属する *OptionGroup* を返します。そのような *OptionGroup* が無い場合は、None を返します。

バージョン番号の出力

optparse では、使用法メッセージと同様にプログラムのバージョン文字列を出力できます。*OptionParser* の version 引数に文字列を渡します:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

%prog は usage と同じような展開を受けます。その他にも version には何でも好きな内容を入れられます。version を指定した場合、*optparse* は自動的に --version オプションをパーザに渡します。コマンドライン中に --version が見つかると、*optparse* は version 文字列を展開して (%prog を置き換えて) 標準出力に出力し、プログラムを終了します。

例えば、/usr/bin/foo という名前のスクリプトなら:

```
$ /usr/bin/foo --version
foo 1.0
```

以下の 2 つのメソッドを、version 文字列を表示するために利用できます:

OptionParser.**print_version**(*file=None*)

現在のプログラムのバージョン (self.version) を *file* (デフォルト: stdout) へ表示します。*print_usage()* と同じく、self.version の中の全ての %prog が現在のプログラム名に置き換えられます。self.version が空文字列だだったり未定義だったときは何もしません。

OptionParser.**get_version**()

print_version() と同じですが、バージョン文字列を表示する代わりに返します。

optparse のエラー処理法

optparse を使う場合に気を付けなければならないエラーには、大きく分けてプログラマ側のエラーとユーザ側のエラーという二つの種類があります。プログラマ側のエラーの多くは、例えば不正なオプション文字列や定義されていないオプション属性の指定、あるいはオプション属性を指定し忘れるといった、誤った *OptionParser.add_option()* 呼び出しによるものです。こうした誤りは通常通りに処理されます。すなわち、例外 (optparse.OptionError や *TypeError*) を送出して、プログラムをクラッシュさせます。

もっと重要なのはユーザ側のエラーの処理です。というのも、ユーザの操作エラーというものはコードの安定性に関係なく起こるからです。*optparse* は、誤ったオプション引数の指定 (整数を引数にとるオプション -n に対して -n 4x と指定してしまうなど) や、引数を指定し忘れた場合 (-n が何らかの引数をとるオプションであるのに、-n が引数の末尾に来ている場合) といった、ユーザによるエラーを自動的に検出します。また、アプリケーション側で定義されたエラー条件が起きた場合、OptionParser.error() を呼び出してエラーを通知できます:

```
(options, args) = parser.parse_args()
...
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

いずれの場合にも *optparse* はエラーを同じやり方で処理します。すなわち、プログラムの使用法メッセージとエラーメッセージを標準エラー出力に出力して、終了ステータス 2 でプログラムを終了させます。

上に挙げた最初の例、すなわち整数を引数にとるオプションにユーザが 4x を指定した場合を考えてみましょう:

```
$ /usr/bin/foo -n 4x
Usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

値を全く指定しない場合には、以下のようになります:

```
$ /usr/bin/foo -n
Usage: foo [options]

foo: error: -n option requires an argument
```

optparse は、常にエラーを引き起こしたオプションについて説明の入ったエラーメッセージを生成するよう気を配ります; 従って、OptionParser.error() をアプリケーションコードから呼び出す場合にも、同じようなメッセージになるようにしてください。

optparse のデフォルトのエラー処理動作が気に入らないのなら、*OptionParser* をサブクラス化して、exit() かつ/または error() をオーバライドする必要があります。

全てをつなぎ合わせる

optparse を使ったスクリプトは、通常以下のようになります:

```
from optparse import OptionParser
...
def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                      help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")
    ...
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print("reading %s..." % options.filename)
    ...

if __name__ == "__main__":
    main()
```

36.1.3 リファレンスガイド

parser を作る

optparse を使う最初の一歩は OptionParser インスタンスを作ることです。

class optparse.OptionParser(...)

OptionParser のコンストラクタの引数はどれも必須ではありませんが、いくつものキーワード引数がオプションとして使えます。これらはキーワード引数として渡さなければなりません。すなわち、引数が宣言されている順番に頼ってはいけません。

usage (**デフォルト**: "%prog [options]") プログラムが間違った方法で実行されるかまたはヘルプオプションを付けて実行された場合に表示される使用法です。*optparse* は使用法の文字列を表示する際に %prog を os.path.basename(sys.argv[0]) (または prog キーワード引数が指定されていればその値) に展開します。使用法メッセージを抑制するためには特別な optparse.SUPPRESS_USAGE という値を指定します。

option_list (**デフォルト**: []) パーザに追加する Option オブジェクトのリストです。option_list の中のオプションは standard_option_list (OptionParser のサブクラスでセットされる可能性のあるクラス属性) の後に追加されますが、バージョンやヘルプのオプションよりは前になります。このオプションの使用は推奨されません。パーザを作成した後で、*add_option()* を使って追加してください。

option_class (**デフォルト**: optparse.Option) *add_option()* でパーザにオプションを追加すると

きに使用されるクラス。

version (**デフォルト**: None) ユーザがバージョンオプションを与えたときに表示されるバージョン文字列です。version に真の値を与えると、*optparse* は自動的に単独のオプション文字列 --version とともにバージョンオプションを追加します。部分文字列 %prog は usage と同様に展開されます。

conflict_handler (**デフォルト**: "error") オプション文字列が衝突するようなオプションがパーザに追加されたときにどうするかを指定します。**オプション間の衝突** 節を参照して下さい。

description (**デフォルト**: None) プログラムの概要を表す一段落のテキストです。*optparse* はユーザがヘルプを要求したときにこの概要を現在のターミナルの幅に合わせて整形し直して表示します (usage の後、オプションリストの前に表示されます)。

formatter (**デフォルト**: **新しい** IndentedHelpFormatter) ヘルプテキストを表示する際に使われる optparse.HelpFormatter のインスタンスです。*optparse* はこの目的のためにすぐ使えるクラスを二つ提供しています。IndentedHelpFormatter と TitledHelpFormatter がそれです。

add_help_option (**デフォルト**: True) もし真ならば、*optparse* はパーザにヘルプオプションを (オプション文字列 -h と --help とともに) 追加します。

prog usage や version の中の %prog を展開するときに os.path.basename(sys.argv[0]) の代わりに使われる文字列です。

epilog (**デフォルト**: None) オプションのヘルプの後に表示されるヘルプテキスト.

パーザへのオプション追加

パーザにオプションを加えていくにはいくつか方法があります。推奨するのは **チュートリアル** 節で示したような *OptionParser.add_option()* を使う方法です。add_option() は以下の二つのうちいずれかの方法で呼び出せます:

- (make_option() などが返す) Option インスタンスを渡します
- make_option() に (すなわち Option のコンストラクタに) 位置引数とキーワード引数の組み合わせを渡して、Option インスタンスを生成させます

もう一つの方法は、あらかじめ作成しておいた Option インスタンスからなるリストを、以下のようにして *OptionParser* のコンストラクタに渡すというものです:

```
option_list = [
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose"),
    ]
parser = OptionParser(option_list=option_list)
```

(make_option() は Option インスタンスを生成するファクトリ関数です; 現在のところ、この関数は Option のコンストラクタの別名にすぎません。*optparse* の将来のバージョンでは、Option を複数のクラスに分割

し、make_option() は適切なクラスを選んでインスタンスを生成するようになる予定です。従って、Option を直接インスタンス化しないでください。)

オプションの定義

各々の Option インスタンス、は -f や --file といった同義のコマンドラインオプションからなる集合を表現しています。一つの Option には任意の数のオプションを短い形式でも長い形式でも指定できます。ただし、少なくとも一つは指定しなければなりません。

正しい方法で Option インスタンスを生成するには、*OptionParser* の add_option() を使います。

OptionParser.**add_option**(*option*)
OptionParser.**add_option**(**opt_str, attr=value, ...*)

短い形式のオプション文字列を一つだけ持つようなオプションを生成するには次のようにします:

```
parser.add_option("-f", attr=value, ...)
```

また、長い形式のオプション文字列を一つだけ持つようなオプションの定義は次のようになります:

```
parser.add_option("--foo", attr=value, ...)
```

キーワード引数は新しい Option オブジェクトの属性を定義します。オプションの属性のうちでもっとも重要なのは *action* です。この属性は、他のどの属性と関連があるか、そしてどの属性が必要かに大きく作用します。関係のないオプション属性を指定したり、必要な属性を指定し忘れたりすると、*optparse* は誤りを解説した OptionError 例外を送出します。

コマンドライン上にあるオプションが見つかったときの *optparse* の振舞いを決定しているのは **アクション** *(action)* です。*optparse* でハードコードされている標準的なアクションには以下のようなものがあります:

"store"　オプションの引数を保存します (デフォルトの動作です)

"store_const"　定数値を保存します

"store_true"　store True

"store_false"　store False

"append"　オプションの引数を指定のリストに追加します

"append_const"　オプションの引数をリストに追加します

"count"　指定のカウンタを 1 増やします

"callback"　指定の関数を呼び出します

"help"　全てのオプションとそのドキュメントの入った使用法メッセージを出力します

(アクションを指定しない場合、デフォルトは "store" になります。このアクションでは、*type* および *dest* オプション属性を指定できます。**標準的なオプション・アクション** を参照してください。)

すでにお分かりのように、ほとんどのアクションはどこかに値を保存したり、値を更新したりします。この目的のために、*optparse* は常に特別なオブジェクトを作り出し、それは通常 options と呼ばれます (optparse.Values のインスタンスになっています)。オプションの引数 (や、その他の様々な値) は、*dest* (保存先: destination) オプション属性に従って、*options* の属性として保存されます。

例えばこれを呼び出した場合

```
parser.parse_args()
```

optparse はまず options オブジェクトを生成します:

```
options = Values()
```

パーザ中で以下のようなオプションが定義されていて

```
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
```

パーズしたコマンドラインに以下のいずれかが入っていた場合:

```
-ffoo
-f foo
--file=foo
--file foo
```

optparse はこのオプションを見つけて、以下と同等の処理を行います

```
options.filename = "foo"
```

type および *dest* オプション属性は *action* と同じくらい重要ですが、**全ての** オプションで意味をなすのは *action* だけなのです。

オプション属性

以下のオプション属性は *OptionParser.add_option()* へのキーワード引数として渡すことができます。特定のオプションに無関係なオプション属性を渡した場合、または必須のオプションを渡しそこなった場合、*optparse* は OptionError を送出します。

Option.action

(デフォルト: "store")

このオプションがコマンドラインにあった場合に *optparse* に何をさせるかを決めます。取りうるオプションについては こちら を参照してください。

Option.type

(デフォルト: "string")

このオプションに与えられる引数の型 (たとえば "string" や "int") です。取りうるオプションについては こちら を参照してください。

Option.**dest**
(デフォルト: オプション文字列を使う)

このオプションのアクションがある値をどこかに書いたり書き換えたりを意味する場合、これは *optparse* にその書く場所を教えます。詳しく言えば *dest* には *optparse* がコマンドラインを解析しながら組み立てる options オブジェクトの属性の名前を指定します。

Option.**default**
コマンドラインに指定がなかったときにこのオプションの対象に使われる値です。*OptionParser.set_defaults()* も参照してください。

Option.**nargs**
(デフォルト: 1)

このオプションがあったときに幾つの *type* 型の引数が消費されるべきかを指定します。1 より大きい場合、*optparse* は *dest* に値のタプルを格納します。

Option.**const**
定数を格納する動作のための、その定数です。

Option.**choices**
"choice" 型オプションに対してユーザが選べる選択肢となる文字列のリストです。

Option.**callback**
アクションが "callback" であるオプションに対し、このオプションがあったときに呼ばれる呼び出し可能オブジェクトです。呼び出し時に渡される引数の詳細については、**オプション処理コールバック** を参照してください。

Option.**callback_args**
Option.**callback_kwargs**
callback に渡される標準的な 4 つのコールバック引数の後ろに追加する、位置引数とキーワード引数。

Option.**help**
ユーザが *help* オプション (--help のような) を指定したときに表示される、使用可能な全オプションのリストの中のこのオプションに関する説明文です。説明文を提供しておかなければ、オプションは説明文なしで表示されます。オプションを隠すには特殊な値 optparse.SUPPRESS_HELP を使います。

Option.**metavar**
(デフォルト: オプション文字列を使う)

説明文を表示する際にオプションの引数の身代わりになるものです。例は **チュートリアル** 節を参照してください。

標準的なオプション・アクション

様々なオプション・アクションにはどれも互いに少しづつ異なった条件と作用があります。ほとんどのアクションに関連するオプション属性がいくつかあり、値を指定して *optparse* の挙動を操作できます。いくつかのアクションには必須の属性があり、必ず値を指定しなければなりません。

- "store" [関連: *type*, *dest*, *nargs*, *choices*]

  オプションの後には必ず引数が続きます。引数は *type* に従って値に変換されて *dest* に保存されます。*nargs* > 1 の場合、複数の引数をコマンドラインから取り出します。引数は全て *type* に従って変換され、*dest* にタプルとして保存されます。**標準のオプション型** 節を参照してください。

  choices を (文字列のリストかタプルで) 指定した場合、型のデフォルト値は "choice" になります。

  type を指定しない場合、デフォルトの値は "string" です。

  dest を指定しない場合、*optparse* は保存先を最初の長い形式のオプション文字列から導出します (例えば、--foo-bar は foo_bar になります)。長い形式のオプション文字列がない場合、*optparse* は最初の短い形式のオプションから保存先の変数名を導出します (-f は f になります)。

  以下はプログラム例です:

  ```
  parser.add_option("-f")
  parser.add_option("-p", type="float", nargs=3, dest="point")
  ```

  とすると、以下のようなコマンドライン

  ```
  -f foo.txt -p 1 -3.5 4 -fbar.txt
  ```

  を解析した場合、*optparse* は以下のように設定を行います

  ```
  options.f = "foo.txt"
  options.point = (1.0, -3.5, 4.0)
  options.f = "bar.txt"
  ```

- "store_const" [関連: *const*; 関連: *dest*]

  値 *const* を *dest* に保存します。

  以下はプログラム例です:

  ```
  parser.add_option("-q", "--quiet",
                    action="store_const", const=0, dest="verbose")
  parser.add_option("-v", "--verbose",
                    action="store_const", const=1, dest="verbose")
  parser.add_option("--noisy",
                    action="store_const", const=2, dest="verbose")
  ```

  とします。--noisy が見つかると、*optparse* は

```
options.verbose = 2
```

- "store_true" [関連: *dest*]

  A special case of "store_const" that stores True to *dest*.

- "store_false" [関連: *dest*]

  Like "store_true", but stores False.

  以下はプログラム例です:

```
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

- "append" [関連: *type*, *dest*, *nargs*, *choices*]

  このオプションの後ろには必ず引数が続きます。引数は *dest* のリストに追加されます。*dest* のデフォルト値を指定しなかった場合、*optparse* がこのオプションを最初にみつけた時点で空のリストを自動的に生成します。*nargs* > 1 の場合、複数の引数をコマンドラインから取り出し、長さ *nargs* のタプルを生成して *dest* に追加します。

  type および *dest* のデフォルト値は "store" アクションと同じです。

  以下はプログラム例です:

```
parser.add_option("-t", "--tracks", action="append", type="int")
```

  -t3 がコマンドライン上で見つかると、*optparse* は:

```
options.tracks = []
options.tracks.append(int("3"))
```

  その後、--tracks=4 が見つかると以下を実行します:

```
options.tracks.append(int("4"))
```

  append アクションは、オプションの現在の値の append メソッドを呼び出します。これは、どのデフォルト値も append メソッドを持っていなければならないことを意味します。また、デフォルト値が空でない場合、オプションの解析結果は、そのデフォルトの要素の後ろにコマンドラインからの値が追加されたものになる、ということも意味します:

```
>>> parser.add_option("--files", action="append", default=['~/.mypkg/defaults'])
>>> opts, args = parser.parse_args(['--files', 'overrides.mypkg'])
>>> opts.files
['~/.mypkg/defaults', 'overrides.mypkg']
```

- "append_const" [関連: *const*; 関連: *dest*]

  "store_const" と同様ですが、*const* の値は *dest* に追加 (append) されます。"append" の場合と

同じように *dest* のデフォルトは None ですがこのオプションを最初にみつけた時点で空のリストを自動的に生成します。

- "count" [関連: *dest*]

  dest に保存されている整数値をインクリメントします。*dest* は (デフォルトの値を指定しない限り) 最初にインクリメントを行う前にゼロに設定されます。

  以下はプログラム例です:

```
parser.add_option("-v", action="count", dest="verbosity")
```

  コマンドライン上で最初に -v が見つかると、*optparse* は:

```
options.verbosity = 0
options.verbosity += 1
```

  以後、-v が見つかるたびに

```
options.verbosity += 1
```

- "callback" [必須: *callback*; 関連: *type*, *nargs*, *callback_args*, *callback_kwargs*]

  callback に指定された関数を次のように呼び出します

```
func(option, opt_str, value, parser, *args, **kwargs)
```

  詳細は、**オプション処理コールバック** 節を参照してください。

- "help"

  現在のオプションパーザ内の全てのオプションに対する完全なヘルプメッセージを出力します。ヘルプメッセージは *OptionParser* のコンストラクタに渡した usage 文字列と、各オプションに渡した *help* 文字列から生成します。

  オプションに *help* 文字列が指定されていなくても、オプションはヘルプメッセージ中に列挙されます。オプションを完全に表示させないようにするには、特殊な値 optparse.SUPPRESS_HELP を使ってください。

  optparse は全ての *OptionParser* に自動的に *help* オプションを追加するので、通常自分で生成する必要はありません。

  以下はプログラム例です:

```
from optparse import OptionParser, SUPPRESS_HELP

# usually, a help option is added automatically, but that can
# be suppressed using the add_help_option argument
parser = OptionParser(add_help_option=False)

parser.add_option("-h", "--help", action="help")
```

(次のページに続く)

(前のページからの続き)

```
parser.add_option("-v", action="store_true", dest="verbose",
                  help="Be moderately verbose")
parser.add_option("--file", dest="filename",
                  help="Input file to read data from")
parser.add_option("--secret", help=SUPPRESS_HELP)
```

optparse がコマンドライン上に -h または --help を見つけると、以下のようなヘルプメッセージを標準出力に出力します (sys.argv[0] は "foo.py" だとします):

```
Usage: foo.py [options]

Options:
  -h, --help        Show this help message and exit
  -v                Be moderately verbose
  --file=FILENAME   Input file to read data from
```

ヘルプメッセージの出力後、*optparse* は sys.exit(0) でプロセスを終了します。

- "version"

  OptionParser に指定されているバージョン番号を標準出力に出力して終了します。バージョン番号は、実際には *OptionParser* の print_version() メソッドで書式化されてから出力されます。通常、*OptionParser* のコンストラクタに version 引数が指定されたときのみ関係のあるアクションです。*help* オプションと同様、*optparse* はこのオプションを必要に応じて自動的に追加するので、version オプションを作成することはほとんどないでしょう。

標準のオプション型

optparse には、"string", "int", "choice", "float", "complex" の 5 種類のビルトインのオプション型があります。新たなオプションの型を追加したければ、*optparse* **の拡張** 節を参照してください。

文字列オプションの引数はチェックや変換を一切受けません: コマンドライン上のテキストは保存先にそのまま保存されます (またはコールバックに渡されます)。

整数引数 ("int" 型) は次のように解析されます:

- 数が 0x から始まるならば、16 進数として読み取られます
- 数が 0 から始まるならば、8 進数として読み取られます
- 数が 0b から始まるならば、2 進数として読み取られます
- それ以外の場合、数は 10 進数として読み取られます

変換は適切な底 (2, 8, 10, 16 のどれか) とともに *int()* を呼び出すことで行なわれます。この変換が失敗した場合 *optparse* の処理も失敗に終わりますが、より役に立つエラーメッセージを出力します。

"float" および "complex" のオプション引数は直接 *float()* や *complex()* で変換されます。エラーは同様の扱いです。

"choice" オプションは "string" オプションのサブタイプです。*choices* オプションの属性 (文字列からなるシーケンス) には、利用できるオプション引数のセットを指定します。optparse.check_choice() はユーザの指定したオプション引数とマスタリストを比較して、無効な文字列が指定された場合には OptionValueError を送出します。

引数を解析する

OptionParser を作成してオプションを追加していく上で大事なポイントは、parse_args() メソッドの呼び出しです:

```
(options, args) = parser.parse_args(args=None, values=None)
```

ここで入力パラメータは

args　処理する引数のリスト (デフォルト: sys.argv[1:])

values　オプション引数を格納する optparse.Values のオブジェクト (デフォルト: 新しい Values のインスタンス) -- 既存のオブジェクトを指定した場合、オプションのデフォルトは初期化されません

であり、戻り値は

options　values に渡されたものと同じオブジェクト、または *optparse* によって生成された optparse.Values インスタンス

args　全てのオプションの処理が終わった後で残った位置引数

一番普通の使い方は一切キーワード引数を使わないというものです。values を指定した場合、それは繰り返される *setattr()* の呼び出し (大雑把に言うと保存される各オプション引数につき一回ずつ) で更新されていき、parse_args() で返されます。

parse_args() が引数リストでエラーに遭遇した場合、OptionParser の error() メソッドを適切なエンドユーザ向けのエラーメッセージとともに呼び出します。この呼び出しにより、最終的に終了ステータス 2 (伝統的な Unix におけるコマンドラインエラーの終了ステータス) でプロセスを終了させることになります。

オプション解析器への問い合わせと操作

オプションパーザのデフォルトの振る舞いは、ある程度カスタマイズすることができます。また、オプションパーザの中を調べることもできます。*OptionParser* は幾つかのヘルパーメソッドを提供しています:

OptionParser.**disable_interspersed_args**()

オプションで無い最初の引数を見つけた時点でパースを止めるように設定します。例えば、-a と -b が両方とも引数を取らないシンプルなオプションだったとすると、*optparse* は通常次の構文を受け付け:

```
prog -a arg1 -b arg2
```

それを次と同じように扱います

```
prog -a -b arg1 arg2
```

この機能を無効にしたいときは、*disable_interspersed_args()* メソッドを呼び出してください。古典的な Unix システムのように、最初のオプションでない引数を見つけたときにオプションの解析を止めるようになります。

別のコマンドを実行するコマンドをプロセッサを作成する際、別のコマンドのオプションと自身のオプションが混ざるのを防ぐために利用することができます。例えば、各コマンドがそれぞれ異なるオプションのセットを持つ場合などに有効です。

OptionParser.**enable_interspersed_args**()
: オプションで無い最初の引数を見つけてもパースを止めないように設定します。オプションとコマンド引数の順序が混ざっても良いようになります。これはデフォルトの動作です。

OptionParser.**get_option**(*opt_str*)
: オプション文字列 *opt_str* に対する Option インスタンスを返します。該当するオプションがなければ None を返します。

OptionParser.**has_option**(*opt_str*)
: *OptionParser* に (-q や --verbose のような) オプション *opt_str* がある場合、True を返します。

OptionParser.**remove_option**(*opt_str*)
: *OptionParser* に *opt_str* に対応するオプションがある場合、そのオプションを削除します。該当するオプションに他のオプション文字列が指定されていた場合、それらのオプション文字列は全て無効になります。*opt_str* がこの *OptionParser* オブジェクトのどのオプションにも属さない場合、*ValueError* を送出します。

オプション間の衝突

注意が足りないと、衝突するオプションを定義してしまうことがあります:

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

(とりわけ、*OptionParser* から標準的なオプションを備えた自前のサブクラスを定義してしまった場合にはよく起きます。)

ユーザがオプションを追加するたびに、*optparse* は既存のオプションとの衝突がないかチェックします。何らかの衝突が見付かると、現在設定されている衝突処理メカニズムを呼び出します。衝突処理メカニズムはコンストラクタ中で呼び出せます:

```
parser = OptionParser(..., conflict_handler=handler)
```

個別にも呼び出せます:

```
parser.set_conflict_handler(handler)
```

衝突時の処理をおこなうハンドラ (handler) には、以下のものが利用できます:

"error" (デフォルト) オプション間の衝突をプログラム上のエラーとみなし、OptionConflictError を送出します

"resolve" オプション間の衝突をインテリジェントに解決します (下記参照)

一例として、衝突をインテリジェントに解決する *OptionParser* を定義し、衝突を起こすようなオプションを追加してみましょう:

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

この時点で、*optparse* はすでに追加済のオプションがオプション文字列 -n を使っていることを検出します。conflict_handler が "resolve" なので、*optparse* は既に追加済のオプションリストの方から -n を除去して問題を解決します。従って、-n の除去されたオプションは --dry-run だけでしか有効にできなくなります。ユーザがヘルプ文字列を要求した場合、問題解決の結果を反映したメッセージが出力されます:

```
Options:
  --dry-run     do no harm
  ...
  -n, --noisy   be noisy
```

これまでに追加したオプション文字列を跡形もなく削り去り、ユーザがそのオプションをコマンドラインから起動する手段をなくせます。この場合、*optparse* はオプションを完全に除去してしまうので、こうしたオプションはヘルプテキストやその他のどこにも表示されなくなります。例えば、現在の *OptionParser* の場合、以下の操作:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

を行った時点で、最初の -n/--dry-run オプションはもはやアクセスできなくなります。このため、*optparse* はオプションを消去してしまい、ヘルプテキストだけが残ります:

```
Options:
  ...
  -n, --noisy   be noisy
  --dry-run     new dry-run option
```

クリーンアップ

OptionParser インスタンスはいくつかの循環参照を抱えています。このことは Python のガーベジコレクタにとって問題になるわけではありませんが、使い終わった OptionParser に対して destroy() を呼び出すことでこの循環参照を意図的に断ち切るという方法を選ぶこともできます。この方法は特に長時間実行するアプリケーションで OptionParser から大きなオブジェクトグラフが到達可能になっているような場合に有用です。

その他のメソッド

OptionParser にはその他にも幾つかの公開されたメソッドがあります:

OptionParser.**set_usage**(*usage*)

上で説明したコンストラクタの usage キーワード引数での規則に従った使用法の文字列をセットします。None を渡すとデフォルトの使用法文字列が使われるようになり、optparse.SUPPRESS_USAGE によって使用法メッセージを抑制できます。

OptionParser.**print_usage**(*file=None*)

現在のプログラムの使用法メッセージ (self.usage) を *file* (デフォルト: stdout) に表示します。self.usage 内にある全ての %prog という文字列は現在のプログラム名に置換されます。self.usage が空もしくは未定義の時は何もしません。

OptionParser.**get_usage**()

print_usage() と同じですが、使用法メッセージを表示する代わりに文字列として返します。

OptionParser.**set_defaults**(*dest=value, ...*)

幾つかの保存先に対してデフォルト値をまとめてセットします。*set_defaults()* を使うのは複数のオプションにデフォルト値をセットする好ましいやり方です。複数のオプションが同じ保存先を共有することがあり得るからです。たとえば幾つかの "mode" オプションが全て同じ保存先をセットするものだったとすると、どのオプションもデフォルトをセットすることができ、しかし最後に指定したものだけが有効になります:

```
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced",
                  default="novice")    # overridden below
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice",
                  default="advanced")  # overrides above setting
```

こうした混乱を避けるために *set_defaults()* を使います:

```
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice")
```

36.1.4 オプション処理コールバック

optparse の組み込みのアクションや型が望みにかなったものでない場合、二つの選択肢があります: 一つは *optparse* の拡張、もう一つは callback オプションの定義です。*optparse* の拡張は汎用性に富んでいますが、単純なケースに対していささか大げさでもあります。大体は簡単なコールバックで事足りるでしょう。

callback オプションの定義は二つのステップからなります:

- "callback" アクションを使ってオプション自体を定義する

- コールバックを書く。コールバックは少なくとも後で説明する 4 つの引数をとる関数 (またはメソッド) でなければなりません

callback オプションの定義

callback オプションを最も簡単に定義するには、*OptionParser.add_option()* メソッドを使います。*action* の他に指定しなければならない属性は callback すなわちコールバックする関数自体です:

```
parser.add_option("-c", action="callback", callback=my_callback)
```

callback は関数 (または呼び出し可能オブジェクト) なので、callback オプションを定義する時にはあらかじめ my_callback() を定義しておかなければなりません。この単純なケースでは、*optparse* は -c が何らかの引数をとるかどうか判別できず、通常は -c が引数を伴わないことを意味します --- 知りたいことはただ単に -c がコマンドライン上に現れたどうかだけです。とはいえ、場合によっては、自分のコールバック関数に任意の個数のコマンドライン引数を消費させたいこともあるでしょう。これがコールバック関数をトリッキーなものにしています; これについてはこの節の後の方で説明します。

optparse は常に四つの引数をコールバックに渡し、その他には *callback_args* および *callback_kwargs* で指定した追加引数しか渡しません。従って、最小のコールバック関数シグネチャは:

```
def my_callback(option, opt, value, parser):
```

コールバックの四つの引数については後で説明します。

callback オプションを定義する場合には、他にもいくつかオプション属性を指定できます:

type 他で使われているのと同じ意味です: "store" や "append" アクションの時と同じく、この属性は *optparse* に引数を一つ消費して *type* で指定した型に変換させます。*optparse* は変換後の値をどこかに保存する代わりにコールバック関数に渡します。

nargs これも他で使われているのと同じ意味です: このオプションが指定されていて、かつ nargs > 1 である場合、*optparse* は *nargs* 個の引数を消費します。このとき各引数は *type* 型に変換できなければなりません。変換後の値はタプルとしてコールバックに渡されます。

callback_args その他の位置引数からなるタプルで、コールバックに渡されます

callback_kwargs その他のキーワード引数からなる辞書で、コールバックに渡されます

コールバック関数はどのように呼び出されるか

コールバックは全て以下の形式で呼び出されます:

```
func(option, opt_str, value, parser, *args, **kwargs)
```

ここでは:

option コールバックを呼び出している Option のインスタンスです

opt_str　は、コールバック呼び出しのきっかけとなったコマンドライン上のオプション文字列です。(長い形式のオプションに対する省略形が使われている場合、opt_str は完全な、正式な形のオプション文字列となります --- 例えば、ユーザが --foobar の短縮形として --foo をコマンドラインに入力した時には、opt_str は "--foobar" となります。)

value　オプションの引数で、コマンドライン上に見つかったものです。*optparse* は、*type* が設定されている場合、単一の引数しかとりません。value の型はオプションの型として指定された型になります。このオプションに対する *type* が None である (引数なしの) 場合、value は None になります。*nargs* > 1 であれば、value は適切な型をもつ値のタプルになります。

parser　現在のオプション解析の全てを駆動している *OptionParser* インスタンスです。この変数が有用なのは、この値を介してインスタンス属性としていくつかの興味深いデータにアクセスできるからです:

parser.largs　現在放置されている引数、すなわち、すでに消費されたものの、オプションでもオプション引数でもない引数からなるリストです。parser.largs は自由に変更でき、たとえば引数を追加したりできます (このリストは args 、すなわち parse_args() の二つ目の戻り値になります)

parser.rargs　現在残っている引数、すなわち、opt_str および value があれば除き、それ以外の引数が残っているリストです。parser.rargs は自由に変更でき、例えばさらに引数を消費したりできます。

parser.values　オプションの値がデフォルトで保存されるオブジェクト (optparse.OptionValues のインスタンス) です。この値を使うと、コールバック関数がオプションの値を記憶するために、他の *optparse* と同じ機構を使えるようにするため、グローバル変数や閉包 (closure) を台無しにしないので便利です。コマンドライン上にすでに現れているオプションの値にもアクセスできます。

args　*callback_args* オプション属性で与えられた任意の位置引数からなるタプルです。

kwargs　*callback_kwargs* オプション属性で与えられた任意のキーワード引数からなるタプルです。

コールバック中で例外を送出する

オプション自体か、あるいはその引数に問題がある場合、コールバック関数は OptionValueError を送出しなければなりません。*optparse* はこの例外をとらえてプログラムを終了させ、ユーザが指定しておいたエラーメッセージを標準エラー出力に出力します。エラーメッセージは明確、簡潔かつ正確で、どのオプションに誤りがあるかを示さなければなりません。さもなければ、ユーザは自分の操作のどこに問題があるかを解決するのに苦労することになります。

コールバックの例 1: ありふれたコールバック

引数をとらず、発見したオプションを単に記録するだけのコールバックオプションの例を以下に示します:

```
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True

parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

もちろん、"store_true" アクションを使っても実現できます。

コールバックの例 2: オプションの順番をチェックする

もう少し面白みのある例を示します: この例では、-b を発見して、その後で -a がコマンドライン中に現れた場合にはエラーになります。

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1
...
parser.add_option("-a", action="callback", callback=check_order)
parser.add_option("-b", action="store_true", dest="b")
```

コールバックの例 3: オプションの順番をチェックする (汎用的)

このコールバック (フラグを立てるが、-b が既に指定されていればエラーになる) を同様の複数のオプションに対して再利用したければ、もう少し作業する必要があります: エラーメッセージとセットされるフラグを一般化しなければなりません。

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("-a", action="callback", callback=check_order, dest='a')
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback", callback=check_order, dest='c')
```

コールバックの例 4: 任意の条件をチェックする

もちろん、単に定義済みのオプションの値を調べるだけにとどまらず、コールバックには任意の条件を入れられます。例えば、満月でなければ呼び出してはならないオプションがあるとしましょう。やらなければならないことはこれだけです:

```
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
```

(次のページに続く)

(前のページからの続き)

```
        raise OptionValueError("%s option invalid when moon is full"
                               % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("--foo",
                  action="callback", callback=check_moon, dest="foo")
```

(is_moon_full() の定義は読者への課題としましょう。)

コールバックの例 5: 固定引数

決まった数の引数をとるようなコールバックオプションを定義するなら、問題はやや興味深くなってきます。引数をとるようコールバックに指定するのは、"store" や "append" オプションの定義に似ています。*type* を定義していれば、そのオプションは引数を受け取ったときに該当する型に変換できなければなりません。さらに *nargs* を指定すれば、オプションは *nargs* 個の引数を受け取ります。

標準の "store" アクションをエミュレートする例を以下に示します:

```
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
...
parser.add_option("--foo",
                  action="callback", callback=store_value,
                  type="int", nargs=3, dest="foo")
```

optparse は 3 個の引数を受け取り、それらを整数に変換するところまで面倒をみてくれます。ユーザは単にそれを保存するだけです。(他の処理もできます; いうまでもなく、この例にはコールバックは必要ありません)

コールバックの例 6: 可変個の引数

あるオプションに可変個の引数を持たせたいと考えているなら、問題はいささか手強くなってきます。この場合、*optparse* では該当する組み込みのオプション解析機能を提供していないので、自分でコールバックを書かなければなりません。さらに、*optparse* が普段処理している、伝統的な Unix コマンドライン解析における難題を自分で解決しなければなりません。とりわけ、コールバック関数では引数が裸の -- や - の場合における慣習的な処理規則:

- either -- or - can be option arguments
- 裸の -- (何らかのオプションの引数でない場合): コマンドライン処理を停止し、-- を無視します
- 裸の - (何らかのオプションの引数でない場合): コマンドライン処理を停止しますが、- は残します (parser.largs に追加します)

オプションが可変個の引数をとるようにさせたいなら、いくつかの巧妙で厄介な問題に配慮しなければなりません。どういう実装をとるかは、アプリケーションでどのようなトレードオフを考慮するかによります (このため、*optparse* では可変個の引数に関する問題を直接的に取り扱わないのです)。

とはいえ、可変個の引数をもつオプションに対するスタブ (stub、仲介インタフェース) を以下に示しておきます:

```
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []

    def floatable(str):
        try:
            float(str)
            return True
        except ValueError:
            return False

    for arg in parser.rargs:
        # stop on --foo like options
        if arg[:2] == "--" and len(arg) > 2:
            break
        # stop on -a, but not on -3 or -3.0
        if arg[:1] == "-" and len(arg) > 1 and not floatable(arg):
            break
        value.append(arg)

    del parser.rargs[:len(value)]
    setattr(parser.values, option.dest, value)

...
parser.add_option("-c", "--callback", dest="vararg_attr",
                  action="callback", callback=vararg_callback)
```

36.1.5 optparse の拡張

optparse がコマンドラインオプションをどのように解釈するかを決める二つの重要な要素はそれぞれのオプションのアクションと型なので、拡張の方向は新しいアクションと型を追加することになると思います。

新しい型の追加

新しい型を追加するためには、*optparse* の Option クラスのサブクラスを自身で定義する必要があります。このクラスには *optparse* における型を定義する一対の属性があります。それは *TYPES* と *TYPE_CHECKER* です。

Option.TYPES
: *TYPES* は型名のタプルです。新しく作るサブクラスでは、タプル *TYPES* を単純に標準のものを利用して新しく定義すると良いでしょう。

Option.TYPE_CHECKER
: TYPE_CHECKER は辞書で型名を型チェック関数に対応付けるものです。型チェック関数は以下のようなシグネチャを持ちます:

```
def check_mytype(option, opt, value)
```

ここで option は Option のインスタンスであり、opt はオプション文字列 (たとえば -f) で、value は望みの型としてチェックされ変換されるべくコマンドラインで与えられる文字列です。check_mytype() は想定されている型 mytype のオブジェクトを返さなければなりません。型チェック関数から返される値は OptionParser.parse_args() で返される OptionValues インスタンスに収められるか、またはコールバックに value パラメータとして渡されます。

型チェック関数は何か問題に遭遇したら OptionValueError を送出しなければなりません。OptionValueError は文字列一つを引数に取り、それはそのまま *OptionParser* の error() メソッドに渡され、そこでプログラム名と文字列 "error:" が前置されてプロセスが終了する前に stderr に出力されます。

馬鹿馬鹿しい例ですが、Python スタイルの複素数を解析する "complex" オプション型を作ってみせることにします。(*optparse* 1.3 が複素数のサポートを組み込んでしまったため以前にも増して馬鹿らしくなりましたが、気にしないでください。)

最初に必要な import 文を書きます:

```
from copy import copy
from optparse import Option, OptionValueError
```

まずは型チェック関数を定義しなければなりません。これは後で (これから定義する Option のサブクラスの *TYPE_CHECKER* クラス属性の中で) 参照されることになります:

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

最後に Option のサブクラスです:

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(もしここで *Option.TYPE_CHECKER* に *copy()* を適用しなければ、*optparse* の Option クラスの *TYPE_CHECKER* 属性をいじってしまうことになります。Python の常として、良いマナーと常識以外にそうすることを止めるものはありません。)

これだけです! もう新しいオプション型を使うスクリプトを他の *optparse* に基づいたスクリプトとまるで同じように書くことができます。ただし、OptionParser に Option でなく MyOption を使うように指示しなければなければなりません:

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

別のやり方として、オプションリストを構築して OptionParser に渡すという方法もあります。add_option() を上でやったように使わないならば、OptionParser にどのクラスを使うのか教える必要はありません:

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

新しいアクションの追加

新しいアクションの追加はもう少しトリッキーです。というのも *optparse* が使っている二つのアクションの分類を理解する必要があるからです:

"store" **アクション** *optparse* が値を現在の OptionValues の属性に格納することになるアクションです。この種類のオプションは Option のコンストラクタに *dest* 属性を与えることが要求されます。

"typed" **アクション** コマンドラインから引数を受け取り、それがある型であることが期待されているアクションです。もう少しはっきり言えば、その型に変換される文字列を受け取るものです。この種類のオプションは Option のコンストラクタに *type* 属性を与えることが要求されます。

この分類には重複する部分があります。デフォルトの "store" アクションには "store", "store_const", "append", "count" などがありますが、デフォルトの "typed" オプションは "store", "append", "callback" の三つです。

アクションを追加する際に、以下の Option のクラス属性 (全て文字列のリストです) の中の少なくとも一つに付け加えることでそのアクションを分類する必要があります:

Option.ACTIONS
: 全てのアクションは ACTIONS にリストされていなければなりません。

Option.STORE_ACTIONS
: "store" アクションはここにもリストされます。

Option.TYPED_ACTIONS
: "typed" アクションはここにもリストされます。

Option.ALWAYS_TYPED_ACTIONS
: 型を取るアクション (つまりそのオプションが値を取る) はここにもリストされます。このことの唯一の効果は *optparse* が、型の指定が無くアクションが *ALWAYS_TYPED_ACTIONS* のリストにあるオプションに、デフォルト型 "string" を割り当てるということだけです。

実際に新しいアクションを実装するには、Option の take_action() メソッドをオーバライドしてそのアクションを認識する場合分けを追加しなければなりません。

例えば、"extend" アクションというのを追加してみましょう。このアクションは標準的な "append" アクションと似ていますが、コマンドラインから一つだけ値を読み取って既存のリストに追加するのではなく、複数の値をコンマ区切りの文字列として読み取ってそれらで既存のリストを拡張します。すなわち、もし --names が "string" 型の "extend" オプションだとすると、次のコマンドライン

```
--names=foo,bar --names blah --names ding,dong
```

の結果は次のリストになります

```
["foo", "bar", "blah", "ding", "dong"]
```

再び Option のサブクラスを定義します:

```
class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)

    def take_action(self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(
                self, action, dest, opt, value, values, parser)
```

注意すべきは次のようなところです:

- "extend" はコマンドラインの値を予期していると同時にその値をどこかに格納しますので、*STORE_ACTIONS* と *TYPED_ACTIONS* の両方に入ります。
- *optparse* が "extend" アクションに "string" 型を割り当てるように "extend" アクションは *ALWAYS_TYPED_ACTIONS* にも入れてあります。
- MyOption.take_action() にはこの新しいアクション一つの扱いだけを実装してあり、他の標準的な *optparse* のアクションについては Option.take_action() に制御を戻すようにしてあります。
- values は optparse_parser.Values クラスのインスタンスであり、非常に有用な ensure_value() メソッドを提供しています。ensure_value() は本質的に安全弁付きの *getattr()* です。次のように呼び出します

  ```
  values.ensure_value(attr, value)
  ```

  values に attr 属性が無いか None だった場合に、ensure_value() は最初に value をセットし、それから value を返します。この振る舞いは "extend", "append", "count" のように、データを変数に集積し、またその変数がある型 (最初の二つはリスト、最後のは整数) であると期待されるアクションを作るのにとても使い易いものです。ensure_value() を使えば、作ったアクションを使うスクリプトはオプションに保存先にデフォルト値をセットすることに煩わされずに済みます。デフォルトを None にしておけば ensure_value() がそれが必要になったときに適当な値を返してくれます。

36.2 imp --- import 内部へのアクセス

ソースコード: Lib/imp.py

バージョン 3.4 で非推奨: *imp* モジュールは *importlib* を後継として廃止されました。

このモジュールは import 文を実装するために使われているメカニズムへのインターフェイスを提供します。次の定数と関数が定義されています:

imp.**get_magic**()

バイトコンパイルされたコードファイル (.pyc ファイル) を認識するために使われるマジック文字列値を返します。(この値は Python の各バージョンで異なります。)

バージョン 3.4 で非推奨: 代わりに *importlib.util.MAGIC_NUMBER* を使用してください。

imp.**get_suffixes**()

3 要素のタプルのリストを返します。それぞれのタプルは特定の種類のモジュールを説明しています。各タプルは (suffix, mode, type) という形式です。ここで、*suffix* は探すファイル名を作るためにモジュール名に追加する文字列です。そのファイルをオープンするために、*mode* は組み込み *open()* 関数へ渡されるモード文字列です (これはテキストファイル対しては 'r' 、バイナリファイルに対しては 'rb' となります)。*type* はファイル型で、以下で説明する値 *PY_SOURCE*, *PY_COMPILED* あるいは、*C_EXTENSION* の一つを取ります。

バージョン 3.3 で非推奨: 代わりに *importlib.machinery* で定義された定数を使ってください。

imp.**find_module**(*name*[, *path*])

モジュール *name* を見つけようとします。*path* が省略されるか None ならば、sys.path によって与えられるディレクトリ名のリストが検索されます。しかし、最初にいくつか特別な場所を検索します。まず、所定の名前をもつ組み込みモジュール (*C_BUILTIN*) を見つけようとします。それから、フリーズされたモジュール (*PY_FROZEN*)、そしていくつかのシステムでは他の場所が同様に検索されます (Windows では、特定のファイルを指すレジストリの中を見ます)。

それ以外の場合、*path* はディレクトリ名のリストでなければなりません。上の *get_suffixes()* が返す拡張子のいずれかを伴ったファイルを各ディレクトリの中で検索します。リスト内の有効でない名前は黙って無視されます (しかし、すべてのリスト項目は文字列でなければなりません)。

検索が成功すれば、戻り値は 3 要素のタプル (file, pathname, description) です:

file は先頭に位置合わせされたオープン **ファイルオブジェクト** で、*pathname* は見つかったファイルのパス名です。そして、*description* は *get_suffixes()* が返すリストに含まれているような 3 要素のタプルで、見つかったモジュールの種類を説明しています。

モジュールが組み込みモジュールか frozen なモジュールの場合、*file* と *pathname* のどちらも None になり、*description* タプルの要素の接尾辞 (拡張子) とモードは空文字列になります。モジュール型は上の括弧の中に示されます。検索に失敗した場合は、*ImportError* が発生します。他の例外は引数または環境に問題があることを示しています。

モジュールがパッケージならば、*file* は None で、*pathname* はパッケージのパスで *description* タプルの最後の項目は *PKG_DIRECTORY* です。

この関数は階層的なモジュール名 (ドットを含む名前) を扱いません。*P.M* 、すなわちパッケージ *P* のサブモジュール *M* を見つけるためには、まず *find_module()* と *load_module()* を使用してパッケージ *P* を見つけてロードして、次に `P.__path__` を *path* 引数にして *find_module()* を呼び出してください。もし *P* 自体がドット付きの名前を持つ場合、このレシピを再帰的に適用してください。

バージョン 3.3 で非推奨: Python 3.3 との互換性が不要であれば、*importlib.util.find_spec()* を使用してください, 互換性が必要な場合は *importlib.find_loader()* を使用してください。前者の使用例は、*importlib* ドキュメントの:ref:*importlib-examples* セクションを参照してください。

`imp.load_module`(*name*, *file*, *pathname*, *description*)

find_module() を使って (あるいは、互換性のある結果を作り出す検索を行って) 以前見つけたモジュールをロードします。この関数はモジュールをインポートするという以上のことを行います: モジュールが既にインポートされているならば、リロードします! *name* 引数は (これがパッケージのサブモジュールならばパッケージ名を含む) 完全なモジュール名を示します。*file* 引数はオープンしたファイルで、*pathname* は対応するファイル名です。モジュールがパッケージであるかファイルからロードされようとしていないとき、これらはそれぞれ `None` と `''` であっても構いません。*get_suffixes()* が返すように *description* 引数はタプルで、どの種類のモジュールがロードされなければならないかを説明するものです。

ロードが成功したならば、戻り値はモジュールオブジェクトです。そうでなければ、例外 (たいていは *ImportError*) が発生します。

重要: *file* 引数が `None` でなければ、例外が発生した場合でも呼び出し側にはそれを閉じる責任があります。これを行うには、`try` ... `finally` 文を使うことが最も良いです。

バージョン 3.3 で非推奨: もし以前は *imp.find_module()* と一緒に使っていたのなら、*importlib.import_module()* を使うことを検討してください。そうでなければ、*imp.find_module()* に対して選択した代替手段によって返されるローダーを使用してください。もし *imp.load_module()* とそれに関連する関数を path 引数付きで直接呼んでいたのなら、*importlib.util.spec_from_file_location()* と *importlib.util.module_from_spec()* を組み合わせて使ってください。様々な手法の詳細については *importlib* ドキュメントの **使用例** 節を参照してください。

`imp.new_module`(*name*)

name という名前の新しい空モジュールオブジェクトを返します。このオブジェクトは `sys.modules` に挿入され **ません** 。

バージョン 3.4 で非推奨: 代わりに:func:*importlib.util.module_from_spec* を使用してください。

`imp.reload`(*module*)

すでにインポートされた *module* を再解釈し、再初期化します。引数はモジュールオブジェクトでなければならないので、予めインポートに成功していなければなりません。この関数はモジュールのソースコードファイルを外部エディタで編集して、Python インタプリタから離れることなく新しいバージョンを試したい際に有効です。戻り値は (*module* 引数と同じ) モジュールオブジェクトです。

`reload(module)` を実行すると、以下の処理が行われます:

- Python モジュールのコードは再コンパイルされ、モジュールレベルのコードは再度実行されます。モジュールの辞書中にある、何らかの名前に結び付けられたオブジェクトを新たに定義しま

す。拡張モジュール中の init 関数が二度呼び出されることはありません。

- Python における他のオブジェクトと同様、以前のオブジェクトのメモリ領域は、参照カウントがゼロにならないかぎり再利用されません。
- モジュール名前空間内の名前は新しいオブジェクト (または更新されたオブジェクト) を指すよう更新されます。
- 以前のオブジェクトが (外部の他のモジュールなどからの) 参照を受けている場合、それらを新たなオブジェクトに再束縛し直すことはないので、必要なら自分で名前空間を更新しなければなりません。

いくつか補足説明があります:

モジュールが再ロードされた際、その辞書 (モジュールのグローバル変数を含みます) はそのまま残ります。名前の再定義を行うと、以前の定義を上書きするので、一般的には問題はありません。新たなバージョンのモジュールが古いバージョンで定義された名前を定義していない場合、古い定義がそのまま残ります。辞書がグローバルテーブルやオブジェクトのキャッシュを維持していれば、この機能をモジュールを有効性を引き出すために使うことができます --- つまり、try 文を使えば、必要に応じてテーブルがあるかどうかをテストし、その初期化を飛ばすことができます:

```
try:
    cache
except NameError:
    cache = {}
```

ビルトインのモジュールや動的にロードされたモジュールをリロードすることは、*sys*, *__main__*, *builtins* を除いて一般にはそれほど有用ではありませんが、合法です。しかし、多くの場合、拡張モジュールは二度以上初期化されるようには作られておらず、リロードされた時に無作為な方法で失敗するかもしれません。

一方のモジュールが from ... import ... を使って、オブジェクトを他方のモジュールからインポートしているなら、他方のモジュールを *reload()* で呼び出しても、そのモジュールからインポートされたオブジェクトを再定義することはできません --- この問題を回避する一つの方法は、from 文を再度実行することで、もう一つの方法は from 文の代わりに import と限定的な名前 (*module.**name*) を使うことです。

あるモジュールがクラスのインスタンスを生成している場合、そのクラスを定義しているモジュールの再ロードはそれらインスタンスのメソッド定義に影響しません --- それらは古いクラス定義を使い続けます。これは派生クラスの場合でも同じです。

バージョン 3.3 で変更: リロードされるモジュール上で、__name__ だけでなく __name__ と __loader__ の両方が定義されていることに依存します。

バージョン 3.4 で非推奨: 代わりに *importlib.reload()* を使用してください。

以下は、**PEP 3147** のバイトコンパイルされたファイルパスを扱うために便利な関数です。

バージョン 3.2 で追加.

imp.cache_from_source(*path*, *debug_override=None*)

ソース *path* に関連付けられたバイトコンパイルされたファイルの **PEP 3147** パスを返します。例えば、*path* が /foo/bar/baz.py なら、Python 3.2 の場合返り値は /foo/bar/__pycache__/baz.cpython-32.pyc になります。cpython-32 という文字列は、現在のマジックタグから得られます (マジックタグについては *get_tag()* を参照; sys.implementation.cache_tag が未定義なら *NotImplementedError* が送出されます)。*debug_override* に True あるいは False を渡すことによって、__debug__ システム値をオーバーライドして最適化されたバイトコードを得ることができます。

path は存在している必要はありません。

バージョン 3.3 で変更: sys.implementation.cache_tag が None の場合、*NotImplementedError* が上げられます。

バージョン 3.4 で非推奨: 代わりに *importlib.util.cache_from_source()* を使用してください。

バージョン 3.5 で変更: *debug_override* 引数は .pyo ファイルを作成することはもうありません。

imp.source_from_cache(*path*)

PEP 3147 ファイル名への *path* が与えられると、関連するソースコードのファイルパスを返します。例えば、*path* が /foo/bar/__pycache__/baz.cpython-32.pyc なら、返されるパスは /foo/bar/baz.py になります。*path* は存在する必要はありませんが、**PEP 3147** フォーマットに一致しない場合は *ValueError* が送出されます。sys.implementation.cache_tag が定義されていない場合、*NotImplementedError* が送出されます。

バージョン 3.3 で変更: sys.implementation.cache_tag が定義されていない場合、*NotImplementedError* が上げられます。

バージョン 3.4 で非推奨: 代わりに *importlib.util.source_from_cache()* を使用してください。

imp.get_tag()

get_magic() によって返されるのと同じ、このバージョンの Python のマジックナンバーと一致する **PEP 3147** のマジックタグ文字列を返します。

バージョン 3.4 で非推奨: Python 3.3 からは sys.implementation.cache_tag を直接使ってください。

以下の関数はインポートシステムの内部ロックメカニズムとのやりとりをサポートします。インポートのロックセマンティクスはリリース毎に変わる可能性のある実装詳細です。ただし、Python は循環インポートがデッドロックなしで動作することを保証しています。

imp.lock_held()

現在グローバルなインポートロックが保持されている場合 True を返し、そうでなければ False を返します。スレッドのないプラットフォームでは常に False を返します。

スレッドを持つプラットフォームでは、まずインポートを実行するスレッドがグローバルなインポートロックを保持し、次にインポートの残りの部分を実行するためにモジュール単位のロックをセットアップします。これは、他のスレッドが同じモジュールをインポートするのをオリジナルのインポートが完了するまでブロックして、他のスレッドがオリジナルのスレッドによって構築された不完全なモジュー

ルオブジェクトを見てしまわないようにします。循環インポートに対しては例外が発生します。これは、そのようなモジュールを構築するためには、どこかの時点で不完全なモジュールオブジェクトを露出しなければならないためです。

バージョン 3.3 で変更: ロックスキームは、大部分がモジュール毎のロックに変わりました。グローバルなインポートロックは、モジュール毎のロックを初期化するようないくつかのクリティカルタスクのために維持されます。

バージョン 3.4 で非推奨.

imp.acquire_lock()

現在のスレッドに対するインタープリタのグローバルなインポートロックを獲得します。このロックは、モジュールをインポートする際にスレッドセーフ性を保証するために、インポートフックによって使用されるべきです。

一旦スレッドがインポートロックを取得したら、その同じスレッドはブロックされることなくそのロックを再度取得できます。スレッドはロックを取得するのと同じだけ解放しなければなりません。

スレッドのないプラットホームではこの関数は何もしません。

バージョン 3.3 で変更: ロックスキームは、大部分がモジュール毎のロックに変わりました。グローバルなインポートロックは、モジュール毎のロックを初期化するようないくつかのクリティカルタスクのために維持されます。

バージョン 3.4 で非推奨.

imp.release_lock()

インタープリタのグローバルなインポートロックを解放します。スレッドのないプラットフォームでは何もしません。

バージョン 3.3 で変更: ロックスキームは、大部分がモジュール毎のロックに変わりました。グローバルなインポートロックは、モジュール毎のロックを初期化するようないくつかのクリティカルタスクのために維持されます。

バージョン 3.4 で非推奨.

整数値をもつ次の定数はこのモジュールの中で定義されており、*find_module()* の検索結果を表すために使われます。

imp.PY_SOURCE

ソースファイルとしてモジュールが発見された。

バージョン 3.3 で非推奨.

imp.PY_COMPILED

コンパイルされたコードオブジェクトファイルとしてモジュールが発見された。

バージョン 3.3 で非推奨.

imp.C_EXTENSION

動的にロード可能な共有ライブラリとしてモジュールが発見された。

バージョン 3.3 で非推奨.

imp.PKG_DIRECTORY
: パッケージディレクトリとしてモジュールが発見された。

バージョン 3.3 で非推奨.

imp.C_BUILTIN
: モジュールが組み込みモジュールとして発見された。

バージョン 3.3 で非推奨.

imp.PY_FROZEN
: モジュールが frozen モジュールとして発見された。

バージョン 3.3 で非推奨.

class imp.NullImporter(*path_string*)
: *NullImporter* 型は **PEP 302** インポートフックで、何もモジュールが見つからなかったときの非ディレクトリパス文字列を処理します。この型を既存のディレクトリや空文字列に対してコールすると *ImportError* が発生します。それ以外の場合は *NullImporter* のインスタンスが返されます。

インスタンスはたった一つのメソッドを持ちます:

find_module(*fullname*[, *path*])
: このメソッドは常に None を返し、要求されたモジュールが見つからなかったことを表します。

バージョン 3.3 で変更: *NullImporter* のインスタンスの代わりに None が sys.path_importer_cache に挿入されます。

バージョン 3.4 で非推奨: 代わりに None を sys.path_importer_cache に挿入してください。

36.2.1 使用例

次の関数は Python 1.4 までの標準 import 文 (階層的なモジュール名がない) をエミュレートします。(この **実装** はそのバージョンでは動作しないでしょう。なぜなら、*find_module()* は拡張されており、また *load_module()* が 1.4 で追加されているからです。)

```
import imp
import sys

def __import__(name, globals=None, locals=None, fromlist=None):
    # Fast path: see if the module has already been imported.
    try:
        return sys.modules[name]
    except KeyError:
        pass

    # If any of the following calls raises an exception,
    # there's a problem we can't handle -- let the caller handle it.
```

(次のページに続く)

(前のページからの続き)

```
    fp, pathname, description = imp.find_module(name)

    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        # Since we may exit via an exception, close fp explicitly.
        if fp:
            fp.close()
```

第

THIRTYSEVEN

ドキュメント化されていないモジュール

現在ドキュメント化されていないが、ドキュメント化すべきモジュールを以下にざっと列挙します。どうぞこれらのドキュメントを寄稿してください！ (電子メールで docs@python.org に送ってください。)

この章のアイデアと元の文章内容は Fredrik Lundh のポストによるものです; この章の特定の内容は実際には改訂されてきています。

37.1 プラットフォーム固有のモジュール

これらのモジュールは *os.path* モジュールを実装するために用いられていますが、ここで触れる内容を超えてドキュメントされていません。これらはもう少しドキュメント化する必要があります。

`ntpath` --- Win32, Win64 プラットフォームにおける *os.path* 実装です。

`posixpath` --- POSIX における *os.path* 実装です。

付録

A

用語集

>>> インタラクティブシェルにおけるデフォルトの Python プロンプトです。インタプリタでインタラクティブに実行されるコード例でよく出てきます。

... 次のものが考えられます:

- インタラクティブシェルにおいて、インデントされたコードブロック、対応する左右の区切り文字の組 (丸括弧、角括弧、波括弧、三重引用符) の内側、デコレーターの後に、コードを入力する際に表示されるデフォルトの Python プロンプトです。
- 組み込みの定数 *Ellipsis* 。

2to3 Python 2.x のコードを Python 3.x のコードに変換するツールです。ソースコードを解析してその解析木を巡回 (traverse) することで検知できる、非互換性の大部分を処理します。

2to3 は標準ライブラリの *lib2to3* として利用できます。単体のツールとしての使えるスクリプトが `Tools/scripts/2to3` として提供されています。*2to3 - Python 2* から *3* への自動コード変換 を参照してください。

abstract base class (抽象基底クラス) 抽象基底クラスは *duck-typing* を補完するもので、*hasattr()* などの別のテクニックでは不恰好であったり微妙に誤る (例えば magic methods の場合) 場合にインターフェースを定義する方法を提供します。ABC は仮想 (virtual) サブクラスを導入します。これは親クラスから継承しませんが、それでも *isinstance()* や *issubclass()* に認識されます; *abc* モジュールのドキュメントを参照してください。Python には、多くの組み込み ABC が同梱されています。その対象は、(*collections.abc* モジュールで) データ構造、(*numbers* モジュールで) 数、(*io* モジュールで) ストリーム、(*importlib.abc* モジュールで) インポートファインダ及びローダーです。*abc* モジュールを利用して独自の ABC を作成できます。

annotation (アノテーション) 変数、クラス属性、関数のパラメータや返り値に関係するラベルです。慣例により *type hint* として使われています。

ローカル変数のアノテーションは実行時にはアクセスできませんが、グローバル変数、クラス属性、関数のアノテーションはそれぞれモジュール、クラス、関数の `__annotations__` 特殊属性に保持されています。

機能の説明がある *variable annotation*, *function annotation*, **PEP 484**, **PEP 526** を参照してください。

argument （実引数）関数を呼び出す際に、**関数**（または **メソッド**）に渡す値です。実引数には２種類あります:

- **キーワード引数**: 関数呼び出しの際に引数の前に識別子がついたもの (例: `name=`) や、`**` に続けた辞書の中の値として渡された引数。例えば、次の *`complex()`* の呼び出しでは、`3` と `5` がキーワード引数です:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- **位置引数**: キーワード引数以外の引数。位置引数は引数リストの先頭に書くことができ、また `*` に続けた *iterable* の要素として渡すことができます。例えば、次の例では `3` と `5` は両方共位置引数です:

```
complex(3, 5)
complex(*(3, 5))
```

実引数は関数の実体において名前付きのローカル変数に割り当てられます。割り当てを行う規則については calls を参照してください。シンタックスにおいて実引数を表すためにあらゆる式を使うことが出来ます。評価された値はローカル変数に割り当てられます。

仮引数 、FAQ の 実引数と仮引数の違いは何ですか? 、**PEP 362** を参照してください。

asynchronous context manager （非同期コンテキストマネージャ）`__aenter__()` と `__aexit__()` メソッドを定義することで `async with` 文内の環境を管理するオブジェクトです。**PEP 492** で導入されました。

asynchronous generator （非同期ジェネレータ）*asynchronous generator iterator* を返す関数です。`async def` で定義されたコルーチン関数に似ていますが、`yield` 式を持つ点で異なります。`yield` 式は `async for` ループで使用できる値の並びを生成するのに使用されます。

通常は非同期ジェネレータ関数を指しますが、文脈によっては **非同期ジェネレータイテレータ** を指す場合があります。意図された意味が明らかでない場合、明瞭化のために完全な単語を使用します。

非同期ジェネレータ関数には、`async for` 文や `async with` 文だけでなく `await` 式もあることがあります。

asynchronous generator iterator （非同期ジェネレータイテレータ）*asynchronous generator* 関数で生成されるオブジェクトです。

これは、`__anext__()` メソッドを使って呼び出されたときに awaitable オブジェクトを返す *asynchronous iterator* です。この awaitable オブジェクトは、次の `yield` 式までの非同期ジェネレータ関数の本体を実行します。

`yield` にくるたびに、その位置での実行状態 (ローカル変数と保留状態の try 文) 処理は一時停止されます。`__anext__()` で返された他の awaitable によって **非同期ジェネレータイテレータ** が実際に再開されたとき、中断した箇所を取得します。**PEP 492** および **PEP 525** を参照してください。

asynchronous iterable （非同期イテラブル）`async for` 文の中で使用できるオブジェクトです。自身の `__aiter__()` メソッドから *asynchronous iterator* を返さなければなりません。**PEP 492** で導入されました。

asynchronous iterator (非同期イテレータ) `__aiter__()` と `__anext__()` メソッドを実装したオブジェクトです。`__anext__` は *awaitable* オブジェクトを返さなければなりません。`async for` は *StopAsyncIteration* 例外を送出するまで、非同期イテレータの `__anext__()` メソッドが返す awaitable を解決します。**PEP 492** で導入されました。

attribute (属性) オブジェクトに関連付けられ、ドット表記式によって名前で参照される値です。例えば、オブジェクト *o* が属性 *a* を持っているとき、その属性は *o.a* で参照されます。

awaitable (待機可能) `await` 式で使用することが出来るオブジェクトです。*coroutine* か、`__await__()` メソッドがあるオブジェクトです。**PEP 492** を参照してください。

BDFL 慈悲深き終身独裁者 (Benevolent Dictator For Life) の略です。Python の作者、Guido van Rossum のことです。

binary file (バイナリファイル) *bytes-like* **オブジェクト** の読み込みおよび書き込みができる **ファイルオブジェクト** です。バイナリファイルの例は、バイナリモード (`'rb'`, `'wb'` or `'rb+'`) で開かれたファイル、`sys.stdin.buffer` 、`sys.stdout.buffer` 、*io.BytesIO* や *gzip.GzipFile*. のインスタンスです。

str オブジェクトの読み書きができるファイルオブジェクトについては、*text file* も参照してください。

bytes-like object bufferobjects をサポートしていて、C 言語の意味で 連続した contiguous バッファーを提供可能なオブジェクト。*bytes*, *bytearray*, *array.array* や、多くの一般的な *memoryview* オブジェクトがこれに当たります。bytes-like オブジェクトは、データ圧縮、バイナリファイルへの保存、ソケットを経由した送信など、バイナリデータを要求するいろいろな操作に利用することができます。

幾つかの操作ではバイナリデータを変更する必要があります。その操作のドキュメントではよく ” 読み書き可能な bytes-like オブジェクト” に言及しています。変更可能なバッファーオブジェクトには、*bytearray* と *bytearray* の *memoryview* などが含まれます。また、他の幾つかの操作では不変なオブジェクト内のバイナリデータ (” 読み出し専用の bytes-like オブジェクト”) を必要します。それには *bytes* と *bytes* の *memoryview* オブジェクトが含まれます。

bytecode (バイトコード) Python のソースコードは、Python プログラムの CPython インタプリタの内部表現であるバイトコードへとコンパイルされます。バイトコードは `.pyc` ファイルにキャッシュされ、同じファイルが二度目に実行されるときはより高速になります (ソースコードからバイトコードへの再度のコンパイルは回避されます)。この ” 中間言語 (intermediate language)” は、各々のバイトコードに対応する機械語を実行する **仮想マシン** で動作するといえます。重要な注意として、バイトコードは異なる Python 仮想マシン間で動作することや、Python リリース間で安定であることは期待されていません。

バイトコードの命令一覧は *dis* **モジュール** にあります。

callback (コールバック) 将来のある時点で実行されるために引数として渡される関数

class (クラス) ユーザー定義オブジェクトを作成するためのテンプレートです。クラス定義は普通、そのクラスのインスタンス上の操作をするメソッドの定義を含みます。

class variable (クラス変数) クラス上に定義され、クラスレベルで (つまり、クラスのインスタンス上ではなしに) 変更されることを目的としている変数です。

coercion　(型強制) 同じ型の 2 引数を伴う演算の最中に行われる、ある型のインスタンスの別の型への暗黙の変換です。例えば、`int(3.15)` は浮動小数点数を整数 3 に変換します。しかし `3+4.5` では、各引数は型が異なり (一つは整数、一つは浮動小数点数)、加算をする前に同じ型に変換できなければ *TypeError* 例外が投げられます。型強制がなかったら、すべての引数は、たとえ互換な型であっても、単に `3+4.5` ではなく `float(3)+4.5` というように、プログラマーが同じ型に正規化しなければいけません。

complex number　(複素数) よく知られている実数系を拡張したもので、すべての数は実部と虚部の和として表されます。虚数は虚数単位 (`-1` の平方根) に実数を掛けたもので、一般に数学では `i` と書かれ、工学では `j` と書かれます。Python は複素数に組み込みで対応し、後者の表記を取っています。虚部は末尾に `j` をつけて書きます。例えば `3+1j` です。*math* モジュールの複素数版を利用するには、*cmath* を使います。複素数の使用はかなり高度な数学の機能です。必要性を感じなければ、ほぼ間違いなく無視してしまってよいでしょう。

context manager　(コンテキストマネージャ) `__enter__()` と `__exit__()` メソッドを定義することで `with` 文内の環境を管理するオブジェクトです。**PEP 343** を参照してください。

context variable　(コンテキスト変数) コンテキストに依存して異なる値を持つ変数。これは、ある変数の値が各々の実行スレッドで異なり得るスレッドローカルストレージに似ています。しかしコンテキスト変数では、1 つの実行スレッドにいくつかのコンテキストがあり得、コンテキスト変数の主な用途は並列な非同期タスクの変数の追跡です。*contextvars* を参照してください。

contiguous　(隣接、連続) バッファが厳密に *C*-**連続** または *Fortran* **連続** である場合に、そのバッファは連続しているとみなせます。ゼロ次元バッファは C 連続であり Fortran 連続です。一次元の配列では、その要素は必ずメモリ上で隣接するように配置され、添字がゼロから始まり増えていく順序で並びます。多次元の C-連続な配列では、メモリアドレス順に要素を巡る際には最後の添え字が最初に変わるのに対し、Fortran 連続な配列では最初の添え字が最初に動きます。

coroutine　(コルーチン) コルーチンはサブルーチンのより一般的な形式です。サブルーチンには決められた地点から入り、別の決められた地点から出ます。コルーチンには多くの様々な地点から入る、出る、再開することができます。コルーチンは `async def` 文で実装できます。**PEP 492** を参照してください。

coroutine function　(コルーチン関数) *coroutine* オブジェクトを返す関数です。コルーチン関数は `async def` 文で実装され、`await`、`async for`、および `async with` キーワードを持つことが出来ます。これらは **PEP 492** で導入されました。

CPython　python.org で配布されている、Python プログラミング言語の標準的な実装です。"CPython" という単語は、この実装を Jython や IronPython といった他の実装と区別する必要が有る場合に利用されます。

decorator　(デコレータ) 別の関数を返す関数で、通常、`@wrapper` 構文で関数変換として適用されます。デコレータの一般的な利用例は、*classmethod()* と *staticmethod()* です。

デコレータの文法はシンタックスシュガーです。次の 2 つの関数定義は意味的に同じものです:

```
def f(...):
    ...
f = staticmethod(f)
```

(次のページに続く)

(前のページからの続き)

```
@staticmethod
def f(...):
    ...
```

同じ概念がクラスにも存在しますが、あまり使われません。デコレータについて詳しくは、関数定義 および クラス定義 のドキュメントを参照してください。

descriptor (デスクリプタ) メソッド `__get__()`, `__set__()`, あるいは `__delete__()` を定義しているオブジェクトです。あるクラス属性がデスクリプタであるとき、属性探索によって、束縛されている特別な動作が呼び出されます。通常、get,set,delete のために *a.b* と書くと、*a* のクラス辞書内でオブジェクト *b* を検索しますが、*b* がデスクリプタであればそれぞれのデスクリプタメソッドが呼び出されます。デスクリプタの理解は、Python を深く理解する上で鍵となります。というのは、デスクリプタこそが、関数、メソッド、プロパティ、クラスメソッド、静的メソッド、そしてスーパクラスの参照といった多くの機能の基盤だからです。

デスクリプタのメソッドに関して詳しくは、descriptors を参照してください。

dictionary (辞書) 任意のキーを値に対応付ける連想配列です。`__hash__()` メソッドと `__eq__()` メソッドを実装した任意のオブジェクトをキーにできます。Perl ではハッシュ (hash) と呼ばれています。

dictionary comprehension (辞書内包表記) iterable 内の全てあるいは一部の要素を処理して、その結果からなる辞書を返すコンパクトな書き方です。`results = {n: n ** 2 for n in range(10)}` とすると、キー `n` を値 `n ** 2` に対応付ける辞書を生成します。comprehensions を参照してください。

dictionary view (辞書ビュー) *dict.keys()*、*dict.values()*、*dict.items()* が返すオブジェクトです。辞書の項目の動的なビューを提供します。すなわち、辞書が変更されるとビューはそれを反映します。辞書ビューを強制的に完全なリストにするには `list(dictview)` を使用してください。**辞書ビューオブジェクト** を参照してください。

docstring クラス、関数、モジュールの最初の式である文字列リテラルです。そのスイートの実行時には無視されますが、コンパイラによって識別され、そのクラス、関数、モジュールの `__doc__` 属性として保存されます。イントロスペクションできる（訳注: 属性として参照できる）ので、オブジェクトのドキュメントを書く標準的な場所です。

duck-typing あるオブジェクトが正しいインタフェースを持っているかを決定するのにオブジェクトの型を見ないプログラミングスタイルです。代わりに、単純にオブジェクトのメソッドや属性が呼ばれたり使われたりします。(「アヒルのように見えて、アヒルのように鳴けば、それはアヒルである。」) インタフェースを型より重視することで、上手くデザインされたコードは、ポリモーフィックな代替を許して柔軟性を向上させます。ダックタイピングは *type()* や *isinstance()* による判定を避けます。(ただし、ダックタイピングを **抽象基底クラス** で補完することもできます。) その代わり、典型的に *hasattr()* 判定や *EAFP* プログラミングを利用します。

EAFP 「認可をとるより許しを請う方が容易 (easier to ask for forgiveness than permission、マーフィーの法則)」の略です。この Python で広く使われているコーディングスタイルでは、通常は有効なキーや属性が存在するものと仮定し、その仮定が誤っていた場合に例外を捕捉します。この簡潔で手早く書けるコーディングスタイルには、`try` 文および `except` 文がたくさんあるのが特徴です。このテクニックは、C のような言語でよく使われている *LBYL* スタイルと対照的なものです。

expression　(式) 何かの値と評価される、一まとまりの構文 (a piece of syntax) です。言い換えると、式とはリテラル、名前、属性アクセス、演算子や関数呼び出しなど、値を返す式の要素の積み重ねです。他の多くの言語と違い、Python では言語の全ての構成要素が式というわけではありません。`while` のように、式としては使えない **文** もあります。代入も式ではなく文です。

extension module　(拡張モジュール) C や C++ で書かれたモジュールで、Python の C API を利用して Python コアやユーザーコードとやりとりします。

f-string　`'f'` や `'F'` が先頭に付いた文字列リテラルは "f-string" と呼ばれ、これは フォーマット済み文字列リテラル の短縮形の名称です。**PEP 498** も参照してください。

file object　(ファイルオブジェクト) 下位のリソースへのファイル志向 API (`read()` や `write()` メソッドを持つもの) を公開しているオブジェクトです。ファイルオブジェクトは、作成された手段によって、実際のディスク上のファイルや、その他のタイプのストレージや通信デバイス (例えば、標準入出力、インメモリバッファ、ソケット、パイプ、等) へのアクセスを媒介できます。ファイルオブジェクトは *file-like objects* や *streams* とも呼ばれます。

ファイルオブジェクトには実際には 3 種類あります: 生の **バイナリーファイル**、バッファされた **バイナリーファイル**、そして **テキストファイル** です。インターフェイスは *io* モジュールで定義されています。ファイルオブジェクトを作る標準的な方法は *open()* 関数を使うことです。

file-like object　*file object* と同義です。

finder　(ファインダ) インポートされているモジュールの *loader* の発見を試行するオブジェクトです。

Python 3.3 以降では 2 種類のファインダがあります。*sys.meta_path* で使用される *meta path finder* と、*sys.path_hooks* で使用される *path entry finder* です。

詳細については **PEP 302**、**PEP 420** および **PEP 451** を参照してください。

floor division　一番近い小さい整数に丸める数学除算。floor division 演算子は `//` です。例えば、`11 // 4` は 2 になり、float の true division の結果 `2.75` と異なります。`(-11) // 4` は `-2.75` を **小さい方に** 丸めるので `-3` になることに注意してください。**PEP 238** を参照してください。

function　(関数) 呼び出し側に値を返す一連の文のことです。関数には 0 以上の **実引数** を渡すことが出来ます。実体の実行時に引数を使用することが出来ます。**仮引数**、**メソッド**、function を参照してください。

function annotation　(関数アノテーション) 関数のパラメータや返り値の *annotation* です。

関数アノテーションは、通常は **型ヒント** のために使われます: 例えば、この関数は 2 つの *int* 型の引数を取ると期待され、また *int* 型の返り値を持つと期待されています。

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

関数アノテーションの文法は function の節で解説されています。

機能の説明がある *variable annotation* と **PEP 484** を参照してください。

__future__　互換性のない新たな言語機能を現在のインタプリタで有効にするためにプログラマが利用できる擬似モジュールです。

__future__ モジュールを import してその変数を評価すれば、新たな機能が初めて追加されたのがいつで、いつ言語デフォルトの機能になるかわかります:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection （ガベージコレクション）これ以降使われることのないメモリを解放する処理です。Python は、参照カウントと、循環参照を検出し破壊する循環ガベージコレクタを使ってガベージコレクションを行います。ガベージコレクタは *gc* モジュールを使って操作できます。

generator （ジェネレータ）*generator iterator* を返す関数です。通常の関数に似ていますが、`yield` 式を持つ点で異なります。`yield` 式は、for ループで使用できたり、*next()* 関数で値を 1 つずつ取り出したりできる、値の並びを生成するのに使用されます。

通常はジェネレータ関数を指しますが、文脈によっては **ジェネレータイテレータ** を指す場合があります。意図された意味が明らかでない場合、明瞭化のために完全な単語を使用します。

generator iterator （ジェネレータイテレータ）*generator* 関数で生成されるオブジェクトです。

`yield` のたびに局所実行状態 (局所変数や未処理の try 文などを含む) を記憶して、処理は一時的に中断されます。**ジェネレータイテレータ** が再開されると、中断した位置を取得します (通常の関数が実行のたびに新しい状態から開始するのと対照的です)。

generator expression （ジェネレータ式）イテレータを返す式です。普通の式に、ループ変数を定義する `for` 節、範囲、そして省略可能な `if` 節がつづいているように見えます。こうして構成された式は、外側の関数に向けて値を生成します:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

generic function （ジェネリック関数）異なる型に対し同じ操作をする関数群から構成される関数です。呼び出し時にどの実装を用いるかはディスパッチアルゴリズムにより決定されます。

single dispatch、*functools.singledispatch()* デコレータ、**PEP 443** を参照してください。

GIL *global interpreter lock* を参照してください。

global interpreter lock （グローバルインタプリタロック）*CPython* インタプリタが利用している、一度に Python の **バイトコード** を実行するスレッドは一つだけであることを保証する仕組みです。これにより (*dict* などの重要な組み込み型を含む) オブジェクトモデルが同時アクセスに対して暗黙的に安全になるので、CPython の実装がシンプルになります。インタプリタ全体をロックすることで、マルチプロセッサマシンが生じる並列化のコストと引き換えに、インタプリタを簡単にマルチスレッド化できるようになります。

ただし、標準あるいは外部のいくつかの拡張モジュールは、圧縮やハッシュ計算などの計算の重い処理をするときに GIL を解除するように設計されています。また、I/O 処理をする場合 GIL は常に解除されます。

過去に ” 自由なマルチスレッド化” したインタプリタ (供用されるデータを細かい粒度でロックする)

が開発されましたが、一般的なシングルプロセッサの場合のパフォーマンスが悪かったので成功しませんでした。このパフォーマンスの問題を克服しようとすると、実装がより複雑になり保守コストが増加すると考えられています。

hash-based pyc　(ハッシュベース pyc ファイル) 正当性を判別するために、対応するソースファイルの最終更新時刻ではなくハッシュ値を使用するバイトコードのキャッシュファイルです。

hashable　(ハッシュ可能) **ハッシュ可能** なオブジェクトとは、生存期間中変わらないハッシュ値を持ち (`__hash__()` メソッドが必要)、他のオブジェクトと比較ができる (`__eq__()` メソッドが必要) オブジェクトです。同値なハッシュ可能オブジェクトは必ず同じハッシュ値を持つ必要があります。

ハッシュ可能なオブジェクトは辞書のキーや集合のメンバーとして使えます。辞書や集合のデータ構造は内部でハッシュ値を使っているからです。

Python のイミュータブルな組み込みオブジェクトは、ほとんどがハッシュ可能です。(リストや辞書のような) ミュータブルなコンテナはハッシュ不可能です。(タプルや frozenset のような) イミュータブルなコンテナは、要素がハッシュ可能であるときのみハッシュ可能です。ユーザー定義のクラスのインスタンスであるようなオブジェクトはデフォルトでハッシュ可能です。それらは全て (自身を除いて) 比較結果は非等価であり、ハッシュ値は *`id()`* より得られます。

IDLE　Python の統合開発環境 (Integrated DeveLopment Environment) です。IDLE は Python の標準的な配布に同梱されている基本的な機能のエディタとインタプリタ環境です。

immutable　(イミュータブル) 固定の値を持ったオブジェクトです。イミュータブルなオブジェクトには、数値、文字列、およびタプルなどがあります。これらのオブジェクトは値を変えられません。別の値を記憶させる際には、新たなオブジェクトを作成しなければなりません。イミュータブルなオブジェクトは、固定のハッシュ値が必要となる状況で重要な役割を果たします。辞書のキーがその例です。

import path　*path based finder* が import するモジュールを検索する場所 (または *path entry*) のリスト。import 中、このリストは通常 *`sys.path`* から来ますが、サブパッケージの場合は親パッケージの `__path__` 属性からも来ます。

importing　あるモジュールの Python コードが別のモジュールの Python コードで使えるようにする処理です。

importer　モジュールを探してロードするオブジェクト。*finder* と *loader* のどちらでもあるオブジェクト。

interactive　(対話的) Python には対話的インタプリタがあり、文や式をインタプリタのプロンプトに入力すると即座に実行されて結果を見ることができます。`python` と何も引数を与えずに実行してください。(コンピュータのメインメニューから Python の対話的インタプリタを起動できるかもしれません。) 対話的インタプリタは、新しいアイデアを試してみたり、モジュールやパッケージの中を覗いてみる (`help(x)` を覚えておいてください) のに非常に便利なツールです。

interpreted　Python はインタプリタ形式の言語であり、コンパイラ言語の対極に位置します。(バイトコードコンパイラがあるために、この区別は曖昧ですが。) ここでのインタプリタ言語とは、ソースコードのファイルを、まず実行可能形式にしてから実行させるといった操作なしに、直接実行できることを意味します。インタプリタ形式の言語は通常、コンパイラ形式の言語よりも開発／デバッグのサイクルは短いものの、プログラムの実行は一般に遅いです。**対話的** も参照してください。

interpreter shutdown　Python インタープリターはシャットダウンを要請された時に、モジュールやすべてのクリティカルな内部構造をなどの、すべての確保したリソースを段階的に開放する、特別なフェーズに入ります。このフェーズは **ガベージコレクタ** を複数回呼び出します。これによりユーザー定義のデストラクターや weakref コールバックが呼び出されることがあります。シャットダウンフェーズ中に実行されるコードは、それが依存するリソースがすでに機能しない (よくある例はライブラリーモジュールや warning 機構です) ために様々な例外に直面します。

インタープリタがシャットダウンする主な理由は `__main__` モジュールや実行されていたスクリプトの実行が終了したことです。

iterable　(反復可能オブジェクト) 要素を一度に 1 つずつ返せるオブジェクトです。反復可能オブジェクトの例には、(*list*, *str*, *tuple* といった) 全てのシーケンス型や、*dict* や **ファイルオブジェクト** といった幾つかの非シーケンス型、あるいは *Sequence* 意味論を実装した `__iter__()` メソッドか `__getitem__()` メソッドを持つ任意のクラスのインスタンスが含まれます。

反復可能オブジェクトは `for` ループ内やその他多くのシーケンス (訳注: ここでのシーケンスとは、シーケンス型ではなくただの列という意味) が必要となる状況 (*zip()*, *map()*, ...) で利用できます。反復可能オブジェクトを組み込み関数 *iter()* の引数として渡すと、オブジェクトに対するイテレータを返します。このイテレータは一連の値を引き渡す際に便利です。通常は反復可能オブジェクトを使う際には、*iter()* を呼んだりイテレータオブジェクトを自分で操作する必要はありません。`for` 文ではこの操作を自動的に行い、一時的な無名の変数を作成してループを回している間イテレータを保持します。**イテレータ** 、**シーケンス** 、**ジェネレータ** も参照してください。

iterator　(イテレータ) データの流れを表現するオブジェクトです。イテレータの *__next__()* メソッドを繰り返し呼び出す (または組み込み関数 *next()* に渡す) と、流れの中の要素を一つずつ返します。データがなくなると、代わりに *StopIteration* 例外を送出します。その時点で、イテレータオブジェクトは尽きており、それ以降は `__next__()` を何度呼んでも *StopIteration* を送出します。イテレータは、そのイテレータオブジェクト自体を返す `__iter__()` メソッドを実装しなければならないので、イテレータは他の iterable を受理するほとんどの場所で利用できます。はっきりとした例外は複数の反復を行うようなコードです。(*list* のような) コンテナオブジェクトは、自身を *iter()* 関数にオブジェクトに渡したり `for` ループ内で使うたびに、新たな未使用のイテレータを生成します。これをイテレータで行おうとすると、前回のイテレーションで使用済みの同じイテレータオブジェクトを単純に返すため、空のコンテナのようになってしまします。

詳細な情報は **イテレータ型** にあります。

key function　(キー関数) キー関数、あるいは照合関数とは、ソートや順序比較のための値を返す呼び出し可能オブジェクト (callable) です。例えば、*locale.strxfrm()* をキー関数に使えば、ロケール依存のソートの慣習にのっとったソートキーを返します。

Python の多くのツールはキー関数を受け取り要素の並び順やグループ化を管理します。*min()*, *max()*, *sorted()*, *list.sort()*, *heapq.merge()*, *heapq.nsmallest()*, *heapq.nlargest()*, *itertools.groupby()* 等があります。

キー関数を作る方法はいくつかあります。例えば *str.lower()* メソッドを大文字小文字を区別しないソートを行うキー関数として使うことが出来ます。あるいは、`lambda r: (r[0], r[2])` のような `lambda` 式からキー関数を作ることができます。また、*operator* モジュールは *attrgetter()*,

itemgetter(), *methodcaller()* という 3 つのキー関数コンストラクタを提供しています。キー関数の作り方と使い方の例は Sorting HOW TO を参照してください。

keyword argument　**実引数** を参照してください。

lambda　(ラムダ) 無名のインライン関数で、関数が呼び出されたときに評価される 1 つの **式** を含みます。ラムダ関数を作る構文は `lambda [parameters]: expression` です。

LBYL　「ころばぬ先の杖 (look before you leap)」の略です。このコーディングスタイルでは、呼び出しや検索を行う前に、明示的に前提条件 (pre-condition) 判定を行います。*EAFP* アプローチと対照的で、`if` 文がたくさん使われるのが特徴的です。

マルチスレッド化された環境では、LBYL アプローチは "見る" 過程と "飛ぶ" 過程の競合状態を引き起こすリスクがあります。例えば、`if key in mapping: return mapping[key]` というコードは、判定の後、別のスレッドが探索の前に *mapping* から *key* を取り除くと失敗します。この問題は、ロックするか EAFP アプローチを使うことで解決できます。

list　(リスト) Python の組み込みの **シーケンス** です。リストという名前ですが、リンクリストではなく、他の言語で言う配列 (array) と同種のもので、要素へのアクセスは O(1) です。

list comprehension　(リスト内包表記) シーケンス中の全てあるいは一部の要素を処理して、その結果からなるリストを返す、コンパクトな方法です。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` とすると、0 から 255 までの偶数を 16 進数表記 (0x..) した文字列からなるリストを生成します。`if` 節はオプションです。`if` 節がない場合、`range(256)` の全ての要素が処理されます。

loader　モジュールをロードするオブジェクト。`load_module()` という名前のメソッドを定義していなければなりません。ローダーは一般的に *finder* から返されます。詳細は **PEP 302** を、*abstract base class* については *importlib.abc.Loader* を参照してください。

magic method　*special method* のくだけた同義語です。

mapping　(マッピング) 任意のキー探索をサポートしていて、*Mapping* か *MutableMapping* の **抽象基底クラス** で指定されたメソッドを実装しているコンテナオブジェクトです。例えば、*dict*, *collections.defaultdict*, *collections.OrderedDict*, *collections.Counter* などです。

meta path finder　*sys.meta_path* を検索して得られた *finder*. meta path finder は *path entry finder* と関係はありますが、別物です。

meta path finder が実装するメソッドについては *importlib.abc.MetaPathFinder* を参照してください。

metaclass　(メタクラス) クラスのクラスです。クラス定義は、クラス名、クラスの辞書と、基底クラスのリストを作ります。メタクラスは、それら 3 つを引数として受け取り、クラスを作る責任を負います。ほとんどのオブジェクト指向言語は (訳注:メタクラスの) デフォルトの実装を提供しています。Python が特別なのはカスタムのメタクラスを作成できる点です。ほとんどのユーザーに取って、メタクラスは全く必要のないものです。しかし、一部の場面では、メタクラスは強力でエレガントな方法を提供します。たとえば属性アクセスのログを取ったり、スレッドセーフ性を追加したり、オブジェクトの生成を追跡したり、シングルトンを実装するなど、多くの場面で利用されます。

詳細は metaclasses を参照してください。

method （メソッド) クラス本体の中で定義された関数。そのクラスのインスタンスの属性として呼び出された場合、メソッドはインスタンスオブジェクトを第一 **引数** として受け取ります (この第一引数は通常 `self` と呼ばれます)。**関数** と **ネストされたスコープ** も参照してください。

method resolution order （メソッド解決順序) 探索中に基底クラスが構成要素を検索される順番です。2.3 以降の Python インタープリタが使用するアルゴリズムの詳細については The Python 2.3 Method Resolution Order を参照してください。

module （モジュール) Python コードの組織単位としてはたらくオブジェクトです。モジュールは任意の Python オブジェクトを含む名前空間を持ちます。モジュールは *importing* の処理によって Python に読み込まれます。

パッケージ を参照してください。

module spec モジュールをロードするのに使われるインポート関連の情報を含む名前空間です。*importlib.machinery.ModuleSpec* のインスタンスです。

MRO *method resolution order* を参照してください。

mutable （ミュータブル) ミュータブルなオブジェクトは、*id()* を変えることなく値を変更できます。**イミュータブル**) も参照してください。

named tuple ”名前付きタプル” という用語は、タプルを継承していて、インデックスが付く要素に対し属性を使ってのアクセスもできる任意の型やクラスに応用されています。その型やクラスは他の機能も持っていることもあります。

time.localtime() や *os.stat()* の返り値を含むいくつかの組み込み型は名前付きタプルです。他の例は *sys.float_info* です:

```
>>> sys.float_info[1]                   # indexed access
1024
>>> sys.float_info.max_exp              # named field access
1024
>>> isinstance(sys.float_info, tuple)   # kind of tuple
True
```

(上の例のように) いくつかの名前付きタプルは組み込み型になっています。その他にも名前付きタプルは、通常のクラス定義で *tuple* を継承し、名前のフィールドを定義して作成できます。そのようなクラスは手動で書いたり、*collections.namedtuple()* ファクトリ関数で作成したりできます。後者の方法は、手動で書いた名前付きタプルや組み込みの名前付きタプルには無い付加的なメソッドを追加できます。

namespace （名前空間) 変数が格納される場所です。名前空間は辞書として実装されます。名前空間にはオブジェクトの (メソッドの) 入れ子になったものだけでなく、局所的なもの、大域的なもの、そして組み込みのものがあります。名前空間は名前の衝突を防ぐことによってモジュール性をサポートする。例えば関数 *builtins.open* と *os.open()* は名前空間で区別されています。また、どのモジュールが関数を実装しているか明示することによって名前空間は可読性と保守性を支援します。例えば、*random.seed()* や *itertools.islice()* と書くと、それぞれモジュール *random* や *itertools* で実装されていることが明らかです。

namespace package （名前空間パッケージ）サブパッケージのコンテナとして提供される **PEP 420** *package* 。Namespace package はおそらく物理表現を持たず、`__init__.py` ファイルがないため、*regular package* と異なります。

module を参照してください。

nested scope （ネストされたスコープ）外側で定義されている変数を参照する機能です。例えば、ある関数が別の関数の中で定義されている場合、内側の関数は外側の関数中の変数を参照できます。ネストされたスコープはデフォルトでは変数の参照だけができ、変数の代入はできないので注意してください。ローカル変数は、最も内側のスコープで変数を読み書きします。同様に、グローバル変数を使うとグローバル名前空間の値を読み書きします。`nonlocal` で外側の変数に書き込めます。

new-style class （新スタイルクラス）今では全てのクラスオブジェクトに使われている味付けの古い名前です。以前の Python のバージョンでは、新スタイルクラスのみが `__slots__` 、デスクリプタ、`__getattribute__()` 、クラスメソッド、そして静的メソッド等の Python の新しい、多様な機能を利用できました。

object （オブジェクト）状態 (属性や値) と定義された振る舞い (メソッド) をもつ全てのデータ。もしくは、全ての **新スタイルクラス** の究極の基底クラスのこと。

package （パッケージ）サブモジュールや再帰的にサブパッケージを含むことの出来る *module* のことです。専門的には、パッケージは `__path__` 属性を持つ Python オブジェクトです。

regular package と *namespace package* を参照してください。

parameter （仮引数）名前付の実体で **関数** (や **メソッド**) の定義において関数が受ける **実引数** を指定します。仮引数には 5 種類あります:

- **位置またはキーワード**: **位置** であるいは **キーワード引数** として渡すことができる引数を指定します。これはたとえば以下の *foo* や *bar* のように、デフォルトの仮引数の種類です:

```
def func(foo, bar=None): ...
```

- 位置専用: 位置によってのみ与えられる引数を指定します。位置専用の引数は 関数定義の引数のリストの中でそれらの後ろに `/` を含めることで定義できます。例えば下記の *posonly1* と *posonly2* は位置専用引数になります:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- **キーワード専用**: キーワードによってのみ与えられる引数を指定します。キーワード専用の引数を定義できる場所は、例えば以下の *kw_only1* や *kw_only2* のように、関数定義の仮引数リストに含めた可変長位置引数または裸の `*` の後です:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- **可変長位置**: (他の仮引数で既に受けられた任意の位置引数に加えて) 任意の個数の位置引数が与えられることを指定します。このような仮引数は、以下の *args* のように仮引数名の前に `*` をつけることで定義できます:

```
def func(*args, **kwargs): ...
```

- **可変長キーワード**: (他の仮引数で既に受けられた任意のキーワード引数に加えて) 任意の個数のキーワード引数が与えられることを指定します。このような仮引数は、上の例の *kwargs* のように仮引数名の前に ** をつけることで定義できます。

仮引数はオプションと必須の引数のどちらも指定でき、オプションの引数にはデフォルト値も指定できます。

仮引数 、FAQ の 実引数と仮引数の違いは何ですか? 、*inspect.Parameter* クラス、function セクション、**PEP 362** を参照してください。

path entry *path based finder* が import するモジュールを探す *import path* 上の 1 つの場所です。

path entry finder *sys.path_hooks* にある callable (つまり *path entry hook*) が返した *finder* です。与えられた *path entry* にあるモジュールを見つける方法を知っています。

パスエントリーファインダが実装するメソッドについては *importlib.abc.PathEntryFinder* を参照してください。

path entry hook `sys.path_hook` リストにある callable で、指定された *path entry* にあるモジュールを見つける方法を知っている場合に *path entry finder* を返します。

path based finder デフォルトの *meta path finder* の 1 つは、モジュールの *import path* を検索します。

path-like object (path-like オブジェクト) ファイルシステムパスを表します。path-like オブジェクトは、パスを表す *str* オブジェクトや *bytes* オブジェクト、または *os.PathLike* プロトコルを実装したオブジェクトのどれかです。*os.PathLike* プロトコルをサポートしているオブジェクトは *os.fspath()* を呼び出すことで *str* または *bytes* のファイルシステムパスに変換できます。*os.fsdecode()* と *os.fsencode()* はそれぞれ *str* あるいは *bytes* になるのを保証するのに使えます。**PEP 519** で導入されました。

PEP Python Enhancement Proposal。PEP は、Python コミュニティに対して情報を提供する、あるいは Python の新機能やその過程や環境について記述する設計文書です。PEP は、機能についての簡潔な技術的仕様と提案する機能の論拠 (理論) を伝えるべきです。

PEP は、新機能の提案にかかる、コミュニティによる問題提起の集積と Python になされる設計決断の文書化のための最上位の機構となることを意図しています。PEP の著者にはコミュニティ内の合意形成を行うこと、反対意見を文書化することの責務があります。

PEP 1 を参照してください。

portion **PEP 420** で定義されている、namespace package に属する、複数のファイルが (zip ファイルに格納されている場合もある) 1 つのディレクトリに格納されたもの。

positional argument (**位置引数**) 実引数 を参照してください。

provisional API (暫定 API) 標準ライブラリの後方互換性保証から計画的に除外されたものです。そのようなインタフェースへの大きな変更は、暫定であるとされている間は期待されていませんが、コア開発者によって必要とみなされれば、後方非互換な変更 (インタフェースの削除まで含まれる) が行われえま

す。このような変更はむやみに行われるものではありません -- これは API を組み込む前には見落とされていた重大な欠陥が露呈したときにのみ行われます。

暫定 API についても、後方互換性のない変更は「最終手段」とみなされています。問題点が判明した場合でも後方互換な解決策を探すべきです。

このプロセスにより、標準ライブラリは問題となるデザインエラーに長い間閉じ込められることなく、時代を超えて進化を続けられます。詳細は **PEP 411** を参照してください。

provisional package *provisional API* を参照してください。

Python 3000 Python 3.x リリースラインのニックネームです。(Python 3 が遠い将来の話だった頃に作られた言葉です。) "Py3k" と略されることもあります。

Pythonic 他の言語で一般的な考え方で書かれたコードではなく、Python の特に一般的なイディオムに従った考え方やコード片。例えば、Python の一般的なイディオムでは `for` 文を使ってイテラブルのすべての要素に渡ってループします。他の多くの言語にはこの仕組みはないので、Python に慣れていない人は代わりに数値のカウンターを使うかもしれません:

```
for i in range(len(food)):
    print(food[i])
```

これに対し、きれいな Pythonic な方法は:

```
for piece in food:
    print(piece)
```

qualified name (修飾名) モジュールのグローバルスコープから、そのモジュールで定義されたクラス、関数、メソッドへの、" パス" を表すドット名表記です。**PEP 3155** で定義されています。トップレベルの関数やクラスでは、修飾名はオブジェクトの名前と同じです:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

モジュールへの参照で使われると、**完全修飾名** *(fully qualified name)* はすべての親パッケージを含む全体のドット名表記、例えば `email.mime.text` を意味します:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count　(参照カウント) あるオブジェクトに対する参照の数。参照カウントが 0 になったとき、そのオブジェクトは破棄されます。参照カウントは通常は Python のコード上には現れませんが、*CPython* 実装の重要な要素です。*sys* モジュールは、プログラマーが任意のオブジェクトの参照カウントを知るための *getrefcount()* 関数を提供しています。

regular package　伝統的な、`__init__.py` ファイルを含むディレクトリとしての *package*。

namespace package を参照してください。

___slots___　クラス内での宣言で、インスタンス属性の領域をあらかじめ定義しておき、インスタンス辞書を排除することで、メモリを節約します。これはよく使われるテクニックですが、正しく扱うには少しトリッキーなので、稀なケース、例えばメモリが死活問題となるアプリケーションでインスタンスが大量に存在する、といったときを除き、使わないのがベストです。

sequence　(シーケンス) 整数インデクスによる効率的な要素アクセスを `__getitem__()` 特殊メソッドを通じてサポートし、長さを返す `__len__()` メソッドを定義した *iterable* です。組み込みシーケンス型には、*list*, *str*, *tuple*, *bytes* などがあります。*dict* は `__getitem__()` と `__len__()` もサポートしますが、検索の際に整数ではなく任意の *immutable* なキーを使うため、シーケンスではなくマッピング (mapping) とみなされているので注意してください。

The *collections.abc.Sequence* abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using *register()*.

set comprehension　A compact way to process all or part of the elements in an iterable and return a set with the results. `results = {c for c in 'abracadabra' if c not in 'abc'}` generates the set of strings `{'r', 'd'}`. See comprehensions.

single dispatch　*generic function* の一種で実装は一つの引数の型により選択されます。

slice　(スライス) 一般に **シーケンス** の一部を含むオブジェクト。スライスは、添字表記 `[]` で与えられた複数の数の間にコロンを書くことで作られます。例えば、`variable_name[1:3:5]` です。角括弧 (添字) 記号は *slice* オブジェクトを内部で利用しています。

special method　(特殊メソッド) ある型に特定の操作、例えば加算をするために Python から暗黙に呼び出されるメソッド。この種類のメソッドは、メソッド名の最初と最後にアンダースコア 2 つがついています。特殊メソッドについては specialnames で解説されています。

statement　(文) 文はスイート (コードの” ブロック”) に不可欠な要素です。文は **式** かキーワードから構成されるもののどちらかです。後者には `if`、`while`、`for` があります。

text encoding　ユニコード文字列をエンコードするコーデックです。

text file　(テキストファイル) *str* オブジェクトを読み書きできる *file object* です。しばしば、テキストファイルは実際にバイト指向のデータストリームにアクセスし、**テキストエンコーディング** を自動的に行います。テキストファイルの例は、*sys.stdin*, *sys.stdout*, *io.StringIO* インスタンスなどをテキストモード (`'r'` or `'w'`) で開いたファイルです。

bytes-like **オブジェクト** を読み書きできるファイルオブジェクトについては、**バイナリファイル** も参照してください。

triple-quoted string　(三重クォート文字列) 3 つの連続したクォート記号 (") かアポストロフィー (') で囲まれた文字列。通常の (一重) クォート文字列に比べて表現できる文字列に違いはありませんが、幾つかの理由で有用です。1 つか 2 つの連続したクォート記号をエスケープ無しに書くことができますし、行継続文字 (\) を使わなくても複数行にまたがることができるので、ドキュメンテーション文字列を書く時に特に便利です。

type　(型) Python オブジェクトの型はオブジェクトがどのようなものかを決めます。あらゆるオブジェクトは型を持っています。オブジェクトの型は *__class__* 属性でアクセスしたり、`type(obj)` で取得したり出来ます。

type alias　(型エイリアス) 型の別名で、型を識別子に代入して作成します。

型エイリアスは **型ヒント** を単純化するのに有用です。例えば:

```
from typing import List, Tuple

def remove_gray_shades(
        colors: List[Tuple[int, int, int]]) -> List[Tuple[int, int, int]]:
    pass
```

これは次のようにより読みやすくできます:

```
from typing import List, Tuple

Color = Tuple[int, int, int]

def remove_gray_shades(colors: List[Color]) -> List[Color]:
    pass
```

機能の説明がある *typing* と **PEP 484** を参照してください。

type hint　(型ヒント) 変数、クラス属性、関数のパラメータや返り値の期待される型を指定する *annotation* です。

型ヒントは必須ではなく Python では強制ではありませんが、静的型解析ツールにとって有用であり、IDE のコード補完とリファクタリングの手助けになります。

グローバル変数、クラス属性、関数で、ローカル変数でないものの型ヒントは *typing.get_type_hints()* で取得できます。

機能の説明がある *typing* と **PEP 484** を参照してください。

universal newlines　テキストストリームの解釈法の一つで、以下のすべてを行末と認識します: Unix の行末規定 `'\n'`、Windows の規定 `'\r\n'`、古い Macintosh の規定 `'\r'`。利用法について詳しくは、**PEP 278** と **PEP 3116** 、さらに *bytes.splitlines()* も参照してください。

variable annotation　(変数アノテーション) 変数あるいはクラス属性の *annotation* 。

変数あるいはクラス属性に注釈を付けたときは、代入部分は任意です:

```
class C:
    field: 'annotation'
```

変数アノテーションは通常は **型ヒント** のために使われます: 例えば、この変数は *int* の値を取ることを期待されています:

```
count: int = 0
```

変数アノテーションの構文については annassign 節で解説しています。

この機能について解説している *function annotation*, **PEP 484**, **PEP 526** を参照してください。

virtual environment （仮想環境）協調的に切り離された実行環境です。これにより Python ユーザとアプリケーションは同じシステム上で動いている他の Python アプリケーションの挙動に干渉することなく Python パッケージのインストールと更新を行うことができます。

venv を参照してください。

virtual machine （仮想マシン）完全にソフトウェアにより定義されたコンピュータ。Python の仮想マシンは、バイトコードコンパイラが出力した **バイトコード** を実行します。

Zen of Python （Python の悟り）Python を理解し利用する上での導きとなる、Python の設計原則と哲学をリストにしたものです。対話プロンプトで "`import this`" とするとこのリストを読めます。

付録

B

このドキュメントについて

このドキュメントは、Python のドキュメントを主要な目的として作られた ドキュメントプロセッサの Sphinx を利用して、reStructuredText 形式のソースから生成されました。

ドキュメントとそのツール群の開発は、Python 自身と同様に完全にボランティアの努力です。もしあなたが貢献したいなら、どのようにすればよいかについて reporting-bugs ページをご覧下さい。新しいボランティアはいつでも歓迎です!（訳注: 日本語訳の問題については、GitHub 上の Issue Tracker で報告をお願いします。）

多大な感謝を:

- Fred L. Drake, Jr., オリジナルの Python ドキュメントツールセットの作成者で、ドキュメントの多くを書きました。
- Docutils プロジェクト. reStructuredText と docutils ツールセットを作成しました。
- Fredrik Lundh の Alternative Python Reference プロジェクトから Sphinx は多くのアイデアを得ました。

B.1 Python ドキュメント 貢献者

多くの方々が Python 言語、Python 標準ライブラリ、そして Python ドキュメンテーションに貢献してくれています。ソース配布物の Misc/ACKS に、それら貢献してくれた人々を部分的にではありますがリストアップしてあります。

Python コミュニティからの情報提供と貢献がなければこの素晴らしいドキュメンテーションは生まれませんでした -- ありがとう!

付録

C

歴史とライセンス

C.1 Python の歴史

Python は 1990 年代の始め、オランダにある Stichting Mathematisch Centrum (CWI, https://www.cwi.nl/ 参照) で Guido van Rossum によって ABC と呼ばれる言語の後継言語として生み出されました。その後多くの人々が Python に貢献していますが、Guido は今日でも Python 製作者の先頭に立っています。

1995 年、Guido は米国ヴァージニア州レストンにある Corporation for National Reserch Initiatives (CNRI, https://www.cnri.reston.va.us/ 参照) で Python の開発に携わり、いくつかのバージョンをリリースしました。

2000 年 3 月、Guido と Python のコア開発チームは BeOpen.com に移り、BeOpen PythonLabs チームを結成しました。同年 10 月、PythonLabs チームは Digital Creations (現在の Zope Corporation, https://www.zope.org/ 参照) に移りました。そして 2001 年、Python に関する知的財産を保有するための非営利組織 Python Software Foundation (PSF、https://www.python.org/psf/ 参照) を立ち上げました。このとき Zope Corporation は PSF の賛助会員になりました。

Python のリリースは全てオープンソース (オープンソースの定義は https://opensource.org/ を参照してください) です。歴史的にみて、ごく一部を除くほとんどの Python リリースは GPL 互換になっています; 各リリースについては下表にまとめてあります。

リリース	ベース	西暦年	権利	GPL 互換
0.9.0 - 1.2	n/a	1991-1995	CWI	yes
1.3 - 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 以降	2.1.1	2001-現在	PSF	yes

注釈: 「GPL 互換」という表現は、Python が GPL で配布されているという意味ではありません。Python のライセンスは全て、GPL と違い、変更したバージョンを配布する際に変更をオープンソースにしなくてもかまいません。GPL 互換のライセンスの下では、GPL でリリースされている他のソフトウェアと Python を組み合わせられますが、それ以外のライセンスではそうではありません。

Guido の指示の下、これらのリリースを可能にしてくださった多くのボランティアのみなさんに感謝します。

C.2 Terms and conditions for accessing or otherwise using Python

Python software and documentation are licensed under the *PSF License Agreement*.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Agreement and the *Zero-Clause BSD license*.

Some software incorporated into Python is under different licenses. The licenses are listed with code falling under that license. See *Licenses and Acknowledgements for Incorporated Software* for an incomplete list of these licenses.

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.8.19

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using Python
   3.8.19 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 3.8.19 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All Rights
   Reserved" are retained in Python 3.8.19 alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.8.19 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee hereby
   agrees to include in any such work a brief summary of the changes made to Python
   3.8.19.

4. PSF is making Python 3.8.19 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR

WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 3.8.19 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.8.19 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.8.19, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 3.8.19, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

(次のページに続く)

(前のページからの続き)

```
5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   Internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
```

(次のページに続く)

(前のページからの続き)

```
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.8.19 DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The `_random` module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
```

(次のページに続く)

(前のページからの続き)

```
   permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 ソケット

The *socket* module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
```

(次のページに続く)

(前のページからの続き)

```
SUCH DAMAGE.
```

C.3.3 Asynchronous socket services

The *asynchat* and *asyncore* modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Cookie management

The *http.cookies* module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
```

(次のページに続く)

(前のページからの続き)

```
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 Execution tracing

The *trace* module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode and UUdecode functions

The *uu* module contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
```

(次のページに続く)

(前のページからの続き)

```
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML Remote Procedure Calls

The *xmlrpc.client* module contains the following notice:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

test_epoll モジュールは次の告知を含んでいます:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

select モジュールは kqueue インターフェースについての次の告知を含んでいます:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
```

(次のページに続く)

(前のページからの続き)

```
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

The file `Python/pyhash.c` contains Marek Majkowski' implementation of Dan Bernstein's SipHash24 algorithm. It contains the following note:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod と dtoa

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from http://www.netlib.org/fp/. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
```

(次のページに続く)

(前のページからの続き)

```
* or modification of this software and in all copies of the supporting
* documentation for such software.
*
* THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
* WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
* REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
* OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
*
************************************************************/
```

C.3.12 OpenSSL

The modules *hashlib*, *posix*, *ssl*, *crypt* use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and Mac OS X installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here:

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   *
   * 1. Redistributions of source code must retain the above copyright
   *    notice, this list of conditions and the following disclaimer.
   *
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in
   *    the documentation and/or other materials provided with the
   *    distribution.
   *
   * 3. All advertising materials mentioning features or use of this
   *    software must display the following acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
   *
   * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
```

(次のページに続く)

(前のページからの続き)

```
    *    endorse or promote products derived from this software without
    *    prior written permission. For written permission, please contact
    *    openssl-core@openssl.org.
    *
    * 5. Products derived from this software may not be called "OpenSSL"
    *    nor may "OpenSSL" appear in their names without prior written
    *    permission of the OpenSSL Project.
    *
    * 6. Redistributions of any form whatsoever must retain the following
    *    acknowledgment:
    *    "This product includes software developed by the OpenSSL Project
    *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
    *
    * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
    * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
    * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
    * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
    * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
    * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
    * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
    * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
    * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
    * OF THE POSSIBILITY OF SUCH DAMAGE.
    * ====================================================================
    *
    * This product includes cryptographic software written by Eric Young
    * (eay@cryptsoft.com).  This product includes software written by Tim
    * Hudson (tjh@cryptsoft.com).
    *
    */

Original SSLeay License
-----------------------

   /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
    * All rights reserved.
    *
    * This package is an SSL implementation written
    * by Eric Young (eay@cryptsoft.com).
    * The implementation was written so as to conform with Netscapes SSL.
    *
    * This library is free for commercial and non-commercial use as long as
    * the following conditions are aheared to.  The following conditions
    * apply to all code found in this distribution, be it the RC4, RSA,
    * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
    * included with this distribution is covered by the same copyright terms
    * except that the holder is Tim Hudson (tjh@cryptsoft.com).
    *
    * Copyright remains Eric Young's, and as such any Copyright notices in
```

(次のページに続く)

(前のページからの続き)

```
 * the code are not to be removed.
 * If this package is used in a product, Eric Young should be given attribution
 * as the author of the parts of the library used.
 * This can be in the form of a textual message at program startup or
 * in documentation (online or textual) provided with the package.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *    "This product includes cryptographic software written by
 *     Eric Young (eay@cryptsoft.com)"
 *    The word 'cryptographic' can be left out if the rouines from the library
 *    being used are not cryptographic related :-).
 * 4. If you include any Windows specific code (or a derivative thereof) from
 *    the apps directory (application code) you must include an acknowledgement:
 *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
 *
 * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * The licence and distribution terms for any publically available version or
 * derivative of this code cannot be changed.  i.e. this code cannot simply be
 * copied and put under another distribution licence
 * [including the GNU Public Licence.]
 */
```

C.3.13 expat

The pyexpat extension is built using an included copy of the expat sources unless the build is configured --with-system-expat:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

The _ctypes extension is built using an included copy of the libffi sources unless the build is configured --with-system-libffi:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
```

(次のページに続く)

(前のページからの続き)

```
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

The *zlib* extension is built using an included copy of the zlib sources if the zlib version found on the system is too old to be used for the build:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

The implementation of the hash table used by the *tracemalloc* で使用しているハッシュテーブルの実装は、cfuhash プロジェクトのものに基づきます:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
```

(次のページに続く)

(前のページからの続き)

```
   notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

The `_decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2016 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
```

(次のページに続く)

(前のページからの続き)

```
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 W3C C14N test suite

The C14N 2.0 test suite in the *test* package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at https://www.w3.org/TR/xml-c14n2-testcases/ and is distributed under the 3-clause BSD license:

> Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang), All Rights Reserved.
>
> Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
>
> - Redistributions of works must retain the original copyright notice, this list of conditions and the following disclaimer.
> - Redistributions in binary form must reproduce the original copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
> - Neither the name of the W3C nor the names of its contributors may be used to endorse or promote products derived from this work without specific prior written permission.
>
> THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

付録

D

COPYRIGHT

Python and this documentation is:

Copyright © 2001-2023 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

ライセンスおよび許諾に関する完全な情報は、歴史とライセンス を参照してください。

参考文献

[Frie09] Friedl, Jeffrey. Mastering Regular Expressions. 3rd ed., O'Reilly Media, 2009. 当書の第三版ではもはや Python についてまったく取り扱っていませんが、初版では良い正規表現を書くことを綿密に取り扱っていました。

[C99] ISO/IEC 9899:1999. "Programming languages -- C." この標準のパブリックドラフトが参照できます: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf。

PYTHON モジュール索引

_

__future__, 2243
__main__, 2194
_dummy_thread, 1099
_thread, 1096

a

abc, 2228
aifc, 1731
argparse, 803
array, 309
ast, 2326
asynchat, 1307
asyncio, 1101
asyncore, 1302
atexit, 2233
audioop, 1727

b

base64, 1442
bdb, 2092
binascii, 1447
binhex, 1446
bisect, 306
builtins, 2194
bz2, 604

c

calendar, 269
cgi, 1540
cgitb, 1549
chunk, 1741
cmath, 373
cmd, 1816
code, 2273
codecs, 200
codeop, 2276
collections, 275
collections.abc, 295
colorsys, 1743
compileall, 2353
concurrent.futures, 1054
configparser, 657
contextlib, 2213
contextvars, 1092
copy, 328
copyreg, 556
cProfile, 2113
crypt (*Unix*), 2414
csv, 649
ctypes, 935
curses (*Unix*), 894
curses.ascii, 919
curses.panel, 922
curses.textpad, 917

d

dataclasses, 2203
datetime, 225
dbm, 562
dbm.dumb, 567
dbm.gnu (*Unix*), 564
dbm.ndbm (*Unix*), 566
decimal, 377
difflib, 163
dis, 2357
distutils, 2143
doctest, 1919
dummy_threading, 1099

e

email, 1327
email.charset, 1394
email.contentmanager, 1365
email.encoders, 1397
email.errors, 1357
email.generator, 1344
email.header, 1390
email.headerregistry, 1358
email.iterators, 1401
email.message, 1329
email.mime, 1387
email.parser, 1339
email.policy, 1348
email.utils, 1398
encodings.idna, 222
encodings.mbcs, 223
encodings.utf_8_sig, 223
ensurepip, 2144
enum, 338
errno, 927

f

faulthandler, 2098
fcntl (*Unix*), 2420
filecmp, 507
fileinput, 498
fnmatch, 517
formatter, 2379
fractions, 410
ftplib, 1612
functools, 452

g

gc, 2244
getopt, 840
getpass, 893
gettext, 1753
glob, 515
grp (*Unix*), 2413
gzip, 600

h

hashlib, 687
heapq, 301
hmac, 700
html, 1453
html.entities, 1459
html.parser, 1454
http, 1601
http.client, 1604
http.cookiejar, 1683
http.cookies, 1678
http.server, 1671

i

imaplib, 1623
imghdr, 1744
imp, 2467
importlib, 2290
importlib.abc, 2293
importlib.machinery, 2303
importlib.metadata, 2316
importlib.resources, 2301
importlib.util, 2309
inspect, 2249
io, 773
ipaddress, 1711
itertools, 435

j

json, 1402
json.tool, 1414

k

keyword, 2342

l

lib2to3, 2065
linecache, 518
locale, 1765
logging, 843
logging.config, 863
logging.handlers, 876
lzma, 609

m

mailbox, 1416
mailcap, 1415
marshal, 561
math, 365
mimetypes, 1438
mmap, 1320
modulefinder, 2285
msilib (*Windows*), 2385
msvcrt (*Windows*), 2392
multiprocessing, 994
multiprocessing.connection, 1032
multiprocessing.dummy, 1037
multiprocessing.managers, 1020
multiprocessing.pool, 1028
multiprocessing.shared_memory, 1048
multiprocessing.sharedctypes, 1018

n

netrc, 678
nis (*Unix*), 2430
nntplib, 1631
numbers, 361

o

operator, 462
optparse, 2433
os, 705
os.path, 491
ossaudiodev (*Linux, FreeBSD*), 1745

p

parser, 2321
pathlib, 471
pdb, 2101
pickle, 535
pickletools, 2375
pipes (*Unix*), 2423
pkgutil, 2282
platform, 923
plistlib, 683
poplib, 1619
posix (*Unix*), 2409
pprint, 329
profile, 2113
pstats, 2115
pty (*Linux*), 2418
pwd (*Unix*), 2410
py_compile, 2350
pyclbr, 2348
pydoc, 1917

q

queue, 1087
quopri, 1450

r

random, 413
re, 139
readline (*Unix*), 185
reprlib, 336
resource (*Unix*), 2424
rlcompleter, 190
runpy, 2287

s

sched, 1085
secrets, 702
select, 1288
selectors, 1297
shelve, 557
shlex, 1822
shutil, 519
signal, 1310
site, 2268
smtpd, 1648
smtplib, 1639
sndhdr, 1745
socket, 1211
socketserver, 1660
spwd (*Unix*), 2412
sqlite3, 568
ssl, 1242
stat, 501
statistics, 421
string, 125
stringprep, 183
struct, 193
subprocess, 1062
sunau, 1734
symbol, 2337
symtable, 2334
sys, 2165
sysconfig, 2189
syslog (*Unix*), 2431

t

tabnanny, 2347
tarfile, 628
telnetlib, 1652
tempfile, 510
termios (*Unix*), 2416
test, 2065
test.support, 2068
test.support.script_helper, 2085
textwrap, 176
threading, 977
time, 790
timeit, 2120
tkinter, 1831
tkinter.scrolledtext (*Tk*), 1875
tkinter.tix, 1868
tkinter.ttk, 1845
token, 2338
tokenize, 2342
trace, 2126
traceback, 2235
tracemalloc, 2129
tty (*Unix*), 2418
turtle, 1775
turtledemo, 1814
types, 321
typing, 1893

U

unicodedata, 181
unittest, 1948
unittest.mock, 1987
urllib, 1563
urllib.error, 1599
urllib.parse, 1588
urllib.request, 1563
urllib.response, 1588
urllib.robotparser, 1600
uu, 1451
uuid, 1656

V

venv, 2146

W

warnings, 2195
wave, 1738
weakref, 312
webbrowser, 1537
winreg (*Windows*), 2395
winsound (*Windows*), 2406
wsgiref, 1550
wsgiref.handlers, 1557
wsgiref.headers, 1553
wsgiref.simple_server, 1554
wsgiref.util, 1550
wsgiref.validate, 1556

X

xdrlib, 679
xml, 1460
xml.dom, 1486
xml.dom.minidom, 1500
xml.dom.pulldom, 1506
xml.etree.ElementTree, 1462
xml.parsers.expat, 1523
xml.parsers.expat.errors, 1532
xml.parsers.expat.model, 1531
xml.sax, 1508
xml.sax.handler, 1510
xml.sax.saxutils, 1517
xml.sax.xmlreader, 1518
xmlrpc.client, 1694
xmlrpc.server, 1704

Z

zipapp, 2156
zipfile, 616
zipimport, 2279
zlib, 595

索引

アルファベット以外

?? in regular expressions, 140
.. in pathnames, 770
..., **2477** ellipsis literal, 35, 106; in doctests, 1928; interpreter prompt, 1924, 2182; placeholder, 180, 330, 336
. (*dot*) in glob-style wildcards, 515; in pathnames, 770, 771; in printf-style formatting, 66, 83; in regular expressions, 140; in string formatting, 128; in Tkinter, 1835
! (*exclamation*) in a command interpreter, 1817; in curses module, 921; in glob-style wildcards, 515, 517; in string formatting, 128; in struct format strings, 195
- (*minus*) binary operator, 39; in doctests, 1930; in glob-style wildcards, 515, 517; in printf-style formatting, 66, 84; in regular expressions, 141; in string formatting, 130; unary operator, 39
! (*pdb command*), 2109
? (*question mark*) in a command interpreter, 1817; in argparse module, 818; in AST grammar, 2327; in glob-style wildcards, 515, 517; in regular expressions, 140; in SQL statements, 580; in struct format strings, 196, 197; replacement character, 204
(*hash*) comment, 2269; in doctests, 1930; in printf-style formatting, 66, 84; in regular expressions, 148; in string formatting, 130
$ (*dollar*) environment variables expansion, 494; in regular expressions, 140; in template strings, 136; interpolation in configuration files, 662
% (*percent*) datetime format, 265, 795, 797; environment variables expansion (*Windows*), 494, 2398; interpolation in configuration files, 662; printf-style formatting, 65, 83; 演算子, 39
& (*ampersand*) 演算子, 41
(? in regular expressions, 142
(?! in regular expressions, 143
(?# in regular expressions, 143
() (*parentheses*) in printf-style formatting, 66, 83; in regular expressions, 142
(?: in regular expressions, 142
(?<! in regular expressions, 144
(?<= in regular expressions, 143
(?= in regular expressions, 143
(?P< in regular expressions, 143
(?P= in regular expressions, 143
*? in regular expressions, 140
* (*asterisk*) in argparse module, 818; in AST grammar, 2327; in glob-style wildcards, 515, 517; in printf-style formatting, 66, 83; in regular expressions, 140; 演算子, 39
** in glob-style wildcards, 516; 演算子, 39
+? in regular expressions, 140
+ (*plus*) binary operator, 39; in argparse module, 818; in doctests, 1930; in printf-style formatting, 66, 84; in regular expressions, 140; in string formatting, 130; unary operator, 39
, (*comma*) in string formatting, 131
/ (*slash*) in pathnames, 771; 演算子, 39
// 演算子, 39
2-digit years, 790
2to3, **2477**
: (*colon*) in SQL statements, 580; in string formatting, 128

    path separator (*POSIX*), 771
; (*semicolon*), 771
< (*less*)
    in string formatting, 130
    in struct format strings, 195
    演算子, 38
<<
    演算子, 41
<=
    演算子, 38
<BLANKLINE>, 1927
!=
    演算子, 38
= (*equals*)
    in string formatting, 130
    in struct format strings, 195
==
    演算子, 38
> (*greater*)
    in string formatting, 130
    in struct format strings, 195
    演算子, 38
>=
    演算子, 38
>>
    演算子, 41
>>>, **2477**
    interpreter prompt, 1924, 2182
@ (*at*)
    in struct format strings, 195
[] (*square brackets*)
    in glob-style wildcards, 515, 517
    in regular expressions, 141
    in string formatting, 128
\ (*backslash*)
    escape sequence, 204
    in pathnames (*Windows*), 771
    in regular expressions, 141, 144
\\
    in regular expressions, 145
\A
    in regular expressions, 144
\a
    in regular expressions, 145
\B
    in regular expressions, 145
\b
    in regular expressions, 144, 145
\D
    in regular expressions, 145
\d
    in regular expressions, 145
\f
    in regular expressions, 145
\g
    in regular expressions, 150
\N
    escape sequence, 204
    in regular expressions, 145
\n
    in regular expressions, 145
\r
    in regular expressions, 145
\S
    in regular expressions, 145
\s
    in regular expressions, 145
\t
    in regular expressions, 145
\U
    escape sequence, 204
    in regular expressions, 145
\u
    escape sequence, 204
    in regular expressions, 145
\v
    in regular expressions, 145
\W
    in regular expressions, 145
\w
    in regular expressions, 145
\x
    escape sequence, 204
    in regular expressions, 145
\Z
    in regular expressions, 145
^ (*caret*)
    in curses module, 921
    in regular expressions, 140, 141
    in string formatting, 130
    marker, 1927, 2236
    演算子, 41
_ (*underscore*)
    gettext, 1754
    in string formatting, 131
__abs__() (*operator* モジュール), 463
__add__() (*operator* モジュール), 463
__and__() (*operator* モジュール), 463
__bases__ (*class* の属性), 107
__breakpointhook__ (*sys* モジュール), 2170
__bytes__() (*email.message.EmailMessage* のメソッド), 1330
__bytes__() (*email.message.Message* のメソッド), 1378
__call__() (*email.headerregistry.HeaderRegistry* のメソッド), 1364
__call__() (*weakref.finalize* のメソッド), 316
__callback__ (*weakref.ref* の属性), 314
__cause__ (*traceback.TracebackException* の属性), 2238
__ceil__() (*fractions.Fraction* のメソッド), 412
__class__ (*instance* の属性), 107
__class__ (*unittest.mock.Mock* の属性), 1998
__code__ (*function object attribute*), 106
__concat__() (*operator* モジュール), 464
__contains__() (*email.message.EmailMessage* のメソッド), 1331
__contains__() (*email.message.Message* のメソッド), 1380
__contains__() (*mailbox.Mailbox* のメソッド), 1419
__contains__() (*operator* モジュール), 464
__context__ (*traceback.TracebackException* の属性), 2238
__copy__() (*copy protocol*), 329
__debug__ (組み込み変数), 35
__deepcopy__() (*copy protocol*), 329
__del__() (*io.IOBase* のメソッド), 779
__delitem__() (*email.message.EmailMessage* のメソッド), 1332
__delitem__() (*email.message.Message* のメソッド), 1381
__delitem__() (*mailbox.Mailbox* のメソッド), 1418
__delitem__() (*mailbox.MH* のメソッド), 1424
__delitem__() (*operator* モジュール), 464
__dict__ (*object* の属性), 107
__dir__() (*unittest.mock.Mock* のメソッド), 1994
__displayhook__ (*sys* モジュール), 2170
__doc__ (*types.ModuleType* の属性), 324
__enter__() (*contextmanager* のメソッド), 103
__enter__() (*winreg.PyHKEY* のメソッド), 2406
__eq__() (*email.charset.Charset* のメソッド), 1396
__eq__() (*email.header.Header* のメソッド), 1393
__eq__() (*instance method*), 38
__eq__() (*memoryview* のメソッド), 88
__eq__() (*operator* モジュール), 462
__excepthook__ (*sys* モジュール), 2170
__exit__() (*contextmanager* のメソッド), 103
__exit__() (*winreg.PyHKEY* のメソッド), 2406
__floor__() (*fractions.Fraction* のメソッド), 412
__floordiv__() (*operator* モジュール), 463
__format__, 14

__format__() (*datetime.date* のメソッド), 236
__format__() (*datetime.datetime* のメソッド), 249
__format__() (*datetime.time* のメソッド), 255
__fspath__() (*os.PathLike* のメソッド), 708
__future__, **2482**
__future__ (モジュール), 2243
__ge__() (*instance method*), 38
__ge__() (*operator* モジュール), 462
__getitem__() (*email.headerregistry.HeaderRegistry* のメソッド), 1364
__getitem__() (*email.message.EmailMessage* のメソッド), 1331
__getitem__() (*email.message.Message* のメソッド), 1380
__getitem__() (*mailbox.Mailbox* のメソッド), 1419
__getitem__() (*operator* モジュール), 464
__getitem__() (*re.Match* のメソッド), 155
__getnewargs__() (*object* のメソッド), 544
__getnewargs_ex__() (*object* のメソッド), 544
__getstate__() (*copy protocol*), 550
__getstate__() (*object* のメソッド), 545
__gt__() (*instance method*), 38
__gt__() (*operator* モジュール), 462
__iadd__() (*operator* モジュール), 468
__iand__() (*operator* モジュール), 468
__iconcat__() (*operator* モジュール), 468
__ifloordiv__() (*operator* モジュール), 469
__ilshift__() (*operator* モジュール), 469
__imatmul__() (*operator* モジュール), 469
__imod__() (*operator* モジュール), 469
__import__() (*importlib* モジュール), 2291
__import__() (組み込み関数), 32
__imul__() (*operator* モジュール), 469
__index__() (*operator* モジュール), 463
__init__() (*difflib.HtmlDiff* のメソッド), 164
__init__() (*logging.Handler* のメソッド), 849
__interactivehook__ (*sys* モジュール), 2178
__inv__() (*operator* モジュール), 463
__invert__() (*operator* モジュール), 463
__ior__() (*operator* モジュール), 469
__ipow__() (*operator* モジュール), 469
__irshift__() (*operator* モジュール), 469
__isub__() (*operator* モジュール), 469
__iter__() (*container* のメソッド), 46
__iter__() (*iterator* のメソッド), 46
__iter__() (*mailbox.Mailbox* のメソッド), 1418
__iter__() (*unittest.TestSuite* のメソッド), 1974
__itruediv__() (*operator* モジュール), 469
__ixor__() (*operator* モジュール), 469
__le__() (*instance method*), 38
__le__() (*operator* モジュール), 462
__len__() (*email.message.EmailMessage* のメソッド), 1331
__len__() (*email.message.Message* のメソッド), 1380
__len__() (*mailbox.Mailbox* のメソッド), 1419
__loader__ (*types.ModuleType* の属性), 324
__lshift__() (*operator* モジュール), 463
__lt__() (*instance method*), 38
__lt__() (*operator* モジュール), 462
__main__
    モジュール, 2287, 2288
__main__ (モジュール), 2194
__matmul__() (*operator* モジュール), 463
__missing__(), 99
__missing__() (*collections.defaultdict* のメソッド), 286
__mod__() (*operator* モジュール), 463
__mro__ (*class* の属性), 107
__mul__() (*operator* モジュール), 463
__name__ (*definition* の属性), 107
__name__ (*types.ModuleType* の属性), 325
__ne__() (*email.charset.Charset* のメソッド), 1396
__ne__() (*email.header.Header* のメソッド), 1393
__ne__() (*instance method*), 38
__ne__() (*operator* モジュール), 462
__neg__() (*operator* モジュール), 463
__next__() (*csv.csvreader* のメソッド), 655
__next__() (*iterator* のメソッド), 46
__not__() (*operator* モジュール), 462
__or__() (*operator* モジュール), 464
__package__ (*types.ModuleType* の属性), 325
__pos__() (*operator* モジュール), 464
__pow__() (*operator* モジュール), 464
__qualname__ (*definition* の属性), 107
__reduce__() (*object* のメソッド), 545
__reduce_ex__() (*object* のメソッド), 546
__repr__() (*multiprocessing.managers.BaseProxy* のメソッド), 1028
__repr__() (*netrc.netrc* のメソッド), 679
__round__() (*fractions.Fraction* のメソッド), 412
__rshift__() (*operator* モジュール), 464
__setitem__() (*email.message.EmailMessage* のメソッド), 1331
__setitem__() (*email.message.Message* のメソッド), 1380
__setitem__() (*mailbox.Mailbox* のメソッド), 1418
__setitem__() (*mailbox.Maildir* のメソッド), 1422
__setitem__() (*operator* モジュール), 465
__setstate__() (*copy protocol*), 550
__setstate__() (*object* のメソッド), 545
__slots__, **2491**
__spec__ (*types.ModuleType* の属性), 325
__stderr__ (*sys* モジュール), 2187
__stdin__ (*sys* モジュール), 2187
__stdout__ (*sys* モジュール), 2187
__str__() (*datetime.date* のメソッド), 236
__str__() (*datetime.datetime* のメソッド), 248
__str__() (*datetime.time* のメソッド), 254
__str__() (*email.charset.Charset* のメソッド), 1396
__str__() (*email.header.Header* のメソッド), 1393
__str__() (*email.headerregistry.Address* のメソッド), 1364
__str__() (*email.headerregistry.Group* のメソッド), 1365
__str__() (*email.message.EmailMessage* のメソッド), 1330
__str__() (*email.message.Message* のメソッド), 1377
__str__() (*multiprocessing.managers.BaseProxy* のメソッド), 1028
__sub__() (*operator* モジュール), 464
__subclasses__() (*class* のメソッド), 108
__subclasshook__() (*abc.ABCMeta* のメソッド), 2229
__suppress_context__ (*traceback.TracebackException* の属性), 2238
__truediv__() (*operator* モジュール), 464
__unraisablehook__ (*sys* モジュール), 2170
__xor__() (*operator* モジュール), 464
anonymous (*ctypes.Structure* の属性), 973
_asdict() (*collections.somenamedtuple* のメソッド), 289
_b_base_ (*ctypes._CData* の属性), 969
_b_needsfree_ (*ctypes._CData* の属性), 969
_callmethod() (*multiprocessing.managers.BaseProxy* のメソッド), 1027
_CData (*ctypes* のクラス), 968
_clear_type_cache() (*sys* モジュール), 2167
_current_frames() (*sys* モジュール), 2167
_debugmallocstats() (*sys* モジュール), 2168
_dummy_thread (モジュール), 1099
_enablelegacywindowsfsencoding() (*sys* モジュール), 2186
_exit() (*os* モジュール), 756
_field_defaults (*collections.somenamedtuple* の属性), 290
_fields (*ast.AST* の属性), 2327
_fields (*collections.somenamedtuple* の属性), 290
fields (*ctypes.Structure* の属性), 973
_flush() (*wsgiref.handlers.BaseHandler* のメソッド), 1558
_FuncPtr (*ctypes* のクラス), 961
_get_child_mock() (*unittest.mock.Mock* のメソッド), 1994
_getframe() (*sys* モジュール), 2175

_getvalue() (*multiprocessing.managers.BaseProxy* のメソッド), 1028
_handle (*ctypes.PyDLL* の属性), 960
length (*ctypes.Array* の属性), 974
_locale
    モジュール, 1765
_make() (*collections.somenamedtuple* のクラスメソッド), 289
_makeResult() (*unittest.TextTestRunner* のメソッド), 1981
_name (*ctypes.PyDLL* の属性), 960
_objects (*ctypes._CData* の属性), 969
pack (*ctypes.Structure* の属性), 973
_parse() (*gettext.NullTranslations* のメソッド), 1757
_Pointer (*ctypes* のクラス), 975
_replace() (*collections.somenamedtuple* のメソッド), 290
_setroot() (*xml.etree.ElementTree.ElementTree* のメソッド), 1480
_SimpleCData (*ctypes* のクラス), 969
_structure() (*email.iterators* モジュール), 1401
_thread (モジュール), 1096
type (*ctypes._Pointer* の属性), 975
type (*ctypes.Array* の属性), 974
_write() (*wsgiref.handlers.BaseHandler* のメソッド), 1558
_xoptions (*sys* モジュール), 2189
{} (*curly brackets*)
    in regular expressions, 141
    in string formatting, 128
| (*vertical bar*)
    in regular expressions, 142
    演算子, 41
~ (*tilde*)
    home directory expansion, 493
    演算子, 41
オブジェクト
    Boolean, 39
    bytearray, 50, 68, 70
    bytes, 68
    complex number, 39
    dictionary, 97
    floating point, 39
    integer, 39
    io.StringIO, 55
    list, 50, 51
    mapping, 97
    memoryview, 68
    method, 105
    numeric, 39
    range, 53
    sequence, 47
    set, 94
    socket, 1211
    string, 55
    traceback, 2170, 2235
    tuple, 50, 52
    type, 30
モジュール
    __main__, 2287, 2288
    _locale, 1765
    array, 68
    base64, 1447
    bdb, 2101
    binhex, 1447
    cmd, 2101
    copy, 556
    crypt, 2411
    dbm.gnu, 559
    dbm.ndbm, 559
    errno, 116
    glob, 517
    imp, 32
    math, 40, 376
    os, 2409
    pickle, 329, 556, 557, 561
    pty, 720
    pwd, 493
    pyexpat, 1523
    re, 56, 517
    shelve, 561
    signal, 1099
    sitecustomize, 2270
    socket, 1537
    stat, 742
    string, 1771
    struct, 1236
    sys, 23
    types, 106
    urllib.request, 1604
    usercustomize, 2270
    uu, 1447
環境変数
    AUDIODEV, 1746
    BROWSER, 1537, 1538
    COLS, 902
    COLUMNS, 902
    COMSPEC, 764, 1068
    HOME, 493, 494
    HOMEDRIVE, 493
    HOMEPATH, 493
    http_proxy, 1564, 1582
    IDLESTARTUP, 1884
    KDEDIR, 1539
    LANG, 1753, 1756, 1766, 1769
    LANGUAGE, 1753, 1756
    LC_ALL, 1753, 1756
    LC_MESSAGES, 1753, 1756
    LINES, 897, 902
    LNAME, 894
    LOGNAME, 710, 894
    MIXERDEV, 1746
    no_proxy, 1568
    PAGER, 1918
    PATH, 755, 761, 762, 771, 1537, 1546, 1548, 2269
    POSIXLY_CORRECT, 841
    PYTHON_DOM, 1488
    PYTHONASYNCIODEBUG, 1161, 1207
    PYTHONBREAKPOINT, 2168
    PYTHONDEVMODE, 2086
    PYTHONDOCS, 1918
    PYTHONDONTWRITEBYTECODE, 2169
    PYTHONFAULTHANDLER, 2098
    PYTHONHOME, 2085
    PYTHONINTMAXSTRDIGITS, 110, 2178
    PYTHONIOENCODING, 2186
    PYTHONLEGACYWINDOWSFSENCODING, 2186
    PYTHONLEGACYWINDOWSSTDIO, 2186
    PYTHONNOUSERSITE, 2271
    PYTHONPATH, 1546, 2085, 2180
    PYTHONPYCACHEPREFIX, 2169
    PYTHONSTARTUP, 189, 1884, 2179, 2270
    PYTHONTRACEMALLOC, 2129, 2130, 2135
    PYTHONUSERBASE, 2271, 2272
    PYTHONUSERSITE, 2085
    PYTHONUTF8, 2186
    PYTHONWARNINGS, 2197, 2198
    SOURCE_DATE_EPOCH, 2351, 2354
    SSL_CERT_FILE, 1287
    SSL_CERT_PATH, 1287
    SSLKEYLOGFILE, 1244, 1245
    SystemRoot, 1070
    TEMP, 513
    TERM, 901
    TMP, 513
    TMPDIR, 513
    TZ, 799, 800
    USER, 894
    USERNAME, 710, 894

    USERPROFILE, 493
    VIRTUAL_ENV, 2148
組み込み関数
    compile, 106, 323, 2324
    complex, 39
    eval, 106, 331, 332, 2324
    exec, 12, 106, 2324
    float, 39
    hash, 50
    int, 39
    len, 47, 97
    max, 47
    min, 47
    slice, 2374
    type, 106

A

-a
    pickletools command line option, 2376
A (*re* モジュール), 147
a2b_base64() (*binascii* モジュール), 1448
a2b_hex() (*binascii* モジュール), 1449
a2b_hqx() (*binascii* モジュール), 1448
a2b_qp() (*binascii* モジュール), 1448
a2b_uu() (*binascii* モジュール), 1447
a85decode() (*base64* モジュール), 1444
a85encode() (*base64* モジュール), 1444
ABC (*abc* のクラス), 2228
abc (モジュール), 2228
ABCMeta (*abc* のクラス), 2228
abiflags (*sys* モジュール), 2165
abort() (*asyncio.DatagramTransport* のメソッド), 1179
abort() (*asyncio.WriteTransport* のメソッド), 1178
abort() (*ftplib.FTP* のメソッド), 1615
abort() (*os* モジュール), 754
abort() (*threading.Barrier* のメソッド), 993
above() (*curses.panel.Panel* のメソッド), 922
ABOVE_NORMAL_PRIORITY_CLASS (*subprocess* モジュール), 1076
abs() (*decimal.Context* のメソッド), 394
abs() (*operator* モジュール), 463
abs() (組み込み関数), 5
AbsoluteLinkError, 631
AbsolutePathError, 631
abspath() (*os.path* モジュール), 492
abstract base class, **2477**
AbstractAsyncContextManager (*contextlib* のクラス), 2213
AbstractBasicAuthHandler (*urllib.request* のクラス), 1568
AbstractChildWatcher (*asyncio* のクラス), 1192
abstractclassmethod() (*abc* モジュール), 2232
AbstractContextManager (*contextlib* のクラス), 2213
AbstractDigestAuthHandler (*urllib.request* のクラス), 1569
AbstractEventLoop (*asyncio* のクラス), 1166
AbstractEventLoopPolicy (*asyncio* のクラス), 1191
AbstractFormatter (*formatter* のクラス), 2382
abstractmethod() (*abc* モジュール), 2230
abstractproperty() (*abc* モジュール), 2232
AbstractSet (*typing* のクラス), 1904
abstractstaticmethod() (*abc* モジュール), 2232
AbstractWriter (*formatter* のクラス), 2383
accept() (*asyncore.dispatcher* のメソッド), 1305
accept() (*multiprocessing.connection.Listener* のメソッド), 1033
accept() (*socket.socket* のメソッド), 1228
access() (*os* モジュール), 727
accumulate() (*itertools* モジュール), 437
aclose() (*contextlib.AsyncExitStack* のメソッド), 2221
acos() (*cmath* モジュール), 374
acos() (*math* モジュール), 370
acosh() (*cmath* モジュール), 375
acosh() (*math* モジュール), 371
acquire() (*_thread.lock* のメソッド), 1098
acquire() (*asyncio.Condition* のメソッド), 1129
acquire() (*asyncio.Lock* のメソッド), 1127
acquire() (*asyncio.Semaphore* のメソッド), 1131
acquire() (*logging.Handler* のメソッド), 849
acquire() (*multiprocessing.Lock* のメソッド), 1014
acquire() (*multiprocessing.RLock* のメソッド), 1015
acquire() (*threading.Condition* のメソッド), 987
acquire() (*threading.Lock* のメソッド), 984
acquire() (*threading.RLock* のメソッド), 985
acquire() (*threading.Semaphore* のメソッド), 989
acquire_lock() (*imp* モジュール), 2471
Action (*argparse* のクラス), 826
action (*optparse.Option* の属性), 2449
ACTIONS (*optparse.Option* の属性), 2465
active_children() (*multiprocessing* モジュール), 1009
active_count() (*threading* モジュール), 977
add() (*audioop* モジュール), 1727
add() (*decimal.Context* のメソッド), 394
add() (*frozenset* のメソッド), 97
add() (*mailbox.Mailbox* のメソッド), 1417
add() (*mailbox.Maildir* のメソッド), 1422
add() (*msilib.RadioButtonGroup* のメソッド), 2391
add() (*operator* モジュール), 463
add() (*pstats.Stats* のメソッド), 2115
add() (*tarfile.TarFile* のメソッド), 636
add() (*tkinter.ttk.Notebook* のメソッド), 1854
add_alias() (*email.charset* モジュール), 1396
add_alternative() (*email.message.EmailMessage* のメソッド), 1338
add_argument() (*argparse.ArgumentParser* のメソッド), 814
add_argument_group() (*argparse.ArgumentParser* のメソッド), 834
add_attachment() (*email.message.EmailMessage* のメソッド), 1338
add_cgi_vars() (*wsgiref.handlers.BaseHandler* のメソッド), 1558
add_charset() (*email.charset* モジュール), 1396
add_child_handler() (*asyncio.AbstractChildWatcher* のメソッド), 1193
add_codec() (*email.charset* モジュール), 1396
add_cookie_header() (*http.cookiejar.CookieJar* のメソッド), 1685
add_data() (*msilib* モジュール), 2386
add_dll_directory() (*os* モジュール), 754
add_done_callback() (*asyncio.Future* のメソッド), 1172
add_done_callback() (*asyncio.Task* のメソッド), 1116
add_done_callback() (*concurrent.futures.Future* のメソッド), 1059
add_fallback() (*gettext.NullTranslations* のメソッド), 1757
add_file() (*msilib.Directory* のメソッド), 2390
add_flag() (*mailbox.MaildirMessage* のメソッド), 1428
add_flag() (*mailbox.mboxMessage* のメソッド), 1430
add_flag() (*mailbox.MMDFMessage* のメソッド), 1435
add_flowing_data() (*formatter.formatter* のメソッド), 2380
add_folder() (*mailbox.Maildir* のメソッド), 1422
add_folder() (*mailbox.MH* のメソッド), 1424
add_get_handler() (*email.contentmanager.ContentManager* のメソッド), 1366
add_handler() (*urllib.request.OpenerDirector* のメソッド), 1572
add_header() (*email.message.EmailMessage* のメソッド), 1332
add_header() (*email.message.Message* のメソッド), 1381
add_header() (*urllib.request.Request* のメソッド), 1571
add_header() (*wsgiref.headers.Headers* のメソッド), 1553
add_history() (*readline* モジュール), 187
add_hor_rule() (*formatter.formatter* のメソッド), 2380
add_label() (*mailbox.BabylMessage* のメソッド), 1433

add_label_data() (formatter.formatter のメソッド), 2380
add_line_break() (formatter.formatter のメソッド), 2380
add_literal_data() (formatter.formatter のメソッド), 2380
add_mutually_exclusive_group() (argparse.ArgumentParser のメソッド), 835
add_option() (optparse.OptionParser のメソッド), 2448
add_parent() (urllib.request.BaseHandler のメソッド), 1573
add_password() (urllib.request.HTTPPasswordMgr のメソッド), 1576
add_password() (urllib.request.HTTPPasswordMgrWithPriorAuth のメソッド), 1577
add_reader() (asyncio.loop のメソッド), 1155
add_related() (email.message.EmailMessage のメソッド), 1338
add_section() (configparser.ConfigParser のメソッド), 673
add_section() (configparser.RawConfigParser のメソッド), 676
add_sequence() (mailbox.MHMessage のメソッド), 1432
add_set_handler() (email.contentmanager.ContentManager のメソッド), 1366
add_signal_handler() (asyncio.loop のメソッド), 1158
add_stream() (msilib モジュール), 2386
add_subparsers() (argparse.ArgumentParser のメソッド), 830
add_tables() (msilib モジュール), 2386
add_type() (mimetypes モジュール), 1440
add_unredirected_header() (urllib.request.Request のメソッド), 1571
add_writer() (asyncio.loop のメソッド), 1155
addAsyncCleanup() (unittest.IsolatedAsyncioTestCase のメソッド), 1971
addaudithook() (sys モジュール), 2165
addch() (curses.window のメソッド), 903
addClassCleanup() (unittest.TestCase のクラスメソッド), 1971
addCleanup() (unittest.TestCase のメソッド), 1970
addcomponent() (turtle.Shape のメソッド), 1810
addError() (unittest.TestResult のメソッド), 1980
addExpectedFailure() (unittest.TestResult のメソッド), 1980
addFailure() (unittest.TestResult のメソッド), 1980
addfile() (tarfile.TarFile のメソッド), 636
addFilter() (logging.Handler のメソッド), 849
addFilter() (logging.Logger のメソッド), 847
addHandler() (logging.Logger のメソッド), 847
addLevelName() (logging モジュール), 859
addModuleCleanup() (unittest モジュール), 1985
addnstr() (curses.window のメソッド), 903
AddPackagePath() (modulefinder モジュール), 2285
addr (smtpd.SMTPChannel の属性), 1651
addr_spec (email.headerregistry.Address の属性), 1364
Address (email.headerregistry のクラス), 1364
address (email.headerregistry.SingleAddressHeader の属性), 1361
address (multiprocessing.connection.Listener の属性), 1033
address (multiprocessing.managers.BaseManager の属性), 1022
address_exclude() (ipaddress.IPv4Network のメソッド), 1719
address_exclude() (ipaddress.IPv6Network のメソッド), 1722
address_family (socketserver.BaseServer の属性), 1664
address_string() (http.server.BaseHTTPRequestHandler のメソッド), 1675
addresses (email.headerregistry.AddressHeader の属性), 1361
addresses (email.headerregistry.Group の属性), 1365
AddressHeader (email.headerregistry のクラス), 1361
addressof() (ctypes モジュール), 965
AddressValueError, 1726
addshape() (turtle モジュール), 1807
addsitedir() (site モジュール), 2271
addSkip() (unittest.TestResult のメソッド), 1980
addstr() (curses.window のメソッド), 903
addSubTest() (unittest.TestResult のメソッド), 1980
addSuccess() (unittest.TestResult のメソッド), 1980
addTest() (unittest.TestSuite のメソッド), 1973
addTests() (unittest.TestSuite のメソッド), 1973
addTypeEqualityFunc() (unittest.TestCase のメソッド), 1968
addUnexpectedSuccess() (unittest.TestResult のメソッド), 1980
adjust_int_max_str_digits() (test.support モジュール), 2083
adjusted() (decimal.Decimal のメソッド), 384
adler32() (zlib モジュール), 595
ADPCM, Intel/DVI, 1727
adpcm2lin() (audioop モジュール), 1727
AF_ALG (socket モジュール), 1218
AF_CAN (socket モジュール), 1216
AF_INET (socket モジュール), 1215
AF_INET6 (socket モジュール), 1215
AF_LINK (socket モジュール), 1218
AF_PACKET (socket モジュール), 1217
AF_QIPCRTR (socket モジュール), 1218
AF_RDS (socket モジュール), 1217
AF_UNIX (socket モジュール), 1215
AF_VSOCK (socket モジュール), 1218
aifc (モジュール), 1731
aifc() (aifc.aifc のメソッド), 1733
AIFF, 1731, 1741
aiff() (aifc.aifc のメソッド), 1733
AIFF-C, 1731, 1741
alarm() (signal モジュール), 1314
A-LAW, 1733, 1745
a-LAW, 1727
alaw2lin() (audioop モジュール), 1727
ALERT_DESCRIPTION_HANDSHAKE_FAILURE (ssl モジュール), 1258
ALERT_DESCRIPTION_INTERNAL_ERROR (ssl モジュール), 1258
AlertDescription (ssl のクラス), 1258
algorithms_available (hashlib モジュール), 689
algorithms_guaranteed (hashlib モジュール), 689
alias (pdb command), 2108
alignment() (ctypes モジュール), 965
alive (weakref.finalize の属性), 316
all() (組み込み関数), 5
all_errors (ftplib モジュール), 1614
all_features (xml.sax.handler モジュール), 1512
all_frames (tracemalloc.Filter の属性), 2137
all_properties (xml.sax.handler モジュール), 1513
all_suffixes() (importlib.machinery モジュール), 2304
all_tasks() (asyncio モジュール), 1113
all_tasks() (asyncio.Task のクラスメソッド), 1117
allocate_lock() (_thread モジュール), 1097
allow_reuse_address (socketserver.BaseServer の属性), 1664
allowed_domains() (http.cookiejar.DefaultCookiePolicy のメソッド), 1690
alt() (curses.ascii モジュール), 921
ALT_DIGITS (locale モジュール), 1769
altsep (os モジュール), 771
altzone (time モジュール), 802
ALWAYS_EQ (test.support モジュール), 2070
ALWAYS_TYPED_ACTIONS (optparse.Option の属性), 2465
AMPER (token モジュール), 2339
AMPEREQUAL (token モジュール), 2340
and
    演算子, 37, 38
and_() (operator モジュール), 463

--annotate
    pickletools command line option, 2376
annotation, 2477
annotation (inspect.Parameter の属性), 2257
answer_challenge() (multiprocessing.connection モジュール), 1032
anticipate_failure() (test.support モジュール), 2076
Any (typing モジュール), 1913
ANY (unittest.mock モジュール), 2027
any() (組み込み関数), 5
AnyStr (typing モジュール), 1916
api_version (sys モジュール), 2188
apop() (poplib.POP3 のメソッド), 1621
append() (array.array のメソッド), 310
append() (collections.deque のメソッド), 282
append() (email.header.Header のメソッド), 1392
append() (imaplib.IMAP4 のメソッド), 1625
append() (msilib.CAB のメソッド), 2389
append() (pipes.Template のメソッド), 2424
append() (sequence method), 50
append() (xml.etree.ElementTree.Element のメソッド), 1477
append_history_file() (readline モジュール), 186
appendChild() (xml.dom.Node のメソッド), 1491
appendleft() (collections.deque のメソッド), 282
application_uri() (wsgiref.util モジュール), 1550
apply (2to3 fixer), 2060
apply() (multiprocessing.pool.Pool のメソッド), 1029
apply_async() (multiprocessing.pool.Pool のメソッド), 1029
apply_defaults() (inspect.BoundArguments のメソッド), 2259
architecture() (platform モジュール), 923
archive (zipimport.zipimporter の属性), 2281
aRepr (reprlib モジュール), 336
argparse (モジュール), 803
args (BaseException の属性), 114
args (functools.partial の属性), 461
args (inspect.BoundArguments の属性), 2259
args (pdb command), 2107
args (subprocess.CompletedProcess の属性), 1063
args (subprocess.Popen の属性), 1074
args_from_interpreter_flags() (test.support モジュール), 2074
argtypes (ctypes._FuncPtr の属性), 961
argument, 2478
ArgumentDefaultsHelpFormatter (argparse のクラス), 809
ArgumentError, 962
ArgumentParser (argparse のクラス), 805
arguments (inspect.BoundArguments の属性), 2259
argv (sys モジュール), 2166
arithmetic, 39
ArithmeticError, 114
array
    モジュール, 68
array (array のクラス), 309
Array (ctypes のクラス), 974
array (モジュール), 309
Array() (multiprocessing モジュール), 1017
Array() (multiprocessing.managers.SyncManager のメソッド), 1023
Array() (multiprocessing.sharedctypes モジュール), 1018
arrays, 309
arraysize (sqlite3.Cursor の属性), 584
article() (nntplib.NNTP のメソッド), 1637
as_bytes() (email.message.EmailMessage のメソッド), 1330
as_bytes() (email.message.Message のメソッド), 1377
as_completed() (asyncio モジュール), 1112
as_completed() (concurrent.futures モジュール), 1061
as_integer_ratio() (decimal.Decimal のメソッド), 384
as_integer_ratio() (float のメソッド), 43
as_integer_ratio() (fractions.Fraction のメソッド), 411
as_integer_ratio() (int のメソッド), 43
AS_IS (formatter モジュール), 2379
as_posix() (pathlib.PurePath のメソッド), 479
as_string() (email.message.EmailMessage のメソッド), 1329
as_string() (email.message.Message のメソッド), 1377
as_tuple() (decimal.Decimal のメソッド), 384
as_uri() (pathlib.PurePath のメソッド), 479
ASCII (re モジュール), 147
ascii() (curses.ascii モジュール), 921
ascii() (組み込み関数), 6
ascii_letters (string モジュール), 125
ascii_lowercase (string モジュール), 125
ascii_uppercase (string モジュール), 125
asctime() (time モジュール), 792
asdict() (dataclasses モジュール), 2207
asin() (cmath モジュール), 374
asin() (math モジュール), 370
asinh() (cmath モジュール), 375
asinh() (math モジュール), 371
assert
    文, 115
assert_any_await() (unittest.mock.AsyncMock のメソッド), 2003
assert_any_call() (unittest.mock.Mock のメソッド), 1992
assert_awaited() (unittest.mock.AsyncMock のメソッド), 2002
assert_awaited_once() (unittest.mock.AsyncMock のメソッド), 2002
assert_awaited_once_with() (unittest.mock.AsyncMock のメソッド), 2003
assert_awaited_with() (unittest.mock.AsyncMock のメソッド), 2002
assert_called() (unittest.mock.Mock のメソッド), 1991
assert_called_once() (unittest.mock.Mock のメソッド), 1991
assert_called_once_with() (unittest.mock.Mock のメソッド), 1992
assert_called_with() (unittest.mock.Mock のメソッド), 1991
assert_has_awaits() (unittest.mock.AsyncMock のメソッド), 2003
assert_has_calls() (unittest.mock.Mock のメソッド), 1992
assert_line_data() (formatter.formatter のメソッド), 2381
assert_not_awaited() (unittest.mock.AsyncMock のメソッド), 2004
assert_not_called() (unittest.mock.Mock のメソッド), 1992
assert_python_failure() (test.support.script_helper モジュール), 2085
assert_python_ok() (test.support.script_helper モジュール), 2085
assertAlmostEqual() (unittest.TestCase のメソッド), 1966
assertCountEqual() (unittest.TestCase のメソッド), 1967
assertDictEqual() (unittest.TestCase のメソッド), 1969
assertEqual() (unittest.TestCase のメソッド), 1962
assertFalse() (unittest.TestCase のメソッド), 1962
assertGreater() (unittest.TestCase のメソッド), 1967
assertGreaterEqual() (unittest.TestCase のメソッド), 1967
assertIn() (unittest.TestCase のメソッド), 1963
AssertionError, 115
assertIs() (unittest.TestCase のメソッド), 1962
assertIsInstance() (unittest.TestCase のメソッド), 1963
assertIsNone() (unittest.TestCase のメソッド), 1963
assertIsNot() (unittest.TestCase のメソッド), 1962
assertIsNotNone() (unittest.TestCase のメソッド), 1963
assertLess() (unittest.TestCase のメソッド), 1967
assertLessEqual() (unittest.TestCase のメソッド), 1967
assertListEqual() (unittest.TestCase のメソッド), 1969
assertLogs() (unittest.TestCase のメソッド), 1966

assertMultiLineEqual() (*unittest.TestCase* のメソッド), 1968
assertNotAlmostEqual() (*unittest.TestCase* のメソッド), 1966
assertNotEqual() (*unittest.TestCase* のメソッド), 1962
assertNotIn() (*unittest.TestCase* のメソッド), 1963
assertNotIsInstance() (*unittest.TestCase* のメソッド), 1963
assertNotRegex() (*unittest.TestCase* のメソッド), 1967
assertRaises() (*unittest.TestCase* のメソッド), 1963
assertRaisesRegex() (*unittest.TestCase* のメソッド), 1964
assertRegex() (*unittest.TestCase* のメソッド), 1967
asserts (*2to3 fixer*), 2060
assertSequenceEqual() (*unittest.TestCase* のメソッド), 1968
assertSetEqual() (*unittest.TestCase* のメソッド), 1969
assertTrue() (*unittest.TestCase* のメソッド), 1962
assertTupleEqual() (*unittest.TestCase* のメソッド), 1969
assertWarns() (*unittest.TestCase* のメソッド), 1964
assertWarnsRegex() (*unittest.TestCase* のメソッド), 1965
assignment
    slice, 50
    subscript, 50
AST (*ast* のクラス), 2326
ast (モジュール), 2326
astimezone() (*datetime.datetime* のメソッド), 244
astuple() (*dataclasses* モジュール), 2208
ASYNC (*token* モジュール), 2341
async_chat (*asynchat* のクラス), 1307
async_chat.ac_in_buffer_size (*asynchat* モジュール), 1307
async_chat.ac_out_buffer_size (*asynchat* モジュール), 1307
AsyncContextManager (*typing* のクラス), 1906
asynccontextmanager() (*contextlib* モジュール), 2214
AsyncExitStack (*contextlib* のクラス), 2220
AsyncGenerator (*collections.abc* のクラス), 300
AsyncGenerator (*typing* のクラス), 1907
AsyncGeneratorType (*types* モジュール), 323
asynchat (モジュール), 1307
asynchronous context manager, **2478**
asynchronous generator, **2478**
asynchronous generator iterator, **2478**
asynchronous iterable, **2478**
asynchronous iterator, **2479**
asyncio (モジュール), 1101
asyncio.subprocess.DEVNULL (*asyncio* モジュール), 1134
asyncio.subprocess.PIPE (*asyncio* モジュール), 1134
asyncio.subprocess.Process (*asyncio* のクラス), 1134
asyncio.subprocess.STDOUT (*asyncio* モジュール), 1134
AsyncIterable (*collections.abc* のクラス), 299
AsyncIterable (*typing* のクラス), 1906
AsyncIterator (*collections.abc* のクラス), 300
AsyncIterator (*typing* のクラス), 1906
AsyncMock (*unittest.mock* のクラス), 2001
asyncore (モジュール), 1302
AsyncResult (*multiprocessing.pool* のクラス), 1031
asyncSetUp() (*unittest.IsolatedAsyncioTestCase* のメソッド), 1971
asyncTearDown() (*unittest.IsolatedAsyncioTestCase* のメソッド), 1971
AT (*token* モジュール), 2341
at_eof() (*asyncio.StreamReader* のメソッド), 1121
atan() (*cmath* モジュール), 374
atan() (*math* モジュール), 370
atan2() (*math* モジュール), 370
atanh() (*cmath* モジュール), 375
atanh() (*math* モジュール), 371
ATEQUAL (*token* モジュール), 2341
atexit (モジュール), 2233
atexit (*weakref.finalize* の属性), 316
atof() (*locale* モジュール), 1771
atoi() (*locale* モジュール), 1771
attach() (*email.message.Message* のメソッド), 1378
attach_loop() (*asyncio.AbstractChildWatcher* のメソッド), 1193
attach_mock() (*unittest.mock.Mock* のメソッド), 1993
AttlistDeclHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1528
attrgetter() (*operator* モジュール), 465
attrib (*xml.etree.ElementTree.Element* の属性), 1477
attribute, **2479**
AttributeError, 115
attributes (*xml.dom.Node* の属性), 1490
AttributesImpl (*xml.sax.xmlreader* のクラス), 1519
AttributesNSImpl (*xml.sax.xmlreader* のクラス), 1519
attroff() (*curses.window* のメソッド), 904
attron() (*curses.window* のメソッド), 904
attrset() (*curses.window* のメソッド), 904
Audio Interchange File Format, 1731, 1741
AUDIO_FILE_ENCODING_ADPCM_G721 (*sunau* モジュール), 1735
AUDIO_FILE_ENCODING_ADPCM_G722 (*sunau* モジュール), 1735
AUDIO_FILE_ENCODING_ADPCM_G723_3 (*sunau* モジュール), 1735
AUDIO_FILE_ENCODING_ADPCM_G723_5 (*sunau* モジュール), 1735
AUDIO_FILE_ENCODING_ALAW_8 (*sunau* モジュール), 1735
AUDIO_FILE_ENCODING_DOUBLE (*sunau* モジュール), 1735
AUDIO_FILE_ENCODING_FLOAT (*sunau* モジュール), 1735
AUDIO_FILE_ENCODING_LINEAR_8 (*sunau* モジュール), 1735
AUDIO_FILE_ENCODING_LINEAR_16 (*sunau* モジュール), 1735
AUDIO_FILE_ENCODING_LINEAR_24 (*sunau* モジュール), 1735
AUDIO_FILE_ENCODING_LINEAR_32 (*sunau* モジュール), 1735
AUDIO_FILE_ENCODING_MULAW_8 (*sunau* モジュール), 1735
AUDIO_FILE_MAGIC (*sunau* モジュール), 1735
AUDIODEV, 1746
audioop (モジュール), 1727
audit events, 2087
audit() (*sys* モジュール), 2166
auditing, 2166
auth() (*ftplib.FTP_TLS* のメソッド), 1618
auth() (*smtplib.SMTP* のメソッド), 1644
authenticate() (*imaplib.IMAP4* のメソッド), 1625
AuthenticationError, 1005
authenticators() (*netrc.netrc* のメソッド), 679
authkey (*multiprocessing.Process* の属性), 1004
auto (*enum* のクラス), 339
autorange() (*timeit.Timer* のメソッド), 2122
avg() (*audioop* モジュール), 1728
avgpp() (*audioop* モジュール), 1728
avoids_symlink_attacks (*shutil.rmtree* の属性), 524
AWAIT (*token* モジュール), 2341
await_args (*unittest.mock.AsyncMock* の属性), 2004
await_args_list (*unittest.mock.AsyncMock* の属性), 2005
await_count (*unittest.mock.AsyncMock* の属性), 2004
awaitable, **2479**
Awaitable (*collections.abc* のクラス), 299
Awaitable (*typing* のクラス), 1906

B

-b
    compileall command line option, 2353
    unittest command line option, 1951
b2a_base64() (*binascii* モジュール), 1448
b2a_hex() (*binascii* モジュール), 1449
b2a_hqx() (*binascii* モジュール), 1448
b2a_qp() (*binascii* モジュール), 1448
b2a_uu() (*binascii* モジュール), 1447
b16decode() (*base64* モジュール), 1444
b16encode() (*base64* モジュール), 1444
b32decode() (*base64* モジュール), 1443
b32encode() (*base64* モジュール), 1443
b64decode() (*base64* モジュール), 1443

b64encode() (*base64* モジュール), 1443
b85decode() (*base64* モジュール), 1445
b85encode() (*base64* モジュール), 1445
Babyl (*mailbox* のクラス), 1425
BabylMessage (*mailbox* のクラス), 1433
back() (*turtle* モジュール), 1781
backslashreplace_errors() (*codecs* モジュール), 206
backup() (*sqlite3.Connection* のメソッド), 579
backward() (*turtle* モジュール), 1781
BadGzipFile, 600
BadStatusLine, 1606
BadZipFile, 616
BadZipfile, 616
Balloon (*tkinter.tix* のクラス), 1870
Barrier (*multiprocessing* のクラス), 1014
Barrier (*threading* のクラス), 992
Barrier() (*multiprocessing.managers.SyncManager* のメソッド), 1022
base64
    encoding, 1442
    モジュール, 1447
base64 (モジュール), 1442
base_exec_prefix (*sys* モジュール), 2166
base_prefix (*sys* モジュール), 2167
BaseCGIHandler (*wsgiref.handlers* のクラス), 1557
BaseCookie (*http.cookies* のクラス), 1679
BaseException, 114
BaseHandler (*urllib.request* のクラス), 1567
BaseHandler (*wsgiref.handlers* のクラス), 1558
BaseHeader (*email.headerregistry* のクラス), 1359
BaseHTTPRequestHandler (*http.server* のクラス), 1671
BaseManager (*multiprocessing.managers* のクラス), 1020
basename() (*os.path* モジュール), 492
BaseProtocol (*asyncio* のクラス), 1180
BaseProxy (*multiprocessing.managers* のクラス), 1027
BaseRequestHandler (*socketserver* のクラス), 1666
BaseRotatingHandler (*logging.handlers* のクラス), 879
BaseSelector (*selectors* のクラス), 1298
BaseServer (*socketserver* のクラス), 1663
basestring (*2to3 fixer*), 2061
BaseTransport (*asyncio* のクラス), 1175
basicConfig() (*logging* モジュール), 860
BasicContext (*decimal* のクラス), 392
BasicInterpolation (*configparser* のクラス), 662
BasicTestRunner (*test.support* のクラス), 2084
baudrate() (*curses* モジュール), 895
bbox() (*tkinter.ttk.Treeview* のメソッド), 1860
BDADDR_ANY (*socket* モジュール), 1218
BDADDR_LOCAL (*socket* モジュール), 1218
bdb
    モジュール, 2101
Bdb (*bdb* のクラス), 2093
bdb (モジュール), 2092
BdbQuit, 2092
BDFL, **2479**
beep() (*curses* モジュール), 895
Beep() (*winsound* モジュール), 2406
BEFORE_ASYNC_WITH (*opcode*), 2365
begin_fill() (*turtle* モジュール), 1792
BEGIN_FINALLY (*opcode*), 2367
begin_poly() (*turtle* モジュール), 1799
below() (*curses.panel.Panel* のメソッド), 922
BELOW_NORMAL_PRIORITY_CLASS (*subprocess* モジュール), 1077
Benchmarking, 2120
benchmarking, 794, 795, 798
--best
    gzip command line option, 603
betavariate() (*random* モジュール), 417
bgcolor() (*turtle* モジュール), 1801
bgpic() (*turtle* モジュール), 1801
bias() (*audioop* モジュール), 1728
bidirectional() (*unicodedata* モジュール), 181
bigaddrspacetest() (*test.support* モジュール), 2078
BigEndianStructure (*ctypes* のクラス), 972
bigmemtest() (*test.support* モジュール), 2078
bin() (組み込み関数), 6
binary
    data, packing, 193
    literals, 39
Binary (*msilib* のクラス), 2386
Binary (*xmlrpc.client* のクラス), 1699
binary file, **2479**
binary mode, 23
binary semaphores, 1096
BINARY_ADD (*opcode*), 2363
BINARY_AND (*opcode*), 2363
BINARY_FLOOR_DIVIDE (*opcode*), 2363
BINARY_LSHIFT (*opcode*), 2363
BINARY_MATRIX_MULTIPLY (*opcode*), 2363
BINARY_MODULO (*opcode*), 2363
BINARY_MULTIPLY (*opcode*), 2363
BINARY_OR (*opcode*), 2364
BINARY_POWER (*opcode*), 2363
BINARY_RSHIFT (*opcode*), 2363
BINARY_SUBSCR (*opcode*), 2363
BINARY_SUBTRACT (*opcode*), 2363
BINARY_TRUE_DIVIDE (*opcode*), 2363
BINARY_XOR (*opcode*), 2364
BinaryIO (*typing* のクラス), 1908
binascii (モジュール), 1447
bind (*widgets*), 1842
bind() (*asyncore.dispatcher* のメソッド), 1304
bind() (*inspect.Signature* のメソッド), 2256
bind() (*socket.socket* のメソッド), 1228
bind_partial() (*inspect.Signature* のメソッド), 2256
bind_port() (*test.support* モジュール), 2080
bind_textdomain_codeset() (*gettext* モジュール), 1754
bind_unix_socket() (*test.support* モジュール), 2080
bindtextdomain() (*gettext* モジュール), 1753
bindtextdomain() (*locale* モジュール), 1773
binhex
    モジュール, 1447
binhex (モジュール), 1446
binhex() (*binhex* モジュール), 1446
bisect (モジュール), 306
bisect() (*bisect* モジュール), 307
bisect_left() (*bisect* モジュール), 306
bisect_right() (*bisect* モジュール), 307
bit_length() (*int* のメソッド), 42
bitmap() (*msilib.Dialog* のメソッド), 2391
bitwise
    operations, 41
bk() (*turtle* モジュール), 1781
bkgd() (*curses.window* のメソッド), 904
bkgdset() (*curses.window* のメソッド), 904
blake2b() (*hashlib* モジュール), 692
blake2b, blake2s, 691
blake2b.MAX_DIGEST_SIZE (*hashlib* モジュール), 693
blake2b.MAX_KEY_SIZE (*hashlib* モジュール), 693
blake2b.PERSON_SIZE (*hashlib* モジュール), 693
blake2b.SALT_SIZE (*hashlib* モジュール), 693
blake2s() (*hashlib* モジュール), 692
blake2s.MAX_DIGEST_SIZE (*hashlib* モジュール), 693
blake2s.MAX_KEY_SIZE (*hashlib* モジュール), 693
blake2s.PERSON_SIZE (*hashlib* モジュール), 693
blake2s.SALT_SIZE (*hashlib* モジュール), 693
block_size (*hmac.HMAC* の属性), 701
blocked_domains() (*http.cookiejar.DefaultCookiePolicy* のメソッド), 1690
BlockingIOError, 120, 775
blocksize (*http.client.HTTPConnection* の属性), 1609
body() (*nntplib.NNTP* のメソッド), 1638
body_encode() (*email.charset.Charset* のメソッド), 1395
body_encoding (*email.charset.Charset* の属性), 1394
body_line_iterator() (*email.iterators* モジュール), 1401

BOM (*codecs* モジュール), 203
BOM_BE (*codecs* モジュール), 203
BOM_LE (*codecs* モジュール), 203
BOM_UTF8 (*codecs* モジュール), 203
BOM_UTF16 (*codecs* モジュール), 203
BOM_UTF16_BE (*codecs* モジュール), 203
BOM_UTF16_LE (*codecs* モジュール), 203
BOM_UTF32 (*codecs* モジュール), 203
BOM_UTF32_BE (*codecs* モジュール), 203
BOM_UTF32_LE (*codecs* モジュール), 203
bool (組み込みクラス), 6
Boolean
    operations, 37, 38
    type, 6
    オブジェクト, 39
    values, 107
BOOLEAN_STATES (*configparser.ConfigParser* の属性), 668
bootstrap() (*ensurepip* モジュール), 2145
border() (*curses.window* のメソッド), 904
bottom() (*curses.panel.Panel* のメソッド), 922
bottom_panel() (*curses.panel* モジュール), 922
BoundArguments (*inspect* のクラス), 2259
BoundaryError, 1357
BoundedSemaphore (*asyncio* のクラス), 1131
BoundedSemaphore (*multiprocessing* のクラス), 1014
BoundedSemaphore (*threading* のクラス), 989
BoundedSemaphore() (*multiprocessing.managers.SyncManager* のメソッド), 1022
box() (*curses.window* のメソッド), 905
bpformat() (*bdb.Breakpoint* のメソッド), 2093
bpprint() (*bdb.Breakpoint* のメソッド), 2093
break (*pdb command*), 2105
break_anywhere() (*bdb.Bdb* のメソッド), 2095
break_here() (*bdb.Bdb* のメソッド), 2095
break_long_words (*textwrap.TextWrapper* の属性), 180
break_on_hyphens (*textwrap.TextWrapper* の属性), 180
Breakpoint (*bdb* のクラス), 2092
breakpoint() (組み込み関数), 6
breakpointhook() (*sys* モジュール), 2167
breakpoints, 1880
broadcast_address (*ipaddress.IPv4Network* の属性), 1718
broadcast_address (*ipaddress.IPv6Network* の属性), 1722
broken (*threading.Barrier* の属性), 993
BrokenBarrierError, 993
BrokenExecutor, 1061
BrokenPipeError, 121
BrokenProcessPool, 1061
BrokenThreadPool, 1061
BROWSER, 1537, 1538
BsdDbShelf (*shelve* のクラス), 559
buf (*multiprocessing.shared_memory.SharedMemory* の属性), 1049
--buffer
    unittest command line option, 1951
buffer (*2to3 fixer*), 2061
buffer (*io.TextIOBase* の属性), 786
buffer (*unittest.TestResult* の属性), 1978
buffer protocol
    binary sequence types, 68
    str (*built-in class*), 56
buffer size, I/O, 23
buffer_info() (*array.array* のメソッド), 310
buffer_size (*xml.parsers.expat.xmlparser* の属性), 1526
buffer_text (*xml.parsers.expat.xmlparser* の属性), 1526
buffer_updated() (*asyncio.BufferedProtocol* のメソッド), 1183
buffer_used (*xml.parsers.expat.xmlparser* の属性), 1526
BufferedIOBase (*io* のクラス), 780
BufferedProtocol (*asyncio* のクラス), 1180
BufferedRandom (*io* のクラス), 785
BufferedReader (*io* のクラス), 783
BufferedRWPair (*io* のクラス), 785
BufferedWriter (*io* のクラス), 784
BufferError, 114
BufferingHandler (*logging.handlers* のクラス), 889
BufferTooShort, 1005
bufsize() (*ossaudiodev.oss_audio_device* のメソッド), 1749
BUILD_CONST_KEY_MAP (*opcode*), 2369
BUILD_LIST (*opcode*), 2369
BUILD_LIST_UNPACK (*opcode*), 2370
BUILD_MAP (*opcode*), 2369
BUILD_MAP_UNPACK (*opcode*), 2370
BUILD_MAP_UNPACK_WITH_CALL (*opcode*), 2370
build_opener() (*urllib.request* モジュール), 1565
BUILD_SET (*opcode*), 2369
BUILD_SET_UNPACK (*opcode*), 2370
BUILD_SLICE (*opcode*), 2373
BUILD_STRING (*opcode*), 2369
BUILD_TUPLE (*opcode*), 2369
BUILD_TUPLE_UNPACK (*opcode*), 2369
BUILD_TUPLE_UNPACK_WITH_CALL (*opcode*), 2369
built-in
    types, 37
builtin_module_names (*sys* モジュール), 2167
BuiltinFunctionType (*types* モジュール), 324
BuiltinImporter (*importlib.machinery* のクラス), 2304
BuiltinMethodType (*types* モジュール), 324
builtins (モジュール), 2194
ButtonBox (*tkinter.tix* のクラス), 1870
bye() (*turtle* モジュール), 1808
byref() (*ctypes* モジュール), 965
bytearray
    formatting, 83
    interpolation, 83
    methods, 71
    オブジェクト, 50, 68, 70
bytearray (組み込みクラス), 70
bytecode, **2479**
byte-code
    file, 2350, 2467
Bytecode (*dis* のクラス), 2358
BYTECODE_SUFFIXES (*importlib.machinery* モジュール), 2304
Bytecode.codeobj (*dis* モジュール), 2358
Bytecode.first_line (*dis* モジュール), 2358
byteorder (*sys* モジュール), 2167
bytes
    formatting, 83
    interpolation, 83
    methods, 71
    str (*built-in class*), 56
    オブジェクト, 68
bytes (*uuid.UUID* の属性), 1657
bytes (組み込みクラス), 68
bytes-like object, **2479**
bytes_le (*uuid.UUID* の属性), 1657
BytesFeedParser (*email.parser* のクラス), 1340
BytesGenerator (*email.generator* のクラス), 1344
BytesHeaderParser (*email.parser* のクラス), 1342
BytesIO (*io* のクラス), 783
BytesParser (*email.parser* のクラス), 1341
ByteString (*collections.abc* のクラス), 298
ByteString (*typing* のクラス), 1905
byteswap() (*array.array* のメソッド), 310
byteswap() (*audioop* モジュール), 1728
BytesWarning, 123
bz2 (モジュール), 604
BZ2Compressor (*bz2* のクラス), 606
BZ2Decompressor (*bz2* のクラス), 606
BZ2File (*bz2* のクラス), 604

C

c

language, 39, 40
structures, 193
-C
trace command line option, 2127
-c
trace command line option, 2127
unittest command line option, 1951
zipapp command line option, 2157
-c <tarfile> <source1> ... <sourceN>
tarfile command line option, 644
-c <zipfile> <source1> ... <sourceN>
zipfile command line option, 627
C14NWriterTarget (*xml.etree.ElementTree* のクラス), 1483
c_bool (*ctypes* のクラス), 972
C_BUILTIN (*imp* モジュール), 2472
c_byte (*ctypes* のクラス), 970
c_char (*ctypes* のクラス), 970
c_char_p (*ctypes* のクラス), 970
c_contiguous (*memoryview* の属性), 94
c_double (*ctypes* のクラス), 970
C_EXTENSION (*imp* モジュール), 2471
c_float (*ctypes* のクラス), 970
c_int (*ctypes* のクラス), 970
c_int8 (*ctypes* のクラス), 970
c_int16 (*ctypes* のクラス), 970
c_int32 (*ctypes* のクラス), 970
c_int64 (*ctypes* のクラス), 971
c_long (*ctypes* のクラス), 971
c_longdouble (*ctypes* のクラス), 970
c_longlong (*ctypes* のクラス), 971
c_short (*ctypes* のクラス), 971
c_size_t (*ctypes* のクラス), 971
c_ssize_t (*ctypes* のクラス), 971
c_ubyte (*ctypes* のクラス), 971
c_uint (*ctypes* のクラス), 971
c_uint8 (*ctypes* のクラス), 971
c_uint16 (*ctypes* のクラス), 971
c_uint32 (*ctypes* のクラス), 971
c_uint64 (*ctypes* のクラス), 971
c_ulong (*ctypes* のクラス), 971
c_ulonglong (*ctypes* のクラス), 971
c_ushort (*ctypes* のクラス), 972
c_void_p (*ctypes* のクラス), 972
c_wchar (*ctypes* のクラス), 972
c_wchar_p (*ctypes* のクラス), 972
CAB (*msilib* のクラス), 2389
cache_from_source() (*imp* モジュール), 2469
cache_from_source() (*importlib.util* モジュール), 2309
cached (*importlib.machinery.ModuleSpec* の属性), 2309
cached_property() (*functools* モジュール), 452
CacheFTPHandler (*urllib.request* のクラス), 1569
calcobjsize() (*test.support* モジュール), 2076
calcsize() (*struct* モジュール), 194
calcvobjsize() (*test.support* モジュール), 2076
Calendar (*calendar* のクラス), 269
calendar (モジュール), 269
calendar() (*calendar* モジュール), 274
call() (*subprocess* モジュール), 1078
call() (*unittest.mock* モジュール), 2025
call_args (*unittest.mock.Mock* の属性), 1996
call_args_list (*unittest.mock.Mock* の属性), 1997
call_at() (*asyncio.loop* のメソッド), 1146
call_count (*unittest.mock.Mock* の属性), 1994
call_exception_handler() (*asyncio.loop* のメソッド), 1161
CALL_FINALLY (*opcode*), 2371
CALL_FUNCTION (*opcode*), 2372
CALL_FUNCTION_EX (*opcode*), 2373
CALL_FUNCTION_KW (*opcode*), 2372
call_later() (*asyncio.loop* のメソッド), 1146
call_list() (*unittest.mock.call* のメソッド), 2025
CALL_METHOD (*opcode*), 2373
call_soon() (*asyncio.loop* のメソッド), 1145
call_soon_threadsafe() (*asyncio.loop* のメソッド), 1145
call_tracing() (*sys* モジュール), 2167
Callable (*collections.abc* のクラス), 298
Callable (*typing* モジュール), 1915
callable() (組み込み関数), 7
CallableProxyType (*weakref* モジュール), 317
callback, **2479**
callback (*optparse.Option* の属性), 2450
callback() (*contextlib.ExitStack* のメソッド), 2220
callback_args (*optparse.Option* の属性), 2450
callback_kwargs (*optparse.Option* の属性), 2450
callbacks (*gc* モジュール), 2247
called (*unittest.mock.Mock* の属性), 1994
CalledProcessError, 1065
CAN_BCM (*socket* モジュール), 1216
can_change_color() (*curses* モジュール), 895
can_fetch() (*urllib.robotparser.RobotFileParser* のメソッド), 1600
CAN_ISOTP (*socket* モジュール), 1217
CAN_RAW_FD_FRAMES (*socket* モジュール), 1217
can_symlink() (*test.support* モジュール), 2076
can_write_eof() (*asyncio.StreamWriter* のメソッド), 1122
can_write_eof() (*asyncio.WriteTransport* のメソッド), 1178
can_xattr() (*test.support* モジュール), 2076
cancel() (*asyncio.Future* のメソッド), 1172
cancel() (*asyncio.Handle* のメソッド), 1164
cancel() (*asyncio.Task* のメソッド), 1114
cancel() (*concurrent.futures.Future* のメソッド), 1059
cancel() (*sched.scheduler* のメソッド), 1086
cancel() (*threading.Timer* のメソッド), 991
cancel_dump_traceback_later() (*faulthandler* モジュール), 2100
cancel_join_thread() (*multiprocessing.Queue* のメソッド), 1008
cancelled() (*asyncio.Future* のメソッド), 1171
cancelled() (*asyncio.Handle* のメソッド), 1164
cancelled() (*asyncio.Task* のメソッド), 1115
cancelled() (*concurrent.futures.Future* のメソッド), 1059
CancelledError, 1061, 1141
CannotSendHeader, 1606
CannotSendRequest, 1606
canonic() (*bdb.Bdb* のメソッド), 2093
canonical() (*decimal.Context* のメソッド), 394
canonical() (*decimal.Decimal* のメソッド), 384
canonicalize() (*xml.etree.ElementTree* モジュール), 1470
capa() (*poplib.POP3* のメソッド), 1620
capitalize() (*bytearray* のメソッド), 78
capitalize() (*bytes* のメソッド), 78
capitalize() (*str* のメソッド), 56
captured_stderr() (*test.support* モジュール), 2074
captured_stdin() (*test.support* モジュール), 2074
captured_stdout() (*test.support* モジュール), 2074
captureWarnings() (*logging* モジュール), 863
capwords() (*string* モジュール), 139
casefold() (*str* のメソッド), 56
cast() (*ctypes* モジュール), 965
cast() (*memoryview* のメソッド), 90
cast() (*typing* モジュール), 1911
cat() (*nis* モジュール), 2430
--catch
unittest command line option, 1951
catch_threading_exception() (*test.support* モジュール), 2080
catch_unraisable_exception() (*test.support* モジュール), 2080
catch_warnings (*warnings* のクラス), 2202
category() (*unicodedata* モジュール), 181
cbreak() (*curses* モジュール), 895
ccc() (*ftplib.FTP_TLS* のメソッド), 1618
C-contiguous, 2480
cdf() (*statistics.NormalDist* のメソッド), 430
CDLL (*ctypes* のクラス), 958

ceil() (*in module math*), 40
ceil() (*math* モジュール), 365
CellType (*types* モジュール), 323
center() (*bytearray* のメソッド), 75
center() (*bytes* のメソッド), 75
center() (*str* のメソッド), 57
CERT_NONE (*ssl* モジュール), 1251
CERT_OPTIONAL (*ssl* モジュール), 1251
CERT_REQUIRED (*ssl* モジュール), 1251
cert_store_stats() (*ssl.SSLContext* のメソッド), 1265
cert_time_to_seconds() (*ssl* モジュール), 1248
CertificateError, 1246
certificates, 1275
CFUNCTYPE() (*ctypes* モジュール), 962
CGI
    debugging, 1547
    exceptions, 1549
    protocol, 1540
    security, 1546
    tracebacks, 1549
cgi (モジュール), 1540
cgi_directories (*http.server.CGIHTTPRequestHandler* の属性), 1678
CGIHandler (*wsgiref.handlers* のクラス), 1557
CGIHTTPRequestHandler (*http.server* のクラス), 1677
cgitb (モジュール), 1549
CGIXMLRPCRequestHandler (*xmlrpc.server* のクラス), 1704
chain() (*itertools* モジュール), 438
chaining
    comparisons, 38
ChainMap (*collections* のクラス), 275
ChainMap (*typing* のクラス), 1907
change_cwd() (*test.support* モジュール), 2074
CHANNEL_BINDING_TYPES (*ssl* モジュール), 1257
channel_class (*smtpd.SMTPServer* の属性), 1649
channels() (*ossaudiodev.oss_audio_device* のメソッド), 1748
CHAR_MAX (*locale* モジュール), 1771
character, 181
CharacterDataHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1528
characters() (*xml.sax.handler.ContentHandler* のメソッド), 1515
characters_written (*BlockingIOError* の属性), 120
Charset (*email.charset* のクラス), 1394
charset() (*gettext.NullTranslations* のメソッド), 1758
chdir() (*os* モジュール), 728
check (*lzma.LZMADecompressor* の属性), 613
check() (*imaplib.IMAP4* のメソッド), 1625
check() (*tabnanny* モジュール), 2348
check__all__() (*test.support* モジュール), 2082
check_call() (*subprocess* モジュール), 1078
check_free_after_iterating() (*test.support* モジュール), 2082
check_hostname (*ssl.SSLContext* の属性), 1272
check_impl_detail() (*test.support* モジュール), 2072
check_no_resource_warning() (*test.support* モジュール), 2073
check_output() (*doctest.OutputChecker* のメソッド), 1943
check_output() (*subprocess* モジュール), 1079
check_returncode() (*subprocess.CompletedProcess* のメソッド), 1064
check_syntax_error() (*test.support* モジュール), 2078
check_syntax_warning() (*test.support* モジュール), 2078
check_unused_args() (*string.Formatter* のメソッド), 127
check_warnings() (*test.support* モジュール), 2072
checkbox() (*msilib.Dialog* のメソッド), 2392
checkcache() (*linecache* モジュール), 519
CHECKED_HASH (*py_compile.PycInvalidationMode* の属性), 2352
checkfuncname() (*bdb* モジュール), 2098
CheckList (*tkinter.tix* のクラス), 1872
checksizeof() (*test.support* モジュール), 2076
checksum
    Cyclic Redundancy Check, 597
chflags() (*os* モジュール), 729
chgat() (*curses.window* のメソッド), 905
childNodes (*xml.dom.Node* の属性), 1490
ChildProcessError, 121
children (*pyclbr.Class* の属性), 2350
children (*pyclbr.Function* の属性), 2349
chmod() (*os* モジュール), 729
chmod() (*pathlib.Path* のメソッド), 483
choice() (*random* モジュール), 415
choice() (*secrets* モジュール), 702
choices (*optparse.Option* の属性), 2450
choices() (*random* モジュール), 415
chown() (*os* モジュール), 730
chown() (*shutil* モジュール), 525
chr() (組み込み関数), 7
chroot() (*os* モジュール), 731
Chunk (*chunk* のクラス), 1741
chunk (モジュール), 1741
cipher
    DES, 2414
cipher() (*ssl.SSLSocket* のメソッド), 1262
circle() (*turtle* モジュール), 1784
CIRCUMFLEX (*token* モジュール), 2340
CIRCUMFLEXEQUAL (*token* モジュール), 2340
Clamped (*decimal* のクラス), 400
class, **2479**
Class (*symtable* のクラス), 2336
Class browser, 1876
class variable, **2479**
classmethod() (組み込み関数), 8
ClassMethodDescriptorType (*types* モジュール), 324
ClassVar (*typing* モジュール), 1915
CLD_CONTINUED (*os* モジュール), 765
CLD_DUMPED (*os* モジュール), 765
CLD_EXITED (*os* モジュール), 765
CLD_TRAPPED (*os* モジュール), 765
clean() (*mailbox.Maildir* のメソッド), 1422
cleandoc() (*inspect* モジュール), 2254
CleanImport (*test.support* のクラス), 2084
clear (*pdb command*), 2105
Clear Breakpoint, 1880
clear() (*asyncio.Event* のメソッド), 1128
clear() (*collections.deque* のメソッド), 282
clear() (*curses.window* のメソッド), 905
clear() (*dict* のメソッド), 99
clear() (*email.message.EmailMessage* のメソッド), 1338
clear() (*frozenset* のメソッド), 97
clear() (*http.cookiejar.CookieJar* のメソッド), 1686
clear() (*mailbox.Mailbox* のメソッド), 1419
clear() (*sequence method*), 50
clear() (*threading.Event* のメソッド), 991
clear() (*turtle* モジュール), 1801
clear() (*xml.etree.ElementTree.Element* のメソッド), 1477
clear_all_breaks() (*bdb.Bdb* のメソッド), 2096
clear_all_file_breaks() (*bdb.Bdb* のメソッド), 2096
clear_bpbynumber() (*bdb.Bdb* のメソッド), 2096
clear_break() (*bdb.Bdb* のメソッド), 2096
clear_cache() (*filecmp* モジュール), 508
clear_content() (*email.message.EmailMessage* のメソッド), 1338
clear_flags() (*decimal.Context* のメソッド), 393
clear_frames() (*traceback* モジュール), 2237
clear_history() (*readline* モジュール), 187
clear_session_cookies() (*http.cookiejar.CookieJar* のメソッド), 1686
clear_traces() (*tracemalloc* モジュール), 2134
clear_traps() (*decimal.Context* のメソッド), 393
clearcache() (*linecache* モジュール), 519
ClearData() (*msilib.Record* のメソッド), 2389
clearok() (*curses.window* のメソッド), 905
clearscreen() (*turtle* モジュール), 1801

clearstamp() (*turtle* モジュール), 1785
clearstamps() (*turtle* モジュール), 1785
Client() (*multiprocessing.connection* モジュール), 1032
client_address (*http.server.BaseHTTPRequestHandler* の属性), 1671
CLOCK_BOOTTIME (*time* モジュール), 800
clock_getres() (*time* モジュール), 792
clock_gettime() (*time* モジュール), 792
clock_gettime_ns() (*time* モジュール), 792
CLOCK_HIGHRES (*time* モジュール), 800
CLOCK_MONOTONIC (*time* モジュール), 801
CLOCK_MONOTONIC_RAW (*time* モジュール), 801
CLOCK_PROCESS_CPUTIME_ID (*time* モジュール), 801
CLOCK_PROF (*time* モジュール), 801
CLOCK_REALTIME (*time* モジュール), 802
clock_settime() (*time* モジュール), 793
clock_settime_ns() (*time* モジュール), 793
CLOCK_THREAD_CPUTIME_ID (*time* モジュール), 801
CLOCK_UPTIME (*time* モジュール), 801
CLOCK_UPTIME_RAW (*time* モジュール), 801
clone() (*email.generator.BytesGenerator* のメソッド), 1345
clone() (*email.generator.Generator* のメソッド), 1347
clone() (*email.policy.Policy* のメソッド), 1351
clone() (*pipes.Template* のメソッド), 2423
clone() (*turtle* モジュール), 1799
cloneNode() (*xml.dom.Node* のメソッド), 1492
close() (*aifc.aifc* のメソッド), 1733
close() (*asyncio.AbstractChildWatcher* のメソッド), 1193
close() (*asyncio.BaseTransport* のメソッド), 1176
close() (*asyncio.loop* のメソッド), 1144
close() (*asyncio.Server* のメソッド), 1165
close() (*asyncio.StreamWriter* のメソッド), 1122
close() (*asyncio.SubprocessTransport* のメソッド), 1180
close() (*asyncore.dispatcher* のメソッド), 1305
close() (*chunk.Chunk* のメソッド), 1742
close() (*contextlib.ExitStack* のメソッド), 2220
close() (*dbm.dumb.dumbdbm* のメソッド), 568
close() (*dbm.gnu.gdbm* のメソッド), 566
close() (*dbm.ndbm.ndbm* のメソッド), 566
close() (*email.parser.BytesFeedParser* のメソッド), 1341
close() (*fileinput* モジュール), 499
close() (*ftplib.FTP* のメソッド), 1618
close() (*html.parser.HTMLParser* のメソッド), 1455
close() (*http.client.HTTPConnection* のメソッド), 1609
close() (*imaplib.IMAP4* のメソッド), 1626
close() (*io.IOBase* のメソッド), 778
close() (*logging.FileHandler* のメソッド), 878
close() (*logging.Handler* のメソッド), 850
close() (*logging.handlers.MemoryHandler* のメソッド), 890
close() (*logging.handlers.NTEventLogHandler* のメソッド), 888
close() (*logging.handlers.SocketHandler* のメソッド), 883
close() (*logging.handlers.SysLogHandler* のメソッド), 885
close() (*mailbox.Mailbox* のメソッド), 1420
close() (*mailbox.Maildir* のメソッド), 1422
close() (*mailbox.MH* のメソッド), 1425
close() (*mmap.mmap* のメソッド), 1322
Close() (*msilib.Database* のメソッド), 2387
Close() (*msilib.View* のメソッド), 2387
close() (*multiprocessing.connection.Connection* のメソッド), 1012
close() (*multiprocessing.connection.Listener* のメソッド), 1033
close() (*multiprocessing.pool.Pool* のメソッド), 1031
close() (*multiprocessing.Process* のメソッド), 1004
close() (*multiprocessing.Queue* のメソッド), 1008
close() (*multiprocessing.shared_memory.SharedMemory* のメソッド), 1049
close() (*os* モジュール), 714
close() (*ossaudiodev.oss_audio_device* のメソッド), 1747
close() (*ossaudiodev.oss_mixer_device* のメソッド), 1750
close() (*os.scandir* のメソッド), 739
close() (*select.devpoll* のメソッド), 1291
close() (*select.epoll* のメソッド), 1292
close() (*select.kqueue* のメソッド), 1295
close() (*selectors.BaseSelector* のメソッド), 1300
close() (*shelve.Shelf* のメソッド), 558
close() (*socket* モジュール), 1222
close() (*socket.socket* のメソッド), 1228
close() (*sqlite3.Connection* のメソッド), 573
close() (*sqlite3.Cursor* のメソッド), 583
close() (*sunau.AU_read* のメソッド), 1736
close() (*sunau.AU_write* のメソッド), 1737
close() (*tarfile.TarFile* のメソッド), 636
close() (*telnetlib.Telnet* のメソッド), 1654
close() (*urllib.request.BaseHandler* のメソッド), 1573
close() (*wave.Wave_read* のメソッド), 1739
close() (*wave.Wave_write* のメソッド), 1740
Close() (*winreg.PyHKEY* のメソッド), 2406
close() (*xml.etree.ElementTree.TreeBuilder* のメソッド), 1482
close() (*xml.etree.ElementTree.XMLParser* のメソッド), 1483
close() (*xml.etree.ElementTree.XMLPullParser* のメソッド), 1485
close() (*xml.sax.xmlreader.IncrementalParser* のメソッド), 1521
close() (*zipfile.ZipFile* のメソッド), 619
close_connection (*http.server.BaseHTTPRequestHandler* の属性), 1672
close_when_done() (*asynchat.async_chat* のメソッド), 1308
closed (*http.client.HTTPResponse* の属性), 1611
closed (*io.IOBase* の属性), 778
closed (*mmap.mmap* の属性), 1323
closed (*ossaudiodev.oss_audio_device* の属性), 1750
closed (*select.devpoll* の属性), 1291
closed (*select.epoll* の属性), 1293
closed (*select.kqueue* の属性), 1295
CloseKey() (*winreg* モジュール), 2395
closelog() (*syslog* モジュール), 2431
closerange() (*os* モジュール), 715
closing() (*contextlib* モジュール), 2215
clrtobot() (*curses.window* のメソッド), 905
clrtoeol() (*curses.window* のメソッド), 905
cmath (モジュール), 373
cmd
    モジュール, 2101
Cmd (*cmd* のクラス), 1816
cmd (*subprocess.CalledProcessError* の属性), 1065
cmd (*subprocess.TimeoutExpired* の属性), 1064
cmd (モジュール), 1816
cmdloop() (*cmd.Cmd* のメソッド), 1817
cmdqueue (*cmd.Cmd* の属性), 1818
cmp() (*filecmp* モジュール), 507
cmp_op (*dis* モジュール), 2374
cmp_to_key() (*functools* モジュール), 452
cmpfiles() (*filecmp* モジュール), 507
CMSG_LEN() (*socket* モジュール), 1226
CMSG_SPACE() (*socket* モジュール), 1226
CO_ASYNC_GENERATOR (*inspect* モジュール), 2268
CO_COROUTINE (*inspect* モジュール), 2267
CO_GENERATOR (*inspect* モジュール), 2267
CO_ITERABLE_COROUTINE (*inspect* モジュール), 2267
CO_NESTED (*inspect* モジュール), 2267
CO_NEWLOCALS (*inspect* モジュール), 2267
CO_NOFREE (*inspect* モジュール), 2267
CO_OPTIMIZED (*inspect* モジュール), 2267
CO_VARARGS (*inspect* モジュール), 2267
CO_VARKEYWORDS (*inspect* モジュール), 2267
code (*SystemExit* の属性), 119
code (*urllib.error.HTTPError* の属性), 1599
code (モジュール), 2273
code (*xml.etree.ElementTree.ParseError* の属性), 1486

code (*xml.parsers.expat.ExpatError* の属性), 1530
code object, 105, 561
code_info() (*dis* モジュール), 2359
CodecInfo (*codecs* のクラス), 201
Codecs, 200
    decode, 200
    encode, 200
codecs (モジュール), 200
coded_value (*http.cookies.Morsel* の属性), 1681
codeop (モジュール), 2276
codepoint2name (*html.entities* モジュール), 1460
codes (*xml.parsers.expat.errors* モジュール), 1532
CODESET (*locale* モジュール), 1767
CodeType (*types* のクラス), 323
coercion, **2480**
col_offset (*ast.AST* の属性), 2327
collapse_addresses() (*ipaddress* モジュール), 1726
collapse_rfc2231_value() (*email.utils* モジュール), 1400
collect() (*gc* モジュール), 2245
collect_incoming_data() (*asynchat.async_chat* のメソッド), 1308
Collection (*collections.abc* のクラス), 298
Collection (*typing* のクラス), 1904
collections (モジュール), 275
collections.abc (モジュール), 295
colno (*json.JSONDecodeError* の属性), 1411
colno (*re.error* の属性), 152
COLON (*token* モジュール), 2338
COLONEQUAL (*token* モジュール), 2341
color() (*turtle* モジュール), 1792
color_content() (*curses* モジュール), 895
color_pair() (*curses* モジュール), 895
colormode() (*turtle* モジュール), 1807
colorsys (モジュール), 1743
COLS, 902
column() (*tkinter.ttk.Treeview* のメソッド), 1860
COLUMNS, 902
columns (*os.terminal_size* の属性), 726
comb() (*math* モジュール), 365
combinations() (*itertools* モジュール), 438
combinations_with_replacement() (*itertools* モジュール), 439
combine() (*datetime.datetime* のクラスメソッド), 240
combining() (*unicodedata* モジュール), 181
ComboBox (*tkinter.tix* のクラス), 1870
Combobox (*tkinter.ttk* のクラス), 1851
COMMA (*token* モジュール), 2339
command (*http.server.BaseHTTPRequestHandler* の属性), 1672
CommandCompiler (*codeop* のクラス), 2277
commands (*pdb command*), 2106
comment (*http.cookiejar.Cookie* の属性), 1692
COMMENT (*token* モジュール), 2341
comment (*zipfile.ZipFile* の属性), 622
comment (*zipfile.ZipInfo* の属性), 626
Comment() (*xml.etree.ElementTree* モジュール), 1471
comment() (*xml.etree.ElementTree.TreeBuilder* のメソッド), 1482
comment_url (*http.cookiejar.Cookie* の属性), 1693
commenters (*shlex.shlex* の属性), 1825
CommentHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1529
commit() (*msilib.CAB* のメソッド), 2389
Commit() (*msilib.Database* のメソッド), 2387
commit() (*sqlite3.Connection* のメソッド), 573
common (*filecmp.dircmp* の属性), 509
Common Gateway Interface, 1540
common_dirs (*filecmp.dircmp* の属性), 509
common_files (*filecmp.dircmp* の属性), 509
common_funny (*filecmp.dircmp* の属性), 509
common_types (*mimetypes* モジュール), 1440
commonpath() (*os.path* モジュール), 492
commonprefix() (*os.path* モジュール), 492
communicate() (*asyncio.asyncio.subprocess.Process* のメソッド), 1135
communicate() (*subprocess.Popen* のメソッド), 1073
compare() (*decimal.Context* のメソッド), 395
compare() (*decimal.Decimal* のメソッド), 385
compare() (*difflib.Differ* のメソッド), 173
compare_digest() (*hmac* モジュール), 701
compare_digest() (*secrets* モジュール), 703
compare_networks() (*ipaddress.IPv4Network* のメソッド), 1720
compare_networks() (*ipaddress.IPv6Network* のメソッド), 1722
COMPARE_OP (*opcode*), 2370
compare_signal() (*decimal.Context* のメソッド), 395
compare_signal() (*decimal.Decimal* のメソッド), 385
compare_to() (*tracemalloc.Snapshot* のメソッド), 2137
compare_total() (*decimal.Context* のメソッド), 395
compare_total() (*decimal.Decimal* のメソッド), 385
compare_total_mag() (*decimal.Context* のメソッド), 395
compare_total_mag() (*decimal.Decimal* のメソッド), 385
comparing
    objects, 38
comparison
    operator, 38
COMPARISON_FLAGS (*doctest* モジュール), 1929
comparisons
    chaining, 38
Compat32 (*email.policy* のクラス), 1356
compat32 (*email.policy* モジュール), 1356
compile
    組み込み関数, 106, 323, 2324
Compile (*codeop* のクラス), 2276
compile() (*parser.ST* のメソッド), 2325
compile() (*py_compile* モジュール), 2351
compile() (*re* モジュール), 146
compile() (組み込み関数), 8
compile_command() (*code* モジュール), 2274
compile_command() (*codeop* モジュール), 2276
compile_dir() (*compileall* モジュール), 2354
compile_file() (*compileall* モジュール), 2355
compile_path() (*compileall* モジュール), 2356
compileall (モジュール), 2353
compileall command line option
    -b, 2353
    -d destdir, 2353
    directory ..., 2353
    -f, 2353
    file ..., 2353
    -i list, 2353
    --invalidation-mode
        [timestamp|checked-hash|unchecked-hash], 2354
    -j N, 2354
    -l, 2353
    -q, 2353
    -r, 2354
    -x regex, 2353
compilest() (*parser* モジュール), 2324
complete() (*rlcompleter.Completer* のメソッド), 191
complete_statement() (*sqlite3* モジュール), 572
completedefault() (*cmd.Cmd* のメソッド), 1818
CompletedProcess (*subprocess* のクラス), 1063
complex
    組み込み関数, 39
Complex (*numbers* のクラス), 361
complex (組み込みクラス), 9
complex number, **2480**
    literals, 39
    オブジェクト, 39
--compress
    zipapp command line option, 2157
compress() (*bz2* モジュール), 607
compress() (*bz2.BZ2Compressor* のメソッド), 606
compress() (*gzip* モジュール), 602

compress() (itertools モジュール), 440
compress() (lzma モジュール), 613
compress() (lzma.LZMACompressor のメソッド), 611
compress() (zlib モジュール), 595
compress() (zlib.Compress のメソッド), 598
compress_size (zipfile.ZipInfo の属性), 626
compress_type (zipfile.ZipInfo の属性), 626
compressed (ipaddress.IPv4Address の属性), 1713
compressed (ipaddress.IPv4Network の属性), 1718
compressed (ipaddress.IPv6Address の属性), 1714
compressed (ipaddress.IPv6Network の属性), 1722
compression() (ssl.SSLSocket のメソッド), 1262
CompressionError, 630
compressobj() (zlib モジュール), 596
COMSPEC, 764, 1068
concat() (operator モジュール), 464
concatenation
    operation, 47
concurrent.futures (モジュール), 1054
Condition (asyncio のクラス), 1129
Condition (multiprocessing のクラス), 1014
condition (pdb command), 2106
Condition (threading のクラス), 987
condition() (msilib.Control のメソッド), 2391
Condition() (multiprocessing.managers.SyncManager のメソッド), 1022
ConfigParser (configparser のクラス), 672
configparser (モジュール), 657
configuration
    file, 657
    file, debugger, 2105
    file, path, 2269
configuration information, 2189
configure() (tkinter.ttk.Style のメソッド), 1864
configure_mock() (unittest.mock.Mock のメソッド), 1993
confstr() (os モジュール), 769
confstr_names (os モジュール), 770
conjugate() (complex number method), 40
conjugate() (decimal.Decimal のメソッド), 385
conjugate() (numbers.Complex のメソッド), 361
conn (smtpd.SMTPChannel の属性), 1651
connect() (asyncore.dispatcher のメソッド), 1304
connect() (ftplib.FTP のメソッド), 1615
connect() (http.client.HTTPConnection のメソッド), 1609
connect() (multiprocessing.managers.BaseManager のメソッド), 1021
connect() (smtplib.SMTP のメソッド), 1642
connect() (socket.socket のメソッド), 1228
connect() (sqlite3 モジュール), 570
connect_accepted_socket() (asyncio.loop のメソッド), 1153
connect_ex() (socket.socket のメソッド), 1229
connect_read_pipe() (asyncio.loop のメソッド), 1158
connect_write_pipe() (asyncio.loop のメソッド), 1158
Connection (multiprocessing.connection のクラス), 1011
Connection (sqlite3 のクラス), 573
connection (sqlite3.Cursor の属性), 584
connection_lost() (asyncio.BaseProtocol のメソッド), 1181
connection_made() (asyncio.BaseProtocol のメソッド), 1181
ConnectionAbortedError, 121
ConnectionError, 121
ConnectionRefusedError, 121
ConnectionResetError, 121
ConnectRegistry() (winreg モジュール), 2395
const (optparse.Option の属性), 2450
constructor() (copyreg モジュール), 556
consumed (asyncio.LimitOverrunError の属性), 1142
container
    iteration over, 46
Container (collections.abc のクラス), 298
Container (typing のクラス), 1904
contains() (operator モジュール), 464
content type
    MIME, 1438
content_disposition (email.headerregistry.ContentDispositionHeader の属性), 1362
content_manager (email.policy.EmailPolicy の属性), 1354
content_type (email.headerregistry.ContentTypeHeader の属性), 1362
ContentDispositionHeader (email.headerregistry のクラス), 1362
ContentHandler (xml.sax.handler のクラス), 1510
ContentManager (email.contentmanager のクラス), 1365
contents (ctypes.__Pointer の属性), 975
contents() (importlib.abc.ResourceReader のメソッド), 2297
contents() (importlib.resources モジュール), 2303
ContentTooShortError, 1599
ContentTransferEncoding (email.headerregistry のクラス), 1362
ContentTypeHeader (email.headerregistry のクラス), 1362
Context (contextvars のクラス), 1093
Context (decimal のクラス), 392
context (ssl.SSLSocket の属性), 1264
context management protocol, 103
context manager, 103, **2480**
context variable, **2480**
context_diff() (difflib モジュール), 165
ContextDecorator (contextlib のクラス), 2218
contextlib (モジュール), 2213
ContextManager (typing のクラス), 1906
contextmanager() (contextlib モジュール), 2213
ContextVar (contextvars のクラス), 1092
contextvars (モジュール), 1092
contiguous, **2480**
contiguous (memoryview の属性), 94
continue (pdb command), 2107
Control (msilib のクラス), 2391
Control (tkinter.tix のクラス), 1870
control() (msilib.Dialog のメソッド), 2391
control() (select.kqueue のメソッド), 1295
controlnames (curses.ascii モジュール), 921
controls() (ossaudiodev.oss_mixer_device のメソッド), 1750
ConversionError, 683
conversions
    numeric, 40
convert_arg_line_to_args() (argparse.ArgumentParser のメソッド), 838
convert_field() (string.Formatter のメソッド), 127
Cookie (http.cookiejar のクラス), 1684
CookieError, 1679
CookieJar (http.cookiejar のクラス), 1683
cookiejar (urllib.request.HTTPCookieProcessor の属性), 1576
CookiePolicy (http.cookiejar のクラス), 1684
Coordinated Universal Time, 791
Copy, 1880
copy
    protocol, 545
    モジュール, 556
copy (モジュール), 328
copy() (collections.deque のメソッド), 282
copy() (contextvars.Context のメソッド), 1094
copy() (copy モジュール), 328
copy() (decimal.Context のメソッド), 393
copy() (dict のメソッド), 99
copy() (frozenset のメソッド), 96
copy() (hashlib.hash のメソッド), 690
copy() (hmac.HMAC のメソッド), 701
copy() (http.cookies.Morsel のメソッド), 1681
copy() (imaplib.IMAP4 のメソッド), 1626
copy() (multiprocessing.sharedctypes モジュール), 1019

copy() (*pipes.Template* のメソッド), 2424
copy() (*sequence method*), 50
copy() (*shutil* モジュール), 521
copy() (*types.MappingProxyType* のメソッド), 326
copy() (*zlib.Compress* のメソッド), 598
copy() (*zlib.Decompress* のメソッド), 599
copy2() (*shutil* モジュール), 522
copy_abs() (*decimal.Context* のメソッド), 395
copy_abs() (*decimal.Decimal* のメソッド), 385
copy_context() (*contextvars* モジュール), 1093
copy_decimal() (*decimal.Context* のメソッド), 393
copy_file_range() (*os* モジュール), 715
copy_location() (*ast* モジュール), 2332
copy_negate() (*decimal.Context* のメソッド), 395
copy_negate() (*decimal.Decimal* のメソッド), 385
copy_sign() (*decimal.Context* のメソッド), 395
copy_sign() (*decimal.Decimal* のメソッド), 386
copyfile() (*shutil* モジュール), 520
copyfileobj() (*shutil* モジュール), 520
copying files, 519
copymode() (*shutil* モジュール), 520
copyreg (モジュール), 556
copyright (*sys* モジュール), 2167
copyright (組み込み変数), 36
copysign() (*math* モジュール), 365
copystat() (*shutil* モジュール), 521
copytree() (*shutil* モジュール), 523
coroutine, **2480**
Coroutine (*collections.abc* のクラス), 299
Coroutine (*typing* のクラス), 1906
coroutine function, **2480**
coroutine() (*asyncio* モジュール), 1118
coroutine() (*types* モジュール), 328
CoroutineType (*types* モジュール), 323
cos() (*cmath* モジュール), 374
cos() (*math* モジュール), 370
cosh() (*cmath* モジュール), 375
cosh() (*math* モジュール), 371
--count
    trace command line option, 2127
count (*tracemalloc.Statistic* の属性), 2139
count (*tracemalloc.StatisticDiff* の属性), 2139
count() (*array.array* のメソッド), 310
count() (*bytearray* のメソッド), 71
count() (*bytes* のメソッド), 71
count() (*collections.deque* のメソッド), 282
count() (*itertools* モジュール), 440
count() (*multiprocessing.shared_memory.ShareableList* のメソッド), 1052
count() (*sequence method*), 47
count() (*str* のメソッド), 57
count_diff (*tracemalloc.StatisticDiff* の属性), 2139
Counter (*collections* のクラス), 278
Counter (*typing* のクラス), 1907
countOf() (*operator* モジュール), 464
countTestCases() (*unittest.TestCase* のメソッド), 1970
countTestCases() (*unittest.TestSuite* のメソッド), 1974
CoverageResults (*trace* のクラス), 2128
--coverdir=<dir>
    trace command line option, 2127
cProfile (モジュール), 2113
CPU time, 795, 798
cpu_count() (*multiprocessing* モジュール), 1009
cpu_count() (*os* モジュール), 770
CPython, **2480**
cpython_only() (*test.support* モジュール), 2077
crawl_delay() (*urllib.robotparser.RobotFileParser* のメソッド), 1600
CRC (*zipfile.ZipInfo* の属性), 626
crc32() (*binascii* モジュール), 1449
crc32() (*zlib* モジュール), 596
crc_hqx() (*binascii* モジュール), 1448
--create <tarfile> <source1> ... <sourceN>
    tarfile command line option, 644
--create <zipfile> <source1> ... <sourceN>
    zipfile command line option, 627
create() (*imaplib.IMAP4* のメソッド), 1626
create() (*venv* モジュール), 2152
create() (*venv.EnvBuilder* のメソッド), 2150
create_aggregate() (*sqlite3.Connection* のメソッド), 574
create_archive() (*zipapp* モジュール), 2158
create_autospec() (*unittest.mock* モジュール), 2027
CREATE_BREAKAWAY_FROM_JOB (*subprocess* モジュール), 1078
create_collation() (*sqlite3.Connection* のメソッド), 575
create_configuration() (*venv.EnvBuilder* のメソッド), 2151
create_connection() (*asyncio.loop* のメソッド), 1148
create_connection() (*socket* モジュール), 1220
create_datagram_endpoint() (*asyncio.loop* のメソッド), 1150
create_decimal() (*decimal.Context* のメソッド), 393
create_decimal_from_float() (*decimal.Context* のメソッド), 394
create_default_context() (*ssl* モジュール), 1244
CREATE_DEFAULT_ERROR_MODE (*subprocess* モジュール), 1077
create_empty_file() (*test.support* モジュール), 2071
create_function() (*sqlite3.Connection* のメソッド), 574
create_future() (*asyncio.loop* のメソッド), 1147
create_module() (*importlib.abc.Loader* のメソッド), 2295
create_module() (*importlib.machinery.ExtensionFileLoader* のメソッド), 2308
CREATE_NEW_CONSOLE (*subprocess* モジュール), 1076
CREATE_NEW_PROCESS_GROUP (*subprocess* モジュール), 1076
CREATE_NO_WINDOW (*subprocess* モジュール), 1077
create_server() (*asyncio.loop* のメソッド), 1152
create_server() (*socket* モジュール), 1220
create_socket() (*asyncore.dispatcher* のメソッド), 1304
create_stats() (*profile.Profile* のメソッド), 2114
create_string_buffer() (*ctypes* モジュール), 965
create_subprocess_exec() (*asyncio* モジュール), 1133
create_subprocess_shell() (*asyncio* モジュール), 1133
create_system (*zipfile.ZipInfo* の属性), 626
create_task() (*asyncio* モジュール), 1107
create_task() (*asyncio.loop* のメソッド), 1147
create_unicode_buffer() (*ctypes* モジュール), 965
create_unix_connection() (*asyncio.loop* のメソッド), 1151
create_unix_server() (*asyncio.loop* のメソッド), 1153
create_version (*zipfile.ZipInfo* の属性), 626
createAttribute() (*xml.dom.Document* のメソッド), 1494
createAttributeNS() (*xml.dom.Document* のメソッド), 1494
createComment() (*xml.dom.Document* のメソッド), 1493
createDocument() (*xml.dom.DOMImplementation* のメソッド), 1489
createDocumentType() (*xml.dom.DOMImplementation* のメソッド), 1489
createElement() (*xml.dom.Document* のメソッド), 1493
createElementNS() (*xml.dom.Document* のメソッド), 1493
createfilehandler() (*tkinter.Widget.tk* のメソッド), 1845
CreateKey() (*winreg* モジュール), 2395
CreateKeyEx() (*winreg* モジュール), 2396
createLock() (*logging.Handler* のメソッド), 849
createLock() (*logging.NullHandler* のメソッド), 878
createProcessingInstruction() (*xml.dom.Document* のメソッド), 1494
CreateRecord() (*msilib* モジュール), 2386
createSocket() (*logging.handlers.SocketHandler* のメソッド), 884
createTextNode() (*xml.dom.Document* のメソッド), 1493
credits (組み込み変数), 36
critical() (*logging* モジュール), 858
critical() (*logging.Logger* のメソッド), 847
CRNCYSTR (*locale* モジュール), 1768
cross() (*audioop* モジュール), 1728

crypt
    モジュール, 2411
crypt (モジュール), 2414
crypt() (crypt モジュール), 2415
crypt(3), 2414, 2415
cryptography, 687
cssclass_month (calendar.HTMLCalendar の属性), 272
cssclass_month_head (calendar.HTMLCalendar の属性), 272
cssclass_noday (calendar.HTMLCalendar の属性), 272
cssclass_year (calendar.HTMLCalendar の属性), 272
cssclass_year_head (calendar.HTMLCalendar の属性), 272
cssclasses (calendar.HTMLCalendar の属性), 271
cssclasses_weekday_head (calendar.HTMLCalendar の属性), 272
csv, 649
csv (モジュール), 649
cte (email.headerregistry.ContentTransferEncoding の属性), 1362
cte_type (email.policy.Policy の属性), 1350
ctermid() (os モジュール), 706
ctime() (datetime.date のメソッド), 236
ctime() (datetime.datetime のメソッド), 248
ctime() (time モジュール), 793
ctrl() (curses.ascii モジュール), 921
CTRL_BREAK_EVENT (signal モジュール), 1313
CTRL_C_EVENT (signal モジュール), 1313
ctypes (モジュール), 935
curdir (os モジュール), 770
currency() (locale モジュール), 1770
current() (tkinter.ttk.Combobox のメソッド), 1851
current_process() (multiprocessing モジュール), 1009
current_task() (asyncio モジュール), 1113
current_task() (asyncio.Task のクラスメソッド), 1117
current_thread() (threading モジュール), 977
CurrentByteIndex (xml.parsers.expat.xmlparser の属性), 1527
CurrentColumnNumber (xml.parsers.expat.xmlparser の属性), 1527
currentframe() (inspect モジュール), 2264
CurrentLineNumber (xml.parsers.expat.xmlparser の属性), 1527
curs_set() (curses モジュール), 896
curses (モジュール), 894
curses.ascii (モジュール), 919
curses.panel (モジュール), 922
curses.textpad (モジュール), 917
Cursor (sqlite3 のクラス), 580
cursor() (sqlite3.Connection のメソッド), 573
cursyncup() (curses.window のメソッド), 905
Cut, 1880
cwd() (ftplib.FTP のメソッド), 1618
cwd() (pathlib.Path のクラスメソッド), 483
cycle() (itertools モジュール), 441
Cyclic Redundancy Check, 597

D

-d
    gzip command line option, 603
-d destdir
    compileall command line option, 2353
D_FMT (locale モジュール), 1767
D_T_FMT (locale モジュール), 1767
daemon (multiprocessing.Process の属性), 1003
daemon (threading.Thread の属性), 983
data
    packing binary, 193
    tabular, 649
data (collections.UserDict の属性), 294
data (collections.UserList の属性), 294
data (collections.UserString の属性), 295
Data (plistlib のクラス), 685
data (select.kevent の属性), 1297
data (selectors.SelectorKey の属性), 1298
data (urllib.request.Request の属性), 1570
data (xml.dom.Comment の属性), 1496
data (xml.dom.ProcessingInstruction の属性), 1497
data (xml.dom.Text の属性), 1496
data (xmlrpc.client.Binary の属性), 1699
data() (xml.etree.ElementTree.TreeBuilder のメソッド), 1482
data_filter() (tarfile モジュール), 641
data_open() (urllib.request.DataHandler のメソッド), 1579
data_received() (asyncio.Protocol のメソッド), 1182
database
    Unicode, 181
DatabaseError, 585
databases, 567
dataclass() (dataclasses モジュール), 2203
dataclasses (モジュール), 2203
datagram_received() (asyncio.DatagramProtocol のメソッド), 1183
DatagramHandler (logging.handlers のクラス), 884
DatagramProtocol (asyncio のクラス), 1180
DatagramRequestHandler (socketserver のクラス), 1666
DatagramTransport (asyncio のクラス), 1175
DataHandler (urllib.request のクラス), 1569
date (datetime のクラス), 232
date() (datetime.datetime のメソッド), 244
date() (nntplib.NNTP のメソッド), 1638
date_time (zipfile.ZipInfo の属性), 625
date_time_string() (http.server.BaseHTTPRequestHandler のメソッド), 1675
DateHeader (email.headerregistry のクラス), 1360
datetime (datetime のクラス), 238
datetime (email.headerregistry.DateHeader の属性), 1360
datetime (モジュール), 225
DateTime (xmlrpc.client のクラス), 1698
day (datetime.date の属性), 234
day (datetime.datetime の属性), 242
day_abbr (calendar モジュール), 274
day_name (calendar モジュール), 274
daylight (time モジュール), 802
Daylight Saving Time, 791
DbfilenameShelf (shelve のクラス), 559
dbm (モジュール), 562
dbm.dumb (モジュール), 567
dbm.gnu
    モジュール, 559
dbm.gnu (モジュール), 564
dbm.ndbm
    モジュール, 559
dbm.ndbm (モジュール), 566
dcgettext() (locale モジュール), 1773
debug (imaplib.IMAP4 の属性), 1630
debug (pdb command), 2109
DEBUG (re モジュール), 147
debug (shlex.shlex の属性), 1826
debug (zipfile.ZipFile の属性), 622
debug() (doctest モジュール), 1945
debug() (logging モジュール), 857
debug() (logging.Logger のメソッド), 845
debug() (pipes.Template のメソッド), 2423
debug() (unittest.TestCase のメソッド), 1961
debug() (unittest.TestSuite のメソッド), 1974
DEBUG_BYTECODE_SUFFIXES (importlib.machinery モジュール), 2303
DEBUG_COLLECTABLE (gc モジュール), 2248
DEBUG_LEAK (gc モジュール), 2248
DEBUG_SAVEALL (gc モジュール), 2248
debug_src() (doctest モジュール), 1946
DEBUG_STATS (gc モジュール), 2248
DEBUG_UNCOLLECTABLE (gc モジュール), 2248

debugger, 1879, 2175, 2184
    configuration file, 2105
debugging, 2101
    CGI, 1547
DebuggingServer (*smtpd* のクラス), 1650
debuglevel (*http.client.HTTPResponse* の属性), 1610
DebugRunner (*doctest* のクラス), 1946
Decimal (*decimal* のクラス), 382
decimal (モジュール), 377
decimal() (*unicodedata* モジュール), 181
DecimalException (*decimal* のクラス), 400
decode
    Codecs, 200
decode (*codecs.CodecInfo* の属性), 201
decode() (*base64* モジュール), 1445
decode() (*bytearray* のメソッド), 72
decode() (*bytes* のメソッド), 72
decode() (*codecs* モジュール), 200
decode() (*codecs.Codec* のメソッド), 207
decode() (*codecs.IncrementalDecoder* のメソッド), 209
decode() (*json.JSONDecoder* のメソッド), 1409
decode() (*quopri* モジュール), 1450
decode() (*uu* モジュール), 1451
decode() (*xmlrpc.client.Binary* のメソッド), 1699
decode() (*xmlrpc.client.DateTime* のメソッド), 1698
decode_header() (*email.header* モジュール), 1393
decode_header() (*nntplib* モジュール), 1639
decode_params() (*email.utils* モジュール), 1401
decode_rfc2231() (*email.utils* モジュール), 1400
decode_source() (*importlib.util* モジュール), 2310
decodebytes() (*base64* モジュール), 1445
DecodedGenerator (*email.generator* のクラス), 1347
decodestring() (*base64* モジュール), 1445
decodestring() (*quopri* モジュール), 1451
decomposition() (*unicodedata* モジュール), 182
--decompress
    gzip command line option, 603
decompress() (*bz2* モジュール), 607
decompress() (*bz2.BZ2Decompressor* のメソッド), 606
decompress() (*gzip* モジュール), 602
decompress() (*lzma* モジュール), 613
decompress() (*lzma.LZMADecompressor* のメソッド), 612
decompress() (*zlib* モジュール), 597
decompress() (*zlib.Decompress* のメソッド), 599
decompressobj() (*zlib* モジュール), 597
decorator, **2480**
DEDENT (*token* モジュール), 2338
dedent() (*textwrap* モジュール), 177
deepcopy() (*copy* モジュール), 328
def_prog_mode() (*curses* モジュール), 896
def_shell_mode() (*curses* モジュール), 896
default (*email.policy* モジュール), 1355
default (*inspect.Parameter* の属性), 2257
default (*optparse.Option* の属性), 2450
DEFAULT (*unittest.mock* モジュール), 2025
default() (*cmd.Cmd* のメソッド), 1818
default() (*json.JSONEncoder* のメソッド), 1410
DEFAULT_BUFFER_SIZE (*io* モジュール), 775
default_bufsize (*xml.dom.pulldom* モジュール), 1507
default_exception_handler() (*asyncio.loop* のメソッド), 1160
default_factory (*collections.defaultdict* の属性), 286
DEFAULT_FORMAT (*tarfile* モジュール), 631
DEFAULT_IGNORES (*filecmp* モジュール), 509
default_open() (*urllib.request.BaseHandler* のメソッド), 1573
DEFAULT_PROTOCOL (*pickle* モジュール), 538
default_timer() (*timeit* モジュール), 2121
DefaultContext (*decimal* のクラス), 392
DefaultCookiePolicy (*http.cookiejar* のクラス), 1684
defaultdict (*collections* のクラス), 286
DefaultDict (*typing* のクラス), 1906
DefaultEventLoopPolicy (*asyncio* のクラス), 1191
DefaultHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1529
DefaultHandlerExpand() (*xml.parsers.expat.xmlparser* のメソッド), 1529
defaults() (*configparser.ConfigParser* のメソッド), 673
DefaultSelector (*selectors* のクラス), 1300
defaultTestLoader (*unittest* モジュール), 1981
defaultTestResult() (*unittest.TestCase* のメソッド), 1970
defects (*email.headerregistry.BaseHeader* の属性), 1359
defects (*email.message.EmailMessage* の属性), 1339
defects (*email.message.Message* の属性), 1387
defpath (*os* モジュール), 771
DefragResult (*urllib.parse* のクラス), 1595
DefragResultBytes (*urllib.parse* のクラス), 1595
degrees() (*math* モジュール), 371
degrees() (*turtle* モジュール), 1788
del
    文, 50, 97
del_param() (*email.message.EmailMessage* のメソッド), 1334
del_param() (*email.message.Message* のメソッド), 1384
delattr() (組み込み関数), 10
delay() (*turtle* モジュール), 1803
delay_output() (*curses* モジュール), 896
delayload (*http.cookiejar.FileCookieJar* の属性), 1687
delch() (*curses.window* のメソッド), 905
dele() (*poplib.POP3* のメソッド), 1621
delete() (*ftplib.FTP* のメソッド), 1618
delete() (*imaplib.IMAP4* のメソッド), 1626
delete() (*tkinter.ttk.Treeview* のメソッド), 1861
DELETE_ATTR (*opcode*), 2368
DELETE_DEREF (*opcode*), 2372
DELETE_FAST (*opcode*), 2371
DELETE_GLOBAL (*opcode*), 2369
DELETE_NAME (*opcode*), 2368
DELETE_SUBSCR (*opcode*), 2365
deleteacl() (*imaplib.IMAP4* のメソッド), 1626
deletefilehandler() (*tkinter.Widget.tk* のメソッド), 1845
DeleteKey() (*winreg* モジュール), 2396
DeleteKeyEx() (*winreg* モジュール), 2396
deleteln() (*curses.window* のメソッド), 905
deleteMe() (*bdb.Breakpoint* のメソッド), 2092
DeleteValue() (*winreg* モジュール), 2397
delimiter (*csv.Dialect* の属性), 654
delitem() (*operator* モジュール), 464
deliver_challenge() (*multiprocessing.connection* モジュール), 1032
delocalize() (*locale* モジュール), 1771
demo_app() (*wsgiref.simple_server* モジュール), 1554
denominator (*fractions.Fraction* の属性), 411
denominator (*numbers.Rational* の属性), 362
DeprecationWarning, 122
deque (*collections* のクラス), 282
Deque (*typing* のクラス), 1905
dequeue() (*logging.handlers.QueueListener* のメソッド), 892
DER_cert_to_PEM_cert() (*ssl* モジュール), 1249
derwin() (*curses.window* のメソッド), 905
DES
    cipher, 2414
description (*inspect.Parameter.kind* の属性), 2258
description (*sqlite3.Cursor* の属性), 584
description() (*nntplib.NNTP* のメソッド), 1636
descriptions() (*nntplib.NNTP* のメソッド), 1635
descriptor, **2481**
dest (*optparse.Option* の属性), 2449
detach() (*io.BufferedIOBase* のメソッド), 781
detach() (*io.TextIOBase* のメソッド), 786
detach() (*socket.socket* のメソッド), 1229
detach() (*tkinter.ttk.Treeview* のメソッド), 1861
detach() (*weakref.finalize* のメソッド), 316
Detach() (*winreg.PyHKEY* のメソッド), 2406
DETACHED_PROCESS (*subprocess* モジュール), 1077

--details
    inspect command line option, 2268
detect_api_mismatch() (test.support モジュール), 2082
detect_encoding() (tokenize モジュール), 2344
deterministic profiling, 2110
device_encoding() (os モジュール), 715
devnull (os モジュール), 771
DEVNULL (subprocess モジュール), 1064
devpoll() (select モジュール), 1288
DevpollSelector (selectors のクラス), 1300
dgettext() (gettext モジュール), 1754
dgettext() (locale モジュール), 1773
Dialect (csv のクラス), 652
dialect (csv.csvreader の属性), 655
dialect (csv.csvwriter の属性), 656
Dialog (msilib のクラス), 2391
dict (2to3 fixer), 2061
Dict (typing のクラス), 1906
dict (組み込みクラス), 98
dict() (multiprocessing.managers.SyncManager のメソッド), 1023
dictConfig() (logging.config モジュール), 864
dictionary, 2481
    type, operations on, 97
    オブジェクト, 97
dictionary comprehension, 2481
dictionary view, 2481
DictReader (csv のクラス), 651
DictWriter (csv のクラス), 652
diff_bytes() (difflib モジュール), 168
diff_files (filecmp.dircmp の属性), 509
Differ (difflib のクラス), 164
difference() (frozenset のメソッド), 96
difference_update() (frozenset のメソッド), 97
difflib (モジュール), 163
digest() (hashlib.hash のメソッド), 689
digest() (hashlib.shake のメソッド), 690
digest() (hmac モジュール), 700
digest() (hmac.HMAC のメソッド), 700
digest_size (hmac.HMAC の属性), 701
digit() (unicodedata モジュール), 181
digits (string モジュール), 125
dir() (ftplib.FTP のメソッド), 1617
dir() (組み込み関数), 10
dircmp (filecmp のクラス), 508
directory
    changing, 728
    creating, 734
    deleting, 524, 737
    site-packages, 2268
    traversal, 749, 751
    walking, 749, 751
directory ...
    compileall command line option, 2353
directory (http.server.SimpleHTTPRequestHandler の属性), 1676
Directory (msilib のクラス), 2390
DirEntry (os のクラス), 739
DirList (tkinter.tix のクラス), 1871
dirname() (os.path モジュール), 493
DirSelectBox (tkinter.tix のクラス), 1871
DirSelectDialog (tkinter.tix のクラス), 1871
DirsOnSysPath (test.support のクラス), 2084
DirTree (tkinter.tix のクラス), 1871
dis (モジュール), 2357
dis() (dis モジュール), 2359
dis() (dis.Bytecode のメソッド), 2358
dis() (pickletools モジュール), 2376
disable (pdb command), 2106
disable() (bdb.Breakpoint のメソッド), 2093
disable() (faulthandler モジュール), 2099
disable() (gc モジュール), 2244
disable() (logging モジュール), 859
disable() (profile.Profile のメソッド), 2114
disable_faulthandler() (test.support モジュール), 2075
disable_gc() (test.support モジュール), 2075
disable_interspersed_args() (optparse.OptionParser のメソッド), 2455
DisableReflectionKey() (winreg モジュール), 2402
disassemble() (dis モジュール), 2360
discard (http.cookiejar.Cookie の属性), 1692
discard() (frozenset のメソッド), 97
discard() (mailbox.Mailbox のメソッド), 1418
discard() (mailbox.MH のメソッド), 1424
discard_buffers() (asynchat.async_chat のメソッド), 1308
disco() (dis モジュール), 2360
discover() (unittest.TestLoader のメソッド), 1976
disk_usage() (shutil モジュール), 525
dispatch_call() (bdb.Bdb のメソッド), 2094
dispatch_exception() (bdb.Bdb のメソッド), 2094
dispatch_line() (bdb.Bdb のメソッド), 2094
dispatch_return() (bdb.Bdb のメソッド), 2094
dispatch_table (pickle.Pickler の属性), 540
dispatcher (asyncore のクラス), 1302
dispatcher_with_send (asyncore のクラス), 1305
display (pdb command), 2108
display_name (email.headerregistry.Address の属性), 1364
display_name (email.headerregistry.Group の属性), 1365
displayhook() (sys モジュール), 2168
dist() (math モジュール), 370
distance() (turtle モジュール), 1788
distb() (dis モジュール), 2360
distutils (モジュール), 2143
divide() (decimal.Context のメソッド), 395
divide_int() (decimal.Context のメソッド), 395
DivisionByZero (decimal のクラス), 400
divmod() (decimal.Context のメソッド), 395
divmod() (組み込み関数), 11
DllCanUnloadNow() (ctypes モジュール), 966
DllGetClassObject() (ctypes モジュール), 966
dllhandle (sys モジュール), 2168
dngettext() (gettext モジュール), 1754
dnpgettext() (gettext モジュール), 1754
do_clear() (bdb.Bdb のメソッド), 2095
do_command() (curses.textpad.Textbox のメソッド), 918
do_GET() (http.server.SimpleHTTPRequestHandler のメソッド), 1676
do_handshake() (ssl.SSLSocket のメソッド), 1261
do_HEAD() (http.server.SimpleHTTPRequestHandler のメソッド), 1676
do_POST() (http.server.CGIHTTPRequestHandler のメソッド), 1678
doc (json.JSONDecodeError の属性), 1411
doc_header (cmd.Cmd の属性), 1819
DocCGIXMLRPCRequestHandler (xmlrpc.server のクラス), 1710
DocFileSuite() (doctest モジュール), 1935
doClassCleanups() (unittest.TestCase のクラスメソッド), 1971
doCleanups() (unittest.TestCase のメソッド), 1970
docmd() (smtplib.SMTP のメソッド), 1642
docstring, 2481
docstring (doctest.DocTest の属性), 1939
DocTest (doctest のクラス), 1938
doctest (モジュール), 1919
DocTestFailure, 1946
DocTestFinder (doctest のクラス), 1940
DocTestParser (doctest のクラス), 1941
DocTestRunner (doctest のクラス), 1941
DocTestSuite() (doctest モジュール), 1936
doctype() (xml.etree.ElementTree.TreeBuilder のメソッド), 1483
documentation
    generation, 1917
    online, 1917

documentElement (*xml.dom.Document* の属性), 1493
DocXMLRPCRequestHandler (*xmlrpc.server* のクラス), 1710
DocXMLRPCServer (*xmlrpc.server* のクラス), 1710
domain (*email.headerregistry.Address* の属性), 1364
domain (*tracemalloc.DomainFilter* の属性), 2136
domain (*tracemalloc.Filter* の属性), 2136
domain (*tracemalloc.Trace* の属性), 2140
domain_initial_dot (*http.cookiejar.Cookie* の属性), 1693
domain_return_ok() (*http.cookiejar.CookiePolicy* のメソッド), 1688
domain_specified (*http.cookiejar.Cookie* の属性), 1693
DomainFilter (*tracemalloc* のクラス), 2136
DomainLiberal (*http.cookiejar.DefaultCookiePolicy* の属性), 1692
DomainRFC2965Match (*http.cookiejar.DefaultCookiePolicy* の属性), 1691
DomainStrict (*http.cookiejar.DefaultCookiePolicy* の属性), 1692
DomainStrictNoDots (*http.cookiejar.DefaultCookiePolicy* の属性), 1691
DomainStrictNonDomain (*http.cookiejar.DefaultCookiePolicy* の属性), 1691
DOMEventStream (*xml.dom.pulldom* のクラス), 1507
DOMException, 1497
doModuleCleanups() (*unittest* モジュール), 1985
DomstringSizeErr, 1497
done() (*asyncio.Future* のメソッド), 1171
done() (*asyncio.Task* のメソッド), 1115
done() (*concurrent.futures.Future* のメソッド), 1059
done() (*turtle* モジュール), 1805
done() (*xdrlib.Unpacker* のメソッド), 681
DONT_ACCEPT_BLANKLINE (*doctest* モジュール), 1927
DONT_ACCEPT_TRUE_FOR_1 (*doctest* モジュール), 1927
dont_write_bytecode (*sys* モジュール), 2169
doRollover() (*logging.handlers.RotatingFileHandler* のメソッド), 881
doRollover() (*logging.handlers.TimedRotatingFileHandler* のメソッド), 883
DOT (*token* モジュール), 2339
dot() (*turtle* モジュール), 1784
DOTALL (*re* モジュール), 148
doublequote (*csv.Dialect* の属性), 654
DOUBLESLASH (*token* モジュール), 2341
DOUBLESLASHEQUAL (*token* モジュール), 2341
DOUBLESTAR (*token* モジュール), 2340
DOUBLESTAREQUAL (*token* モジュール), 2341
doupdate() (*curses* モジュール), 896
down (*pdb command*), 2105
down() (*turtle* モジュール), 1789
dpgettext() (*gettext* モジュール), 1754
drain() (*asyncio.StreamWriter* のメソッド), 1122
drop_whitespace (*textwrap.TextWrapper* の属性), 179
dropwhile() (*itertools* モジュール), 441
dst() (*datetime.datetime* のメソッド), 245
dst() (*datetime.time* のメソッド), 255
dst() (*datetime.timezone* のメソッド), 264
dst() (*datetime.tzinfo* のメソッド), 257
DTDHandler (*xml.sax.handler* のクラス), 1511
duck-typing, **2481**
DumbWriter (*formatter* のクラス), 2383
dummy_threading (モジュール), 1099
dump() (*ast* モジュール), 2334
dump() (*json* モジュール), 1405
dump() (*marshal* モジュール), 561
dump() (*pickle* モジュール), 538
dump() (*pickle.Pickler* のメソッド), 540
dump() (*plistlib* モジュール), 684
dump() (*tracemalloc.Snapshot* のメソッド), 2138
dump() (*xml.etree.ElementTree* モジュール), 1471
dump_stats() (*profile.Profile* のメソッド), 2114
dump_stats() (*pstats.Stats* のメソッド), 2115
dump_traceback() (*faulthandler* モジュール), 2099
dump_traceback_later() (*faulthandler* モジュール), 2100
dumps() (*json* モジュール), 1406
dumps() (*marshal* モジュール), 562
dumps() (*pickle* モジュール), 538
dumps() (*plistlib* モジュール), 684
dumps() (*xmlrpc.client* モジュール), 1702
dup() (*os* モジュール), 715
dup() (*socket.socket* のメソッド), 1229
dup2() (*os* モジュール), 715
DUP_TOP (*opcode*), 2362
DUP_TOP_TWO (*opcode*), 2362
DuplicateOptionError, 677
DuplicateSectionError, 677
dwFlags (*subprocess.STARTUPINFO* の属性), 1075
DynamicClassAttribute() (*types* モジュール), 327

E

-e
    tokenize command line option, 2345
e (*cmath* モジュール), 376
e (*math* モジュール), 372
-e <tarfile> [<output_dir>]
    tarfile command line option, 644
-e <zipfile> <output_dir>
    zipfile command line option, 627
E2BIG (*errno* モジュール), 928
EACCES (*errno* モジュール), 928
EADDRINUSE (*errno* モジュール), 933
EADDRNOTAVAIL (*errno* モジュール), 933
EADV (*errno* モジュール), 931
EAFNOSUPPORT (*errno* モジュール), 933
EAFP, **2481**
EAGAIN (*errno* モジュール), 928
EALREADY (*errno* モジュール), 934
east_asian_width() (*unicodedata* モジュール), 181
EBADE (*errno* モジュール), 930
EBADF (*errno* モジュール), 928
EBADFD (*errno* モジュール), 932
EBADMSG (*errno* モジュール), 931
EBADR (*errno* モジュール), 930
EBADRQC (*errno* モジュール), 930
EBADSLT (*errno* モジュール), 930
EBFONT (*errno* モジュール), 931
EBUSY (*errno* モジュール), 928
ECHILD (*errno* モジュール), 928
echo() (*curses* モジュール), 896
echochar() (*curses.window* のメソッド), 906
ECHRNG (*errno* モジュール), 930
ECOMM (*errno* モジュール), 931
ECONNABORTED (*errno* モジュール), 933
ECONNREFUSED (*errno* モジュール), 934
ECONNRESET (*errno* モジュール), 933
EDEADLK (*errno* モジュール), 929
EDEADLOCK (*errno* モジュール), 931
EDESTADDRREQ (*errno* モジュール), 932
edit() (*curses.textpad.Textbox* のメソッド), 917
EDOM (*errno* モジュール), 929
EDOTDOT (*errno* モジュール), 931
EDQUOT (*errno* モジュール), 934
EEXIST (*errno* モジュール), 928
EFAULT (*errno* モジュール), 928
EFBIG (*errno* モジュール), 929
effective() (*bdb* モジュール), 2098
ehlo() (*smtplib.SMTP* のメソッド), 1643
ehlo_or_helo_if_needed() (*smtplib.SMTP* のメソッド), 1643
EHOSTDOWN (*errno* モジュール), 934
EHOSTUNREACH (*errno* モジュール), 934
EIDRM (*errno* モジュール), 930
EILSEQ (*errno* モジュール), 932
EINPROGRESS (*errno* モジュール), 934
EINTR (*errno* モジュール), 927

EINVAL (*errno* モジュール), 928
EIO (*errno* モジュール), 927
EISCONN (*errno* モジュール), 933
EISDIR (*errno* モジュール), 928
EISNAM (*errno* モジュール), 934
EL2HLT (*errno* モジュール), 930
EL2NSYNC (*errno* モジュール), 930
EL3HLT (*errno* モジュール), 930
EL3RST (*errno* モジュール), 930
Element (*xml.etree.ElementTree* のクラス), 1476
element_create() (*tkinter.ttk.Style* のメソッド), 1866
element_names() (*tkinter.ttk.Style* のメソッド), 1867
element_options() (*tkinter.ttk.Style* のメソッド), 1867
ElementDeclHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1528
elements() (*collections.Counter* のメソッド), 279
ElementTree (*xml.etree.ElementTree* のクラス), 1480
ELIBACC (*errno* モジュール), 932
ELIBBAD (*errno* モジュール), 932
ELIBEXEC (*errno* モジュール), 932
ELIBMAX (*errno* モジュール), 932
ELIBSCN (*errno* モジュール), 932
Ellinghouse, Lance, 1451
ELLIPSIS (*doctest* モジュール), 1928
ELLIPSIS (*token* モジュール), 2341
Ellipsis (組み込み変数), 35
ELNRNG (*errno* モジュール), 930
ELOOP (*errno* モジュール), 929
email (モジュール), 1327
email.charset (モジュール), 1394
email.contentmanager (モジュール), 1365
email.encoders (モジュール), 1397
email.errors (モジュール), 1357
email.generator (モジュール), 1344
email.header (モジュール), 1390
email.headerregistry (モジュール), 1358
email.iterators (モジュール), 1401
EmailMessage (*email.message* のクラス), 1329
email.message (モジュール), 1329
email.mime (モジュール), 1387
email.parser (モジュール), 1339
EmailPolicy (*email.policy* のクラス), 1353
email.policy (モジュール), 1348
email.utils (モジュール), 1398
EMFILE (*errno* モジュール), 929
emit() (*logging.FileHandler* のメソッド), 878
emit() (*logging.Handler* のメソッド), 850
emit() (*logging.handlers.BufferingHandler* のメソッド), 889
emit() (*logging.handlers.DatagramHandler* のメソッド), 884
emit() (*logging.handlers.HTTPHandler* のメソッド), 890
emit() (*logging.handlers.NTEventLogHandler* のメソッド), 888
emit() (*logging.handlers.QueueHandler* のメソッド), 891
emit() (*logging.handlers.RotatingFileHandler* のメソッド), 881
emit() (*logging.handlers.SMTPHandler* のメソッド), 889
emit() (*logging.handlers.SocketHandler* のメソッド), 883
emit() (*logging.handlers.SysLogHandler* のメソッド), 885
emit() (*logging.handlers.TimedRotatingFileHandler* のメソッド), 883
emit() (*logging.handlers.WatchedFileHandler* のメソッド), 879
emit() (*logging.NullHandler* のメソッド), 878
emit() (*logging.StreamHandler* のメソッド), 877
EMLINK (*errno* モジュール), 929
Empty, 1088
empty (*inspect.Parameter* の属性), 2257
empty (*inspect.Signature* の属性), 2256
empty() (*asyncio.Queue* のメソッド), 1138
empty() (*multiprocessing.Queue* のメソッド), 1007
empty() (*multiprocessing.SimpleQueue* のメソッド), 1008
empty() (*queue.Queue* のメソッド), 1089
empty() (*queue.SimpleQueue* のメソッド), 1091
empty() (*sched.scheduler* のメソッド), 1086
EMPTY_NAMESPACE (*xml.dom* モジュール), 1488
emptyline() (*cmd.Cmd* のメソッド), 1818
EMSGSIZE (*errno* モジュール), 932
EMULTIHOP (*errno* モジュール), 931
enable (*pdb command*), 2106
enable() (*bdb.Breakpoint* のメソッド), 2093
enable() (*cgitb* モジュール), 1549
enable() (*faulthandler* モジュール), 2099
enable() (*gc* モジュール), 2244
enable() (*imaplib.IMAP4* のメソッド), 1626
enable() (*profile.Profile* のメソッド), 2114
enable_callback_tracebacks() (*sqlite3* モジュール), 572
enable_interspersed_args() (*optparse.OptionParser* のメソッド), 2456
enable_load_extension() (*sqlite3.Connection* のメソッド), 576
enable_traversal() (*tkinter.ttk.Notebook* のメソッド), 1855
ENABLE_USER_SITE (*site* モジュール), 2271
EnableReflectionKey() (*winreg* モジュール), 2402
ENAMETOOLONG (*errno* モジュール), 929
ENAVAIL (*errno* モジュール), 934
enclose() (*curses.window* のメソッド), 906
encode
    Codecs, 200
encode (*codecs.CodecInfo* の属性), 201
encode() (*base64* モジュール), 1445
encode() (*codecs* モジュール), 200
encode() (*codecs.Codec* のメソッド), 207
encode() (*codecs.IncrementalEncoder* のメソッド), 208
encode() (*email.header.Header* のメソッド), 1392
encode() (*json.JSONEncoder* のメソッド), 1410
encode() (*quopri* モジュール), 1450
encode() (*str* のメソッド), 57
encode() (*uu* モジュール), 1451
encode() (*xmlrpc.client.Binary* のメソッド), 1699
encode() (*xmlrpc.client.DateTime* のメソッド), 1698
encode_7or8bit() (*email.encoders* モジュール), 1397
encode_base64() (*email.encoders* モジュール), 1397
encode_noop() (*email.encoders* モジュール), 1398
encode_quopri() (*email.encoders* モジュール), 1397
encode_rfc2231() (*email.utils* モジュール), 1400
encodebytes() (*base64* モジュール), 1445
EncodedFile() (*codecs* モジュール), 203
encodePriority() (*logging.handlers.SysLogHandler* のメソッド), 886
encodestring() (*base64* モジュール), 1446
encodestring() (*quopri* モジュール), 1451
encoding
    base64, 1442
    quoted-printable, 1450
encoding (*curses.window* の属性), 906
encoding (*io.TextIOBase* の属性), 785
ENCODING (*tarfile* モジュール), 631
ENCODING (*token* モジュール), 2341
encoding (*UnicodeError* の属性), 119
encodings_map (*mimetypes* モジュール), 1440
encodings_map (*mimetypes.MimeTypes* の属性), 1441
encodings.idna (モジュール), 222
encodings.mbcs (モジュール), 223
encodings.utf_8_sig (モジュール), 223
end (*UnicodeError* の属性), 120
end() (*re.Match* のメソッド), 156
end() (*xml.etree.ElementTree.TreeBuilder* のメソッド), 1482
END_ASYNC_FOR (*opcode*), 2365
end_col_offset (*ast.AST* の属性), 2327
end_fill() (*turtle* モジュール), 1793
END_FINALLY (*opcode*), 2367

end_headers() (*http.server.BaseHTTPRequestHandler* のメソッド), 1674
end_lineno (*ast.AST* の属性), 2327
end_ns() (*xml.etree.ElementTree.TreeBuilder* のメソッド), 1483
end_paragraph() (*formatter.formatter* のメソッド), 2380
end_poly() (*turtle* モジュール), 1799
EndCdataSectionHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1529
EndDoctypeDeclHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1528
endDocument() (*xml.sax.handler.ContentHandler* のメソッド), 1513
endElement() (*xml.sax.handler.ContentHandler* のメソッド), 1514
EndElementHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1528
endElementNS() (*xml.sax.handler.ContentHandler* のメソッド), 1515
endheaders() (*http.client.HTTPConnection* のメソッド), 1609
ENDMARKER (*token* モジュール), 2338
EndNamespaceDeclHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1529
endpos (*re.Match* の属性), 156
endPrefixMapping() (*xml.sax.handler.ContentHandler* のメソッド), 1514
endswith() (*bytearray* のメソッド), 72
endswith() (*bytes* のメソッド), 72
endswith() (*str* のメソッド), 57
endwin() (*curses* モジュール), 896
ENETDOWN (*errno* モジュール), 933
ENETRESET (*errno* モジュール), 933
ENETUNREACH (*errno* モジュール), 933
ENFILE (*errno* モジュール), 928
ENOANO (*errno* モジュール), 930
ENOBUFS (*errno* モジュール), 933
ENOCSI (*errno* モジュール), 930
ENODATA (*errno* モジュール), 931
ENODEV (*errno* モジュール), 928
ENOENT (*errno* モジュール), 927
ENOEXEC (*errno* モジュール), 928
ENOLCK (*errno* モジュール), 929
ENOLINK (*errno* モジュール), 931
ENOMEM (*errno* モジュール), 928
ENOMSG (*errno* モジュール), 930
ENONET (*errno* モジュール), 931
ENOPKG (*errno* モジュール), 931
ENOPROTOOPT (*errno* モジュール), 933
ENOSPC (*errno* モジュール), 929
ENOSR (*errno* モジュール), 931
ENOSTR (*errno* モジュール), 931
ENOSYS (*errno* モジュール), 929
ENOTBLK (*errno* モジュール), 928
ENOTCONN (*errno* モジュール), 933
ENOTDIR (*errno* モジュール), 928
ENOTEMPTY (*errno* モジュール), 929
ENOTNAM (*errno* モジュール), 934
ENOTSOCK (*errno* モジュール), 932
ENOTTY (*errno* モジュール), 929
ENOTUNIQ (*errno* モジュール), 932
enqueue() (*logging.handlers.QueueHandler* のメソッド), 891
enqueue_sentinel() (*logging.handlers.QueueListener* のメソッド), 893
ensure_directories() (*venv.EnvBuilder* のメソッド), 2151
ensure_future() (*asyncio* モジュール), 1170
ensurepip (モジュール), 2144
enter() (*sched.scheduler* のメソッド), 1086
enter_async_context() (*contextlib.AsyncExitStack* のメソッド), 2221
enter_context() (*contextlib.ExitStack* のメソッド), 2219
enterabs() (*sched.scheduler* のメソッド), 1086
entities (*xml.dom.DocumentType* の属性), 1493
EntityDeclHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1529
entitydefs (*html.entities* モジュール), 1459
EntityResolver (*xml.sax.handler* のクラス), 1511
Enum (*enum* のクラス), 338
enum (モジュール), 338
enum_certificates() (*ssl* モジュール), 1249
enum_crls() (*ssl* モジュール), 1250
enumerate() (*threading* モジュール), 978
enumerate() (組み込み関数), 11
EnumKey() (*winreg* モジュール), 2397
EnumValue() (*winreg* モジュール), 2397
EnvBuilder (*venv* のクラス), 2150
environ (*os* モジュール), 706
environ (*posix* モジュール), 2410
environb (*os* モジュール), 707
environment variables
    deleting, 714
    setting, 711
EnvironmentError, 120
Environments
    virtual, 2146
EnvironmentVarGuard (*test.support* のクラス), 2083
ENXIO (*errno* モジュール), 927
eof (*bz2.BZ2Decompressor* の属性), 606
eof (*lzma.LZMADecompressor* の属性), 613
eof (*shlex.shlex* の属性), 1826
eof (*ssl.MemoryBIO* の属性), 1284
eof (*zlib.Decompress* の属性), 598
eof_received() (*asyncio.BufferedProtocol* のメソッド), 1183
eof_received() (*asyncio.Protocol* のメソッド), 1182
EOFError, 115
EOPNOTSUPP (*errno* モジュール), 933
EOVERFLOW (*errno* モジュール), 932
EPERM (*errno* モジュール), 927
EPFNOSUPPORT (*errno* モジュール), 933
epilogue (*email.message.EmailMessage* の属性), 1339
epilogue (*email.message.Message* の属性), 1386
EPIPE (*errno* モジュール), 929
epoch, 790
epoll() (*select* モジュール), 1289
EpollSelector (*selectors* のクラス), 1300
EPROTO (*errno* モジュール), 931
EPROTONOSUPPORT (*errno* モジュール), 933
EPROTOTYPE (*errno* モジュール), 932
eq() (*operator* モジュール), 462
EQEQUAL (*token* モジュール), 2339
EQUAL (*token* モジュール), 2339
ERA (*locale* モジュール), 1768
ERA_D_FMT (*locale* モジュール), 1768
ERA_D_T_FMT (*locale* モジュール), 1768
ERA_T_FMT (*locale* モジュール), 1769
ERANGE (*errno* モジュール), 929
erase() (*curses.window* のメソッド), 906
erasechar() (*curses* モジュール), 896
EREMCHG (*errno* モジュール), 932
EREMOTE (*errno* モジュール), 931
EREMOTEIO (*errno* モジュール), 934
ERESTART (*errno* モジュール), 932
erf() (*math* モジュール), 371
erfc() (*math* モジュール), 371
EROFS (*errno* モジュール), 929
ERR (*curses* モジュール), 911
errcheck (*ctypes._FuncPtr* の属性), 961
errcode (*xmlrpc.client.ProtocolError* の属性), 1701
errmsg (*xmlrpc.client.ProtocolError* の属性), 1701
errno
    モジュール, 116
errno (*OSError* の属性), 116
errno (モジュール), 927

Error, 328, 526, 585, 653, 677, 683, 1437, 1447, 1450, 1451, 1538, 1735, 1738, 1765
error, 151, 194, 562, 564, 566, 567, 595, 705, 841, 895, 1096, 1214, 1288, 1523, 1727, 2424, 2430
error() (*argparse.ArgumentParser* のメソッド), 838
error() (*logging* モジュール), 858
error() (*logging.Logger* のメソッド), 847
error() (*urllib.request.OpenerDirector* のメソッド), 1572
error() (*xml.sax.handler.ErrorHandler* のメソッド), 1516
error_body (*wsgiref.handlers.BaseHandler* の属性), 1560
error_content_type (*http.server.BaseHTTPRequestHandler* の属性), 1673
error_headers (*wsgiref.handlers.BaseHandler* の属性), 1560
error_leader() (*shlex.shlex* のメソッド), 1825
error_message_format (*http.server.BaseHTTPRequestHandler* の属性), 1673
error_output() (*wsgiref.handlers.BaseHandler* のメソッド), 1560
error_perm, 1614
error_proto, 1614, 1620
error_received() (*asyncio.DatagramProtocol* のメソッド), 1183
error_reply, 1614
error_status (*wsgiref.handlers.BaseHandler* の属性), 1560
error_temp, 1614
ErrorByteIndex (*xml.parsers.expat.xmlparser* の属性), 1527
errorcode (*errno* モジュール), 927
ErrorCode (*xml.parsers.expat.xmlparser* の属性), 1527
ErrorColumnNumber (*xml.parsers.expat.xmlparser* の属性), 1527
ErrorHandler (*xml.sax.handler* のクラス), 1511
errorlevel (*tarfile.TarFile* の属性), 635
ErrorLineNumber (*xml.parsers.expat.xmlparser* の属性), 1527
Errors
    logging, 843
errors (*io.TextIOBase* の属性), 785
errors (*unittest.TestLoader* の属性), 1975
errors (*unittest.TestResult* の属性), 1978
ErrorString() (*xml.parsers.expat* モジュール), 1524
ERRORTOKEN (*token* モジュール), 2341
escape (*shlex.shlex* の属性), 1826
escape() (*glob* モジュール), 516
escape() (*html* モジュール), 1453
escape() (*re* モジュール), 151
escape() (*xml.sax.saxutils* モジュール), 1517
escapechar (*csv.Dialect* の属性), 654
escapedquotes (*shlex.shlex* の属性), 1826
ESHUTDOWN (*errno* モジュール), 933
ESOCKTNOSUPPORT (*errno* モジュール), 933
ESPIPE (*errno* モジュール), 929
ESRCH (*errno* モジュール), 927
ESRMNT (*errno* モジュール), 931
ESTALE (*errno* モジュール), 934
ESTRPIPE (*errno* モジュール), 932
ETIME (*errno* モジュール), 931
ETIMEDOUT (*errno* モジュール), 934
Etiny() (*decimal.Context* のメソッド), 394
ETOOMANYREFS (*errno* モジュール), 934
Etop() (*decimal.Context* のメソッド), 394
ETXTBSY (*errno* モジュール), 929
EUCLEAN (*errno* モジュール), 934
EUNATCH (*errno* モジュール), 930
EUSERS (*errno* モジュール), 932
eval
    組み込み関数, 106, 331, 332, 2324
eval() (組み込み関数), 11
Event (*asyncio* のクラス), 1128
Event (*multiprocessing* のクラス), 1014
Event (*threading* のクラス), 990
event scheduling, 1085
event() (*msilib.Control* のメソッド), 2391
Event() (*multiprocessing.managers.SyncManager* のメソッド), 1023
events (*selectors.SelectorKey* の属性), 1298
events (*widgets*), 1842
EWOULDBLOCK (*errno* モジュール), 930
EX_CANTCREAT (*os* モジュール), 757
EX_CONFIG (*os* モジュール), 757
EX_DATAERR (*os* モジュール), 756
EX_IOERR (*os* モジュール), 757
EX_NOHOST (*os* モジュール), 756
EX_NOINPUT (*os* モジュール), 756
EX_NOPERM (*os* モジュール), 757
EX_NOTFOUND (*os* モジュール), 758
EX_NOUSER (*os* モジュール), 756
EX_OK (*os* モジュール), 756
EX_OSERR (*os* モジュール), 757
EX_OSFILE (*os* モジュール), 757
EX_PROTOCOL (*os* モジュール), 757
EX_SOFTWARE (*os* モジュール), 757
EX_TEMPFAIL (*os* モジュール), 757
EX_UNAVAILABLE (*os* モジュール), 757
EX_USAGE (*os* モジュール), 756
--exact
    tokenize command line option, 2345
Example (*doctest* のクラス), 1939
example (*doctest.DocTestFailure* の属性), 1946
example (*doctest.UnexpectedException* の属性), 1947
examples (*doctest.DocTest* の属性), 1938
exc_info (*doctest.UnexpectedException* の属性), 1947
exc_info() (*sys* モジュール), 2170
exc_msg (*doctest.Example* の属性), 1939
exc_type (*traceback.TracebackException* の属性), 2238
excel (*csv* のクラス), 652
excel_tab (*csv* のクラス), 652
except
    文, 113
except (*2to3 fixer*), 2061
excepthook() (*in module sys*), 1549
excepthook() (*sys* モジュール), 2169
excepthook() (*threading* モジュール), 978
Exception, 114
EXCEPTION (*tkinter* モジュール), 1845
exception() (*asyncio.Future* のメソッド), 1172
exception() (*asyncio.Task* のメソッド), 1116
exception() (*concurrent.futures.Future* のメソッド), 1059
exception() (*logging* モジュール), 858
exception() (*logging.Logger* のメソッド), 847
exceptions
    in CGI scripts, 1549
EXDEV (*errno* モジュール), 928
exec
    組み込み関数, 12, 106, 2324
exec (*2to3 fixer*), 2061
exec() (組み込み関数), 12
exec_module() (*importlib.abc.InspectLoader* のメソッド), 2299
exec_module() (*importlib.abc.Loader* のメソッド), 2295
exec_module() (*importlib.abc.SourceLoader* のメソッド), 2301
exec_module() (*importlib.machinery.ExtensionFileLoader* のメソッド), 2308
exec_prefix (*sys* モジュール), 2170
execfile (*2to3 fixer*), 2061
execl() (*os* モジュール), 755
execle() (*os* モジュール), 755
execlp() (*os* モジュール), 755
execlpe() (*os* モジュール), 755
executable (*sys* モジュール), 2170
Executable Zip Files, 2156

Execute() (*msilib.View* のメソッド), 2387
execute() (*sqlite3.Connection* のメソッド), 573
execute() (*sqlite3.Cursor* のメソッド), 580
executemany() (*sqlite3.Connection* のメソッド), 573
executemany() (*sqlite3.Cursor* のメソッド), 581
executescript() (*sqlite3.Connection* のメソッド), 573
executescript() (*sqlite3.Cursor* のメソッド), 582
ExecutionLoader (*importlib.abc* のクラス), 2299
Executor (*concurrent.futures* のクラス), 1054
execv() (*os* モジュール), 755
execve() (*os* モジュール), 755
execvp() (*os* モジュール), 755
execvpe() (*os* モジュール), 755
ExFileSelectBox (*tkinter.tix* のクラス), 1871
EXFULL (*errno* モジュール), 930
exists() (*os.path* モジュール), 493
exists() (*pathlib.Path* のメソッド), 483
exists() (*tkinter.ttk.Treeview* のメソッド), 1861
exists() (*zipfile.Path* のメソッド), 623
exit (組み込み変数), 36
exit() (*_thread* モジュール), 1097
exit() (*argparse.ArgumentParser* のメソッド), 838
exit() (*sys* モジュール), 2171
exitcode (*multiprocessing.Process* の属性), 1003
exitfunc (*2to3 fixer*), 2061
exitonclick() (*turtle* モジュール), 1808
ExitStack (*contextlib* のクラス), 2219
exp() (*cmath* モジュール), 374
exp() (*decimal.Context* のメソッド), 395
exp() (*decimal.Decimal* のメソッド), 386
exp() (*math* モジュール), 369
expand() (*re.Match* のメソッド), 154
expand_tabs (*textwrap.TextWrapper* の属性), 178
ExpandEnvironmentStrings() (*winreg* モジュール), 2398
expandNode() (*xml.dom.pulldom.DOMEventStream* のメソッド), 1508
expandtabs() (*bytearray* のメソッド), 78
expandtabs() (*bytes* のメソッド), 78
expandtabs() (*str* のメソッド), 57
expanduser() (*os.path* モジュール), 493
expanduser() (*pathlib.Path* のメソッド), 484
expandvars() (*os.path* モジュール), 494
Expat, 1523
ExpatError, 1523
expect() (*telnetlib.Telnet* のメソッド), 1655
expected (*asyncio.IncompleteReadError* の属性), 1141
expectedFailure() (*unittest* モジュール), 1958
expectedFailures (*unittest.TestResult* の属性), 1978
expires (*http.cookiejar.Cookie* の属性), 1692
exploded (*ipaddress.IPv4Address* の属性), 1713
exploded (*ipaddress.IPv4Network* の属性), 1718
exploded (*ipaddress.IPv6Address* の属性), 1714
exploded (*ipaddress.IPv6Network* の属性), 1722
expm1() (*math* モジュール), 369
expovariate() (*random* モジュール), 417
expr() (*parser* モジュール), 2322
expression, **2482**
expunge() (*imaplib.IMAP4* のメソッド), 1626
extend() (*array.array* のメソッド), 310
extend() (*collections.deque* のメソッド), 282
extend() (*sequence method*), 50
extend() (*xml.etree.ElementTree.Element* のメソッド), 1477
extend_path() (*pkgutil* モジュール), 2282
EXTENDED_ARG (*opcode*), 2374
ExtendedContext (*decimal* のクラス), 392
ExtendedInterpolation (*configparser* のクラス), 662
extendleft() (*collections.deque* のメソッド), 282
extension module, **2482**
EXTENSION_SUFFIXES (*importlib.machinery* モジュール), 2304
ExtensionFileLoader (*importlib.machinery* のクラス), 2307
extensions_map (*http.server.SimpleHTTPRequestHandler* の属性), 1675
External Data Representation, 537, 679
external_attr (*zipfile.ZipInfo* の属性), 626
ExternalClashError, 1437
ExternalEntityParserCreate() (*xml.parsers.expat.xmlparser* のメソッド), 1525
ExternalEntityRefHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1530
extra (*zipfile.ZipInfo* の属性), 626
--extract <tarfile> [<output_dir>]
    tarfile command line option, 644
--extract <zipfile> <output_dir>
    zipfile command line option, 627
extract() (*tarfile.TarFile* のメソッド), 634
extract() (*traceback.StackSummary* のクラスメソッド), 2239
extract() (*zipfile.ZipFile* のメソッド), 620
extract_cookies() (*http.cookiejar.CookieJar* のメソッド), 1685
extract_stack() (*traceback* モジュール), 2236
extract_tb() (*traceback* モジュール), 2236
extract_version (*zipfile.ZipInfo* の属性), 626
extractall() (*tarfile.TarFile* のメソッド), 634
extractall() (*zipfile.ZipFile* のメソッド), 620
ExtractError, 631
extractfile() (*tarfile.TarFile* のメソッド), 635
extraction_filter (*tarfile.TarFile* の属性), 635
extsep (*os* モジュール), 771

F

-f
    compileall command line option, 2353
    trace command line option, 2127
    unittest command line option, 1952
f-string, **2482**
f_contiguous (*memoryview* の属性), 94
F_LOCK (*os* モジュール), 717
F_OK (*os* モジュール), 728
F_TEST (*os* モジュール), 717
F_TLOCK (*os* モジュール), 717
F_ULOCK (*os* モジュール), 717
fabs() (*math* モジュール), 365
factorial() (*math* モジュール), 365
factory() (*importlib.util.LazyLoader* のクラスメソッド), 2313
fail() (*unittest.TestCase* のメソッド), 1969
FAIL_FAST (*doctest* モジュール), 1929
--failfast
    unittest command line option, 1952
failfast (*unittest.TestResult* の属性), 1979
failureException (*unittest.TestCase* の属性), 1969
failures (*unittest.TestResult* の属性), 1978
FakePath (*test.support* のクラス), 2084
False, 37, 107
false, 37
False (*Built-in object*), 37
False (組み込み変数), 35
family (*socket.socket* の属性), 1237
FancyURLopener (*urllib.request* のクラス), 1586
--fast
    gzip command line option, 603
fast (*pickle.Pickler* の属性), 541
FastChildWatcher (*asyncio* のクラス), 1194
fatalError() (*xml.sax.handler.ErrorHandler* のメソッド), 1516
Fault (*xmlrpc.client* のクラス), 1700
faultCode (*xmlrpc.client.Fault* の属性), 1700
faulthandler (モジュール), 2098
faultString (*xmlrpc.client.Fault* の属性), 1700
fchdir() (*os* モジュール), 731
fchmod() (*os* モジュール), 716

fchown() (*os* モジュール), 716
FCICreate() (*msilib* モジュール), 2385
fcntl (モジュール), 2420
fcntl() (*fcntl* モジュール), 2420
fd (*selectors.SelectorKey* の属性), 1298
fd() (*turtle* モジュール), 1781
fd_count() (*test.support* モジュール), 2071
fdatasync() (*os* モジュール), 716
fdopen() (*os* モジュール), 714
Feature (*msilib* のクラス), 2390
feature_external_ges (*xml.sax.handler* モジュール), 1512
feature_external_pes (*xml.sax.handler* モジュール), 1512
feature_namespace_prefixes (*xml.sax.handler* モジュール), 1511
feature_namespaces (*xml.sax.handler* モジュール), 1511
feature_string_interning (*xml.sax.handler* モジュール), 1511
feature_validation (*xml.sax.handler* モジュール), 1511
feed() (*email.parser.BytesFeedParser* のメソッド), 1341
feed() (*html.parser.HTMLParser* のメソッド), 1455
feed() (*xml.etree.ElementTree.XMLParser* のメソッド), 1484
feed() (*xml.etree.ElementTree.XMLPullParser* のメソッド), 1485
feed() (*xml.sax.xmlreader.IncrementalParser* のメソッド), 1521
FeedParser (*email.parser* のクラス), 1341
fetch() (*imaplib.IMAP4* のメソッド), 1626
Fetch() (*msilib.View* のメソッド), 2387
fetchall() (*sqlite3.Cursor* のメソッド), 583
fetchmany() (*sqlite3.Cursor* のメソッド), 583
fetchone() (*sqlite3.Cursor* のメソッド), 583
fflags (*select.kevent* の属性), 1296
Field (*dataclasses* のクラス), 2207
field() (*dataclasses* モジュール), 2205
field_size_limit() (*csv* モジュール), 651
fieldnames (*csv.csvreader* の属性), 655
fields (*uuid.UUID* の属性), 1657
fields() (*dataclasses* モジュール), 2207
file
    byte-code, 2350, 2467
    configuration, 657
    copying, 519
    debugger configuration, 2105
    gzip command line option, 603
    .ini, 657
    large files, 2409
    mime.types, 1440
    modes, 20
    path configuration, 2269
    .pdbrc, 2105
    plist, 683
    temporary, 510
file ...
    compileall command line option, 2353
file (*pyclbr.Class* の属性), 2350
file (*pyclbr.Function* の属性), 2349
file control
    UNIX, 2420
file name
    temporary, 510
file object, **2482**
    io module, 773
    open() built-in function, 20
--file=<file>
    trace command line option, 2127
file-like object, **2482**
FILE_ATTRIBUTE_ARCHIVE (*stat* モジュール), 506
FILE_ATTRIBUTE_COMPRESSED (*stat* モジュール), 506
FILE_ATTRIBUTE_DEVICE (*stat* モジュール), 506
FILE_ATTRIBUTE_DIRECTORY (*stat* モジュール), 506
FILE_ATTRIBUTE_ENCRYPTED (*stat* モジュール), 506
FILE_ATTRIBUTE_HIDDEN (*stat* モジュール), 506
FILE_ATTRIBUTE_INTEGRITY_STREAM (*stat* モジュール), 506
FILE_ATTRIBUTE_NO_SCRUB_DATA (*stat* モジュール), 506
FILE_ATTRIBUTE_NORMAL (*stat* モジュール), 506
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED (*stat* モジュール), 506
FILE_ATTRIBUTE_OFFLINE (*stat* モジュール), 506
FILE_ATTRIBUTE_READONLY (*stat* モジュール), 506
FILE_ATTRIBUTE_REPARSE_POINT (*stat* モジュール), 506
FILE_ATTRIBUTE_SPARSE_FILE (*stat* モジュール), 506
FILE_ATTRIBUTE_SYSTEM (*stat* モジュール), 506
FILE_ATTRIBUTE_TEMPORARY (*stat* モジュール), 506
FILE_ATTRIBUTE_VIRTUAL (*stat* モジュール), 506
file_dispatcher (*asyncore* のクラス), 1305
file_open() (*urllib.request.FileHandler* のメソッド), 1579
file_size (*zipfile.ZipInfo* の属性), 626
file_wrapper (*asyncore* のクラス), 1305
filecmp (モジュール), 507
fileConfig() (*logging.config* モジュール), 864
FileCookieJar (*http.cookiejar* のクラス), 1683
FileEntry (*tkinter.tix* のクラス), 1871
FileExistsError, 121
FileFinder (*importlib.machinery* のクラス), 2305
FileHandler (*logging* のクラス), 877
FileHandler (*urllib.request* のクラス), 1569
FileInput (*fileinput* のクラス), 499
fileinput (モジュール), 498
FileIO (*io* のクラス), 782
filelineno() (*fileinput* モジュール), 499
FileLoader (*importlib.abc* のクラス), 2299
filemode() (*stat* モジュール), 502
filename (*doctest.DocTest* の属性), 1938
filename (*http.cookiejar.FileCookieJar* の属性), 1687
filename (*OSError* の属性), 117
filename (*SyntaxError* の属性), 118
filename (*traceback.TracebackException* の属性), 2238
filename (*tracemalloc.Frame* の属性), 2137
filename (*zipfile.ZipFile* の属性), 622
filename (*zipfile.ZipInfo* の属性), 625
filename() (*fileinput* モジュール), 499
filename2 (*OSError* の属性), 117
filename_only (*tabnanny* モジュール), 2348
filename_pattern (*tracemalloc.Filter* の属性), 2137
filenames
    pathname expansion, 515
    wildcard expansion, 517
fileno() (*fileinput* モジュール), 499
fileno() (*http.client.HTTPResponse* のメソッド), 1610
fileno() (*io.IOBase* のメソッド), 778
fileno() (*multiprocessing.connection.Connection* のメソッド), 1012
fileno() (*ossaudiodev.oss_audio_device* のメソッド), 1747
fileno() (*ossaudiodev.oss_mixer_device* のメソッド), 1750
fileno() (*select.devpoll* のメソッド), 1291
fileno() (*select.epoll* のメソッド), 1293
fileno() (*select.kqueue* のメソッド), 1295
fileno() (*selectors.DevpollSelector* のメソッド), 1300
fileno() (*selectors.EpollSelector* のメソッド), 1300
fileno() (*selectors.KqueueSelector* のメソッド), 1301
fileno() (*socketserver.BaseServer* のメソッド), 1663
fileno() (*socket.socket* のメソッド), 1229
fileno() (*telnetlib.Telnet* のメソッド), 1654
FileNotFoundError, 121
fileobj (*selectors.SelectorKey* の属性), 1298
FileSelectBox (*tkinter.tix* のクラス), 1871
FileType (*argparse* のクラス), 834
FileWrapper (*wsgiref.util* のクラス), 1552
fill() (*textwrap* モジュール), 176
fill() (*textwrap.TextWrapper* のメソッド), 180
fillcolor() (*turtle* モジュール), 1791
filling() (*turtle* モジュール), 1792
filter (*2to3 fixer*), 2061

Filter (*logging* のクラス), 852
filter (*select.kevent* の属性), 1295
Filter (*tracemalloc* のクラス), 2136
--filter <filtername>
    tarfile command line option, 645
filter() (*curses* モジュール), 896
filter() (*fnmatch* モジュール), 518
filter() (*logging.Filter* のメソッド), 852
filter() (*logging.Handler* のメソッド), 849
filter() (*logging.Logger* のメソッド), 847
filter() (組み込み関数), 13
FILTER_DIR (*unittest.mock* モジュール), 2028
filter_traces() (*tracemalloc.Snapshot* のメソッド), 2138
FilterError, 631
filterfalse() (*itertools* モジュール), 441
filterwarnings() (*warnings* モジュール), 2202
Final (*typing* モジュール), 1916
final() (*typing* モジュール), 1912
finalize (*weakref* のクラス), 316
find() (*bytearray* のメソッド), 72
find() (*bytes* のメソッド), 72
find() (*doctest.DocTestFinder* のメソッド), 1940
find() (*gettext* モジュール), 1755
find() (*mmap.mmap* のメソッド), 1323
find() (*str* のメソッド), 57
find() (*xml.etree.ElementTree.Element* のメソッド), 1477
find() (*xml.etree.ElementTree.ElementTree* のメソッド), 1480
find_class() (*pickle protocol*), 554
find_class() (*pickle.Unpickler* のメソッド), 542
find_library() (*ctypes.util* モジュール), 966
find_loader() (*importlib* モジュール), 2291
find_loader() (*importlib.abc.PathEntryFinder* のメソッド), 2294
find_loader() (*importlib.machinery.FileFinder* のメソッド), 2306
find_loader() (*pkgutil* モジュール), 2283
find_longest_match() (*difflib.SequenceMatcher* のメソッド), 169
find_module() (*imp* モジュール), 2467
find_module() (*imp.NullImporter* のメソッド), 2472
find_module() (*importlib.abc.Finder* のメソッド), 2293
find_module() (*importlib.abc.MetaPathFinder* のメソッド), 2294
find_module() (*importlib.abc.PathEntryFinder* のメソッド), 2295
find_module() (*importlib.machinery.PathFinder* のクラスメソッド), 2305
find_module() (*zipimport.zipimporter* のメソッド), 2280
find_msvcrt() (*ctypes.util* モジュール), 966
find_spec() (*importlib.abc.MetaPathFinder* のメソッド), 2294
find_spec() (*importlib.abc.PathEntryFinder* のメソッド), 2294
find_spec() (*importlib.machinery.FileFinder* のメソッド), 2306
find_spec() (*importlib.machinery.PathFinder* のクラスメソッド), 2305
find_spec() (*importlib.util* モジュール), 2311
find_unused_port() (*test.support* モジュール), 2081
find_user_password() (*urllib.request.HTTPPasswordMgr* のメソッド), 1576
find_user_password() (*urllib.request.HTTPPasswordMgrWithPriorAuth* のメソッド), 1577
findall() (*re* モジュール), 149
findall() (*re.Pattern* のメソッド), 153
findall() (*xml.etree.ElementTree.Element* のメソッド), 1477
findall() (*xml.etree.ElementTree.ElementTree* のメソッド), 1480
findCaller() (*logging.Logger* のメソッド), 847
finder, **2482**
Finder (*importlib.abc* のクラス), 2293
findfactor() (*audioop* モジュール), 1728
findfile() (*test.support* モジュール), 2071
findfit() (*audioop* モジュール), 1728
finditer() (*re* モジュール), 149
finditer() (*re.Pattern* のメソッド), 153
findlabels() (*dis* モジュール), 2361
findlinestarts() (*dis* モジュール), 2360
findmatch() (*mailcap* モジュール), 1415
findmax() (*audioop* モジュール), 1728
findtext() (*xml.etree.ElementTree.Element* のメソッド), 1478
findtext() (*xml.etree.ElementTree.ElementTree* のメソッド), 1480
finish() (*socketserver.BaseRequestHandler* のメソッド), 1666
finish_request() (*socketserver.BaseServer* のメソッド), 1665
firstChild (*xml.dom.Node* の属性), 1490
firstkey() (*dbm.gnu.gdbm* のメソッド), 565
firstweekday() (*calendar* モジュール), 273
fix_missing_locations() (*ast* モジュール), 2332
fix_sentence_endings (*textwrap.TextWrapper* の属性), 179
Flag (*enum* のクラス), 339
flag_bits (*zipfile.ZipInfo* の属性), 626
flags (*re.Pattern* の属性), 153
flags (*select.kevent* の属性), 1296
flags (*sys* モジュール), 2171
flash() (*curses* モジュール), 897
flatten() (*email.generator.BytesGenerator* のメソッド), 1345
flatten() (*email.generator.Generator* のメソッド), 1346
flattening
    objects, 535
float
    組み込み関数, 39
float (組み込みクラス), 13
float_info (*sys* モジュール), 2172
float_repr_style (*sys* モジュール), 2173
floating point
    literals, 39
    オブジェクト, 39
FloatingPointError, 115
FloatOperation (*decimal* のクラス), 401
flock() (*fcntl* モジュール), 2421
floor division, **2482**
floor() (*in module math*), 40
floor() (*math* モジュール), 366
floordiv() (*operator* モジュール), 463
flush() (*bz2.BZ2Compressor* のメソッド), 606
flush() (*formatter.writer* のメソッド), 2382
flush() (*io.BufferedWriter* のメソッド), 784
flush() (*io.IOBase* のメソッド), 778
flush() (*logging.Handler* のメソッド), 849
flush() (*logging.handlers.BufferingHandler* のメソッド), 889
flush() (*logging.handlers.MemoryHandler* のメソッド), 890
flush() (*logging.StreamHandler* のメソッド), 877
flush() (*lzma.LZMACompressor* のメソッド), 612
flush() (*mailbox.Mailbox* のメソッド), 1420
flush() (*mailbox.Maildir* のメソッド), 1422
flush() (*mailbox.MH* のメソッド), 1425
flush() (*mmap.mmap* のメソッド), 1323
flush() (*xml.etree.ElementTree.XMLParser* のメソッド), 1484
flush() (*xml.etree.ElementTree.XMLPullParser* のメソッド), 1485
flush() (*zlib.Compress* のメソッド), 598
flush() (*zlib.Decompress* のメソッド), 599
flush_headers() (*http.server.BaseHTTPRequestHandler* のメソッド), 1674

flush_softspace() (*formatter.formatter* のメソッド), 2380
flushinp() (*curses* モジュール), 897
FlushKey() (*winreg* モジュール), 2398
fma() (*decimal.Context* のメソッド), 395
fma() (*decimal.Decimal* のメソッド), 386
fmean() (*statistics* モジュール), 422
fmod() (*math* モジュール), 366
FMT_BINARY (*plistlib* モジュール), 686
FMT_XML (*plistlib* モジュール), 686
fnmatch (モジュール), 517
fnmatch() (*fnmatch* モジュール), 517
fnmatchcase() (*fnmatch* モジュール), 518
focus() (*tkinter.ttk.Treeview* のメソッド), 1861
fold (*datetime.datetime* の属性), 242
fold (*datetime.time* の属性), 252
fold() (*email.headerregistry.BaseHeader* のメソッド), 1359
fold() (*email.policy.Compat32* のメソッド), 1356
fold() (*email.policy.EmailPolicy* のメソッド), 1354
fold() (*email.policy.Policy* のメソッド), 1352
fold_binary() (*email.policy.Compat32* のメソッド), 1356
fold_binary() (*email.policy.EmailPolicy* のメソッド), 1354
fold_binary() (*email.policy.Policy* のメソッド), 1353
FOR_ITER (*opcode*), 2371
forget() (*test.support* モジュール), 2070
forget() (*tkinter.ttk.Notebook* のメソッド), 1854
fork() (*os* モジュール), 758
fork() (*pty* モジュール), 2418
ForkingMixIn (*socketserver* のクラス), 1662
ForkingTCPServer (*socketserver* のクラス), 1662
ForkingUDPServer (*socketserver* のクラス), 1662
forkpty() (*os* モジュール), 758
Form (*tkinter.tix* のクラス), 1873
format (*memoryview* の属性), 93
format (*multiprocessing.shared_memory.ShareableList* の属性), 1052
format (*struct.Struct* の属性), 199
format() (*locale* モジュール), 1770
format() (*logging.Formatter* のメソッド), 851
format() (*logging.Handler* のメソッド), 850
format() (*pprint.PrettyPrinter* のメソッド), 332
format() (*str* のメソッド), 58
format() (*string.Formatter* のメソッド), 126
format() (*traceback.StackSummary* のメソッド), 2239
format() (*traceback.TracebackException* のメソッド), 2238
format() (*tracemalloc.Traceback* のメソッド), 2140
format() (組み込み関数), 14
format_datetime() (*email.utils* モジュール), 1400
format_exc() (*traceback* モジュール), 2237
format_exception() (*traceback* モジュール), 2237
format_exception_only() (*traceback* モジュール), 2237
format_exception_only() (*traceback.TracebackException* のメソッド), 2239
format_field() (*string.Formatter* のメソッド), 127
format_help() (*argparse.ArgumentParser* のメソッド), 837
format_list() (*traceback* モジュール), 2236
format_map() (*str* のメソッド), 58
format_stack() (*traceback* モジュール), 2237
format_stack_entry() (*bdb.Bdb* のメソッド), 2097
format_string() (*locale* モジュール), 1770
format_tb() (*traceback* モジュール), 2237
format_usage() (*argparse.ArgumentParser* のメソッド), 837
FORMAT_VALUE (*opcode*), 2374
formataddr() (*email.utils* モジュール), 1399
formatargspec() (*inspect* モジュール), 2261
formatargvalues() (*inspect* モジュール), 2262
formatdate() (*email.utils* モジュール), 1400
FormatError, 1437
FormatError() (*ctypes* モジュール), 966
formatException() (*logging.Formatter* のメソッド), 852
formatmonth() (*calendar.HTMLCalendar* のメソッド), 271
formatmonth() (*calendar.TextCalendar* のメソッド), 271
formatStack() (*logging.Formatter* のメソッド), 852
Formatter (*logging* のクラス), 850
Formatter (*string* のクラス), 126
formatter (モジュール), 2379
formatTime() (*logging.Formatter* のメソッド), 851
formatting
    bytearray (%), 83
    bytes (%), 83
formatting, string (%), 65
formatwarning() (*warnings* モジュール), 2201
formatyear() (*calendar.HTMLCalendar* のメソッド), 271
formatyear() (*calendar.TextCalendar* のメソッド), 271
formatyearpage() (*calendar.HTMLCalendar* のメソッド), 271
Fortran contiguous, 2480
forward() (*turtle* モジュール), 1781
ForwardRef (*typing* のクラス), 1910
found_terminator() (*asynchat.async_chat* のメソッド), 1308
fpathconf() (*os* モジュール), 716
fqdn (*smtpd.SMTPChannel* の属性), 1651
Fraction (*fractions* のクラス), 410
fractions (モジュール), 410
frame (*tkinter.scrolledtext.ScrolledText* の属性), 1875
Frame (*tracemalloc* のクラス), 2137
FrameSummary (*traceback* のクラス), 2240
FrameType (*types* モジュール), 325
freeze() (*gc* モジュール), 2247
freeze_support() (*multiprocessing* モジュール), 1010
frexp() (*math* モジュール), 366
from_address() (*ctypes._CData* のメソッド), 968
from_buffer() (*ctypes._CData* のメソッド), 968
from_buffer_copy() (*ctypes._CData* のメソッド), 968
from_bytes() (*int* のクラスメソッド), 43
from_callable() (*inspect.Signature* のクラスメソッド), 2257
from_decimal() (*fractions.Fraction* のメソッド), 412
from_exception() (*traceback.TracebackException* のクラスメソッド), 2238
from_file() (*zipfile.ZipInfo* のクラスメソッド), 625
from_float() (*decimal.Decimal* のメソッド), 386
from_float() (*fractions.Fraction* のメソッド), 412
from_iterable() (*itertools.chain* のクラスメソッド), 438
from_list() (*traceback.StackSummary* のクラスメソッド), 2239
from_param() (*ctypes._CData* のメソッド), 969
from_samples() (*statistics.NormalDist* のクラスメソッド), 430
from_traceback() (*dis.Bytecode* のクラスメソッド), 2358
frombuf() (*tarfile.TarInfo* のクラスメソッド), 637
frombytes() (*array.array* のメソッド), 310
fromfd() (*select.epoll* のメソッド), 1293
fromfd() (*select.kqueue* のメソッド), 1295
fromfd() (*socket* モジュール), 1221
fromfile() (*array.array* のメソッド), 311
fromhex() (*bytearray* のクラスメソッド), 70
fromhex() (*bytes* のクラスメソッド), 69
fromhex() (*float* のクラスメソッド), 44
fromisocalendar() (*datetime.date* のクラスメソッド), 233
fromisocalendar() (*datetime.datetime* のクラスメソッド), 241
fromisoformat() (*datetime.date* のクラスメソッド), 233
fromisoformat() (*datetime.datetime* のクラスメソッド), 241
fromisoformat() (*datetime.time* のクラスメソッド), 253
fromkeys() (*collections.Counter* のメソッド), 280
fromkeys() (*dict* のクラスメソッド), 99
fromlist() (*array.array* のメソッド), 311
fromordinal() (*datetime.date* のクラスメソッド), 233
fromordinal() (*datetime.datetime* のクラスメソッド), 240
fromshare() (*socket* モジュール), 1221
fromstring() (*array.array* のメソッド), 311

fromstring() (*xml.etree.ElementTree* モジュール), 1472
fromstringlist() (*xml.etree.ElementTree* モジュール), 1472
fromtarfile() (*tarfile.TarInfo* のクラスメソッド), 637
fromtimestamp() (*datetime.date* のクラスメソッド), 233
fromtimestamp() (*datetime.datetime* のクラスメソッド), 239
fromunicode() (*array.array* のメソッド), 311
fromutc() (*datetime.timezone* のメソッド), 264
fromutc() (*datetime.tzinfo* のメソッド), 258
FrozenImporter (*importlib.machinery* のクラス), 2304
FrozenInstanceError, 2213
FrozenSet (*typing* のクラス), 1905
frozenset (組み込みクラス), 95
fs_is_case_insensitive() (*test.support* モジュール), 2081
FS_NONASCII (*test.support* モジュール), 2069
fsdecode() (*os* モジュール), 707
fsencode() (*os* モジュール), 707
fspath() (*os* モジュール), 708
fstat() (*os* モジュール), 716
fstatvfs() (*os* モジュール), 717
fsum() (*math* モジュール), 366
fsync() (*os* モジュール), 717
FTP, 1587
    ftplib (*standard module*), 1612
    protocol, 1587, 1612
FTP (*ftplib* のクラス), 1613
ftp_open() (*urllib.request.FTPHandler* のメソッド), 1580
FTP_TLS (*ftplib* のクラス), 1613
FTPHandler (*urllib.request* のクラス), 1569
ftplib (モジュール), 1612
ftruncate() (*os* モジュール), 717
Full, 1088
full() (*asyncio.Queue* のメソッド), 1138
full() (*multiprocessing.Queue* のメソッド), 1007
full() (*queue.Queue* のメソッド), 1089
full_url (*urllib.request.Request* の属性), 1570
fullmatch() (*re* モジュール), 148
fullmatch() (*re.Pattern* のメソッド), 153
fully_trusted_filter() (*tarfile* モジュール), 641
func (*functools.partial* の属性), 461
funcattrs (*2to3 fixer*), 2061
function, **2482**
Function (*symtable* のクラス), 2335
function annotation, **2482**
FunctionTestCase (*unittest* のクラス), 1972
FunctionType (*types* モジュール), 323
functools (モジュール), 452
funny_files (*filecmp.dircmp* の属性), 509
future (*2to3 fixer*), 2062
Future (*asyncio* のクラス), 1171
Future (*concurrent.futures* のクラス), 1059
FutureWarning, 122
fwalk() (*os* モジュール), 751

G

-g
    trace command line option, 2127
G.722, 1733
gaierror, 1215
gamma() (*math* モジュール), 372
gammavariate() (*random* モジュール), 417
garbage (*gc* モジュール), 2247
garbage collection, **2483**
gather() (*asyncio* モジュール), 1108
gather() (*curses.textpad.Textbox* のメソッド), 918
gauss() (*random* モジュール), 417
gc (モジュール), 2244
gc_collect() (*test.support* モジュール), 2075
gcd() (*fractions* モジュール), 413
gcd() (*math* モジュール), 366
ge() (*operator* モジュール), 462
gen_uuid() (*msilib* モジュール), 2386
generate_tokens() (*tokenize* モジュール), 2343
generator, **2483**
Generator (*collections.abc* のクラス), 298
Generator (*email.generator* のクラス), 1346
Generator (*typing* のクラス), 1907
generator expression, **2483**
generator iterator, **2483**
GeneratorExit, 115
GeneratorType (*types* モジュール), 323
Generic (*typing* のクラス), 1902
generic function, **2483**
generic_visit() (*ast.NodeVisitor* のメソッド), 2333
genops() (*pickletools* モジュール), 2376
geometric_mean() (*statistics* モジュール), 423
get() (*asyncio.Queue* のメソッド), 1138
get() (*configparser.ConfigParser* のメソッド), 674
get() (*contextvars.Context* のメソッド), 1095
get() (*contextvars.ContextVar* のメソッド), 1092
get() (*dict* のメソッド), 100
get() (*email.message.EmailMessage* のメソッド), 1332
get() (*email.message.Message* のメソッド), 1381
get() (*mailbox.Mailbox* のメソッド), 1419
get() (*multiprocessing.pool.AsyncResult* のメソッド), 1031
get() (*multiprocessing.Queue* のメソッド), 1007
get() (*multiprocessing.SimpleQueue* のメソッド), 1008
get() (*ossaudiodev.oss_mixer_device* のメソッド), 1751
get() (*queue.Queue* のメソッド), 1089
get() (*queue.SimpleQueue* のメソッド), 1091
get() (*tkinter.ttk.Combobox* のメソッド), 1851
get() (*tkinter.ttk.Spinbox* のメソッド), 1852
get() (*types.MappingProxyType* のメソッド), 326
get() (*webbrowser* モジュール), 1538
get() (*xml.etree.ElementTree.Element* のメソッド), 1477
GET_AITER (*opcode*), 2365
get_all() (*email.message.EmailMessage* のメソッド), 1332
get_all() (*email.message.Message* のメソッド), 1381
get_all() (*wsgiref.headers.Headers* のメソッド), 1553
get_all_breaks() (*bdb.Bdb* のメソッド), 2097
get_all_start_methods() (*multiprocessing* モジュール), 1010
GET_ANEXT (*opcode*), 2365
get_app() (*wsgiref.simple_server.WSGIServer* のメソッド), 1555
get_archive_formats() (*shutil* モジュール), 529
get_args() (*typing* モジュール), 1911
get_asyncgen_hooks() (*sys* モジュール), 2176
get_attribute() (*test.support* モジュール), 2079
GET_AWAITABLE (*opcode*), 2365
get_begidx() (*readline* モジュール), 188
get_blocking() (*os* モジュール), 717
get_body() (*email.message.EmailMessage* のメソッド), 1336
get_body_encoding() (*email.charset.Charset* のメソッド), 1395
get_boundary() (*email.message.EmailMessage* のメソッド), 1334
get_boundary() (*email.message.Message* のメソッド), 1384
get_bpbynumber() (*bdb.Bdb* のメソッド), 2096
get_break() (*bdb.Bdb* のメソッド), 2097
get_breaks() (*bdb.Bdb* のメソッド), 2097
get_buffer() (*asyncio.BufferedProtocol* のメソッド), 1183
get_buffer() (*xdrlib.Packer* のメソッド), 680
get_buffer() (*xdrlib.Unpacker* のメソッド), 681
get_bytes() (*mailbox.Mailbox* のメソッド), 1419
get_ca_certs() (*ssl.SSLContext* のメソッド), 1267
get_cache_token() (*abc* モジュール), 2233
get_channel_binding() (*ssl.SSLSocket* のメソッド), 1262
get_charset() (*email.message.Message* のメソッド), 1380
get_charsets() (*email.message.EmailMessage* のメソッド), 1335
get_charsets() (*email.message.Message* のメソッド), 1385

get_child_watcher() (*asyncio* モジュール), 1192
get_child_watcher() (*asyncio.AbstractEventLoopPolicy* のメソッド), 1191
get_children() (*symtable.SymbolTable* のメソッド), 2335
get_children() (*tkinter.ttk.Treeview* のメソッド), 1860
get_ciphers() (*ssl.SSLContext* のメソッド), 1267
get_clock_info() (*time* モジュール), 793
get_close_matches() (*difflib* モジュール), 166
get_code() (*importlib.abc.InspectLoader* のメソッド), 2298
get_code() (*importlib.abc.SourceLoader* のメソッド), 2301
get_code() (*importlib.machinery.ExtensionFileLoader* のメソッド), 2308
get_code() (*importlib.machinery.SourcelessFileLoader* のメソッド), 2307
get_code() (*zipimport.zipimporter* のメソッド), 2280
get_completer() (*readline* モジュール), 188
get_completer_delims() (*readline* モジュール), 188
get_completion_type() (*readline* モジュール), 188
get_config_h_filename() (*sysconfig* モジュール), 2193
get_config_var() (*sysconfig* モジュール), 2190
get_config_vars() (*sysconfig* モジュール), 2190
get_content() (*email.contentmanager* モジュール), 1367
get_content() (*email.contentmanager.ContentManager* のメソッド), 1365
get_content() (*email.message.EmailMessage* のメソッド), 1337
get_content_charset() (*email.message.EmailMessage* のメソッド), 1335
get_content_charset() (*email.message.Message* のメソッド), 1385
get_content_disposition() (*email.message.EmailMessage* のメソッド), 1335
get_content_disposition() (*email.message.Message* のメソッド), 1385
get_content_maintype() (*email.message.EmailMessage* のメソッド), 1333
get_content_maintype() (*email.message.Message* のメソッド), 1382
get_content_subtype() (*email.message.EmailMessage* のメソッド), 1333
get_content_subtype() (*email.message.Message* のメソッド), 1382
get_content_type() (*email.message.EmailMessage* のメソッド), 1333
get_content_type() (*email.message.Message* のメソッド), 1382
get_context() (*multiprocessing* モジュール), 1010
get_coro() (*asyncio.Task* のメソッド), 1117
get_coroutine_origin_tracking_depth() (*sys* モジュール), 2176
get_count() (*gc* モジュール), 2246
get_current_history_length() (*readline* モジュール), 187
get_data() (*importlib.abc.FileLoader* のメソッド), 2300
get_data() (*importlib.abc.ResourceLoader* のメソッド), 2298
get_data() (*pkgutil* モジュール), 2285
get_data() (*zipimport.zipimporter* のメソッド), 2280
get_date() (*mailbox.MaildirMessage* のメソッド), 1429
get_debug() (*asyncio.loop* のメソッド), 1161
get_debug() (*gc* モジュール), 2245
get_default() (*argparse.ArgumentParser* のメソッド), 836
get_default_domain() (*nis* モジュール), 2430
get_default_type() (*email.message.EmailMessage* のメソッド), 1333
get_default_type() (*email.message.Message* のメソッド), 1382
get_default_verify_paths() (*ssl* モジュール), 1249
get_dialect() (*csv* モジュール), 651
get_docstring() (*ast* モジュール), 2331
get_doctest() (*doctest.DocTestParser* のメソッド), 1941
get_endidx() (*readline* モジュール), 188
get_environ() (*wsgiref.simple_server.WSGIRequestHandler* のメソッド), 1555
get_errno() (*ctypes* モジュール), 966
get_event_loop() (*asyncio* モジュール), 1142
get_event_loop() (*asyncio.AbstractEventLoopPolicy* のメソッド), 1191
get_event_loop_policy() (*asyncio* モジュール), 1191
get_examples() (*doctest.DocTestParser* のメソッド), 1941
get_exception_handler() (*asyncio.loop* のメソッド), 1160
get_exec_path() (*os* モジュール), 708
get_extra_info() (*asyncio.BaseTransport* のメソッド), 1176
get_extra_info() (*asyncio.StreamWriter* のメソッド), 1122
get_field() (*string.Formatter* のメソッド), 127
get_file() (*mailbox.Babyl* のメソッド), 1426
get_file() (*mailbox.Mailbox* のメソッド), 1419
get_file() (*mailbox.Maildir* のメソッド), 1422
get_file() (*mailbox.mbox* のメソッド), 1423
get_file() (*mailbox.MH* のメソッド), 1425
get_file() (*mailbox.MMDF* のメソッド), 1426
get_file_breaks() (*bdb.Bdb* のメソッド), 2097
get_filename() (*email.message.EmailMessage* のメソッド), 1334
get_filename() (*email.message.Message* のメソッド), 1384
get_filename() (*importlib.abc.ExecutionLoader* のメソッド), 2299
get_filename() (*importlib.abc.FileLoader* のメソッド), 2300
get_filename() (*importlib.machinery.ExtensionFileLoader* のメソッド), 2308
get_filename() (*zipimport.zipimporter* のメソッド), 2280
get_flags() (*mailbox.MaildirMessage* のメソッド), 1428
get_flags() (*mailbox.mboxMessage* のメソッド), 1430
get_flags() (*mailbox.MMDFMessage* のメソッド), 1435
get_folder() (*mailbox.Maildir* のメソッド), 1421
get_folder() (*mailbox.MH* のメソッド), 1424
get_frees() (*symtable.Function* のメソッド), 2336
get_freeze_count() (*gc* モジュール), 2247
get_from() (*mailbox.mboxMessage* のメソッド), 1430
get_from() (*mailbox.MMDFMessage* のメソッド), 1435
get_full_url() (*urllib.request.Request* のメソッド), 1571
get_globals() (*symtable.Function* のメソッド), 2336
get_grouped_opcodes() (*difflib.SequenceMatcher* のメソッド), 171
get_handle_inheritable() (*os* モジュール), 726
get_header() (*urllib.request.Request* のメソッド), 1571
get_history_item() (*readline* モジュール), 187
get_history_length() (*readline* モジュール), 186
get_id() (*symtable.SymbolTable* のメソッド), 2335
get_ident() (*_thread* モジュール), 1097
get_ident() (*threading* モジュール), 978
get_identifiers() (*symtable.SymbolTable* のメソッド), 2335
get_importer() (*pkgutil* モジュール), 2283
get_info() (*mailbox.MaildirMessage* のメソッド), 1429
get_inheritable() (*os* モジュール), 726
get_inheritable() (*socket.socket* のメソッド), 1229
get_instructions() (*dis* モジュール), 2360
get_int_max_str_digits() (*sys* モジュール), 2174
get_interpreter() (*zipapp* モジュール), 2159
GET_ITER (*opcode*), 2363
get_key() (*selectors.BaseSelector* のメソッド), 1300
get_labels() (*mailbox.Babyl* のメソッド), 1425
get_labels() (*mailbox.BabylMessage* のメソッド), 1433
get_last_error() (*ctypes* モジュール), 966
get_line_buffer() (*readline* モジュール), 186
get_lineno() (*symtable.SymbolTable* のメソッド), 2335
get_loader() (*pkgutil* モジュール), 2283
get_locals() (*symtable.Function* のメソッド), 2336
get_logger() (*multiprocessing* モジュール), 1036
get_loop() (*asyncio.Future* のメソッド), 1172
get_loop() (*asyncio.Server* のメソッド), 1165

get_magic() (*imp* モジュール), 2467
get_makefile_filename() (*sysconfig* モジュール), 2193
get_map() (*selectors.BaseSelector* のメソッド), 1300
get_matching_blocks() (*difflib.SequenceMatcher* のメソッド), 170
get_message() (*mailbox.Mailbox* のメソッド), 1419
get_method() (*urllib.request.Request* のメソッド), 1571
get_methods() (*symtable.Class* のメソッド), 2336
get_mixed_type_key() (*ipaddress* モジュール), 1726
get_name() (*asyncio.Task* のメソッド), 1117
get_name() (*symtable.Symbol* のメソッド), 2336
get_name() (*symtable.SymbolTable* のメソッド), 2335
get_namespace() (*symtable.Symbol* のメソッド), 2337
get_namespaces() (*symtable.Symbol* のメソッド), 2337
get_native_id() (*_thread* モジュール), 1097
get_native_id() (*threading* モジュール), 978
get_nonlocals() (*symtable.Function* のメソッド), 2336
get_nonstandard_attr() (*http.cookiejar.Cookie* のメソッド), 1693
get_nowait() (*asyncio.Queue* のメソッド), 1138
get_nowait() (*multiprocessing.Queue* のメソッド), 1008
get_nowait() (*queue.Queue* のメソッド), 1089
get_nowait() (*queue.SimpleQueue* のメソッド), 1091
get_object_traceback() (*tracemalloc* モジュール), 2134
get_objects() (*gc* モジュール), 2245
get_opcodes() (*difflib.SequenceMatcher* のメソッド), 170
get_option() (*optparse.OptionParser* のメソッド), 2456
get_option_group() (*optparse.OptionParser* のメソッド), 2444
get_origin() (*typing* モジュール), 1911
get_original_stdout() (*test.support* モジュール), 2074
get_osfhandle() (*msvcrt* モジュール), 2393
get_output_charset() (*email.charset.Charset* のメソッド), 1395
get_param() (*email.message.Message* のメソッド), 1383
get_parameters() (*symtable.Function* のメソッド), 2336
get_params() (*email.message.Message* のメソッド), 1383
get_path() (*sysconfig* モジュール), 2191
get_path_names() (*sysconfig* モジュール), 2191
get_paths() (*sysconfig* モジュール), 2192
get_payload() (*email.message.Message* のメソッド), 1378
get_pid() (*asyncio.SubprocessTransport* のメソッド), 1179
get_pipe_transport() (*asyncio.SubprocessTransport* のメソッド), 1179
get_platform() (*sysconfig* モジュール), 2192
get_poly() (*turtle* モジュール), 1799
get_position() (*xdrlib.Unpacker* のメソッド), 681
get_protocol() (*asyncio.BaseTransport* のメソッド), 1177
get_python_version() (*sysconfig* モジュール), 2192
get_recsrc() (*ossaudiodev.oss_mixer_device* のメソッド), 1751
get_referents() (*gc* モジュール), 2246
get_referrers() (*gc* モジュール), 2246
get_request() (*socketserver.BaseServer* のメソッド), 1665
get_returncode() (*asyncio.SubprocessTransport* のメソッド), 1179
get_running_loop() (*asyncio* モジュール), 1142
get_scheme() (*wsgiref.handlers.BaseHandler* のメソッド), 1559
get_scheme_names() (*sysconfig* モジュール), 2191
get_sequences() (*mailbox.MH* のメソッド), 1424
get_sequences() (*mailbox.MHMessage* のメソッド), 1432
get_server() (*multiprocessing.managers.BaseManager* のメソッド), 1021
get_server_certificate() (*ssl* モジュール), 1248
get_shapepoly() (*turtle* モジュール), 1797
get_socket() (*telnetlib.Telnet* のメソッド), 1654
get_source() (*importlib.abc.InspectLoader* のメソッド), 2298
get_source() (*importlib.abc.SourceLoader* のメソッド), 2301
get_source() (*importlib.machinery.ExtensionFileLoader* のメソッド), 2308
get_source() (*importlib.machinery.SourcelessFileLoader* のメソッド), 2307
get_source() (*zipimport.zipimporter* のメソッド), 2280
get_source_segment() (*ast* モジュール), 2332
get_stack() (*asyncio.Task* のメソッド), 1116
get_stack() (*bdb.Bdb* のメソッド), 2097
get_start_method() (*multiprocessing* モジュール), 1010
get_starttag_text() (*html.parser.HTMLParser* のメソッド), 1455
get_stats() (*gc* モジュール), 2245
get_stderr() (*wsgiref.handlers.BaseHandler* のメソッド), 1558
get_stderr() (*wsgiref.simple_server.WSGIRequestHandler* のメソッド), 1555
get_stdin() (*wsgiref.handlers.BaseHandler* のメソッド), 1558
get_string() (*mailbox.Mailbox* のメソッド), 1419
get_subdir() (*mailbox.MaildirMessage* のメソッド), 1428
get_suffixes() (*imp* モジュール), 2467
get_symbols() (*symtable.SymbolTable* のメソッド), 2335
get_tag() (*imp* モジュール), 2470
get_task_factory() (*asyncio.loop* のメソッド), 1148
get_terminal_size() (*os* モジュール), 725
get_terminal_size() (*shutil* モジュール), 532
get_terminator() (*asynchat.async_chat* のメソッド), 1308
get_threshold() (*gc* モジュール), 2246
get_token() (*shlex.shlex* のメソッド), 1824
get_traceback_limit() (*tracemalloc* モジュール), 2134
get_traced_memory() (*tracemalloc* モジュール), 2134
get_tracemalloc_memory() (*tracemalloc* モジュール), 2135
get_type() (*symtable.SymbolTable* のメソッド), 2335
get_type_hints() (*typing* モジュール), 1911
get_unixfrom() (*email.message.EmailMessage* のメソッド), 1331
get_unixfrom() (*email.message.Message* のメソッド), 1378
get_unpack_formats() (*shutil* モジュール), 530
get_usage() (*optparse.OptionParser* のメソッド), 2458
get_value() (*string.Formatter* のメソッド), 127
get_version() (*optparse.OptionParser* のメソッド), 2444
get_visible() (*mailbox.BabylMessage* のメソッド), 1433
get_wch() (*curses.window* のメソッド), 906
get_write_buffer_limits() (*asyncio.WriteTransport* のメソッド), 1178
get_write_buffer_size() (*asyncio.WriteTransport* のメソッド), 1178
GET_YIELD_FROM_ITER (*opcode*), 2363
getacl() (*imaplib.IMAP4* のメソッド), 1626
getaddresses() (*email.utils* モジュール), 1399
getaddrinfo() (*asyncio.loop* のメソッド), 1157
getaddrinfo() (*socket* モジュール), 1222
getallocatedblocks() (*sys* モジュール), 2173
getandroidapilevel() (*sys* モジュール), 2173
getannotation() (*imaplib.IMAP4* のメソッド), 1626
getargspec() (*inspect* モジュール), 2260
getargvalues() (*inspect* モジュール), 2261
getatime() (*os.path* モジュール), 494
getattr() (**組み込み関数**), 15
getattr_static() (*inspect* モジュール), 2265
getAttribute() (*xml.dom.Element* のメソッド), 1494
getAttributeNode() (*xml.dom.Element* のメソッド), 1494
getAttributeNodeNS() (*xml.dom.Element* のメソッド), 1495
getAttributeNS() (*xml.dom.Element* のメソッド), 1495
GetBase() (*xml.parsers.expat.xmlparser* のメソッド), 1525
getbegyx() (*curses.window* のメソッド), 906
getbkgd() (*curses.window* のメソッド), 906
getblocking() (*socket.socket* のメソッド), 1230
getboolean() (*configparser.ConfigParser* のメソッド), 675
getbuffer() (*io.BytesIO* のメソッド), 783
getByteStream() (*xml.sax.xmlreader.InputSource* のメソッド), 1522
getcallargs() (*inspect* モジュール), 2262

getcanvas() (*turtle* モジュール), 1807
getcapabilities() (*nntplib.NNTP* のメソッド), 1634
getcaps() (*mailcap* モジュール), 1416
getch() (*curses.window* のメソッド), 906
getch() (*msvcrt* モジュール), 2394
getCharacterStream() (*xml.sax.xmlreader.InputSource* のメソッド), 1522
getche() (*msvcrt* モジュール), 2394
getcheckinterval() (*sys* モジュール), 2173
getChild() (*logging.Logger* のメソッド), 845
getchildren() (*xml.etree.ElementTree.Element* のメソッド), 1478
getclasstree() (*inspect* モジュール), 2260
getclosurevars() (*inspect* モジュール), 2263
GetColumnInfo() (*msilib.View* のメソッド), 2387
getColumnNumber() (*xml.sax.xmlreader.Locator* のメソッド), 1521
getcomments() (*inspect* モジュール), 2254
getcompname() (*aifc.aifc* のメソッド), 1732
getcompname() (*sunau.AU_read* のメソッド), 1736
getcompname() (*wave.Wave_read* のメソッド), 1739
getcomptype() (*aifc.aifc* のメソッド), 1732
getcomptype() (*sunau.AU_read* のメソッド), 1736
getcomptype() (*wave.Wave_read* のメソッド), 1739
getContentHandler() (*xml.sax.xmlreader.XMLReader* のメソッド), 1519
getcontext() (*decimal* モジュール), 391
getcoroutinelocals() (*inspect* モジュール), 2267
getcoroutinestate() (*inspect* モジュール), 2266
getctime() (*os.path* モジュール), 494
getcwd() (*os* モジュール), 731
getcwdb() (*os* モジュール), 731
getcwdu (*2to3 fixer*), 2062
getdecoder() (*codecs* モジュール), 201
getdefaultencoding() (*sys* モジュール), 2173
getdefaultlocale() (*locale* モジュール), 1769
getdefaulttimeout() (*socket* モジュール), 1226
getdlopenflags() (*sys* モジュール), 2173
getdoc() (*inspect* モジュール), 2254
getDOMImplementation() (*xml.dom* モジュール), 1488
getDTDHandler() (*xml.sax.xmlreader.XMLReader* のメソッド), 1519
getEffectiveLevel() (*logging.Logger* のメソッド), 845
getegid() (*os* モジュール), 709
getElementsByTagName() (*xml.dom.Document* のメソッド), 1494
getElementsByTagName() (*xml.dom.Element* のメソッド), 1494
getElementsByTagNameNS() (*xml.dom.Document* のメソッド), 1494
getElementsByTagNameNS() (*xml.dom.Element* のメソッド), 1494
getencoder() (*codecs* モジュール), 201
getEncoding() (*xml.sax.xmlreader.InputSource* のメソッド), 1522
getEntityResolver() (*xml.sax.xmlreader.XMLReader* のメソッド), 1520
getenv() (*os* モジュール), 708
getenvb() (*os* モジュール), 708
getErrorHandler() (*xml.sax.xmlreader.XMLReader* のメソッド), 1520
geteuid() (*os* モジュール), 709
getEvent() (*xml.dom.pulldom.DOMEventStream* のメソッド), 1507
getEventCategory() (*logging.handlers.NTEventLogHandler* のメソッド), 888
getEventType() (*logging.handlers.NTEventLogHandler* のメソッド), 888
getException() (*xml.sax.SAXException* のメソッド), 1510
getFeature() (*xml.sax.xmlreader.XMLReader* のメソッド), 1520
GetFieldCount() (*msilib.Record* のメソッド), 2388
getfile() (*inspect* モジュール), 2254
getfilesystemencodeerrors() (*sys* モジュール), 2174
getfilesystemencoding() (*sys* モジュール), 2174
getfirst() (*cgi.FieldStorage* のメソッド), 1544
getfloat() (*configparser.ConfigParser* のメソッド), 674
getfmts() (*ossaudiodev.oss_audio_device* のメソッド), 1748
getfqdn() (*socket* モジュール), 1222
getframeinfo() (*inspect* モジュール), 2264
getframerate() (*aifc.aifc* のメソッド), 1732
getframerate() (*sunau.AU_read* のメソッド), 1736
getframerate() (*wave.Wave_read* のメソッド), 1739
getfullargspec() (*inspect* モジュール), 2260
getgeneratorlocals() (*inspect* モジュール), 2266
getgeneratorstate() (*inspect* モジュール), 2266
getgid() (*os* モジュール), 709
getgrall() (*grp* モジュール), 2413
getgrgid() (*grp* モジュール), 2413
getgrnam() (*grp* モジュール), 2413
getgrouplist() (*os* モジュール), 709
getgroups() (*os* モジュール), 709
getheader() (*http.client.HTTPResponse* のメソッド), 1610
getheaders() (*http.client.HTTPResponse* のメソッド), 1610
gethostbyaddr() (*in module socket*), 713
gethostbyaddr() (*socket* モジュール), 1223
gethostbyname() (*socket* モジュール), 1223
gethostbyname_ex() (*socket* モジュール), 1223
gethostname() (*in module socket*), 713
gethostname() (*socket* モジュール), 1223
getincrementaldecoder() (*codecs* モジュール), 201
getincrementalencoder() (*codecs* モジュール), 201
getinfo() (*zipfile.ZipFile* のメソッド), 619
getinnerframes() (*inspect* モジュール), 2264
GetInputContext() (*xml.parsers.expat.xmlparser* のメソッド), 1525
getint() (*configparser.ConfigParser* のメソッド), 674
GetInteger() (*msilib.Record* のメソッド), 2388
getitem() (*operator* モジュール), 464
getiterator() (*xml.etree.ElementTree.Element* のメソッド), 1478
getiterator() (*xml.etree.ElementTree.ElementTree* のメソッド), 1480
getitimer() (*signal* モジュール), 1316
getkey() (*curses.window* のメソッド), 906
GetLastError() (*ctypes* モジュール), 966
getLength() (*xml.sax.xmlreader.Attributes* のメソッド), 1522
getLevelName() (*logging* モジュール), 859
getline() (*linecache* モジュール), 518
getLineNumber() (*xml.sax.xmlreader.Locator* のメソッド), 1521
getlist() (*cgi.FieldStorage* のメソッド), 1544
getloadavg() (*os* モジュール), 770
getlocale() (*locale* モジュール), 1769
getLogger() (*logging* モジュール), 856
getLoggerClass() (*logging* モジュール), 857
getlogin() (*os* モジュール), 709
getLogRecordFactory() (*logging* モジュール), 857
getmark() (*aifc.aifc* のメソッド), 1732
getmark() (*sunau.AU_read* のメソッド), 1737
getmark() (*wave.Wave_read* のメソッド), 1739
getmarkers() (*aifc.aifc* のメソッド), 1732
getmarkers() (*sunau.AU_read* のメソッド), 1737
getmarkers() (*wave.Wave_read* のメソッド), 1739
getmaxyx() (*curses.window* のメソッド), 906
getmember() (*tarfile.TarFile* のメソッド), 633
getmembers() (*inspect* モジュール), 2251
getmembers() (*tarfile.TarFile* のメソッド), 633
getMessage() (*logging.LogRecord* のメソッド), 854
getMessage() (*xml.sax.SAXException* のメソッド), 1510
getMessageID() (*logging.handlers.NTEventLogHandler* のメソッド), 888

getmodule() (*inspect* モジュール), 2254
getmodulename() (*inspect* モジュール), 2251
getmouse() (*curses* モジュール), 897
getmro() (*inspect* モジュール), 2262
getmtime() (*os.path* モジュール), 494
getname() (*chunk.Chunk* のメソッド), 1742
getName() (*threading.Thread* のメソッド), 982
getNameByQName() (*xml.sax.xmlreader.AttributesNS* のメソッド), 1523
getnameinfo() (*asyncio.loop* のメソッド), 1157
getnameinfo() (*socket* モジュール), 1223
getnames() (*tarfile.TarFile* のメソッド), 633
getNames() (*xml.sax.xmlreader.Attributes* のメソッド), 1522
getnchannels() (*aifc.aifc* のメソッド), 1732
getnchannels() (*sunau.AU_read* のメソッド), 1736
getnchannels() (*wave.Wave_read* のメソッド), 1739
getnframes() (*aifc.aifc* のメソッド), 1732
getnframes() (*sunau.AU_read* のメソッド), 1736
getnframes() (*wave.Wave_read* のメソッド), 1739
getnode, 1658
getnode() (*uuid* モジュール), 1658
getopt (モジュール), 840
getopt() (*getopt* モジュール), 840
GetoptError, 841
getouterframes() (*inspect* モジュール), 2264
getoutput() (*subprocess* モジュール), 1084
getpagesize() (*resource* モジュール), 2429
getparams() (*aifc.aifc* のメソッド), 1732
getparams() (*sunau.AU_read* のメソッド), 1736
getparams() (*wave.Wave_read* のメソッド), 1739
getparyx() (*curses.window* のメソッド), 906
getpass (モジュール), 893
getpass() (*getpass* モジュール), 893
GetPassWarning, 893
getpeercert() (*ssl.SSLSocket* のメソッド), 1261
getpeername() (*socket.socket* のメソッド), 1229
getpen() (*turtle* モジュール), 1799
getpgid() (*os* モジュール), 710
getpgrp() (*os* モジュール), 710
getpid() (*os* モジュール), 710
getpos() (*html.parser.HTMLParser* のメソッド), 1455
getppid() (*os* モジュール), 710
getpreferredencoding() (*locale* モジュール), 1769
getpriority() (*os* モジュール), 710
getprofile() (*sys* モジュール), 2175
GetProperty() (*msilib.SummaryInformation* のメソッド), 2388
getProperty() (*xml.sax.xmlreader.XMLReader* のメソッド), 1520
GetPropertyCount() (*msilib.SummaryInformation* のメソッド), 2388
getprotobyname() (*socket* モジュール), 1224
getproxies() (*urllib.request* モジュール), 1565
getPublicId() (*xml.sax.xmlreader.InputSource* のメソッド), 1521
getPublicId() (*xml.sax.xmlreader.Locator* のメソッド), 1521
getpwall() (*pwd* モジュール), 2411
getpwnam() (*pwd* モジュール), 2411
getpwuid() (*pwd* モジュール), 2411
getQNameByName() (*xml.sax.xmlreader.AttributesNS* のメソッド), 1523
getQNames() (*xml.sax.xmlreader.AttributesNS* のメソッド), 1523
getquota() (*imaplib.IMAP4* のメソッド), 1626
getquotaroot() (*imaplib.IMAP4* のメソッド), 1626
getrandbits() (*random* モジュール), 414
getrandom() (*os* モジュール), 772
getreader() (*codecs* モジュール), 202
getrecursionlimit() (*sys* モジュール), 2174
getrefcount() (*sys* モジュール), 2174
GetReparseDeferralEnabled() (*xml.parsers.expat.xmlparser* のメソッド), 1526
getresgid() (*os* モジュール), 711
getresponse() (*http.client.HTTPConnection* のメソッド), 1608
getresuid() (*os* モジュール), 710
getrlimit() (*resource* モジュール), 2425
getroot() (*xml.etree.ElementTree.ElementTree* のメソッド), 1480
getrusage() (*resource* モジュール), 2428
getsample() (*audioop* モジュール), 1728
getsampwidth() (*aifc.aifc* のメソッド), 1732
getsampwidth() (*sunau.AU_read* のメソッド), 1736
getsampwidth() (*wave.Wave_read* のメソッド), 1739
getscreen() (*turtle* モジュール), 1800
getservbyname() (*socket* モジュール), 1224
getservbyport() (*socket* モジュール), 1224
GetSetDescriptorType (*types* モジュール), 325
getshapes() (*turtle* モジュール), 1807
getsid() (*os* モジュール), 713
getsignal() (*signal* モジュール), 1314
getsitepackages() (*site* モジュール), 2271
getsize() (*chunk.Chunk* のメソッド), 1742
getsize() (*os.path* モジュール), 494
getsizeof() (*sys* モジュール), 2175
getsockname() (*socket.socket* のメソッド), 1230
getsockopt() (*socket.socket* のメソッド), 1230
getsource() (*inspect* モジュール), 2254
getsourcefile() (*inspect* モジュール), 2254
getsourcelines() (*inspect* モジュール), 2254
getspall() (*spwd* モジュール), 2412
getspnam() (*spwd* モジュール), 2412
getstate() (*codecs.IncrementalDecoder* のメソッド), 209
getstate() (*codecs.IncrementalEncoder* のメソッド), 208
getstate() (*random* モジュール), 414
getstatusoutput() (*subprocess* モジュール), 1084
getstr() (*curses.window* のメソッド), 907
GetString() (*msilib.Record* のメソッド), 2388
getSubject() (*logging.handlers.SMTPHandler* のメソッド), 889
GetSummaryInformation() (*msilib.Database* のメソッド), 2387
getswitchinterval() (*sys* モジュール), 2175
getSystemId() (*xml.sax.xmlreader.InputSource* のメソッド), 1521
getSystemId() (*xml.sax.xmlreader.Locator* のメソッド), 1521
getsyx() (*curses* モジュール), 897
gettarinfo() (*tarfile.TarFile* のメソッド), 636
gettempdir() (*tempfile* モジュール), 513
gettempdirb() (*tempfile* モジュール), 513
gettempprefix() (*tempfile* モジュール), 513
gettempprefixb() (*tempfile* モジュール), 513
getTestCaseNames() (*unittest.TestLoader* のメソッド), 1976
gettext (モジュール), 1753
gettext() (*gettext* モジュール), 1754
gettext() (*gettext.GNUTranslations* のメソッド), 1759
gettext() (*gettext.NullTranslations* のメソッド), 1757
gettext() (*locale* モジュール), 1773
gettimeout() (*socket.socket* のメソッド), 1230
gettrace() (*sys* モジュール), 2175
getturtle() (*turtle* モジュール), 1799
getType() (*xml.sax.xmlreader.Attributes* のメソッド), 1522
getuid() (*os* モジュール), 711
geturl() (*urllib.parse.urllib.parse.SplitResult* のメソッド), 1595
getuser() (*getpass* モジュール), 894
getuserbase() (*site* モジュール), 2271
getusersitepackages() (*site* モジュール), 2272
getvalue() (*io.BytesIO* のメソッド), 783
getvalue() (*io.StringIO* のメソッド), 788

getValue() (*xml.sax.xmlreader.Attributes* のメソッド), 1522
getValueByQName() (*xml.sax.xmlreader.AttributesNS* のメソッド), 1523
getwch() (*msvcrt* モジュール), 2394
getwche() (*msvcrt* モジュール), 2394
getweakrefcount() (*weakref* モジュール), 314
getweakrefs() (*weakref* モジュール), 314
getwelcome() (*ftplib.FTP* のメソッド), 1615
getwelcome() (*nntplib.NNTP* のメソッド), 1634
getwelcome() (*poplib.POP3* のメソッド), 1620
getwin() (*curses* モジュール), 897
getwindowsversion() (*sys* モジュール), 2175
getwriter() (*codecs* モジュール), 202
getxattr() (*os* モジュール), 753
getyx() (*curses.window* のメソッド), 907
gid (*tarfile.TarInfo* の属性), 638
GIL, **2483**
glob
    モジュール, 517
glob (モジュール), 515
glob() (*glob* モジュール), 516
glob() (*msilib.Directory* のメソッド), 2390
glob() (*pathlib.Path* のメソッド), 484
global interpreter lock, **2483**
globals() (組み込み関数), 15
globs (*doctest.DocTest* の属性), 1938
gmtime() (*time* モジュール), 794
gname (*tarfile.TarInfo* の属性), 638
GNOME, 1761
GNU_FORMAT (*tarfile* モジュール), 631
gnu_getopt() (*getopt* モジュール), 841
GNUTranslations (*gettext* のクラス), 1759
got (*doctest.DocTestFailure* の属性), 1946
goto() (*turtle* モジュール), 1782
Graphical User Interface, 1831
GREATER (*token* モジュール), 2339
GREATEREQUAL (*token* モジュール), 2340
Greenwich Mean Time, 791
GRND_NONBLOCK (*os* モジュール), 773
GRND_RANDOM (*os* モジュール), 773
Group (*email.headerregistry* のクラス), 1365
group() (*nntplib.NNTP* のメソッド), 1636
group() (*pathlib.Path* のメソッド), 484
group() (*re.Match* のメソッド), 154
groupby() (*itertools* モジュール), 442
groupdict() (*re.Match* のメソッド), 155
groupindex (*re.Pattern* の属性), 153
groups (*email.headerregistry.AddressHeader* の属性), 1361
groups (*re.Pattern* の属性), 153
groups() (*re.Match* のメソッド), 155
grp (モジュール), 2413
gt() (*operator* モジュール), 462
guess_all_extensions() (*mimetypes* モジュール), 1439
guess_all_extensions() (*mimetypes.MimeTypes* のメソッド), 1441
guess_extension() (*mimetypes* モジュール), 1439
guess_extension() (*mimetypes.MimeTypes* のメソッド), 1441
guess_scheme() (*wsgiref.util* モジュール), 1550
guess_type() (*mimetypes* モジュール), 1439
guess_type() (*mimetypes.MimeTypes* のメソッド), 1441
GUI, 1831
gzip (モジュール), 600
gzip command line option
    --best, 603
    -d, 603
    --decompress, 603
    --fast, 603
    file, 603
    -h, 603
    --help, 603
GzipFile (*gzip* のクラス), 601

H

-h
    gzip command line option, 603
    json.tool command line option, 1415
    timeit command line option, 2124
    tokenize command line option, 2345
    zipapp command line option, 2158
halfdelay() (*curses* モジュール), 897
Handle (*asyncio* のクラス), 1164
handle() (*http.server.BaseHTTPRequestHandler* のメソッド), 1673
handle() (*logging.Handler* のメソッド), 850
handle() (*logging.handlers.QueueListener* のメソッド), 892
handle() (*logging.Logger* のメソッド), 848
handle() (*logging.NullHandler* のメソッド), 878
handle() (*socketserver.BaseRequestHandler* のメソッド), 1666
handle() (*wsgiref.simple_server.WSGIRequestHandler* のメソッド), 1555
handle_accept() (*asyncore.dispatcher* のメソッド), 1303
handle_accepted() (*asyncore.dispatcher* のメソッド), 1304
handle_charref() (*html.parser.HTMLParser* のメソッド), 1456
handle_close() (*asyncore.dispatcher* のメソッド), 1303
handle_comment() (*html.parser.HTMLParser* のメソッド), 1456
handle_connect() (*asyncore.dispatcher* のメソッド), 1303
handle_data() (*html.parser.HTMLParser* のメソッド), 1456
handle_decl() (*html.parser.HTMLParser* のメソッド), 1456
handle_defect() (*email.policy.Policy* のメソッド), 1351
handle_endtag() (*html.parser.HTMLParser* のメソッド), 1456
handle_entityref() (*html.parser.HTMLParser* のメソッド), 1456
handle_error() (*asyncore.dispatcher* のメソッド), 1303
handle_error() (*socketserver.BaseServer* のメソッド), 1665
handle_expect_100() (*http.server.BaseHTTPRequestHandler* のメソッド), 1673
handle_expt() (*asyncore.dispatcher* のメソッド), 1303
handle_one_request() (*http.server.BaseHTTPRequestHandler* のメソッド), 1673
handle_pi() (*html.parser.HTMLParser* のメソッド), 1456
handle_read() (*asyncore.dispatcher* のメソッド), 1303
handle_request() (*socketserver.BaseServer* のメソッド), 1663
handle_request() (*xmlrpc.server.CGIXMLRPCRequestHandler* のメソッド), 1709
handle_startendtag() (*html.parser.HTMLParser* のメソッド), 1456
handle_starttag() (*html.parser.HTMLParser* のメソッド), 1455
handle_timeout() (*socketserver.BaseServer* のメソッド), 1665
handle_write() (*asyncore.dispatcher* のメソッド), 1303
handleError() (*logging.Handler* のメソッド), 850
handleError() (*logging.handlers.SocketHandler* のメソッド), 883
Handler (*logging* のクラス), 849
handler() (*cgitb* モジュール), 1549
harmonic_mean() (*statistics* モジュール), 423
HAS_ALPN (*ssl* モジュール), 1256
has_children() (*symtable.SymbolTable* のメソッド), 2335
has_colors() (*curses* モジュール), 897
has_dualstack_ipv6() (*socket* モジュール), 1221
HAS_ECDH (*ssl* モジュール), 1256
has_exec() (*symtable.SymbolTable* のメソッド), 2335
has_extn() (*smtplib.SMTP* のメソッド), 1643

has_header() (*csv.Sniffer* のメソッド), 653
has_header() (*urllib.request.Request* のメソッド), 1571
has_ic() (*curses* モジュール), 897
has_il() (*curses* モジュール), 897
has_ipv6 (*socket* モジュール), 1218
has_key (*2to3 fixer*), 2062
has_key() (*curses* モジュール), 897
has_location (*importlib.machinery.ModuleSpec* の属性), 2309
HAS_NEVER_CHECK_COMMON_NAME (*ssl* モジュール), 1256
has_nonstandard_attr() (*http.cookiejar.Cookie* のメソッド), 1693
HAS_NPN (*ssl* モジュール), 1256
has_option() (*configparser.ConfigParser* のメソッド), 673
has_option() (*optparse.OptionParser* のメソッド), 2456
has_section() (*configparser.ConfigParser* のメソッド), 673
HAS_SNI (*ssl* モジュール), 1256
HAS_SSLv2 (*ssl* モジュール), 1256
HAS_SSLv3 (*ssl* モジュール), 1257
has_ticket (*ssl.SSLSession* の属性), 1285
HAS_TLSv1 (*ssl* モジュール), 1257
HAS_TLSv1_1 (*ssl* モジュール), 1257
HAS_TLSv1_2 (*ssl* モジュール), 1257
HAS_TLSv1_3 (*ssl* モジュール), 1257
hasattr() (組み込み関数), 15
hasAttribute() (*xml.dom.Element* のメソッド), 1494
hasAttributeNS() (*xml.dom.Element* のメソッド), 1494
hasAttributes() (*xml.dom.Node* のメソッド), 1491
hasChildNodes() (*xml.dom.Node* のメソッド), 1491
hascompare (*dis* モジュール), 2375
hasconst (*dis* モジュール), 2374
hasFeature() (*xml.dom.DOMImplementation* のメソッド), 1489
hasfree (*dis* モジュール), 2375
hash
    組み込み関数, 50
hash() (組み込み関数), 15
hash-based pyc, **2484**
hash_info (*sys* モジュール), 2177
hashable, **2484**
Hashable (*collections.abc* のクラス), 298
Hashable (*typing* のクラス), 1904
hasHandlers() (*logging.Logger* のメソッド), 848
hash.block_size (*hashlib* モジュール), 689
hash.digest_size (*hashlib* モジュール), 689
hashlib (モジュール), 687
hasjabs (*dis* モジュール), 2375
hasjrel (*dis* モジュール), 2375
haslocal (*dis* モジュール), 2375
hasname (*dis* モジュール), 2375
HAVE_ARGUMENT (*opcode*), 2374
HAVE_CONTEXTVAR (*decimal* モジュール), 399
HAVE_DOCSTRINGS (*test.support* モジュール), 2070
HAVE_THREADS (*decimal* モジュール), 399
HCI_DATA_DIR (*socket* モジュール), 1218
HCI_FILTER (*socket* モジュール), 1218
HCI_TIME_STAMP (*socket* モジュール), 1218
head() (*nntplib.NNTP* のメソッド), 1638
Header (*email.header* のクラス), 1391
header_encode() (*email.charset.Charset* のメソッド), 1395
header_encode_lines() (*email.charset.Charset* のメソッド), 1395
header_encoding (*email.charset.Charset* の属性), 1394
header_factory (*email.policy.EmailPolicy* の属性), 1353
header_fetch_parse() (*email.policy.Compat32* のメソッド), 1356
header_fetch_parse() (*email.policy.EmailPolicy* のメソッド), 1354
header_fetch_parse() (*email.policy.Policy* のメソッド), 1352
header_items() (*urllib.request.Request* のメソッド), 1571
header_max_count() (*email.policy.EmailPolicy* のメソッド), 1354
header_max_count() (*email.policy.Policy* のメソッド), 1351
header_offset (*zipfile.ZipInfo* の属性), 626
header_source_parse() (*email.policy.Compat32* のメソッド), 1356
header_source_parse() (*email.policy.EmailPolicy* のメソッド), 1354
header_source_parse() (*email.policy.Policy* のメソッド), 1352
header_store_parse() (*email.policy.Compat32* のメソッド), 1356
header_store_parse() (*email.policy.EmailPolicy* のメソッド), 1354
header_store_parse() (*email.policy.Policy* のメソッド), 1352
HeaderError, 631
HeaderParseError, 1357
HeaderParser (*email.parser* のクラス), 1342
HeaderRegistry (*email.headerregistry* のクラス), 1363
headers
    MIME, 1439, 1540
headers (*http.server.BaseHTTPRequestHandler* の属性), 1672
headers (*urllib.error.HTTPError* の属性), 1599
Headers (*wsgiref.headers* のクラス), 1553
headers (*xmlrpc.client.ProtocolError* の属性), 1701
heading() (*tkinter.ttk.Treeview* のメソッド), 1861
heading() (*turtle* モジュール), 1787
heapify() (*heapq* モジュール), 302
heapmin() (*msvcrt* モジュール), 2394
heappop() (*heapq* モジュール), 301
heappush() (*heapq* モジュール), 301
heappushpop() (*heapq* モジュール), 302
heapq (モジュール), 301
heapreplace() (*heapq* モジュール), 302
helo() (*smtplib.SMTP* のメソッド), 1643
help
    online, 1917
--help
    gzip command line option, 603
    json.tool command line option, 1415
    timeit command line option, 2124
    tokenize command line option, 2345
    trace command line option, 2126
    zipapp command line option, 2158
help (*optparse.Option* の属性), 2450
help (*pdb command*), 2105
help() (*nntplib.NNTP* のメソッド), 1637
help() (組み込み関数), 15
herror, 1214
hex (*uuid.UUID* の属性), 1658
hex() (*bytearray* のメソッド), 70
hex() (*bytes* のメソッド), 69
hex() (*float* のメソッド), 44
hex() (*memoryview* のメソッド), 89
hex() (組み込み関数), 15
hexadecimal
    literals, 39
hexbin() (*binhex* モジュール), 1446
hexdigest() (*hashlib.hash* のメソッド), 690
hexdigest() (*hashlib.shake* のメソッド), 690
hexdigest() (*hmac.HMAC* のメソッド), 701
hexdigits (*string* モジュール), 125
hexlify() (*binascii* モジュール), 1449
hexversion (*sys* モジュール), 2177
hidden() (*curses.panel.Panel* のメソッド), 922
hide() (*curses.panel.Panel* のメソッド), 922
hide() (*tkinter.ttk.Notebook* のメソッド), 1854
hide_cookie2 (*http.cookiejar.CookiePolicy* の属性), 1689
hideturtle() (*turtle* モジュール), 1794
HierarchyRequestErr, 1497
HIGH_PRIORITY_CLASS (*subprocess* モジュール), 1077

HIGHEST_PROTOCOL (*pickle* モジュール), 538
HKEY_CLASSES_ROOT (*winreg* モジュール), 2403
HKEY_CURRENT_CONFIG (*winreg* モジュール), 2403
HKEY_CURRENT_USER (*winreg* モジュール), 2403
HKEY_DYN_DATA (*winreg* モジュール), 2403
HKEY_LOCAL_MACHINE (*winreg* モジュール), 2403
HKEY_PERFORMANCE_DATA (*winreg* モジュール), 2403
HKEY_USERS (*winreg* モジュール), 2403
hline() (*curses.window* のメソッド), 907
HList (*tkinter.tix* のクラス), 1872
hls_to_rgb() (*colorsys* モジュール), 1743
hmac (モジュール), 700
HOME, 493, 494
home() (*pathlib.Path* のクラスメソッド), 483
home() (*turtle* モジュール), 1783
HOMEDRIVE, 493
HOMEPATH, 493
hook_compressed() (*fileinput* モジュール), 500
hook_encoded() (*fileinput* モジュール), 500
host (*urllib.request.Request* の属性), 1570
hostmask (*ipaddress.IPv4Network* の属性), 1718
hostmask (*ipaddress.IPv6Network* の属性), 1722
hostname_checks_common_name (*ssl.SSLContext* の属性), 1274
hosts (*netrc.netrc* の属性), 679
hosts() (*ipaddress.IPv4Network* のメソッド), 1719
hosts() (*ipaddress.IPv6Network* のメソッド), 1722
hour (*datetime.datetime* の属性), 242
hour (*datetime.time* の属性), 252
HRESULT (*ctypes* のクラス), 972
hStdError (*subprocess.STARTUPINFO* の属性), 1075
hStdInput (*subprocess.STARTUPINFO* の属性), 1075
hStdOutput (*subprocess.STARTUPINFO* の属性), 1075
hsv_to_rgb() (*colorsys* モジュール), 1743
ht() (*turtle* モジュール), 1794
HTML, 1454, 1587
html (モジュール), 1453
html() (*cgitb* モジュール), 1549
html5 (*html.entities* モジュール), 1459
HTMLCalendar (*calendar* のクラス), 271
HtmlDiff (*difflib* のクラス), 164
html.entities (モジュール), 1459
HTMLParser (*html.parser* のクラス), 1454
html.parser (モジュール), 1454
htonl() (*socket* モジュール), 1224
htons() (*socket* モジュール), 1224
HTTP
    http (*standard module*), 1601
    http.client (*standard module*), 1604
    protocol, 1540, 1587, 1601, 1604, 1671
HTTP (*email.policy* モジュール), 1355
http (モジュール), 1601
http_error_301() (*urllib.request.HTTPRedirectHandler* のメソッド), 1575
http_error_302() (*urllib.request.HTTPRedirectHandler* のメソッド), 1575
http_error_303() (*urllib.request.HTTPRedirectHandler* のメソッド), 1575
http_error_307() (*urllib.request.HTTPRedirectHandler* のメソッド), 1576
http_error_401() (*urllib.request.HTTPBasicAuthHandler* のメソッド), 1578
http_error_401() (*urllib.request.HTTPDigestAuthHandler* のメソッド), 1578
http_error_407() (*urllib.request.ProxyBasicAuthHandler* のメソッド), 1578
http_error_407() (*urllib.request.ProxyDigestAuthHandler* のメソッド), 1578
http_error_auth_reqed() (*urllib.request.AbstractBasicAuthHandler* のメソッド), 1577
http_error_auth_reqed() (*urllib.request.AbstractDigestAuthHandler* のメソッド), 1578
http_error_default() (*urllib.request.BaseHandler* のメソッド), 1574
http_open() (*urllib.request.HTTPHandler* のメソッド), 1579
HTTP_PORT (*http.client* モジュール), 1607
http_proxy, 1564, 1582
http_response() (*urllib.request.HTTPErrorProcessor* のメソッド), 1580
http_version (*wsgiref.handlers.BaseHandler* の属性), 1561
HTTPBasicAuthHandler (*urllib.request* のクラス), 1568
http.client (モジュール), 1604
HTTPConnection (*http.client* のクラス), 1604
http.cookiejar (モジュール), 1683
HTTPCookieProcessor (*urllib.request* のクラス), 1567
http.cookies (モジュール), 1678
httpd, 1671
HTTPDefaultErrorHandler (*urllib.request* のクラス), 1567
HTTPDigestAuthHandler (*urllib.request* のクラス), 1569
HTTPError, 1599
HTTPErrorProcessor (*urllib.request* のクラス), 1570
HTTPException, 1606
HTTPHandler (*logging.handlers* のクラス), 890
HTTPHandler (*urllib.request* のクラス), 1569
HTTPPasswordMgr (*urllib.request* のクラス), 1568
HTTPPasswordMgrWithDefaultRealm (*urllib.request* のクラス), 1568
HTTPPasswordMgrWithPriorAuth (*urllib.request* のクラス), 1568
HTTPRedirectHandler (*urllib.request* のクラス), 1567
HTTPResponse (*http.client* のクラス), 1605
https_open() (*urllib.request.HTTPSHandler* のメソッド), 1579
HTTPS_PORT (*http.client* モジュール), 1607
https_response() (*urllib.request.HTTPErrorProcessor* のメソッド), 1580
HTTPSConnection (*http.client* のクラス), 1604
http.server
    security, 1678
HTTPServer (*http.server* のクラス), 1671
http.server (モジュール), 1671
HTTPSHandler (*urllib.request* のクラス), 1569
HTTPStatus (*http* のクラス), 1601
hypot() (*math* モジュール), 370

I

I (*re* モジュール), 147
-i list
    compileall command line option, 2353
I/O control
    buffering, 23, 1231
    POSIX, 2416
    tty, 2416
    UNIX, 2420
iadd() (*operator* モジュール), 468
iand() (*operator* モジュール), 468
iconcat() (*operator* モジュール), 468
id (*ssl.SSLSession* の属性), 1285
id() (*unittest.TestCase* のメソッド), 1970
id() (組み込み関数), 16
idcok() (*curses.window* のメソッド), 907
ident (*select.kevent* の属性), 1295
ident (*threading.Thread* の属性), 982
identchars (*cmd.Cmd* の属性), 1818
identify() (*tkinter.ttk.Notebook* のメソッド), 1854
identify() (*tkinter.ttk.Treeview* のメソッド), 1862
identify() (*tkinter.ttk.Widget* のメソッド), 1849
identify_column() (*tkinter.ttk.Treeview* のメソッド), 1862

identify_element() (tkinter.ttk.Treeview のメソッド), 1862
identify_region() (tkinter.ttk.Treeview のメソッド), 1862
identify_row() (tkinter.ttk.Treeview のメソッド), 1862
idioms (2to3 fixer), 2062
IDLE, 1875, **2484**
IDLE_PRIORITY_CLASS (subprocess モジュール), 1077
IDLESTARTUP, 1884
idlok() (curses.window のメソッド), 907
if
    文, 37
if_indextoname() (socket モジュール), 1227
if_nameindex() (socket モジュール), 1227
if_nametoindex() (socket モジュール), 1227
ifloordiv() (operator モジュール), 469
iglob() (glob モジュール), 516
ignorableWhitespace() (xml.sax.handler.ContentHandler のメソッド), 1515
ignore (pdb command), 2106
ignore_errors() (codecs モジュール), 206
IGNORE_EXCEPTION_DETAIL (doctest モジュール), 1928
ignore_patterns() (shutil モジュール), 522
IGNORECASE (re モジュール), 147
--ignore-dir=<dir>
    trace command line option, 2128
--ignore-module=<mod>
    trace command line option, 2128
ihave() (nntplib.NNTP のメソッド), 1638
IISCGIHandler (wsgiref.handlers のクラス), 1557
ilshift() (operator モジュール), 469
imag (numbers.Complex の属性), 361
imap() (multiprocessing.pool.Pool のメソッド), 1030
IMAP4
    protocol, 1623
IMAP4 (imaplib のクラス), 1623
IMAP4_SSL
    protocol, 1623
IMAP4_SSL (imaplib のクラス), 1623
IMAP4_stream
    protocol, 1623
IMAP4_stream (imaplib のクラス), 1624
IMAP4.abort, 1623
IMAP4.error, 1623
IMAP4.readonly, 1623
imap_unordered() (multiprocessing.pool.Pool のメソッド), 1030
imaplib (モジュール), 1623
imatmul() (operator モジュール), 469
imghdr (モジュール), 1744
immedok() (curses.window のメソッド), 907
immutable, **2484**
    sequence types, 50
imod() (operator モジュール), 469
imp
    モジュール, 32
imp (モジュール), 2467
ImpImporter (pkgutil のクラス), 2282
impl_detail() (test.support モジュール), 2077
implementation (sys モジュール), 2177
ImpLoader (pkgutil のクラス), 2282
import
    文, 32, 2269, 2467
import (2to3 fixer), 2062
import path, **2484**
import_fresh_module() (test.support モジュール), 2078
IMPORT_FROM (opcode), 2370
import_module() (importlib モジュール), 2291
import_module() (test.support モジュール), 2078
IMPORT_NAME (opcode), 2370
IMPORT_STAR (opcode), 2366
importer, **2484**
ImportError, 115
importing, **2484**
importlib (モジュール), 2290
importlib.abc (モジュール), 2293
importlib.machinery (モジュール), 2303
importlib.metadata (モジュール), 2316
importlib.resources (モジュール), 2301
importlib.util (モジュール), 2309
imports (2to3 fixer), 2062
imports2 (2to3 fixer), 2062
ImportWarning, 122
ImproperConnectionState, 1606
imul() (operator モジュール), 469
in
    演算子, 39, 47
in_dll() (ctypes._CData のメソッド), 969
in_table_a1() (stringprep モジュール), 183
in_table_b1() (stringprep モジュール), 184
in_table_c3() (stringprep モジュール), 184
in_table_c4() (stringprep モジュール), 184
in_table_c5() (stringprep モジュール), 184
in_table_c6() (stringprep モジュール), 184
in_table_c7() (stringprep モジュール), 184
in_table_c8() (stringprep モジュール), 184
in_table_c9() (stringprep モジュール), 184
in_table_c11() (stringprep モジュール), 184
in_table_c11_c12() (stringprep モジュール), 184
in_table_c12() (stringprep モジュール), 184
in_table_c21() (stringprep モジュール), 184
in_table_c21_c22() (stringprep モジュール), 184
in_table_c22() (stringprep モジュール), 184
in_table_d1() (stringprep モジュール), 185
in_table_d2() (stringprep モジュール), 185
in_transaction (sqlite3.Connection の属性), 573
inch() (curses.window のメソッド), 907
inclusive (tracemalloc.DomainFilter の属性), 2136
inclusive (tracemalloc.Filter の属性), 2136
Incomplete, 1450
IncompleteRead, 1606
IncompleteReadError, 1141
increment_lineno() (ast モジュール), 2332
IncrementalDecoder (codecs のクラス), 208
incrementaldecoder (codecs.CodecInfo の属性), 201
IncrementalEncoder (codecs のクラス), 208
incrementalencoder (codecs.CodecInfo の属性), 201
IncrementalNewlineDecoder (io のクラス), 789
IncrementalParser (xml.sax.xmlreader のクラス), 1518
indent (doctest.Example の属性), 1939
INDENT (token モジュール), 2338
indent() (textwrap モジュール), 177
IndentationError, 118
--indentlevel=<num>
    pickletools command line option, 2376
index() (array.array のメソッド), 311
index() (bytearray のメソッド), 73
index() (bytes のメソッド), 73
index() (collections.deque のメソッド), 283
index() (multiprocessing.shared_memory.ShareableList のメソッド), 1052
index() (operator モジュール), 463
index() (sequence method), 47
index() (str のメソッド), 58
index() (tkinter.ttk.Notebook のメソッド), 1854
index() (tkinter.ttk.Treeview のメソッド), 1862
IndexError, 115
indexOf() (operator モジュール), 465
IndexSizeErr, 1497
inet_aton() (socket モジュール), 1225
inet_ntoa() (socket モジュール), 1225
inet_ntop() (socket モジュール), 1225
inet_pton() (socket モジュール), 1225
Inexact (decimal のクラス), 400
inf (cmath モジュール), 376
inf (math モジュール), 372
infile

    json.tool command line option, 1414
infile (*shlex.shlex* の属性), 1826
Infinity, 13
infj (*cmath* モジュール), 376
--info
    zipapp command line option, 2158
info() (*dis.Bytecode* のメソッド), 2358
info() (*gettext.NullTranslations* のメソッド), 1758
info() (*logging* モジュール), 858
info() (*logging.Logger* のメソッド), 846
infolist() (*zipfile.ZipFile* のメソッド), 619
.ini
    file, 657
ini file, 657
init() (*mimetypes* モジュール), 1439
init_color() (*curses* モジュール), 898
init_database() (*msilib* モジュール), 2386
init_pair() (*curses* モジュール), 898
inited (*mimetypes* モジュール), 1440
initgroups() (*os* モジュール), 711
initial_indent (*textwrap.TextWrapper* の属性), 179
initscr() (*curses* モジュール), 898
inode() (*os.DirEntry* のメソッド), 740
INPLACE_ADD (*opcode*), 2364
INPLACE_AND (*opcode*), 2364
INPLACE_FLOOR_DIVIDE (*opcode*), 2364
INPLACE_LSHIFT (*opcode*), 2364
INPLACE_MATRIX_MULTIPLY (*opcode*), 2364
INPLACE_MODULO (*opcode*), 2364
INPLACE_MULTIPLY (*opcode*), 2364
INPLACE_OR (*opcode*), 2365
INPLACE_POWER (*opcode*), 2364
INPLACE_RSHIFT (*opcode*), 2364
INPLACE_SUBTRACT (*opcode*), 2364
INPLACE_TRUE_DIVIDE (*opcode*), 2364
INPLACE_XOR (*opcode*), 2364
input (*2to3 fixer*), 2062
input() (*fileinput* モジュール), 498
input() (組み込み関数), 16
input_charset (*email.charset.Charset* の属性), 1394
input_codec (*email.charset.Charset* の属性), 1395
InputOnly (*tkinter.tix* のクラス), 1873
InputSource (*xml.sax.xmlreader* のクラス), 1518
insch() (*curses.window* のメソッド), 907
insdelln() (*curses.window* のメソッド), 907
insert() (*array.array* のメソッド), 311
insert() (*collections.deque* のメソッド), 283
insert() (*sequence method*), 50
insert() (*tkinter.ttk.Notebook* のメソッド), 1854
insert() (*tkinter.ttk.Treeview* のメソッド), 1862
insert() (*xml.etree.ElementTree.Element* のメソッド), 1478
insert_text() (*readline* モジュール), 186
insertBefore() (*xml.dom.Node* のメソッド), 1491
insertln() (*curses.window* のメソッド), 907
insnstr() (*curses.window* のメソッド), 907
insort() (*bisect* モジュール), 307
insort_left() (*bisect* モジュール), 307
insort_right() (*bisect* モジュール), 307
inspect (モジュール), 2249
inspect command line option
    --details, 2268
InspectLoader (*importlib.abc* のクラス), 2298
insstr() (*curses.window* のメソッド), 908
install() (*gettext* モジュール), 1756
install() (*gettext.NullTranslations* のメソッド), 1758
install_opener() (*urllib.request* モジュール), 1565
install_scripts() (*venv.EnvBuilder* のメソッド), 2151
installHandler() (*unittest* モジュール), 1986
instate() (*tkinter.ttk.Widget* のメソッド), 1849
instr() (*curses.window* のメソッド), 908
instream (*shlex.shlex* の属性), 1826
Instruction (*dis* のクラス), 2361
Instruction.arg (*dis* モジュール), 2361
Instruction.argrepr (*dis* モジュール), 2361
Instruction.argval (*dis* モジュール), 2361
Instruction.is_jump_target (*dis* モジュール), 2362
Instruction.offset (*dis* モジュール), 2361
Instruction.opcode (*dis* モジュール), 2361
Instruction.opname (*dis* モジュール), 2361
Instruction.starts_line (*dis* モジュール), 2361
int
    組み込み関数, 39
int (*uuid.UUID* の属性), 1658
int (組み込みクラス), 16
Int2AP() (*imaplib* モジュール), 1624
int_info (*sys* モジュール), 2178
integer
    literals, 39
    types, operations on, 41
    オブジェクト, 39
Integral (*numbers* のクラス), 362
Integrated Development Environment, 1875
IntegrityError, 585
Intel/DVI ADPCM, 1727
IntEnum (*enum* のクラス), 338
interact (*pdb command*), 2108
interact() (*code* モジュール), 2273
interact() (*code.InteractiveConsole* のメソッド), 2275
interact() (*telnetlib.Telnet* のメソッド), 1655
interactive, **2484**
InteractiveConsole (*code* のクラス), 2273
InteractiveInterpreter (*code* のクラス), 2273
intern (*2to3 fixer*), 2062
intern() (*sys* モジュール), 2179
internal_attr (*zipfile.ZipInfo* の属性), 626
Internaldate2tuple() (*imaplib* モジュール), 1624
internalSubset (*xml.dom.DocumentType* の属性), 1493
Internet, 1537
interpolation
    bytearray (%), 83
    bytes (%), 83
interpolation, string (%), 65
InterpolationDepthError, 677
InterpolationError, 677
InterpolationMissingOptionError, 677
InterpolationSyntaxError, 678
interpreted, **2484**
interpreter prompts, 2182
interpreter shutdown, **2485**
interpreter_requires_environment()
        (*test.support.script_helper* モジュール), 2085
interrupt() (*sqlite3.Connection* のメソッド), 576
interrupt_main() (*_thread* モジュール), 1097
InterruptedError, 121
intersection() (*frozenset* のメソッド), 96
intersection_update() (*frozenset* のメソッド), 97
IntFlag (*enum* のクラス), 339
intro (*cmd.Cmd* の属性), 1819
InuseAttributeErr, 1497
inv() (*operator* モジュール), 463
inv_cdf() (*statistics.NormalDist* のメソッド), 430
InvalidAccessErr, 1497
invalidate_caches() (*importlib* モジュール), 2291
invalidate_caches() (*importlib.abc.MetaPathFinder* のメソッド), 2294
invalidate_caches() (*importlib.abc.PathEntryFinder* のメソッド), 2295
invalidate_caches() (*importlib.machinery.FileFinder* のメソッド), 2306
invalidate_caches() (*importlib.machinery.PathFinder* のクラスメソッド), 2305
--invalidation-mode
        [timestamp|checked-hash|unchecked-hash]
    compileall command line option, 2354
InvalidCharacterErr, 1497

InvalidModificationErr, 1498
InvalidOperation (*decimal* のクラス), 400
InvalidStateErr, 1498
InvalidStateError, 1061, 1141
InvalidURL, 1606
invert() (*operator* モジュール), 463
IO (*typing* のクラス), 1908
io (モジュール), 773
IO_REPARSE_TAG_APPEXECLINK (*stat* モジュール), 507
IO_REPARSE_TAG_MOUNT_POINT (*stat* モジュール), 507
IO_REPARSE_TAG_SYMLINK (*stat* モジュール), 507
IOBase (*io* のクラス), 777
ioctl() (*fcntl* モジュール), 2421
ioctl() (*socket.socket* のメソッド), 1230
IOCTL_VM_SOCKETS_GET_LOCAL_CID (*socket* モジュール), 1218
IOError, 120
ior() (*operator* モジュール), 469
io.StringIO
    オブジェクト, 55
ip (*ipaddress.IPv4Interface* の属性), 1724
ip (*ipaddress.IPv6Interface* の属性), 1725
ip_address() (*ipaddress* モジュール), 1711
ip_interface() (*ipaddress* モジュール), 1712
ip_network() (*ipaddress* モジュール), 1712
ipaddress (モジュール), 1711
ipow() (*operator* モジュール), 469
ipv4_mapped (*ipaddress.IPv6Address* の属性), 1715
IPv4Address (*ipaddress* のクラス), 1712
IPv4Interface (*ipaddress* のクラス), 1724
IPv4Network (*ipaddress* のクラス), 1717
IPV6_ENABLED (*test.support* モジュール), 2069
IPv6Address (*ipaddress* のクラス), 1714
IPv6Interface (*ipaddress* のクラス), 1724
IPv6Network (*ipaddress* のクラス), 1721
irshift() (*operator* モジュール), 469
is
    演算子, 38
is not
    演算子, 38
is_() (*operator* モジュール), 462
is_absolute() (*pathlib.PurePath* のメソッド), 479
is_active() (*asyncio.AbstractChildWatcher* のメソッド), 1193
is_alive() (*multiprocessing.Process* のメソッド), 1003
is_alive() (*threading.Thread* のメソッド), 983
is_android (*test.support* モジュール), 2069
is_annotated() (*symtable.Symbol* のメソッド), 2336
is_assigned() (*symtable.Symbol* のメソッド), 2337
is_attachment() (*email.message.EmailMessage* のメソッド), 1335
is_authenticated() (*urllib.request.HTTPPasswordMgrWithPriorAuth* のメソッド), 1577
is_block_device() (*pathlib.Path* のメソッド), 485
is_blocked() (*http.cookiejar.DefaultCookiePolicy* のメソッド), 1690
is_canonical() (*decimal.Context* のメソッド), 395
is_canonical() (*decimal.Decimal* のメソッド), 387
is_char_device() (*pathlib.Path* のメソッド), 485
IS_CHARACTER_JUNK() (*difflib* モジュール), 168
is_check_supported() (*lzma* モジュール), 613
is_closed() (*asyncio.loop* のメソッド), 1144
is_closing() (*asyncio.BaseTransport* のメソッド), 1176
is_closing() (*asyncio.StreamWriter* のメソッド), 1122
is_dataclass() (*dataclasses* モジュール), 2209
is_declared_global() (*symtable.Symbol* のメソッド), 2336
is_dir() (*os.DirEntry* のメソッド), 740
is_dir() (*pathlib.Path* のメソッド), 484
is_dir() (*zipfile.Path* のメソッド), 623
is_dir() (*zipfile.ZipInfo* のメソッド), 625
is_enabled() (*faulthandler* モジュール), 2099
is_expired() (*http.cookiejar.Cookie* のメソッド), 1693
is_fifo() (*pathlib.Path* のメソッド), 485
is_file() (*os.DirEntry* のメソッド), 740
is_file() (*pathlib.Path* のメソッド), 485
is_file() (*zipfile.Path* のメソッド), 623
is_finalizing() (*sys* モジュール), 2179
is_finite() (*decimal.Context* のメソッド), 395
is_finite() (*decimal.Decimal* のメソッド), 387
is_free() (*symtable.Symbol* のメソッド), 2337
is_global (*ipaddress.IPv4Address* の属性), 1714
is_global (*ipaddress.IPv6Address* の属性), 1715
is_global() (*symtable.Symbol* のメソッド), 2336
is_hop_by_hop() (*wsgiref.util* モジュール), 1552
is_imported() (*symtable.Symbol* のメソッド), 2336
is_infinite() (*decimal.Context* のメソッド), 395
is_infinite() (*decimal.Decimal* のメソッド), 387
is_integer() (*float* のメソッド), 43
is_jython (*test.support* モジュール), 2069
IS_LINE_JUNK() (*difflib* モジュール), 168
is_linetouched() (*curses.window* のメソッド), 908
is_link_local (*ipaddress.IPv4Address* の属性), 1714
is_link_local (*ipaddress.IPv4Network* の属性), 1718
is_link_local (*ipaddress.IPv6Address* の属性), 1715
is_link_local (*ipaddress.IPv6Network* の属性), 1722
is_local() (*symtable.Symbol* のメソッド), 2336
is_loopback (*ipaddress.IPv4Address* の属性), 1714
is_loopback (*ipaddress.IPv4Network* の属性), 1718
is_loopback (*ipaddress.IPv6Address* の属性), 1715
is_loopback (*ipaddress.IPv6Network* の属性), 1721
is_mount() (*pathlib.Path* のメソッド), 485
is_multicast (*ipaddress.IPv4Address* の属性), 1713
is_multicast (*ipaddress.IPv4Network* の属性), 1718
is_multicast (*ipaddress.IPv6Address* の属性), 1715
is_multicast (*ipaddress.IPv6Network* の属性), 1721
is_multipart() (*email.message.EmailMessage* のメソッド), 1330
is_multipart() (*email.message.Message* のメソッド), 1378
is_namespace() (*symtable.Symbol* のメソッド), 2337
is_nan() (*decimal.Context* のメソッド), 395
is_nan() (*decimal.Decimal* のメソッド), 387
is_nested() (*symtable.SymbolTable* のメソッド), 2335
is_nonlocal() (*symtable.Symbol* のメソッド), 2336
is_normal() (*decimal.Context* のメソッド), 395
is_normal() (*decimal.Decimal* のメソッド), 387
is_normalized() (*unicodedata* モジュール), 182
is_not() (*operator* モジュール), 462
is_not_allowed() (*http.cookiejar.DefaultCookiePolicy* のメソッド), 1690
is_optimized() (*symtable.SymbolTable* のメソッド), 2335
is_package() (*importlib.abc.InspectLoader* のメソッド), 2298
is_package() (*importlib.abc.SourceLoader* のメソッド), 2301
is_package() (*importlib.machinery.ExtensionFileLoader* のメソッド), 2308
is_package() (*importlib.machinery.SourceFileLoader* のメソッド), 2306
is_package() (*importlib.machinery.SourcelessFileLoader* のメソッド), 2307
is_package() (*zipimport.zipimporter* のメソッド), 2281
is_parameter() (*symtable.Symbol* のメソッド), 2336
is_private (*ipaddress.IPv4Address* の属性), 1714
is_private (*ipaddress.IPv4Network* の属性), 1718
is_private (*ipaddress.IPv6Address* の属性), 1715
is_private (*ipaddress.IPv6Network* の属性), 1721
is_python_build() (*sysconfig* モジュール), 2193
is_qnan() (*decimal.Context* のメソッド), 395
is_qnan() (*decimal.Decimal* のメソッド), 387
is_reading() (*asyncio.ReadTransport* のメソッド), 1177
is_referenced() (*symtable.Symbol* のメソッド), 2336
is_reserved (*ipaddress.IPv4Address* の属性), 1714
is_reserved (*ipaddress.IPv4Network* の属性), 1718
is_reserved (*ipaddress.IPv6Address* の属性), 1715
is_reserved (*ipaddress.IPv6Network* の属性), 1721

is_reserved() (*pathlib.PurePath* のメソッド), 479
is_resource() (*importlib.abc.ResourceReader* のメソッド), 2297
is_resource() (*importlib.resources* モジュール), 2303
is_resource_enabled() (*test.support* モジュール), 2071
is_running() (*asyncio.loop* のメソッド), 1144
is_safe (*uuid.UUID* の属性), 1658
is_serving() (*asyncio.Server* のメソッド), 1165
is_set() (*asyncio.Event* のメソッド), 1129
is_set() (*threading.Event* のメソッド), 990
is_signed() (*decimal.Context* のメソッド), 396
is_signed() (*decimal.Decimal* のメソッド), 387
is_site_local (*ipaddress.IPv6Address* の属性), 1715
is_site_local (*ipaddress.IPv6Network* の属性), 1722
is_snan() (*decimal.Context* のメソッド), 396
is_snan() (*decimal.Decimal* のメソッド), 387
is_socket() (*pathlib.Path* のメソッド), 485
is_subnormal() (*decimal.Context* のメソッド), 396
is_subnormal() (*decimal.Decimal* のメソッド), 387
is_symlink() (*os.DirEntry* のメソッド), 741
is_symlink() (*pathlib.Path* のメソッド), 485
is_tarfile() (*tarfile* モジュール), 630
is_term_resized() (*curses* モジュール), 898
is_tracing() (*tracemalloc* モジュール), 2135
is_tracked() (*gc* モジュール), 2246
is_unspecified (*ipaddress.IPv4Address* の属性), 1714
is_unspecified (*ipaddress.IPv4Network* の属性), 1718
is_unspecified (*ipaddress.IPv6Address* の属性), 1715
is_unspecified (*ipaddress.IPv6Network* の属性), 1721
is_wintouched() (*curses.window* のメソッド), 908
is_zero() (*decimal.Context* のメソッド), 396
is_zero() (*decimal.Decimal* のメソッド), 387
is_zipfile() (*zipfile* モジュール), 617
isabs() (*os.path* モジュール), 494
isabstract() (*inspect* モジュール), 2253
IsADirectoryError, 121
isalnum() (*bytearray* のメソッド), 78
isalnum() (*bytes* のメソッド), 78
isalnum() (*curses.ascii* モジュール), 920
isalnum() (*str* のメソッド), 58
isalpha() (*bytearray* のメソッド), 79
isalpha() (*bytes* のメソッド), 79
isalpha() (*curses.ascii* モジュール), 920
isalpha() (*str* のメソッド), 59
isascii() (*bytearray* のメソッド), 79
isascii() (*bytes* のメソッド), 79
isascii() (*curses.ascii* モジュール), 920
isascii() (*str* のメソッド), 59
isasyncgen() (*inspect* モジュール), 2252
isasyncgenfunction() (*inspect* モジュール), 2252
isatty() (*chunk.Chunk* のメソッド), 1742
isatty() (*io.IOBase* のメソッド), 778
isatty() (*os* モジュール), 717
isawaitable() (*inspect* モジュール), 2252
isblank() (*curses.ascii* モジュール), 920
isblk() (*tarfile.TarInfo* のメソッド), 639
isbuiltin() (*inspect* モジュール), 2253
ischr() (*tarfile.TarInfo* のメソッド), 639
isclass() (*inspect* モジュール), 2251
isclose() (*cmath* モジュール), 375
isclose() (*math* モジュール), 366
iscntrl() (*curses.ascii* モジュール), 920
iscode() (*inspect* モジュール), 2253
iscoroutine() (*asyncio* モジュール), 1118
iscoroutine() (*inspect* モジュール), 2252
iscoroutinefunction() (*asyncio* モジュール), 1118
iscoroutinefunction() (*inspect* モジュール), 2252
isctrl() (*curses.ascii* モジュール), 921
isDaemon() (*threading.Thread* のメソッド), 983
isdatadescriptor() (*inspect* モジュール), 2253
isdecimal() (*str* のメソッド), 59
isdev() (*tarfile.TarInfo* のメソッド), 639
isdigit() (*bytearray* のメソッド), 79
isdigit() (*bytes* のメソッド), 79
isdigit() (*curses.ascii* モジュール), 920
isdigit() (*str* のメソッド), 59
isdir() (*os.path* モジュール), 494
isdir() (*tarfile.TarInfo* のメソッド), 639
isdisjoint() (*frozenset* のメソッド), 95
isdown() (*turtle* モジュール), 1790
iselement() (*xml.etree.ElementTree* モジュール), 1472
isenabled() (*gc* モジュール), 2245
isEnabledFor() (*logging.Logger* のメソッド), 845
isendwin() (*curses* モジュール), 898
ISEOF() (*token* モジュール), 2338
isexpr() (*parser* モジュール), 2324
isexpr() (*parser.ST* のメソッド), 2325
isfifo() (*tarfile.TarInfo* のメソッド), 639
isfile() (*os.path* モジュール), 494
isfile() (*tarfile.TarInfo* のメソッド), 639
isfinite() (*cmath* モジュール), 375
isfinite() (*math* モジュール), 367
isfirstline() (*fileinput* モジュール), 499
isframe() (*inspect* モジュール), 2253
isfunction() (*inspect* モジュール), 2251
isfuture() (*asyncio* モジュール), 1170
isgenerator() (*inspect* モジュール), 2252
isgeneratorfunction() (*inspect* モジュール), 2251
isgetsetdescriptor() (*inspect* モジュール), 2253
isgraph() (*curses.ascii* モジュール), 920
isidentifier() (*str* のメソッド), 59
isinf() (*cmath* モジュール), 375
isinf() (*math* モジュール), 367
isinstance (*2to3 fixer*), 2062
isinstance() (組み込み関数), 17
iskeyword() (*keyword* モジュール), 2342
isleap() (*calendar* モジュール), 273
islice() (*itertools* モジュール), 443
islink() (*os.path* モジュール), 495
islnk() (*tarfile.TarInfo* のメソッド), 639
islower() (*bytearray* のメソッド), 79
islower() (*bytes* のメソッド), 79
islower() (*curses.ascii* モジュール), 920
islower() (*str* のメソッド), 59
ismemberdescriptor() (*inspect* モジュール), 2253
ismeta() (*curses.ascii* モジュール), 921
ismethod() (*inspect* モジュール), 2251
ismethoddescriptor() (*inspect* モジュール), 2253
ismodule() (*inspect* モジュール), 2251
ismount() (*os.path* モジュール), 495
isnan() (*cmath* モジュール), 375
isnan() (*math* モジュール), 367
ISNONTERMINAL() (*token* モジュール), 2338
isnumeric() (*str* のメソッド), 60
isocalendar() (*datetime.date* のメソッド), 235
isocalendar() (*datetime.datetime* のメソッド), 247
isoformat() (*datetime.date* のメソッド), 236
isoformat() (*datetime.datetime* のメソッド), 247
isoformat() (*datetime.time* のメソッド), 254
IsolatedAsyncioTestCase (*unittest* のクラス), 1971
isolation_level (*sqlite3.Connection* の属性), 573
isoweekday() (*datetime.date* のメソッド), 235
isoweekday() (*datetime.datetime* のメソッド), 247
isprint() (*curses.ascii* モジュール), 920
isprintable() (*str* のメソッド), 60
ispunct() (*curses.ascii* モジュール), 920
isqrt() (*math* モジュール), 367
isreadable() (*pprint* モジュール), 331
isreadable() (*pprint.PrettyPrinter* のメソッド), 332
isrecursive() (*pprint* モジュール), 331
isrecursive() (*pprint.PrettyPrinter* のメソッド), 332
isreg() (*tarfile.TarInfo* のメソッド), 639
isReservedKey() (*http.cookies.Morsel* のメソッド), 1681
isroutine() (*inspect* モジュール), 2253
isSameNode() (*xml.dom.Node* のメソッド), 1491
isspace() (*bytearray* のメソッド), 80

isspace() (*bytes* のメソッド), 80
isspace() (*curses.ascii* モジュール), 920
isspace() (*str* のメソッド), 60
isstdin() (*fileinput* モジュール), 499
issubclass() (組み込み関数), 17
issubset() (*frozenset* のメソッド), 95
issuite() (*parser* モジュール), 2324
issuite() (*parser.ST* のメソッド), 2325
issuperset() (*frozenset* のメソッド), 95
issym() (*tarfile.TarInfo* のメソッド), 639
ISTERMINAL() (*token* モジュール), 2338
istitle() (*bytearray* のメソッド), 80
istitle() (*bytes* のメソッド), 80
istitle() (*str* のメソッド), 60
istraceback() (*inspect* モジュール), 2253
isub() (*operator* モジュール), 469
isupper() (*bytearray* のメソッド), 80
isupper() (*bytes* のメソッド), 80
isupper() (*curses.ascii* モジュール), 921
isupper() (*str* のメソッド), 60
isvisible() (*turtle* モジュール), 1794
isxdigit() (*curses.ascii* モジュール), 921
item() (*tkinter.ttk.Treeview* のメソッド), 1862
item() (*xml.dom.NamedNodeMap* のメソッド), 1496
item() (*xml.dom.NodeList* のメソッド), 1492
itemgetter() (*operator* モジュール), 465
items() (*configparser.ConfigParser* のメソッド), 675
items() (*contextvars.Context* のメソッド), 1095
items() (*dict* のメソッド), 100
items() (*email.message.EmailMessage* のメソッド), 1332
items() (*email.message.Message* のメソッド), 1381
items() (*mailbox.Mailbox* のメソッド), 1418
items() (*types.MappingProxyType* のメソッド), 326
items() (*xml.etree.ElementTree.Element* のメソッド), 1477
itemsize (*array.array* の属性), 310
itemsize (*memoryview* の属性), 93
ItemsView (*collections.abc* のクラス), 299
ItemsView (*typing* のクラス), 1905
iter() (組み込み関数), 17
iter() (*xml.etree.ElementTree.Element* のメソッド), 1478
iter() (*xml.etree.ElementTree.ElementTree* のメソッド), 1480
iter_attachments() (*email.message.EmailMessage* のメソッド), 1337
iter_child_nodes() (*ast* モジュール), 2332
iter_fields() (*ast* モジュール), 2332
iter_importers() (*pkgutil* モジュール), 2283
iter_modules() (*pkgutil* モジュール), 2284
iter_parts() (*email.message.EmailMessage* のメソッド), 1337
iter_unpack() (*struct* モジュール), 194
iter_unpack() (*struct.Struct* のメソッド), 199
iterable, **2485**
Iterable (*collections.abc* のクラス), 298
Iterable (*typing* のクラス), 1903
iterator, **2485**
Iterator (*collections.abc* のクラス), 298
Iterator (*typing* のクラス), 1903
iterator protocol, 46
iterdecode() (*codecs* モジュール), 203
iterdir() (*pathlib.Path* のメソッド), 486
iterdir() (*zipfile.Path* のメソッド), 623
iterdump() (*sqlite3.Connection* のメソッド), 579
iterencode() (*codecs* モジュール), 203
iterencode() (*json.JSONEncoder* のメソッド), 1410
iterfind() (*xml.etree.ElementTree.Element* のメソッド), 1478
iterfind() (*xml.etree.ElementTree.ElementTree* のメソッド), 1480
iteritems() (*mailbox.Mailbox* のメソッド), 1418
iterkeys() (*mailbox.Mailbox* のメソッド), 1418
itermonthdates() (*calendar.Calendar* のメソッド), 269
itermonthdays() (*calendar.Calendar* のメソッド), 269
itermonthdays2() (*calendar.Calendar* のメソッド), 270
itermonthdays3() (*calendar.Calendar* のメソッド), 270
itermonthdays4() (*calendar.Calendar* のメソッド), 270
iterparse() (*xml.etree.ElementTree* モジュール), 1472
itertext() (*xml.etree.ElementTree.Element* のメソッド), 1478
itertools (*2to3 fixer*), 2062
itertools (モジュール), 435
itertools_imports (*2to3 fixer*), 2062
itervalues() (*mailbox.Mailbox* のメソッド), 1418
iterweekdays() (*calendar.Calendar* のメソッド), 269
ITIMER_PROF (*signal* モジュール), 1313
ITIMER_REAL (*signal* モジュール), 1313
ITIMER_VIRTUAL (*signal* モジュール), 1313
ItimerError, 1314
itruediv() (*operator* モジュール), 469
ixor() (*operator* モジュール), 469

J

-j N
    compileall command line option, 2354
Jansen, Jack, 1451
java_ver() (*platform* モジュール), 926
join() (*asyncio.Queue* のメソッド), 1138
join() (*bytearray* のメソッド), 73
join() (*bytes* のメソッド), 73
join() (*multiprocessing.JoinableQueue* のメソッド), 1009
join() (*multiprocessing.pool.Pool* のメソッド), 1031
join() (*multiprocessing.Process* のメソッド), 1003
join() (*os.path* モジュール), 495
join() (*queue.Queue* のメソッド), 1090
join() (*shlex* モジュール), 1822
join() (*str* のメソッド), 60
join() (*threading.Thread* のメソッド), 982
join_thread() (*multiprocessing.Queue* のメソッド), 1008
join_thread() (*test.support* モジュール), 2079
JoinableQueue (*multiprocessing* のクラス), 1009
joinpath() (*pathlib.PurePath* のメソッド), 480
js_output() (*http.cookies.BaseCookie* のメソッド), 1679
js_output() (*http.cookies.Morsel* のメソッド), 1681
json (モジュール), 1402
JSONDecodeError, 1411
JSONDecoder (*json* のクラス), 1408
JSONEncoder (*json* のクラス), 1409
--json-lines
    json.tool command line option, 1414
json.tool (モジュール), 1414
json.tool command line option
    -h, 1415
    --help, 1415
    infile, 1414
    --json-lines, 1414
    outfile, 1414
    --sort-keys, 1414
jump (*pdb command*), 2107
JUMP_ABSOLUTE (*opcode*), 2371
JUMP_FORWARD (*opcode*), 2370
JUMP_IF_FALSE_OR_POP (*opcode*), 2371
JUMP_IF_TRUE_OR_POP (*opcode*), 2371

K

-k
    unittest command line option, 1952
kbhit() (*msvcrt* モジュール), 2394
KDEDIR, 1539
kevent() (*select* モジュール), 1289
key (*http.cookies.Morsel* の属性), 1681
key function, **2485**
KEY_ALL_ACCESS (*winreg* モジュール), 2403
KEY_CREATE_LINK (*winreg* モジュール), 2404
KEY_CREATE_SUB_KEY (*winreg* モジュール), 2404
KEY_ENUMERATE_SUB_KEYS (*winreg* モジュール), 2404

KEY_EXECUTE (*winreg* モジュール), 2403
KEY_NOTIFY (*winreg* モジュール), 2404
KEY_QUERY_VALUE (*winreg* モジュール), 2403
KEY_READ (*winreg* モジュール), 2403
KEY_SET_VALUE (*winreg* モジュール), 2404
KEY_WOW64_32KEY (*winreg* モジュール), 2404
KEY_WOW64_64KEY (*winreg* モジュール), 2404
KEY_WRITE (*winreg* モジュール), 2403
KeyboardInterrupt, 115
KeyError, 115
keylog_filename (*ssl.SSLContext* の属性), 1273
keyname() (*curses* モジュール), 898
keypad() (*curses.window* のメソッド), 908
keyrefs() (*weakref.WeakKeyDictionary* のメソッド), 315
keys() (*contextvars.Context* のメソッド), 1095
keys() (*dict* のメソッド), 100
keys() (*email.message.EmailMessage* のメソッド), 1332
keys() (*email.message.Message* のメソッド), 1381
keys() (*mailbox.Mailbox* のメソッド), 1418
keys() (*sqlite3.Row* のメソッド), 584
keys() (*types.MappingProxyType* のメソッド), 326
keys() (*xml.etree.ElementTree.Element* のメソッド), 1477
KeysView (*collections.abc* のクラス), 299
KeysView (*typing* のクラス), 1905
keyword (モジュール), 2342
keyword argument, **2486**
keywords (*functools.partial* の属性), 461
kill() (*asyncio.asyncio.subprocess.Process* のメソッド), 1135
kill() (*asyncio.SubprocessTransport* のメソッド), 1179
kill() (*multiprocessing.Process* のメソッド), 1004
kill() (*os* モジュール), 758
kill() (*subprocess.Popen* のメソッド), 1074
kill_python() (*test.support.script_helper* モジュール), 2085
killchar() (*curses* モジュール), 898
killpg() (*os* モジュール), 758
kind (*inspect.Parameter* の属性), 2257
knownfiles (*mimetypes* モジュール), 1440
kqueue() (*select* モジュール), 1289
KqueueSelector (*selectors* のクラス), 1300
kwargs (*inspect.BoundArguments* の属性), 2259
kwlist (*keyword* モジュール), 2342

L

-l
    compileall command line option, 2353
    pickletools command line option, 2376
    trace command line option, 2127
L (*re* モジュール), 147
-l <tarfile>
    tarfile command line option, 644
-l <zipfile>
    zipfile command line option, 627
LabelEntry (*tkinter.tix* のクラス), 1870
LabelFrame (*tkinter.tix* のクラス), 1870
lambda, **2486**
LambdaType (*types* モジュール), 323
LANG, 1753, 1756, 1766, 1769
LANGUAGE, 1753, 1756
language
    C, 39, 40
large files, 2409
LARGEST (*test.support* モジュール), 2070
LargeZipFile, 616
last() (*nntplib.NNTP* のメソッド), 1637
last_accepted (*multiprocessing.connection.Listener* の属性), 1033
last_traceback (*sys* モジュール), 2179
last_type (*sys* モジュール), 2179
last_value (*sys* モジュール), 2179
lastChild (*xml.dom.Node* の属性), 1490
lastcmd (*cmd.Cmd* の属性), 1818
lastgroup (*re.Match* の属性), 156
lastindex (*re.Match* の属性), 156
lastResort (*logging* モジュール), 862
lastrowid (*sqlite3.Cursor* の属性), 583
layout() (*tkinter.ttk.Style* のメソッド), 1866
lazycache() (*linecache* モジュール), 519
LazyLoader (*importlib.util* のクラス), 2312
LBRACE (*token* モジュール), 2339
LBYL, **2486**
LC_ALL, 1753, 1756
LC_ALL (*locale* モジュール), 1771
LC_COLLATE (*locale* モジュール), 1771
LC_CTYPE (*locale* モジュール), 1771
LC_MESSAGES, 1753, 1756
LC_MESSAGES (*locale* モジュール), 1771
LC_MONETARY (*locale* モジュール), 1771
LC_NUMERIC (*locale* モジュール), 1771
LC_TIME (*locale* モジュール), 1771
lchflags() (*os* モジュール), 731
lchmod() (*os* モジュール), 731
lchmod() (*pathlib.Path* のメソッド), 486
lchown() (*os* モジュール), 732
ldexp() (*math* モジュール), 367
ldgettext() (*gettext* モジュール), 1755
ldngettext() (*gettext* モジュール), 1755
le() (*operator* モジュール), 462
leapdays() (*calendar* モジュール), 273
leaveok() (*curses.window* のメソッド), 908
left (*filecmp.dircmp* の属性), 508
left() (*turtle* モジュール), 1782
left_list (*filecmp.dircmp* の属性), 509
left_only (*filecmp.dircmp* の属性), 509
LEFTSHIFT (*token* モジュール), 2340
LEFTSHIFTEQUAL (*token* モジュール), 2340
len
    組み込み関数, 47, 97
len() (組み込み関数), 18
length (*xml.dom.NamedNodeMap* の属性), 1496
length (*xml.dom.NodeList* の属性), 1492
length_hint() (*operator* モジュール), 465
LESS (*token* モジュール), 2339
LESSEQUAL (*token* モジュール), 2340
lexists() (*os.path* モジュール), 493
lgamma() (*math* モジュール), 372
lgettext() (*gettext* モジュール), 1755
lgettext() (*gettext.GNUTranslations* のメソッド), 1760
lgettext() (*gettext.NullTranslations* のメソッド), 1758
lib2to3 (モジュール), 2065
libc_ver() (*platform* モジュール), 927
library (*dbm.ndbm* モジュール), 566
library (*ssl.SSLError* の属性), 1245
LibraryLoader (*ctypes* のクラス), 960
license (組み込み変数), 36
LifoQueue (*asyncio* のクラス), 1139
LifoQueue (*queue* のクラス), 1088
light-weight processes, 1096
limit_denominator() (*fractions.Fraction* のメソッド), 412
LimitOverrunError, 1141
lin2adpcm() (*audioop* モジュール), 1728
lin2alaw() (*audioop* モジュール), 1729
lin2lin() (*audioop* モジュール), 1729
lin2ulaw() (*audioop* モジュール), 1729
line() (*msilib.Dialog* のメソッド), 2391
line_buffering (*io.TextIOWrapper* の属性), 788
line_num (*csv.csvreader* の属性), 655
line-buffered I/O, 23
linecache (モジュール), 518
lineno (*ast.AST* の属性), 2327
lineno (*doctest.DocTest* の属性), 1939
lineno (*doctest.Example* の属性), 1939
lineno (*json.JSONDecodeError* の属性), 1411
lineno (*pyclbr.Class* の属性), 2350

lineno (*pyclbr.Function* の属性), 2349
lineno (*re.error* の属性), 152
lineno (*shlex.shlex* の属性), 1826
lineno (*SyntaxError* の属性), 118
lineno (*traceback.TracebackException* の属性), 2238
lineno (*tracemalloc.Filter* の属性), 2137
lineno (*tracemalloc.Frame* の属性), 2137
lineno (*xml.parsers.expat.ExpatError* の属性), 1530
lineno() (*fileinput* モジュール), 499
LINES, 897, 902
lines (*os.terminal_size* の属性), 726
linesep (*email.policy.Policy* の属性), 1350
linesep (*os* モジュール), 771
lineterminator (*csv.Dialect* の属性), 654
LineTooLong, 1606
link() (*os* モジュール), 732
link_to() (*pathlib.Path* のメソッド), 489
linkname (*tarfile.TarInfo* の属性), 638
LinkOutsideDestinationError, 631
list, **2486**
  type, operations on, 50
  オブジェクト, 50, 51
list (*pdb command*), 2107
List (*typing* のクラス), 1905
list (組み込みクラス), 51
--list <tarfile>
  tarfile command line option, 644
--list <zipfile>
  zipfile command line option, 627
list comprehension, **2486**
list() (*imaplib.IMAP4* のメソッド), 1626
list() (*multiprocessing.managers.SyncManager* のメソッド), 1023
list() (*nntplib.NNTP* のメソッド), 1635
list() (*poplib.POP3* のメソッド), 1621
list() (*tarfile.TarFile* のメソッド), 634
LIST_APPEND (*opcode*), 2366
list_dialects() (*csv* モジュール), 651
list_folders() (*mailbox.Maildir* のメソッド), 1421
list_folders() (*mailbox.MH* のメソッド), 1424
listdir() (*os* モジュール), 732
listen() (*asyncore.dispatcher* のメソッド), 1304
listen() (*logging.config* モジュール), 865
listen() (*socket.socket* のメソッド), 1230
listen() (*turtle* モジュール), 1804
Listener (*multiprocessing.connection* のクラス), 1032
--listfuncs
  trace command line option, 2127
listMethods() (*xmlrpc.client.ServerProxy.system* のメソッド), 1697
ListNoteBook (*tkinter.tix* のクラス), 1872
listxattr() (*os* モジュール), 753
Literal (*typing* モジュール), 1915
literal_eval() (*ast* モジュール), 2331
literals
  binary, 39
  complex number, 39
  floating point, 39
  hexadecimal, 39
  integer, 39
  numeric, 39
  octal, 39
LittleEndianStructure (*ctypes* のクラス), 972
ljust() (*bytearray* のメソッド), 75
ljust() (*bytes* のメソッド), 75
ljust() (*str* のメソッド), 60
LK_LOCK (*msvcrt* モジュール), 2393
LK_NBLCK (*msvcrt* モジュール), 2393
LK_NBRLCK (*msvcrt* モジュール), 2393
LK_RLCK (*msvcrt* モジュール), 2393
LK_UNLCK (*msvcrt* モジュール), 2393
ll (*pdb command*), 2107
LMTP (*smtplib* のクラス), 1641
ln() (*decimal.Context* のメソッド), 396
ln() (*decimal.Decimal* のメソッド), 387
LNAME, 894
lngettext() (*gettext* モジュール), 1755
lngettext() (*gettext.GNUTranslations* のメソッド), 1760
lngettext() (*gettext.NullTranslations* のメソッド), 1758
load() (*http.cookiejar.FileCookieJar* のメソッド), 1687
load() (*http.cookies.BaseCookie* のメソッド), 1680
load() (*json* モジュール), 1406
load() (*marshal* モジュール), 562
load() (*pickle* モジュール), 538
load() (*pickle.Unpickler* のメソッド), 542
load() (*plistlib* モジュール), 683
load() (*tracemalloc.Snapshot* のクラスメソッド), 2138
LOAD_ATTR (*opcode*), 2370
LOAD_BUILD_CLASS (*opcode*), 2367
load_cert_chain() (*ssl.SSLContext* のメソッド), 1265
LOAD_CLASSDEREF (*opcode*), 2372
LOAD_CLOSURE (*opcode*), 2371
LOAD_CONST (*opcode*), 2369
load_default_certs() (*ssl.SSLContext* のメソッド), 1266
LOAD_DEREF (*opcode*), 2372
load_dh_params() (*ssl.SSLContext* のメソッド), 1270
load_extension() (*sqlite3.Connection* のメソッド), 577
LOAD_FAST (*opcode*), 2371
LOAD_GLOBAL (*opcode*), 2371
LOAD_METHOD (*opcode*), 2373
load_module() (*imp* モジュール), 2468
load_module() (*importlib.abc.FileLoader* のメソッド), 2300
load_module() (*importlib.abc.InspectLoader* のメソッド), 2299
load_module() (*importlib.abc.Loader* のメソッド), 2296
load_module() (*importlib.abc.SourceLoader* のメソッド), 2301
load_module() (*importlib.machinery.SourceFileLoader* のメソッド), 2307
load_module() (*importlib.machinery.SourcelessFileLoader* のメソッド), 2307
load_module() (*zipimport.zipimporter* のメソッド), 2281
LOAD_NAME (*opcode*), 2369
load_package_tests() (*test.support* モジュール), 2081
load_verify_locations() (*ssl.SSLContext* のメソッド), 1266
loader, **2486**
Loader (*importlib.abc* のクラス), 2295
loader (*importlib.machinery.ModuleSpec* の属性), 2308
loader_state (*importlib.machinery.ModuleSpec* の属性), 2309
LoadError, 1683
LoadKey() (*winreg* モジュール), 2398
LoadLibrary() (*ctypes.LibraryLoader* のメソッド), 960
loads() (*json* モジュール), 1407
loads() (*marshal* モジュール), 562
loads() (*pickle* モジュール), 539
loads() (*plistlib* モジュール), 684
loads() (*xmlrpc.client* モジュール), 1702
loadTestsFromModule() (*unittest.TestLoader* のメソッド), 1975
loadTestsFromName() (*unittest.TestLoader* のメソッド), 1975
loadTestsFromNames() (*unittest.TestLoader* のメソッド), 1976
loadTestsFromTestCase() (*unittest.TestLoader* のメソッド), 1975
local (*threading* のクラス), 980
localcontext() (*decimal* モジュール), 391
LOCALE (*re* モジュール), 147
locale (モジュール), 1765
localeconv() (*locale* モジュール), 1766
LocaleHTMLCalendar (*calendar* のクラス), 273
LocaleTextCalendar (*calendar* のクラス), 273
localName (*xml.dom.Attr* の属性), 1495
localName (*xml.dom.Node* の属性), 1490

--locals
    unittest command line option, 1952
locals() (組み込み関数), 18
localtime() (email.utils モジュール), 1398
localtime() (time モジュール), 794
Locator (xml.sax.xmlreader のクラス), 1518
Lock (asyncio のクラス), 1127
Lock (multiprocessing のクラス), 1014
Lock (threading のクラス), 984
lock() (mailbox.Babyl のメソッド), 1426
lock() (mailbox.Mailbox のメソッド), 1420
lock() (mailbox.Maildir のメソッド), 1422
lock() (mailbox.mbox のメソッド), 1423
lock() (mailbox.MH のメソッド), 1425
lock() (mailbox.MMDF のメソッド), 1426
Lock() (multiprocessing.managers.SyncManager のメソッド), 1023
lock_held() (imp モジュール), 2470
locked() (_thread.lock のメソッド), 1098
locked() (asyncio.Condition のメソッド), 1130
locked() (asyncio.Lock のメソッド), 1127
locked() (asyncio.Semaphore のメソッド), 1131
locked() (threading.Lock のメソッド), 984
lockf() (fcntl モジュール), 2422
lockf() (os モジュール), 717
locking() (msvcrt モジュール), 2393
LockType (_thread モジュール), 1096
log() (cmath モジュール), 374
log() (logging モジュール), 858
log() (logging.Logger のメソッド), 847
log() (math モジュール), 369
log1p() (math モジュール), 369
log2() (math モジュール), 369
log10() (cmath モジュール), 374
log10() (decimal.Context のメソッド), 396
log10() (decimal.Decimal のメソッド), 387
log10() (math モジュール), 369
log_date_time_string() (http.server.BaseHTTPRequestHandler のメソッド), 1675
log_error() (http.server.BaseHTTPRequestHandler のメソッド), 1675
log_exception() (wsgiref.handlers.BaseHandler のメソッド), 1560
log_message() (http.server.BaseHTTPRequestHandler のメソッド), 1675
log_request() (http.server.BaseHTTPRequestHandler のメソッド), 1674
log_to_stderr() (multiprocessing モジュール), 1036
logb() (decimal.Context のメソッド), 396
logb() (decimal.Decimal のメソッド), 387
Logger (logging のクラス), 844
LoggerAdapter (logging のクラス), 856
logging
    Errors, 843
logging (モジュール), 843
logging.config (モジュール), 863
logging.handlers (モジュール), 876
logical_and() (decimal.Context のメソッド), 396
logical_and() (decimal.Decimal のメソッド), 387
logical_invert() (decimal.Context のメソッド), 396
logical_invert() (decimal.Decimal のメソッド), 388
logical_or() (decimal.Context のメソッド), 396
logical_or() (decimal.Decimal のメソッド), 388
logical_xor() (decimal.Context のメソッド), 396
logical_xor() (decimal.Decimal のメソッド), 388
login() (ftplib.FTP のメソッド), 1615
login() (imaplib.IMAP4 のメソッド), 1627
login() (nntplib.NNTP のメソッド), 1634
login() (smtplib.SMTP のメソッド), 1643
login_cram_md5() (imaplib.IMAP4 のメソッド), 1627
LOGNAME, 710, 894
lognormvariate() (random モジュール), 417
logout() (imaplib.IMAP4 のメソッド), 1627
LogRecord (logging のクラス), 853
long (2to3 fixer), 2063
longMessage (unittest.TestCase の属性), 1969
longname() (curses モジュール), 898
lookup() (codecs モジュール), 200
lookup() (symtable.SymbolTable のメソッド), 2335
lookup() (tkinter.ttk.Style のメソッド), 1865
lookup() (unicodedata モジュール), 181
lookup_error() (codecs モジュール), 206
LookupError, 114
loop() (asyncore モジュール), 1302
lower() (bytearray のメソッド), 80
lower() (bytes のメソッド), 80
lower() (str のメソッド), 60
LPAR (token モジュール), 2338
lpAttributeList (subprocess.STARTUPINFO の属性), 1075
lru_cache() (functools モジュール), 453
lseek() (os モジュール), 718
lshift() (operator モジュール), 463
LSQB (token モジュール), 2338
lstat() (os モジュール), 733
lstat() (pathlib.Path のメソッド), 486
lstrip() (bytearray のメソッド), 75
lstrip() (bytes のメソッド), 75
lstrip() (str のメソッド), 60
lsub() (imaplib.IMAP4 のメソッド), 1627
lt() (operator モジュール), 462
lt() (turtle モジュール), 1782
LWPCookieJar (http.cookiejar のクラス), 1688
lzma (モジュール), 609
LZMACompressor (lzma のクラス), 611
LZMADecompressor (lzma のクラス), 612
LZMAError, 609
LZMAFile (lzma のクラス), 609

M

-m
    pickletools command line option, 2376
    trace command line option, 2127
M (re モジュール), 147
-m <mainfn>
    zipapp command line option, 2157
mac_ver() (platform モジュール), 926
machine() (platform モジュール), 924
macros (netrc.netrc の属性), 679
MADV_AUTOSYNC (mmap モジュール), 1325
MADV_CORE (mmap モジュール), 1325
MADV_DODUMP (mmap モジュール), 1325
MADV_DOFORK (mmap モジュール), 1325
MADV_DONTDUMP (mmap モジュール), 1325
MADV_DONTFORK (mmap モジュール), 1325
MADV_DONTNEED (mmap モジュール), 1325
MADV_FREE (mmap モジュール), 1325
MADV_HUGEPAGE (mmap モジュール), 1325
MADV_HWPOISON (mmap モジュール), 1325
MADV_MERGEABLE (mmap モジュール), 1325
MADV_NOCORE (mmap モジュール), 1325
MADV_NOHUGEPAGE (mmap モジュール), 1325
MADV_NORMAL (mmap モジュール), 1325
MADV_NOSYNC (mmap モジュール), 1325
MADV_PROTECT (mmap モジュール), 1325
MADV_RANDOM (mmap モジュール), 1325
MADV_REMOVE (mmap モジュール), 1325
MADV_SEQUENTIAL (mmap モジュール), 1325
MADV_SOFT_OFFLINE (mmap モジュール), 1325
MADV_UNMERGEABLE (mmap モジュール), 1325
MADV_WILLNEED (mmap モジュール), 1325
madvise() (mmap.mmap のメソッド), 1323
magic
    method, 2486

magic method, **2486**
MAGIC_NUMBER (*importlib.util* モジュール), 2309
MagicMock (*unittest.mock* のクラス), 2022
Mailbox (*mailbox* のクラス), 1417
mailbox (モジュール), 1416
mailcap (モジュール), 1415
Maildir (*mailbox* のクラス), 1421
MaildirMessage (*mailbox* のクラス), 1427
mailfrom (*smtpd.SMTPChannel* の属性), 1651
MailmanProxy (*smtpd* のクラス), 1650
main() (*py_compile* モジュール), 2352
main() (*site* モジュール), 2271
main() (*unittest* モジュール), 1981
--main=<mainfn>
    zipapp command line option, 2157
main_thread() (*threading* モジュール), 979
mainloop() (*turtle* モジュール), 1805
maintype (*email.headerregistry.ContentTypeHeader* の属性), 1362
major (*email.headerregistry.MIMEVersionHeader* の属性), 1362
major() (*os* モジュール), 735
make_alternative() (*email.message.EmailMessage* のメソッド), 1337
make_archive() (*shutil* モジュール), 528
make_bad_fd() (*test.support* モジュール), 2078
make_cookies() (*http.cookiejar.CookieJar* のメソッド), 1686
make_dataclass() (*dataclasses* モジュール), 2208
make_file() (*difflib.HtmlDiff* のメソッド), 164
MAKE_FUNCTION (*opcode*), 2373
make_header() (*email.header* モジュール), 1393
make_legacy_pyc() (*test.support* モジュール), 2071
make_mixed() (*email.message.EmailMessage* のメソッド), 1338
make_msgid() (*email.utils* モジュール), 1398
make_parser() (*xml.sax* モジュール), 1508
make_pkg() (*test.support.script_helper* モジュール), 2086
make_related() (*email.message.EmailMessage* のメソッド), 1337
make_script() (*test.support.script_helper* モジュール), 2086
make_server() (*wsgiref.simple_server* モジュール), 1554
make_table() (*difflib.HtmlDiff* のメソッド), 165
make_zip_pkg() (*test.support.script_helper* モジュール), 2086
make_zip_script() (*test.support.script_helper* モジュール), 2086
makedev() (*os* モジュール), 735
makedirs() (*os* モジュール), 734
makeelement() (*xml.etree.ElementTree.Element* のメソッド), 1478
makefile() (*socket.socket* のメソッド), 1230
makeLogRecord() (*logging* モジュール), 860
makePickle() (*logging.handlers.SocketHandler* のメソッド), 883
makeRecord() (*logging.Logger* のメソッド), 848
makeSocket() (*logging.handlers.DatagramHandler* のメソッド), 884
makeSocket() (*logging.handlers.SocketHandler* のメソッド), 883
maketrans() (*bytearray* の静的メソッド), 73
maketrans() (*bytes* の静的メソッド), 73
maketrans() (*str* の静的メソッド), 61
mangle_from_ (*email.policy.Compat32* の属性), 1356
mangle_from_ (*email.policy.Policy* の属性), 1350
map (*2to3 fixer*), 2063
map() (*concurrent.futures.Executor* のメソッド), 1054
map() (*multiprocessing.pool.Pool* のメソッド), 1029
map() (*tkinter.ttk.Style* のメソッド), 1865
map() (組み込み関数), 18
MAP_ADD (*opcode*), 2366
map_async() (*multiprocessing.pool.Pool* のメソッド), 1030
map_table_b2() (*stringprep* モジュール), 184
map_table_b3() (*stringprep* モジュール), 184
map_to_type() (*email.headerregistry.HeaderRegistry* のメソッド), 1364
mapLogRecord() (*logging.handlers.HTTPHandler* のメソッド), 890
mapping, **2486**
    types, operations on, 97
    オブジェクト, 97
Mapping (*collections.abc* のクラス), 299
Mapping (*typing* のクラス), 1904
mapping() (*msilib.Control* のメソッド), 2391
MappingProxyType (*types* のクラス), 326
MappingView (*collections.abc* のクラス), 299
MappingView (*typing* のクラス), 1905
mapPriority() (*logging.handlers.SysLogHandler* のメソッド), 887
maps (*collections.ChainMap* の属性), 275
maps() (*nis* モジュール), 2430
marshal (モジュール), 561
marshalling
    objects, 535
masking
    operations, 41
Match (*typing* のクラス), 1908
match() (*nis* モジュール), 2430
match() (*pathlib.PurePath* のメソッド), 480
match() (*re* モジュール), 148
match() (*re.Pattern* のメソッド), 152
match_hostname() (*ssl* モジュール), 1247
match_test() (*test.support* モジュール), 2071
match_value() (*test.support.Matcher* のメソッド), 2084
Matcher (*test.support* のクラス), 2084
matches() (*test.support.Matcher* のメソッド), 2084
math
    モジュール, 40, 376
math (モジュール), 365
matmul() (*operator* モジュール), 463
max
    組み込み関数, 47
max (*datetime.date* の属性), 233
max (*datetime.datetime* の属性), 242
max (*datetime.time* の属性), 252
max (*datetime.timedelta* の属性), 229
max() (*audioop* モジュール), 1729
max() (*decimal.Context* のメソッド), 396
max() (*decimal.Decimal* のメソッド), 388
max() (組み込み関数), 18
max_count (*email.headerregistry.BaseHeader* の属性), 1359
MAX_EMAX (*decimal* モジュール), 399
MAX_INTERPOLATION_DEPTH (*configparser* モジュール), 676
max_line_length (*email.policy.Policy* の属性), 1350
max_lines (*textwrap.TextWrapper* の属性), 180
max_mag() (*decimal.Context* のメソッド), 396
max_mag() (*decimal.Decimal* のメソッド), 388
max_memuse (*test.support* モジュール), 2070
MAX_PREC (*decimal* モジュール), 399
max_prefixlen (*ipaddress.IPv4Address* の属性), 1713
max_prefixlen (*ipaddress.IPv4Network* の属性), 1718
max_prefixlen (*ipaddress.IPv6Address* の属性), 1715
max_prefixlen (*ipaddress.IPv6Network* の属性), 1721
MAX_Py_ssize_t (*test.support* モジュール), 2070
maxarray (*reprlib.Repr* の属性), 337
maxdeque (*reprlib.Repr* の属性), 337
maxdict (*reprlib.Repr* の属性), 337
maxDiff (*unittest.TestCase* の属性), 1969
maxfrozenset (*reprlib.Repr* の属性), 337
MAXIMUM_SUPPORTED (*ssl.TLSVersion* の属性), 1258
maximum_version (*ssl.SSLContext* の属性), 1273
maxlen (*collections.deque* の属性), 283
maxlevel (*reprlib.Repr* の属性), 337
maxlist (*reprlib.Repr* の属性), 337
maxlong (*reprlib.Repr* の属性), 337

maxother (*reprlib.Repr* の属性), 337
maxpp() (*audioop* モジュール), 1729
maxset (*reprlib.Repr* の属性), 337
maxsize (*asyncio.Queue* の属性), 1138
maxsize (*sys* モジュール), 2179
maxstring (*reprlib.Repr* の属性), 337
maxtuple (*reprlib.Repr* の属性), 337
maxunicode (*sys* モジュール), 2179
MAXYEAR (*datetime* モジュール), 226
MB_ICONASTERISK (*winsound* モジュール), 2408
MB_ICONEXCLAMATION (*winsound* モジュール), 2408
MB_ICONHAND (*winsound* モジュール), 2408
MB_ICONQUESTION (*winsound* モジュール), 2408
MB_OK (*winsound* モジュール), 2408
mbox (*mailbox* のクラス), 1423
mboxMessage (*mailbox* のクラス), 1430
mean (*statistics.NormalDist* の属性), 429
mean() (*statistics* モジュール), 422
median (*statistics.NormalDist* の属性), 430
median() (*statistics* モジュール), 424
median_grouped() (*statistics* モジュール), 425
median_high() (*statistics* モジュール), 424
median_low() (*statistics* モジュール), 424
MemberDescriptorType (*types* モジュール), 325
memfd_create() (*os* モジュール), 752
memmove() (*ctypes* モジュール), 967
--memo
    pickletools command line option, 2376
MemoryBIO (*ssl* のクラス), 1284
MemoryError, 115
MemoryHandler (*logging.handlers* のクラス), 889
memoryview
    オブジェクト, 68
memoryview (組み込みクラス), 86
memset() (*ctypes* モジュール), 967
merge() (*heapq* モジュール), 302
Message (*email.message* のクラス), 1376
Message (*mailbox* のクラス), 1427
message digest, MD5, 687
message_factory (*email.policy.Policy* の属性), 1351
message_from_binary_file() (*email* モジュール), 1343
message_from_bytes() (*email* モジュール), 1342
message_from_file() (*email* モジュール), 1343
message_from_string() (*email* モジュール), 1343
MessageBeep() (*winsound* モジュール), 2407
MessageClass (*http.server.BaseHTTPRequestHandler* の属性), 1673
MessageError, 1357
MessageParseError, 1357
messages (*xml.parsers.expat.errors* モジュール), 1532
meta path finder, **2486**
meta() (*curses* モジュール), 899
meta_path (*sys* モジュール), 2179
metaclass, **2486**
metaclass (*2to3 fixer*), 2063
MetaPathFinder (*importlib.abc* のクラス), 2293
metavar (*optparse.Option* の属性), 2450
MetavarTypeHelpFormatter (*argparse* のクラス), 809
Meter (*tkinter.tix* のクラス), 1870
method, **2487**
    magic, 2486
    special, 2491
    オブジェクト, 105
method (*urllib.request.Request* の属性), 1570
method resolution order, **2487**
METHOD_BLOWFISH (*crypt* モジュール), 2414
method_calls (*unittest.mock.Mock* の属性), 1997
METHOD_CRYPT (*crypt* モジュール), 2414
METHOD_MD5 (*crypt* モジュール), 2414
METHOD_SHA256 (*crypt* モジュール), 2414
METHOD_SHA512 (*crypt* モジュール), 2414
methodattrs (*2to3 fixer*), 2063
methodcaller() (*operator* モジュール), 466
MethodDescriptorType (*types* モジュール), 324
methodHelp() (*xmlrpc.client.ServerProxy.system* のメソッド), 1697
methods
    bytearray, 71
    bytes, 71
    string, 56
methods (*crypt* モジュール), 2415
methods (*pyclbr.Class* の属性), 2350
methodSignature() (*xmlrpc.client.ServerProxy.system* のメソッド), 1697
MethodType (*types* モジュール), 324
MethodWrapperType (*types* モジュール), 324
MFD_ALLOW_SEALING (*os* モジュール), 752
MFD_CLOEXEC (*os* モジュール), 752
MFD_HUGE_1GB (*os* モジュール), 752
MFD_HUGE_1MB (*os* モジュール), 752
MFD_HUGE_2GB (*os* モジュール), 752
MFD_HUGE_2MB (*os* モジュール), 752
MFD_HUGE_8MB (*os* モジュール), 752
MFD_HUGE_16GB (*os* モジュール), 752
MFD_HUGE_16MB (*os* モジュール), 752
MFD_HUGE_32MB (*os* モジュール), 752
MFD_HUGE_64KB (*os* モジュール), 752
MFD_HUGE_256MB (*os* モジュール), 752
MFD_HUGE_512KB (*os* モジュール), 752
MFD_HUGE_512MB (*os* モジュール), 752
MFD_HUGE_MASK (*os* モジュール), 752
MFD_HUGE_SHIFT (*os* モジュール), 752
MFD_HUGETLB (*os* モジュール), 752
MH (*mailbox* のクラス), 1423
MHMessage (*mailbox* のクラス), 1432
microsecond (*datetime.datetime* の属性), 242
microsecond (*datetime.time* の属性), 252
MIME
    base64 encoding, 1442
    content type, 1438
    headers, 1439, 1540
    quoted-printable encoding, 1450
MIMEApplication (*email.mime.application* のクラス), 1388
MIMEAudio (*email.mime.audio* のクラス), 1389
MIMEBase (*email.mime.base* のクラス), 1387
MIMEImage (*email.mime.image* のクラス), 1389
MIMEMessage (*email.mime.message* のクラス), 1390
MIMEMultipart (*email.mime.multipart* のクラス), 1388
MIMENonMultipart (*email.mime.nonmultipart* のクラス), 1388
MIMEPart (*email.message* のクラス), 1339
MIMEText (*email.mime.text* のクラス), 1390
MimeTypes (*mimetypes* のクラス), 1441
mimetypes (モジュール), 1438
MIMEVersionHeader (*email.headerregistry* のクラス), 1362
min
    組み込み関数, 47
min (*datetime.date* の属性), 233
min (*datetime.datetime* の属性), 242
min (*datetime.time* の属性), 252
min (*datetime.timedelta* の属性), 229
min() (*decimal.Context* のメソッド), 396
min() (*decimal.Decimal* のメソッド), 388
min() (組み込み関数), 19
MIN_EMIN (*decimal* モジュール), 399
MIN_ETINY (*decimal* モジュール), 399
min_mag() (*decimal.Context* のメソッド), 396
min_mag() (*decimal.Decimal* のメソッド), 388
MINEQUAL (*token* モジュール), 2340
MINIMUM_SUPPORTED (*ssl.TLSVersion* の属性), 1258
minimum_version (*ssl.SSLContext* の属性), 1273
minmax() (*audioop* モジュール), 1729
minor (*email.headerregistry.MIMEVersionHeader* の属性), 1362
minor() (*os* モジュール), 735
MINUS (*token* モジュール), 2339

minus() (*decimal.Context* のメソッド), 396
minute (*datetime.datetime* の属性), 242
minute (*datetime.time* の属性), 252
MINYEAR (*datetime* モジュール), 226
mirrored() (*unicodedata* モジュール), 181
misc_header (*cmd.Cmd* の属性), 1819
--missing
    trace command line option, 2127
MISSING (*contextvars.Token* の属性), 1093
MISSING_C_DOCSTRINGS (*test.support* モジュール), 2070
missing_compiler_executable() (*test.support* モジュール), 2082
MissingSectionHeaderError, 678
MIXERDEV, 1746
mkd() (*ftplib.FTP* のメソッド), 1618
mkdir() (*os* モジュール), 733
mkdir() (*pathlib.Path* のメソッド), 486
mkdtemp() (*tempfile* モジュール), 513
mkfifo() (*os* モジュール), 734
mknod() (*os* モジュール), 735
mksalt() (*crypt* モジュール), 2415
mkstemp() (*tempfile* モジュール), 512
mktemp() (*tempfile* モジュール), 515
mktime() (*time* モジュール), 794
mktime_tz() (*email.utils* モジュール), 1400
mlsd() (*ftplib.FTP* のメソッド), 1617
mmap (*mmap* のクラス), 1321
mmap (モジュール), 1320
MMDF (*mailbox* のクラス), 1426
MMDFMessage (*mailbox* のクラス), 1435
Mock (*unittest.mock* のクラス), 1990
mock_add_spec() (*unittest.mock.Mock* のメソッド), 1993
mock_calls (*unittest.mock.Mock* の属性), 1998
mock_open() (*unittest.mock* モジュール), 2029
mod() (*operator* モジュール), 463
mode (*io.FileIO* の属性), 783
mode (*ossaudiodev.oss_audio_device* の属性), 1750
mode (*statistics.NormalDist* の属性), 430
mode (*tarfile.TarInfo* の属性), 638
mode() (*statistics* モジュール), 425
mode() (*turtle* モジュール), 1806
modes
    file, 20
modf() (*math* モジュール), 367
modified() (*urllib.robotparser.RobotFileParser* のメソッド), 1600
Modify() (*msilib.View* のメソッド), 2387
modify() (*select.devpoll* のメソッド), 1291
modify() (*select.epoll* のメソッド), 1293
modify() (*selectors.BaseSelector* のメソッド), 1299
modify() (*select.poll* のメソッド), 1294
module, **2487**
    search path, 519, 2180, 2268
module (*pyclbr.Class* の属性), 2350
module (*pyclbr.Function* の属性), 2349
module spec, **2487**
module_for_loader() (*importlib.util* モジュール), 2311
module_from_spec() (*importlib.util* モジュール), 2311
module_repr() (*importlib.abc.Loader* のメソッド), 2296
ModuleFinder (*modulefinder* のクラス), 2285
modulefinder (モジュール), 2285
ModuleInfo (*pkgutil* のクラス), 2282
ModuleNotFoundError, 115
modules (*modulefinder.ModuleFinder* の属性), 2286
modules (*sys* モジュール), 2180
modules_cleanup() (*test.support* モジュール), 2079
modules_setup() (*test.support* モジュール), 2079
ModuleSpec (*importlib.machinery* のクラス), 2308
ModuleType (*types* のクラス), 324
monotonic() (*time* モジュール), 794
monotonic_ns() (*time* モジュール), 794
month (*datetime.date* の属性), 234
month (*datetime.datetime* の属性), 242
month() (*calendar* モジュール), 274
month_abbr (*calendar* モジュール), 274
month_name (*calendar* モジュール), 274
monthcalendar() (*calendar* モジュール), 273
monthdatescalendar() (*calendar.Calendar* のメソッド), 270
monthdays2calendar() (*calendar.Calendar* のメソッド), 270
monthdayscalendar() (*calendar.Calendar* のメソッド), 270
monthrange() (*calendar* モジュール), 273
Morsel (*http.cookies* のクラス), 1680
most_common() (*collections.Counter* のメソッド), 279
mouseinterval() (*curses* モジュール), 899
mousemask() (*curses* モジュール), 899
move() (*curses.panel.Panel* のメソッド), 923
move() (*curses.window* のメソッド), 908
move() (*mmap.mmap* のメソッド), 1323
move() (*shutil* モジュール), 524
move() (*tkinter.ttk.Treeview* のメソッド), 1863
move_to_end() (*collections.OrderedDict* のメソッド), 292
MozillaCookieJar (*http.cookiejar* のクラス), 1687
MRO, **2487**
mro() (*class* のメソッド), 108
msg (*http.client.HTTPResponse* の属性), 1610
msg (*json.JSONDecodeError* の属性), 1411
msg (*re.error* の属性), 151
msg (*traceback.TracebackException* の属性), 2238
msg() (*telnetlib.Telnet* のメソッド), 1654
msi, 2385
msilib (モジュール), 2385
msvcrt (モジュール), 2392
mt_interact() (*telnetlib.Telnet* のメソッド), 1655
mtime (*gzip.GzipFile* の属性), 602
mtime (*tarfile.TarInfo* の属性), 638
mtime() (*urllib.robotparser.RobotFileParser* のメソッド), 1600
mul() (*audioop* モジュール), 1729
mul() (*operator* モジュール), 463
MultiCall (*xmlrpc.client* のクラス), 1701
MULTILINE (*re* モジュール), 147
MultiLoopChildWatcher (*asyncio* のクラス), 1193
multimode() (*statistics* モジュール), 426
MultipartConversionError, 1357
multiply() (*decimal.Context* のメソッド), 396
multiprocessing (モジュール), 994
multiprocessing.connection (モジュール), 1032
multiprocessing.dummy (モジュール), 1037
multiprocessing.Manager() (*multiprocessing.sharedctypes* モジュール), 1020
multiprocessing.managers (モジュール), 1020
multiprocessing.pool (モジュール), 1028
multiprocessing.shared_memory (モジュール), 1048
multiprocessing.sharedctypes (モジュール), 1018
mutable, **2487**
    sequence types, 50
MutableMapping (*collections.abc* のクラス), 299
MutableMapping (*typing* のクラス), 1904
MutableSequence (*collections.abc* のクラス), 298
MutableSequence (*typing* のクラス), 1905
MutableSet (*collections.abc* のクラス), 299
MutableSet (*typing* のクラス), 1904
mvderwin() (*curses.window* のメソッド), 908
mvwin() (*curses.window* のメソッド), 908
myrights() (*imaplib.IMAP4* のメソッド), 1627

N

-n N
    timeit command line option, 2123
N_TOKENS (*token* モジュール), 2341
n_waiting (*threading.Barrier* の属性), 993
name (*codecs.CodecInfo* の属性), 201
name (*contextvars.ContextVar* の属性), 1092

name (*doctest.DocTest* の属性), 1938
name (*email.headerregistry.BaseHeader* の属性), 1359
name (*hashlib.hash* の属性), 689
name (*hmac.HMAC* の属性), 701
name (*http.cookiejar.Cookie* の属性), 1692
name (*importlib.abc.FileLoader* の属性), 2299
name (*importlib.machinery.ExtensionFileLoader* の属性), 2307
name (*importlib.machinery.ModuleSpec* の属性), 2308
name (*importlib.machinery.SourceFileLoader* の属性), 2306
name (*importlib.machinery.SourcelessFileLoader* の属性), 2307
name (*inspect.Parameter* の属性), 2257
name (*io.FileIO* の属性), 783
name (*multiprocessing.Process* の属性), 1003
name (*multiprocessing.shared_memory.SharedMemory* の属性), 1049
name (*os* モジュール), 706
name (*os.DirEntry* の属性), 740
name (*ossaudiodev.oss_audio_device* の属性), 1750
name (*pyclbr.Class* の属性), 2350
name (*pyclbr.Function* の属性), 2349
name (*tarfile.TarInfo* の属性), 637
name (*threading.Thread* の属性), 982
NAME (*token* モジュール), 2338
name (*xml.dom.Attr* の属性), 1495
name (*xml.dom.DocumentType* の属性), 1493
name (*zipfile.Path* の属性), 623
name() (*unicodedata* モジュール), 181
name2codepoint (*html.entities* モジュール), 1459
Named Shared Memory, 1048
named tuple, **2487**
NamedTemporaryFile() (*tempfile* モジュール), 511
NamedTuple (*typing* のクラス), 1909
namedtuple() (*collections* モジュール), 288
NameError, 116
namelist() (*zipfile.ZipFile* のメソッド), 619
nameprep() (*encodings.idna* モジュール), 223
namer (*logging.handlers.BaseRotatingHandler* の属性), 879
namereplace_errors() (*codecs* モジュール), 206
namespace, **2487**
Namespace (*argparse* のクラス), 830
Namespace (*multiprocessing.managers* のクラス), 1023
namespace package, **2488**
namespace() (*imaplib.IMAP4* のメソッド), 1627
Namespace() (*multiprocessing.managers.SyncManager* のメソッド), 1023
NAMESPACE_DNS (*uuid* モジュール), 1659
NAMESPACE_OID (*uuid* モジュール), 1659
NAMESPACE_URL (*uuid* モジュール), 1659
NAMESPACE_X500 (*uuid* モジュール), 1659
NamespaceErr, 1498
namespaceURI (*xml.dom.Node* の属性), 1491
NaN, 13
nan (*cmath* モジュール), 376
nan (*math* モジュール), 372
nanj (*cmath* モジュール), 376
NannyNag, 2348
napms() (*curses* モジュール), 899
nargs (*optparse.Option* の属性), 2450
native_id (*threading.Thread* の属性), 982
nbytes (*memoryview* の属性), 92
ncurses_version (*curses* モジュール), 911
ndiff() (*difflib* モジュール), 166
ndim (*memoryview* の属性), 93
ne (*2to3 fixer*), 2063
ne() (*operator* モジュール), 462
needs_input (*bz2.BZ2Decompressor* の属性), 607
needs_input (*lzma.LZMADecompressor* の属性), 613
neg() (*operator* モジュール), 463
nested scope, **2488**
netmask (*ipaddress.IPv4Network* の属性), 1718
netmask (*ipaddress.IPv6Network* の属性), 1722
NetmaskValueError, 1726
netrc (*netrc* のクラス), 678
netrc (モジュール), 678
NetrcParseError, 678
netscape (*http.cookiejar.CookiePolicy* の属性), 1689
network (*ipaddress.IPv4Interface* の属性), 1724
network (*ipaddress.IPv6Interface* の属性), 1725
Network News Transfer Protocol, 1631
network_address (*ipaddress.IPv4Network* の属性), 1718
network_address (*ipaddress.IPv6Network* の属性), 1722
new() (*hashlib* モジュール), 688
new() (*hmac* モジュール), 700
new-style class, **2488**
new_alignment() (*formatter.writer* のメソッド), 2382
new_child() (*collections.ChainMap* のメソッド), 276
new_class() (*types* モジュール), 322
new_event_loop() (*asyncio* モジュール), 1143
new_event_loop() (*asyncio.AbstractEventLoopPolicy* のメソッド), 1191
new_font() (*formatter.writer* のメソッド), 2382
new_margin() (*formatter.writer* のメソッド), 2382
new_module() (*imp* モジュール), 2468
new_panel() (*curses.panel* モジュール), 922
new_spacing() (*formatter.writer* のメソッド), 2382
new_styles() (*formatter.writer* のメソッド), 2382
newgroups() (*nntplib.NNTP* のメソッド), 1635
NEWLINE (*token* モジュール), 2338
newlines (*io.TextIOBase* の属性), 785
newnews() (*nntplib.NNTP* のメソッド), 1635
newpad() (*curses* モジュール), 899
NewType() (*typing* モジュール), 1911
newwin() (*curses* モジュール), 899
next (*2to3 fixer*), 2063
next (*pdb command*), 2107
next() (*nntplib.NNTP* のメソッド), 1637
next() (*tarfile.TarFile* のメソッド), 634
next() (*tkinter.ttk.Treeview* のメソッド), 1863
next() (組み込み関数), 19
next_minus() (*decimal.Context* のメソッド), 397
next_minus() (*decimal.Decimal* のメソッド), 388
next_plus() (*decimal.Context* のメソッド), 397
next_plus() (*decimal.Decimal* のメソッド), 388
next_toward() (*decimal.Context* のメソッド), 397
next_toward() (*decimal.Decimal* のメソッド), 388
nextfile() (*fileinput* モジュール), 499
nextkey() (*dbm.gnu.gdbm* のメソッド), 565
nextSibling (*xml.dom.Node* の属性), 1490
ngettext() (*gettext* モジュール), 1754
ngettext() (*gettext.GNUTranslations* のメソッド), 1759
ngettext() (*gettext.NullTranslations* のメソッド), 1757
nice() (*os* モジュール), 759
nis (モジュール), 2430
NL (*token* モジュール), 2341
nl() (*curses* モジュール), 899
nl_langinfo() (*locale* モジュール), 1767
nlargest() (*heapq* モジュール), 302
nlst() (*ftplib.FTP* のメソッド), 1617
NNTP
    protocol, 1631
NNTP (*nntplib* のクラス), 1632
nntp_implementation (*nntplib.NNTP* の属性), 1633
NNTP_SSL (*nntplib* のクラス), 1632
nntp_version (*nntplib.NNTP* の属性), 1633
NNTPDataError, 1633
NNTPError, 1633
nntplib (モジュール), 1631
NNTPPermanentError, 1633
NNTPProtocolError, 1633
NNTPReplyError, 1633
NNTPTemporaryError, 1633
no_proxy, 1568
no_tracing() (*test.support* モジュール), 2077
no_type_check() (*typing* モジュール), 1912

no_type_check_decorator() (*typing* モジュール), 1912
nocbreak() (*curses* モジュール), 899
NoDataAllowedErr, 1498
node() (*platform* モジュール), 924
nodelay() (*curses.window* のメソッド), 908
nodeName (*xml.dom.Node* の属性), 1491
NodeTransformer (*ast* のクラス), 2333
nodeType (*xml.dom.Node* の属性), 1490
nodeValue (*xml.dom.Node* の属性), 1491
NodeVisitor (*ast* のクラス), 2332
noecho() (*curses* モジュール), 899
NOEXPR (*locale* モジュール), 1768
NoModificationAllowedErr, 1498
nonblock() (*ossaudiodev.oss_audio_device* のメソッド), 1748
NonCallableMagicMock (*unittest.mock* のクラス), 2022
NonCallableMock (*unittest.mock* のクラス), 1999
None (*Built-in object*), 37
None (組み込み変数), 35
nonl() (*curses* モジュール), 900
nonzero (*2to3 fixer*), 2063
noop() (*imaplib.IMAP4* のメソッド), 1627
noop() (*poplib.POP3* のメソッド), 1621
NoOptionError, 677
NOP (*opcode*), 2362
noqiflush() (*curses* モジュール), 900
noraw() (*curses* モジュール), 900
--no-report
    trace command line option, 2127
NoReturn (*typing* モジュール), 1913
NORMAL_PRIORITY_CLASS (*subprocess* モジュール), 1077
NormalDist (*statistics* のクラス), 429
normalize() (*decimal.Context* のメソッド), 397
normalize() (*decimal.Decimal* のメソッド), 388
normalize() (*locale* モジュール), 1770
normalize() (*unicodedata* モジュール), 182
normalize() (*xml.dom.Node* のメソッド), 1492
NORMALIZE_WHITESPACE (*doctest* モジュール), 1928
normalvariate() (*random* モジュール), 417
normcase() (*os.path* モジュール), 495
normpath() (*os.path* モジュール), 495
NoSectionError, 677
NoSuchMailboxError, 1437
not
    演算子, 38
not in
    演算子, 39, 47
not_() (*operator* モジュール), 462
NotADirectoryError, 121
notationDecl() (*xml.sax.handler.DTDHandler* のメソッド), 1516
NotationDeclHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1529
notations (*xml.dom.DocumentType* の属性), 1493
NotConnected, 1606
NoteBook (*tkinter.tix* のクラス), 1872
Notebook (*tkinter.ttk* のクラス), 1854
NotEmptyError, 1437
NOTEQUAL (*token* モジュール), 2339
NotFoundErr, 1498
notify() (*asyncio.Condition* のメソッド), 1129
notify() (*threading.Condition* のメソッド), 988
notify_all() (*asyncio.Condition* のメソッド), 1130
notify_all() (*threading.Condition* のメソッド), 988
notimeout() (*curses.window* のメソッド), 909
NotImplemented (組み込み変数), 35
NotImplementedError, 116
NotStandaloneHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1530
NotSupportedErr, 1498
NotSupportedError, 586
noutrefresh() (*curses.window* のメソッド), 909
now() (*datetime.datetime* のクラスメソッド), 239
npgettext() (*gettext* モジュール), 1754
npgettext() (*gettext.GNUTranslations* のメソッド), 1760
npgettext() (*gettext.NullTranslations* のメソッド), 1757
NSIG (*signal* モジュール), 1313
nsmallest() (*heapq* モジュール), 302
NT_OFFSET (*token* モジュール), 2341
NTEventLogHandler (*logging.handlers* のクラス), 887
ntohl() (*socket* モジュール), 1224
ntohs() (*socket* モジュール), 1224
ntransfercmd() (*ftplib.FTP* のメソッド), 1617
nullcontext() (*contextlib* モジュール), 2215
NullFormatter (*formatter* のクラス), 2382
NullHandler (*logging* のクラス), 878
NullImporter (*imp* のクラス), 2472
NullTranslations (*gettext* のクラス), 1757
NullWriter (*formatter* のクラス), 2383
num_addresses (*ipaddress.IPv4Network* の属性), 1719
num_addresses (*ipaddress.IPv6Network* の属性), 1722
num_tickets (*ssl.SSLContext* の属性), 1273
Number (*numbers* のクラス), 361
NUMBER (*token* モジュール), 2338
--number=N
    timeit command line option, 2123
number_class() (*decimal.Context* のメソッド), 397
number_class() (*decimal.Decimal* のメソッド), 388
numbers (モジュール), 361
numerator (*fractions.Fraction* の属性), 411
numerator (*numbers.Rational* の属性), 362
numeric
    conversions, 40
    literals, 39
    object, 38
    types, operations on, 40
    オブジェクト, 39
numeric() (*unicodedata* モジュール), 181
Numerical Python, 27
numinput() (*turtle* モジュール), 1806
numliterals (*2to3 fixer*), 2063

O

-o
    pickletools command line option, 2376
-o <output>
    zipapp command line option, 2157
O_APPEND (*os* モジュール), 719
O_ASYNC (*os* モジュール), 719
O_BINARY (*os* モジュール), 719
O_CLOEXEC (*os* モジュール), 719
O_CREAT (*os* モジュール), 719
O_DIRECT (*os* モジュール), 719
O_DIRECTORY (*os* モジュール), 719
O_DSYNC (*os* モジュール), 719
O_EXCL (*os* モジュール), 719
O_EXLOCK (*os* モジュール), 719
O_NDELAY (*os* モジュール), 719
O_NOATIME (*os* モジュール), 719
O_NOCTTY (*os* モジュール), 719
O_NOFOLLOW (*os* モジュール), 719
O_NOINHERIT (*os* モジュール), 719
O_NONBLOCK (*os* モジュール), 719
O_PATH (*os* モジュール), 719
O_RANDOM (*os* モジュール), 719
O_RDONLY (*os* モジュール), 719
O_RDWR (*os* モジュール), 719
O_RSYNC (*os* モジュール), 719
O_SEQUENTIAL (*os* モジュール), 719
O_SHLOCK (*os* モジュール), 719
O_SHORT_LIVED (*os* モジュール), 719
O_SYNC (*os* モジュール), 719
O_TEMPORARY (*os* モジュール), 719
O_TEXT (*os* モジュール), 719
O_TMPFILE (*os* モジュール), 719

O_TRUNC (*os* モジュール), 719
O_WRONLY (*os* モジュール), 719
obj (*memoryview* の属性), 92
object, **2488**
    code, 105, 561
    numeric, 38
object (*UnicodeError* の属性), 120
object (組み込みクラス), 19
objects
    comparing, 38
    flattening, 535
    marshalling, 535
    persistent, 535
    pickling, 535
    serializing, 535
obufcount() (*ossaudiodev.oss_audio_device* のメソッド), 1749
obuffree() (*ossaudiodev.oss_audio_device* のメソッド), 1749
oct() (組み込み関数), 20
octal
    literals, 39
octdigits (*string* モジュール), 125
offset (*SyntaxError* の属性), 118
offset (*traceback.TracebackException* の属性), 2238
offset (*xml.parsers.expat.ExpatError* の属性), 1530
OK (*curses* モジュール), 911
old_value (*contextvars.Token* の属性), 1093
OleDLL (*ctypes* のクラス), 958
onclick() (*turtle* モジュール), 1804
ondrag() (*turtle* モジュール), 1798
onecmd() (*cmd.Cmd* のメソッド), 1817
onkey() (*turtle* モジュール), 1804
onkeypress() (*turtle* モジュール), 1804
onkeyrelease() (*turtle* モジュール), 1804
onrelease() (*turtle* モジュール), 1798
onscreenclick() (*turtle* モジュール), 1804
ontimer() (*turtle* モジュール), 1805
OP (*token* モジュール), 2341
OP_ALL (*ssl* モジュール), 1254
OP_CIPHER_SERVER_PREFERENCE (*ssl* モジュール), 1255
OP_ENABLE_MIDDLEBOX_COMPAT (*ssl* モジュール), 1255
OP_IGNORE_UNEXPECTED_EOF (*ssl* モジュール), 1256
OP_NO_COMPRESSION (*ssl* モジュール), 1255
OP_NO_RENEGOTIATION (*ssl* モジュール), 1255
OP_NO_SSLv2 (*ssl* モジュール), 1254
OP_NO_SSLv3 (*ssl* モジュール), 1254
OP_NO_TICKET (*ssl* モジュール), 1256
OP_NO_TLSv1 (*ssl* モジュール), 1254
OP_NO_TLSv1_1 (*ssl* モジュール), 1254
OP_NO_TLSv1_2 (*ssl* モジュール), 1254
OP_NO_TLSv1_3 (*ssl* モジュール), 1255
OP_SINGLE_DH_USE (*ssl* モジュール), 1255
OP_SINGLE_ECDH_USE (*ssl* モジュール), 1255
open() (*aifc* モジュール), 1731
open() (*bz2* モジュール), 604
open() (*codecs* モジュール), 202
open() (*dbm* モジュール), 563
open() (*dbm.dumb* モジュール), 567
open() (*dbm.gnu* モジュール), 564
open() (*dbm.ndbm* モジュール), 566
open() (*gzip* モジュール), 600
open() (*imaplib.IMAP4* のメソッド), 1627
open() (*io* モジュール), 775
open() (*lzma* モジュール), 609
open() (*os* モジュール), 718
open() (*ossaudiodev* モジュール), 1746
open() (*pathlib.Path* のメソッド), 486
open() (*pipes.Template* のメソッド), 2424
open() (*shelve* モジュール), 557
open() (*sunau* モジュール), 1734
open() (*tarfile* モジュール), 629
open() (*tarfile.TarFile* のクラスメソッド), 633
open() (*telnetlib.Telnet* のメソッド), 1654
open() (*tokenize* モジュール), 2344
open() (*urllib.request.OpenerDirector* のメソッド), 1572
open() (*urllib.request.URLopener* のメソッド), 1585
open() (*wave* モジュール), 1738
open() (*webbrowser* モジュール), 1538
open() (*webbrowser.controller* のメソッド), 1540
open() (組み込み関数), 20
open() (*zipfile.Path* のメソッド), 623
open() (*zipfile.ZipFile* のメソッド), 619
open_binary() (*importlib.resources* モジュール), 2302
open_code() (*io* モジュール), 775
open_connection() (*asyncio* モジュール), 1119
open_new() (*webbrowser* モジュール), 1538
open_new() (*webbrowser.controller* のメソッド), 1540
open_new_tab() (*webbrowser* モジュール), 1538
open_new_tab() (*webbrowser.controller* のメソッド), 1540
open_osfhandle() (*msvcrt* モジュール), 2393
open_resource() (*importlib.abc.ResourceReader* のメソッド), 2297
open_text() (*importlib.resources* モジュール), 2302
open_unix_connection() (*asyncio* モジュール), 1120
open_unknown() (*urllib.request.URLopener* のメソッド), 1585
open_urlresource() (*test.support* モジュール), 2078
OpenDatabase() (*msilib* モジュール), 2385
OpenerDirector (*urllib.request* のクラス), 1567
openfp() (*sunau* モジュール), 1735
openfp() (*wave* モジュール), 1738
OpenKey() (*winreg* モジュール), 2399
OpenKeyEx() (*winreg* モジュール), 2399
openlog() (*syslog* モジュール), 2431
openmixer() (*ossaudiodev* モジュール), 1746
openpty() (*os* モジュール), 720
openpty() (*pty* モジュール), 2418
OpenSSL
    (use in module hashlib), 688
    (use in module ssl), 1242
OPENSSL_VERSION (*ssl* モジュール), 1257
OPENSSL_VERSION_INFO (*ssl* モジュール), 1257
OPENSSL_VERSION_NUMBER (*ssl* モジュール), 1257
OpenView() (*msilib.Database* のメソッド), 2387
operation
    concatenation, 47
    repetition, 47
    slice, 47
    subscript, 47
OperationalError, 586
operations
    bitwise, 41
    Boolean, 37, 38
    masking, 41
    shifting, 41
operations on
    dictionary type, 97
    integer types, 41
    list type, 50
    mapping types, 97
    numeric types, 40
    sequence types, 47, 50
operator
    - (*minus*), 39
    + (*plus*), 39
    comparison, 38
operator (*2to3 fixer*), 2063
operator (モジュール), 462
opmap (*dis* モジュール), 2374
opname (*dis* モジュール), 2374
optim_args_from_interpreter_flags() (*test.support* モジュール), 2074
optimize() (*pickletools* モジュール), 2376
OPTIMIZED_BYTECODE_SUFFIXES (*importlib.machinery* モジュール), 2304

Optional (*typing* モジュール), 1914
OptionGroup (*optparse* のクラス), 2442
OptionMenu (*tkinter.tix* のクラス), 1870
OptionParser (*optparse* のクラス), 2446
options (*doctest.Example* の属性), 1939
Options (*ssl* のクラス), 1256
options (*ssl.SSLContext* の属性), 1274
options() (*configparser.ConfigParser* のメソッド), 673
optionxform() (*configparser.ConfigParser* のメソッド), 675
optparse (モジュール), 2433
or
    演算子, 37, 38
or_() (*operator* モジュール), 464
ord() (組み込み関数), 24
ordered_attributes (*xml.parsers.expat.xmlparser* の属性), 1527
OrderedDict (*collections* のクラス), 292
OrderedDict (*typing* のクラス), 1907
origin (*importlib.machinery.ModuleSpec* の属性), 2309
origin_req_host (*urllib.request.Request* の属性), 1570
origin_server (*wsgiref.handlers.BaseHandler* の属性), 1561
os
    モジュール, 2409
os (モジュール), 705
os_environ (*wsgiref.handlers.BaseHandler* の属性), 1559
OSError, 116
os.path (モジュール), 491
ossaudiodev (モジュール), 1745
OSSAudioError, 1746
outfile
    json.tool command line option, 1414
output (*subprocess.CalledProcessError* の属性), 1065
output (*subprocess.TimeoutExpired* の属性), 1064
output (*unittest.TestCase* の属性), 1966
output() (*http.cookies.BaseCookie* のメソッド), 1679
output() (*http.cookies.Morsel* のメソッド), 1681
--output=<file>
    pickletools command line option, 2376
--output=<output>
    zipapp command line option, 2157
output_charset (*email.charset.Charset* の属性), 1395
output_charset() (*gettext.NullTranslations* のメソッド), 1758
output_codec (*email.charset.Charset* の属性), 1395
output_difference() (*doctest.OutputChecker* のメソッド), 1943
OutputChecker (*doctest* のクラス), 1943
OutputString() (*http.cookies.Morsel* のメソッド), 1681
OutsideDestinationError, 631
over() (*nntplib.NNTP* のメソッド), 1636
Overflow (*decimal* のクラス), 401
OverflowError, 117
overlap() (*statistics.NormalDist* のメソッド), 430
overlaps() (*ipaddress.IPv4Network* のメソッド), 1719
overlaps() (*ipaddress.IPv6Network* のメソッド), 1722
overlay() (*curses.window* のメソッド), 909
overload() (*typing* モジュール), 1911
overwrite() (*curses.window* のメソッド), 909
owner() (*pathlib.Path* のメソッド), 487

P

-p
    pickletools command line option, 2376
    timeit command line option, 2123
    unittest-discover command line option, 1953
p (*pdb command*), 2108
-p <interpreter>
    zipapp command line option, 2157
P_ALL (*os* モジュール), 765
P_DETACH (*os* モジュール), 763
P_NOWAIT (*os* モジュール), 762
P_NOWAITO (*os* モジュール), 762
P_OVERLAY (*os* モジュール), 763
P_PGID (*os* モジュール), 765
P_PID (*os* モジュール), 765
P_WAIT (*os* モジュール), 763
pack() (*mailbox.MH* のメソッド), 1424
pack() (*struct* モジュール), 194
pack() (*struct.Struct* のメソッド), 199
pack_array() (*xdrlib.Packer* のメソッド), 681
pack_bytes() (*xdrlib.Packer* のメソッド), 680
pack_double() (*xdrlib.Packer* のメソッド), 680
pack_farray() (*xdrlib.Packer* のメソッド), 681
pack_float() (*xdrlib.Packer* のメソッド), 680
pack_fopaque() (*xdrlib.Packer* のメソッド), 680
pack_fstring() (*xdrlib.Packer* のメソッド), 680
pack_into() (*struct* モジュール), 194
pack_into() (*struct.Struct* のメソッド), 199
pack_list() (*xdrlib.Packer* のメソッド), 681
pack_opaque() (*xdrlib.Packer* のメソッド), 680
pack_string() (*xdrlib.Packer* のメソッド), 680
package, 2269, **2488**
Package (*importlib.resources* モジュール), 2302
packed (*ipaddress.IPv4Address* の属性), 1713
packed (*ipaddress.IPv6Address* の属性), 1715
Packer (*xdrlib* のクラス), 679
packing
    binary data, 193
packing (*widgets*), 1839
PAGER, 1918
pair_content() (*curses* モジュール), 900
pair_number() (*curses* モジュール), 900
PanedWindow (*tkinter.tix* のクラス), 1872
parameter, **2488**
Parameter (*inspect* のクラス), 2257
ParameterizedMIMEHeader (*email.headerregistry* のクラス), 1362
parameters (*inspect.Signature* の属性), 2256
params (*email.headerregistry.ParameterizedMIMEHeader* の属性), 1362
pardir (*os* モジュール), 770
paren (*2to3 fixer*), 2063
parent (*importlib.machinery.ModuleSpec* の属性), 2309
parent (*pyclbr.Class* の属性), 2350
parent (*pyclbr.Function* の属性), 2349
parent (*urllib.request.BaseHandler* の属性), 1573
parent() (*tkinter.ttk.Treeview* のメソッド), 1863
parent_process() (*multiprocessing* モジュール), 1009
parentNode (*xml.dom.Node* の属性), 1490
parents (*collections.ChainMap* の属性), 276
paretovariate() (*random* モジュール), 417
parse() (*ast* モジュール), 2331
parse() (*cgi* モジュール), 1545
parse() (*doctest.DocTestParser* のメソッド), 1941
parse() (*email.parser.BytesParser* のメソッド), 1341
parse() (*email.parser.Parser* のメソッド), 1342
parse() (*string.Formatter* のメソッド), 126
parse() (*urllib.robotparser.RobotFileParser* のメソッド), 1600
parse() (*xml.dom.minidom* モジュール), 1501
parse() (*xml.dom.pulldom* モジュール), 1507
parse() (*xml.etree.ElementTree* モジュール), 1473
parse() (*xml.etree.ElementTree.ElementTree* のメソッド), 1480
Parse() (*xml.parsers.expat.xmlparser* のメソッド), 1525
parse() (*xml.sax* モジュール), 1509
parse() (*xml.sax.xmlreader.XMLReader* のメソッド), 1519
parse_and_bind() (*readline* モジュール), 186
parse_args() (*argparse.ArgumentParser* のメソッド), 826
PARSE_COLNAMES (*sqlite3* モジュール), 570
parse_config_h() (*sysconfig* モジュール), 2193
PARSE_DECLTYPES (*sqlite3* モジュール), 570
parse_header() (*cgi* モジュール), 1545

parse_headers() (*http.client* モジュール), 1605
parse_intermixed_args() (*argparse.ArgumentParser* のメソッド), 838
parse_known_args() (*argparse.ArgumentParser* のメソッド), 837
parse_known_intermixed_args() (*argparse.ArgumentParser* のメソッド), 838
parse_multipart() (*cgi* モジュール), 1545
parse_qs() (*urllib.parse* モジュール), 1590
parse_qsl() (*urllib.parse* モジュール), 1591
parseaddr() (*email.utils* モジュール), 1398
parsebytes() (*email.parser.BytesParser* のメソッド), 1342
parsedate() (*email.utils* モジュール), 1399
parsedate_to_datetime() (*email.utils* モジュール), 1399
parsedate_tz() (*email.utils* モジュール), 1399
ParseError (*xml.etree.ElementTree* のクラス), 1486
ParseFile() (*xml.parsers.expat.xmlparser* のメソッド), 1525
ParseFlags() (*imaplib* モジュール), 1624
Parser (*email.parser* のクラス), 1342
parser (モジュール), 2321
ParserCreate() (*xml.parsers.expat* モジュール), 1524
ParserError, 2324
ParseResult (*urllib.parse* のクラス), 1595
ParseResultBytes (*urllib.parse* のクラス), 1596
parsestr() (*email.parser.Parser* のメソッド), 1342
parseString() (*xml.dom.minidom* モジュール), 1501
parseString() (*xml.dom.pulldom* モジュール), 1507
parseString() (*xml.sax* モジュール), 1509
parsing
    Python source code, 2321
    URL, 1588
ParsingError, 678
partial (*asyncio.IncompleteReadError* の属性), 1141
partial() (*functools* モジュール), 455
partial() (*imaplib.IMAP4* のメソッド), 1627
partialmethod (*functools* のクラス), 456
parties (*threading.Barrier* の属性), 993
partition() (*bytearray* のメソッド), 73
partition() (*bytes* のメソッド), 73
partition() (*str* のメソッド), 61
pass_() (*poplib.POP3* のメソッド), 1621
Paste, 1880
patch() (*test.support* モジュール), 2082
patch() (*unittest.mock* モジュール), 2009
patch.dict() (*unittest.mock* モジュール), 2013
patch.multiple() (*unittest.mock* モジュール), 2015
patch.object() (*unittest.mock* モジュール), 2013
patch.stopall() (*unittest.mock* モジュール), 2017
PATH, 755, 761, 762, 771, 1537, 1546, 1548, 2269
path
    configuration file, 2269
    module search, 519, 2180, 2268
    operations, 471, 491
path (*http.cookiejar.Cookie* の属性), 1692
path (*http.server.BaseHTTPRequestHandler* の属性), 1672
path (*importlib.abc.FileLoader* の属性), 2299
path (*importlib.machinery.ExtensionFileLoader* の属性), 2308
path (*importlib.machinery.FileFinder* の属性), 2306
path (*importlib.machinery.SourceFileLoader* の属性), 2306
path (*importlib.machinery.SourcelessFileLoader* の属性), 2307
path (*os.DirEntry* の属性), 740
Path (*pathlib* のクラス), 482
path (*sys* モジュール), 2180
Path (*zipfile* のクラス), 622
path based finder, **2489**
Path browser, 1876
path entry, **2489**
path entry finder, **2489**
path entry hook, **2489**
path() (*importlib.resources* モジュール), 2303
path-like object, **2489**
path_hook() (*importlib.machinery.FileFinder* のクラスメソッド), 2306
path_hooks (*sys* モジュール), 2180
path_importer_cache (*sys* モジュール), 2180
path_mtime() (*importlib.abc.SourceLoader* のメソッド), 2300
path_return_ok() (*http.cookiejar.CookiePolicy* のメソッド), 1689
path_stats() (*importlib.abc.SourceLoader* のメソッド), 2300
path_stats() (*importlib.machinery.SourceFileLoader* のメソッド), 2306
pathconf() (*os* モジュール), 735
pathconf_names (*os* モジュール), 735
PathEntryFinder (*importlib.abc* のクラス), 2294
PathFinder (*importlib.machinery* のクラス), 2305
pathlib (モジュール), 471
PathLike (*os* のクラス), 708
pathname2url() (*urllib.request* モジュール), 1565
pathsep (*os* モジュール), 771
pattern (*re.error* の属性), 151
pattern (*re.Pattern* の属性), 153
Pattern (*typing* のクラス), 1908
--pattern pattern
    unittest-discover command line option, 1953
pause() (*signal* モジュール), 1314
pause_reading() (*asyncio.ReadTransport* のメソッド), 1177
pause_writing() (*asyncio.BaseProtocol* のメソッド), 1181
PAX_FORMAT (*tarfile* モジュール), 631
pax_headers (*tarfile.TarFile* の属性), 636
pax_headers (*tarfile.TarInfo* の属性), 638
pbkdf2_hmac() (*hashlib* モジュール), 690
pd() (*turtle* モジュール), 1789
Pdb (*class in pdb*), 2101
Pdb (*pdb* のクラス), 2103
pdb (モジュール), 2101
.pdbrc
    file, 2105
pdf() (*statistics.NormalDist* のメソッド), 430
peek() (*bz2.BZ2File* のメソッド), 605
peek() (*gzip.GzipFile* のメソッド), 601
peek() (*io.BufferedReader* のメソッド), 784
peek() (*lzma.LZMAFile* のメソッド), 610
peek() (*weakref.finalize* のメソッド), 316
peer (*smtpd.SMTPChannel* の属性), 1651
PEM_cert_to_DER_cert() (*ssl* モジュール), 1249
pen() (*turtle* モジュール), 1789
pencolor() (*turtle* モジュール), 1790
pending (*ssl.MemoryBIO* の属性), 1284
pending() (*ssl.SSLSocket* のメソッド), 1264
PendingDeprecationWarning, 122
pendown() (*turtle* モジュール), 1789
pensize() (*turtle* モジュール), 1789
penup() (*turtle* モジュール), 1789
PEP, **2489**
PERCENT (*token* モジュール), 2339
PERCENTEQUAL (*token* モジュール), 2340
perf_counter() (*time* モジュール), 794
perf_counter_ns() (*time* モジュール), 794
Performance, 2120
perm() (*math* モジュール), 367
PermissionError, 122
permutations() (*itertools* モジュール), 443
Persist() (*msilib.SummaryInformation* のメソッド), 2388
persistence, 535
persistent
    objects, 535
persistent_id (*pickle protocol*), 547
persistent_id() (*pickle.Pickler* のメソッド), 540
persistent_load (*pickle protocol*), 547
persistent_load() (*pickle.Unpickler* のメソッド), 542

PF_CAN (*socket* モジュール), 1216
PF_PACKET (*socket* モジュール), 1217
PF_RDS (*socket* モジュール), 1217
pformat() (*pprint* モジュール), 330
pformat() (*pprint.PrettyPrinter* のメソッド), 332
pgettext() (*gettext* モジュール), 1754
pgettext() (*gettext.GNUTranslations* のメソッド), 1760
pgettext() (*gettext.NullTranslations* のメソッド), 1757
PGO (*test.support* モジュール), 2069
phase() (*cmath* モジュール), 373
pi (*cmath* モジュール), 376
pi (*math* モジュール), 372
pi() (*xml.etree.ElementTree.TreeBuilder* のメソッド), 1482
pickle
    モジュール, 329, 556, 557, 561
pickle (モジュール), 535
pickle() (*copyreg* モジュール), 557
PickleBuffer (*pickle* のクラス), 542
PickleError, 539
Pickler (*pickle* のクラス), 539
pickletools (モジュール), 2375
pickletools command line option
    -a, 2376
    --annotate, 2376
    --indentlevel=<num>, 2376
    -l, 2376
    -m, 2376
    --memo, 2376
    -o, 2376
    --output=<file>, 2376
    -p, 2376
    --preamble=<preamble>, 2376
pickling
    objects, 535
PicklingError, 539
pid (*asyncio.asyncio.subprocess.Process* の属性), 1136
pid (*multiprocessing.Process* の属性), 1003
pid (*subprocess.Popen* の属性), 1074
PIPE (*subprocess* モジュール), 1064
Pipe() (*multiprocessing* モジュール), 1006
pipe() (*os* モジュール), 720
pipe2() (*os* モジュール), 720
PIPE_BUF (*select* モジュール), 1290
pipe_connection_lost() (*asyncio.SubprocessProtocol* のメソッド), 1184
pipe_data_received() (*asyncio.SubprocessProtocol* のメソッド), 1184
PIPE_MAX_SIZE (*test.support* モジュール), 2069
pipes (モジュール), 2423
PKG_DIRECTORY (*imp* モジュール), 2472
pkgutil (モジュール), 2282
placeholder (*textwrap.TextWrapper* の属性), 180
platform (*sys* モジュール), 2181
platform (モジュール), 923
platform() (*platform* モジュール), 924
PlaySound() (*winsound* モジュール), 2406
plist
    file, 683
plistlib (モジュール), 683
plock() (*os* モジュール), 759
PLUS (*token* モジュール), 2339
plus() (*decimal.Context* のメソッド), 397
PLUSEQUAL (*token* モジュール), 2340
pm() (*pdb* モジュール), 2103
POINTER() (*ctypes* モジュール), 967
pointer() (*ctypes* モジュール), 967
polar() (*cmath* モジュール), 374
Policy (*email.policy* のクラス), 1350
poll() (*multiprocessing.connection.Connection* のメソッド), 1012
poll() (*select* モジュール), 1289
poll() (*select.devpoll* のメソッド), 1291
poll() (*select.epoll* のメソッド), 1293
poll() (*select.poll* のメソッド), 1294
poll() (*subprocess.Popen* のメソッド), 1072
PollSelector (*selectors* のクラス), 1300
Pool (*multiprocessing.pool* のクラス), 1028
pop() (*array.array* のメソッド), 311
pop() (*collections.deque* のメソッド), 283
pop() (*dict* のメソッド), 100
pop() (*frozenset* のメソッド), 97
pop() (*mailbox.Mailbox* のメソッド), 1420
pop() (*sequence method*), 50
POP3
    protocol, 1619
POP3 (*poplib* のクラス), 1619
POP3_SSL (*poplib* のクラス), 1619
pop_alignment() (*formatter.formatter* のメソッド), 2381
pop_all() (*contextlib.ExitStack* のメソッド), 2220
POP_BLOCK (*opcode*), 2366
POP_EXCEPT (*opcode*), 2366
POP_FINALLY (*opcode*), 2367
pop_font() (*formatter.formatter* のメソッド), 2381
POP_JUMP_IF_FALSE (*opcode*), 2371
POP_JUMP_IF_TRUE (*opcode*), 2370
pop_margin() (*formatter.formatter* のメソッド), 2381
pop_source() (*shlex.shlex* のメソッド), 1825
pop_style() (*formatter.formatter* のメソッド), 2381
POP_TOP (*opcode*), 2362
Popen (*subprocess* のクラス), 1067
popen() (*in module os*), 1290
popen() (*os* モジュール), 759
popitem() (*collections.OrderedDict* のメソッド), 292
popitem() (*dict* のメソッド), 100
popitem() (*mailbox.Mailbox* のメソッド), 1420
popleft() (*collections.deque* のメソッド), 283
poplib (モジュール), 1619
PopupMenu (*tkinter.tix* のクラス), 1870
port (*http.cookiejar.Cookie* の属性), 1692
port_specified (*http.cookiejar.Cookie* の属性), 1693
portion, **2489**
pos (*json.JSONDecodeError* の属性), 1411
pos (*re.error* の属性), 152
pos (*re.Match* の属性), 156
pos() (*operator* モジュール), 464
pos() (*turtle* モジュール), 1787
position (*xml.etree.ElementTree.ParseError* の属性), 1486
position() (*turtle* モジュール), 1787
positional argument (位置引数), **2489**
POSIX
    I/O control, 2416
    threads, 1096
posix (モジュール), 2409
POSIX Shared Memory, 1048
POSIX_FADV_DONTNEED (*os* モジュール), 720
POSIX_FADV_NOREUSE (*os* モジュール), 720
POSIX_FADV_NORMAL (*os* モジュール), 720
POSIX_FADV_RANDOM (*os* モジュール), 720
POSIX_FADV_SEQUENTIAL (*os* モジュール), 720
POSIX_FADV_WILLNEED (*os* モジュール), 720
posix_fadvise() (*os* モジュール), 720
posix_fallocate() (*os* モジュール), 720
posix_spawn() (*os* モジュール), 759
POSIX_SPAWN_CLOSE (*os* モジュール), 760
POSIX_SPAWN_DUP2 (*os* モジュール), 760
POSIX_SPAWN_OPEN (*os* モジュール), 759
posix_spawnp() (*os* モジュール), 760
POSIXLY_CORRECT, 841
PosixPath (*pathlib* のクラス), 482
post() (*nntplib.NNTP* のメソッド), 1638
post() (*ossaudiodev.oss_audio_device* のメソッド), 1749
post_handshake_auth (*ssl.SSLContext* の属性), 1274
post_mortem() (*pdb* モジュール), 2103
post_setup() (*venv.EnvBuilder* のメソッド), 2151
postcmd() (*cmd.Cmd* のメソッド), 1818
postloop() (*cmd.Cmd* のメソッド), 1818

pow() (*math* モジュール), 369
pow() (*operator* モジュール), 464
pow() (組み込み関数), 24
power() (*decimal.Context* のメソッド), 397
pp (*pdb command*), 2108
pp() (*pprint* モジュール), 331
pprint (モジュール), 329
pprint() (*pprint* モジュール), 331
pprint() (*pprint.PrettyPrinter* のメソッド), 332
prcal() (*calendar* モジュール), 274
pread() (*os* モジュール), 721
preadv() (*os* モジュール), 721
preamble (*email.message.EmailMessage* の属性), 1338
preamble (*email.message.Message* の属性), 1386
--preamble=<preamble>
    pickletools command line option, 2376
precmd() (*cmd.Cmd* のメソッド), 1818
prefix (*sys* モジュール), 2181
prefix (*xml.dom.Attr* の属性), 1495
prefix (*xml.dom.Node* の属性), 1490
prefix (*zipimport.zipimporter* の属性), 2281
PREFIXES (*site* モジュール), 2271
prefixlen (*ipaddress.IPv4Network* の属性), 1719
prefixlen (*ipaddress.IPv6Network* の属性), 1722
preloop() (*cmd.Cmd* のメソッド), 1818
prepare() (*logging.handlers.QueueHandler* のメソッド), 891
prepare() (*logging.handlers.QueueListener* のメソッド), 892
prepare_class() (*types* モジュール), 322
prepare_input_source() (*xml.sax.saxutils* モジュール), 1518
prepend() (*pipes.Template* のメソッド), 2424
PrettyPrinter (*pprint* のクラス), 330
prev() (*tkinter.ttk.Treeview* のメソッド), 1863
previousSibling (*xml.dom.Node* の属性), 1490
print (*2to3 fixer*), 2064
print() (組み込み関数), 24
print_callees() (*pstats.Stats* のメソッド), 2117
print_callers() (*pstats.Stats* のメソッド), 2117
print_directory() (*cgi* モジュール), 1545
print_environ() (*cgi* モジュール), 1545
print_environ_usage() (*cgi* モジュール), 1545
print_exc() (*timeit.Timer* のメソッド), 2123
print_exc() (*traceback* モジュール), 2236
print_exception() (*traceback* モジュール), 2236
PRINT_EXPR (*opcode*), 2366
print_form() (*cgi* モジュール), 1545
print_help() (*argparse.ArgumentParser* のメソッド), 837
print_last() (*traceback* モジュール), 2236
print_stack() (*asyncio.Task* のメソッド), 1116
print_stack() (*traceback* モジュール), 2236
print_stats() (*profile.Profile* のメソッド), 2114
print_stats() (*pstats.Stats* のメソッド), 2116
print_tb() (*traceback* モジュール), 2235
print_usage() (*argparse.ArgumentParser* のメソッド), 837
print_usage() (*optparse.OptionParser* のメソッド), 2458
print_version() (*optparse.OptionParser* のメソッド), 2444
printable (*string* モジュール), 126
printdir() (*zipfile.ZipFile* のメソッド), 621
printf-style formatting, 65, 83
PRIO_PGRP (*os* モジュール), 710
PRIO_PROCESS (*os* モジュール), 710
PRIO_USER (*os* モジュール), 710
PriorityQueue (*asyncio* のクラス), 1139
PriorityQueue (*queue* のクラス), 1088
prlimit() (*resource* モジュール), 2425
prmonth() (*calendar* モジュール), 274
prmonth() (*calendar.TextCalendar* のメソッド), 271
ProactorEventLoop (*asyncio* のクラス), 1166
process
    group, 709, 710
    id, 710
    id of parent, 710
    killing, 758, 759
    scheduling priority, 710, 712
    signalling, 758, 759
--process
    timeit command line option, 2123
Process (*multiprocessing* のクラス), 1002
process() (*logging.LoggerAdapter* のメソッド), 856
process_exited() (*asyncio.SubprocessProtocol* のメソッド), 1184
process_message() (*smtpd.SMTPServer* のメソッド), 1648
process_request() (*socketserver.BaseServer* のメソッド), 1665
process_time() (*time* モジュール), 795
process_time_ns() (*time* モジュール), 795
process_tokens() (*tabnanny* モジュール), 2348
ProcessError, 1005
processes, light-weight, 1096
ProcessingInstruction() (*xml.etree.ElementTree* モジュール), 1473
processingInstruction() (*xml.sax.handler.ContentHandler* のメソッド), 1515
ProcessingInstructionHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1528
ProcessLookupError, 122
processor time, 795, 798
processor() (*platform* モジュール), 924
ProcessPoolExecutor (*concurrent.futures* のクラス), 1057
prod() (*math* モジュール), 368
product() (*itertools* モジュール), 444
Profile (*profile* のクラス), 2113
profile (モジュール), 2113
profile function, 979, 2175, 2182
profiler, 2175, 2182
profiling, deterministic, 2110
ProgrammingError, 586
Progressbar (*tkinter.ttk* のクラス), 1856
prompt (*cmd.Cmd* の属性), 1818
prompt_user_passwd() (*urllib.request.FancyURLopener* のメソッド), 1586
prompts, interpreter, 2182
propagate (*logging.Logger* の属性), 844
property (組み込みクラス), 25
property list, 683
property_declaration_handler (*xml.sax.handler* モジュール), 1512
property_dom_node (*xml.sax.handler* モジュール), 1512
property_lexical_handler (*xml.sax.handler* モジュール), 1512
property_xml_string (*xml.sax.handler* モジュール), 1513
PropertyMock (*unittest.mock* のクラス), 2000
prot_c() (*ftplib.FTP_TLS* のメソッド), 1619
prot_p() (*ftplib.FTP_TLS* のメソッド), 1619
proto (*socket.socket* の属性), 1237
protocol
    CGI, 1540
    context management, 103
    copy, 545
    FTP, 1587, 1612
    HTTP, 1540, 1587, 1601, 1604, 1671
    IMAP4, 1623
    IMAP4_SSL, 1623
    IMAP4_stream, 1623
    iterator, 46
    NNTP, 1631
    POP3, 1619
    SMTP, 1639
    Telnet, 1652
Protocol (*asyncio* のクラス), 1180
protocol (*ssl.SSLContext* の属性), 1274
Protocol (*typing* のクラス), 1902

PROTOCOL_SSLv2 (*ssl* モジュール), 1253
PROTOCOL_SSLv3 (*ssl* モジュール), 1253
PROTOCOL_SSLv23 (*ssl* モジュール), 1253
PROTOCOL_TLS (*ssl* モジュール), 1252
PROTOCOL_TLS_CLIENT (*ssl* モジュール), 1252
PROTOCOL_TLS_SERVER (*ssl* モジュール), 1252
PROTOCOL_TLSv1 (*ssl* モジュール), 1253
PROTOCOL_TLSv1_1 (*ssl* モジュール), 1253
PROTOCOL_TLSv1_2 (*ssl* モジュール), 1253
protocol_version (*http.server.BaseHTTPRequestHandler* の属性), 1673
PROTOCOL_VERSION (*imaplib.IMAP4* の属性), 1630
ProtocolError (*xmlrpc.client* のクラス), 1701
provisional API, **2489**
provisional package, **2490**
proxy() (*weakref* モジュール), 314
proxyauth() (*imaplib.IMAP4* のメソッド), 1627
ProxyBasicAuthHandler (*urllib.request* のクラス), 1568
ProxyDigestAuthHandler (*urllib.request* のクラス), 1569
ProxyHandler (*urllib.request* のクラス), 1567
ProxyType (*weakref* モジュール), 317
ProxyTypes (*weakref* モジュール), 317
pryear() (*calendar.TextCalendar* のメソッド), 271
ps1 (*sys* モジュール), 2182
ps2 (*sys* モジュール), 2182
pstats (モジュール), 2115
pstdev() (*statistics* モジュール), 426
pthread_getcpuclockid() (*time* モジュール), 792
pthread_kill() (*signal* モジュール), 1315
pthread_sigmask() (*signal* モジュール), 1315
pthreads, 1096
pty
    モジュール, 720
pty (モジュール), 2418
pu() (*turtle* モジュール), 1789
publicId (*xml.dom.DocumentType* の属性), 1492
PullDom (*xml.dom.pulldom* のクラス), 1507
punctuation (*string* モジュール), 126
punctuation_chars (*shlex.shlex* の属性), 1827
PurePath (*pathlib* のクラス), 473
PurePath.anchor (*pathlib* モジュール), 477
PurePath.drive (*pathlib* モジュール), 476
PurePath.name (*pathlib* モジュール), 478
PurePath.parent (*pathlib* モジュール), 477
PurePath.parents (*pathlib* モジュール), 477
PurePath.parts (*pathlib* モジュール), 476
PurePath.root (*pathlib* モジュール), 476
PurePath.stem (*pathlib* モジュール), 478
PurePath.suffix (*pathlib* モジュール), 478
PurePath.suffixes (*pathlib* モジュール), 478
PurePosixPath (*pathlib* のクラス), 474
PureProxy (*smtpd* のクラス), 1650
PureWindowsPath (*pathlib* のクラス), 474
purge() (*re* モジュール), 151
Purpose.CLIENT_AUTH (*ssl* モジュール), 1258
Purpose.SERVER_AUTH (*ssl* モジュール), 1258
push() (*asynchat.async_chat* のメソッド), 1308
push() (*code.InteractiveConsole* のメソッド), 2275
push() (*contextlib.ExitStack* のメソッド), 2220
push_alignment() (*formatter.formatter* のメソッド), 2381
push_async_callback() (*contextlib.AsyncExitStack* のメソッド), 2221
push_async_exit() (*contextlib.AsyncExitStack* のメソッド), 2221
push_font() (*formatter.formatter* のメソッド), 2381
push_margin() (*formatter.formatter* のメソッド), 2381
push_source() (*shlex.shlex* のメソッド), 1825
push_style() (*formatter.formatter* のメソッド), 2381
push_token() (*shlex.shlex* のメソッド), 1824
push_with_producer() (*asynchat.async_chat* のメソッド), 1308
pushbutton() (*msilib.Dialog* のメソッド), 2392
put() (*asyncio.Queue* のメソッド), 1139
put() (*multiprocessing.Queue* のメソッド), 1007
put() (*multiprocessing.SimpleQueue* のメソッド), 1009
put() (*queue.Queue* のメソッド), 1089
put() (*queue.SimpleQueue* のメソッド), 1091
put_nowait() (*asyncio.Queue* のメソッド), 1139
put_nowait() (*multiprocessing.Queue* のメソッド), 1007
put_nowait() (*queue.Queue* のメソッド), 1089
put_nowait() (*queue.SimpleQueue* のメソッド), 1091
putch() (*msvcrt* モジュール), 2394
putenv() (*os* モジュール), 711
putheader() (*http.client.HTTPConnection* のメソッド), 1609
putp() (*curses* モジュール), 900
putrequest() (*http.client.HTTPConnection* のメソッド), 1609
putwch() (*msvcrt* モジュール), 2394
putwin() (*curses.window* のメソッド), 909
pvariance() (*statistics* モジュール), 426
pwd
    モジュール, 493
pwd (モジュール), 2410
pwd() (*ftplib.FTP* のメソッド), 1618
pwrite() (*os* モジュール), 722
pwritev() (*os* モジュール), 722
py_compile (モジュール), 2350
PY_COMPILED (*imp* モジュール), 2471
PY_FROZEN (*imp* モジュール), 2472
py_object (*ctypes* のクラス), 972
PY_SOURCE (*imp* モジュール), 2471
pycache_prefix (*sys* モジュール), 2169
PycInvalidationMode (*py_compile* のクラス), 2352
pyclbr (モジュール), 2348
PyCompileError, 2350
PyDLL (*ctypes* のクラス), 958
pydoc (モジュール), 1917
pyexpat
    モジュール, 1523
PYFUNCTYPE() (*ctypes* モジュール), 962
Python 3000, **2490**
Python Editor, 1875
Python Enhancement Proposals
    PEP 1, 2489
    PEP 205, 317
    PEP 227, 2244
    PEP 235, 2290
    PEP 237, 67, 86
    PEP 238, 2244, 2482
    PEP 249, 568, 570
    PEP 255, 2244
    PEP 263, 2290, 2343, 2344
    PEP 273, 2279
    PEP 278, 2492
    PEP 282, 529, 863
    PEP 292, 136
    PEP 302, 32, 519, 2180, 2181, 2280, 22822285, 2287, 2290, 2293, 2295, 2298, 2299, 2472, 2482, 2486
    PEP 305, 649
    PEP 307, 537
    PEP 324, 1062
    PEP 328, 32, 2244, 2290
    PEP 338, 2289
    PEP 342, 298
    PEP 343, 2225, 2244, 2480
    PEP 362, 2260, 2478, 2489
    PEP 366, 2289, 2290
    PEP 370, 2272
    PEP 378, 131
    PEP 383, 205, 1211
    PEP 393, 213, 2179
    PEP 397, 2149
    PEP 405, 2146
    PEP 411, 2176, 2177, 2185, 2186, 2490
    PEP 420, 2290, 2482, 2488, 2489

PEP 421, 2178
PEP 428, 472
PEP 442, 2247
PEP 443, 2483
PEP 451, 2180, 2283, 22882290, 2482
PEP 453, 2144
PEP 461, 86
PEP 468, 293
PEP 475, 24, 121, 718, 723, 725, 766, 795, 1228, 1229, 1231, 12331235, 1290, 12921295, 1300, 1318, 1319
PEP 479, 118, 2244
PEP 483, 1893
PEP 484, 1893, 1895, 19001903, 1912, 2331, 2477, 2482, 2492, 2493
PEP 485, 367, 376
PEP 488, 2071, 2290, 2310, 2351
PEP 489, 2290, 2304, 2308
PEP 492, 300, 2267, 2268, 24782480
PEP 498, 2482
PEP 506, 702
PEP 515, 131
PEP 519, 2489
PEP 524, 772
PEP 525, 300, 2176, 2185, 2268, 2478
PEP 526, 1893, 1909, 1910, 1915, 2203, 2210, 2331, 2477, 2493
PEP 529, 731, 2174, 2186
PEP 544, 1893, 1900
PEP 552, 2290, 2351
PEP 557, 2203
PEP 560, 322
PEP 563, 2244
PEP 566, 2316, 2319
PEP 567, 1092, 1146, 1147, 1172
PEP 574, 537, 554
PEP 578, 2087, 2165
PEP 586, 1893, 1915
PEP 589, 1893, 1910
PEP 591, 1893, 1912, 1916
PEP 706, 640
PEP 3101, 126, 127
PEP 3105, 2244
PEP 3112, 2244
PEP 3115, 322
PEP 3116, 2492
PEP 3118, 88
PEP 3119, 301, 2228
PEP 3120, 2290
PEP 3141, 361, 2228
PEP 3147, 2071, 2288, 2290, 2310, 2351, 23542356, 2469, 2470
PEP 3148, 1061
PEP 3149, 2165
PEP 3151, 122, 1214, 1288, 2424
PEP 3154, 537
PEP 3155, 2490
PEP 3333, 1550, 1551, 15531556, 1560, 1561
--python=<interpreter>
zipapp command line option, 2157
python_branch() (*platform* モジュール), 924
python_build() (*platform* モジュール), 924
python_compiler() (*platform* モジュール), 924
PYTHON_DOM, 1488
python_implementation() (*platform* モジュール), 924
python_is_optimized() (*test.support* モジュール), 2071
python_revision() (*platform* モジュール), 925
python_version() (*platform* モジュール), 925
python_version_tuple() (*platform* モジュール), 925
PYTHONASYNCIODEBUG, 1161, 1207
PYTHONBREAKPOINT, 2168
PYTHONDEVMODE, 2086
PYTHONDOCS, 1918
PYTHONDONTWRITEBYTECODE, 2169
PYTHONFAULTHANDLER, 2098
PYTHONHOME, 2085
Pythonic, **2490**
PYTHONINTMAXSTRDIGITS, 110, 2178
PYTHONIOENCODING, 2186
PYTHONLEGACYWINDOWSFSENCODING, 2186
PYTHONLEGACYWINDOWSSTDIO, 2186
PYTHONNOUSERSITE, 2271
PYTHONPATH, 1546, 2085, 2180
PYTHONPYCACHEPREFIX, 2169
PYTHONSTARTUP, 189, 1884, 2179, 2270
PYTHONTRACEMALLOC, 2129, 2130, 2135
PYTHONUSERBASE, 2271, 2272
PYTHONUSERSITE, 2085
PYTHONUTF8, 2186
PYTHONWARNINGS, 2197, 2198
PyZipFile (*zipfile* のクラス), 623

Q

-q
compileall command line option, 2353
qiflush() (*curses* モジュール), 900
QName (*xml.etree.ElementTree* のクラス), 1482
qsize() (*asyncio.Queue* のメソッド), 1139
qsize() (*multiprocessing.Queue* のメソッド), 1007
qsize() (*queue.Queue* のメソッド), 1089
qsize() (*queue.SimpleQueue* のメソッド), 1091
qualified name, **2490**
quantiles() (*statistics* モジュール), 428
quantiles() (*statistics.NormalDist* のメソッド), 431
quantize() (*decimal.Context* のメソッド), 397
quantize() (*decimal.Decimal* のメソッド), 389
QueryInfoKey() (*winreg* モジュール), 2399
QueryReflectionKey() (*winreg* モジュール), 2402
QueryValue() (*winreg* モジュール), 2400
QueryValueEx() (*winreg* モジュール), 2400
Queue (*asyncio* のクラス), 1138
Queue (*multiprocessing* のクラス), 1007
Queue (*queue* のクラス), 1088
queue (*sched.scheduler* の属性), 1087
queue (モジュール), 1087
Queue() (*multiprocessing.managers.SyncManager* のメソッド), 1023
QueueEmpty, 1139
QueueFull, 1139
QueueHandler (*logging.handlers* のクラス), 891
QueueListener (*logging.handlers* のクラス), 892
quick_ratio() (*difflib.SequenceMatcher* のメソッド), 171
quit (*pdb command*), 2109
quit (組み込み変数), 36
quit() (*ftplib.FTP* のメソッド), 1618
quit() (*nntplib.NNTP* のメソッド), 1634
quit() (*poplib.POP3* のメソッド), 1621
quit() (*smtplib.SMTP* のメソッド), 1647
quopri (モジュール), 1450
quote() (*email.utils* モジュール), 1398
quote() (*shlex* モジュール), 1823
quote() (*urllib.parse* モジュール), 1596
QUOTE_ALL (*csv* モジュール), 653
quote_from_bytes() (*urllib.parse* モジュール), 1596
QUOTE_MINIMAL (*csv* モジュール), 653
QUOTE_NONE (*csv* モジュール), 653
QUOTE_NONNUMERIC (*csv* モジュール), 653
quote_plus() (*urllib.parse* モジュール), 1596
quoteattr() (*xml.sax.saxutils* モジュール), 1517
quotechar (*csv.Dialect* の属性), 654
quoted-printable
encoding, 1450
quotes (*shlex.shlex* の属性), 1826
quoting (*csv.Dialect* の属性), 654

R

-R
    trace command line option, 2127
-r
    compileall command line option, 2354
    trace command line option, 2127
-r N
    timeit command line option, 2123
R_OK (*os* モジュール), 728
radians() (*math* モジュール), 371
radians() (*turtle* モジュール), 1788
RadioButtonGroup (*msilib* のクラス), 2391
radiogroup() (*msilib.Dialog* のメソッド), 2392
radix() (*decimal.Context* のメソッド), 398
radix() (*decimal.Decimal* のメソッド), 389
RADIXCHAR (*locale* モジュール), 1768
raise
    文, 113
raise (*2to3 fixer*), 2064
raise_on_defect (*email.policy.Policy* の属性), 1350
raise_signal() (*signal* モジュール), 1315
RAISE_VARARGS (*opcode*), 2372
RAND_add() (*ssl* モジュール), 1247
RAND_bytes() (*ssl* モジュール), 1246
RAND_egd() (*ssl* モジュール), 1247
RAND_pseudo_bytes() (*ssl* モジュール), 1246
RAND_status() (*ssl* モジュール), 1247
randbelow() (*secrets* モジュール), 702
randbits() (*secrets* モジュール), 702
randint() (*random* モジュール), 415
Random (*random* のクラス), 418
random (モジュール), 413
random() (*random* モジュール), 416
randrange() (*random* モジュール), 415
range
    オブジェクト, 53
range (組み込みクラス), 53
RARROW (*token* モジュール), 2341
ratecv() (*audioop* モジュール), 1729
ratio() (*difflib.SequenceMatcher* のメソッド), 171
Rational (*numbers* のクラス), 362
raw (*io.BufferedIOBase* の属性), 780
raw() (*curses* モジュール), 900
raw() (*pickle.PickleBuffer* のメソッド), 542
raw_data_manager (*email.contentmanager* モジュール), 1367
raw_decode() (*json.JSONDecoder* のメソッド), 1409
raw_input (*2to3 fixer*), 2064
raw_input() (*code.InteractiveConsole* のメソッド), 2275
RawArray() (*multiprocessing.sharedctypes* モジュール), 1018
RawConfigParser (*configparser* のクラス), 676
RawDescriptionHelpFormatter (*argparse* のクラス), 809
RawIOBase (*io* のクラス), 779
RawPen (*turtle* のクラス), 1809
RawTextHelpFormatter (*argparse* のクラス), 809
RawTurtle (*turtle* のクラス), 1809
RawValue() (*multiprocessing.sharedctypes* モジュール), 1018
RBRACE (*token* モジュール), 2339
rcpttos (*smtpd.SMTPChannel* の属性), 1651
re
    モジュール, 56, 517
re (*re.Match* の属性), 156
re (モジュール), 139
read() (*asyncio.StreamReader* のメソッド), 1120
read() (*chunk.Chunk* のメソッド), 1742
read() (*codecs.StreamReader* のメソッド), 211
read() (*configparser.ConfigParser* のメソッド), 673
read() (*http.client.HTTPResponse* のメソッド), 1610
read() (*imaplib.IMAP4* のメソッド), 1627
read() (*io.BufferedIOBase* のメソッド), 781
read() (*io.BufferedReader* のメソッド), 784
read() (*io.RawIOBase* のメソッド), 780
read() (*io.TextIOBase* のメソッド), 786
read() (*mimetypes.MimeTypes* のメソッド), 1441
read() (*mmap.mmap* のメソッド), 1323
read() (*os* モジュール), 723
read() (*ossaudiodev.oss_audio_device* のメソッド), 1747
read() (*ssl.MemoryBIO* のメソッド), 1284
read() (*ssl.SSLSocket* のメソッド), 1260
read() (*urllib.robotparser.RobotFileParser* のメソッド), 1600
read() (*zipfile.ZipFile* のメソッド), 621
read1() (*io.BufferedIOBase* のメソッド), 781
read1() (*io.BufferedReader* のメソッド), 784
read1() (*io.BytesIO* のメソッド), 783
read_all() (*telnetlib.Telnet* のメソッド), 1653
read_binary() (*importlib.resources* モジュール), 2302
read_byte() (*mmap.mmap* のメソッド), 1323
read_bytes() (*pathlib.Path* のメソッド), 487
read_bytes() (*zipfile.Path* のメソッド), 623
read_dict() (*configparser.ConfigParser* のメソッド), 674
read_eager() (*telnetlib.Telnet* のメソッド), 1654
read_environ() (*wsgiref.handlers* モジュール), 1561
read_events() (*xml.etree.ElementTree.XMLPullParser* のメソッド), 1485
read_file() (*configparser.ConfigParser* のメソッド), 674
read_history_file() (*readline* モジュール), 186
read_init_file() (*readline* モジュール), 186
read_lazy() (*telnetlib.Telnet* のメソッド), 1654
read_mime_types() (*mimetypes* モジュール), 1440
read_sb_data() (*telnetlib.Telnet* のメソッド), 1654
read_some() (*telnetlib.Telnet* のメソッド), 1653
read_string() (*configparser.ConfigParser* のメソッド), 674
read_text() (*importlib.resources* モジュール), 2303
read_text() (*pathlib.Path* のメソッド), 487
read_text() (*zipfile.Path* のメソッド), 623
read_token() (*shlex.shlex* のメソッド), 1824
read_until() (*telnetlib.Telnet* のメソッド), 1653
read_very_eager() (*telnetlib.Telnet* のメソッド), 1653
read_very_lazy() (*telnetlib.Telnet* のメソッド), 1654
read_windows_registry() (*mimetypes.MimeTypes* のメソッド), 1442
READABLE (*tkinter* モジュール), 1845
readable() (*asyncore.dispatcher* のメソッド), 1304
readable() (*io.IOBase* のメソッド), 778
readall() (*io.RawIOBase* のメソッド), 780
reader() (*csv* モジュール), 650
ReadError, 630
readexactly() (*asyncio.StreamReader* のメソッド), 1121
readfp() (*configparser.ConfigParser* のメソッド), 676
readfp() (*mimetypes.MimeTypes* のメソッド), 1442
readframes() (*aifc.aifc* のメソッド), 1732
readframes() (*sunau.AU_read* のメソッド), 1736
readframes() (*wave.Wave_read* のメソッド), 1739
readinto() (*http.client.HTTPResponse* のメソッド), 1610
readinto() (*io.BufferedIOBase* のメソッド), 781
readinto() (*io.RawIOBase* のメソッド), 780
readinto1() (*io.BufferedIOBase* のメソッド), 781
readinto1() (*io.BytesIO* のメソッド), 783
readline (モジュール), 185
readline() (*asyncio.StreamReader* のメソッド), 1121
readline() (*codecs.StreamReader* のメソッド), 211
readline() (*imaplib.IMAP4* のメソッド), 1627
readline() (*io.IOBase* のメソッド), 778
readline() (*io.TextIOBase* のメソッド), 786
readline() (*mmap.mmap* のメソッド), 1324
readlines() (*codecs.StreamReader* のメソッド), 211
readlines() (*io.IOBase* のメソッド), 778
readlink() (*os* モジュール), 736
readmodule() (*pyclbr* モジュール), 2348
readmodule_ex() (*pyclbr* モジュール), 2348
readonly (*memoryview* の属性), 93
readPlist() (*plistlib* モジュール), 685
readPlistFromBytes() (*plistlib* モジュール), 685

ReadTransport (*asyncio* のクラス), 1175
readuntil() (*asyncio.StreamReader* のメソッド), 1121
readv() (*os* モジュール), 724
ready() (*multiprocessing.pool.AsyncResult* のメソッド), 1031
Real (*numbers* のクラス), 361
real (*numbers.Complex* の属性), 361
Real Media File Format, 1741
real_max_memuse (*test.support* モジュール), 2070
real_quick_ratio() (*difflib.SequenceMatcher* のメソッド), 172
realpath() (*os.path* モジュール), 496
REALTIME_PRIORITY_CLASS (*subprocess* モジュール), 1077
reap_children() (*test.support* モジュール), 2079
reap_threads() (*test.support* モジュール), 2077
reason (*http.client.HTTPResponse* の属性), 1610
reason (*ssl.SSLError* の属性), 1245
reason (*UnicodeError* の属性), 119
reason (*urllib.error.HTTPError* の属性), 1599
reason (*urllib.error.URLError* の属性), 1599
reattach() (*tkinter.ttk.Treeview* のメソッド), 1863
reccontrols() (*ossaudiodev.oss_mixer_device* のメソッド), 1751
received_data (*smtpd.SMTPChannel* の属性), 1651
received_lines (*smtpd.SMTPChannel* の属性), 1651
recent() (*imaplib.IMAP4* のメソッド), 1628
reconfigure() (*io.TextIOWrapper* のメソッド), 788
record_original_stdout() (*test.support* モジュール), 2073
records (*unittest.TestCase* の属性), 1966
rect() (*cmath* モジュール), 374
rectangle() (*curses.textpad* モジュール), 917
RecursionError, 117
recursive_repr() (*reprlib* モジュール), 336
recv() (*asyncore.dispatcher* のメソッド), 1304
recv() (*multiprocessing.connection.Connection* のメソッド), 1011
recv() (*socket.socket* のメソッド), 1231
recv_bytes() (*multiprocessing.connection.Connection* のメソッド), 1012
recv_bytes_into() (*multiprocessing.connection.Connection* のメソッド), 1012
recv_into() (*socket.socket* のメソッド), 1233
recvfrom() (*socket.socket* のメソッド), 1231
recvfrom_into() (*socket.socket* のメソッド), 1233
recvmsg() (*socket.socket* のメソッド), 1231
recvmsg_into() (*socket.socket* のメソッド), 1233
redirect_request() (*urllib.request.HTTPRedirectHandler* のメソッド), 1575
redirect_stderr() (*contextlib* モジュール), 2217
redirect_stdout() (*contextlib* モジュール), 2217
redisplay() (*readline* モジュール), 186
redrawln() (*curses.window* のメソッド), 909
redrawwin() (*curses.window* のメソッド), 909
reduce (*2to3 fixer*), 2064
reduce() (*functools* モジュール), 456
reducer_override() (*pickle.Pickler* のメソッド), 541
ref (*weakref* のクラス), 313
refcount_test() (*test.support* モジュール), 2077
reference count, **2491**
ReferenceError, 117
ReferenceType (*weakref* モジュール), 316
refold_source (*email.policy.EmailPolicy* の属性), 1353
refresh() (*curses.window* のメソッド), 909
REG_BINARY (*winreg* モジュール), 2404
REG_DWORD (*winreg* モジュール), 2404
REG_DWORD_BIG_ENDIAN (*winreg* モジュール), 2404
REG_DWORD_LITTLE_ENDIAN (*winreg* モジュール), 2404
REG_EXPAND_SZ (*winreg* モジュール), 2404
REG_FULL_RESOURCE_DESCRIPTOR (*winreg* モジュール), 2405
REG_LINK (*winreg* モジュール), 2404
REG_MULTI_SZ (*winreg* モジュール), 2404
REG_NONE (*winreg* モジュール), 2405
REG_QWORD (*winreg* モジュール), 2405
REG_QWORD_LITTLE_ENDIAN (*winreg* モジュール), 2405
REG_RESOURCE_LIST (*winreg* モジュール), 2405
REG_RESOURCE_REQUIREMENTS_LIST (*winreg* モジュール), 2405
REG_SZ (*winreg* モジュール), 2405
register() (*abc.ABCMeta* のメソッド), 2229
register() (*atexit* モジュール), 2234
register() (*codecs* モジュール), 202
register() (*faulthandler* モジュール), 2100
register() (*multiprocessing.managers.BaseManager* のメソッド), 1021
register() (*select.devpoll* のメソッド), 1291
register() (*select.epoll* のメソッド), 1293
register() (*selectors.BaseSelector* のメソッド), 1299
register() (*select.poll* のメソッド), 1293
register() (*webbrowser* モジュール), 1538
register_adapter() (*sqlite3* モジュール), 572
register_archive_format() (*shutil* モジュール), 529
register_at_fork() (*os* モジュール), 761
register_converter() (*sqlite3* モジュール), 571
register_defect() (*email.policy.Policy* のメソッド), 1351
register_dialect() (*csv* モジュール), 651
register_error() (*codecs* モジュール), 205
register_function() (*xmlrpc.server.CGIXMLRPCRequestHandler* のメソッド), 1709
register_function() (*xmlrpc.server.SimpleXMLRPCServer* のメソッド), 1705
register_instance() (*xmlrpc.server.CGIXMLRPCRequestHandler* のメソッド), 1709
register_instance() (*xmlrpc.server.SimpleXMLRPCServer* のメソッド), 1705
register_introspection_functions() (*xmlrpc.server.CGIXMLRPCRequestHandler* のメソッド), 1709
register_introspection_functions() (*xmlrpc.server.SimpleXMLRPCServer* のメソッド), 1705
register_multicall_functions() (*xmlrpc.server.CGIXMLRPCRequestHandler* のメソッド), 1709
register_multicall_functions() (*xmlrpc.server.SimpleXMLRPCServer* のメソッド), 1706
register_namespace() (*xml.etree.ElementTree* モジュール), 1473
register_optionflag() (*doctest* モジュール), 1930
register_shape() (*turtle* モジュール), 1807
register_unpack_format() (*shutil* モジュール), 530
registerDOMImplementation() (*xml.dom* モジュール), 1488
registerResult() (*unittest* モジュール), 1986
regular package, **2491**
relative
    URL, 1588
relative_to() (*pathlib.PurePath* のメソッド), 480
release() (*_thread.lock* のメソッド), 1098
release() (*asyncio.Condition* のメソッド), 1130
release() (*asyncio.Lock* のメソッド), 1127
release() (*asyncio.Semaphore* のメソッド), 1131
release() (*logging.Handler* のメソッド), 849
release() (*memoryview* のメソッド), 90
release() (*multiprocessing.Lock* のメソッド), 1015
release() (*multiprocessing.RLock* のメソッド), 1015
release() (*pickle.PickleBuffer* のメソッド), 542
release() (*platform* モジュール), 925
release() (*threading.Condition* のメソッド), 987
release() (*threading.Lock* のメソッド), 984
release() (*threading.RLock* のメソッド), 985

release() (*threading.Semaphore* のメソッド), 989
release_lock() (*imp* モジュール), 2471
reload (*2to3 fixer*), 2064
reload() (*imp* モジュール), 2468
reload() (*importlib* モジュール), 2292
relpath() (*os.path* モジュール), 496
remainder() (*decimal.Context* のメソッド), 398
remainder() (*math* モジュール), 368
remainder_near() (*decimal.Context* のメソッド), 398
remainder_near() (*decimal.Decimal* のメソッド), 389
RemoteDisconnected, 1606
remove() (*array.array* のメソッド), 311
remove() (*collections.deque* のメソッド), 283
remove() (*frozenset* のメソッド), 97
remove() (*mailbox.Mailbox* のメソッド), 1418
remove() (*mailbox.MH* のメソッド), 1424
remove() (*os* モジュール), 736
remove() (*sequence method*), 50
remove() (*xml.etree.ElementTree.Element* のメソッド), 1478
remove_child_handler() (*asyncio.AbstractChildWatcher* のメソッド), 1193
remove_done_callback() (*asyncio.Future* のメソッド), 1172
remove_done_callback() (*asyncio.Task* のメソッド), 1116
remove_flag() (*mailbox.MaildirMessage* のメソッド), 1428
remove_flag() (*mailbox.mboxMessage* のメソッド), 1431
remove_flag() (*mailbox.MMDFMessage* のメソッド), 1435
remove_folder() (*mailbox.Maildir* のメソッド), 1422
remove_folder() (*mailbox.MH* のメソッド), 1424
remove_header() (*urllib.request.Request* のメソッド), 1571
remove_history_item() (*readline* モジュール), 187
remove_label() (*mailbox.BabylMessage* のメソッド), 1433
remove_option() (*configparser.ConfigParser* のメソッド), 675
remove_option() (*optparse.OptionParser* のメソッド), 2456
remove_pyc() (*msilib.Directory* のメソッド), 2390
remove_reader() (*asyncio.loop* のメソッド), 1155
remove_section() (*configparser.ConfigParser* のメソッド), 675
remove_sequence() (*mailbox.MHMessage* のメソッド), 1432
remove_signal_handler() (*asyncio.loop* のメソッド), 1159
remove_writer() (*asyncio.loop* のメソッド), 1155
removeAttribute() (*xml.dom.Element* のメソッド), 1495
removeAttributeNode() (*xml.dom.Element* のメソッド), 1495
removeAttributeNS() (*xml.dom.Element* のメソッド), 1495
removeChild() (*xml.dom.Node* のメソッド), 1491
removedirs() (*os* モジュール), 736
removeFilter() (*logging.Handler* のメソッド), 849
removeFilter() (*logging.Logger* のメソッド), 847
removeHandler() (*logging.Logger* のメソッド), 847
removeHandler() (*unittest* モジュール), 1986
removeResult() (*unittest* モジュール), 1986
removexattr() (*os* モジュール), 753
rename() (*ftplib.FTP* のメソッド), 1617
rename() (*imaplib.IMAP4* のメソッド), 1628
rename() (*os* モジュール), 737
rename() (*pathlib.Path* のメソッド), 487
renames (*2to3 fixer*), 2064
renames() (*os* モジュール), 737
reopenIfNeeded() (*logging.handlers.WatchedFileHandler* のメソッド), 879
reorganize() (*dbm.gnu.gdbm* のメソッド), 565
repeat() (*itertools* モジュール), 445
repeat() (*timeit* モジュール), 2121
repeat() (*timeit.Timer* のメソッド), 2122
--repeat=N
    timeit command line option, 2123
repetition
    operation, 47
replace() (*bytearray* のメソッド), 73
replace() (*bytes* のメソッド), 73
replace() (*curses.panel.Panel* のメソッド), 923
replace() (*dataclasses* モジュール), 2209
replace() (*datetime.date* のメソッド), 234
replace() (*datetime.datetime* のメソッド), 244
replace() (*datetime.time* のメソッド), 253
replace() (*inspect.Parameter* のメソッド), 2258
replace() (*inspect.Signature* のメソッド), 2256
replace() (*os* モジュール), 737
replace() (*pathlib.Path* のメソッド), 488
replace() (*str* のメソッド), 61
replace() (*types.CodeType* のメソッド), 323
replace_errors() (*codecs* モジュール), 206
replace_header() (*email.message.EmailMessage* のメソッド), 1333
replace_header() (*email.message.Message* のメソッド), 1382
replace_history_item() (*readline* モジュール), 187
replace_whitespace (*textwrap.TextWrapper* の属性), 179
replaceChild() (*xml.dom.Node* のメソッド), 1491
ReplacePackage() (*modulefinder* モジュール), 2285
--report
    trace command line option, 2127
report() (*filecmp.dircmp* のメソッド), 508
report() (*modulefinder.ModuleFinder* のメソッド), 2285
REPORT_CDIFF (*doctest* モジュール), 1929
report_failure() (*doctest.DocTestRunner* のメソッド), 1942
report_full_closure() (*filecmp.dircmp* のメソッド), 508
REPORT_NDIFF (*doctest* モジュール), 1929
REPORT_ONLY_FIRST_FAILURE (*doctest* モジュール), 1929
report_partial_closure() (*filecmp.dircmp* のメソッド), 508
report_start() (*doctest.DocTestRunner* のメソッド), 1942
report_success() (*doctest.DocTestRunner* のメソッド), 1942
REPORT_UDIFF (*doctest* モジュール), 1929
report_unexpected_exception() (*doctest.DocTestRunner* のメソッド), 1942
REPORTING_FLAGS (*doctest* モジュール), 1930
repr (*2to3 fixer*), 2064
Repr (*reprlib* のクラス), 336
repr() (*reprlib* モジュール), 336
repr() (*reprlib.Repr* のメソッド), 337
repr() (組み込み関数), 26
repr1() (*reprlib.Repr* のメソッド), 337
reprlib (モジュール), 336
Request (*urllib.request* のクラス), 1566
request() (*http.client.HTTPConnection* のメソッド), 1607
request_queue_size (*socketserver.BaseServer* の属性), 1664
request_rate() (*urllib.robotparser.RobotFileParser* のメソッド), 1600
request_uri() (*wsgiref.util* モジュール), 1550
request_version (*http.server.BaseHTTPRequestHandler* の属性), 1672
RequestHandlerClass (*socketserver.BaseServer* の属性), 1664
requestline (*http.server.BaseHTTPRequestHandler* の属性), 1672
requires() (*test.support* モジュール), 2071
requires_bz2() (*test.support* モジュール), 2077
requires_docstrings() (*test.support* モジュール), 2077
requires_freebsd_version() (*test.support* モジュール), 2077
requires_gzip() (*test.support* モジュール), 2077
requires_IEEE_754() (*test.support* モジュール), 2077
requires_linux_version() (*test.support* モジュール), 2077
requires_lzma() (*test.support* モジュール), 2077
requires_mac_version() (*test.support* モジュール), 2077
requires_resource() (*test.support* モジュール), 2077
requires_zlib() (*test.support* モジュール), 2077

reserved (*zipfile.ZipInfo* の属性), 626
RESERVED_FUTURE (*uuid* モジュール), 1659
RESERVED_MICROSOFT (*uuid* モジュール), 1659
RESERVED_NCS (*uuid* モジュール), 1659
reset() (*bdb.Bdb* のメソッド), 2093
reset() (*codecs.IncrementalDecoder* のメソッド), 209
reset() (*codecs.IncrementalEncoder* のメソッド), 208
reset() (*codecs.StreamReader* のメソッド), 211
reset() (*codecs.StreamWriter* のメソッド), 210
reset() (*contextvars.ContextVar* のメソッド), 1093
reset() (*html.parser.HTMLParser* のメソッド), 1455
reset() (*ossaudiodev.oss_audio_device* のメソッド), 1749
reset() (*pipes.Template* のメソッド), 2423
reset() (*threading.Barrier* のメソッド), 993
reset() (*turtle* モジュール), 1802
reset() (*xdrlib.Packer* のメソッド), 680
reset() (*xdrlib.Unpacker* のメソッド), 681
reset() (*xml.dom.pulldom.DOMEventStream* のメソッド), 1508
reset() (*xml.sax.xmlreader.IncrementalParser* のメソッド), 1521
reset_mock() (*unittest.mock.AsyncMock* のメソッド), 2004
reset_mock() (*unittest.mock.Mock* のメソッド), 1993
reset_prog_mode() (*curses* モジュール), 900
reset_shell_mode() (*curses* モジュール), 900
resetbuffer() (*code.InteractiveConsole* のメソッド), 2275
resetlocale() (*locale* モジュール), 1770
resetscreen() (*turtle* モジュール), 1802
resetty() (*curses* モジュール), 900
resetwarnings() (*warnings* モジュール), 2202
resize() (*ctypes* モジュール), 967
resize() (*curses.window* のメソッド), 910
resize() (*mmap.mmap* のメソッド), 1324
resize_term() (*curses* モジュール), 900
resizemode() (*turtle* モジュール), 1794
resizeterm() (*curses* モジュール), 901
resolution (*datetime.date* の属性), 233
resolution (*datetime.datetime* の属性), 242
resolution (*datetime.time* の属性), 252
resolution (*datetime.timedelta* の属性), 229
resolve() (*pathlib.Path* のメソッド), 488
resolve_bases() (*types* モジュール), 322
resolve_name() (*importlib.util* モジュール), 2310
resolveEntity() (*xml.sax.handler.EntityResolver* のメソッド), 1516
Resource (*importlib.resources* モジュール), 2302
resource (モジュール), 2424
resource_path() (*importlib.abc.ResourceReader* のメソッド), 2297
ResourceDenied, 2068
ResourceLoader (*importlib.abc* のクラス), 2298
ResourceReader (*importlib.abc* のクラス), 2297
ResourceWarning, 123
response (*nntplib.NNTPError* の属性), 1633
response() (*imaplib.IMAP4* のメソッド), 1628
ResponseNotReady, 1606
responses (*http.client* モジュール), 1607
responses (*http.server.BaseHTTPRequestHandler* の属性), 1673
restart (*pdb command*), 2109
restore() (*difflib* モジュール), 167
restype (*ctypes.__FuncPtr* の属性), 961
result() (*asyncio.Future* のメソッド), 1171
result() (*asyncio.Task* のメソッド), 1115
result() (*concurrent.futures.Future* のメソッド), 1059
results() (*trace.Trace* のメソッド), 2128
resume_reading() (*asyncio.ReadTransport* のメソッド), 1177
resume_writing() (*asyncio.BaseProtocol* のメソッド), 1181
retr() (*poplib.POP3* のメソッド), 1621
retrbinary() (*ftplib.FTP* のメソッド), 1616
retrieve() (*urllib.request.URLopener* のメソッド), 1585
retrlines() (*ftplib.FTP* のメソッド), 1616
return (*pdb command*), 2107
return_annotation (*inspect.Signature* の属性), 2256
return_ok() (*http.cookiejar.CookiePolicy* のメソッド), 1688
RETURN_VALUE (*opcode*), 2366
return_value (*unittest.mock.Mock* の属性), 1995
returncode (*asyncio.asyncio.subprocess.Process* の属性), 1136
returncode (*subprocess.CalledProcessError* の属性), 1065
returncode (*subprocess.CompletedProcess* の属性), 1063
returncode (*subprocess.Popen* の属性), 1074
retval (*pdb command*), 2109
reverse() (*array.array* のメソッド), 311
reverse() (*audioop* モジュール), 1730
reverse() (*collections.deque* のメソッド), 283
reverse() (*sequence method*), 50
reverse_order() (*pstats.Stats* のメソッド), 2116
reverse_pointer (*ipaddress.IPv4Address* の属性), 1713
reverse_pointer (*ipaddress.IPv6Address* の属性), 1715
reversed() (組み込み関数), 27
Reversible (*collections.abc* のクラス), 298
Reversible (*typing* のクラス), 1903
revert() (*http.cookiejar.FileCookieJar* のメソッド), 1687
rewind() (*aifc.aifc* のメソッド), 1732
rewind() (*sunau.AU_read* のメソッド), 1736
rewind() (*wave.Wave_read* のメソッド), 1739
RFC
    RFC 821, 1639, 1642
    RFC 822, 797, 1369, 1391, 1609, 1643, 1645, 1647, 1759
    RFC 854, 1652, 1653
    RFC 959, 1612, 1617
    RFC 977, 1631
    RFC 1014, 679, 680
    RFC 1123, 797
    RFC 1321, 687
    RFC 1422, 1276
    RFC 1521, 1446, 1450
    RFC 1522, 1448, 1450
    RFC 1524, 1415
    RFC 1730, 1623
    RFC 1738, 1598
    RFC 1750, 1247
    RFC 1766, 1769
    RFC 1808, 1589, 1598
    RFC 1832, 680
    RFC 1869, 1639, 1642
    RFC 1870, 1648, 1652
    RFC 1939, 1619
    RFC 2045, 1327, 1333, 1362, 1363, 1382, 1384, 1391, 1442, 1445, 1446
    RFC 2045#section-6.8, 1699
    RFC 2046, 1327, 1368, 1391
    RFC 2047, 1327, 1353, 1359, 1360, 1391, 1392, 1399
    RFC 2060, 1623, 1629
    RFC 2068, 1678
    RFC 2087, 1626, 1628
    RFC 2104, 700
    RFC 2109, 16781681, 16831685, 1690, 1692, 1693
    RFC 2183, 1327, 1335, 1385
    RFC 2231, 1327, 1333, 1334, 1381, 1383, 1384, 1391, 1400, 1401
    RFC 2295, 1603
    RFC 2342, 1627
    RFC 2368, 1598
    RFC 2373, 1714
    RFC 2396, 1592, 1596, 1598
    RFC 2397, 1579
    RFC 2449, 1621
    RFC 2518, 1602
    RFC 2595, 1619, 1622
    RFC 2616, 1552, 1556, 1575, 1586, 1599
    RFC 2732, 1598

RFC 2774, 1603
RFC 2818, 1247
RFC 2822, 796, 797, 1380, 1391, 1392, 13981400, 1427, 1605, 1672
RFC 2964, 1685
RFC 2965, 1566, 1570, 16831685, 1687, 1689, 1691, 1692, 1694
RFC 2980, 1631, 1638
RFC 3056, 1715
RFC 3171, 1714
RFC 3229, 1602
RFC 3280, 1261
RFC 3330, 1714
RFC 3454, 183
RFC 3490, 220, 222, 223
RFC 3490#section-3.1, 222
RFC 3492, 220, 222
RFC 3493, 1242
RFC 3501, 1629
RFC 3542, 1226
RFC 3548, 1442, 1444, 1448
RFC 3659, 1617
RFC 3879, 1715
RFC 3927, 1714
RFC 3977, 1631, 1633, 1634, 1636, 1638
RFC 3986, 1590, 1593, 1596, 1598, 1672
RFC 4086, 1288
RFC 4122, 16561659
RFC 4180, 649
RFC 4193, 1715
RFC 4217, 1613
RFC 4291, 1714
RFC 4380, 1715
RFC 4627, 1402, 1413
RFC 4642, 1632
RFC 4918, 1602, 1603
RFC 4954, 1644
RFC 5161, 1626
RFC 5246, 1258, 1288
RFC 5280, 1247, 1248, 1288
RFC 5321, 1365, 1648, 1649
RFC 5322, 1329, 1341, 1345, 1346, 1350, 13521354, 1357, 1358, 1360, 1361, 1364, 1365, 1376, 1646
RFC 5424, 885
RFC 5735, 1714
RFC 5842, 1602, 1603
RFC 5891, 222
RFC 5895, 222
RFC 5929, 1263
RFC 6066, 1256, 1269, 1288
RFC 6125, 1247, 1248
RFC 6152, 1648
RFC 6531, 1330, 1353, 1640, 1648, 1650
RFC 6532, 1329, 1341, 1353
RFC 6585, 1603
RFC 6855, 1626
RFC 6856, 1622
RFC 7159, 1402, 1411, 1413
RFC 7230, 1566, 1609
RFC 7231, 1602, 1603
RFC 7232, 1602, 1603
RFC 7233, 1602, 1603
RFC 7235, 1602, 1603
RFC 7238, 1602
RFC 7301, 1256, 1269
RFC 7525, 1288
RFC 7540, 1603
RFC 7693, 691
RFC 7725, 1603
RFC 7914, 691
RFC 8305, 1149
rfc2109 (*http.cookiejar.Cookie* の属性), 1693
rfc2109_as_netscape (*http.cookiejar.DefaultCookiePolicy* の属性), 1690
rfc2965 (*http.cookiejar.CookiePolicy* の属性), 1689
RFC_4122 (*uuid* モジュール), 1659
rfile (*http.server.BaseHTTPRequestHandler* の属性), 1672
rfind() (*bytearray* のメソッド), 74
rfind() (*bytes* のメソッド), 74
rfind() (*mmap.mmap* のメソッド), 1324
rfind() (*str* のメソッド), 61
rgb_to_hls() (*colorsys* モジュール), 1743
rgb_to_hsv() (*colorsys* モジュール), 1743
rgb_to_yiq() (*colorsys* モジュール), 1743
rglob() (*pathlib.Path* のメソッド), 488
right (*filecmp.dircmp* の属性), 508
right() (*turtle* モジュール), 1781
right_list (*filecmp.dircmp* の属性), 509
right_only (*filecmp.dircmp* の属性), 509
RIGHTSHIFT (*token* モジュール), 2340
RIGHTSHIFTEQUAL (*token* モジュール), 2340
rindex() (*bytearray* のメソッド), 74
rindex() (*bytes* のメソッド), 74
rindex() (*str* のメソッド), 61
rjust() (*bytearray* のメソッド), 76
rjust() (*bytes* のメソッド), 76
rjust() (*str* のメソッド), 61
rlcompleter (モジュール), 190
rlecode_hqx() (*binascii* モジュール), 1448
rledecode_hqx() (*binascii* モジュール), 1448
RLIM_INFINITY (*resource* モジュール), 2425
RLIMIT_AS (*resource* モジュール), 2427
RLIMIT_CORE (*resource* モジュール), 2426
RLIMIT_CPU (*resource* モジュール), 2426
RLIMIT_DATA (*resource* モジュール), 2426
RLIMIT_FSIZE (*resource* モジュール), 2426
RLIMIT_MEMLOCK (*resource* モジュール), 2427
RLIMIT_MSGQUEUE (*resource* モジュール), 2427
RLIMIT_NICE (*resource* モジュール), 2427
RLIMIT_NOFILE (*resource* モジュール), 2426
RLIMIT_NPROC (*resource* モジュール), 2426
RLIMIT_NPTS (*resource* モジュール), 2428
RLIMIT_OFILE (*resource* モジュール), 2426
RLIMIT_RSS (*resource* モジュール), 2426
RLIMIT_RTPRIO (*resource* モジュール), 2427
RLIMIT_RTTIME (*resource* モジュール), 2427
RLIMIT_SBSIZE (*resource* モジュール), 2427
RLIMIT_SIGPENDING (*resource* モジュール), 2427
RLIMIT_STACK (*resource* モジュール), 2426
RLIMIT_SWAP (*resource* モジュール), 2428
RLIMIT_VMEM (*resource* モジュール), 2427
RLock (*multiprocessing* のクラス), 1015
RLock (*threading* のクラス), 985
RLock() (*multiprocessing.managers.SyncManager* のメソッド), 1023
rmd() (*ftplib.FTP* のメソッド), 1618
rmdir() (*os* モジュール), 738
rmdir() (*pathlib.Path* のメソッド), 488
rmdir() (*test.support* モジュール), 2070
RMFF, 1741
rms() (*audioop* モジュール), 1730
rmtree() (*shutil* モジュール), 524
rmtree() (*test.support* モジュール), 2071
RobotFileParser (*urllib.robotparser* のクラス), 1600
robots.txt, 1600
rollback() (*sqlite3.Connection* のメソッド), 573
ROT_FOUR (*opcode*), 2362
ROT_THREE (*opcode*), 2362
ROT_TWO (*opcode*), 2362
rotate() (*collections.deque* のメソッド), 283
rotate() (*decimal.Context* のメソッド), 398
rotate() (*decimal.Decimal* のメソッド), 390
rotate() (*logging.handlers.BaseRotatingHandler* のメソッド), 880

RotatingFileHandler (*logging.handlers* のクラス), 881
rotation_filename() (*logging.handlers.BaseRotatingHandler* のメソッド), 880
rotator (*logging.handlers.BaseRotatingHandler* の属性), 879
round() (組み込み関数), 27
ROUND_05UP (*decimal* モジュール), 400
ROUND_CEILING (*decimal* モジュール), 399
ROUND_DOWN (*decimal* モジュール), 399
ROUND_FLOOR (*decimal* モジュール), 399
ROUND_HALF_DOWN (*decimal* モジュール), 399
ROUND_HALF_EVEN (*decimal* モジュール), 399
ROUND_HALF_UP (*decimal* モジュール), 399
ROUND_UP (*decimal* モジュール), 399
Rounded (*decimal* のクラス), 401
Row (*sqlite3* のクラス), 584
row_factory (*sqlite3.Connection* の属性), 577
rowcount (*sqlite3.Cursor* の属性), 583
RPAR (*token* モジュール), 2338
rpartition() (*bytearray* のメソッド), 74
rpartition() (*bytes* のメソッド), 74
rpartition() (*str* のメソッド), 61
rpc_paths (*xmlrpc.server.SimpleXMLRPCRequestHandler* の属性), 1706
rpop() (*poplib.POP3* のメソッド), 1621
rset() (*poplib.POP3* のメソッド), 1621
rshift() (*operator* モジュール), 464
rsplit() (*bytearray* のメソッド), 76
rsplit() (*bytes* のメソッド), 76
rsplit() (*str* のメソッド), 61
RSQB (*token* モジュール), 2338
rstrip() (*bytearray* のメソッド), 76
rstrip() (*bytes* のメソッド), 76
rstrip() (*str* のメソッド), 61
rt() (*turtle* モジュール), 1781
RTLD_DEEPBIND (*os* モジュール), 771
RTLD_GLOBAL (*os* モジュール), 771
RTLD_LAZY (*os* モジュール), 771
RTLD_LOCAL (*os* モジュール), 771
RTLD_NODELETE (*os* モジュール), 771
RTLD_NOLOAD (*os* モジュール), 771
RTLD_NOW (*os* モジュール), 771
ruler (*cmd.Cmd* の属性), 1819
run (*pdb command*), 2109
Run script, 1878
run() (*asyncio* モジュール), 1106
run() (*bdb.Bdb* のメソッド), 2097
run() (*contextvars.Context* のメソッド), 1094
run() (*doctest.DocTestRunner* のメソッド), 1942
run() (*multiprocessing.Process* のメソッド), 1002
run() (*pdb* モジュール), 2103
run() (*pdb.Pdb* のメソッド), 2104
run() (*profile* モジュール), 2113
run() (*profile.Profile* のメソッド), 2114
run() (*sched.scheduler* のメソッド), 1087
run() (*subprocess* モジュール), 1062
run() (*test.support.BasicTestRunner* のメソッド), 2084
run() (*threading.Thread* のメソッド), 982
run() (*trace.Trace* のメソッド), 2128
run() (*unittest.IsolatedAsyncioTestCase* のメソッド), 1971
run() (*unittest.TestCase* のメソッド), 1961
run() (*unittest.TestSuite* のメソッド), 1974
run() (*unittest.TextTestRunner* のメソッド), 1981
run() (*wsgiref.handlers.BaseHandler* のメソッド), 1558
run_coroutine_threadsafe() (*asyncio* モジュール), 1112
run_docstring_examples() (*doctest* モジュール), 1934
run_doctest() (*test.support* モジュール), 2072
run_forever() (*asyncio.loop* のメソッド), 1144
run_in_executor() (*asyncio.loop* のメソッド), 1159
run_in_subinterp() (*test.support* モジュール), 2082
run_module() (*runpy* モジュール), 2287
run_path() (*runpy* モジュール), 2288
run_python_until_end() (*test.support.script_helper* モジュール), 2085
run_script() (*modulefinder.ModuleFinder* のメソッド), 2286
run_unittest() (*test.support* モジュール), 2072
run_until_complete() (*asyncio.loop* のメソッド), 1144
run_with_locale() (*test.support* モジュール), 2076
run_with_tz() (*test.support* モジュール), 2076
runcall() (*bdb.Bdb* のメソッド), 2098
runcall() (*pdb* モジュール), 2103
runcall() (*pdb.Pdb* のメソッド), 2104
runcall() (*profile.Profile* のメソッド), 2114
runcode() (*code.InteractiveInterpreter* のメソッド), 2274
runctx() (*bdb.Bdb* のメソッド), 2097
runctx() (*profile* モジュール), 2113
runctx() (*profile.Profile* のメソッド), 2114
runctx() (*trace.Trace* のメソッド), 2128
runeval() (*bdb.Bdb* のメソッド), 2097
runeval() (*pdb* モジュール), 2103
runeval() (*pdb.Pdb* のメソッド), 2104
runfunc() (*trace.Trace* のメソッド), 2128
running() (*concurrent.futures.Future* のメソッド), 1059
runpy (モジュール), 2287
runsource() (*code.InteractiveInterpreter* のメソッド), 2274
runtime_checkable() (*typing* モジュール), 1913
RuntimeError, 117
RuntimeWarning, 122
RUSAGE_BOTH (*resource* モジュール), 2429
RUSAGE_CHILDREN (*resource* モジュール), 2429
RUSAGE_SELF (*resource* モジュール), 2429
RUSAGE_THREAD (*resource* モジュール), 2429
RWF_DSYNC (*os* モジュール), 722
RWF_HIPRI (*os* モジュール), 722
RWF_NOWAIT (*os* モジュール), 721
RWF_SYNC (*os* モジュール), 723

S

-s
    trace command line option, 2127
    unittest-discover command line option, 1953
S (*re* モジュール), 148
-s S
    timeit command line option, 2123
S_ENFMT (*stat* モジュール), 505
S_IEXEC (*stat* モジュール), 505
S_IFBLK (*stat* モジュール), 504
S_IFCHR (*stat* モジュール), 504
S_IFDIR (*stat* モジュール), 504
S_IFDOOR (*stat* モジュール), 504
S_IFIFO (*stat* モジュール), 504
S_IFLNK (*stat* モジュール), 503
S_IFMT() (*stat* モジュール), 502
S_IFPORT (*stat* モジュール), 504
S_IFREG (*stat* モジュール), 503
S_IFSOCK (*stat* モジュール), 503
S_IFWHT (*stat* モジュール), 504
S_IMODE() (*stat* モジュール), 501
S_IREAD (*stat* モジュール), 505
S_IRGRP (*stat* モジュール), 505
S_IROTH (*stat* モジュール), 505
S_IRUSR (*stat* モジュール), 505
S_IRWXG (*stat* モジュール), 505
S_IRWXO (*stat* モジュール), 505
S_IRWXU (*stat* モジュール), 505
S_ISBLK() (*stat* モジュール), 501
S_ISCHR() (*stat* モジュール), 501
S_ISDIR() (*stat* モジュール), 501
S_ISDOOR() (*stat* モジュール), 501
S_ISFIFO() (*stat* モジュール), 501
S_ISGID (*stat* モジュール), 504
S_ISLNK() (*stat* モジュール), 501

S_ISPORT() (stat モジュール), 501
S_ISREG() (stat モジュール), 501
S_ISSOCK() (stat モジュール), 501
S_ISUID (stat モジュール), 504
S_ISVTX (stat モジュール), 504
S_ISWHT() (stat モジュール), 501
S_IWGRP (stat モジュール), 505
S_IWOTH (stat モジュール), 505
S_IWRITE (stat モジュール), 505
S_IWUSR (stat モジュール), 505
S_IXGRP (stat モジュール), 505
S_IXOTH (stat モジュール), 505
S_IXUSR (stat モジュール), 505
safe (uuid.SafeUUID の属性), 1656
safe_substitute() (string.Template のメソッド), 137
SafeChildWatcher (asyncio のクラス), 1194
saferepr() (pprint モジュール), 331
SafeUUID (uuid のクラス), 1656
same_files (filecmp.dircmp の属性), 509
same_quantum() (decimal.Context のメソッド), 398
same_quantum() (decimal.Decimal のメソッド), 390
samefile() (os.path モジュール), 496
samefile() (pathlib.Path のメソッド), 488
SameFileError, 520
sameopenfile() (os.path モジュール), 496
samestat() (os.path モジュール), 497
sample() (random モジュール), 416
samples() (statistics.NormalDist のメソッド), 430
save() (http.cookiejar.FileCookieJar のメソッド), 1686
SAVEDCWD (test.support モジュール), 2069
SaveKey() (winreg モジュール), 2400
SaveSignals (test.support のクラス), 2084
savetty() (curses モジュール), 901
SAX2DOM (xml.dom.pulldom のクラス), 1507
SAXException, 1509
SAXNotRecognizedException, 1510
SAXNotSupportedException, 1510
SAXParseException, 1509
scaleb() (decimal.Context のメソッド), 398
scaleb() (decimal.Decimal のメソッド), 390
scandir() (os モジュール), 738
scanf(), 158
sched (モジュール), 1085
SCHED_BATCH (os モジュール), 768
SCHED_FIFO (os モジュール), 768
sched_get_priority_max() (os モジュール), 769
sched_get_priority_min() (os モジュール), 768
sched_getaffinity() (os モジュール), 769
sched_getparam() (os モジュール), 769
sched_getscheduler() (os モジュール), 769
SCHED_IDLE (os モジュール), 768
SCHED_OTHER (os モジュール), 768
sched_param (os のクラス), 768
sched_priority (os.sched_param の属性), 768
SCHED_RESET_ON_FORK (os モジュール), 768
SCHED_RR (os モジュール), 768
sched_rr_get_interval() (os モジュール), 769
sched_setaffinity() (os モジュール), 769
sched_setparam() (os モジュール), 769
sched_setscheduler() (os モジュール), 769
SCHED_SPORADIC (os モジュール), 768
sched_yield() (os モジュール), 769
scheduler (sched のクラス), 1085
schema (msilib モジュール), 2392
Screen (turtle のクラス), 1810
screensize() (turtle モジュール), 1802
script_from_examples() (doctest モジュール), 1944
scroll() (curses.window のメソッド), 910
ScrolledCanvas (turtle のクラス), 1810
scrollok() (curses.window のメソッド), 910
scrypt() (hashlib モジュール), 691
seal() (unittest.mock モジュール), 2034
search
    path, module, 519, 2180, 2268
search() (imaplib.IMAP4 のメソッド), 1628
search() (re モジュール), 148
search() (re.Pattern のメソッド), 152
second (datetime.datetime の属性), 242
second (datetime.time の属性), 252
seconds since the epoch, 790
secrets (モジュール), 702
SECTCRE (configparser.ConfigParser の属性), 669
sections() (configparser.ConfigParser のメソッド), 673
secure (http.cookiejar.Cookie の属性), 1692
secure hash algorithm, SHA1, SHA224, SHA256, SHA384, SHA512, 687
Secure Sockets Layer, 1242
security
    CGI, 1546
    http.server, 1678
see() (tkinter.ttk.Treeview のメソッド), 1863
seed() (random モジュール), 414
seek() (chunk.Chunk のメソッド), 1742
seek() (io.IOBase のメソッド), 778
seek() (io.TextIOBase のメソッド), 786
seek() (mmap.mmap のメソッド), 1324
SEEK_CUR (os モジュール), 718
SEEK_END (os モジュール), 718
SEEK_SET (os モジュール), 718
seekable() (io.IOBase のメソッド), 779
seen_greeting (smtpd.SMTPChannel の属性), 1651
Select (tkinter.tix のクラス), 1870
select (モジュール), 1288
select() (imaplib.IMAP4 のメソッド), 1628
select() (select モジュール), 1290
select() (selectors.BaseSelector のメソッド), 1299
select() (tkinter.ttk.Notebook のメソッド), 1855
selected_alpn_protocol() (ssl.SSLSocket のメソッド), 1263
selected_npn_protocol() (ssl.SSLSocket のメソッド), 1263
selection() (tkinter.ttk.Treeview のメソッド), 1863
selection_add() (tkinter.ttk.Treeview のメソッド), 1863
selection_remove() (tkinter.ttk.Treeview のメソッド), 1863
selection_set() (tkinter.ttk.Treeview のメソッド), 1863
selection_toggle() (tkinter.ttk.Treeview のメソッド), 1863
selector (urllib.request.Request の属性), 1570
SelectorEventLoop (asyncio のクラス), 1166
SelectorKey (selectors のクラス), 1298
selectors (モジュール), 1297
SelectSelector (selectors のクラス), 1300
Semaphore (asyncio のクラス), 1130
Semaphore (multiprocessing のクラス), 1016
Semaphore (threading のクラス), 989
Semaphore() (multiprocessing.managers.SyncManager のメソッド), 1023
semaphores, binary, 1096
SEMI (token モジュール), 2339
send() (asyncore.dispatcher のメソッド), 1304
send() (http.client.HTTPConnection のメソッド), 1609
send() (imaplib.IMAP4 のメソッド), 1628
send() (logging.handlers.DatagramHandler のメソッド), 885
send() (logging.handlers.SocketHandler のメソッド), 884
send() (multiprocessing.connection.Connection のメソッド), 1011
send() (socket.socket のメソッド), 1233
send_bytes() (multiprocessing.connection.Connection のメソッド), 1012
send_error() (http.server.BaseHTTPRequestHandler のメソッド), 1673
send_flowing_data() (formatter.writer のメソッド), 2383
send_header() (http.server.BaseHTTPRequestHandler のメソッド), 1674

send_hor_rule() (*formatter.writer* のメソッド), 2383
send_label_data() (*formatter.writer* のメソッド), 2383
send_line_break() (*formatter.writer* のメソッド), 2383
send_literal_data() (*formatter.writer* のメソッド), 2383
send_message() (*smtplib.SMTP* のメソッド), 1646
send_paragraph() (*formatter.writer* のメソッド), 2383
send_response() (*http.server.BaseHTTPRequestHandler* のメソッド), 1674
send_response_only() (*http.server.BaseHTTPRequestHandler* のメソッド), 1674
send_signal() (*asyncio.asyncio.subprocess.Process* のメソッド), 1135
send_signal() (*asyncio.SubprocessTransport* のメソッド), 1180
send_signal() (*subprocess.Popen* のメソッド), 1073
sendall() (*socket.socket* のメソッド), 1234
sendcmd() (*ftplib.FTP* のメソッド), 1615
sendfile() (*asyncio.loop* のメソッド), 1154
sendfile() (*os* モジュール), 723
sendfile() (*socket.socket* のメソッド), 1235
sendfile() (*wsgiref.handlers.BaseHandler* のメソッド), 1561
SendfileNotAvailableError, 1141
sendmail() (*smtplib.SMTP* のメソッド), 1645
sendmsg() (*socket.socket* のメソッド), 1234
sendmsg_afalg() (*socket.socket* のメソッド), 1235
sendto() (*asyncio.DatagramTransport* のメソッド), 1179
sendto() (*socket.socket* のメソッド), 1234
sentinel (*multiprocessing.Process* の属性), 1004
sentinel (*unittest.mock* モジュール), 2024
sep (*os* モジュール), 771
sequence, **2491**
  iteration, 46
  types, immutable, 50
  types, mutable, 50
  types, operations on, 47, 50
  オブジェクト, 47
Sequence (*collections.abc* のクラス), 298
sequence (*msilib* モジュール), 2392
Sequence (*typing* のクラス), 1904
sequence2st() (*parser* モジュール), 2323
SequenceMatcher (*difflib* のクラス), 169
serializing
  objects, 535
serve_forever() (*asyncio.Server* のメソッド), 1165
serve_forever() (*socketserver.BaseServer* のメソッド), 1663
server
  WWW, 1540, 1671
Server (*asyncio* のクラス), 1164
server (*http.server.BaseHTTPRequestHandler* の属性), 1672
server_activate() (*socketserver.BaseServer* のメソッド), 1665
server_address (*socketserver.BaseServer* の属性), 1664
server_bind() (*socketserver.BaseServer* のメソッド), 1665
server_close() (*socketserver.BaseServer* のメソッド), 1664
server_hostname (*ssl.SSLSocket* の属性), 1264
server_side (*ssl.SSLSocket* の属性), 1264
server_software (*wsgiref.handlers.BaseHandler* の属性), 1559
server_version (*http.server.BaseHTTPRequestHandler* の属性), 1672
server_version (*http.server.SimpleHTTPRequestHandler* の属性), 1675
ServerProxy (*xmlrpc.client* のクラス), 1695
service_actions() (*socketserver.BaseServer* のメソッド), 1664
session (*ssl.SSLSocket* の属性), 1264
session_reused (*ssl.SSLSocket* の属性), 1264
session_stats() (*ssl.SSLContext* のメソッド), 1272
set
  オブジェクト, 94
Set (*collections.abc* のクラス), 299
Set (*typing* のクラス), 1905
set (組み込みクラス), 95
Set Breakpoint, 1880
set comprehension, **2491**
set() (*asyncio.Event* のメソッド), 1128
set() (*configparser.ConfigParser* のメソッド), 675
set() (*configparser.RawConfigParser* のメソッド), 677
set() (*contextvars.ContextVar* のメソッド), 1092
set() (*http.cookies.Morsel* のメソッド), 1681
set() (*ossaudiodev.oss_mixer_device* のメソッド), 1751
set() (*test.support.EnvironmentVarGuard* のメソッド), 2083
set() (*threading.Event* のメソッド), 991
set() (*tkinter.ttk.Combobox* のメソッド), 1851
set() (*tkinter.ttk.Spinbox* のメソッド), 1852
set() (*tkinter.ttk.Treeview* のメソッド), 1864
set() (*xml.etree.ElementTree.Element* のメソッド), 1477
SET_ADD (*opcode*), 2366
set_allowed_domains() (*http.cookiejar.DefaultCookiePolicy* のメソッド), 1690
set_alpn_protocols() (*ssl.SSLContext* のメソッド), 1268
set_app() (*wsgiref.simple_server.WSGIServer* のメソッド), 1555
set_asyncgen_hooks() (*sys* モジュール), 2185
set_authorizer() (*sqlite3.Connection* のメソッド), 576
set_auto_history() (*readline* モジュール), 187
set_blocked_domains() (*http.cookiejar.DefaultCookiePolicy* のメソッド), 1690
set_blocking() (*os* モジュール), 724
set_boundary() (*email.message.EmailMessage* のメソッド), 1335
set_boundary() (*email.message.Message* のメソッド), 1384
set_break() (*bdb.Bdb* のメソッド), 2096
set_charset() (*email.message.Message* のメソッド), 1379
set_child_watcher() (*asyncio* モジュール), 1192
set_child_watcher() (*asyncio.AbstractEventLoopPolicy* のメソッド), 1191
set_children() (*tkinter.ttk.Treeview* のメソッド), 1860
set_ciphers() (*ssl.SSLContext* のメソッド), 1268
set_completer() (*readline* モジュール), 188
set_completer_delims() (*readline* モジュール), 188
set_completion_display_matches_hook() (*readline* モジュール), 189
set_content() (*email.contentmanager* モジュール), 1367
set_content() (*email.contentmanager.ContentManager* のメソッド), 1366
set_content() (*email.message.EmailMessage* のメソッド), 1337
set_continue() (*bdb.Bdb* のメソッド), 2096
set_cookie() (*http.cookiejar.CookieJar* のメソッド), 1686
set_cookie_if_ok() (*http.cookiejar.CookieJar* のメソッド), 1686
set_coroutine_origin_tracking_depth() (*sys* モジュール), 2185
set_current() (*msilib.Feature* のメソッド), 2390
set_data() (*importlib.abc.SourceLoader* のメソッド), 2301
set_data() (*importlib.machinery.SourceFileLoader* のメソッド), 2307
set_date() (*mailbox.MaildirMessage* のメソッド), 1429
set_debug() (*asyncio.loop* のメソッド), 1161
set_debug() (*gc* モジュール), 2245
set_debuglevel() (*ftplib.FTP* のメソッド), 1615
set_debuglevel() (*http.client.HTTPConnection* のメソッド), 1608
set_debuglevel() (*nntplib.NNTP* のメソッド), 1638
set_debuglevel() (*poplib.POP3* のメソッド), 1620
set_debuglevel() (*smtplib.SMTP* のメソッド), 1642
set_debuglevel() (*telnetlib.Telnet* のメソッド), 1654

set_default_executor() (asyncio.loop のメソッド), 1160
set_default_type() (email.message.EmailMessage のメソッド), 1334
set_default_type() (email.message.Message のメソッド), 1382
set_default_verify_paths() (ssl.SSLContext のメソッド), 1268
set_defaults() (argparse.ArgumentParser のメソッド), 836
set_defaults() (optparse.OptionParser のメソッド), 2458
set_ecdh_curve() (ssl.SSLContext のメソッド), 1270
set_errno() (ctypes モジュール), 967
set_event_loop() (asyncio モジュール), 1142
set_event_loop() (asyncio.AbstractEventLoopPolicy のメソッド), 1191
set_event_loop_policy() (asyncio モジュール), 1191
set_exception() (asyncio.Future のメソッド), 1171
set_exception() (concurrent.futures.Future のメソッド), 1060
set_exception_handler() (asyncio.loop のメソッド), 1160
set_executable() (multiprocessing モジュール), 1011
set_flags() (mailbox.MaildirMessage のメソッド), 1428
set_flags() (mailbox.mboxMessage のメソッド), 1430
set_flags() (mailbox.MMDFMessage のメソッド), 1435
set_from() (mailbox.mboxMessage のメソッド), 1430
set_from() (mailbox.MMDFMessage のメソッド), 1435
set_handle_inheritable() (os モジュール), 726
set_history_length() (readline モジュール), 186
set_info() (mailbox.MaildirMessage のメソッド), 1429
set_inheritable() (os モジュール), 726
set_inheritable() (socket.socket のメソッド), 1235
set_int_max_str_digits() (sys モジュール), 2182
set_labels() (mailbox.BabylMessage のメソッド), 1433
set_last_error() (ctypes モジュール), 967
set_literal (2to3 fixer), 2064
set_loader() (importlib.util モジュール), 2312
set_match_tests() (test.support モジュール), 2072
set_memlimit() (test.support モジュール), 2073
set_name() (asyncio.Task のメソッド), 1117
set_next() (bdb.Bdb のメソッド), 2096
set_nonstandard_attr() (http.cookiejar.Cookie のメソッド), 1693
set_npn_protocols() (ssl.SSLContext のメソッド), 1269
set_ok() (http.cookiejar.CookiePolicy のメソッド), 1688
set_option_negotiation_callback() (telnetlib.Telnet のメソッド), 1655
set_output_charset() (gettext.NullTranslations のメソッド), 1758
set_package() (importlib.util モジュール), 2312
set_param() (email.message.EmailMessage のメソッド), 1334
set_param() (email.message.Message のメソッド), 1383
set_pasv() (ftplib.FTP のメソッド), 1616
set_payload() (email.message.Message のメソッド), 1379
set_policy() (http.cookiejar.CookieJar のメソッド), 1686
set_position() (xdrlib.Unpacker のメソッド), 681
set_pre_input_hook() (readline モジュール), 188
set_progress_handler() (sqlite3.Connection のメソッド), 576
set_protocol() (asyncio.BaseTransport のメソッド), 1177
set_proxy() (urllib.request.Request のメソッド), 1571
set_quit() (bdb.Bdb のメソッド), 2096
set_recsrc() (ossaudiodev.oss_mixer_device のメソッド), 1751
set_result() (asyncio.Future のメソッド), 1171
set_result() (concurrent.futures.Future のメソッド), 1060
set_return() (bdb.Bdb のメソッド), 2096
set_running_or_notify_cancel() (concurrent.futures.Future のメソッド), 1060
set_seq1() (difflib.SequenceMatcher のメソッド), 169
set_seq2() (difflib.SequenceMatcher のメソッド), 169
set_seqs() (difflib.SequenceMatcher のメソッド), 169
set_sequences() (mailbox.MH のメソッド), 1424
set_sequences() (mailbox.MHMessage のメソッド), 1432
set_server_documentation() (xmlrpc.server.DocCGIXMLRPCRequestHandler のメソッド), 1711
set_server_documentation() (xmlrpc.server.DocXMLRPCServer のメソッド), 1710
set_server_name() (xmlrpc.server.DocCGIXMLRPCRequestHandler のメソッド), 1711
set_server_name() (xmlrpc.server.DocXMLRPCServer のメソッド), 1710
set_server_title() (xmlrpc.server.DocCGIXMLRPCRequestHandler のメソッド), 1711
set_server_title() (xmlrpc.server.DocXMLRPCServer のメソッド), 1710
set_servername_callback (ssl.SSLContext の属性), 1270
set_spacing() (formatter.formatter のメソッド), 2381
set_start_method() (multiprocessing モジュール), 1011
set_startup_hook() (readline モジュール), 188
set_step() (bdb.Bdb のメソッド), 2095
set_subdir() (mailbox.MaildirMessage のメソッド), 1428
set_task_factory() (asyncio.loop のメソッド), 1147
set_terminator() (asynchat.async_chat のメソッド), 1308
set_threshold() (gc モジュール), 2245
set_trace() (bdb モジュール), 2098
set_trace() (bdb.Bdb のメソッド), 2096
set_trace() (pdb モジュール), 2103
set_trace() (pdb.Pdb のメソッド), 2104
set_trace_callback() (sqlite3.Connection のメソッド), 576
set_tunnel() (http.client.HTTPConnection のメソッド), 1608
set_type() (email.message.Message のメソッド), 1384
set_unittest_reportflags() (doctest モジュール), 1937
set_unixfrom() (email.message.EmailMessage のメソッド), 1331
set_unixfrom() (email.message.Message のメソッド), 1378
set_until() (bdb.Bdb のメソッド), 2096
set_url() (urllib.robotparser.RobotFileParser のメソッド), 1600
set_usage() (optparse.OptionParser のメソッド), 2458
set_userptr() (curses.panel.Panel のメソッド), 923
set_visible() (mailbox.BabylMessage のメソッド), 1433
set_wakeup_fd() (signal モジュール), 1316
set_write_buffer_limits() (asyncio.WriteTransport のメソッド), 1178
setacl() (imaplib.IMAP4 のメソッド), 1628
setannotation() (imaplib.IMAP4 のメソッド), 1628
setattr() (組み込み関数), 27
setAttribute() (xml.dom.Element のメソッド), 1495
setAttributeNode() (xml.dom.Element のメソッド), 1495
setAttributeNodeNS() (xml.dom.Element のメソッド), 1495
setAttributeNS() (xml.dom.Element のメソッド), 1495
SetBase() (xml.parsers.expat.xmlparser のメソッド), 1525
setblocking() (socket.socket のメソッド), 1235
setByteStream() (xml.sax.xmlreader.InputSource のメソッド), 1522
setcbreak() (tty モジュール), 2418
setCharacterStream() (xml.sax.xmlreader.InputSource のメソッド), 1522
setcheckinterval() (sys モジュール), 2182
setcomptype() (aifc.aifc のメソッド), 1733
setcomptype() (sunau.AU_write のメソッド), 1737
setcomptype() (wave.Wave_write のメソッド), 1740
setContentHandler() (xml.sax.xmlreader.XMLReader のメソッド), 1519
setcontext() (decimal モジュール), 391
setDaemon() (threading.Thread のメソッド), 983
setdefault() (dict のメソッド), 100
setdefault() (http.cookies.Morsel のメソッド), 1681

setdefaulttimeout() (*socket* モジュール), 1226
setdlopenflags() (*sys* モジュール), 2182
setDocumentLocator() (*xml.sax.handler.ContentHandler* のメソッド), 1513
setDTDHandler() (*xml.sax.xmlreader.XMLReader* のメソッド), 1520
setegid() (*os* モジュール), 711
setEncoding() (*xml.sax.xmlreader.InputSource* のメソッド), 1521
setEntityResolver() (*xml.sax.xmlreader.XMLReader* のメソッド), 1520
setErrorHandler() (*xml.sax.xmlreader.XMLReader* のメソッド), 1520
seteuid() (*os* モジュール), 711
setFeature() (*xml.sax.xmlreader.XMLReader* のメソッド), 1520
setfirstweekday() (*calendar* モジュール), 273
setfmt() (*ossaudiodev.oss_audio_device* のメソッド), 1748
setFormatter() (*logging.Handler* のメソッド), 849
setframerate() (*aifc.aifc* のメソッド), 1733
setframerate() (*sunau.AU_write* のメソッド), 1737
setframerate() (*wave.Wave_write* のメソッド), 1740
setgid() (*os* モジュール), 711
setgroups() (*os* モジュール), 712
seth() (*turtle* モジュール), 1783
setheading() (*turtle* モジュール), 1783
sethostname() (*socket* モジュール), 1226
SetInteger() (*msilib.Record* のメソッド), 2389
setitem() (*operator* モジュール), 465
setitimer() (*signal* モジュール), 1316
setLevel() (*logging.Handler* のメソッド), 849
setLevel() (*logging.Logger* のメソッド), 844
setlocale() (*locale* モジュール), 1765
setLocale() (*xml.sax.xmlreader.XMLReader* のメソッド), 1520
setLoggerClass() (*logging* モジュール), 861
setlogmask() (*syslog* モジュール), 2431
setLogRecordFactory() (*logging* モジュール), 862
setmark() (*aifc.aifc* のメソッド), 1733
setMaxConns() (*urllib.request.CacheFTPHandler* のメソッド), 1580
setmode() (*msvcrt* モジュール), 2393
setName() (*threading.Thread* のメソッド), 982
setnchannels() (*aifc.aifc* のメソッド), 1733
setnchannels() (*sunau.AU_write* のメソッド), 1737
setnchannels() (*wave.Wave_write* のメソッド), 1740
setnframes() (*aifc.aifc* のメソッド), 1733
setnframes() (*sunau.AU_write* のメソッド), 1737
setnframes() (*wave.Wave_write* のメソッド), 1740
SetParamEntityParsing() (*xml.parsers.expat.xmlparser* のメソッド), 1525
setparameters() (*ossaudiodev.oss_audio_device* のメソッド), 1749
setparams() (*aifc.aifc* のメソッド), 1733
setparams() (*sunau.AU_write* のメソッド), 1737
setparams() (*wave.Wave_write* のメソッド), 1740
setpassword() (*zipfile.ZipFile* のメソッド), 621
setpgid() (*os* モジュール), 712
setpgrp() (*os* モジュール), 712
setpos() (*aifc.aifc* のメソッド), 1732
setpos() (*sunau.AU_read* のメソッド), 1736
setpos() (*turtle* モジュール), 1782
setpos() (*wave.Wave_read* のメソッド), 1739
setposition() (*turtle* モジュール), 1782
setpriority() (*os* モジュール), 712
setprofile() (*sys* モジュール), 2182
setprofile() (*threading* モジュール), 979
SetProperty() (*msilib.SummaryInformation* のメソッド), 2388
setProperty() (*xml.sax.xmlreader.XMLReader* のメソッド), 1520
setPublicId() (*xml.sax.xmlreader.InputSource* のメソッド), 1521
setquota() (*imaplib.IMAP4* のメソッド), 1628
setraw() (*tty* モジュール), 2418
setrecursionlimit() (*sys* モジュール), 2183
setregid() (*os* モジュール), 712
SetReparseDeferralEnabled() (*xml.parsers.expat.xmlparser* のメソッド), 1526
setresgid() (*os* モジュール), 712
setresuid() (*os* モジュール), 713
setreuid() (*os* モジュール), 713
setrlimit() (*resource* モジュール), 2425
setsampwidth() (*aifc.aifc* のメソッド), 1733
setsampwidth() (*sunau.AU_write* のメソッド), 1737
setsampwidth() (*wave.Wave_write* のメソッド), 1740
setscrreg() (*curses.window* のメソッド), 910
setsid() (*os* モジュール), 713
setsockopt() (*socket.socket* のメソッド), 1236
setstate() (*codecs.IncrementalDecoder* のメソッド), 209
setstate() (*codecs.IncrementalEncoder* のメソッド), 208
setstate() (*random* モジュール), 414
setStream() (*logging.StreamHandler* のメソッド), 877
SetStream() (*msilib.Record* のメソッド), 2388
SetString() (*msilib.Record* のメソッド), 2388
setswitchinterval() (*sys* モジュール), 2183
setswitchinterval() (*test.support* モジュール), 2072
setSystemId() (*xml.sax.xmlreader.InputSource* のメソッド), 1521
setsyx() (*curses* モジュール), 901
setTarget() (*logging.handlers.MemoryHandler* のメソッド), 890
settiltangle() (*turtle* モジュール), 1796
settimeout() (*socket.socket* のメソッド), 1236
setTimeout() (*urllib.request.CacheFTPHandler* のメソッド), 1580
settrace() (*sys* モジュール), 2184
settrace() (*threading* モジュール), 979
setuid() (*os* モジュール), 713
setundobuffer() (*turtle* モジュール), 1800
setup() (*socketserver.BaseRequestHandler* のメソッド), 1666
setup() (*turtle* モジュール), 1809
setUp() (*unittest.TestCase* のメソッド), 1960
--setup=S
    timeit command line option, 2123
SETUP_ANNOTATIONS (*opcode*), 2366
SETUP_ASYNC_WITH (*opcode*), 2365
setup_environ() (*wsgiref.handlers.BaseHandler* のメソッド), 1559
SETUP_FINALLY (*opcode*), 2371
setup_python() (*venv.EnvBuilder* のメソッド), 2151
setup_scripts() (*venv.EnvBuilder* のメソッド), 2151
setup_testing_defaults() (*wsgiref.util* モジュール), 1551
SETUP_WITH (*opcode*), 2367
setUpClass() (*unittest.TestCase* のメソッド), 1960
setupterm() (*curses* モジュール), 901
SetValue() (*winreg* モジュール), 2401
SetValueEx() (*winreg* モジュール), 2401
setworldcoordinates() (*turtle* モジュール), 1802
setx() (*turtle* モジュール), 1782
setxattr() (*os* モジュール), 753
sety() (*turtle* モジュール), 1783
SF_APPEND (*stat* モジュール), 506
SF_ARCHIVED (*stat* モジュール), 506
SF_IMMUTABLE (*stat* モジュール), 506
SF_MNOWAIT (*os* モジュール), 724
SF_NODISKIO (*os* モジュール), 724
SF_NOUNLINK (*stat* モジュール), 506
SF_SNAPSHOT (*stat* モジュール), 506
SF_SYNC (*os* モジュール), 724
shape (*memoryview* の属性), 94
Shape (*turtle* のクラス), 1810
shape() (*turtle* モジュール), 1794

shapesize() (*turtle* モジュール), 1795
shapetransform() (*turtle* モジュール), 1797
share() (*socket.socket* のメソッド), 1236
ShareableList (*multiprocessing.shared_memory* のクラス), 1052
ShareableList() (*multiprocessing.managers.SharedMemoryManager* のメソッド), 1051
Shared Memory, 1048
shared_ciphers() (*ssl.SSLSocket* のメソッド), 1262
SharedMemory (*multiprocessing.shared_memory* のクラス), 1048
SharedMemory() (*multiprocessing.managers.SharedMemoryManager* のメソッド), 1051
SharedMemoryManager (*multiprocessing.managers* のクラス), 1051
shearfactor() (*turtle* モジュール), 1795
Shelf (*shelve* のクラス), 559
shelve
    モジュール, 561
shelve (モジュール), 557
shield() (*asyncio* モジュール), 1109
shift() (*decimal.Context* のメソッド), 398
shift() (*decimal.Decimal* のメソッド), 390
shift_path_info() (*wsgiref.util* モジュール), 1551
shifting
    operations, 41
shlex (*shlex* のクラス), 1823
shlex (モジュール), 1822
shm (*multiprocessing.shared_memory.ShareableList* の属性), 1052
shortDescription() (*unittest.TestCase* のメソッド), 1970
shorten() (*textwrap* モジュール), 177
shouldFlush() (*logging.handlers.BufferingHandler* のメソッド), 889
shouldFlush() (*logging.handlers.MemoryHandler* のメソッド), 890
shouldStop (*unittest.TestResult* の属性), 1978
show() (*curses.panel.Panel* のメソッド), 923
show_code() (*dis* モジュール), 2359
showsyntaxerror() (*code.InteractiveInterpreter* のメソッド), 2274
showtraceback() (*code.InteractiveInterpreter* のメソッド), 2275
showturtle() (*turtle* モジュール), 1794
showwarning() (*warnings* モジュール), 2201
shuffle() (*random* モジュール), 415
shutdown() (*concurrent.futures.Executor* のメソッド), 1055
shutdown() (*imaplib.IMAP4* のメソッド), 1628
shutdown() (*logging* モジュール), 861
shutdown() (*multiprocessing.managers.BaseManager* のメソッド), 1021
shutdown() (*socketserver.BaseServer* のメソッド), 1664
shutdown() (*socket.socket* のメソッド), 1236
shutdown_asyncgens() (*asyncio.loop* のメソッド), 1145
shutil (モジュール), 519
side_effect (*unittest.mock.Mock* の属性), 1995
SIG_BLOCK (*signal* モジュール), 1313
SIG_DFL (*signal* モジュール), 1311
SIG_IGN (*signal* モジュール), 1311
SIG_SETMASK (*signal* モジュール), 1314
SIG_UNBLOCK (*signal* モジュール), 1313
SIGABRT (*signal* モジュール), 1311
SIGALRM (*signal* モジュール), 1311
SIGBREAK (*signal* モジュール), 1311
SIGBUS (*signal* モジュール), 1311
SIGCHLD (*signal* モジュール), 1311
SIGCLD (*signal* モジュール), 1311
SIGCONT (*signal* モジュール), 1311
SIGFPE (*signal* モジュール), 1312
SIGHUP (*signal* モジュール), 1312
SIGILL (*signal* モジュール), 1312
SIGINT (*signal* モジュール), 1312
siginterrupt() (*signal* モジュール), 1317
SIGKILL (*signal* モジュール), 1312
signal
    モジュール, 1099
signal (モジュール), 1310
signal() (*signal* モジュール), 1317
Signature (*inspect* のクラス), 2255
signature (*inspect.BoundArguments* の属性), 2259
signature() (*inspect* モジュール), 2255
sigpending() (*signal* モジュール), 1317
SIGPIPE (*signal* モジュール), 1312
SIGSEGV (*signal* モジュール), 1312
SIGTERM (*signal* モジュール), 1312
sigtimedwait() (*signal* モジュール), 1318
SIGUSR1 (*signal* モジュール), 1312
SIGUSR2 (*signal* モジュール), 1312
sigwait() (*signal* モジュール), 1318
sigwaitinfo() (*signal* モジュール), 1318
SIGWINCH (*signal* モジュール), 1312
Simple Mail Transfer Protocol, 1639
SimpleCookie (*http.cookies* のクラス), 1679
simplefilter() (*warnings* モジュール), 2202
SimpleHandler (*wsgiref.handlers* のクラス), 1558
SimpleHTTPRequestHandler (*http.server* のクラス), 1675
SimpleNamespace (*types* のクラス), 327
SimpleQueue (*multiprocessing* のクラス), 1008
SimpleQueue (*queue* のクラス), 1088
SimpleXMLRPCRequestHandler (*xmlrpc.server* のクラス), 1705
SimpleXMLRPCServer (*xmlrpc.server* のクラス), 1704
sin() (*cmath* モジュール), 374
sin() (*math* モジュール), 370
single dispatch, **2491**
SingleAddressHeader (*email.headerregistry* のクラス), 1361
singledispatch() (*functools* モジュール), 457
singledispatchmethod (*functools* のクラス), 459
sinh() (*cmath* モジュール), 375
sinh() (*math* モジュール), 371
SIO_KEEPALIVE_VALS (*socket* モジュール), 1217
SIO_LOOPBACK_FAST_PATH (*socket* モジュール), 1217
SIO_RCVALL (*socket* モジュール), 1217
site (モジュール), 2268
site command line option
    --user-base, 2272
    --user-site, 2272
site_maps() (*urllib.robotparser.RobotFileParser* のメソッド), 1600
sitecustomize
    モジュール, 2270
site-packages
    directory, 2268
sixtofour (*ipaddress.IPv6Address* の属性), 1715
size (*multiprocessing.shared_memory.SharedMemory* の属性), 1049
size (*struct.Struct* の属性), 200
size (*tarfile.TarInfo* の属性), 637
size (*tracemalloc.Statistic* の属性), 2139
size (*tracemalloc.StatisticDiff* の属性), 2139
size (*tracemalloc.Trace* の属性), 2140
size() (*ftplib.FTP* のメソッド), 1618
size() (*mmap.mmap* のメソッド), 1324
size_diff (*tracemalloc.StatisticDiff* の属性), 2139
Sized (*collections.abc* のクラス), 298
Sized (*typing* のクラス), 1904
sizeof() (*ctypes* モジュール), 967
SKIP (*doctest* モジュール), 1929
skip() (*chunk.Chunk* のメソッド), 1742
skip() (*unittest* モジュール), 1958
skip_unless_bind_unix_socket() (*test.support* モジュール), 2076
skip_unless_symlink() (*test.support* モジュール), 2076

skip_unless_xattr() (*test.support* モジュール), 2076
skipIf() (*unittest* モジュール), 1958
skipinitialspace (*csv.Dialect* の属性), 654
skipped (*unittest.TestResult* の属性), 1978
skippedEntity() (*xml.sax.handler.ContentHandler* のメソッド), 1515
SkipTest, 1958
skipTest() (*unittest.TestCase* のメソッド), 1961
skipUnless() (*unittest* モジュール), 1958
SLASH (*token* モジュール), 2339
SLASHEQUAL (*token* モジュール), 2340
slave() (*nntplib.NNTP* のメソッド), 1638
sleep() (*asyncio* モジュール), 1107
sleep() (*time* モジュール), 795
slice, **2491**
    assignment, 50
    operation, 47
    組み込み関数, 2374
slice (組み込みクラス), 27
SMALLEST (*test.support* モジュール), 2070
SMTP
    protocol, 1639
SMTP (*email.policy* モジュール), 1355
SMTP (*smtplib* のクラス), 1639
smtp_server (*smtpd.SMTPChannel* の属性), 1651
SMTP_SSL (*smtplib* のクラス), 1640
smtp_state (*smtpd.SMTPChannel* の属性), 1651
SMTPAuthenticationError, 1642
SMTPChannel (*smtpd* のクラス), 1650
SMTPConnectError, 1641
smtpd (モジュール), 1648
SMTPDataError, 1641
SMTPException, 1641
SMTPHandler (*logging.handlers* のクラス), 888
SMTPHeloError, 1641
smtplib (モジュール), 1639
SMTPNotSupportedError, 1642
SMTPRecipientsRefused, 1641
SMTPResponseException, 1641
SMTPSenderRefused, 1641
SMTPServer (*smtpd* のクラス), 1648
SMTPServerDisconnected, 1641
SMTPUTF8 (*email.policy* モジュール), 1355
Snapshot (*tracemalloc* のクラス), 2137
SND_ALIAS (*winsound* モジュール), 2407
SND_ASYNC (*winsound* モジュール), 2408
SND_FILENAME (*winsound* モジュール), 2407
SND_LOOP (*winsound* モジュール), 2407
SND_MEMORY (*winsound* モジュール), 2407
SND_NODEFAULT (*winsound* モジュール), 2408
SND_NOSTOP (*winsound* モジュール), 2408
SND_NOWAIT (*winsound* モジュール), 2408
SND_PURGE (*winsound* モジュール), 2408
sndhdr (モジュール), 1745
sni_callback (*ssl.SSLContext* の属性), 1269
sniff() (*csv.Sniffer* のメソッド), 653
Sniffer (*csv* のクラス), 653
sock_accept() (*asyncio.loop* のメソッド), 1156
SOCK_CLOEXEC (*socket* モジュール), 1215
sock_connect() (*asyncio.loop* のメソッド), 1156
SOCK_DGRAM (*socket* モジュール), 1215
SOCK_MAX_SIZE (*test.support* モジュール), 2069
SOCK_NONBLOCK (*socket* モジュール), 1215
SOCK_RAW (*socket* モジュール), 1215
SOCK_RDM (*socket* モジュール), 1215
sock_recv() (*asyncio.loop* のメソッド), 1155
sock_recv_into() (*asyncio.loop* のメソッド), 1156
sock_sendall() (*asyncio.loop* のメソッド), 1156
sock_sendfile() (*asyncio.loop* のメソッド), 1157
SOCK_SEQPACKET (*socket* モジュール), 1215
SOCK_STREAM (*socket* モジュール), 1215
socket
    オブジェクト, 1211
    モジュール, 1537
socket (*socketserver.BaseServer* の属性), 1664
socket (モジュール), 1211
socket() (*imaplib.IMAP4* のメソッド), 1629
socket() (*in module socket*), 1290
socket() (*socket* モジュール), 1219
socket_type (*socketserver.BaseServer* の属性), 1664
SocketHandler (*logging.handlers* のクラス), 883
socketpair() (*socket* モジュール), 1219
sockets (*asyncio.Server* の属性), 1166
socketserver (モジュール), 1660
SocketType (*socket* モジュール), 1221
SOL_ALG (*socket* モジュール), 1218
SOL_RDS (*socket* モジュール), 1217
SOMAXCONN (*socket* モジュール), 1216
sort() (*imaplib.IMAP4* のメソッド), 1629
sort() (*list* のメソッド), 51
sort_stats() (*pstats.Stats* のメソッド), 2115
sortdict() (*test.support* モジュール), 2071
sorted() (組み込み関数), 28
--sort-keys
    json.tool command line option, 1414
sortTestMethodsUsing (*unittest.TestLoader* の属性), 1977
source (*doctest.Example* の属性), 1939
source (*pdb command*), 2108
source (*shlex.shlex* の属性), 1826
SOURCE_DATE_EPOCH, 2351, 2354
source_from_cache() (*imp* モジュール), 2470
source_from_cache() (*importlib.util* モジュール), 2310
source_hash() (*importlib.util* モジュール), 2312
SOURCE_SUFFIXES (*importlib.machinery* モジュール), 2303
source_to_code() (*importlib.abc.InspectLoader* の静的メソッド), 2299
SourceFileLoader (*importlib.machinery* のクラス), 2306
sourcehook() (*shlex.shlex* のメソッド), 1824
SourcelessFileLoader (*importlib.machinery* のクラス), 2307
SourceLoader (*importlib.abc* のクラス), 2300
space
    in printf-style formatting, 66, 84
    in string formatting, 130
span() (*re.Match* のメソッド), 156
spawn() (*pty* モジュール), 2419
spawn_python() (*test.support.script_helper* モジュール), 2085
spawnl() (*os* モジュール), 761
spawnle() (*os* モジュール), 761
spawnlp() (*os* モジュール), 761
spawnlpe() (*os* モジュール), 761
spawnv() (*os* モジュール), 761
spawnve() (*os* モジュール), 761
spawnvp() (*os* モジュール), 761
spawnvpe() (*os* モジュール), 761
spec_from_file_location() (*importlib.util* モジュール), 2312
spec_from_loader() (*importlib.util* モジュール), 2312
special
    method, 2491
special method, **2491**
SpecialFileError, 631
specified_attributes (*xml.parsers.expat.xmlparser* の属性), 1527
speed() (*ossaudiodev.oss_audio_device* のメソッド), 1748
speed() (*turtle* モジュール), 1786
Spinbox (*tkinter.ttk* のクラス), 1852
split() (*bytearray* のメソッド), 76
split() (*bytes* のメソッド), 76
split() (*os.path* モジュール), 497
split() (*re* モジュール), 149
split() (*re.Pattern* のメソッド), 153
split() (*shlex* モジュール), 1822
split() (*str* のメソッド), 62
splitdrive() (*os.path* モジュール), 497

splitext() (*os.path* モジュール), 497
splitlines() (*bytearray* のメソッド), 81
splitlines() (*bytes* のメソッド), 81
splitlines() (*str* のメソッド), 62
SplitResult (*urllib.parse* のクラス), 1595
SplitResultBytes (*urllib.parse* のクラス), 1596
SpooledTemporaryFile() (*tempfile* モジュール), 511
sprintf-style formatting, 65, 83
spwd (モジュール), 2412
sqlite3 (モジュール), 568
sqlite_version (*sqlite3* モジュール), 570
sqlite_version_info (*sqlite3* モジュール), 570
sqrt() (*cmath* モジュール), 374
sqrt() (*decimal.Context* のメソッド), 398
sqrt() (*decimal.Decimal* のメソッド), 390
sqrt() (*math* モジュール), 369
SSL, 1242
ssl (モジュール), 1242
SSL_CERT_FILE, 1287
SSL_CERT_PATH, 1287
ssl_version (*ftplib.FTP_TLS* の属性), 1618
SSLCertVerificationError, 1246
SSLContext (*ssl* のクラス), 1264
SSLEOFError, 1246
SSLError, 1245
SSLErrorNumber (*ssl* のクラス), 1258
SSLKEYLOGFILE, 1244, 1245
SSLObject (*ssl* のクラス), 1282
sslobject_class (*ssl.SSLContext* の属性), 1272
SSLSession (*ssl* のクラス), 1285
SSLSocket (*ssl* のクラス), 1259
sslsocket_class (*ssl.SSLContext* の属性), 1271
SSLSyscallError, 1246
SSLv3 (*ssl.TLSVersion* の属性), 1259
SSLWantReadError, 1245
SSLWantWriteError, 1245
SSLZeroReturnError, 1245
st() (*turtle* モジュール), 1794
st2list() (*parser* モジュール), 2323
st2tuple() (*parser* モジュール), 2323
st_atime (*os.stat_result* の属性), 743
ST_ATIME (*stat* モジュール), 503
st_atime_ns (*os.stat_result* の属性), 743
st_birthtime (*os.stat_result* の属性), 744
st_blksize (*os.stat_result* の属性), 744
st_blocks (*os.stat_result* の属性), 744
st_creator (*os.stat_result* の属性), 745
st_ctime (*os.stat_result* の属性), 743
ST_CTIME (*stat* モジュール), 503
st_ctime_ns (*os.stat_result* の属性), 743
st_dev (*os.stat_result* の属性), 743
ST_DEV (*stat* モジュール), 503
st_file_attributes (*os.stat_result* の属性), 745
st_flags (*os.stat_result* の属性), 744
st_fstype (*os.stat_result* の属性), 744
st_gen (*os.stat_result* の属性), 744
st_gid (*os.stat_result* の属性), 743
ST_GID (*stat* モジュール), 503
st_ino (*os.stat_result* の属性), 743
ST_INO (*stat* モジュール), 503
st_mode (*os.stat_result* の属性), 743
ST_MODE (*stat* モジュール), 503
st_mtime (*os.stat_result* の属性), 743
ST_MTIME (*stat* モジュール), 503
st_mtime_ns (*os.stat_result* の属性), 743
st_nlink (*os.stat_result* の属性), 743
ST_NLINK (*stat* モジュール), 503
st_rdev (*os.stat_result* の属性), 744
st_reparse_tag (*os.stat_result* の属性), 745
st_rsize (*os.stat_result* の属性), 744
st_size (*os.stat_result* の属性), 743
ST_SIZE (*stat* モジュール), 503
st_type (*os.stat_result* の属性), 745
st_uid (*os.stat_result* の属性), 743
ST_UID (*stat* モジュール), 503
stack (*traceback.TracebackException* の属性), 2238
stack viewer, 1879
stack() (*inspect* モジュール), 2264
stack_effect() (*dis* モジュール), 2361
stack_size() (*_thread* モジュール), 1097
stack_size() (*threading* モジュール), 979
stackable
    streams, 200
StackSummary (*traceback* のクラス), 2239
stamp() (*turtle* モジュール), 1785
standard_b64decode() (*base64* モジュール), 1443
standard_b64encode() (*base64* モジュール), 1443
standarderror (*2to3 fixer*), 2064
standend() (*curses.window* のメソッド), 910
standout() (*curses.window* のメソッド), 910
STAR (*token* モジュール), 2339
STAREQUAL (*token* モジュール), 2340
starmap() (*itertools* モジュール), 445
starmap() (*multiprocessing.pool.Pool* のメソッド), 1030
starmap_async() (*multiprocessing.pool.Pool* のメソッド), 1030
start (*range* の属性), 54
start (*UnicodeError* の属性), 120
start() (*logging.handlers.QueueListener* のメソッド), 892
start() (*multiprocessing.managers.BaseManager* のメソッド), 1021
start() (*multiprocessing.Process* のメソッド), 1002
start() (*re.Match* のメソッド), 156
start() (*threading.Thread* のメソッド), 981
start() (*tkinter.ttk.Progressbar* のメソッド), 1856
start() (*tracemalloc* モジュール), 2135
start() (*xml.etree.ElementTree.TreeBuilder* のメソッド), 1482
start_color() (*curses* モジュール), 901
start_component() (*msilib.Directory* のメソッド), 2390
start_new_thread() (*_thread* モジュール), 1096
start_ns() (*xml.etree.ElementTree.TreeBuilder* のメソッド), 1483
start_server() (*asyncio* モジュール), 1119
start_serving() (*asyncio.Server* のメソッド), 1165
start_threads() (*test.support* モジュール), 2076
start_tls() (*asyncio.loop* のメソッド), 1154
start_unix_server() (*asyncio* モジュール), 1120
StartCdataSectionHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1529
--start-directory directory
    unittest-discover command line option, 1953
StartDoctypeDeclHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1528
startDocument() (*xml.sax.handler.ContentHandler* のメソッド), 1513
startElement() (*xml.sax.handler.ContentHandler* のメソッド), 1514
StartElementHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1528
startElementNS() (*xml.sax.handler.ContentHandler* のメソッド), 1514
STARTF_USESHOWWINDOW (*subprocess* モジュール), 1076
STARTF_USESTDHANDLES (*subprocess* モジュール), 1076
startfile() (*os* モジュール), 763
StartNamespaceDeclHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1529
startPrefixMapping() (*xml.sax.handler.ContentHandler* のメソッド), 1514
startswith() (*bytearray* のメソッド), 74
startswith() (*bytes* のメソッド), 74
startswith() (*str* のメソッド), 63
startTest() (*unittest.TestResult* のメソッド), 1979
startTestRun() (*unittest.TestResult* のメソッド), 1979
starttls() (*imaplib.IMAP4* のメソッド), 1629
starttls() (*nntplib.NNTP* のメソッド), 1634

starttls() (*smtplib.SMTP* のメソッド), 1644
STARTUPINFO (*subprocess* のクラス), 1075
stat
    モジュール, 742
stat (モジュール), 501
stat() (*nntplib.NNTP* のメソッド), 1637
stat() (*os* モジュール), 742
stat() (*os.DirEntry* のメソッド), 741
stat() (*pathlib.Path* のメソッド), 483
stat() (*poplib.POP3* のメソッド), 1621
stat_result (*os* のクラス), 742
state() (*tkinter.ttk.Widget* のメソッド), 1849
statement, **2491**
staticmethod() (組み込み関数), 28
Statistic (*tracemalloc* のクラス), 2139
StatisticDiff (*tracemalloc* のクラス), 2139
statistics (モジュール), 421
statistics() (*tracemalloc.Snapshot* のメソッド), 2138
StatisticsError, 429
Stats (*pstats* のクラス), 2115
status (*http.client.HTTPResponse* の属性), 1610
status() (*imaplib.IMAP4* のメソッド), 1629
statvfs() (*os* モジュール), 745
STD_ERROR_HANDLE (*subprocess* モジュール), 1076
STD_INPUT_HANDLE (*subprocess* モジュール), 1076
STD_OUTPUT_HANDLE (*subprocess* モジュール), 1076
StdButtonBox (*tkinter.tix* のクラス), 1870
stderr (*asyncio.asyncio.subprocess.Process* の属性), 1136
stderr (*subprocess.CalledProcessError* の属性), 1065
stderr (*subprocess.CompletedProcess* の属性), 1064
stderr (*subprocess.Popen* の属性), 1074
stderr (*subprocess.TimeoutExpired* の属性), 1065
stderr (*sys* モジュール), 2186
stdev (*statistics.NormalDist* の属性), 430
stdev() (*statistics* モジュール), 427
stdin (*asyncio.asyncio.subprocess.Process* の属性), 1136
stdin (*subprocess.Popen* の属性), 1074
stdin (*sys* モジュール), 2186
stdout (*asyncio.asyncio.subprocess.Process* の属性), 1136
STDOUT (*subprocess* モジュール), 1064
stdout (*subprocess.CalledProcessError* の属性), 1065
stdout (*subprocess.CompletedProcess* の属性), 1064
stdout (*subprocess.Popen* の属性), 1074
stdout (*subprocess.TimeoutExpired* の属性), 1065
stdout (*sys* モジュール), 2186
step (*pdb command*), 2107
step (*range* の属性), 54
step() (*tkinter.ttk.Progressbar* のメソッド), 1856
stereocontrols() (*ossaudiodev.oss_mixer_device* のメソッド), 1750
stls() (*poplib.POP3* のメソッド), 1622
stop (*range* の属性), 54
stop() (*asyncio.loop* のメソッド), 1144
stop() (*logging.handlers.QueueListener* のメソッド), 893
stop() (*tkinter.ttk.Progressbar* のメソッド), 1856
stop() (*tracemalloc* モジュール), 2135
stop() (*unittest.TestResult* のメソッド), 1979
stop_here() (*bdb.Bdb* のメソッド), 2095
StopAsyncIteration, 118
StopIteration, 117
stopListening() (*logging.config* モジュール), 866
stopTest() (*unittest.TestResult* のメソッド), 1979
stopTestRun() (*unittest.TestResult* のメソッド), 1979
storbinary() (*ftplib.FTP* のメソッド), 1616
store() (*imaplib.IMAP4* のメソッド), 1629
STORE_ACTIONS (*optparse.Option* の属性), 2465
STORE_ATTR (*opcode*), 2368
STORE_DEREF (*opcode*), 2372
STORE_FAST (*opcode*), 2371
STORE_GLOBAL (*opcode*), 2368
STORE_NAME (*opcode*), 2368
STORE_SUBSCR (*opcode*), 2365
storlines() (*ftplib.FTP* のメソッド), 1616
str (*built-in class*)
    (see also string), 55
str (組み込みクラス), 55
str() (*locale* モジュール), 1771
strcoll() (*locale* モジュール), 1770
StreamError, 631
StreamHandler (*logging* のクラス), 877
StreamReader (*asyncio* のクラス), 1120
StreamReader (*codecs* のクラス), 210
streamreader (*codecs.CodecInfo* の属性), 201
StreamReaderWriter (*codecs* のクラス), 212
StreamRecoder (*codecs* のクラス), 212
StreamRequestHandler (*socketserver* のクラス), 1666
streams, 200
    stackable, 200
StreamWriter (*asyncio* のクラス), 1121
StreamWriter (*codecs* のクラス), 210
streamwriter (*codecs.CodecInfo* の属性), 201
strerror (*OSError* の属性), 117
strerror() (*os* モジュール), 713
strftime() (*datetime.date* のメソッド), 236
strftime() (*datetime.datetime* のメソッド), 249
strftime() (*datetime.time* のメソッド), 255
strftime() (*time* モジュール), 795
strict (*csv.Dialect* の属性), 655
strict (*email.policy* モジュール), 1355
strict_domain (*http.cookiejar.DefaultCookiePolicy* の属性), 1691
strict_errors() (*codecs* モジュール), 206
strict_ns_domain (*http.cookiejar.DefaultCookiePolicy* の属性), 1691
strict_ns_set_initial_dollar (*http.cookiejar.DefaultCookiePolicy* の属性), 1691
strict_ns_set_path (*http.cookiejar.DefaultCookiePolicy* の属性), 1691
strict_ns_unverifiable (*http.cookiejar.DefaultCookiePolicy* の属性), 1691
strict_rfc2965_unverifiable (*http.cookiejar.DefaultCookiePolicy* の属性), 1691
strides (*memoryview* の属性), 94
string
    format() (*built-in function*), 14
    formatting, printf, 65
    interpolation, printf, 65
    methods, 56
    str (*built-in class*), 55
    str() (*built-in function*), 28
    text sequence type, 55
    オブジェクト, 55
    モジュール, 1771
string (*re.Match* の属性), 157
STRING (*token* モジュール), 2338
string (モジュール), 125
string_at() (*ctypes* モジュール), 967
StringIO (*io* のクラス), 788
stringprep (モジュール), 183
strip() (*bytearray* のメソッド), 77
strip() (*bytes* のメソッド), 77
strip() (*str* のメソッド), 63
strip_dirs() (*pstats.Stats* のメソッド), 2115
strip_python_strerr() (*test.support* モジュール), 2074
stripspaces (*curses.textpad.Textbox* の属性), 918
strptime() (*datetime.datetime* のクラスメソッド), 241
strptime() (*time* モジュール), 797
strsignal() (*signal* モジュール), 1314
struct
    モジュール, 1236
Struct (*struct* のクラス), 199
struct (モジュール), 193
struct_time (*time* のクラス), 797
Structure (*ctypes* のクラス), 972
structures
    C, 193

strxfrm() (*locale* モジュール), 1770
STType (*parser* モジュール), 2325
Style (*tkinter.ttk* のクラス), 1864
sub() (*operator* モジュール), 464
sub() (*re* モジュール), 150
sub() (*re.Pattern* のメソッド), 153
subdirs (*filecmp.dircmp* の属性), 509
SubElement() (*xml.etree.ElementTree* モジュール), 1473
submit() (*concurrent.futures.Executor* のメソッド), 1054
submodule_search_locations (*importlib.machinery.ModuleSpec* の属性), 2309
subn() (*re* モジュール), 151
subn() (*re.Pattern* のメソッド), 153
subnet_of() (*ipaddress.IPv4Network* のメソッド), 1720
subnet_of() (*ipaddress.IPv6Network* のメソッド), 1722
subnets() (*ipaddress.IPv4Network* のメソッド), 1719
subnets() (*ipaddress.IPv6Network* のメソッド), 1722
Subnormal (*decimal* のクラス), 401
suboffsets (*memoryview* の属性), 94
subpad() (*curses.window* のメソッド), 910
subprocess (モジュール), 1062
subprocess_exec() (*asyncio.loop* のメソッド), 1162
subprocess_shell() (*asyncio.loop* のメソッド), 1163
SubprocessError, 1064
SubprocessProtocol (*asyncio* のクラス), 1180
SubprocessTransport (*asyncio* のクラス), 1175
subscribe() (*imaplib.IMAP4* のメソッド), 1630
subscript
    assignment, 50
    operation, 47
subsequent_indent (*textwrap.TextWrapper* の属性), 179
substitute() (*string.Template* のメソッド), 137
subTest() (*unittest.TestCase* のメソッド), 1961
subtract() (*collections.Counter* のメソッド), 279
subtract() (*decimal.Context* のメソッド), 398
subtype (*email.headerregistry.ContentTypeHeader* の属性), 1362
subwin() (*curses.window* のメソッド), 910
successful() (*multiprocessing.pool.AsyncResult* のメソッド), 1031
suffix_map (*mimetypes* モジュール), 1440
suffix_map (*mimetypes.MimeTypes* の属性), 1441
suite() (*parser* モジュール), 2322
suiteClass (*unittest.TestLoader* の属性), 1977
sum() (組み込み関数), 29
summarize() (*doctest.DocTestRunner* のメソッド), 1943
summarize_address_range() (*ipaddress* モジュール), 1725
--summary
    trace command line option, 2127
sunau (モジュール), 1734
super (*pyclbr.Class* の属性), 2350
super() (組み込み関数), 29
supernet() (*ipaddress.IPv4Network* のメソッド), 1720
supernet() (*ipaddress.IPv6Network* のメソッド), 1722
supernet_of() (*ipaddress.IPv4Network* のメソッド), 1720
supernet_of() (*ipaddress.IPv6Network* のメソッド), 1722
supports_bytes_environ (*os* モジュール), 713
supports_dir_fd (*os* モジュール), 746
supports_effective_ids (*os* モジュール), 746
supports_fd (*os* モジュール), 747
supports_follow_symlinks (*os* モジュール), 747
supports_unicode_filenames (*os.path* モジュール), 497
SupportsAbs (*typing* のクラス), 1904
SupportsBytes (*typing* のクラス), 1904
SupportsComplex (*typing* のクラス), 1904
SupportsFloat (*typing* のクラス), 1904
SupportsIndex (*typing* のクラス), 1904
SupportsInt (*typing* のクラス), 1903
SupportsRound (*typing* のクラス), 1904
suppress() (*contextlib* モジュール), 2216
SuppressCrashReport (*test.support* のクラス), 2083
SW_HIDE (*subprocess* モジュール), 1076
swap_attr() (*test.support* モジュール), 2075
swap_item() (*test.support* モジュール), 2075
swapcase() (*bytearray* のメソッド), 81
swapcase() (*bytes* のメソッド), 81
swapcase() (*str* のメソッド), 64
sym_name (*symbol* モジュール), 2337
Symbol (*symtable* のクラス), 2336
symbol (モジュール), 2337
SymbolTable (*symtable* のクラス), 2335
symlink() (*os* モジュール), 747
symlink_to() (*pathlib.Path* のメソッド), 489
symmetric_difference() (*frozenset* のメソッド), 96
symmetric_difference_update() (*frozenset* のメソッド), 97
symtable (モジュール), 2334
symtable() (*symtable* モジュール), 2335
sync() (*dbm.dumb.dumbdbm* のメソッド), 568
sync() (*dbm.gnu.gdbm* のメソッド), 565
sync() (*os* モジュール), 748
sync() (*ossaudiodev.oss_audio_device* のメソッド), 1749
sync() (*shelve.Shelf* のメソッド), 558
syncdown() (*curses.window* のメソッド), 910
synchronized() (*multiprocessing.sharedctypes* モジュール), 1019
SyncManager (*multiprocessing.managers* のクラス), 1022
syncok() (*curses.window* のメソッド), 910
syncup() (*curses.window* のメソッド), 910
SyntaxErr, 1498
SyntaxError, 118
SyntaxWarning, 122
sys
    モジュール, 23
sys (モジュール), 2165
sys_exc (*2to3 fixer*), 2064
sys_version (*http.server.BaseHTTPRequestHandler* の属性), 1672
sysconf() (*os* モジュール), 770
sysconf_names (*os* モジュール), 770
sysconfig (モジュール), 2189
syslog (モジュール), 2431
syslog() (*syslog* モジュール), 2431
SysLogHandler (*logging.handlers* のクラス), 885
system() (*os* モジュール), 763
system() (*platform* モジュール), 925
system_alias() (*platform* モジュール), 925
system_must_validate_cert() (*test.support* モジュール), 2071
SystemError, 118
SystemExit, 119
systemId (*xml.dom.DocumentType* の属性), 1493
SystemRandom (*random* のクラス), 418
SystemRandom (*secrets* のクラス), 702
SystemRoot, 1070

T

-T
    trace command line option, 2127
-t
    trace command line option, 2127
    unittest-discover command line option, 1953
-t <tarfile>
    tarfile command line option, 644
-t <zipfile>
    zipfile command line option, 627
T_FMT (*locale* モジュール), 1767
T_FMT_AMPM (*locale* モジュール), 1767
tab() (*tkinter.ttk.Notebook* のメソッド), 1855
TabError, 118
tabnanny (モジュール), 2347
tabs() (*tkinter.ttk.Notebook* のメソッド), 1855
tabsize (*textwrap.TextWrapper* の属性), 179
tabular
    data, 649

tag (*xml.etree.ElementTree.Element* の属性), 1476
tag_bind() (*tkinter.ttk.Treeview* のメソッド), 1864
tag_configure() (*tkinter.ttk.Treeview* のメソッド), 1864
tag_has() (*tkinter.ttk.Treeview* のメソッド), 1864
tagName (*xml.dom.Element* の属性), 1494
tail (*xml.etree.ElementTree.Element* の属性), 1476
take_snapshot() (*tracemalloc* モジュール), 2135
takewhile() (*itertools* モジュール), 445
tan() (*cmath* モジュール), 374
tan() (*math* モジュール), 370
tanh() (*cmath* モジュール), 375
tanh() (*math* モジュール), 371
tar_filter() (*tarfile* モジュール), 641
TarError, 630
TarFile (*tarfile* のクラス), 632
tarfile (モジュール), 628
tarfile command line option
    -c <tarfile> <source1> ... <sourceN>, 644
    --create <tarfile> <source1> ... <sourceN>, 644
    -e <tarfile> [<output_dir>], 644
    --extract <tarfile> [<output_dir>], 644
    --filter <filtername>, 645
    -l <tarfile>, 644
    --list <tarfile>, 644
    -t <tarfile>, 644
    --test <tarfile>, 644
    -v, 645
    --verbose, 645
target (*xml.dom.ProcessingInstruction* の属性), 1497
TarInfo (*tarfile* のクラス), 637
tarinfo (*tarfile.FilterError* の属性), 631
Task (*asyncio* のクラス), 1114
task_done() (*asyncio.Queue* のメソッド), 1139
task_done() (*multiprocessing.JoinableQueue* のメソッド), 1009
task_done() (*queue.Queue* のメソッド), 1089
tau (*cmath* モジュール), 376
tau (*math* モジュール), 372
tb_locals (*unittest.TestResult* の属性), 1979
tbreak (*pdb command*), 2105
tcdrain() (*termios* モジュール), 2417
tcflow() (*termios* モジュール), 2417
tcflush() (*termios* モジュール), 2417
tcgetattr() (*termios* モジュール), 2416
tcgetpgrp() (*os* モジュール), 724
Tcl() (*tkinter* モジュール), 1832
TCPServer (*socketserver* のクラス), 1660
tcsendbreak() (*termios* モジュール), 2417
tcsetattr() (*termios* モジュール), 2417
tcsetpgrp() (*os* モジュール), 724
tearDown() (*unittest.TestCase* のメソッド), 1960
tearDownClass() (*unittest.TestCase* のメソッド), 1961
tee() (*itertools* モジュール), 446
tell() (*aifc.aifc* のメソッド), 1733
tell() (*chunk.Chunk* のメソッド), 1742
tell() (*io.IOBase* のメソッド), 779
tell() (*io.TextIOBase* のメソッド), 786
tell() (*mmap.mmap* のメソッド), 1324
tell() (*sunau.AU_read* のメソッド), 1736
tell() (*sunau.AU_write* のメソッド), 1737
tell() (*wave.Wave_read* のメソッド), 1739
tell() (*wave.Wave_write* のメソッド), 1740
Telnet (*telnetlib* のクラス), 1652
telnetlib (モジュール), 1652
TEMP, 513
temp_cwd() (*test.support* モジュール), 2075
temp_dir() (*test.support* モジュール), 2074
temp_umask() (*test.support* モジュール), 2075
tempdir (*tempfile* モジュール), 514
tempfile (モジュール), 510
Template (*pipes* のクラス), 2423
Template (*string* のクラス), 137
template (*string.Template* の属性), 137
temporary
    file, 510
    file name, 510
TemporaryDirectory() (*tempfile* モジュール), 511
TemporaryFile() (*tempfile* モジュール), 510
teredo (*ipaddress.IPv6Address* の属性), 1715
TERM, 901
termattrs() (*curses* モジュール), 901
terminal_size (*os* のクラス), 726
terminate() (*asyncio.asyncio.subprocess.Process* のメソッド), 1135
terminate() (*asyncio.SubprocessTransport* のメソッド), 1180
terminate() (*multiprocessing.pool.Pool* のメソッド), 1031
terminate() (*multiprocessing.Process* のメソッド), 1004
terminate() (*subprocess.Popen* のメソッド), 1074
termios (モジュール), 2416
termname() (*curses* モジュール), 901
test (*doctest.DocTestFailure* の属性), 1946
test (*doctest.UnexpectedException* の属性), 1947
test (モジュール), 2065
--test <tarfile>
    tarfile command line option, 644
--test <zipfile>
    zipfile command line option, 627
test() (*cgi* モジュール), 1545
TEST_DATA_DIR (*test.support* モジュール), 2070
TEST_HOME_DIR (*test.support* モジュール), 2070
TEST_HTTP_URL (*test.support* モジュール), 2070
TEST_SUPPORT_DIR (*test.support* モジュール), 2069
TestCase (*unittest* のクラス), 1960
TestFailed, 2068
testfile() (*doctest* モジュール), 1932
TESTFN (*test.support* モジュール), 2069
TESTFN_ENCODING (*test.support* モジュール), 2069
TESTFN_NONASCII (*test.support* モジュール), 2069
TESTFN_UNDECODABLE (*test.support* モジュール), 2069
TESTFN_UNENCODABLE (*test.support* モジュール), 2069
TESTFN_UNICODE (*test.support* モジュール), 2069
TestHandler (*test.support* のクラス), 2084
TestLoader (*unittest* のクラス), 1975
testMethodPrefix (*unittest.TestLoader* の属性), 1977
testmod() (*doctest* モジュール), 1933
testNamePatterns (*unittest.TestLoader* の属性), 1977
TestResult (*unittest* のクラス), 1978
tests (*imghdr* モジュール), 1744
testsource() (*doctest* モジュール), 1945
testsRun (*unittest.TestResult* の属性), 1978
TestSuite (*unittest* のクラス), 1973
test.support (モジュール), 2068
test.support.script_helper (モジュール), 2085
testzip() (*zipfile.ZipFile* のメソッド), 621
text (*msilib* モジュール), 2392
text (*SyntaxError* の属性), 118
text (*traceback.TracebackException* の属性), 2238
Text (*typing* のクラス), 1908
text (*xml.etree.ElementTree.Element* の属性), 1476
text encoding, **2491**
text file, **2491**
text mode, 23
text() (*cgitb* モジュール), 1549
text() (*msilib.Dialog* のメソッド), 2391
text_factory (*sqlite3.Connection* の属性), 578
Textbox (*curses.textpad* のクラス), 917
TextCalendar (*calendar* のクラス), 270
textdomain() (*gettext* モジュール), 1754
textdomain() (*locale* モジュール), 1773
textinput() (*turtle* モジュール), 1806
TextIO (*typing* のクラス), 1908
TextIOBase (*io* のクラス), 785
TextIOWrapper (*io* のクラス), 787
TextTestResult (*unittest* のクラス), 1980
TextTestRunner (*unittest* のクラス), 1981

textwrap (モジュール), 176
TextWrapper (*textwrap* のクラス), 178
theme_create() (*tkinter.ttk.Style* のメソッド), 1867
theme_names() (*tkinter.ttk.Style* のメソッド), 1868
theme_settings() (*tkinter.ttk.Style* のメソッド), 1867
theme_use() (*tkinter.ttk.Style* のメソッド), 1868
THOUSEP (*locale* モジュール), 1768
Thread (*threading* のクラス), 981
thread() (*imaplib.IMAP4* のメソッド), 1630
thread_info (*sys* モジュール), 2187
thread_time() (*time* モジュール), 798
thread_time_ns() (*time* モジュール), 799
ThreadedChildWatcher (*asyncio* のクラス), 1193
threading (モジュール), 977
threading_cleanup() (*test.support* モジュール), 2079
threading_setup() (*test.support* モジュール), 2079
ThreadingHTTPServer (*http.server* のクラス), 1671
ThreadingMixIn (*socketserver* のクラス), 1662
ThreadingTCPServer (*socketserver* のクラス), 1662
ThreadingUDPServer (*socketserver* のクラス), 1662
ThreadPool (*multiprocessing.pool* のクラス), 1037
ThreadPoolExecutor (*concurrent.futures* のクラス), 1056
threads
    POSIX, 1096
throw (*2to3 fixer*), 2064
ticket_lifetime_hint (*ssl.SSLSession* の属性), 1285
tigetflag() (*curses* モジュール), 901
tigetnum() (*curses* モジュール), 901
tigetstr() (*curses* モジュール), 901
TILDE (*token* モジュール), 2340
tilt() (*turtle* モジュール), 1796
tiltangle() (*turtle* モジュール), 1796
time (*datetime* のクラス), 251
time (*ssl.SSLSession* の属性), 1285
time (モジュール), 790
time() (*asyncio.loop* のメソッド), 1147
time() (*datetime.datetime* のメソッド), 244
time() (*time* モジュール), 798
Time2Internaldate() (*imaplib* モジュール), 1624
time_ns() (*time* モジュール), 799
timedelta (*datetime* のクラス), 228
TimedRotatingFileHandler (*logging.handlers* のクラス), 881
timegm() (*calendar* モジュール), 274
timeit (モジュール), 2120
timeit command line option
    -h, 2124
    --help, 2124
    -n N, 2123
    --number=N, 2123
    -p, 2123
    --process, 2123
    -r N, 2123
    --repeat=N, 2123
    -s S, 2123
    --setup=S, 2123
    -u, 2123
    --unit=U, 2123
    -v, 2124
    --verbose, 2124
timeit() (*timeit* モジュール), 2121
timeit() (*timeit.Timer* のメソッド), 2122
timeout, 1215
timeout (*socketserver.BaseServer* の属性), 1664
timeout (*ssl.SSLSession* の属性), 1285
timeout (*subprocess.TimeoutExpired* の属性), 1064
timeout() (*curses.window* のメソッド), 910
TIMEOUT_MAX (*_thread* モジュール), 1098
TIMEOUT_MAX (*threading* モジュール), 979
TimeoutError, 122, 1005, 1061, 1141
TimeoutExpired, 1064
Timer (*threading* のクラス), 991
Timer (*timeit* のクラス), 2121
TimerHandle (*asyncio* のクラス), 1164
times() (*os* モジュール), 764
TIMESTAMP (*py_compile.PycInvalidationMode* の属性), 2352
timestamp() (*datetime.datetime* のメソッド), 246
timetuple() (*datetime.date* のメソッド), 235
timetuple() (*datetime.datetime* のメソッド), 245
timetz() (*datetime.datetime* のメソッド), 244
timezone (*datetime* のクラス), 264
timezone (*time* モジュール), 802
--timing
    trace command line option, 2127
title() (*bytearray* のメソッド), 82
title() (*bytes* のメソッド), 82
title() (*str* のメソッド), 64
title() (*turtle* モジュール), 1809
Tix, 1868
tix_addbitmapdir() (*tkinter.tix.tixCommand* のメソッド), 1874
tix_cget() (*tkinter.tix.tixCommand* のメソッド), 1873
tix_configure() (*tkinter.tix.tixCommand* のメソッド), 1873
tix_filedialog() (*tkinter.tix.tixCommand* のメソッド), 1874
tix_getbitmap() (*tkinter.tix.tixCommand* のメソッド), 1874
tix_getimage() (*tkinter.tix.tixCommand* のメソッド), 1874
tix_option_get() (*tkinter.tix.tixCommand* のメソッド), 1874
tix_resetoptions() (*tkinter.tix.tixCommand* のメソッド), 1874
tixCommand (*tkinter.tix* のクラス), 1873
Tk, 1831
Tk (*tkinter* のクラス), 1832
Tk (*tkinter.tix* のクラス), 1869
Tk Option Data Types, 1841
Tkinter, 1831
tkinter (モジュール), 1831
tkinter.scrolledtext (モジュール), 1875
tkinter.tix (モジュール), 1868
tkinter.ttk (モジュール), 1845
TList (*tkinter.tix* のクラス), 1872
TLS, 1242
TLSv1 (*ssl.TLSVersion* の属性), 1259
TLSv1_1 (*ssl.TLSVersion* の属性), 1259
TLSv1_2 (*ssl.TLSVersion* の属性), 1259
TLSv1_3 (*ssl.TLSVersion* の属性), 1259
TLSVersion (*ssl* のクラス), 1258
TMP, 513
TMPDIR, 513
to_bytes() (*int* のメソッド), 42
to_eng_string() (*decimal.Context* のメソッド), 398
to_eng_string() (*decimal.Decimal* のメソッド), 390
to_integral() (*decimal.Decimal* のメソッド), 390
to_integral_exact() (*decimal.Context* のメソッド), 398
to_integral_exact() (*decimal.Decimal* のメソッド), 391
to_integral_value() (*decimal.Decimal* のメソッド), 391
to_sci_string() (*decimal.Context* のメソッド), 398
ToASCII() (*encodings.idna* モジュール), 223
tobuf() (*tarfile.TarInfo* のメソッド), 637
tobytes() (*array.array* のメソッド), 311
tobytes() (*memoryview* のメソッド), 88
today() (*datetime.date* のクラスメソッド), 233
today() (*datetime.datetime* のクラスメソッド), 238
tofile() (*array.array* のメソッド), 311
tok_name (*token* モジュール), 2338
Token (*contextvars* のクラス), 1093
token (*shlex.shlex* の属性), 1826
token (モジュール), 2338
token_bytes() (*secrets* モジュール), 702
token_hex() (*secrets* モジュール), 703
token_urlsafe() (*secrets* モジュール), 703
TokenError, 2344

tokenize (モジュール), 2342
tokenize command line option
    -e, 2345
    --exact, 2345
    -h, 2345
    --help, 2345
tokenize() (*tokenize* モジュール), 2343
tolist() (*array.array* のメソッド), 311
tolist() (*memoryview* のメソッド), 89
tolist() (*parser.ST* のメソッド), 2325
tomono() (*audioop* モジュール), 1730
toordinal() (*datetime.date* のメソッド), 235
toordinal() (*datetime.datetime* のメソッド), 246
top() (*curses.panel.Panel* のメソッド), 923
top() (*poplib.POP3* のメソッド), 1621
top_panel() (*curses.panel* モジュール), 922
--top-level-directory directory
    unittest-discover command line option, 1953
toprettyxml() (*xml.dom.minidom.Node* のメソッド), 1503
toreadonly() (*memoryview* のメソッド), 89
tostereo() (*audioop* モジュール), 1730
tostring() (*array.array* のメソッド), 312
tostring() (*xml.etree.ElementTree* モジュール), 1473
tostringlist() (*xml.etree.ElementTree* モジュール), 1474
total_changes (*sqlite3.Connection* の属性), 579
total_ordering() (*functools* モジュール), 454
total_seconds() (*datetime.timedelta* のメソッド), 231
totuple() (*parser.ST* のメソッド), 2325
touch() (*pathlib.Path* のメソッド), 489
touchline() (*curses.window* のメソッド), 911
touchwin() (*curses.window* のメソッド), 911
tounicode() (*array.array* のメソッド), 312
ToUnicode() (*encodings.idna* モジュール), 223
towards() (*turtle* モジュール), 1787
toxml() (*xml.dom.minidom.Node* のメソッド), 1503
tparm() (*curses* モジュール), 901
--trace
    trace command line option, 2127
Trace (*trace* のクラス), 2128
Trace (*tracemalloc* のクラス), 2140
trace (モジュール), 2126
trace command line option
    -C, 2127
    -c, 2127
    --count, 2127
    --coverdir=<dir>, 2127
    -f, 2127
    --file=<file>, 2127
    -g, 2127
    --help, 2126
    --ignore-dir=<dir>, 2128
    --ignore-module=<mod>, 2128
    -l, 2127
    --listfuncs, 2127
    -m, 2127
    --missing, 2127
    --no-report, 2127
    -R, 2127
    -r, 2127
    --report, 2127
    -s, 2127
    --summary, 2127
    -T, 2127
    -t, 2127
    --timing, 2127
    --trace, 2127
    --trackcalls, 2127
    --version, 2126
trace function, 979, 2175, 2184
trace() (*inspect* モジュール), 2264
trace_dispatch() (*bdb.Bdb* のメソッド), 2094
traceback
    オブジェクト, 2170, 2235
Traceback (*tracemalloc* のクラス), 2140
traceback (*tracemalloc.Statistic* の属性), 2139
traceback (*tracemalloc.StatisticDiff* の属性), 2139
traceback (*tracemalloc.Trace* の属性), 2140
traceback (モジュール), 2235
traceback_limit (*tracemalloc.Snapshot* の属性), 2138
traceback_limit (*wsgiref.handlers.BaseHandler* の属性), 1560
TracebackException (*traceback* のクラス), 2238
tracebacklimit (*sys* モジュール), 2188
tracebacks
    in CGI scripts, 1549
TracebackType (*types* のクラス), 325
tracemalloc (モジュール), 2129
tracer() (*turtle* モジュール), 1803
traces (*tracemalloc.Snapshot* の属性), 2138
--trackcalls
    trace command line option, 2127
transfercmd() (*ftplib.FTP* のメソッド), 1616
transient_internet() (*test.support* モジュール), 2075
TransientResource (*test.support* のクラス), 2083
translate() (*bytearray* のメソッド), 74
translate() (*bytes* のメソッド), 74
translate() (*fnmatch* モジュール), 518
translate() (*str* のメソッド), 64
translation() (*gettext* モジュール), 1756
Transport (*asyncio* のクラス), 1175
transport (*asyncio.StreamWriter* の属性), 1122
Transport Layer Security, 1242
Tree (*tkinter.tix* のクラス), 1872
TreeBuilder (*xml.etree.ElementTree* のクラス), 1482
Treeview (*tkinter.ttk* のクラス), 1860
triangular() (*random* モジュール), 416
triple-quoted string, **2492**
True, 37, 107
true, 37
True (組み込み変数), 35
truediv() (*operator* モジュール), 464
trunc() (*in module math*), 40
trunc() (*math* モジュール), 368
truncate() (*io.IOBase* のメソッド), 779
truncate() (*os* モジュール), 748
truth
    value, 37
truth() (*operator* モジュール), 462
try
    文, 113
ttk, 1845
tty
    I/O control, 2416
tty (モジュール), 2418
ttyname() (*os* モジュール), 724
tuple
    オブジェクト, 50, 52
Tuple (*typing* モジュール), 1914
tuple (組み込みクラス), 52
tuple2st() (*parser* モジュール), 2323
tuple_params (*2to3 fixer*), 2064
Turtle (*turtle* のクラス), 1809
turtle (モジュール), 1775
turtledemo (モジュール), 1814
turtles() (*turtle* モジュール), 1808
TurtleScreen (*turtle* のクラス), 1809
turtlesize() (*turtle* モジュール), 1795
type, **2492**
    Boolean, 6
    operations on dictionary, 97
    operations on list, 50
    オブジェクト, 30
    組み込み関数, 106
type (*optparse.Option* の属性), 2449
type (*socket.socket* の属性), 1237
type (*tarfile.TarInfo* の属性), 638

Type (*typing* のクラス), 1903
type (*urllib.request.Request* の属性), 1570
type (組み込みクラス), 30
type alias, **2492**
type hint, **2492**
type_check_only() (*typing* モジュール), 1912
TYPE_CHECKER (*optparse.Option* の属性), 2463
TYPE_CHECKING (*typing* モジュール), 1916
TYPE_COMMENT (*token* モジュール), 2341
TYPE_IGNORE (*token* モジュール), 2341
typeahead() (*curses* モジュール), 902
typecode (*array.array* の属性), 310
typecodes (*array* モジュール), 310
TYPED_ACTIONS (*optparse.Option* の属性), 2465
typed_subpart_iterator() (*email.iterators* モジュール), 1401
TypedDict (*typing* のクラス), 1910
TypeError, 119
types
    built-in, 37
    immutable sequence, 50
    mutable sequence, 50
    operations on integer, 41
    operations on mapping, 97
    operations on numeric, 40
    operations on sequence, 47, 50
    モジュール, 106
types (*2to3 fixer*), 2064
TYPES (*optparse.Option* の属性), 2463
types (モジュール), 321
types_map (*mimetypes* モジュール), 1440
types_map (*mimetypes.MimeTypes* の属性), 1441
types_map_inv (*mimetypes.MimeTypes* の属性), 1441
TypeVar (*typing* のクラス), 1901
typing (モジュール), 1893
TZ, 799, 800
tzinfo (*datetime* のクラス), 256
tzinfo (*datetime.datetime* の属性), 242
tzinfo (*datetime.time* の属性), 252
tzname (*time* モジュール), 802
tzname() (*datetime.datetime* のメソッド), 245
tzname() (*datetime.time* のメソッド), 255
tzname() (*datetime.timezone* のメソッド), 264
tzname() (*datetime.tzinfo* のメソッド), 258
tzset() (*time* モジュール), 799

U

-u
    timeit command line option, 2123
ucd_3_2_0 (*unicodedata* モジュール), 182
udata (*select.kevent* の属性), 1297
UDPServer (*socketserver* のクラス), 1661
UF_APPEND (*stat* モジュール), 506
UF_COMPRESSED (*stat* モジュール), 506
UF_HIDDEN (*stat* モジュール), 506
UF_IMMUTABLE (*stat* モジュール), 506
UF_NODUMP (*stat* モジュール), 506
UF_NOUNLINK (*stat* モジュール), 506
UF_OPAQUE (*stat* モジュール), 506
UID (*plistlib* のクラス), 685
uid (*tarfile.TarInfo* の属性), 638
uid() (*imaplib.IMAP4* のメソッド), 1630
uidl() (*poplib.POP3* のメソッド), 1622
u-LAW, 1727, 1733, 1745
ulaw2lin() (*audioop* モジュール), 1730
umask() (*os* モジュール), 713
unalias (*pdb command*), 2109
uname (*tarfile.TarInfo* の属性), 638
uname() (*os* モジュール), 713
uname() (*platform* モジュール), 925
UNARY_INVERT (*opcode*), 2362
UNARY_NEGATIVE (*opcode*), 2362
UNARY_NOT (*opcode*), 2362
UNARY_POSITIVE (*opcode*), 2362
UnboundLocalError, 119
unbuffered I/O, 23
UNC paths
    and os.makedirs(), 734
UNCHECKED_HASH (*py_compile.PycInvalidationMode* の属性), 2352
unconsumed_tail (*zlib.Decompress* の属性), 598
unctrl() (*curses* モジュール), 902
unctrl() (*curses.ascii* モジュール), 921
Underflow (*decimal* のクラス), 401
undisplay (*pdb command*), 2108
undo() (*turtle* モジュール), 1786
undobufferentries() (*turtle* モジュール), 1800
undoc_header (*cmd.Cmd* の属性), 1819
unescape() (*html* モジュール), 1453
unescape() (*xml.sax.saxutils* モジュール), 1517
UnexpectedException, 1946
unexpectedSuccesses (*unittest.TestResult* の属性), 1978
unfreeze() (*gc* モジュール), 2247
unget_wch() (*curses* モジュール), 902
ungetch() (*curses* モジュール), 902
ungetch() (*msvcrt* モジュール), 2394
ungetmouse() (*curses* モジュール), 902
ungetwch() (*msvcrt* モジュール), 2394
unhexlify() (*binascii* モジュール), 1449
Unicode, 181, 200
    database, 181
unicode (*2to3 fixer*), 2064
unicodedata (モジュール), 181
UnicodeDecodeError, 120
UnicodeEncodeError, 120
UnicodeError, 119
UnicodeTranslateError, 120
UnicodeWarning, 123
unidata_version (*unicodedata* モジュール), 182
unified_diff() (*difflib* モジュール), 167
uniform() (*random* モジュール), 416
UnimplementedFileMode, 1606
Union (*ctypes* のクラス), 972
Union (*typing* モジュール), 1913
union() (*frozenset* のメソッド), 95
unique() (*enum* モジュール), 342
--unit=U
    timeit command line option, 2123
unittest (モジュール), 1948
unittest command line option
    -b, 1951
    --buffer, 1951
    -c, 1951
    --catch, 1951
    -f, 1952
    --failfast, 1952
    -k, 1952
    --locals, 1952
unittest-discover command line option
    -p, 1953
    --pattern pattern, 1953
    -s, 1953
    --start-directory directory, 1953
    -t, 1953
    --top-level-directory directory, 1953
    -v, 1952
    --verbose, 1952
unittest.mock (モジュール), 1987
universal newlines, **2492**
    bytearray.splitlines method, 81
    bytes.splitlines method, 81
    csv.reader function, 650
    importlib.abc.InspectLoader.get_source method, 2298
    io.IncrementalNewlineDecoder class, 789

    io.TextIOWrapper class, 787
    open() built-in function, 22
    str.splitlines method, 62
    subprocess module, 1066
UNIX
    file control, 2420
    I/O control, 2420
unix_dialect (*csv* のクラス), 652
unix_shell (*test.support* モジュール), 2069
UnixDatagramServer (*socketserver* のクラス), 1661
UnixStreamServer (*socketserver* のクラス), 1661
unknown (*uuid.SafeUUID* の属性), 1656
unknown_decl() (*html.parser.HTMLParser* のメソッド), 1457
unknown_open() (*urllib.request.BaseHandler* のメソッド), 1574
unknown_open() (*urllib.request.UnknownHandler* のメソッド), 1580
UnknownHandler (*urllib.request* のクラス), 1569
UnknownProtocol, 1606
UnknownTransferEncoding, 1606
unlink() (*multiprocessing.shared_memory.SharedMemory* のメソッド), 1049
unlink() (*os* モジュール), 748
unlink() (*pathlib.Path* のメソッド), 489
unlink() (*test.support* モジュール), 2070
unlink() (*xml.dom.minidom.Node* のメソッド), 1502
unload() (*test.support* モジュール), 2070
unlock() (*mailbox.Babyl* のメソッド), 1426
unlock() (*mailbox.Mailbox* のメソッド), 1420
unlock() (*mailbox.Maildir* のメソッド), 1422
unlock() (*mailbox.mbox* のメソッド), 1423
unlock() (*mailbox.MH* のメソッド), 1425
unlock() (*mailbox.MMDF* のメソッド), 1426
unpack() (*struct* モジュール), 194
unpack() (*struct.Struct* のメソッド), 199
unpack_archive() (*shutil* モジュール), 530
unpack_array() (*xdrlib.Unpacker* のメソッド), 682
unpack_bytes() (*xdrlib.Unpacker* のメソッド), 682
unpack_double() (*xdrlib.Unpacker* のメソッド), 682
UNPACK_EX (*opcode*), 2368
unpack_farray() (*xdrlib.Unpacker* のメソッド), 682
unpack_float() (*xdrlib.Unpacker* のメソッド), 681
unpack_fopaque() (*xdrlib.Unpacker* のメソッド), 682
unpack_from() (*struct* モジュール), 194
unpack_from() (*struct.Struct* のメソッド), 199
unpack_fstring() (*xdrlib.Unpacker* のメソッド), 682
unpack_list() (*xdrlib.Unpacker* のメソッド), 682
unpack_opaque() (*xdrlib.Unpacker* のメソッド), 682
UNPACK_SEQUENCE (*opcode*), 2368
unpack_string() (*xdrlib.Unpacker* のメソッド), 682
Unpacker (*xdrlib* のクラス), 679
unparsedEntityDecl() (*xml.sax.handler.DTDHandler* のメソッド), 1516
UnparsedEntityDeclHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1529
Unpickler (*pickle* のクラス), 541
UnpicklingError, 539
unquote() (*email.utils* モジュール), 1398
unquote() (*urllib.parse* モジュール), 1597
unquote_plus() (*urllib.parse* モジュール), 1597
unquote_to_bytes() (*urllib.parse* モジュール), 1597
unraisablehook() (*sys* モジュール), 2188
unregister() (*atexit* モジュール), 2234
unregister() (*faulthandler* モジュール), 2100
unregister() (*select.devpoll* のメソッド), 1291
unregister() (*select.epoll* のメソッド), 1293
unregister() (*selectors.BaseSelector* のメソッド), 1299
unregister() (*select.poll* のメソッド), 1294
unregister_archive_format() (*shutil* モジュール), 529
unregister_dialect() (*csv* モジュール), 651
unregister_unpack_format() (*shutil* モジュール), 530
unsafe (*uuid.SafeUUID* の属性), 1656
unset() (*test.support.EnvironmentVarGuard* のメソッド), 2083
unsetenv() (*os* モジュール), 714
UnstructuredHeader (*email.headerregistry* のクラス), 1360
unsubscribe() (*imaplib.IMAP4* のメソッド), 1630
UnsupportedOperation, 775
until (*pdb command*), 2107
untokenize() (*tokenize* モジュール), 2343
untouchwin() (*curses.window* のメソッド), 911
unused_data (*bz2.BZ2Decompressor* の属性), 607
unused_data (*lzma.LZMADecompressor* の属性), 613
unused_data (*zlib.Decompress* の属性), 598
unverifiable (*urllib.request.Request* の属性), 1570
unwrap() (*inspect* モジュール), 2263
unwrap() (*ssl.SSLSocket* のメソッド), 1263
unwrap() (*urllib.parse* モジュール), 1594
up (*pdb command*), 2105
up() (*turtle* モジュール), 1789
update() (*collections.Counter* のメソッド), 280
update() (*dict* のメソッド), 100
update() (*frozenset* のメソッド), 96
update() (*hashlib.hash* のメソッド), 689
update() (*hmac.HMAC* のメソッド), 700
update() (*http.cookies.Morsel* のメソッド), 1681
update() (*mailbox.Mailbox* のメソッド), 1420
update() (*mailbox.Maildir* のメソッド), 1422
update() (*trace.CoverageResults* のメソッド), 2129
update() (*turtle* モジュール), 1803
update_authenticated() (*urllib.request.HTTPPasswordMgrWithPriorAuth* のメソッド), 1577
update_lines_cols() (*curses* モジュール), 902
update_panels() (*curses.panel* モジュール), 922
update_visible() (*mailbox.BabylMessage* のメソッド), 1434
update_wrapper() (*functools* モジュール), 460
upper() (*bytearray* のメソッド), 82
upper() (*bytes* のメソッド), 82
upper() (*str* のメソッド), 65
urandom() (*os* モジュール), 772
URL, 1540, 1588, 1600, 1671
    parsing, 1588
    relative, 1588
url (*xmlrpc.client.ProtocolError* の属性), 1701
url2pathname() (*urllib.request* モジュール), 1565
urlcleanup() (*urllib.request* モジュール), 1585
urldefrag() (*urllib.parse* モジュール), 1593
urlencode() (*urllib.parse* モジュール), 1597
URLError, 1599
urljoin() (*urllib.parse* モジュール), 1593
urllib (*2to3 fixer*), 2064
urllib (モジュール), 1563
urllib.error (モジュール), 1599
urllib.parse (モジュール), 1588
urllib.request
    モジュール, 1604
urllib.request (モジュール), 1563
urllib.response (モジュール), 1588
urllib.robotparser (モジュール), 1600
urlopen() (*urllib.request* モジュール), 1563
URLopener (*urllib.request* のクラス), 1585
urlparse() (*urllib.parse* モジュール), 1588
urlretrieve() (*urllib.request* モジュール), 1584
urlsafe_b64decode() (*base64* モジュール), 1443
urlsafe_b64encode() (*base64* モジュール), 1443
urlsplit() (*urllib.parse* モジュール), 1592
urlunparse() (*urllib.parse* モジュール), 1591
urlunsplit() (*urllib.parse* モジュール), 1593
urn (*uuid.UUID* の属性), 1658
use_default_colors() (*curses* モジュール), 903
use_env() (*curses* モジュール), 902
use_rawinput (*cmd.Cmd* の属性), 1819

UseForeignDTD() (*xml.parsers.expat.xmlparser* のメソッド), 1525
USER, 894
user
    effective id, 709
    id, 711
    id, setting, 713
user() (*poplib.POP3* のメソッド), 1621
USER_BASE (*site* モジュール), 2271
user_call() (*bdb.Bdb* のメソッド), 2095
user_exception() (*bdb.Bdb* のメソッド), 2095
user_line() (*bdb.Bdb* のメソッド), 2095
user_return() (*bdb.Bdb* のメソッド), 2095
USER_SITE (*site* モジュール), 2271
--user-base
    site command line option, 2272
usercustomize
    モジュール, 2270
UserDict (*collections* のクラス), 294
UserList (*collections* のクラス), 294
USERNAME, 710, 894
username (*email.headerregistry.Address* の属性), 1364
USERPROFILE, 493
userptr() (*curses.panel.Panel* のメソッド), 923
--user-site
    site command line option, 2272
UserString (*collections* のクラス), 295
UserWarning, 122
USTAR_FORMAT (*tarfile* モジュール), 631
UTC, 791
utc (*datetime.timezone* の属性), 265
utcfromtimestamp() (*datetime.datetime* のクラスメソッド), 240
utcnow() (*datetime.datetime* のクラスメソッド), 239
utcoffset() (*datetime.datetime* のメソッド), 245
utcoffset() (*datetime.time* のメソッド), 255
utcoffset() (*datetime.timezone* のメソッド), 264
utcoffset() (*datetime.tzinfo* のメソッド), 256
utctimetuple() (*datetime.datetime* のメソッド), 246
utf8 (*email.policy.EmailPolicy* の属性), 1353
utf8() (*poplib.POP3* のメソッド), 1622
utf8_enabled (*imaplib.IMAP4* の属性), 1630
utime() (*os* モジュール), 748
uu
    モジュール, 1447
uu (モジュール), 1451
UUID (*uuid* のクラス), 1657
uuid (モジュール), 1656
uuid1, 1659
uuid1() (*uuid* モジュール), 1658
uuid3, 1659
uuid3() (*uuid* モジュール), 1659
uuid4, 1659
uuid4() (*uuid* モジュール), 1659
uuid5, 1659
uuid5() (*uuid* モジュール), 1659
UuidCreate() (*msilib* モジュール), 2385

V

-v
    tarfile command line option, 645
    timeit command line option, 2124
    unittest-discover command line option, 1952
v4_int_to_packed() (*ipaddress* モジュール), 1725
v6_int_to_packed() (*ipaddress* モジュール), 1725
valid_signals() (*signal* モジュール), 1314
validator() (*wsgiref.validate* モジュール), 1556
value
    truth, 37
value (*ctypes._SimpleCData* の属性), 969
value (*http.cookiejar.Cookie* の属性), 1692
value (*http.cookies.Morsel* の属性), 1681
value (*xml.dom.Attr* の属性), 1496
Value() (*multiprocessing* モジュール), 1017
Value() (*multiprocessing.managers.SyncManager* のメソッド), 1023
Value() (*multiprocessing.sharedctypes* モジュール), 1018
value_decode() (*http.cookies.BaseCookie* のメソッド), 1679
value_encode() (*http.cookies.BaseCookie* のメソッド), 1679
ValueError, 120
valuerefs() (*weakref.WeakValueDictionary* のメソッド), 315
values
    Boolean, 107
values() (*contextvars.Context* のメソッド), 1095
values() (*dict* のメソッド), 100
values() (*email.message.EmailMessage* のメソッド), 1332
values() (*email.message.Message* のメソッド), 1381
values() (*mailbox.Mailbox* のメソッド), 1418
values() (*types.MappingProxyType* のメソッド), 326
ValuesView (*collections.abc* のクラス), 299
ValuesView (*typing* のクラス), 1905
var (*contextvars.Token* の属性), 1093
variable annotation, **2492**
variance (*statistics.NormalDist* の属性), 430
variance() (*statistics* モジュール), 427
variant (*uuid.UUID* の属性), 1658
vars() (組み込み関数), 30
vbar (*tkinter.scrolledtext.ScrolledText* の属性), 1875
VBAR (*token* モジュール), 2339
VBAREQUAL (*token* モジュール), 2340
Vec2D (*turtle* のクラス), 1810
venv (モジュール), 2146
--verbose
    tarfile command line option, 645
    timeit command line option, 2124
    unittest-discover command line option, 1952
VERBOSE (*re* モジュール), 148
verbose (*tabnanny* モジュール), 2348
verbose (*test.support* モジュール), 2068
verify() (*smtplib.SMTP* のメソッド), 1643
verify_client_post_handshake() (*ssl.SSLSocket* のメソッド), 1263
verify_code (*ssl.SSLCertVerificationError* の属性), 1246
VERIFY_CRL_CHECK_CHAIN (*ssl* モジュール), 1252
VERIFY_CRL_CHECK_LEAF (*ssl* モジュール), 1252
VERIFY_DEFAULT (*ssl* モジュール), 1251
verify_flags (*ssl.SSLContext* の属性), 1275
verify_message (*ssl.SSLCertVerificationError* の属性), 1246
verify_mode (*ssl.SSLContext* の属性), 1275
verify_request() (*socketserver.BaseServer* のメソッド), 1665
VERIFY_X509_STRICT (*ssl* モジュール), 1252
VERIFY_X509_TRUSTED_FIRST (*ssl* モジュール), 1252
VerifyFlags (*ssl* のクラス), 1252
VerifyMode (*ssl* のクラス), 1251
--version
    trace command line option, 2126
version (*curses* モジュール), 911
version (*email.headerregistry.MIMEVersionHeader* の属性), 1362
version (*http.client.HTTPResponse* の属性), 1610
version (*http.cookiejar.Cookie* の属性), 1692
version (*ipaddress.IPv4Address* の属性), 1713
version (*ipaddress.IPv4Network* の属性), 1718
version (*ipaddress.IPv6Address* の属性), 1715
version (*ipaddress.IPv6Network* の属性), 1721
version (*marshal* モジュール), 562
version (*sqlite3* モジュール), 570
version (*sys* モジュール), 2188
version (*urllib.request.URLopener* の属性), 1586
version (*uuid.UUID* の属性), 1658

version() (*ensurepip* モジュール), 2145
version() (*platform* モジュール), 925
version() (*ssl.SSLSocket* のメソッド), 1263
version_info (*sqlite3* モジュール), 570
version_info (*sys* モジュール), 2188
version_string() (*http.server.BaseHTTPRequestHandler* のメソッド), 1675
vformat() (*string.Formatter* のメソッド), 126
virtual
    Environments, 2146
virtual environment, **2493**
virtual machine, **2493**
VIRTUAL_ENV, 2148
visit() (*ast.NodeVisitor* のメソッド), 2332
vline() (*curses.window* のメソッド), 911
voidcmd() (*ftplib.FTP* のメソッド), 1616
volume (*zipfile.ZipInfo* の属性), 626
vonmisesvariate() (*random* モジュール), 417

W

W_OK (*os* モジュール), 728
wait() (*asyncio* モジュール), 1111
wait() (*asyncio.asyncio.subprocess.Process* のメソッド), 1134
wait() (*asyncio.Condition* のメソッド), 1130
wait() (*asyncio.Event* のメソッド), 1128
wait() (*concurrent.futures* モジュール), 1060
wait() (*multiprocessing.connection* モジュール), 1033
wait() (*multiprocessing.pool.AsyncResult* のメソッド), 1031
wait() (*os* モジュール), 764
wait() (*subprocess.Popen* のメソッド), 1072
wait() (*threading.Barrier* のメソッド), 992
wait() (*threading.Condition* のメソッド), 987
wait() (*threading.Event* のメソッド), 991
wait3() (*os* モジュール), 766
wait4() (*os* モジュール), 766
wait_closed() (*asyncio.Server* のメソッド), 1166
wait_closed() (*asyncio.StreamWriter* のメソッド), 1123
wait_for() (*asyncio* モジュール), 1110
wait_for() (*asyncio.Condition* のメソッド), 1130
wait_for() (*threading.Condition* のメソッド), 988
wait_threads_exit() (*test.support* モジュール), 2076
waitid() (*os* モジュール), 765
waitpid() (*os* モジュール), 765
walk() (*ast* モジュール), 2332
walk() (*email.message.EmailMessage* のメソッド), 1335
walk() (*email.message.Message* のメソッド), 1385
walk() (*os* モジュール), 749
walk_packages() (*pkgutil* モジュール), 2284
walk_stack() (*traceback* モジュール), 2237
walk_tb() (*traceback* モジュール), 2237
want (*doctest.Example* の属性), 1939
warn() (*warnings* モジュール), 2201
warn_explicit() (*warnings* モジュール), 2201
Warning, 122, 585
warning() (*logging* モジュール), 858
warning() (*logging.Logger* のメソッド), 846
warning() (*xml.sax.handler.ErrorHandler* のメソッド), 1516
warnings, 2195
warnings (モジュール), 2195
WarningsRecorder (*test.support* のクラス), 2084
warnoptions (*sys* モジュール), 2189
wasSuccessful() (*unittest.TestResult* のメソッド), 1979
WatchedFileHandler (*logging.handlers* のクラス), 879
wave (モジュール), 1738
WCONTINUED (*os* モジュール), 766
WCOREDUMP() (*os* モジュール), 767
文
    assert, 115
    del, 50, 97
    except, 113
    if, 37
    import, 32, 2269, 2467
    raise, 113
    try, 113
    while, 37
WeakKeyDictionary (*weakref* のクラス), 314
WeakMethod (*weakref* のクラス), 315
weakref (モジュール), 312
WeakSet (*weakref* のクラス), 315
WeakValueDictionary (*weakref* のクラス), 315
webbrowser (モジュール), 1537
weekday() (*calendar* モジュール), 273
weekday() (*datetime.date* のメソッド), 235
weekday() (*datetime.datetime* のメソッド), 247
weekheader() (*calendar* モジュール), 273
weibullvariate() (*random* モジュール), 417
WEXITED (*os* モジュール), 765
WEXITSTATUS() (*os* モジュール), 767
wfile (*http.server.BaseHTTPRequestHandler* の属性), 1672
what() (*imghdr* モジュール), 1744
what() (*sndhdr* モジュール), 1745
whathdr() (*sndhdr* モジュール), 1745
whatis (*pdb command*), 2108
when() (*asyncio.TimerHandle* のメソッド), 1164
where (*pdb command*), 2105
which() (*shutil* モジュール), 525
whichdb() (*dbm* モジュール), 562
while
    文, 37
whitespace (*shlex.shlex* の属性), 1825
whitespace (*string* モジュール), 126
whitespace_split (*shlex.shlex* の属性), 1826
Widget (*tkinter.ttk* のクラス), 1849
width (*textwrap.TextWrapper* の属性), 178
width() (*turtle* モジュール), 1789
WIFCONTINUED() (*os* モジュール), 767
WIFEXITED() (*os* モジュール), 767
WIFSIGNALED() (*os* モジュール), 767
WIFSTOPPED() (*os* モジュール), 767
win32_edition() (*platform* モジュール), 926
win32_is_iot() (*platform* モジュール), 926
win32_ver() (*platform* モジュール), 926
WinDLL (*ctypes* のクラス), 958
window manager (*widgets*), 1841
window() (*curses.panel.Panel* のメソッド), 923
window_height() (*turtle* モジュール), 1808
window_width() (*turtle* モジュール), 1808
Windows ini file, 657
WindowsError, 120
WindowsPath (*pathlib* のクラス), 482
WindowsProactorEventLoopPolicy (*asyncio* のクラス), 1192
WindowsRegistryFinder (*importlib.machinery* のクラス), 2304
WindowsSelectorEventLoopPolicy (*asyncio* のクラス), 1192
winerror (*OSError* の属性), 116
WinError() (*ctypes* モジュール), 968
WINFUNCTYPE() (*ctypes* モジュール), 962
winreg (モジュール), 2395
WinSock, 1290
winsound (モジュール), 2406
winver (*sys* モジュール), 2189
WITH_CLEANUP_FINISH (*opcode*), 2368
WITH_CLEANUP_START (*opcode*), 2368
with_hostmask (*ipaddress.IPv4Interface* の属性), 1724
with_hostmask (*ipaddress.IPv4Network* の属性), 1719
with_hostmask (*ipaddress.IPv6Interface* の属性), 1725
with_hostmask (*ipaddress.IPv6Network* の属性), 1722
with_name() (*pathlib.PurePath* のメソッド), 481
with_netmask (*ipaddress.IPv4Interface* の属性), 1724

with_netmask (*ipaddress.IPv4Network* の属性), 1718
with_netmask (*ipaddress.IPv6Interface* の属性), 1725
with_netmask (*ipaddress.IPv6Network* の属性), 1722
with_prefixlen (*ipaddress.IPv4Interface* の属性), 1724
with_prefixlen (*ipaddress.IPv4Network* の属性), 1718
with_prefixlen (*ipaddress.IPv6Interface* の属性), 1725
with_prefixlen (*ipaddress.IPv6Network* の属性), 1722
with_pymalloc() (*test.support* モジュール), 2071
with_suffix() (*pathlib.PurePath* のメソッド), 481
with_traceback() (*BaseException* のメソッド), 114
演算子
    % (*percent*), 39
    & (*ampersand*), 41
    * (*asterisk*), 39
    **, 39
    / (*slash*), 39
    //, 39
    < (*less*), 38
    <<, 41
    <=, 38
    !=, 38
    ==, 38
    > (*greater*), 38
    >=, 38
    >>, 41
    ^ (*caret*), 41
    | (*vertical bar*), 41
    ~ (*tilde*), 41
    and, 37, 38
    in, 39, 47
    is, 38
    is not, 38
    not, 38
    not in, 39, 47
    or, 37, 38
WNOHANG (*os* モジュール), 766
WNOWAIT (*os* モジュール), 765
wordchars (*shlex.shlex* の属性), 1825
World Wide Web, 1537, 1588, 1600
wrap() (*textwrap* モジュール), 176
wrap() (*textwrap.TextWrapper* のメソッド), 180
wrap_bio() (*ssl.SSLContext* のメソッド), 1271
wrap_future() (*asyncio* モジュール), 1170
wrap_socket() (*ssl* モジュール), 1250
wrap_socket() (*ssl.SSLContext* のメソッド), 1270
wrapper() (*curses* モジュール), 903
WrapperDescriptorType (*types* モジュール), 324
wraps() (*functools* モジュール), 460
WRITABLE (*tkinter* モジュール), 1845
writable() (*asyncore.dispatcher* のメソッド), 1304
writable() (*io.IOBase* のメソッド), 779
write() (*asyncio.StreamWriter* のメソッド), 1121
write() (*asyncio.WriteTransport* のメソッド), 1178
write() (*codecs.StreamWriter* のメソッド), 210
write() (*code.InteractiveInterpreter* のメソッド), 2275
write() (*configparser.ConfigParser* のメソッド), 675
write() (*email.generator.BytesGenerator* のメソッド), 1345
write() (*email.generator.Generator* のメソッド), 1347
write() (*io.BufferedIOBase* のメソッド), 782
write() (*io.BufferedWriter* のメソッド), 785
write() (*io.RawIOBase* のメソッド), 780
write() (*io.TextIOBase* のメソッド), 786
write() (*mmap.mmap* のメソッド), 1324
write() (*os* モジュール), 725
write() (*ossaudiodev.oss_audio_device* のメソッド), 1747
write() (*ssl.MemoryBIO* のメソッド), 1284
write() (*ssl.SSLSocket* のメソッド), 1260
write() (*telnetlib.Telnet* のメソッド), 1655
write() (*turtle* モジュール), 1793
write() (*xml.etree.ElementTree.ElementTree* のメソッド), 1480
write() (*zipfile.ZipFile* のメソッド), 621
write_byte() (*mmap.mmap* のメソッド), 1324
write_bytes() (*pathlib.Path* のメソッド), 490
write_docstringdict() (*turtle* モジュール), 1812
write_eof() (*asyncio.StreamWriter* のメソッド), 1122
write_eof() (*asyncio.WriteTransport* のメソッド), 1179
write_eof() (*ssl.MemoryBIO* のメソッド), 1284
write_history_file() (*readline* モジュール), 186
write_results() (*trace.CoverageResults* のメソッド), 2129
write_text() (*pathlib.Path* のメソッド), 490
write_through (*io.TextIOWrapper* の属性), 788
writeall() (*ossaudiodev.oss_audio_device* のメソッド), 1747
writeframes() (*aifc.aifc* のメソッド), 1734
writeframes() (*sunau.AU_write* のメソッド), 1737
writeframes() (*wave.Wave_write* のメソッド), 1741
writeframesraw() (*aifc.aifc* のメソッド), 1734
writeframesraw() (*sunau.AU_write* のメソッド), 1737
writeframesraw() (*wave.Wave_write* のメソッド), 1741
writeheader() (*csv.DictWriter* のメソッド), 656
writelines() (*asyncio.StreamWriter* のメソッド), 1122
writelines() (*asyncio.WriteTransport* のメソッド), 1178
writelines() (*codecs.StreamWriter* のメソッド), 210
writelines() (*io.IOBase* のメソッド), 779
writePlist() (*plistlib* モジュール), 685
writePlistToBytes() (*plistlib* モジュール), 685
writepy() (*zipfile.PyZipFile* のメソッド), 623
writer (*formatter.formatter* の属性), 2380
writer() (*csv* モジュール), 650
writerow() (*csv.csvwriter* のメソッド), 655
writerows() (*csv.csvwriter* のメソッド), 655
writestr() (*zipfile.ZipFile* のメソッド), 622
WriteTransport (*asyncio* のクラス), 1175
writev() (*os* モジュール), 725
writexml() (*xml.dom.minidom.Node* のメソッド), 1502
WrongDocumentErr, 1498
ws_comma (*2to3 fixer*), 2065
wsgi_file_wrapper (*wsgiref.handlers.BaseHandler* の属性), 1560
wsgi_multiprocess (*wsgiref.handlers.BaseHandler* の属性), 1559
wsgi_multithread (*wsgiref.handlers.BaseHandler* の属性), 1559
wsgi_run_once (*wsgiref.handlers.BaseHandler* の属性), 1559
wsgiref (モジュール), 1550
wsgiref.handlers (モジュール), 1557
wsgiref.headers (モジュール), 1553
wsgiref.simple_server (モジュール), 1554
wsgiref.util (モジュール), 1550
wsgiref.validate (モジュール), 1556
WSGIRequestHandler (*wsgiref.simple_server* のクラス), 1555
WSGIServer (*wsgiref.simple_server* のクラス), 1554
wShowWindow (*subprocess.STARTUPINFO* の属性), 1075
WSTOPPED (*os* モジュール), 765
WSTOPSIG() (*os* モジュール), 767
wstring_at() (*ctypes* モジュール), 968
WTERMSIG() (*os* モジュール), 767
WUNTRACED (*os* モジュール), 766
WWW, 1537, 1588, 1600
    server, 1540, 1671

X

X (*re* モジュール), 148
-x regex
    compileall command line option, 2353
X509 certificate, 1275
X_OK (*os* モジュール), 728
xatom() (*imaplib.IMAP4* のメソッド), 1630
XATTR_CREATE (*os* モジュール), 754
XATTR_REPLACE (*os* モジュール), 754
XATTR_SIZE_MAX (*os* モジュール), 754

xcor() (*turtle* モジュール), 1787
XDR, 679
xdrlib (モジュール), 679
xhdr() (*nntplib.NNTP* のメソッド), 1638
XHTML, 1454
XHTML_NAMESPACE (*xml.dom* モジュール), 1488
xml (モジュール), 1460
XML() (*xml.etree.ElementTree* モジュール), 1474
XML_ERROR_ABORTED (*xml.parsers.expat.errors* モジュール), 1535
XML_ERROR_ASYNC_ENTITY (*xml.parsers.expat.errors* モジュール), 1532
XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF (*xml.parsers.expat.errors* モジュール), 1532
XML_ERROR_BAD_CHAR_REF (*xml.parsers.expat.errors* モジュール), 1533
XML_ERROR_BINARY_ENTITY_REF (*xml.parsers.expat.errors* モジュール), 1533
XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING (*xml.parsers.expat.errors* モジュール), 1534
XML_ERROR_DUPLICATE_ATTRIBUTE (*xml.parsers.expat.errors* モジュール), 1533
XML_ERROR_ENTITY_DECLARED_IN_PE (*xml.parsers.expat.errors* モジュール), 1534
XML_ERROR_EXTERNAL_ENTITY_HANDLING (*xml.parsers.expat.errors* モジュール), 1534
XML_ERROR_FEATURE_REQUIRES_XML_DTD (*xml.parsers.expat.errors* モジュール), 1534
XML_ERROR_FINISHED (*xml.parsers.expat.errors* モジュール), 1535
XML_ERROR_INCOMPLETE_PE (*xml.parsers.expat.errors* モジュール), 1534
XML_ERROR_INCORRECT_ENCODING (*xml.parsers.expat.errors* モジュール), 1533
XML_ERROR_INVALID_TOKEN (*xml.parsers.expat.errors* モジュール), 1533
XML_ERROR_JUNK_AFTER_DOC_ELEMENT (*xml.parsers.expat.errors* モジュール), 1533
XML_ERROR_MISPLACED_XML_PI (*xml.parsers.expat.errors* モジュール), 1533
XML_ERROR_NO_ELEMENTS (*xml.parsers.expat.errors* モジュール), 1533
XML_ERROR_NO_MEMORY (*xml.parsers.expat.errors* モジュール), 1533
XML_ERROR_NOT_STANDALONE (*xml.parsers.expat.errors* モジュール), 1534
XML_ERROR_NOT_SUSPENDED (*xml.parsers.expat.errors* モジュール), 1534
XML_ERROR_PARAM_ENTITY_REF (*xml.parsers.expat.errors* モジュール), 1533
XML_ERROR_PARTIAL_CHAR (*xml.parsers.expat.errors* モジュール), 1533
XML_ERROR_PUBLICID (*xml.parsers.expat.errors* モジュール), 1534
XML_ERROR_RECURSIVE_ENTITY_REF (*xml.parsers.expat.errors* モジュール), 1533
XML_ERROR_SUSPEND_PE (*xml.parsers.expat.errors* モジュール), 1535
XML_ERROR_SUSPENDED (*xml.parsers.expat.errors* モジュール), 1534
XML_ERROR_SYNTAX (*xml.parsers.expat.errors* モジュール), 1533
XML_ERROR_TAG_MISMATCH (*xml.parsers.expat.errors* モジュール), 1533
XML_ERROR_TEXT_DECL (*xml.parsers.expat.errors* モジュール), 1534
XML_ERROR_UNBOUND_PREFIX (*xml.parsers.expat.errors* モジュール), 1534
XML_ERROR_UNCLOSED_CDATA_SECTION (*xml.parsers.expat.errors* モジュール), 1534
XML_ERROR_UNCLOSED_TOKEN (*xml.parsers.expat.errors* モジュール), 1533
XML_ERROR_UNDECLARING_PREFIX (*xml.parsers.expat.errors* モジュール), 1534
XML_ERROR_UNDEFINED_ENTITY (*xml.parsers.expat.errors* モジュール), 1533
XML_ERROR_UNEXPECTED_STATE (*xml.parsers.expat.errors* モジュール), 1534
XML_ERROR_UNKNOWN_ENCODING (*xml.parsers.expat.errors* モジュール), 1534
XML_ERROR_XML_DECL (*xml.parsers.expat.errors* モジュール), 1534
XML_NAMESPACE (*xml.dom* モジュール), 1488
xmlcharrefreplace_errors() (*codecs* モジュール), 206
XmlDeclHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1528
xml.dom (モジュール), 1486
xml.dom.minidom (モジュール), 1500
xml.dom.pulldom (モジュール), 1506
xml.etree.ElementInclude.default_loader() (*xml.etree.ElementTree* モジュール), 1476
xml.etree.ElementInclude.include() (*xml.etree.ElementTree* モジュール), 1476
xml.etree.ElementTree (モジュール), 1462
XMLFilterBase (*xml.sax.saxutils* のクラス), 1517
XMLGenerator (*xml.sax.saxutils* のクラス), 1517
XMLID() (*xml.etree.ElementTree* モジュール), 1474
XMLNS_NAMESPACE (*xml.dom* モジュール), 1488
XMLParser (*xml.etree.ElementTree* のクラス), 1483
xml.parsers.expat (モジュール), 1523
xml.parsers.expat.errors (モジュール), 1532
xml.parsers.expat.model (モジュール), 1531
XMLParserType (*xml.parsers.expat* モジュール), 1523
XMLPullParser (*xml.etree.ElementTree* のクラス), 1485
XMLReader (*xml.sax.xmlreader* のクラス), 1518
xmlrpc.client (モジュール), 1694
xmlrpc.server (モジュール), 1704
xml.sax (モジュール), 1508
xml.sax.handler (モジュール), 1510
xml.sax.saxutils (モジュール), 1517
xml.sax.xmlreader (モジュール), 1518
xor() (*operator* モジュール), 464
xover() (*nntplib.NNTP* のメソッド), 1639
xpath() (*nntplib.NNTP* のメソッド), 1639
xrange (*2to3 fixer*), 2065
xreadlines (*2to3 fixer*), 2065
xview() (*tkinter.ttk.Treeview* のメソッド), 1864

Y

ycor() (*turtle* モジュール), 1787
year (*datetime.date* の属性), 234
year (*datetime.datetime* の属性), 242
Year 2038, 790
yeardatescalendar() (*calendar.Calendar* のメソッド), 270
yeardays2calendar() (*calendar.Calendar* のメソッド), 270
yeardayscalendar() (*calendar.Calendar* のメソッド), 270
YESEXPR (*locale* モジュール), 1768
YIELD_FROM (*opcode*), 2366
YIELD_VALUE (*opcode*), 2366
yiq_to_rgb() (*colorsys* モジュール), 1743
yview() (*tkinter.ttk.Treeview* のメソッド), 1864

Z

Zen of Python, **2493**
ZeroDivisionError, 120
zfill() (*bytearray* のメソッド), 83
zfill() (*bytes* のメソッド), 83
zfill() (*str* のメソッド), 65
zip (*2to3 fixer*), 2065
zip() (組み込み関数), 31
ZIP_BZIP2 (*zipfile* モジュール), 617
ZIP_DEFLATED (*zipfile* モジュール), 617
zip_longest() (*itertools* モジュール), 446
ZIP_LZMA (*zipfile* モジュール), 617

ZIP_STORED (*zipfile* モジュール), 617
zipapp (モジュール), 2156
zipapp command line option
    -c, 2157
    --compress, 2157
    -h, 2158
    --help, 2158
    --info, 2158
    -m <mainfn>, 2157
    --main=<mainfn>, 2157
    -o <output>, 2157
    --output=<output>, 2157
    -p <interpreter>, 2157
    --python=<interpreter>, 2157
zipfile (モジュール), 616
ZipFile (*zipfile* のクラス), 618
zipfile command line option
    -c <zipfile> <source1> ... <sourceN>, 627
    --create <zipfile> <source1> ... <sourceN>, 627
    -e <zipfile> <output_dir>, 627
    --extract <zipfile> <output_dir>, 627
    -l <zipfile>, 627
    --list <zipfile>, 627
    -t <zipfile>, 627
    --test <zipfile>, 627
zipimport (モジュール), 2279
zipimporter (*zipimport* のクラス), 2280
ZipImportError, 2280
ZipInfo (*zipfile* のクラス), 617
zlib (モジュール), 595
ZLIB_RUNTIME_VERSION (*zlib* モジュール), 599
ZLIB_VERSION (*zlib* モジュール), 599