
The Python Library Reference

リリース 2.7.17

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

12 月 24, 2019

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

目次

第 1 章	はじめに	3
第 2 章	組み込み関数	5
第 3 章	非必須組み込み関数 (Non-essential Built-in Functions)	33
第 4 章	組み込み定数	35
4.1	site モジュールで追加される定数	36
第 5 章	組み込み型	37
5.1	真理値判定	37
5.2	ブール演算 — and, or, not	38
5.3	比較	38
5.4	数値型 int, float, long, complex	39
5.5	イテレータ型	43
5.6	シーケンス型 — str, unicode, list, tuple, bytearray, buffer, xrange	45
5.7	set (集合) 型 — set, frozenset	60
5.8	マッピング型 — dict	63
5.9	ファイルオブジェクト	69
5.10	メモリビュー型	74
5.11	コンテキストマネージャ型	76
5.12	その他の組み込み型	77
5.13	特殊属性	80
第 6 章	組み込み例外	81
6.1	例外のクラス階層	88
第 7 章	文字列処理	91
7.1	string — 一般的な文字列操作	91
7.2	re — 正規表現操作	106
7.3	struct — 文字列データをパックされたバイナリデータとして解釈する	126
7.4	diffilib — 差分の計算を助ける	132
7.5	StringIO — ファイルのように文字列を読み書きする	145
7.6	cStringIO — 高速化された StringIO	146
7.7	textwrap — テキストの折り返しと詰め込み	147
7.8	codecs — codec レジストリと基底クラス	150
7.9	unicodedata — Unicode データベース	170

7.10	stringprep — インターネットのための文字列調製	172
7.11	fpformat — 浮動小数点数の変換	174
第 8 章	データ型	175
8.1	datetime — 基本的な日付型および時間型	175
8.2	calendar — 一般的なカレンダーに関する関数群	205
8.3	collections — 高性能なコンテナ・データ型	210
8.4	heapq — ヒープキューアルゴリズム	228
8.5	bisect — 配列二分法アルゴリズム	233
8.6	array — 効率のよい数値アレイ	236
8.7	sets — 重複のない要素の順序なしコレクション	240
8.8	sched — イベントスケジューラ	244
8.9	mutex — 排他制御	246
8.10	Queue — 同期キュークラス	247
8.11	weakref — 弱参照	250
8.12	UserDict — 辞書オブジェクトのためのクラスラッパー	255
8.13	UserList — リストオブジェクトのためのクラスラッパー	256
8.14	UserString — 文字列オブジェクトのためのクラスラッパー	257
8.15	types — 組み込み型の名前	258
8.16	new — ランタイム内部オブジェクトの作成	262
8.17	copy — 浅いコピーおよび深いコピー操作	263
8.18	pprint — データ出力の整然化	264
8.19	repr — もう一つの repr() の実装	267
第 9 章	数値と数学モジュール	271
9.1	numbers — 数の抽象基底クラス	271
9.2	math — 数学関数	275
9.3	cmath — 複素数のための数学関数	280
9.4	decimal — 10 進固定及び浮動小数点数の算術演算	284
9.5	fractions — 有理数	315
9.6	random — 擬似乱数を生成する	317
9.7	itertools — 効率的なループ実行のためのイテレータ生成関数	323
9.8	functools — 高次関数と呼び出し可能オブジェクトの操作	339
9.9	operator — 関数形式の標準演算子	342
第 10 章	ファイルとディレクトリへのアクセス	353
10.1	os.path — 共通のパス名操作	353
10.2	fileinput — 複数の入力ストリームをまたいだ行の繰り返し処理をサポートする	358
10.3	stat — stat() の結果を解釈する	361
10.4	statvfs — os.statvfs() で使われる定数群	366
10.5	filecmp — ファイルおよびディレクトリの比較	367
10.6	tempfile — 一時的なファイルやディレクトリの生成	369
10.7	glob — Unix 形式のパス名のパターン展開	373
10.8	fnmatch — Unix ファイル名のパターンマッチ	374
10.9	linecache — テキストラインにランダムアクセスする	376

10.10	shutil — 高水準のファイル操作	376
10.11	dircache — キャッシュされたディレクトリー一覧の生成	382
10.12	macpath — Mac OS 9 のパス操作関数	383
第 11 章	データの永続化	385
11.1	pickle — Python オブジェクトの直列化	385
11.2	cPickle — より高速な pickle	399
11.3	copy_reg — pickle サポート関数を登録する	399
11.4	shelve — Python オブジェクトの永続化	400
11.5	marshal — 内部使用向けの Python オブジェクト整列化	403
11.6	anydbm — DBM 形式のデータベースへの汎用アクセスインタフェース	405
11.7	whichdb — どの DBM モジュールがデータベースを作ったかを推測する	407
11.8	dbm — UNIX dbm のシンプルなインタフェース	408
11.9	gdbm — GNU による dbm の再実装	409
11.10	dbhash — BSD データベースライブラリへの DBM 形式のインタフェース	410
11.11	bsddb — Berkeley DB ライブラリへのインタフェース	412
11.12	dumbdbm — 可搬性のある DBM 実装	415
11.13	sqlite3 — SQLite データベースに対する DB-API 2.0 インタフェース	416
第 12 章	データ圧縮とアーカイブ	441
12.1	zlib — gzip 互換の圧縮	441
12.2	gzip — gzip ファイルのサポート	445
12.3	bz2 — bzip2 互換の圧縮ライブラリ	447
12.4	zipfile — ZIP アーカイブの処理	450
12.5	tarfile — tar アーカイブファイルの読み書き	458
第 13 章	ファイルフォーマット	471
13.1	csv — CSV ファイルの読み書き	471
13.2	ConfigParser — 設定ファイルの構文解析器	481
13.3	robotparser — robots.txt のためのパーザ	489
13.4	netrc — netrc ファイルの処理	490
13.5	xdrllib — XDR データのエンコードおよびデコード	491
13.6	plistlib — Mac OS X .plist ファイルの生成と解析	495
第 14 章	暗号関連のサービス	499
14.1	hashlib — セキュアハッシュおよびメッセージダイジェスト	499
14.2	hmac — メッセージ認証のための鍵付きハッシュ化	502
14.3	md5 — MD5 メッセージダイジェストアルゴリズム	503
14.4	sha — SHA-1 メッセージダイジェストアルゴリズム	505
第 15 章	汎用オペレーティングシステムサービス	507
15.1	os — 雑多なオペレーティングシステムインタフェース	507
15.2	io — ストリームを扱うコアツール	545
15.3	time — 時刻データへのアクセスと変換	559
15.4	argparse — コマンドラインオプション、引数、サブコマンドのパarser	567

15.5	<code>optparse</code> — コマンドラインオプション解析器	601
15.6	<code>getopt</code> — C 言語スタイルのコマンドラインオプションパーサ	634
15.7	<code>logging</code> — Python 用ロギング機能	637
15.8	<code>logging.config</code> — ロギングの環境設定	653
15.9	<code>logging.handlers</code> — ロギングハンドラ	665
15.10	<code>getpass</code> — 可搬性のあるパスワード入力機構	676
15.11	<code>curses</code> — 文字セル表示を扱うための端末操作	676
15.12	<code>curses.textpad</code> — <code>curses</code> プログラムのためのテキスト入力ウィジェット	698
15.13	<code>curses.ascii</code> — ASCII 文字に関するユーティリティ	700
15.14	<code>curses.panel</code> — <code>curses</code> のためのパネルスタック拡張	703
15.15	<code>platform</code> — 実行中プラットフォームの固有情報を参照する	704
15.16	<code>errno</code> — 標準の <code>errno</code> システムシンボル	708
15.17	<code>ctypes</code> — Python のための外部関数ライブラリ	716
第 16 章	オプションのオペレーティングシステムサービス	757
16.1	<code>select</code> — I/O 処理の完了を待機する	757
16.2	<code>threading</code> — 高水準のスレッドインタフェース	763
16.3	<code>thread</code> — マルチスレッドのコントロール	777
16.4	<code>dummy_threading</code> — <code>threading</code> の代替モジュール	779
16.5	<code>dummy_thread</code> — <code>thread</code> の代替モジュール	779
16.6	<code>multiprocessing</code> — プロセススペースの ”並列処理” インタフェース	780
16.7	<code>mmap</code> — メモリマップファイル	842
16.8	<code>readline</code> — GNU <code>readline</code> のインタフェース	846
16.9	<code>rlcompleter</code> — GNU <code>readline</code> 向け補完関数	851
第 17 章	プロセス間通信とネットワーク	853
17.1	<code>subprocess</code> — サブプロセス管理	853
17.2	<code>socket</code> — 低レベルネットワークインタフェース	868
17.3	<code>ssl</code> — ソケットオブジェクトに対する TLS/SSL ラッパー	883
17.4	<code>signal</code> — 非同期イベントにハンドラを設定する	914
17.5	<code>popen2</code> — アクセス可能な I/O ストリームを持つサブプロセス生成	918
17.6	<code>asyncore</code> — 非同期ソケットハンドラ	921
17.7	<code>asynchat</code> — 非同期ソケットコマンド/レスポンスハンドラ	926
第 18 章	インターネット上のデータの操作	931
18.1	<code>email</code> — 電子メールと MIME 処理のためのパッケージ	931
18.2	<code>json</code> — JSON エンコーダおよびデコーダ	972
18.3	<code>mailcap</code> — <code>mailcap</code> ファイルの操作	982
18.4	<code>mailbox</code> — 様々な形式のメールボックス操作	984
18.5	<code>mhlib</code> — MH のメールボックスへのアクセス機構	1008
18.6	<code>mimetools</code> — MIME メッセージを解析するためのツール	1010
18.7	<code>mimetypes</code> — ファイル名を MIME 型へマップする	1012
18.8	<code>MimeWriter</code> — 汎用 MIME ファイルライター	1015
18.9	<code>mimify</code> — 電子メールメッセージの MIME 処理	1016
18.10	<code>multifile</code> — 個別の部分を含んだファイル群のサポート	1018

18.11	rfc822	— RFC 2822 準拠のメールヘッダ読み出し	1020
18.12	base64	— RFC 3548: Base16, Base32, Base64 データの符号化	1025
18.13	binhex	— binhex4 形式ファイルのエンコードおよびデコード	1028
18.14	binascii	— バイナリデータと ASCII データとの間での変換	1029
18.15	quopri	— MIME quoted-printable 形式データのエンコードおよびデコード	1031
18.16	uu	— uuencode 形式のエンコードとデコード	1032
第 19 章 構造化マークアップツール			1035
19.1	HTMLParser	— HTML および XHTML のシンプルなパーサー	1035
19.2	sgmllib	— 単純な SGML パーザ	1041
19.3	htmllib	— HTML 文書の解析器	1045
19.4	htmlentitydefs	— HTML 一般エンティティの定義	1047
19.5	XML を扱うモジュール群		1047
19.6	XML の脆弱性		1048
19.7	xml.etree.ElementTree	— ElementTree XML API	1049
19.8	xml.dom	— 文書オブジェクトモデル (DOM) API	1065
19.9	xml.dom.minidom	— 最小限の DOM の実装	1079
19.10	xml.dom.pulldom	— 部分的な DOM ツリー構築のサポート	1084
19.11	xml.sax	— SAX2 パーサのサポート	1085
19.12	xml.sax.handler	— SAX ハンドラの基底クラス	1087
19.13	xml.sax.saxutils	— SAX ユーティリティ	1094
19.14	xml.sax.xmlreader	— XML パーサのインタフェース	1095
19.15	xml.parsers.expat	— Expat を使った高速な XML 解析	1100
第 20 章 インターネットプロトコルとサポート			1113
20.1	webbrowser	— 便利なウェブブラウザコントローラー	1113
20.2	cgi	— CGI (ゲートウェイインタフェース規格) のサポート	1116
20.3	cgitb	— CGI スクリプトのトレースバック管理機構	1125
20.4	wsgiref	— WSGI ユーティリティとリファレンス実装	1126
20.5	urllib	— URL による任意のリソースへのアクセス	1137
20.6	urllib2	— URL を開くための拡張可能なライブラリ	1146
20.7	httplib	— HTTP プロトコルクライアント	1161
20.8	ftplib	— FTP プロトコルクライアント	1169
20.9	poplib	— POP3 プロトコルクライアント	1175
20.10	imaplib	— IMAP4 プロトコルクライアント	1178
20.11	nntplib	— NNTP プロトコルクライアント	1185
20.12	smtplib	— SMTP プロトコルクライアント	1190
20.13	smtpd	— SMTP サーバー	1196
20.14	telnetlib	— Telnet クライアント	1197
20.15	uuid	— RFC 4122 に準拠した UUID オブジェクト	1200
20.16	urlparse	— URL を解析して構成要素にする	1204
20.17	SocketServer	— ネットワークサーバ構築のためのフレームワーク	1210
20.18	BaseHTTPServer	— 基本的な機能を持つ HTTP サーバ	1219
20.19	SimpleHTTPServer	— 簡潔な HTTP リクエストハンドラ	1223

20.20	CGIHTTPServer — CGI 実行機能付き HTTP リクエスト処理機構	1225
20.21	cookielib — HTTP クライアント用の Cookie 処理	1227
20.22	Cookie — HTTP の状態管理	1238
20.23	xmlrpclib — XML-RPC クライアントアクセス	1243
20.24	SimpleXMLRPCServer — 基本的な XML-RPC サーバー	1252
20.25	DocXMLRPCServer — セルフ-ドキュメンティング XML-RPC サーバ	1257
第 21 章	マルチメディアサービス	1259
21.1	audioop — 生の音声データを操作する	1259
21.2	imageop — 生の画像データを操作する	1263
21.3	aifc — AIFF および AIFC ファイルの読み書き	1264
21.4	sunau — Sun AU ファイルの読み書き	1267
21.5	wave — WAV ファイルの読み書き	1271
21.6	chunk — IFF チャンクデータの読み込み	1274
21.7	colorsys — 色体系間の変換	1275
21.8	imghdr — 画像の形式を決定する	1276
21.9	sndhdr — サウンドファイルの識別	1277
21.10	ossaudiodev — OSS 互換オーディオデバイスへのアクセス	1278
第 22 章	国際化	1285
22.1	gettext — 多言語対応に関する国際化サービス	1285
22.2	locale — 国際化サービス	1297
第 23 章	プログラムのフレームワーク	1307
23.1	cmd — 行指向のコマンドインタプリタのサポート	1307
23.2	shlex — 単純な字句解析	1310
第 24 章	Tk を用いたグラフィカルユーザインターフェイス	1315
24.1	Tkinter — Tcl/Tk への Python インタフェース	1315
24.2	ttk — Tk のテーマ付きウィジェット	1329
24.3	Tix — Tk の拡張ウィジェット	1352
24.4	ScrolledText — スクロールするテキストウィジェット	1358
24.5	turtle — Tk のためのタートルグラフィックス	1359
24.6	IDLE	1396
24.7	他のグラフィカルユーザインタフェースパッケージ	1406
第 25 章	開発ツール	1409
25.1	pydoc — ドキュメント生成とオンラインヘルプシステム	1409
25.2	doctest — 対話的な実行例をテストする	1410
25.3	unittest — ユニットテストフレームワーク	1441
25.4	2to3 - Python 2 から 3 への自動コード変換	1473
25.5	test — Python 用回帰テストパッケージ	1479
25.6	test.support — Utility functions for tests	1482
第 26 章	デバッグとプロファイル	1489
26.1	bdb — デバッガーフレームワーク	1489

26.2	pdb — Python デバッガ	1495
26.3	デバッグコマンド	1497
26.4	Python プロファイラ	1501
26.5	hotshot — ハイパフォーマンス・ロギング・プロファイラ	1511
26.6	timeit — 小さなコード断片の実行時間計測	1513
26.7	trace — Python ステートメント実行のトレースと追跡	1518
第 27 章	ソフトウェア・パッケージと配布	1523
27.1	distutils — Python モジュールの構築とインストール	1523
27.2	ensurepip — pip インストーラのブートストラップ	1524
第 28 章	Python ランタイムサービス	1527
28.1	sys — システムパラメータと関数	1527
28.2	sysconfig — Python の構成情報にアクセスする	1542
28.3	__builtin__ — 組み込みオブジェクト	1546
28.4	future.builtins — Python 3 のビルトイン	1547
28.5	__main__ — トップレベルのスクリプト環境	1548
28.6	warnings — 警告の制御	1548
28.7	contextlib — with 文コンテキスト用ユーティリティ	1555
28.8	abc — 抽象基底クラス	1557
28.9	atexit — 終了ハンドラ	1560
28.10	traceback — スタックトレースの表示または取得	1562
28.11	__future__ — future 文の定義	1566
28.12	gc — ガベージコレクタインターフェース	1568
28.13	inspect — 活動中のオブジェクトの情報を取得する	1571
28.14	site — サイト固有の設定フック	1579
28.15	user — ユーザー設定のフック	1582
28.16	fpectl — 浮動小数点例外の制御	1583
第 29 章	カスタム Python インタプリタ	1587
29.1	code — インタプリタ基底クラス	1587
29.2	codeop — Python コードをコンパイルする	1589
第 30 章	制限実行 (restricted execution)	1593
30.1	rexec — 制限実行のフレームワーク	1594
30.2	Bastion — オブジェクトに対するアクセスの制限	1598
第 31 章	モジュールのインポート	1601
31.1	imp — import 内部へアクセスする	1601
31.2	importlib — __import__() の便利なラッパー	1606
31.3	imputil — Import ユーティリティ	1606
31.4	zipimport — Zip アーカイブからモジュールを import する	1610
31.5	pkgutil — パッケージ拡張ユーティリティ	1612
31.6	modulefinder — スクリプト中で使われているモジュールを検索する	1615
31.7	runpy — Python モジュールの位置特定と実行	1617

第 32 章	Python 言語サービス	1621
32.1	parser — Python 解析木にアクセスする	1621
32.2	ast — 抽象構文木	1626
32.3	symtable — コンパイラの記号表へのアクセス	1633
32.4	symbol — Python 解析木と共に使われる定数	1635
32.5	token — Python 解析木と共に使われる定数	1636
32.6	keyword — Python キーワードチェック	1638
32.7	tokenize — Python ソースのためのトークナイザ	1638
32.8	tabnanny — あいまいなインデントの検出	1640
32.9	pyclbr — Python クラスブラウザサポート	1641
32.10	py_compile — Python ソースファイルのコンパイル	1643
32.11	compileall — Python ライブラリをバイトコンパイル	1643
32.12	dis — Python バイトコードの逆アセンブラ	1646
32.13	pickletools — pickle 開発者のためのツール群	1657
第 33 章	Python コンパイラパッケージ	1659
33.1	基本的なインターフェイス	1659
33.2	制限	1660
33.3	Python 抽象構文	1660
33.4	Visitor を使って AST をわたり歩く	1667
33.5	バイトコード生成	1668
第 34 章	各種サービス	1669
34.1	formatter — 汎用の出力書式化機構	1669
第 35 章	MS Windows 固有のサービス	1675
35.1	msilib — Microsoft インストーラーファイルの読み書き	1675
35.2	msvcrt — MS VC++ 実行時システムの有用なルーチン群	1682
35.3	_winreg — Windows レジストリへのアクセス	1684
35.4	winsound — Windows 用の音声再生インタフェース	1694
第 36 章	Unix 固有のサービス	1697
36.1	posix — 最も一般的な POSIX システムコール群	1697
36.2	pwd — パスワードデータベースへのアクセスを提供する	1698
36.3	spwd — シャドウパスワードデータベース	1699
36.4	grp — グループデータベースへのアクセス	1700
36.5	crypt — Unix パスワードをチェックするための関数	1701
36.6	dl — 共有オブジェクトの C 関数の呼び出し	1702
36.7	termios — POSIX スタイルの端末制御	1704
36.8	tty — 端末制御のための関数群	1705
36.9	pty — 擬似端末ユーティリティ	1705
36.10	fcntl — fcntl および ioctl システムコール	1706
36.11	pipes — シェルパイプラインへのインタフェース	1708
36.12	posixfile — ロック機構をサポートするファイル類似オブジェクト	1711
36.13	resource — リソース使用状態の情報	1713

36.14	<code>nis</code> — Sun の NIS (Yellow Pages) へのインタフェース	1717
36.15	<code>syslog</code> — Unix <code>syslog</code> ライブラリルーチン群	1718
36.16	<code>commands</code> — コマンド実行ユーティリティ	1719
第 37 章 Mac OS X 固有のサービス		1721
37.1	<code>ic</code> — Mac OS X のインターネット設定へのアクセス	1721
37.2	<code>MacOS</code> — Mac OS インタプリタ機能へのアクセス	1723
37.3	<code>macostools</code> — ファイル操作を便利にするルーチン集	1725
37.4	<code>findertools</code> — finder の Apple Events インターフェース	1726
37.5	<code>EasyDialogs</code> — 基本的な Macintosh ダイアログ	1727
37.6	<code>FrameWork</code> — 対話型アプリケーション・フレームワーク	1730
37.7	<code>autoGIL</code> — イベントループ中のグローバルインタプリタの取り扱い	1735
37.8	Mac OS ツールボックスモジュール	1735
37.9	<code>ColorPicker</code> — 色選択ダイアログ	1739
第 38 章 MacPython OSA モジュール		1741
38.1	<code>gensuitemodule</code> — OSA スタブ作成パッケージ	1742
38.2	<code>aetools</code> — OSA クライアントのサポート	1743
38.3	<code>aepack</code> — Python 変数と AppleEvent データコンテナ間の変換	1744
38.4	<code>aetypes</code> — AppleEvent オブジェクト	1746
38.5	<code>MiniAEFrame</code> — オープンスクリプティングアーキテクチャサーバのサポート	1748
第 39 章 SGI IRIX 固有のサービス		1749
39.1	<code>al</code> — SGI のオーディオ機能	1749
39.2	<code>AL</code> — <code>al</code> モジュールで使われる定数	1751
39.3	<code>cd</code> — SGI システムの CD-ROM へのアクセス	1752
39.4	<code>fl</code> — グラフィカルユーザーインターフェースのための FORMS ライブラリ	1756
39.5	<code>FL</code> — <code>fl</code> モジュールで使用される定数	1762
39.6	<code>flp</code> — 保存された FORMS デザインをロードする関数	1762
39.7	<code>fm</code> — <i>Font Manager</i> インターフェース	1763
39.8	<code>gl</code> — <i>Graphics Library</i> インターフェース	1764
39.9	<code>DEVICE</code> — <code>gl</code> モジュールで使われる定数	1766
39.10	<code>GL</code> — <code>gl</code> モジュールで使われる定数	1766
39.11	<code>imgfile</code> — SGI <code>imglib</code> ファイルのサポート	1767
39.12	<code>jpeg</code> — JPEG ファイルの読み書きを行う	1768
第 40 章 SunOS 固有のサービス		1771
40.1	<code>sunaudiodev</code> — Sun オーディオハードウェアへのアクセス	1771
40.2	<code>SUNAUDIODEV</code> — <code>sunaudiodev</code> で使われる定数	1773
第 41 章 ドキュメント化されていないモジュール		1775
41.1	雑多な有用ユーティリティ	1775
41.2	プラットフォーム固有のモジュール	1775
41.3	マルチメディア関連	1775
41.4	文書化されていない Mac OS モジュール	1776

41.5	撤廃されたもの	1777
41.6	SGI 固有の拡張モジュール	1778
付録 A	用語集	1779
付録 B	このドキュメントについて	1791
B.1	Python ドキュメント 貢献者	1791
付録 C	歴史とライセンス	1793
C.1	Python の歴史	1793
C.2	Terms and conditions for accessing or otherwise using Python	1794
C.3	取り入れられているソフトウェアのライセンスと謝辞	1798
付録 D	Copyright	1811
参考文献		1813
Python モジュール索引		1815
索引		1819

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

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

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

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

第 1 章

はじめに

この “Python ライブラリ” には様々な内容が収録されています。

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

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

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

このマニュアルの構成は “内部から外部へ:” つまり、最初に組み込みのデータ型を記述し、組み込みの関数および例外、そして最後に各モジュールといった形になっています。モジュールは関係のあるものでグループ化して一つの章にしています。章の順番付けや各章内のモジュールの順番付けは、大まかに重要性の高いものから低いものになっています。

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

ら始めるとよいでしょう。マニュアルの他の部分は、この節の内容について知っているものとして書かれているからです。

それでは、ショーの始まりです！

第 2 章

組み込み関数

Python インタプリタは数多くの組み込み関数を持っていて、いつでも利用することができます。それらの関数をアルファベット順に挙げます。

		組み込み関数		
<code>abs()</code>	<code>divmod()</code>	<code>input()</code>	<code>open()</code>	<code>staticmethod()</code>
<code>all()</code>	<code>enumerate()</code>	<code>int()</code>	<code>ord()</code>	<code>str()</code>
<code>any()</code>	<code>eval()</code>	<code>isinstance()</code>	<code>pow()</code>	<code>sum()</code>
<code>basestring()</code>	<code>execfile()</code>	<code>issubclass()</code>	<code>print()</code>	<code>super()</code>
<code>bin()</code>	<code>file()</code>	<code>iter()</code>	<code>property()</code>	<code>tuple()</code>
<code>bool()</code>	<code>filter()</code>	<code>len()</code>	<code>range()</code>	<code>type()</code>
<code>bytearray()</code>	<code>float()</code>	<code>list()</code>	<code>raw_input()</code>	<code>unichr()</code>
<code>callable()</code>	<code>format()</code>	<code>locals()</code>	<code>reduce()</code>	<code>unicode()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>long()</code>	<code>reload()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>map()</code>	<code>repr()</code>	<code>xrange()</code>
<code>cmp()</code>	<code>globals()</code>	<code>max()</code>	<code>reversed()</code>	<code>zip()</code>
<code>compile()</code>	<code>hasattr()</code>	<code>memoryview()</code>	<code>round()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hash()</code>	<code>min()</code>	<code>set()</code>	
<code>delattr()</code>	<code>help()</code>	<code>next()</code>	<code>setattr()</code>	
<code>dict()</code>	<code>hex()</code>	<code>object()</code>	<code>slice()</code>	
<code>dir()</code>	<code>id()</code>	<code>oct()</code>	<code>sorted()</code>	

これらに加えて、今では不可欠なものとは考えることがなくなった 4 つの組み込み関数があります: `apply()`, `buffer()`, `coerce()`, `intern()` です。これらは *非必須組み込み関数 (Non-essential Built-in Functions)* セクションで説明しています。

`abs(x)`

数値の絶対値を返します。引数として通常の整数、長整数、浮動小数点数をとることができます。引数が複素数の場合、その大きさ (magnitude) が返されます。

`all(iterable)`

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

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

バージョン 2.5 で追加.

any(*iterable*)

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

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

バージョン 2.5 で追加.

basestring()

この抽象型は、 `str` および `unicode` のスーパークラスです。この型は呼び出したりインスタス化したりはできませんが、オブジェクトが `str` や `unicode` のインスタスであるかどうかを調べる際に利用できます。 `isinstance(obj, basestring)` は `isinstance(obj, (str, unicode))` と等価です。

バージョン 2.3 で追加.

bin(*x*)

整数を二進文字列に変換します。結果は Python の式としても使える形式になります。 *x* が Python の `int` オブジェクトでない場合、整数値を返す `__index__()` メソッドが定義されていなければなりません。

バージョン 2.6 で追加.

class bool(*x*)

ブール値、即ち `True` または `False` のどちらかを返します。 *x* は標準の [真理値判定プロシージャ](#) を用いて変換されます。 *x* が偽または省略されている場合、この関数は `False` を返します。それ以外の場合、 `True` を返します。 `bool` クラスは `int` クラスの派生クラスです。 `bool` からさらに派生することはできません。ブール値のインスタスは `False` と `True` のみです。

バージョン 2.2.1 で追加.

バージョン 2.3 で変更: 引数が与えられなかった場合、この関数は `False` を返します。

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

新しいバイト配列を返します。 `bytearray` クラスは $0 \leq x < 256$ の範囲の整数からなる *mutable* なシーケンスです。 [ミュータブルなシーケンス型](#) に記述されている *mutable* なシーケンスに対する普通のメソッドの大半を備えています。また、 `str` 型が持つメソッドの大半も備えています ([文字列メソッド 参照](#))。

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

- それが *unicode* なら、*encoding* (と、オプションの *errors*) パラメタも与えなければなりません。このとき *bytearray()* は *unicode* を *unicode.encode()* でバイトに変換して返します。
- これが 整数 なら、配列はそのサイズになり、null バイトで初期化されます。
- これが バッファ インタフェースに適合するオブジェクトなら、そのオブジェクトの読み込み専用バッファがバイト配列の初期化に使われます。
- これが イテラブル なら、それは範囲 $0 \leq x < 256$ 内の整数のイテラブルであることが必要で、それらが配列の初期の内容になります。

引数がなければ、長さ 0 の配列が生成されます。

バージョン 2.6 で追加.

callable (*object*)

引数 *object* が呼び出し可能オブジェクトであれば、*True* を返します。そうでなければ、*False* を返します。この関数が真を返しても *object* の呼び出しは失敗する可能性があります、偽を返した場合は決して成功することはありません。クラスは呼び出し可能 (クラスを呼び出すと新しいインスタンスを返します) なことと、クラスのインスタンスがメソッド *__call__()* を持つ場合には呼び出しが可能なることに注意してください。

chr (*i*)

ASCII コードが整数 *i* となるような文字 1 字からなる文字列を返します。例えば、*chr(97)* は文字列 'a' を返します。この関数は *ord()* の逆です。引数は [0..255] の両端を含む範囲内に収まらなければなりません; *i* が範囲外の値のときには *ValueError* が送出されます。 *unichr()* も参照下さい。

classmethod (*function*)

function のクラスメソッドを返します。

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

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

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

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

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

クラスメソッドについて詳しい情報は *types* を参照してください。

バージョン 2.2 で追加.

バージョン 2.4 で変更: 関数デコレータ構文を追加しました.

cmp (*x*, *y*)

二つのオブジェクト *x* および *y* を比較し、その結果に従って整数を返します。戻り値は $x < y$ のときには負、 $x == y$ の時にはゼロ、 $x > y$ には厳密に正の値になります。

compile (*source*, *filename*, *mode* [, *flags* [, *dont_inherit*]])

source をコードオブジェクト、もしくは、AST オブジェクトにコンパイルします。コードオブジェクトは `exec` 文で実行したり、`eval()` 呼び出しで評価できます。*source* は Unicode 文字列、*Latin-1* エンコードのバイト列、AST オブジェクトのいずれでもかまいません。AST オブジェクトへの、また、AST オブジェクトからのコンパイルの方法は、`ast` モジュールのドキュメントを参照してください。

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

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

オプション引数 *flags* および *dont_inherit* は、*source* のコンパイルにどの future 文 ([PEP 236](#) 参照) を作用させるかを制御します。どちらも与えていない (または両方ともゼロ) ならば、`compile()` を呼び出している側のコードで有効な future 文を有効にしてコードをコンパイルします。*flags* が与えられていて、*dont_inherit* は与えられていない (またはゼロ) ならば、それに加えて *flags* に指定された future 文が使われます。*dont_inherit* がゼロでない整数ならば、*flags* の値そのものが使われ、コンパイルの呼び出して周辺で有効な future 文は無視されます。

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

この関数は、コンパイルするソースが不正である場合 `SyntaxError` を、ソースがヌルバイトを含む場合 `TypeError` を送出します。

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

注釈: 複数行に渡るコードの文字列を 'single' や 'eval' モードでコンパイルするとき、入力是一つ以上の改行文字で終端されなければなりません。これは、`code` モジュールで不完全な文と完全な文を検知しやすくするためです。

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

バージョン 2.3 で変更: *flags* と *dont_inherit* 引数が追加されました。

バージョン 2.6 で変更: AST オブジェクトのコンパイルをサポートしました。

バージョン 2.7 で変更: Windows や Mac の改行文字を使えるようになりました。また、`'exec'` モードで改行文字は必要なくなりました。 *optimize* パラメタを追加しました。

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

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

注釈: 文字列から変換するとき、その文字列は中央の `+` や `-` 演算子の周りに空白を含んではなりません。例えば、`complex('1+2j')` はいいですが、`complex('1 + 2j')` は *ValueError* を送出します。

複素数型については [数値型 *int*, *float*, *long*, *complex*](#) に説明があります。

delattr(*object, name*)

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

class dict(***kwarg*)

class dict(*mapping, **kwarg*)

class dict(*iterable, **kwarg*)

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

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

dir(*[object]*)

引数がない場合、現在のローカルスコープにある名前の一覧を返します。引数がある場合、そのオブジェクトの有効な属性の一覧を返そうと試みます。

オブジェクトが `__dir__()` という名のメソッドを持つなら、そのメソッドが呼び出され、属性の一覧を返さなければなりません。これにより、カスタムの `__getattr__()` や `__getattribute__()` 関数を実装するオブジェクトは、*dir()* が属性を報告するやり方をカスタマイズできます。

オブジェクトが `__dir__()` を提供していない場合、定義されていればオブジェクトの `__dict__` 属性から、そして型オブジェクトから、情報を収集しようと試みます。結果の一覧は完全であるとは限らず、また、カスタムの `__getattr__()` を持つ場合、不正確になるかもしれません。

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

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

返されるリストはアルファベット順に並べられています。例えば:

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

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

divmod(*a*, *b*)

2 つの (複素数でない) 数値を引数として取り、整数の除法を行ったときの商と剰余からなるペアを返します。被演算子が型混合である場合、2 進算術演算子での規則が適用されます。通常の整数と長整数の場合、結果は (`a // b`, `a % b`) と同じです。浮動小数点数の場合、結果は (`q`, `a % b`) であり、`q` は通常 `math.floor(a / b)` ですが、そうではなく 1 になることもあります。いずれにせよ、`q * b + a % b` は `a` に非常に近い値になり、`a % b` がゼロでない値の場合、その符号は `b` と同じで、`0 <= abs(a % b) < abs(b)` になります。

バージョン 2.3 で変更: 複素数に対する `divmod()` の使用は廃用されました。

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

列挙オブジェクトを返します。`sequence` はシーケンス型、イテレータ型、反復をサポートする他のオブジェクト型のいずれかでなければなりません。`enumerate()` が返すイテレータの `next()` メソッドは、(ゼロから始まる) カウント値と、値だけ `sequence` を反復操作して得られる、対応するオブジェクトを含むタプルを返します。`enumerate()` はインデックス付けされた値の列: (0, `seq[0]`), (1, `seq[1]`), (2, `seq[2]`), ... を得るのに便利です。例

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

次と等価です:

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

バージョン 2.3 で追加.

バージョン 2.6 で変更: *start* 引数が追加されました。

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

引数には、Unicode 文字列または *Latin-1* エンコードのバイト文字列と、オプションの引数 *globals*、*locals* をとります。 *globals* を与える場合は辞書でなくてはなりません。 *locals* を与える場合は任意のマッピングオブジェクトにできます。

バージョン 2.4 で変更: 以前は *locals* も辞書でなければなりませんでした。

引数 *expression* は Python の表現式 (技術的にいうと、条件のリストです) として構文解釈され、評価されます。このとき辞書 *globals* および *locals* はそれぞれグローバルおよびローカルな名前空間として使われます。 *global* 辞書が存在するが、`'__builtins__'` が欠けている場合、*expression* を解析する前に現在のグローバル変数を *globals* にコピーします。このことから、*expression* は通常、標準の `__builtin__` モジュールへの完全なアクセスを有し、制限された環境が伝播するようになっています。 *locals* 辞書が省略された場合、標準の値として *globals* に設定されます。辞書が両方とも省略された場合、表現式は `eval()` が呼び出されている環境の下で実行されます。構文エラーは例外として報告されます。以下に例を示します:

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

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

ヒント: 文の動的な実行は `exec` 文でサポートされています。ファイルからの文の実行は関数 `execfile()` でサポートされています。関数 `globals()` および `locals()` は、それぞれ現在のグローバルおよびローカルな辞書を返すので、`eval()` や `execfile()` で使うことができます。

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

execfile (*filename* [, *globals* [, *locals*]])

この関数は `exec` 文に似ていますが、文字列の代わりにファイルに対して構文解釈を行います。`import` 文と違って、モジュール管理機構を使いません — この関数はファイルを無条件に読み込み、新たなモジュールを生成しません。^{*1}

引数は、ファイル名と、2つのオプションな辞書です。ファイルはパースされて、(モジュールに対してそうするのと同じように) Python ステートメントのシーケンスとして評価されます。この際 *globals* と *locals* がそれぞれグローバル名前空間、ローカル名前空間として使われます。*locals* を指定する場合は何らかのマッピング型オブジェクトでなければなりません。モジュールレベルではグローバルとローカルは同じ辞書であることを忘れないで下さい。*globals* と *locals* として別々にオブジェクトを渡す場合、コードはクラス定義に埋め込まれたかのように実行されます。

バージョン 2.4 で変更: 以前は *locals* も辞書でなければなりませんでした。

locals 辞書が省略された場合、標準の値として *globals* に設定されます。辞書が両方とも省略された場合、表現式は `execfiles()` が呼び出されている環境の下で実行されます。戻り値は `None` です。

注釈: 標準では *locals* は後に述べる関数 `locals()` のように動作します: 標準の *locals* 辞書に対する変更を試みてはいけません。`execfile()` の呼び出しが返る時にコードが *locals* に与える影響を知りたいなら、明示的に *loacals* 辞書を渡してください。`execfile()` は関数のローカルを変更するための信頼性のある方法として使うことはできません。

file (*name* [, *mode* [, *buffering*]])

file 型のコンストラクタです。詳しくは [ファイルオブジェクト](#) 節を参照してください。コンストラクタの引数は後述の `open()` 組み込み関数と同じです。

ファイルを開くときは、このコンストラクタを直接呼ばずに `open()` を呼び出すのが望ましい方法です。*file* は型テストにより適しています (たとえば `isinstance(f, file)` と書くような)。

バージョン 2.2 で追加。

filter (*function*, *iterable*)

iterable のうち、*function* が真を返すような要素からなるリストを構築します。*iterable* はシーケンスか、反復をサポートするコンテナか、イテレータです。*iterable* が文字列型かタプル型の場合、結果も同じ型になります。そうでない場合はリストとなります。*function* が `None` の場合、恒等関数を仮定します。すなわち、*iterable* の偽となる要素は除去されます。

function が `None` ではない場合、`filter(function, iterable)` は `[item for item in iterable if function(item)]` と同等です。*function* が `None` の場合 `[item for item in iterable if item]` と同等です。

この関数のイテレータ版である `itertools.ifilter()` と `itertools.ifilterfalse()` についても参照して下さい、変種として *function* が `false` を返す場合に要素を返す変種も含んでいます。

^{*1} この関数は比較的に利用されない関数なので、構文になるかどうかは保証できません。

class float (*[x]*)

数または文字列 *x* から生成された浮動小数点数を返します。

引数が文字列の場合、十進の数または浮動小数点数を含んでいなければなりません。符号が付いていてもかまいません。また、空白文字中に埋め込まれていてもかまいません。引数は `[+|-]nan`、`[+|-]inf` であっても構いません。それ以外の場合、引数は通常整数、長整数、または浮動小数点数をとることができ、同じ値の浮動小数点数が (Python の浮動小数点精度で) 返されます。引数が指定されなかった場合、`0.0` を返します。

注釈: 文字列で値を渡す際、背後の C ライブラリによって NaN および Infinity が返されるかもしれません。float は文字列、`nan`、`inf`、および `-inf` を、それぞれ、NaN、正の無限大、負の無限大として解釈します。大文字小文字の違い、+ 記号、および、`nan` に対する - 記号は無視されます。

浮動小数点数型については、[数値型 `int`, `float`, `long`, `complex`](#) も参照下さい。

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

value を *format_spec* で制御される "フォーマット化" 表現に変換します。 *format_spec* の解釈は *value* 引数の型に依存しますが、ほとんどの組み込み型で使われる標準的な構文が存在します: [書式指定ミニ言語仕様](#)。

注釈: `format(value, format_spec)` は単に `value.__format__(format_spec)` を呼び出すだけです。

バージョン 2.6 で追加。

class frozenset (*[iterable]*)

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

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

バージョン 2.4 で追加。

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

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

globals ()

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

hasattr (*object*, *name*)

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

hash (*object*)

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

help ([*object*])

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

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

バージョン 2.2 で追加。

hex (*x*)

(任意のサイズの) 整数を、先頭に "0x" が付いた小文字の 16 進数文字列に変換します。例えば:

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
>>> hex(1L)
'0x1L'
```

x が Python の `int` オブジェクトでも `long` オブジェクトでもない場合、文字列を返す `__hex__()` メソッドを定義していなければなりません。

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

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

バージョン 2.4 で変更: 以前は符号なしのリテラルしか返しませんでした。

id (*object*)

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

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

input ([*prompt*])

`eval(raw_input(prompt))` と同じです。

この関数はユーザエラーを捕捉しません。入力が構文的に正しくない場合、`SyntaxError` が送出されます。式を評価する際にエラーが生じた場合、他の例外も送出されるかもしれません。

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

一般的なユーザからの入力のための関数としては `raw_input()` を使うことを検討してください。

class int (*x=0*)

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

数値または文字列 *x* から生成された整数を返します。引数を与えられない場合には 0 を返します。*x* が数値である場合は、通常整数、長整数、または浮動小数点数を返します。浮動小数点数については、これは 0 に向かう方向へ切り詰めます。引数が整数で通常整数の範囲外であれば、長整数を代わりに返します。

x が数値でない場合、あるいは *base* が与えられた場合、*x* は基数 *base* の integer literal で表された、バイト文字列または Unicode 文字列、のインスタンスでなければなりません。オプションで、リテラルの前に + あるいは - を (中間のスペースなしで) 付けることができます。また、リテラルは余白で囲むことができます。基数 *n* のリテラルの各桁は 0 から *n*-1 の数を、値 10-35 を持つ a から z (または A から Z) を含んで表現します。デフォルトの *base* は 10 です。許される値は 0 と 2-36 です。基数 2, 8, 16 のリテラルは、オプションでコード中の整数リテラルのように 0b/0B, 0o/0O/0, 0x/0X を前に付けることができます。基数 0 を渡すと、文字列を、実際の基数 2, 8, 10, 16 のどれかで整数リテラルとして正確に解釈します。

整数型については、[数値型 `int`, `float`, `long`, `complex`](#) も参照下さい。

isinstance (*object*, *classinfo*)

引数 *object* が引数 *classinfo* のインスタンスであるか、(直接または間接的な、もしくは [virtual](#) の) サブクラスのインスタンスの場合に真を返します。また、*classinfo* が型オブジェクト (新スタイルクラス) であり、*object* がその型のオブジェクトであるか、または、(直接または間接的な、もしくは [virtual](#) の) サブクラスの場合にも真を返します。*object* がクラスインスタンスや与えられた型のオブジェクトでない場合、この関数は常に偽を返します。*classinfo* がクラスや型オブジェクトのタプル (あるいはそういったタプルを再帰的に含むタプル) の場合、*object* がそれらクラスや型のいずれかのインスタンスであれば真を返します。*classinfo* がクラス、型、クラスや型からなるタプル、そういったタプルが再帰構造をとっているタプルのいずれでもない場合、例外 `TypeError` が送出されます。

バージョン 2.2 で変更: 型情報からなるタプルへのサポートが追加されました。

issubclass (*class*, *classinfo*)

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

バージョン 2.3 で変更: 型情報からなるタプルへのサポートが追加されました。

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

`iterator` (イテレータ) オブジェクトを返します。2 つ目の引数があるかどうかで、最初の引数の解釈は非常に異なります。2 つ目の引数がない場合、`o` は反復プロトコル (`__iter__()` メソッド) か、シーケンス型プロトコル (引数が 0 から開始する `__getitem__()` メソッド) をサポートする集合オブジェクトでなければなりません。これらのプロトコルが両方ともサポートされていない場合、`TypeError` が送出されます。2 つ目の引数 `sentinel` が与えられていれば、`o` は呼び出し可能なオブジェクトでなければなりません。この場合に生成されるイテレータは、`next()` を呼び出す毎に `o` を引数無しで呼び出します。返された値が `sentinel` と等しければ、`StopIteration` が送出されます。そうでない場合、戻り値がそのまま返されます。

`iter()` の 2 つめの形式の便利な使用法の一つは、ファイルの行を特定の行まで読み進めることです。以下の例では `readline()` が空文字列を返すまでファイルを読み進めます:

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

バージョン 2.2 で追加.

len(*s*)

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

class list([*iterable*])

iterable の要素と同じ要素をもち、かつ順番も同じなリストを返します。*iterable* はシーケンス、反復処理をサポートするコンテナ、あるいはイテレータオブジェクトです。*sequence* がすでにリストの場合、`iterable[:]` と同様にコピーを作成して返します。例えば、`list('abc')` は `['a', 'b', 'c']` および `list((1, 2, 3))` は `[1, 2, 3]` を返します。引数が与えられなかった場合、新しい空のリスト `[]` を返します。

`list` は変更可能なシーケンス型であり、[シーケンス型 — *str*, *unicode*, *list*, *tuple*, *bytearray*, *buffer*, *xrange*](#) に記述があります。他のコンテナ型については組み込み型の *dict*, *set*, および *tuple* クラスと、`collections` モジュールを参照下さい。

locals()

現在のローカルシンボルテーブルを表す辞書を更新して返します。関数ブロックで `locals()` を呼び出した場合自由変数が返されます、クラスブロックでは返されません。

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

class long (*x*=0)

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

文字列または数値 *x* から長整数オブジェクトを構築して返します。引数が文字列の場合は、任意サイズの符号付の数を表していなければなりません。空白で囲まれていても構いません。*base* 引数は `int()` のものと同じように解釈され、また、*x* が文字列の時だけ与えることができます。文字列以外の引数は通常整数、長整数、浮動小数点数を与えることが出来、同じ値としての長整数を返します。浮動小数

点数から整数への変換では、(ゼロに向かう方向へ) 切り詰められます。引数が与えられなければ 0L を返します。

長整数型については、[数値型](#) *int*, *float*, *long*, *complex* も参照下さい。

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

function を *iterable* の全ての要素に適用し、返された値からなるリストを返します。追加の *iterable* 引数を与えた場合、*function* はそれらを引数として取らなければならない、関数はそのリストの全ての要素について個別に適用されます; 他のリストより短いリストがある場合、要素 `None` で延長されます。*function* が `None` の場合、恒等関数であると仮定されます; すなわち、複数のリスト引数が存在する場合、*map*() は全てのリスト引数に対し、対応する要素からなるタプルからなるリストを返します (転置操作のようなものです)。*list* 引数はどのようなシーケンス型でもかまいません; 結果は常にリストになります。

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

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

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

キーワード無しの引数が 1 つだけ与えられた場合、*iterable* は空でない (文字列、タプル、リストなどの) *iterable* でなくてはなりません。 *iterable* の最大の要素が返されます。 2 つ以上のキーワード無しの引数が与えられた場合、その引数の中で最大のものが返されます。

オプションの *key* 引数には `list.sort()` で使われるのと同じような 1 引数の順序付け関数を指定します。 *key* を指定する場合はキーワード形式でなければなりません (たとえば `max(a, b, c, key=func)`)。

バージョン 2.5 で変更: オプションの *key* 引数が追加されました。

memoryview (*obj*)

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

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

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

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

キーワード無しの引数が 1 つだけ与えられた場合、*iterable* は空でない (文字列、タプル、リストなどの) *iterable* でなくてはなりません。 *iterable* の最小の要素が返されます。 2 つ以上のキーワード無しの引数が与えられた場合、その引数の中で最小のものが返されます。

オプションの *key* 引数には `list.sort()` で使われるのと同じような 1 引数の順序付け関数を指定します。 *key* を指定する場合はキーワード形式でなければなりません (たとえば `min(a, b, c, key=func)`)。

バージョン 2.5 で変更: オプションの *key* 引数が追加されました。

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

iterator から、*next*() メソッドにより、次の要素を取得します。 もし、*default* が与えられると、イテレータが空である場合に、それが返されます。 それ以外の場合は、*StopIteration* が送出されます。

バージョン 2.6 で追加.

class object

ユーザ定義の属性やメソッドを持たない、新しいオブジェクトを返します。 `object()` は新スタイルのクラスの、基底クラスです。これは、新スタイルのクラスのインスタンスに共通のメソッド群を持ちます。

バージョン 2.2 で追加.

バージョン 2.3 で変更: この関数はいかなる引数も受け付けません。以前は、引数を受理しましたが無視していました。

oct (x)

(任意のサイズの) 整数を 8 進の文字列に変換します。結果は Python の 式としても使える形式になります。

バージョン 2.4 で変更: 以前は符号なしのリテラルしか返しませんでした。

open (name[, mode[, buffering]])

ファイルを開いて、 **ファイルオブジェクト** にて説明される、 `file` オブジェクトを返します。もし、ファイルが開けないなら、 `IOError` が送出されます。ファイルを開くときは `file` のコンストラクタを直接呼ばずに `open()` を使うのが望ましい方法です。

最初の 2 つの引数は `stdio` の `fopen()` と同じです: `filename` は開きたいファイルの名前で、 `mode` はファイルをどのようにして開くかを指定します。

最もよく使われる `mode` の値は、読み出しの `'r'`、書き込み (ファイルがすでに存在すれば切り詰められます) の `'w'`、追記書き込みの `'a'` です (いくつかの Unix システムでは、全ての書き込みが現在のファイルシーク位置に関係なくファイルの末尾に追加されます)。 `mode` が省略された場合、標準の値は `'r'` になります。デフォルトではテキストモードでファイルを開きます。 `'\n'` 文字は、プラットフォームでの改行の表現に変換されます。移植性を高めるために、バイナリファイルを開くときには、 `mode` の値に `'b'` を追加しなければなりません。 (バイナリファイルとテキストファイルを区別なく扱うようなシステムでも、ドキュメンテーションの代わりになるので便利です。) 他に `mode` に与えられる可能性のある値については後述します。

オプションの `buffering` 引数は、ファイルのために必要とするバッファのサイズを指定します: 0 は非バッファリング、1 は行単位バッファリング、その他の正の値は指定した値 (の近似値) の (バイト) サイズをもつバッファを使用することを意味します。 `buffering` の値が負の場合、システムの標準を使います。通常、tty 端末は行単位のバッファリングであり、その他のファイルは完全なバッファリングです。省略された場合、システムの標準の値が使われます。*2

`'r+'`, `'w+'`, および `'a+'` はファイルを更新のために開きます (開いて読み書きします); `'w+'` はファイルがすでに存在すれば切り詰めるので注意してください。バイナリとテキストファイルを区別するシステムでは、ファイルをバイナリモードで開くためには `'b'` を追加してください; 区別しないシステムでは `'b'` は無視されます。

*2 現状では、 `setvbuf()` を持っていないシステムでは、バッファサイズを指定しても効果はありません。バッファサイズを指定するためのインタフェースは `setvbuf()` を使っては行われていません。何らかの I/O が実行された後で呼び出されるとコアダンプすることがあり、どのような場合にそうなるかを決定する信頼性のある方法がないからです。

標準の `fopen()` における `mode` の値に加えて、`'U'` または `'rU'` を使うことができます。Python が *universal newlines* サポートを行っている (標準ではしていません) 場合、ファイルがテキストファイルで開かれますが、行末文字として Unix における慣行である `'\n'`、Macintosh における慣行である `'\r'`、Windows における慣行である `'\r\n'` のいずれを使うこともできます。これらの改行文字の外部表現はどれも、Python プログラムからは `'\n'` に見えます。Python が *universal newlines* サポートなしで構築されている場合、`mode 'U'` は通常のテキストモードと同様になります。開かれたファイルオブジェクトはまた、`newlines` と呼ばれる属性を持っており、その値は `None` (改行が見つからなかった場合)、`'\n'`、`'\r'`、`'\r\n'`、または見つかった全ての改行タイプを含むタプルになります。

`'U'` を取り除いた後のモードは `'r'`、`'w'`、`'a'` のいずれかで始まる、というのが Python における規則です。

Python では、`fileinput`、`os`、`os.path`、`tempfile`、`shutil` などの多数のファイル操作モジュールが提供されています。

バージョン 2.5 で変更: モード文字列の先頭についての制限が導入されました。

`ord(c)`

長さ 1 の与えられた文字列に対し、その文字列が unicode オブジェクトならば Unicode コードポイントを表す整数を、8 ビット文字列ならばそのバイトの値を返します。たとえば、`ord('a')` は整数 97 を返し、`ord(u'\u2020')` は 8224 を返します。この値は 8 ビット文字列に対する `chr()` の逆であり、unicode オブジェクトに対する `unichr()` の逆です。引数が unicode で Python が UCS2 Unicode 対応版ならば、その文字のコードポイントは両端を含めて `[0..65535]` の範囲に入っていなければなりません。この範囲から外れると文字列の長さが 2 になり、`TypeError` が送出されることになります。

`pow(x, y[, z])`

x の y 乗を返します; z があれば、 x の y 乗に対する z のモジュロを返します (`pow(x, y) % z` より効率よく計算されます)。二引数の形式 `pow(x, y)` は、冪乗演算子を使った `x**y` と等価です。

引数は数値型でなくてはなりません。型混合の場合、2 進算術演算における型強制規則が適用されます。通常整数、および、長整数の被演算子に対しては、二つ目の引数が負の数でない限り、結果は (型強制後の) 被演算子と同じ型になります; 負の場合、全ての引数は浮動小数点型に変換され、浮動小数点型の結果が返されます。例えば、`10**2` は 100 を返しますが、`10**-2` は 0.01 を返します。(最後に述べた機能は Python 2.2 で追加されたものです。Python 2.1 以前では、双方の引数が整数で二つ目の値が負の場合、例外が送出されます。) 二つ目の引数が負の場合、三つめの引数は無視されます。 z がある場合、 x および y は整数型でなければならず、 y は非負の値でなくてはなりません (この制限は Python 2.2 で追加されました。Python 2.1 以前では、3 つの浮動小数点引数を持つ `pow()` は浮動小数点の丸めに関する偶発誤差により、プラットフォーム依存の結果を返します)。

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

`object` (複数でも可) を `sep` で区切りながらストリーム `file` に表示し、最後に `end` を表示します。 `sep`、`end` そして `file` が与えられる場合、キーワード引数として与えられる必要があります。

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

file 引数は、`write(string)` メソッドを持つオブジェクトでなければなりません。指定されないか、`None` であった場合には、`sys.stdout` が使われます。出力のバッファリングは *file* により決定されます。たとえば即座に画面に現れて欲しければ、`file.flush()` を使ってそれを保障してください。

注釈: この関数は `print` という名前が `print` ステートメントとして解釈されるため、通常は使用できません。ステートメントを無効化して、`print()` 関数を使うためには、以下の `future` ステートメントをモジュールの最初に書いて下さい。:

```
from __future__ import print_function
```

バージョン 2.6 で追加.

class property (*[fget[, fset[, fdel[, doc]]]*)

new-style class (新しい形式のクラス) (*object* から派生したクラス) における `property` 属性を返します。

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

典型的な使用法は、属性 *x* の処理の定義です:

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

    def getx(self):
        return self._x

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

    def delx(self):
        del self._x

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

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

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

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

    @property
```

(次のページに続く)

(前のページからの続き)

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

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

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

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

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

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

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

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

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

バージョン 2.2 で追加.

バージョン 2.5 で変更: `doc` が与えられない場合に `fget` のドキュメント文字列を使う。

バージョン 2.6 で変更: `getter`, `setter` そして `deleter` 属性が追加されました。

`range(stop)`

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

等差数列を含むリストを生成するための多機能関数です。for ループでよく使われます。引数は通常の整数でなければなりません。step 引数が無視された場合、標準の値 1 になります。start 引数が省略された場合、標準の値 0 になります。完全な形式では、通常の整数列 `[start, start + step, start + 2 * step, ...]` を返します。step が正の値の場合、最後の要素は stop よりも小さい `start + i * step` の最大値になります; step が負の値の場合、最後の要素は stop よりも大きい `start + i * step` の最小値になります。step はゼロであってはなりません (さもなければ `ValueError` が送出されます)。以下に例を示します。:

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

`raw_input([prompt])`

引数 *prompt* が存在する場合、末尾の改行を除いて標準出力に出力されます。次に、この関数は入力から 1 行を読み込んで文字列に変換して (末尾の改行を除いて) 返します。EOF が読み込まれると `EOFError` が送出されます。以下に例を示します。:

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

`readline` モジュールが読み込まれていれば、`raw_input()` は精緻な行編集およびヒストリ機能を提供します。

`reduce(function, iterable[, initializer])`

iterable の要素に対して、*iterable* を単一の値に短縮するような形で 2 つの引数をもつ *function* を左から右に累積的に適用します。例えば、`reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` は `((((1+2)+3)+4)+5)` を計算します。左引数 *x* は累計の値になり、右引数 *y* は *iterable* から取り出した更新値になります。オプションの *initializer* が存在する場合、計算の際に *iterable* の先頭に置かれます。また、*iterable* が空の場合には標準の値になります。*initializer* が与えられておらず、*iterable* が単一の要素しか持っていない場合、最初の要素が返されます。これは大体以下と等価です:

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        try:
            initializer = next(it)
        except StopIteration:
            raise TypeError('reduce() of empty sequence with no initial value')
    accum_value = initializer
    for x in it:
        accum_value = function(accum_value, x)
    return accum_value
```

`reload(module)`

すでにインポートされた *module* を再解釈し、再初期化します。引数はモジュールオブジェクトでなければならないので、予めインポートに成功していなければなりません。この関数はモジュールのソースコードファイルを外部エディタで編集して、Python インタプリタから離れることなく新しいバージョンを試したい際に有効です。戻り値は (*module* 引数と同じ) モジュールオブジェクトです。

`reload(module)` を実行すると、以下の処理が行われます:

- Python モジュールのコードは再コンパイルされ、モジュールレベルのコードは再度実行されます。モジュールの辞書中にある、何らかの名前に結び付けられたオブジェクトを新たに定義します。拡張モジュール中の `init` 関数が二度呼び出されることはありません。
- Python における他のオブジェクトと同様、以前のオブジェクトのメモリ領域は、参照カウントがゼロにならないかぎり再利用されません。
- モジュール名前空間内の名前は新しいオブジェクト (または更新されたオブジェクト) を指すよう更新されます。
- 以前のオブジェクトが (外部の他のモジュールなどからの) 参照を受けている場合、それらを新たなオブジェクトに再束縛し直すことはないので、必要なら自分で名前空間を更新しなければなりません。

いくつか補足説明があります:

モジュールが再ロードされた際、その辞書 (モジュールのグローバル変数を含みます) はそのまま残ります。名前の再定義を行うと、以前の定義を上書きするので、一般的には問題はありません。新たなバージョンのモジュールが古いバージョンで定義された名前を定義していない場合、古い定義がそのまま残ります。辞書がグローバルテーブルやオブジェクトのキャッシュを維持していれば、この機能をモジュールを有効性を引き出すために使うことができます — つまり、`try` 文を使えば、必要に応じてテーブルがあるかどうかをテストし、その初期化を飛ばすことができます:

```
try:
    cache
except NameError:
    cache = {}
```

組み込みモジュールや動的にロードされるモジュールを再ロードすることは、一般的にそれほど便利ではありません。 `sys`, `__main__`, `builtins` やその他重要なモジュールの再ロードはお勧め出来ません。多くの場合、拡張モジュールは 1 度以上初期化されるようには設計されておらず、再ロードされた場合には何らかの理由で失敗するかもしれません。

一方のモジュールが `from ... import ...` を使って、オブジェクトを他方のモジュールからインポートしているなら、他方のモジュールを `reload()` で呼び出しても、そのモジュールからインポートされたオブジェクトを再定義することはできません — この問題を回避する一つの方法は、`from` 文を再度実行することで、もう一つの方法は `from` 文の代わりに `import` と限定的な名前 (`module.*name*`) を使うことです。

あるモジュールがクラスのインスタンスを生成している場合、そのクラスを定義しているモジュールの再ロードはそれらインスタンスのメソッド定義に影響しません — それらは古いクラス定義を使い続けます。これは派生クラスの場合でも同じです。

repr (*object*)

オブジェクトの印字可能な表現を含む文字列を返します。これは型変換で得られる (逆クオートの) 値と同じです。通常の関数としてこの操作にアクセスできるとたまに便利です。この関数は多くの型について、`eval()` に渡されたときに同じ値を持つようなオブジェクトを表す文字列を生成しようとします。そうでない場合は、角括弧に囲まれたオブジェクトの型の名前と追加の情報 (大抵の場合はオブジェクトの名前とアドレスを含みます) を返します。クラスは、`__repr__()` メソッドを定義することで、この関数によりそのクラスのインスタンスが返すものを制御することができます。

reversed (*seq*)

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

バージョン 2.4 で追加。

バージョン 2.6 で変更: カスタムの `__reversed__()` メソッドが使えるようになりました。

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

number を小数点以下 *ndigits* 桁で丸めた浮動小数点数の値を返します。 *ndigits* が省略されると、デフォルトはゼロになります。結果は浮動小数点数です。値は最も近い 10 のマイナス *ndigits* 乗の倍数に丸められます。二つの倍数との距離が等しい場合、ゼロから離れる方向に丸められます (従って、例えば `round(0.5)` は 1.0 になり、`round(-0.5)` は -1.0 になります)。

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

class set ([*iterable*])

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

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

バージョン 2.4 で追加。

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

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

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

`range(start, stop, step)` で指定されるインデクスの集合を表す、スライス (*slice*) オブジェクトを返します。引数 *start* および *step* はデフォルトでは `None` です。スライスオブジェクトは読み出し専用の属性 *start*、*stop* および *step* を持ち、これらは単に引数で使われた 値 (またはデ

フォルト値) を返します。これらの値には、その他のはっきりとした機能はありません。しかしながら、これらの値は Numerical Python および、その他のサードパーティによる拡張で利用されています。スライスオブジェクトは拡張されたインデックス指定構文が使われる際にも生成されます。例えば `a[start:stop:step]` や `a[start:stop, i]` です。この関数の代替となるイテレータを返す関数、`itertools.islice()` も参照してください。

sorted(*iterable*[, *cmp*[, *key*[, *reverse*]]])

iterable の要素を並べ替えた新たなリストを返します。

オプション引数 *cmp*, *key*, および *reverse* の意味は `list.sort()` メソッドと同じです。(ミュータブルなシーケンス型 節に説明があります。)

cmp は 2 つの引数 (*iterable* の要素) からなるカスタムの比較関数を指定します。これは最初の引数が 2 つ目の引数に比べて小さい、等しい、大きいかに応じて負数、ゼロ、正数を返します。 `cmp=lambda x, y: cmp(x.lower(), y.lower())`。デフォルト値は `None` です。

key は 1 つの引数からなる関数を指定します。これはリストの各要素から比較のキーを取り出すのに使われます: `key=str.lower`。デフォルト値は `None` です (要素を直接比較します)。

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

一般的に、*key* および *reverse* の変換プロセスは同等の *cmp* 関数を指定するより早く動作します。これは *key* および *reverse* がそれぞれの要素に一度だけ触れる間に、*cmp* はリストのそれぞれの要素に対して複数回呼ばれることによるものです。旧式の *cmp* 関数を *key* 関数に変換するには `functools.cmp_to_key()` を使用してください。

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

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

バージョン 2.4 で追加。

staticmethod(*function*)

function の静的メソッドを返します。

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

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

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

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

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

静的メソッドについて詳しい情報は `types` を参照してください。

バージョン 2.2 で追加.

バージョン 2.4 で変更: 関数デコレータ構文を追加しました.

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

オブジェクトをうまく印字可能な形に表現したものを含む文字列を返します。文字列に対してはその文字列自体を返します。`repr(object)` との違いは、`str(object)` は常に `eval()` が受理できるような文字列を返そうと試みるわけではないという点です; この関数の目的は印字可能な文字列を返すところにあります。引数が与えられなかった場合、空の文字列 '' を返します。

文字列についての詳細は、シーケンスの機能についての説明、[シーケンス型 — str, unicode, list, tuple, bytearray, buffer, xrange](#) を参照下さい (文字列はシーケンスです)。また、文字列特有のメソッドについては、[文字列メソッド](#) を参照下さい。整形した文字列を出力するためには、テンプレート文字列か、[文字列フォーマット操作](#) にて説明される `%` 演算子を使用して下さい。さらには、[文字列処理](#) と `unicode()` も参照下さい。

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

start と *iterable* の要素を左から右へ合計し、総和を返します。*start* はデフォルトで 0 です。*iterable* の要素は通常は数値で、*start* の値は文字列であってはなりません。

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

バージョン 2.3 で追加.

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

メソッドの呼び出しを *type* の親または兄弟クラスに委譲するプロキシオブジェクトを返します。これはクラスの中でオーバーライドされた継承メソッドにアクセスするのに便利です。探索の順序は、*type* 自身が飛ばされるのをのぞいて `getattr()` で使われるのと同じです。

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

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

注釈: `super()` は *new-style class* に対してのみ動作します。

`super` の典型的な用途は 2 つあります。単一の継承をしているクラス階層構造では、`super` は名前を明

示することなく親クラスを参照するのに使え、これでコードはメンテナンスしやすくなります。この用途の `super` は他のプログラミング言語で見られるものと近い方向性です。

2 つ目の用途は、動的な実行環境下での複数の継承の共同をサポートすることです。この用途は Python 特有で、静的にコンパイルされる言語や、単一の継承しかサポートしない言語では見られないものです。これは複数の基底クラスが同じメソッドを実装する "diamond diagram" を実装できるようにします。良い設計のために、このメソッドがすべての場合に同じ形式で呼び出せるべきです (呼び出しの順序が実行時に決定されることや、順序がクラスの階層の変更に対応することや、その順序には実行時まで未知の兄弟クラスが含まれることが理由です)。

両方のケースにおいて、典型的なスーパークラスの呼び出しはこうになるでしょう。

```
class C(B):
    def method(self, arg):
        super(C, self).method(arg)
```

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

また、`super()` の使用がメソッド内部に限定されないことにも注目して下さい。引数を 2 つ渡す形式の呼び出しは、必要な要素を正確に指定するので、適当な参照を作ることができます。

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

バージョン 2.2 で追加.

`tuple([iterable])`

`iterable` の要素と要素が同じで、かつ順番も同じになるタプルを返します。 `iterable` はシーケンス型、反復をサポートするコンテナ型、およびイテレータオブジェクトをとることができます。 `iterable` がすでにタブルの場合、そのタブルを変更せずに返します。例えば、`tuple('abc')` は ('a', 'b', 'c') を返し、`tuple([1, 2, 3])` は (1, 2, 3) を返します。

`tuple` クラスは、不変のシーケンス型で、[シーケンス型 — str, unicode, list, tuple, bytearray, buffer, xrange](#) にて説明されます。他のコンテナ型については、組み込みクラスの `dict`, `list`, および `set` と、`collections` モジュールを参照下さい。

`class type(object)`

`class type(name, bases, dict)`

一つの引数を取り、`object` の型を返します。戻り値は型オブジェクトです。オブジェクトの型の検査には `isinstance()` 組み込み関数を使うことが推奨されます。

引数が 3 つの場合、新しい型オブジェクトを返します。本質的には `class` 文の動的な形式です。 `name` 文字列はクラス名で、`__name__` 属性になります。 `bases` タプルは基底クラスの羅列で、`__bases__` 属性になります。 `dict` 辞書はクラス本体の定義を含む名前空間で、`__dict__` 属性になります。たとえば、以下の二つの文は同じ `type` オブジェクトを作ります:

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

バージョン 2.2 で追加.

`unichr(i)`

Unicode におけるコードが整数 *i* になるような文字 1 文字からなる Unicode 文字列を返します。例えば、`unichr(97)` は文字列 `u'a'` を返します。この関数は Unicode 文字列に対する `ord()` の逆です。引数の正当な範囲は Python がどのように構成されているかに依存しています — UCS2 ならば `[0..0xFFFF]` であり UCS4 ならば `[0..0x10FFFF]` であり、このどちらかです。それ以外の値に対しては `ValueError` が送出されます。ASCII の 8 ビットの文字列に対しては、`chr()` を参照下さい。

バージョン 2.0 で追加.

`unicode(object=’')`

`unicode(object[, encoding[, errors]])`

以下のモードのうち一つを使って、*object* の Unicode 文字列バージョンを返します:

もし *encoding* かつ/または *errors* が与えられていれば、`unicode()` は 8 ビットの文字列または文字列バッファになっているオブジェクトを *encoding* の codec を使ってデコードします。*encoding* 引数はエンコーディング名を与える文字列です; 未知のエンコーディングの場合、`LookupError` が送出されます。エラー処理は *errors* に従って行われます; このパラメータは入力エンコーディング中で無効な文字の扱い方を指定します。*errors* が `'strict'` (標準の設定です) の場合、エラー発生時には `ValueError` が送出されます。一方、`'ignore'` では、エラーは暗黙のうちに無視されるようになり、`'replace'` では公式の置換文字、`U+FFFD` を使って、デコードできなかった文字を置き換えます。`codecs` モジュールについても参照してください。

オプションのパラメータが与えられていない場合、`unicode()` は `str()` の動作をまねます。ただし、8 ビット文字列ではなく、Unicode 文字列を返します。もっと詳しくいえば、*object* が Unicode 文字列かそのサブクラスなら、デコード処理を一切介することなく Unicode 文字列を返すということです。

`__unicode__()` メソッドを提供しているオブジェクトの場合、`unicode()` はこのメソッドを引数なしで呼び出して Unicode 文字列を生成します。それ以外のオブジェクトの場合、8 ビットの文字列か、オブジェクトのデータ表現 (representation) を呼び出し、その後デフォルトエンコーディングで `'strict'` モードの codec を使って Unicode 文字列に変換します。

Unicode 文字列についてのさらなる情報については、シーケンス型の機能 についての説明、[シーケンス型 — `str`, `unicode`, `list`, `tuple`, `bytearray`, `buffer`, `xrange`](#) を参照下さい (Unicode 文字列はシーケンスです)。また、文字列特有のメソッドについては、[文字列メソッド](#) を参照下さい。整形した文字列を出力するためには、テンプレート文字列か、[文字列フォーマット操作](#) にて説明される `%` 演算子を使用して下さい。さらには、[文字列処理](#) と `str()` も参照下さい。

バージョン 2.0 で追加.

バージョン 2.2 で変更: `__unicode__()` のサポートが追加されました.

vars ([*object*])

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

モジュールやインスタンスのようなオブジェクトには、更新可能な `__dict__` 属性があります。しかし、それ以外のオブジェクトでは `__dict__` 属性への書き込みに制限があるかもしれません。(例えば新スタイルクラスは、辞書を直接更新されることを防ぐために `dictproxy` を使っています。)

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

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

この関数は `range()` に非常によく似ていますが、リストの代わりに `XRange` 型を返します。このオブジェクトは不透明なシーケンス型で、対応するリストと同じ値を持ちますが、それらの値全てを同時に記憶しません。`range()` に対する `xrange()` の利点は微々たるものです (`xrange()` は要求に応じて値を生成するからです) ただし、メモリ量の厳しい計算機で巨大な範囲の値を使う時や、(ループがよく `break` で中断されるといったように) 範囲中の全ての値を使うとは限らない場合はその限りではありません。`xrange` オブジェクトについてのさらに詳しい情報については、`XRange` 型とシーケンス型 — `str`, `unicode`, `list`, `tuple`, `bytearray`, `buffer`, `xrange` を参照して下さい。

`xrange()` はシンプルさと速度のために定義されている関数であり、その実現のために実装上の制限を課している場合があります。Python の C 実装では、全ての引数をネイティブの C long 型 (Python の "short" 整数型) に制限しており、要素数がネイティブの C long 型の範囲内に収まるよう要求しています。もし大きな範囲が必要ならば、別の実装である `itertools` モジュールの、`islice(count(start, step), (stop-start+step-1+2*(step<0))//step)` を使うのが巧い方法かも知れません。

zip ([*iterable*, ...])

この関数はタプルのリストを返します。このリストの *i* 番目のタプルは各引数のシーケンスまたはイテレート可能オブジェクト中の *i* 番目の要素を含みます。返されるリストは引数のシーケンスのうち長さが最小のものの長さに切り詰められます。引数が全て同じ長さの際には、`zip()` は初期値引数が `None` の `map()` と似ています。引数が単一のシーケンスの場合、1 要素のタプルからなるリストを返します。引数を指定しない場合、空のリストを返します。

イテラブルの左から右への評価順序が保証されます。そのため `zip(*[iter(s)]*n)` を使ってデータ系列を *n* 長のグループにクラスタリングできます。

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

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

バージョン 2.0 で追加.

バージョン 2.4 で変更: これまでは、`zip()` は少なくとも一つの引数を要求しており、空のリストを返す代わりに `TypeError` を送出していました。

```
--import--(name[, globals[, locals[, fromlist[, level]]]])
```

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

この関数は `import` ステートメントにより呼び出されます。これは (`--builtin--` モジュールをインポートし、`--builtin--` `--import--` を割り当てることで) `import` ステートメントの意味を変更するための置き換えが可能です。今では、フックをインポートするほうが、大抵の場合簡単です ([PEP 302](#) を参照下さい)。 `--import--()` を直接使用することは稀で、例外は、実行時に名前が決定するモジュールをインポートするときです。

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

`level` は絶対、もしくは、相対のどちらのインポートを使うかを指定します。デフォルトは `-1` で絶対、相対インポートの両方を試みます。 `0` は絶対インポートのみ実行します。正の `level` の値は、`--import--()` を呼び出したディレクトリから検索対象となる親ディレクトリの階層を示します。

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

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

```
spam = __import__('spam', globals(), locals(), [], -1)
```

文 `import spam.ham` は、この呼び出しになります:

```
spam = __import__('spam.ham', globals(), locals(), [], -1)
```

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

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

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], -1)
eggs = _temp.eggs
saus = _temp.sausage
```

ここで、`--import--()` から `spam.ham` モジュールが返されます。このオブジェクトから、インポー

トされる名前が取り出され、それぞれの名前として代入されます。

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

バージョン 2.5 で変更: level パラメータが追加されました。

バージョン 2.5 で変更: Keyword サポートパラメータが追加されました。

第 3 章

非必須組み込み関数 (Non-essential Built-in Functions)

いくつかの組み込み関数は、現代的な Python プログラミングを行う場合には、必ずしも学習したり、知っていたり、使ったりする必要がなくなりました。こうした関数は古いバージョンの Python 向け書かれたプログラムとの互換性を維持するだけの目的で残されています。

Python のプログラマ、教官、学生、そして本の著者は、こうした関数を飛ばしてもかまわず、その際に何か重要なことを忘れていると思う必要もありません。

apply (*function*, *args*[, *keywords*])

引数 *function* は呼び出しができるオブジェクト (ユーザ定義および組み込みの関数またはメソッド、またはクラスオブジェクト) でなければなりません。 *args* はシーケンス型でなくてはなりません。 *function* は引数リスト *args* を使って呼び出されます; 引数の数はタプルの長さになります。オプションの引数 *keywords* を与える場合、 *keywords* は文字列のキーを持つ辞書でなければなりません。これは引数リストの最後に追加されるキーワード引数です。 *apply()* の呼び出しは、単なる *function(args)* の呼び出しとは異なります。というのは、 *apply()* の場合、引数は常に一つだからです。 *apply()* は *function(*args, **keywords)* を使うのと等価です。

バージョン 2.3 で非推奨: *apply(function, args, keywords)* ではなく *function(*args, **keywords)* を使ってください (tut-unpacking-arguments 参照)。

buffer (*object*[, *offset*[, *size*]])

引数 *object* は (文字列、アレイ、バッファといった) バッファ呼び出しインタフェースをサポートするオブジェクトでなければなりません。引数 *object* を参照する新たなバッファオブジェクトが生成されます。返されるバッファオブジェクトは *object* の先頭 (または *offset*) から のスライスになります。スライスの末端は *object* の末端まで (または引数 *size* で与えられた長さになるまで) です。

coerce (*x*, *y*)

二つの数値型の引数を共通の型に変換して、変換後の値からなるタプルを返します。変換に使われる規則は算術演算における規則と同じです。型変換が不可能である場合、 *TypeError* を送出します。

intern (*string*)

string を "隔離" された文字列のテーブルに入力し、隔離された文字列を返します – この文字列は *string* 自体がコピーです。隔離された文字列は辞書検索のパフォーマンスを少しだけ向上させるのに有効です

– 辞書中のキーが隔離されており、検索するキーが隔離されている場合、(ハッシュ化後の) キーの比較は文字列の比較ではなくポインタの比較で行うことができるからです。通常、Python プログラム内で利用されている名前は自動的に隔離され、モジュール、クラス、またはインスタンス属性を保持するための辞書は隔離されたキーを持っています。

バージョン 2.3 で変更: 隔離された文字列の有効期限は (Python 2.2 またはそれ以前は永続的でしたが) 永続的ではなくなりました; `intern()` の恩恵を受けるためには、`intern()` の返す値に対する参照を保持しなければなりません。

脚注

第 4 章

組み込み定数

組み込み名前空間にはいくつかの定数があります。定数の一覧:

False

`bool` 型における、偽を表す値です。

バージョン 2.3 で追加.

True

`bool` 型における、真を表す値です。

バージョン 2.3 で追加.

None

`types.NoneType` の唯一の値です。None は、例えば関数にデフォルト値が渡されないときのように、値がないことを表すためにしばしば用いられます。

バージョン 2.4 で変更: None に対する代入は不正であり、`SyntaxError` を送出します。

NotImplemented

”拡張比較 (rich comparison)” 特殊メソッド (`__eq__()`, `__lt__()`, と仲間たち) が返す特別な値で、相手の型に対する比較が実装されていないことを示します。

Ellipsis

拡張スライス文において用いられる特殊な値です。

__debug__

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

注釈: `None` と `__debug__` という名前は再代入できないので (これらに対する代入は、たとえ属性名としてであっても `SyntaxError` が送出されます)、これらは「真の」定数であると考えることができます。

バージョン 2.7 で変更: 属性名としての `__debug__` への代入が禁止されました。

4.1 site モジュールで追加される定数

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

quit (`[code=None]`)

exit (`[code=None]`)

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

copyright

credits

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

license

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

第 5 章

組み込み型

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

注釈: これまでの (リリース 2.2 までの) Python の歴史では、組み込み型はオブジェクト指向における継承を行う際に雛型にできないという点で、ユーザ定義型とは異なっていました。いまではこのような制限はなくなっています。

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

演算によっては、複数の型でサポートされているものがあります; 特に、ほぼ全てのオブジェクトについて、比較、真理値テスト、(`repr()` 関数や、わずかに異なる `str()` 関数による) 文字列への変換を行うことができます。オブジェクトが `print()` 関数で印字されるとき、文字列に変換する関数が暗黙に使われます。

5.1 真理値判定

どのオブジェクトも真理値を判定でき、`if` や `while` 条件に、または以下のブール演算の被演算子に使えます。以下の値は偽と見なされます:

- `None`
- `False`
- 数値型におけるゼロ。例えば `0`, `0L`, `0.0`, `0j`。
- 空のシーケンス。例えば `''`, `()`, `[]`。
- 空のマッピング。例えば `{}`。
- ユーザ定義クラスのインスタンスで、そのクラスが `__nonzero__()` または `__len__()` メソッドを定義していれば、それらのメソッドが整数 `0` または `bool` 値 `False` を返すとき。^{*1}

^{*1} これらの特殊なメソッドのさらなる情報は、Python リファレンスマニュアル (customization) を参照下さい。

それ以外の全ての値は真と見なされます — 従って、多くの型のオブジェクトは常に真です。

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

5.2 ブール演算 — `and`, `or`, `not`

以下にブール演算を、優先順位が低い順に示します:

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

注釈:

- (1) この演算子は短絡評価されます。つまり第一引数が偽のときにのみ、第二引数が評価されます。
- (2) この演算子は短絡評価されます。つまり第一引数が真のときにのみ、第二引数が評価されます。
- (3) `not` は非ブール演算子よりも優先度が低いので、`not a == b` は `not (a == b)` と解釈され、`a == not b` は構文エラーです。

5.3 比較

比較演算は全てのオブジェクトでサポートされています。比較演算子は全て同じ演算優先度を持っています(ブール演算より高い演算優先度です)。比較は任意の形で連鎖させることができます; 例えば、`x < y <= z` は `x < y and y <= z` と等価で、違うのは `y` が一度だけしか評価されないということです(どちらの場合でも、`x < y` が偽となった場合には `z` は評価されません)。

以下の表に比較演算をまとめます:

演算	意味	注釈
<code><</code>	より小さい	
<code><=</code>	以下	
<code>></code>	より大きい	
<code>>=</code>	以上	
<code>==</code>	等しい	
<code>!=</code>	等しくない	(1)
<code>is</code>	同一のオブジェクトである	
<code>is not</code>	同一のオブジェクトでない	

注釈:

- (1) `!=` は `<>` とも書けますがこれは時代遅れの書き方で、後方互換性維持のためだけに残されています。新しいコードでは常に `!=` を使うべきです。

数値型間の比較が文字列間の比較でないかぎり、異なる型のオブジェクトを比較しても等価になることはありません; これらのオブジェクトの順番付けは一貫してはいますが任意のものです (従って要素の型が一樣でないシーケンスをソートした結果は一貫したものになります)。さらに、(例えばファイルオブジェクトのように) 型によっては、その型の 2 つのオブジェクトの不等性だけの、縮退した比較の概念しかサポートしないものもあります。繰り返しますが、そのようなオブジェクトも任意の順番付けをされていますが、それは一貫したものです。被演算子が複素数の場合、演算子 `<`, `<=`, `>` および `>=` は例外 `TypeError` を送出します。

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

クラスのインスタンスは、そのクラスが十分なだけの拡張比較メソッド (`__lt__()`, `__le__()`, `__gt__()`, `__ge__()`) または `__cmp__()` メソッドを定義していない限り、同じクラスの別のインスタンスや他の型のオブジェクトとは順序付けできません。

数値型を除き、異なる型のオブジェクトは型の名前で順番付けされます; 適当な比較をサポートしていないある型のオブジェクトはアドレスによって順番付けされます。

同じ優先度を持つ演算子としてさらに 2 つ、シーケンス型でのみ `in` および `not in` がサポートされています (以下を参照)。

5.4 数値型 `int`, `float`, `long`, `complex`

4 つの異なる数値型があります: 通常の整数型, 長整数型, 浮動小数点型, 複素数型 です。さらに、真偽値 (Boolean) 型も通常の整数型のサブタイプです。通常の整数 (単に 整数型 と呼ばれます) は C 言語の `long` 型を使って実装されており、少なくとも 32 ビットの精度があります (`sys.maxint` は常に通常の整数の各プラットフォームにおける最大値にセットされており、最小値は `-sys.maxint - 1` になります)。長整数型には精度の制限がありません。浮動小数点型はたいていは C の `double` を使って実装されています; あなたのプログラムが動作するマシンでの浮動小数点型の精度と内部表現は、`sys.float_info` から利用できます。複素数型は実部と虚部を持ち、それぞれ浮動小数点数です。複素数 `z` から実部および虚部を取り出すには、`z.real` および `z.imag` を使ってください。(標準ライブラリには、追加の数値型、分数を保持する `fractions` や、ユーザ定義の精度の浮動小数点数を保持する `decimal` があります。)

数値は、数値リテラルや組み込み関数や演算子の戻り値として生成されます。修飾のない整数リテラル (2 進表現や、16 進表現や 8 進表現の値も含みます) は、通常の整数値を表します。値が通常の整数で表すには大きすぎる場合、`'L'` または `'l'` が末尾につく整数リテラルは長整数型を表します (`'L'` が望ましいです。というのは `11` は `11` と非常に紛らわしいからです!)。小数点または指数表記のある数値リテラルは浮動小数点数を表します。数値リテラルに `'j'` または `'J'` をつけると虚数 (実数部がゼロの複素数) を表し、それと整数や浮動小数点数を足すと実部と虚部を持つ複素数が得られます。

Python は型混合の演算を完全にサポートします: ある 2 項演算子が互いに異なる数値型の被演算子を持つ場合、より "制限された" 型の被演算子是他方の型に合わせて広げられます。ここで通常の整数は長整数より制限されており、長整数は浮動小数点数より制限されており、浮動小数点は複素数より制限されていま

す。型混合の数値間での比較も同じ規則に従います。^{*2} コンストラクタ `int()`, `long()`, `float()`, および `complex()` を使って、特定の型の数を生成することができます。

全ての組み込み数値型は以下の演算をサポートします。演算子の優先度については、`power`, および、あとのセクションを参照下さい。

演算	結果	注釈
<code>x + y</code>	x と y の和	
<code>x - y</code>	x と y の差	
<code>x * y</code>	x と y の積	
<code>x / y</code>	x と y の商	(1)
<code>x // y</code>	x と y の商 (を切り下げたもの)	(4)(5)
<code>x % y</code>	x / y の剰余	(4)
<code>-x</code>	x の符号反転	
<code>+x</code>	x そのまま	
<code>abs(x)</code>	x の絶対値または大きさ	(3)
<code>int(x)</code>	x の整数への変換	(2)
<code>long(x)</code>	x の長整数への変換	(2)
<code>float(x)</code>	x の浮動小数点数への変換	(6)
<code>complex(re, im)</code>	実部 re , 虚部 im の複素数。 im の既定値はゼロ。	
<code>c.conjugate()</code>	複素数 c の共役複素数 (実数に対しては同じ値を返す)	
<code>divmod(x, y)</code>	$(x // y, x \% y)$ からなるペア	(3)(4)
<code>pow(x, y)</code>	x の y 乗	(3)(7)
<code>x ** y</code>	x の y 乗	(7)

注釈:

- (1) (通常および長) 整数の割り算では、結果は整数になります。この場合値は常にマイナス無限大の方向に丸められます: つまり、 $1/2$ は 0、 $(-1)/2$ は -1、 $1/(-1)$ は -1、そして $(-1)/(-2)$ は 0 になります。被演算子の両方が長整数の場合、計算値に関わらず結果は長整数で返されるので注意してください。
- (2) 浮動小数点数から `int()`, または、`long()` を使った変換では、関連する関数、`math.trunc()` のようにゼロ方向へ丸められます。下方向への丸めには `math.floor()` を使い、上方向への丸めには `math.ceil()` を使って下さい。
- (3) 完全な記述については、[組み込み関数](#), を参照してください。
- (4) バージョン 2.3 で非推奨: 切り捨て除算演算子、モジュロ演算子、および `divmod()` 関数は、複素数に対してはもはや定義されていません。目的に合うならば、代わりに `abs()` を使って浮動小数点に変換してください。
- (5) 整数の除算とも呼ばれます。結果の型は整数型とは限りませんが、結果の値は整数です。
- (6) 浮動小数点数は、文字列 "nan" と "inf" を、必要なら接頭辞 "+" または "-" と共に、非数 (Not a Number (NaN)) や正、負の無限大として受け付けます。

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

バージョン 2.6 で追加.

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

全ての `numbers.Real` 型 (`int`, `long`, および、`float`) は以下の演算を含みます。:

演算	結果
<code>math.trunc(x)</code>	<code>x</code> を <i>Integral</i> (整数) に切り捨てます
<code>round(x, n)</code>	<code>x</code> を <code>n</code> 桁に丸めます。丸め方は四捨五入です。 <code>n</code> が省略されれば 0 がデフォルトとなります。(–訳注: Python 3 と仕様が違うので注意してください。1.5 と 2.5 の <code>round</code> は、Python 2 ではそれぞれ 2、3 となり、Python 3 では 2、2 となります。後者の丸めは .5 を丸める際に偶数になる方へ丸めるので、「偶数丸め」と言います。–)
<code>math.floor(x)</code>	<code>x</code> 以下の最大の整数を浮動小数点数として返します
<code>math.ceil(x)</code>	<code>x</code> 以上の最小の整数を浮動小数点数として返します

5.4.1 整数型におけるビット単位演算

ビット単位演算は、整数に対してのみ意味があります。負の数は、その 2 の補数の値として扱われます (演算中にオーバーフローが起こらないように十分なビット数があるものと仮定します)。

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

以下の表では、ビット単位演算を優先順位が低い順に並べています:

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

注釈:

- (1) 負値のシフト数は不正であり、`ValueError` が送出されます。
- (2) `n` ビットの左シフトは、`pow(2, n)` による乗算と等価です。通常整数の範囲を超える結果になる場合は長整数で返されます。
- (3) `n` ビットの右シフトは、`pow(2, n)` による除算と等価です。

5.4.2 整数型における追加のメソッド

整数型は `numbers.Integral abstract base class` を実装します。さらに、追加のメソッドを一つ提供します。

`int.bit_length()`

`long.bit_length()`

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

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

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

次と等価です:

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

バージョン 2.7 で追加.

5.4.3 浮動小数点数に対する追加のメソッド

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

`float.as_integer_ratio()`

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

バージョン 2.6 で追加.

`float.is_integer()`

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

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

バージョン 2.6 で追加.

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

`float.hex()`

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

バージョン 2.6 で追加.

`float.fromhex(s)`

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

バージョン 2.6 で追加.

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

16 進文字列表現は以下の書式となります:

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

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

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

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

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

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

5.5 イテレータ型

バージョン 2.2 で追加.

Python はコンテナでの反復処理の概念をサポートしています。この概念は 2 つの別々のメソッドを使って実

装されています; これらのメソッドを使ってユーザ定義のクラスで反復を行えるようにできます。後に詳しく述べるシーケンスは、必ず反復処理メソッドをサポートしています。

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

```
container.__iter__()
```

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

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

```
iterator.__iter__()
```

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

```
iterator.next()
```

コンテナの次のアイテムを返します。もしそれ以上アイテム無ければ `StopIteration` 例外を送出します。このメソッドは Python/C API での Python オブジェクトの型構造体の `tp_iternext` スロットに対応します。

Python では、いくつかのイテレータオブジェクトを定義しています。これらは一般のおよび特殊なシーケンス型、辞書型、そして他のさらに特殊化された形式に渡る反復をサポートします。特殊型は、イテレータプロトコルの実装以外は重要ではありません。

このプロトコルでは、イテレータの `next()` メソッドが一旦 `StopIteration` を送出したなら、以降の呼び出しでも例外を送出し続けることを期待します。この特性に従わない実装は壊れているとみなされます。(この制約は Python 2.3 で追加されました。Python 2.2 では色々なイテレータが、このルールによれば壊れています。)

5.5.1 ジェネレータ型

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

5.6 シーケンス型 — str, unicode, list, tuple, bytearray, buffer, xrange

シーケンス型には 7 つあります: 文字列、Unicode 文字列、リスト、タプル、バイト配列 (bytearray)、バッファ、そして xrange オブジェクトです。

他のコンテナ型については、組み込みクラスの `dict` および `set` を参照下さい。

文字列リテラルは `'xyzzy'`, `"frobozz"` といったように、単引用符または二重引用符の中に書かれます。文字列リテラルについての詳細は、`strings` を参照下さい。Unicode 文字列はほとんど文字列と同じですが、`u'abc'`, `u"def"` といったように先頭に文字 `'u'` を付けて指定します。リストは `[a, b, c]` のように要素をコンマで区切り角括弧で囲って生成します。タプルは `a, b, c` のようにコンマ演算子で区切って生成します (角括弧の中には入れません)。丸括弧で囲っても囲わなくてもかまいませんが、空のタプルは `()` のように丸括弧で囲わなければなりません。要素が一つのタプルでは、例えば `(d,)` のように、要素の後ろにコンマをつければなりません。

バイト配列は、組み込み関数 `bytearray()` で構成されます。

バッファオブジェクトは Python の構文上では直接サポートされていませんが、組み込み関数 `buffer()` で生成することができます。バッファオブジェクトは結合や反復をサポートしていません。

xrange オブジェクトは、オブジェクトを生成するための特殊な構文がない点でバッファに似ていて、関数 `xrange()` で生成します。xrange オブジェクトはスライス、結合、反復をサポートせず、`in`, `not in`, `min()` または `max()` は効率的ではありません。

ほとんどのシーケンス型は以下の演算操作をサポートします。`in` および `not in` は比較演算とおなじ優先度を持っています。`+` および `*` は対応する数値演算とおなじ優先度です。^{*3} **ミュータブルなシーケンス型** で追加のメソッドが提供されています。

以下のテーブルはシーケンス型の演算を優先度の低いものから順に挙げたものです。テーブル内の *s* および *t* は同じ型のシーケンスです; *n*, *i* および *j* は整数です:

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

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

シーケンス型は比較演算子もサポートします。特にタプルとリストは相当する要素による辞書編集方式的に比較されます。つまり、等しいということは、ふたつのシーケンスの長さ、型が同じであり、全ての要素が等しいということです (詳細は言語リファレンスの `comparisons` を参照下さい)。

注釈:

- (1) s が文字列または Unicode 文字列の場合、演算操作 `in` および `not in` は部分文字列の一致テストと同じように動作します。バージョン 2.3 以前の Python では、 x は長さ 1 の文字列である必要がありました。Python 2.3 以降では、 x はどんな長さでも構いません。
- (2) n が 0 以下の値の場合、0 として扱われます (これは s と同じ型の空のシーケンスを表します)。シーケンス s の要素はコピーされないのご注意ください; コピーではなく要素に対する参照カウントが増えます。これは Python に慣れていないプログラマをよく悩ませます。例えば以下のコードを考えます:

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

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

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

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

- (3) i または j が負の数の場合、インデックスはシーケンスの末端からの相対インデックスになります: `len(s) + i` または `len(s) + j` が代わりに使われます。ただし `-0` はやはり 0 であることに注意してください。
- (4) s の i から j へのスライスは $i \leq k < j$ となるようなインデックス k を持つ要素からなるシーケンスとして定義されます。 i または j が `len(s)` よりも大きい場合、`len(s)` を使います。 i が省略されるか `None` だった場合、0 を使います。 j が省略されるか `None` だった場合、`len(s)` を使います。 i が j 以上の場合、スライスは空のシーケンスになります。
- (5) s の「 i から j まででステップが k のスライス」は、インデックス $x = i + n*k$ (ただし n は $0 \leq n < (j-i)/k$ を満たす任意の整数) を持つ要素からなるシーケンスとして定義されます。言い換えるとインデックスは $i, i+k, i+2*k, i+3*k$ と続き、 j に達したところでストップします (ただし j は含みません)。 k が正の数である場合、 i または j が `len(s)` より大きければ `len(s)` を代わりに使用します。 k が負の数である場合、 i または j が `len(s) - 1` より大きければ `len(s) - 1` を代わりに使用します。 i または j を省略または `None` を指定すると、“端” (どちらの端かは k の符号に依存) の

値を代わりに使用します。なお k はゼロにできないので注意してください。また k に `None` を指定すると、`1` が指定されたものとして扱われます。

- (6) s と t の両者が文字列であるとき、CPython のようないくつかの Python 実装では、`s = s + t` や `s += t` という書式で代入をするのに in-place optimization が働きます。このような時、最適化は二乗の実行時間の低減をもたらします。この最適化はバージョンや実装に依存します。実行効率が必要なコードでは、バージョンと実装が変わっても、連結の線形的実行効率を保証する `str.join()` を使うのがより望ましいでしょう。

バージョン 2.4 で変更: 以前、文字列の連結は in-place で再帰されませんでした。

5.6.1 文字列メソッド

8-bit 文字列と Unicode オブジェクトは、どちらも以下に挙げるメソッドに対応しています。この中には、`bytearray` オブジェクトで使えるものもあります。

さらに、Python の文字列は [シーケンス型](#) — `str`, `unicode`, `list`, `tuple`, `bytearray`, `buffer`, `xrange` に記載されるシーケンス型のメソッドもサポートします。書式指定して文字列を出力するためには、テンプレート文字列を使うか、[文字列フォーマット操作](#) に記載される `%` 演算子を使います。正規表現に基づく文字列操作関数については、`re` モジュールを参照下さい。

`str.capitalize()`

最初の文字を大文字にし、残りを小文字にした文字列のコピーを返します。

8 ビット文字列では、メソッドはロケール依存になります。

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

`width` の長さをもつ中央寄せされた文字列を返します。パディングには `fillchar` で指定された値 (デフォルトではスペース) が使われます。

バージョン 2.4 で変更: 引数 `fillchar` に対応。

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

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

`str.decode([encoding[, errors]])`

codec に登録された文字コード系 `encoding` を使って文字列をデコードします。`encoding` は標準でデフォルトの文字列エンコーディングになります。標準とは異なるエラー処理を行うために `errors` を与えることができます。標準のエラー処理は `'strict'` で、エンコードに関するエラーは `UnicodeError` を送出します。他に利用できる値は `'ignore'`, `'replace'` および関数 `codecs.register_error()` によって登録された名前です。これについてはセクション [Codec 基底クラス](#) 節を参照してください。

バージョン 2.2 で追加。

バージョン 2.3 で変更: その他のエラーハンドリングスキーマがサポートされました。

バージョン 2.7 で変更: キーワード引数のサポートが追加されました。

`str.encode([encoding[, errors]])`

文字列のエンコードされたバージョンを返します。デフォルトのエンコーディングは現在の文字列エンコーディングのデフォルトです。標準とは異なるエラー処理を行うために *errors* を与えることができます。標準のエラー処理は 'strict' で、エンコードに関するエラーは *UnicodeError* を送出します。他に利用できる値は 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' および関数 *codecs.register_error()* によって登録された名前です。これについてはセクション *Codec 基底クラス* を参照してください。利用可能なエンコーディングの一覧は、セクション *標準エンコーディング* を参照してください。

バージョン 2.0 で追加。

バージョン 2.3 で変更: 'xmlcharrefreplace', 'backslashreplace' およびその他のエラーハンドリングスキーマがサポートされました。

バージョン 2.7 で変更: キーワード引数のサポートが追加されました。

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

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

バージョン 2.5 で変更: *suffix* でタプルを受け付けるようになりました。

`str.expandtabs([tabsize])`

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

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

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

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

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

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

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

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

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

書式指定のオプションについては、書式指定文字列を規定する [書式指定文字列の文法](#) を参照下さい。

この文字列書式指定のメソッドは Python 3 での新しい標準であり、新しいコードでは、[文字列フォーマット操作](#) で規定される `%` を使った書式指定より好ましい書き方です。

バージョン 2.6 で追加。

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

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

`str.isalnum()`

文字列中の全ての文字が英数文字で、かつ 1 文字以上ある場合には真を返し、そうでない場合は偽を返します。

8 ビット文字列では、メソッドはロケール依存になります。

`str.isalpha()`

文字列中の全ての文字が英文字で、かつ 1 文字以上ある場合には真を返し、そうでない場合は偽を返します。

8 ビット文字列では、メソッドはロケール依存になります。

`str.isdigit()`

文字列中に数字しかない場合には真を返し、その他の場合は偽を返します。

8 ビット文字列では、メソッドはロケール依存になります。

`str.islower()`

文字列中の大小文字の区別のある文字^{*4} 全てが小文字で、かつ大小文字の区別のある文字が 1 文字以上あるなら真を、そうでなければ偽を返します。

8 ビット文字列では、メソッドはロケール依存になります。

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

`str.isspace()`

文字列が空白文字だけからなり、かつ 1 文字以上ある場合には真を返し、そうでない場合は偽を返します。

8 ビット文字列では、メソッドはロケール依存になります。

`str.istitle()`

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

8 ビット文字列では、メソッドはロケール依存になります。

`str.isupper()`

文字列中の大小文字の区別のある文字^{*4} 全てが大文字で、かつ大小文字の区別のある文字が 1 文字以上あるなら真を、そうでなければ偽を返します。

8 ビット文字列では、メソッドはロケール依存になります。

`str.join(iterable)`

イテラブル中の文字列を結合した文字列オブジェクトを返します。*iterable* に Unicode オブジェクトがある場合は、代わりに Unicode を返します。文字列でないあるいは Unicode オブジェクトでない値が存在する場合は、`TypeError` が送出されます。要素間の区切り文字列は、このメソッドを提供する文字列です。

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

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

バージョン 2.4 で変更: 引数 *fillchar* に対応。

`str.lower()`

全ての大小文字の区別のある文字^{*4} が小文字に変換された、文字列のコピーを返します。

8 ビット文字列では、メソッドはロケール依存になります。

`str.lstrip([chars])`

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

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

バージョン 2.2.2 で変更: 引数 *chars* をサポートしました。

`str.partition(sep)`

文字列を *sep* の最初の出現位置で区切り、3 要素のタプルを返します。タプルの内容は、区切りの前の

部分、区切り文字列そのもの、そして区切りの後ろの部分です。もし区切れなければ、タプルには元の文字列そのものとその後ろに二つの空文字列が入ります。

バージョン 2.5 で追加.

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

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

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

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

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

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

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

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

バージョン 2.4 で変更: 引数 *fillchar* に対応.

`str.rpartition(sep)`

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

バージョン 2.5 で追加.

`str.rsplit([sep[, maxsplit]])`

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

バージョン 2.4 で追加.

`str.rstrip([chars])`

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

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

バージョン 2.2.2 で変更: 引数 *chars* をサポートしました.

`str.split([sep[, maxsplit]])`

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

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

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

例えば、`' 1 2 3 '.split()` は `['1', '2', '3']` を返し、`' 1 2 3 '.split(None, 1)` は `['1', '2 3 ']` を返します。

`str.splitlines([keepends])`

文字列を改行部分で分解し、各行からなるリストを返します。このメソッドは、行の分割に *universal newlines* アプローチを使います。*keepends* に真が与えない限り、返されるリストに改行は含まれません。

Python は `"\r"`, `"\n"`, `"\r\n"` を 8-bit 文字列の行の境界として認識します。

例えば:

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

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

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

比較のために `split('\n')` は以下のようになります:

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

`unicode.splitlines([keepends])`

str.splitlines() のように、文字列を行のリストにして返します。ただし Unicode のメソッドで

は、8-bit 文字列の行の境界として認識される *universal newlines* よりも多くの次に挙げる行の境界で分割します。

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

バージョン 2.7 で変更: `\v` と `\f` が行境界のリストに追加されました。

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

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

バージョン 2.5 で変更: *prefix* でタプルを受け付けるようになりました。

`str.strip([chars])`

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

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

バージョン 2.2.2 で変更: 引数 *chars* をサポートしました。

`str.swapcase()`

文字列をコピーし、大文字は小文字に、小文字は大文字に変換して返します。

8 ビット文字列では、メソッドはロケール依存になります。

`str.title()`

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

このアルゴリズムは、連続した文字の集まりという、言語から独立した単純な単語の定義を使います。

この定義は多くの状況ではうまく機能しますが、短縮形や所有格のアポストロフィが単語の境界になってしまい、望みの結果を得られない場合があります:

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

正規表現を使うことでアポストロフィに対応できます:

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

8 ビット文字列では、メソッドはロケール依存になります。

`str.translate(table[, deletechars])`

文字列をコピーし、オプション引数の文字列 *deletechars* の中に含まれる文字を全て除去します。その後、残った文字を変換テーブル *table* に従ってマップして返します。変換テーブルは長さ 256 の文字列でなければなりません。

トランスレーションテーブル作成のために、*string* モジュールの *maketrans()* 補助関数を使うこともできます。文字列型オブジェクトに対しては、*table* 引数に *None* を与えることで、文字の削除だけを実施します。:

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

バージョン 2.6 で追加: *None* の *table* 引数をサポートしました。

Unicode オブジェクトの場合、*translate()* メソッドはオプションの *deletechars* 引数を受理しません。その代わりに、メソッドはすべての文字が与えられた変換テーブルで対応付けされている *s* のコピーを返します。この変換テーブルは Unicode 順 (ordinal) から Unicode 順、Unicode 文字列、または *None* への対応付けでなくてはなりません。対応付けされていない文字は何もせず放置されます。 *None* に対応付けられた文字は削除されます。ちなみに、より柔軟性のあるアプローチは、自作の文字対応付けを行う codec を *codecs* モジュールを使って作成することです (例えば *encodings.cp1251* を参照してください。)

`str.upper()`

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

8 ビット文字列では、メソッドはロケール依存になります。

`str.zfill(width)`

数値文字列の左側をゼロ詰めし、幅 *width* の文字列で返します。符号接頭辞も正しく扱われます。 *width* が `len(s)` 以下の場合もとの文字列自体が返されます。

バージョン 2.2.2 で追加。

以下のメソッドは、Unicode オブジェクトにのみ実装されます：

`unicode.isnumeric()`

数字を表す文字のみで構成される場合、`True` を返します。それ以外の場合は `False` を返します。数字を表す文字には、0 から 9 までの数字と、Unicode の数字プロパティを持つ全ての文字が含まれます。(e.g. U+2155, VULGAR FRACTION ONE FIFTH)

`unicode.isdecimal()`

10 進数文字のみで構成される場合、`True` を返します。それ以外の場合は、`False` を返します。10 進数文字には 0 から 9 までの数字と、10 進基数表記に使われる全ての文字が含まれます。(e.g. U+0660, ARABIC-INDIC DIGIT ZERO)

5.6.2 文字列フォーマット操作

文字列および Unicode オブジェクトには固有の操作: `%` 演算子 (モジュロ) があります。この演算子は文字列フォーマット化 または 補間 演算としても知られています。 `format % values` (*format* は文字列または Unicode オブジェクト) とすると、*format* 中の `%` 変換指定は *values* 中のゼロ個またはそれ以上の要素で置換されます。この動作は C 言語における `sprintf()` に似ています。 *format* が Unicode オブジェクトであるか、または `%s` 変換を使って Unicode オブジェクトが変換される場合、その結果も Unicode オブジェクトになります。

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

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

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

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

6. 精度長変換子 (オプション)。

7. 変換型。

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

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

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

変換フラグ文字を以下に示します:

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

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

変換型を以下に示します:

変換	意味	注釈
'd'	符号付き 10 進整数。	
'i'	符号付き 10 進整数。	
'o'	符号付き 8 進数。	(1)
'u'	旧式の型 – 'd' と同じです。	(7)
'x'	符号付き 16 進数 (小文字)。	(2)
'X'	符号付き 16 進数 (大文字)。	(2)
'e'	指数表記の浮動小数点数 (小文字)。	(3)
'E'	指数表記の浮動小数点数 (大文字)。	(3)
'f'	10 進浮動小数点数。	(3)
'F'	10 進浮動小数点数。	(3)
'g'	浮動小数点数。指数部が -4 以上または精度以下の場合には小文字指数表記、それ以外の場合には 10 進表記。	(4)
'G'	浮動小数点数。指数部が -4 以上または精度以下の場合には大文字指数表記、それ以外の場合には 10 進表記。	(4)
'c'	文字一文字 (整数または一文字からなる文字列を受理します)。	
'r'	文字列 (Python オブジェクトを <i>repr()</i> で変換します)。	(5)
's'	文字列 (Python オブジェクトを <i>str()</i> で変換します)。	(6)
'%'	引数を変換せず、返される文字列中では文字 '%' になります。	

注釈:

(1) この形式の出力にした場合、変換結果の先頭の数字がゼロ ('0') でないときには、数字の先頭と左側のパディングとの間にゼロを挿入します。

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

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

指定精度は小数点の後の桁数を決定し、そのデフォルトは 6 です。

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

指定精度は小数点の前後の有効桁数を決定し、そのデフォルトは 6 です。

(5) %r 変換は Python 2.0 で追加されました。

指定精度は最大文字数を決定します。

(6) オブジェクトや与えられた書式が *unicode* 文字列の場合、変換後の文字列も *unicode* になります。

指定精度は最大文字数を決定します。

(7) [PEP 237](#) を参照してください。

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

バージョン 2.7 で変更: 絶対値が `1e50` を超える数値の `%f` 変換が `%g` 変換に置き換えられなくなりました。

その他の文字列操作は標準モジュール `string` および `re` で定義されています。

5.6.3 xrange 型

`xrange` 型は値の変更不能なシーケンスで、広範なループ処理に使われています。 `xrange` 型の利点は、`xrange` オブジェクトは表現する値域の大きさにかかわらず常に同じ量のメモリしか占めないということです。はっきりしたパフォーマンス上の利点はありません。

`XRange` オブジェクトは非常に限られた振る舞い、すなわち、インデックス検索、反復、`len()` 関数のみをサポートしています。

5.6.4 ミュータブルなシーケンス型

リストとバイト配列 (`bytearray`) オブジェクトは、オブジェクトをインプレースに変更できるようにする追加の操作をサポートします。他のミュータブルなシーケンス型 (を言語に追加するとき) も、それらの操作をサポートするべきです。文字列およびタプルはイミュータブルなシーケンス型です: これらのオブジェクトは一度生成されたら変更できません。ミュータブルなシーケンス型では以下の操作が定義されています (ここで `x` は任意のオブジェクトとします)。

演算	結果	注釈
<code>s[i] = x</code>	<code>s</code> の要素 <code>i</code> を <code>x</code> と入れ替えます	
<code>s[i:j] = t</code>	<code>s</code> の <code>i</code> から <code>j</code> 番目までのスライスをイテラブル <code>t</code> の内容に入れ替えます	
<code>del s[i:j]</code>	<code>s[i:j] = []</code> と同じです	
<code>s[i:j:k] = t</code>	<code>s[i:j:k]</code> の要素を <code>t</code> の要素と入れ替えます	(1)
<code>del s[i:j:k]</code>	リストから <code>s[i:j:k]</code> の要素を削除します	
<code>s.append(x)</code>	<code>s[len(s):len(s)] = [x]</code> と同じです	(2)
<code>s.extend(t)</code> または <code>s += t</code>	<code>s[len(s):len(s)] = t</code> とほとんど同じです	(3)
<code>s *= n</code>	<code>s</code> をその内容を <code>n</code> 回繰り返したもので更新	(11)
<code>s.count(x)</code>	<code>s[i] == x</code> となる <code>i</code> の個数を返します	
<code>s.index(x[, i[, j]])</code>	<code>s[k] == x</code> かつ <code>i <= k < j</code> となる最小の <code>k</code> を返します	(4)
<code>s.insert(i, x)</code>	<code>s[i:i] = [x]</code> と同じです	(5)
<code>s.pop([i])</code>	<code>x = s[i]; del s[i]; return x</code> と同じです	(6)
<code>s.remove(x)</code>	<code>del s[s.index(x)]</code> と同じです	(4)
<code>s.reverse()</code>	<code>s</code> をインプレースに逆転させます	(7)
<code>s.sort([cmp[, key[, reverse]])</code>	<code>s</code> の要素をインプレースに並べ替えます	(7)(8)(9)(10)

注釈:

- (1) *i* は置き換えるスライスと同じ長さでなければいけません。
- (2) かつての Python の C 実装では、複数パラメタを受理し、暗黙にそれらをタプルに結合していました。この間違った機能は Python 1.4 で撤廃され、Python 2.0 の導入とともにエラーにするようになりました。
- (3) *x* は任意のイテラブルオブジェクトにできます。
- (4) *x* が *s* 中に見つからなかった場合 `ValueError` を送出します。負のインデックスが二番目または三番目のパラメタとして `index()` メソッドに渡されると、これらの値にはスライスのインデックスと同様にリストの長さが加算されます。加算後もまだ負の場合、その値はスライスのインデックスと同様にゼロに切り詰められます。

バージョン 2.3 で変更: 以前は `index()` には開始と終了位置を指定する引数がありませんでした。

- (5) `insert()` の最初のパラメタとして負のインデックスが渡された場合、スライスのインデックスと同じく、リストの長さが加算されます。それでも負の値を取る場合、スライスのインデックスと同じく、0 に丸められます。

バージョン 2.3 で変更: 以前は、すべての負値は 0 に丸められていました。

- (6) `pop()` メソッドのオプションの引数 *i* は標準で -1 なので、標準では最後の要素をリストから除去して返します。
- (7) `sort()` および `reverse()` メソッドは大きなリストを並べ替えたり反転したりする際、容量の節約のためにリストを直接変更します。副作用があることをユーザに思い出させるために、これらの操作は並べ替えまたは反転されたリストを返しません。
- (8) `sort()` メソッドは、比較を制御するためにオプションの引数をとります。

`cmp` は 2 つの引数 (リストの要素) からなるカスタムの比較関数を指定します。これは始めの引数が 2 つ目の引数に比べて小さい、等しい、大きいかに応じて負数、ゼロ、正数を返します。 `cmp=lambda x,y: cmp(x.lower(), y.lower())`。デフォルト値は `None` です。

`key` は 1 つの引数からなる関数を指定します。これはリストの各要素から比較のキーを取り出すのに使われます: `key=str.lower`。デフォルト値は `None` です。

`reverse` はブール値です。True に設定された場合、リストの要素は各比較が反転したように並び替えられます。

一般的に、`key` および `reverse` の変換プロセスは同等の `cmp` 関数を指定するより早く動作します。これは `key` および `reverse` がそれぞれの要素に一度だけ触れる間に、`cmp` はリストのそれぞれの要素に対して複数回呼ばれることによるものです。旧式の `cmp` 関数を `key` 関数に変換するには `functools.cmp_to_key()` を使用してください。

バージョン 2.3 で変更: `None` を渡すのと、`cmp` を省略した場合とで、同等に扱うサポートを追加。

バージョン 2.4 で変更: `key` および `reverse` のサポートを追加。

- (9) Python 2.3 より、`sort()` メソッドは安定していることが保証されています。ソートは、等しい要素の相対順序が変更されないことが保証されていれば、安定しています — これは複合的なパスでソートを行なう（例えば部署でソートして、それから給与の等級でソートする）のに役立ちます。
- (10) リストがソートされている間、または変更しようとする試みの影響中、あるいは検査中でさえ、リストは未定義です。Python 2.3 とそれ以降の C 実装では、それらが続いている間、リストは空として出力され、リストがソート中に変更されていることを検知できたら `ValueError` を送出します。
- (11) 値 `n` は整数または `__index__()` を実装したオブジェクトです。`n` にゼロや負数を与えると、シーケンスをクリアします。シーケンス内の要素はコピーされません; コピーではなく要素に対する参照カウントが増えます。`s * n` について [シーケンス型 — `str`, `unicode`, `list`, `tuple`, `bytearray`, `buffer`, `xrange`](#) で説明したとおりです。

5.7 set (集合) 型 — `set`, `frozenset`

`set` オブジェクトは順序付けされていない [hashable](#) (ハッシュ可能な) オブジェクトのコレクションです。よくある使い方には、メンバーシップのテスト、数列から重複を削除する、そして論理積、論理和、差集合、対称差など数学的演算の計算が含まれます。(他のコンテナ型については、組み込みクラスの [dict](#), [list](#), [tuple](#), および、モジュール [collections](#) を参照下さい)

バージョン 2.4 で追加.

集合は、他のコレクションと同様、`x in set`, `len(set)`, `for x in set` をサポートします。コレクションには順序がないので、集合は挿入の順序や要素の位置を記録しません。従って、集合はインデクシング、スライシング、その他のシーケンス的な振舞いをサポートしません。

`set` および `frozenset` という、2 つの組み込みの集合型があります。`set` はミュータブルで、`add()` や `remove()` のようなメソッドを使って内容を変更できます。ミュータブルなため、ハッシュ値を持たず、また辞書のキーや他の集合の要素として用いることができません。一方、`frozenset` 型はイミュータブルで、[ハッシュ可能](#) です。作成後に内容を改変できないため、辞書のキーや他の集合の要素として用いることができます。

Python 2.7 では、空でない `set` (`frozenset` ではない) は、`set` コンストラクタに加え、要素を波カッコ中にカンマで区切って列挙することでも生成できます。例: `{'jack', 'sjoerd'}`。

どちらのクラスのコンストラクタも同様に働きます:

```
class set ([iterable])
class frozenset ([iterable])
```

`iterable` から要素を取り込んだ、新しい `set` もしくは `frozenset` オブジェクトを返します。集合の要素は [ハッシュ可能](#) なものでなくてはなりません。集合の集合を表現するためには、内側の集合は `frozenset` オブジェクトでなくてはなりません。`iterable` が指定されない場合、新しい空の集合が返されます。

`set` および `frozenset` のインスタンスは以下の操作を提供します:

`len(s)`

集合 `s` の要素数 (`s` の濃度) を返します。

x in s

x が *s* のメンバーに含まれるか判定します。

x not in s

x が *s* のメンバーに含まれていないことを判定します。

isdisjoint (*other*)

集合が *other* と共通の要素を持たないとき、`True` を返します。集合はそれらの積集合が空集合となるときのみ、互いに素 (disjoint) となります。

バージョン 2.6 で追加.

issubset (*other*)

set <= other

set の全ての要素が *other* に含まれるか判定します。

set < other

set が *other* の真部分集合であるかを判定します。つまり、`set <= other and set != other` と等価です。

issuperset (*other*)

set >= other

other の全ての要素が *set* に含まれるか判定します。

set > other

set が *other* の真上位集合であるかを判定します。つまり、`set >= other and set != other` と等価です。

union (**others*)

set | other | ...

set と全ての *other* の要素からなる新しい集合を返します。

バージョン 2.6 で変更: 複数のイテラブルからの入力を受け入れるようになりました。

intersection (**others*)

set & other & ...

set と全ての *other* に共通する要素を持つ、新しい集合を返します。

バージョン 2.6 で変更: 複数のイテラブルからの入力を受け入れるようになりました。

difference (**others*)

set - other - ...

set に含まれて、かつ、全ての *other* に含まれない要素を持つ、新しい集合を返します。

バージョン 2.6 で変更: 複数のイテラブルからの入力を受け入れるようになりました。

symmetric_difference (*other*)

set ^ other

set と *other* のいずれか一方だけに含まれる要素を持つ新しい集合を返します。

copy()

Return a shallow copy of the set.

なお、演算子でない版の `union()`, `intersection()`, `difference()`, `symmetric_difference()`, `issubset()`, `issuperset()` メソッドは、任意のイテラブルを引数として受け付けます。対して、演算子を使う版では、引数は集合でなくてはなりません。これは、`set('abc') & 'cbs'` のような誤りがちな構文を予防し、より読みやすい `set('abc').intersection('cbs')` を支持します。

`set` と `frozenset` のどちらも、集合同士の比較をサポートします。二つの集合は、それぞれの集合の要素全てが他方にも含まれている (互いに他方の部分集合である) とき、かつそのときに限り等しいです。一方の集合が他方の集合の真部分集合である (部分集合であるが等しくない) とき、かつそのときに限り一方の集合は他方の集合より小さいです。一方の集合が他方の集合の真上位集合である (上位集合であるが等しくない) とき、かつそのときに限り一方の集合は他方の集合より大きいです。

`set` のインスタンスは、`frozenset` のインスタンスと、要素に基づいて比較されます。例えば、`set('abc') == frozenset('abc')` や `set('abc') in set([frozenset('abc')])` は `True` を返します。

部分集合と等価性の比較は全順序付けを行う関数へと一般化することはできません。例えば、互いに素である二つの非空集合は、等しくなく、他方の部分集合でもありませんから、以下のすべてに `False` を返します: `a < b`, `a == b`, そして `a > b`。そのため、`set` は `__cmp__()` メソッドを実装しません。

集合は半順序 (部分集合関係) しか定義しないので、集合のリストにおける `list.sort()` メソッドの出力は未定義です。

集合の要素は、辞書のキーのように、ハッシュ可能 (*hashable*) でなければなりません。

`set` インスタンスと `frozenset` インスタンスを取り混ぜての二項演算は、第一被演算子の型を返します。例えば: `frozenset('ab') | set('bc')` は `frozenset` インスタンスを返します。

以下の表に挙げる演算は `set` に適用されますが、`frozenset` のイミュータブルなインスタンスには適用されません:

update(*others)**set |= other | ...**

全ての other の要素を追加し、set を更新します。

バージョン 2.6 で変更: 複数のイテラブルからの入力を受け入れるようになりました。

intersection.update(*others)**set &= other & ...**

元の set と全ての other に共通する要素だけを残して set を更新します。

バージョン 2.6 で変更: 複数のイテラブルからの入力を受け入れるようになりました。

difference.update(*others)**set -= other | ...**

other に含まれる要素を取り除き、set を更新します。

バージョン 2.6 で変更: 複数のイテラブルからの入力を受け入れるようになりました。

`symmetric_difference_update` (*other*)

`set` $\hat{=}$ *other*

どちらかにのみ含まれて、共通には持たない要素のみで `set` を更新します。

`add` (*elem*)

要素 *elem* を `set` に追加します。

`remove` (*elem*)

要素 *elem* を `set` から取り除きます。 *elem* が `set` に含まれていなければ `KeyError` を送出します。

`discard` (*elem*)

要素 *elem* が `set` に含まれていれば、取り除きます。

`pop` ()

任意に要素をから返し、それを `set` から取り除きます。 `set` が空であれば、 `KeyError` を送出します。

`clear` ()

`set` の全ての要素を取り除きます。

なお、演算子でない版の `update()`, `intersection_update()`, `difference_update()`, および `symmetric_difference_update()` メソッドは、任意のイテラブルを引数として受け付けます。

`__contains__()`, `remove()`, `discard()` メソッドの引数 *elem* は集合かもしれないことに注意してください。その集合と等価な `frozenset` の検索をサポートするために、 *elem* から一時的な `frozenset` を作成します。

参考:

組み込み `set` 型との比較 `sets` モジュールと組み込み `set` 型の違い

5.8 マッピング型 — `dict`

マッピング (*mapping*) オブジェクトは、ハッシュ可能 (*hashable*) な値を任意のオブジェクトに対応付けます。マッピングはミュータブルなオブジェクトです。現在、標準マッピング型は辞書 (*dictionary*) だけです。(他のコンテナについては組み込みの `list`, `set`, および `tuple` クラスと、 `collections` モジュールを参照下さい。)

辞書のキーは ほぼ 任意の値です。ハッシュ可能 (*hashable*) でない値、つまり、リストや辞書その他のミュータブルな型 (オブジェクトの同一性ではなく値で比較されるもの) はキーとして使用できません。キーとして使われる数値型は通常の数値比較のルールに従います: もしふたつの数値が (例えば 1 と 1.0 のように) 等しければ、同じ辞書の項目として互換的に使用できます。(ただし、コンピュータは浮動小数点数を近似値として保管するので、辞書型のキーとして使用するのはいちいち賢くありません。)

辞書は `key: value` 対のカンマ区切りのリストを波括弧でくくることで作成できます。例えば: `{'jack': 4098, 'sjoerd': 4127}` あるいは `{4098: 'jack', 4127: 'sjoerd'}`。あるいは、 `dict` コンストラクタでも作成できます。

`class dict` (***kwarg*)

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

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

オプションの位置引数と空集合の可能性もあるキーワード引数から初期化された新しい辞書を返します。

位置引数が何も与えられなかった場合、空の辞書が作成されます。位置引数が与えられ、それがマッピングオブジェクトだった場合、そのマッピングオブジェクトと同じキーと値のペアを持つ辞書が作成されます。それ以外の場合、位置引数は *iterable* オブジェクトでなければなりません。iterable のそれぞれの要素自身は、ちょうど 2 個のオブジェクトを持つイテラブルでなければなりません。それぞれの要素の最初のオブジェクトは新しい辞書のキーになり、2 番目のオブジェクトはそれに対応する値になります。同一のキーが 2 回以上現れた場合は、そのキーの最後の値が新しい辞書での対応する値になります。

キーワード引数が与えられた場合、キーワード引数とその値が位置引数から作られた辞書に追加されます。既に存在しているキーが追加された場合、キーワード引数の値は位置引数の値を置き換えます。

例を出すと、次の例は全て {"one": 1, "two": 2, "three": 3} に等しい辞書を返します:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

最初の例のようにキーワード引数を与える方法では、キーは有効な Python の識別子でなければなりません。それ以外の方法では、辞書のキーとして有効などんなキーでも使えます。

バージョン 2.2 で追加.

バージョン 2.3 で変更: キーワード引数からの辞書型の作成のサポートが追加されました。

以下は辞書型がサポートする操作です (それゆえ、カスタムのマップ型もこれらの操作をサポートするべきです):

len(d)

辞書 *d* の項目数を返します。

d[key]

d のキー *key* の項目を返します。マップに *key* が存在しなければ、*KeyError* を送出します。

辞書のサブクラスが `__missing__()` メソッドを定義しておらず、*key* が存在しないなら、*d[key]* 演算はこのメソッドをキー *key* を引数として呼び出します。結果として、*d[key]* 演算は、キーが存在しなければ、`__missing__(key)` の呼び出しによって返されまたは送出されたものを何でも、返したりは送出します。他の演算やメソッドは `__missing__()` を呼び出しません。`__missing__()` が定義されていないければ、*KeyError* が送出されます。`__missing__()` はメソッドでなければならず、インスタンス変数であってはなりません:


```

>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1

```

ここでお見せした例は `collections.Counter` 実装の一部です。これとは違った `__missing__` が `collections.defaultdict` で使われています。

バージョン 2.5 で追加: 辞書のサブクラスで `__missing__` メソッドを認識するようになりました。

d[key] = value

`d[key]` に *value* を設定します。

del d[key]

`d` から `d[key]` を削除します。マップに *key* が存在しなければ、`KeyError` を送出します。

key in d

`d` がキー *key* を持っていれば `True` を、そうでなければ、`False` を返します。

バージョン 2.2 で追加.

key not in d

`not key in d` と等価です。

バージョン 2.2 で追加.

iter(d)

辞書のキーに渡るイテレータを返します。これは `iterkeys()` へのショートカットです。

clear()

辞書の全ての項目を消去します。

copy()

辞書の浅いコピーを返します。

fromkeys(seq[, value])

`seq` からキーを取り、値を *value* に設定した、新しい辞書を作成します。

`fromkeys()` は新しい辞書を返すクラスメソッドです。 *value* のデフォルト値は `None` です。

バージョン 2.3 で追加.

get(key[, default])

key が辞書にあれば *key* に対する値を、そうでなければ *default* を返します。 *default* が与えられなかった場合、デフォルトでは `None` となります。そのため、このメソッドは `KeyError` を送出することはありません。

has_key(*key*)

辞書に *key* が存在するかを確認します。 *key in d* が可能になっているので、 *has_key()* は廃れた用法です。

items()

辞書のコピーを (*key*, *value*) の対のリストとして返します。

キーと値のリストは任意の順序で返されますが、ランダムではなく、Python の実装と、辞書への挿入、および、削除操作の来歴によって決まります。

items(), *keys()*, *values()*, *iteritems()*, *iterkeys()* および *itervalues()* は辞書に干渉して更新しなければ、返却順は直接対応関係にあります。つまり (*value*, *key*) の対を *zip()* を使って生成することができます: *pairs = zip(d.values(), d.keys())* 。 *iterkeys()* と *itervalues()* でも同じです: *pairs = zip(d.itervalues(), d.iterkeys())* は先と同じ値を返します。同じリストを得るには *pairs = [(v, k) for (k, v) in d.iteritems()]* としても出来ます。

iteritems()

辞書の (*key*, *value*) の対をイテレータで返します。 *dict.items()* の記述も参照下さい。

iteritems() を辞書の項目の追加や削除と同時に行うと、 *RuntimeError* を送出されるか全ての項目に対する反復に失敗することになります。

バージョン 2.2 で追加.

iterkeys()

辞書のキーをイテレータで返します。 *dict.items()* の記述も参照下さい。

iterkeys() を辞書の項目の追加や削除と同時に行うと、 *RuntimeError* を送出されるか全ての項目に対する反復に失敗することになります。

バージョン 2.2 で追加.

itervalues()

辞書の値をイテレータで返します。 *dict.items()* の記述も参照下さい。

itervalues() を辞書の項目の追加や削除と同時に行うと、 *RuntimeError* を送出されるか全ての項目に対する反復に失敗することになります。

バージョン 2.2 で追加.

keys()

辞書のキーのリストのコピーを返します。 *dict.items()* の記述も参照下さい。

pop(*key*[, *default*])

key が辞書に存在すればその値を辞書から消去して返し、そうでなければ *default* を返します。*default* が与えず、かつ *key* が辞書に存在しなければ *KeyError* を送出します。

バージョン 2.3 で追加.

popitem()

任意の (*key*, *value*) 対を辞書から消去して返します。

集合のアルゴリズムで使われるのと同じように、`popitem()` は辞書を破壊的にイテレートするのに便利です。辞書が空であれば、`popitem()` の呼び出しは `KeyError` を送出します。

setdefault (`key` [, `default`])

もし、`key` が辞書に存在すれば、その値を返します。そうでなければ、値を `default` として `key` を挿入し、`default` を返します。`default` のデフォルトは `None` です。

update ([`other`])

辞書の内容を `other` のキーと値で更新します。既存のキーは上書きされます。返り値は `None` です。

`update()` は、他の辞書オブジェクトでもキー/値の対のイテラブル(タプル、もしくは、長さが2のイテラブル)でも、どちらでも受け付けます。キーワード引数が指定されれば、そのキー/値の対で辞書を更新します: `d.update(red=1, blue=2)`。

バージョン 2.4 で変更: キーと値の対のイテラブル、および、キーワード引数を引数として与えることができるようになりました。

values ()

辞書の値のリストのコピーを返します。`dict.items()` の記述も参照下さい。

viewitems ()

辞書のアイテム ((`key`, `value`) のペア) の新しいビューを返します。ビューオブジェクトについては下に説明があります。

バージョン 2.7 で追加.

viewkeys ()

辞書のキーの新しいビューを返します。ビューオブジェクトについては下に説明があります。

バージョン 2.7 で追加.

viewvalues ()

辞書の値の新しいビューを返します。ビューオブジェクトについては下に説明があります。

バージョン 2.7 で追加.

辞書は、同じ (`key`, `value`) の対を持つときのみ等しくなります。

5.8.1 辞書ビューオブジェクト

`dict.viewkeys()`, `dict.viewvalues()`, `dict.viewitems()` によって返されるオブジェクトは、ビューオブジェクトです。これらは、辞書の項目の動的なビューを提供し、辞書が変更された時、ビューはその変更を反映します。

辞書ビューは、イテレートすることで対応するデータを `yield` できます。また、帰属判定をサポートします:

len (`dictview`)

辞書の項目数を返します。

iter(dictview)

辞書のキー、値、または `((key, value)` のタプルとして表される) 項目に渡るイテレータを返します。

キーと値のリストはある任意の順序でイテレートされますが、ランダムではなく、Python の実装によって変わり、辞書への挿入や削除の履歴に依存します。キー、値、要素のビューを通して、辞書の変更を挟まずにイテレートされたら、その要素の順序は完全に一致します。これにより、`(value, key)` の対を `zip()` で作成できます: `pairs = zip(d.values(), d.keys())`。同じリストを作成する他の方法は、`pairs = [(v, k) for (k, v) in d.items()]` です。

辞書の項目の追加や削除中にビューをイテレートすると、`RuntimeError` を送出したり、すべての項目に渡ってイテレートできなかったりします。

x in dictview

`x` が下にある辞書のキー、値、または項目 (項目の場合、`x` は `(key, value)` タプルであるべきです) にあるとき `True` を返します。

キーのビューは、項目が一意的でハッシュ可能であるという点で、集合に似ています。すべての値がハッシュ可能なら、`(key, value)` 対も一意的でハッシュ可能なので、要素のビューも集合に似ています。(値のビューは、要素が一般に一意的でないことから、集合に似ているとは考えられません。) これによりこれらへの集合演算が利用出来ます (“other” はもう一つのビューが集合です):

dictview & other

辞書ビューと別のオブジェクトの共通部分を新しい集合として返します。

dictview | other

辞書ビューと別のオブジェクトの合併集合を新しい集合として返します。

dictview - other

辞書ビューと別のオブジェクトの差集合 (`dictview` に属して `other` に属さないすべての要素) を新しい集合として返します。

dictview ^ other

辞書ビューと別のオブジェクトの対称差 (`dictview` と `other` のどちらかに属すが両方には属さないすべての要素) を新しい集合として返します。

辞書ビューの使用法の例:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.viewkeys()
>>> values = dishes.viewvalues()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504

>>> # keys and values are iterated over in the same order
```

(次のページに続く)

(前のページからの続き)

```
>>> list(keys)
['eggs', 'bacon', 'sausage', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['spam', 'bacon']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
```

5.9 ファイルオブジェクト

ファイルオブジェクトはCの `stdio` パッケージを使って実装されており、組み込み関数の `open()` で生成することができます。ファイルオブジェクトはまた、`os.popen()` や `os.fdopen()`、ソケットオブジェクトの `makefile()` メソッドのような、他の組み込み関数およびメソッドによっても返されます。一時ファイルは `tempfile` モジュールを使って生成でき、ファイルやディレクトリのコピー、移動、消去などの高次の操作は `shutil` モジュールで行えます。

ファイル操作がI/O関連の理由で失敗した場合例外 `IOError` が送出されます。この理由には例えば `seek()` を端末デバイスに行ったり、読み出し専用で開いたファイルに書き込みを行うといった、何らかの理由によってそのファイルで定義されていない操作を行ったような場合も含まれます。

ファイルは以下のメソッドを持ちます:

`file.close()`

ファイルを閉じます。閉じられたファイルはそれ以後読み書きすることはできません。ファイルが開かれていることが必要な操作は、ファイルが閉じられた後はすべて `ValueError` を送出します。`close()` を一度以上呼び出してもかまいません。

Python 2.5 から `with` 文を使えばこのメソッドを直接呼び出す必要はなくなりました。たとえば、以下のコードは `f` を `with` ブロックを抜ける際に自動的に閉じます。

```
from __future__ import with_statement # This isn't required in Python 2.6

with open("hello.txt") as f:
    for line in f:
        print line,
```

古いバージョンの Python では同じ効果を得るために次のようにしなければいけませんでした。

```
f = open("hello.txt")
try:
```

(次のページに続く)

(前のページからの続き)

```
for line in f:
    print line,
finally:
    f.close()
```

注釈: 全ての Python の ”ファイル的” 型が `with` 文用のコンテキストマネージャとして使えるわけではありません。もし、全てのファイル的オブジェクトで動くようにコードを書きたいのならば、オブジェクトを直接使うのではなく `contextlib` にある `contextlib.closing()` 関数を使うと良いでしょう。

`file.flush()`

`stdio` の `fflush()` のように、内部バッファをフラッシュします。ファイル類似のオブジェクトによっては、この操作は何も行いません。

注釈: `flush()` は必ずしもファイルのデータをディスクに書き込むとは限りません。そのような挙動を保証するには `flush()` の後に `os.fsync()` を使って下さい。

`file.fileno()`

背後にある実装系がオペレーティングシステムに I/O 操作を要求するために用いる、整数の ”ファイル記述子” (ファイルデスク립タ) を返します。この値は他の用途として、`fcntl` モジュールや `os.read()` やその仲間のような、ファイル記述子を必要とする低レベルのインタフェースで役に立ちます。

注釈: 本物のファイルデスク립タを持たないファイル類似のオブジェクトは、このメソッドを提供すべきではありません。

`file.isatty()`

ファイルが tty (または類似の) デバイスに接続されている場合 `True` を返し、そうでない場合 `False` を返します。

注釈: ファイル類似のオブジェクトが実際のファイルに関連付けられていない場合、このメソッドを実装すべきではありません。

`file.next()`

ファイルオブジェクトはそれ自身がイテレータです。すなわち、`iter(f)` は (`f` が閉じられていない限り) `f` を返します。 `for` ループ (例えば `for line in f: print line.strip()`) のようにファイルがイテレータとして使われた場合、`next()` メソッドが繰り返し呼び出されます。ファイルが読み込みのために開かれている場合、このメソッドは次の入力行を返すか、または、EOF に到達したときに `StopIteration` を送出します (ファイルが書き込みモードで開かれている場合、動作は未

定義です)。ファイル内の各行に対する `for` ループ (非常によくある操作です) を効率的な方法で行うために、`next()` メソッドは隠蔽された先読みバッファを使います。先読みバッファを使った結果として、(`readline()` のような) 他のファイルメソッドと `next()` を組み合わせて使うとうまく動作しません。しかし、`seek()` を使ってファイル位置を絶対指定しなおすと、先読みバッファは消去されます。

バージョン 2.3 で追加。

`file.read([size])`

最大で `size` バイトをファイルから読み込みます (`size` バイトを取得する前に EOF に到達した場合、それ以下の長さになります)。 `size` 引数が負であるか省略された場合、EOF に到達するまでの全てのデータを読み込みます。読み出されたバイト列は文字列オブジェクトとして返されます。直後に EOF に到達した場合、空の文字列が返されます。(端末のようなある種のファイルでは、EOF に到達した後でファイルを読みつづけることにも意味があります)。このメソッドは、`size` バイトに可能な限り近くデータを取得するために、背後の C 関数 `fread()` を 1 度以上呼び出すかもしれないので注意してください。また、非ブロック・モードでは、`size` パラメータが与えられなくても、要求されたよりも少ないデータが返される場合があることに注意してください。

注釈: この関数は単純に、背後の C 関数 `fread()` のラッパーです。そのため、EOF が予期せず検出されたりされなかったりといった個別の事情があっても同じように振舞うだけです。

`file.readline([size])`

ファイルから一行全部を読み込みます。行末の改行文字は文字列に残ります (ただし、ファイルが不完全な行で終わっていたら、存在しないかもしれません)。*6 `size` 引数が与えられ、負でなければ、それが (行末の改行文字を含む) 最大バイト数となり、不完全な行でも返されます。 `size` が 0 でなければ、空の文字列が返されるのは、即座に EOF に到達したとき だけ です。

注釈: C `stdio` の `fgets()` と違い、入力中にヌル文字 (`'\0'`) が含まれていれば、ヌル文字を含んだ文字列が返されます。

`file.readlines([sizehint])`

`readline()` を使ってに到達するまで読み出し、EOF 読み出された行を含むリストを返します。オプションの `sizehint` 引数が存在すれば、EOF まで読み出す代わりに完全な行を全体で大体 `sizehint` バイトになるように (おそらく内部バッファサイズを切り詰めて) 読み出します。ファイル類似のインタフェースを実装しているオブジェクトは、`sizehint` を実装できないか効率的に実装できない場合には無視してもかまいません。

`file.xreadlines()`

このメソッドは `iter(f)` と同じ結果を返します。

バージョン 2.1 で追加。

*6 改行を残す利点は、空の文字列が返ると EOF を示し、紛らわしくなくなるからです。また、ファイルの最後の行が改行で終わっているかそうでない (ありえることです!) か (例えば、ファイルを行単位で読みながらその完全なコピーを作成した場合には問題になります) を調べることができます。

バージョン 2.3 で非推奨: 代わりに `for line in file` を使ってください。

`file.seek(offset[, whence])`

C stdio の `fseek()` が行うように、ファイルの現在位置を設定します。 `whence` 引数はオプションで、標準の値は `os.SEEK_SET` もしくは 0 (絶対位置指定) です; 他に取り得る値は `os.SEEK_CUR` もしくは 1 (現在のファイル位置から相対的に seek する) および `os.SEEK_END` もしくは 2 (ファイルの末端から相対的に seek する) です。戻り値はありません。

例えば、`f.seek(2, os.SEEK_CUR)` 位置を 2 つ進めます。 `f.seek(-3, os.SEEK_END)` では終端の 3 つ手前に設定します。

ファイルを追記モード (モード 'a' または 'a+') で開いた場合、書き込みを行うまでに行った `seek()` 操作はすべて元に戻されるので注意してください。ファイルが追記のみの書き込みモード ('a') で開かれた場合、このメソッドは実質何も行いませんが、読み込みが可能な追記モード ('a+') で開かれたファイルでは役に立ちます。ファイルをテキストモード ('b' なしで) 開いた場合、`tell()` が返すオフセットのみが正しい値になります。他のオフセット値を使った場合、その振る舞いは未定義です。

全てのファイルオブジェクトが seek できるとは限らないので注意してください。

バージョン 2.6 で変更: オフセットに浮動小数点数を渡すことは非推奨となりました。

`file.tell()`

C の stdio の `ftell()` のように、ファイルの現在位置を返します。

注釈: Windows では、(`fgets()` の後で) Unix スタイルの改行のファイルを読むときに `tell()` が不正な値を返すことがあります。この問題に遭遇しないためにはバイナリーモード ('rb') を使うようにしてください。

`file.truncate([size])`

ファイルのサイズを切り詰めます。オプションの `size` が存在すれば、ファイルは (最大で) 指定されたサイズに切り詰められます。標準設定のサイズの値は、現在のファイル位置までのファイルサイズです。現在のファイル位置は変更されません。指定されたサイズがファイルの現在のサイズを越える場合、その結果はプラットフォーム依存なので注意してください: 可能性としては、ファイルは変更されないか、指定されたサイズまでゼロで埋められるか、指定されたサイズまで未定義の新たな内容で埋められるか、があります。利用可能な環境: Windows, 多くの Unix 系。

`file.write(str)`

文字列をファイルに書き込みます。戻り値はありません。バッファリングによって、`flush()` または `close()` が呼び出されるまで実際にファイル中に文字列が書き込まれないこともあります。

`file.writelines(sequence)`

文字列からなるシーケンスをファイルに書き込みます。シーケンスは文字列を生成する反復可能なオブジェクトなら何でもかまいません。よくあるのは文字列からなるリストです。戻り値はありません。(関数の名前は `readlines()` と対応づけてつけられました; `writelines()` は行間の区切りを追加しません。)

ファイルはイテレータプロトコルをサポートします。各反復操作では `readline()` と同じ結果を返し、反復

は `readline()` メソッドが空文字列を返した際に終了します。

ファイルオブジェクトはまた、多くの興味深い属性を提供します。これらはファイル類似オブジェクトでは必要ではありませんが、特定のオブジェクトにとって意味を持たせたいなら実装しなければなりません。

`file.closed`

現在のファイルオブジェクトの状態を示すブール値です。この値は読み出し専用の属性です; `close()` メソッドがこの値を変更します。全てのファイル類似オブジェクトで利用可能とは限りません。

`file.encoding`

このファイルが使っているエンコーディングです。Unicode 文字列がファイルに書き込まれる際、Unicode 文字列はこのエンコーディングを使ってバイト文字列に変換されます。さらに、ファイルが端末に接続されている場合、この属性は端末が使っているとおぼしきエンコーディング (この情報は端末がうまく設定されていない場合には不正確なこともあります) を与えます。この属性は読み出し専用で、すべてのファイル類似オブジェクトにあるとは限りません。またこの値は `None` のこともあり、この場合、ファイルは Unicode 文字列の変換のためにシステムのデフォルトエンコーディングを使います。

バージョン 2.3 で追加。

`file.errors`

エンコーディングに用いられる、Unicode エラーハンドラです。

バージョン 2.6 で追加。

`file.mode`

ファイルの I/O モードです。ファイルが組み込み関数 `open()` で作成された場合、この値は引数 `mode` の値になります。この値は読み出し専用の属性で、全てのファイル類似オブジェクトに存在するとは限りません。

`file.name`

ファイルオブジェクトが `open()` を使って生成された場合は、そのファイル名です。そうでなければ、ファイルオブジェクト生成の起源を示す何らかの文字列になり、`<...>` の形式をとります。この値は読み出し専用の属性で、全てのファイル類似オブジェクトに存在するとは限りません。

`file.newlines`

Python が *universal newlines* を (デフォルトどおり) 有効にしてビルドされているなら、この読み込み専用属性が存在し、ファイルが *universal newlines* で開かれたファイルで、ファイルの読み込み中にあった改行の種類を記録します。取り得る値は `'\r'`, `'\n'`, `'\r\n'`, `None` (不明であるか、まだ改行を読み込んでいない)、または、複数の改行方式の種類が存在したことを表す、見つかったすべての改行の種類を含むタプルです。 *universal newlines* で開かれたのでないファイルに対しては、この属性の値は `None` になります。

`file.softspace`

`print` 文を使った場合、他の値を出力する前にスペース文字を出力する必要があるかどうかを示すブール値です。ファイルオブジェクトをシミュレート仕様とするクラスは書き込み可能な *softspace* 属性を持たなければならず、この値はゼロに初期化されなければなりません。この値は Python で実装されているほとんどのクラスで自動的に初期化されます (属性へのアクセス手段を上書きするようなオブジェクトでは注意が必要です); C で実装された型では、書き込み可能な *softspace* 属性を提供しなければなりません。

注釈: この属性は `print` 文を制御するために用いられるのではなく、`print` の実装にその内部状態を追跡させるためにあります。

5.10 メモリビュー型

バージョン 2.7 で追加.

`memoryview` オブジェクトは、Python コードが、バッファプロトコルをサポートするオブジェクトの内部データへ、コピーすることなくアクセスすることを可能にします。メモリは通常、単純なバイト列として解釈されます。

class `memoryview` (*obj*)

obj を参照する `memoryview` を作成します。 *obj* はバッファプロトコルをサポートしていなければなりません。バッファプロトコルをサポートする組み込みオブジェクトには、`str`、`bytearray` などがあります (ただし、`unicode` は違います)。

`memoryview` には 要素 の概念があり、それが起源のオブジェクト *obj* によって扱われる原子的なメモリの単位になります。多くの単純な型、例えば `str` や `bytearray` では、要素は単バイトになりますが、他のサードパーティの型では、要素はより大きくなりえます。

`len(view)` は、メモリビュー *view* の要素の総数を返します。 `itemsizesize` 属性で一つの要素内のバイト数を取得できます。

`memoryview` はスライスしてデータを晒すことに対応しています。一つのインデックスを渡すと一つの要素を `str` オブジェクトとして返します。完全なスライシングは部分ビューになります:

```
>>> v = memoryview('abcefg')
>>> v[1]
'b'
>>> v[-1]
'g'
>>> v[1:4]
<memory at 0x77ab28>
>>> v[1:4].tobytes()
'bce'
```

メモリビューが基にしているオブジェクトがデータの変更に対応していれば、メモリビューはスライス代入に対応します:

```
>>> data = bytearray('abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = 'z'
>>> data
bytearray(b'zbcefg')
```

(次のページに続く)

(前のページからの続き)

```
>>> v[1:4] = '123'
>>> data
bytearray(b'z123fg')
>>> v[2] = 'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot modify size of memoryview object
```

この通り、メモリビューオブジェクトの長さは変えられません。

`memoryview` には 2 つのメソッドがあります。

`tobytes()`

バッファ中のデータをバイト文字列 (クラス `str` のオブジェクト) として返します:

```
>>> m = memoryview("abc")
>>> m.tobytes()
'abc'
```

`tolist()`

バッファ中のデータを整数のリストとして返します:

```
>>> memoryview("abc").tolist()
[97, 98, 99]
```

読み込み専用の属性もいくつか使えます:

`format`

ビューのそれぞれの要素に対する、(`struct` モジュールのスタイルでの) フォーマットを含む文字列です。デフォルトは 'B' で、単純なバイト文字列です。

`itemsize`

メモリビューのそれぞれの要素のバイト数です。

`shape`

メモリの形状を N 次元配列として与える、長さ `ndim` の整数のタプルです。

`ndim`

メモリが表す多次元配列が何次元かを示す整数です。

`strides`

配列のそれぞれの次元に対して、それぞれの要素にアクセスするのに必要なバイト数を表す、長さ `ndim` の整数のタプルです。

`readonly`

メモリが読み込み専用かを表すブールです。

5.11 コンテキストマネージャ型

バージョン 2.5 で追加.

Python の `with` 文は、コンテキストマネージャによって定義される実行時コンテキストの概念をサポートします。これは、ユーザ定義クラスが文の本体が実行される前に進入し文の終わりで脱出する実行時コンテキストを定義できるようにする一対のメソッドを使って実装されます:

コンテキスト管理プロトコル (*context management protocol*) は実行時コンテキストを定義するコンテキストマネージャオブジェクトが提供すべき一対のメソッドから成ります:

`contextmanager.__enter__()`

実行時コンテキストに入り、このオブジェクトまたは他の実行時コンテキストに関連したオブジェクトを返します。このメソッドが返す値はこのコンテキストマネージャを使う `with` 文の `as` 節の識別子に束縛されます。

自分自身を返すコンテキストマネージャの例としてファイルオブジェクトがあります。ファイルオブジェクトは `__enter__()` から自分自身を返して `open()` が `with` 文のコンテキスト式として使われるようにします。

関連オブジェクトを返すコンテキストマネージャの例としては `decimal.localcontext()` が返すものがあります。このマネージャはアクティブな 10 進数コンテキストをオリジナルのコンテキストのコピーにセットしてそのコピーを返します。こうすることで、`with` 文の本体の内部で、外側のコードに影響を与えずに、10 進数コンテキストを変更できます。

`contextmanager.__exit__(exc_type, exc_val, exc_tb)`

実行時コンテキストから抜け、例外 (がもし起こっていたとしても) を抑制することを示すブール値フラグを返します。 `with` 文の本体を実行中に例外が起こったならば、引数にはその例外の型と値とトレースバック情報を渡します。そうでなければ、引数は全て `None` です。

このメソッドから真値が返されると `with` 文は例外の発生を抑え、 `with` 文の直後の文に実行を続けます。そうでなければ、このメソッドの実行を終えると例外の伝播が続きます。このメソッドの実行中に起きた例外は `with` 文の本体の実行中に起こった例外を置き換えてしまいます。

渡された例外を明示的に再送出すべきではありません。その代わりに、このメソッドが偽の値を返すことでメソッドの正常終了と送出された例外を抑制しないことを伝えるべきです。このようにすれば (`contextlib.nested` のような) コンテキストマネージャは `__exit__()` メソッド自体が失敗したのかどうかを簡単に見分けることができます。

Python は幾つかのコンテキストマネージャを、易しいスレッド同期・ファイルなどのオブジェクトの即時クローズ・単純化されたアクティブな 10 進算術コンテキストのサポートのために用意しています。各型はコンテキスト管理プロトコルを実装しているという以上の特別の取り扱いを受けるわけではありません。例については `contextlib` モジュールを参照下さい。

Python のジェネレータ (*generator*) と `contextlib.contextmanager` デコレータ (*decorator*) はこのプロトコルの簡便な実装方法を提供します。ジェネレータ関数を `contextlib.contextmanager` でデコレートすると、デコレートしなければ返されるイテレータを返す代わりに、必要な `__enter__()` および `__exit__()` メソッドを実装したコンテキストマネージャを返すようになります。

これらのメソッドのために Python/C API の中の Python オブジェクトの型構造体に特別なスロットが作られたわけではないことに注意してください。これらのメソッドを定義したい拡張型はこれらを通常の Python からアクセスできるメソッドとして提供しなければなりません。実行時コンテキストを準備するオーバーヘッドに比べたら、一回のクラス辞書の探索のオーバーヘッドは無視できます。

5.12 その他の組み込み型

インタプリタは、その他いくつかの種類のオブジェクトをサポートします。これらのほとんどは 1 または 2 つの演算だけをサポートします。

5.12.1 モジュール

モジュールに対する唯一の特殊な演算は属性アクセス: `m.name` です。ここで *m* はモジュールで、*name* は *m* のシンボルテーブル上に定義された名前にアクセスします。モジュール属性に代入することもできます。(なお、`import` 文は、厳密に言えば、モジュールオブジェクトに対する演算ではありません; `import foo` は *foo* と名づけられたモジュールオブジェクトの存在を必要とはせず、*foo* と名づけられたモジュールの (外部の) 定義を必要とします。)

全てのモジュールにある特殊属性が `__dict__` です。これはモジュールのシンボルテーブルを含む辞書です。この辞書を書き換えると実際にモジュールのシンボルテーブルを変更することができますが、`__dict__` 属性を直接代入することはできません (`m.__dict__['a'] = 1` と書いて `m.a` を 1 に定義することはできますが、`m.__dict__ = {}` と書くことはできません)。 `__dict__` を直接書き換えることは推奨されません。

インタプリタ内に組み込まれたモジュールは、`<module 'sys' (built-in)>` のように書かれます。ファイルから読み出された場合、`<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>` と書かれます。

5.12.2 クラスおよびクラスインスタンス

これらについては `objects` および `class` を参照下さい。

5.12.3 関数

関数オブジェクトは関数定義によって生成されます。関数オブジェクトに対する唯一の操作は、それを呼び出すことです: `func(argument-list)`。

関数オブジェクトには実際には二種類あります: 組み込み関数とユーザ定義関数です。どちらも同じ操作 (関数の呼び出し) をサポートしますが、実装は異なるので、オブジェクトの型も異なります。

詳細は、`function` を参照下さい。

5.12.4 メソッド

メソッドは属性表記を使って呼び出される関数です。メソッドには二種類あります: (リストの `append()` のような) 組み込みメソッドと、クラスインスタンスのメソッドです。組み込みメソッドは、それをサポートする型と一緒に記述されています。

実装では、クラスインスタンスのメソッドに 2 つの読み込み専用の属性を追加しています: `m.im_self` はメソッドが操作するオブジェクトで、`m.im_func` はメソッドを実装している関数です。 `m(arg-1, arg-2, ..., arg-n)` の呼び出しは、`m.im_func(m.im_self, arg-1, arg-2, ..., arg-n)` の呼び出しと完全に等価です。

クラスインスタンスメソッドには、メソッドがインスタンスからアクセスされるかクラスからアクセスされるかによって、それぞれ バインド (束縛) または 非バインド (非束縛) があります。メソッドが非バインドメソッドの場合、`im_self` 属性は `None` になるため、呼び出す際には `self` オブジェクトを明示的に第一引数として指定しなければなりません。この場合、`self` は非バインドメソッドのクラス (またはそのサブクラス) のインスタンスでなければならず、そうでなければ `TypeError` が送出されます。

関数オブジェクトと同じく、メソッドオブジェクトは任意の属性を取得できます。しかし、メソッド属性は実際には背後の関数オブジェクト (`meth.im_func`) に記憶されているので、バインドメソッド、非バインドメソッドへのメソッド属性の設定は許されていません。メソッドへの属性の設定を試みると `AttributeError` が送出されます。メソッド属性を設定するためには、その背後の関数オブジェクトで明示的にセットする必要があります:

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method' # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'instancemethod' object has no attribute 'whoami'
>>> c.method.im_func.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

詳細は、`types` を参照下さい。

5.12.5 コードオブジェクト

コードオブジェクトは、関数本体のような ”擬似コンパイルされた” Python の実行可能コードを表すために実装系によって使われます。コードオブジェクトはグローバルな実行環境への参照を持たない点で関数オブジェクトとは異なります。コードオブジェクトは組み込み関数 `compile()` によって返され、また関数オブジェクトの `func_code` 属性として取り出せます。 `code` モジュールも参照下さい。

コードオブジェクトは、`exec()` 文や組み込み関数 `eval()` に (ソース文字列の代わりに) 渡すことで、実行や評価できます。

詳細は、`types` を参照下さい。

5.12.6 型オブジェクト

型オブジェクトは様々なオブジェクト型を表します。オブジェクトの型は組み込み関数 `type()` でアクセスされます。型オブジェクトには特有の操作はありません。標準モジュール `types` には全ての組み込み型名が定義されています。

型はこのように書き表されます: `<type 'int'>`。

5.12.7 ニルオブジェクト

このオブジェクトは明示的に値を返さない関数によって返されます。このオブジェクトには特有の操作はありません。ニルオブジェクトは一つだけで、`None` (組み込み名) と名づけられています。

`None` と書き表されます。

5.12.8 Ellipsis オブジェクト

このオブジェクトは拡張スライス表記によって使われます (slicings を参照下さい)。特殊な操作は何もサポートしていません。省略表記オブジェクトは一つだけで、その名前は `Ellipsis` (組み込み名) です。

`Ellipsis` と書き表されます。添え字として使う場合 `...` とも書けます。例えば `seq[...]` のように。

5.12.9 NotImplemented オブジェクト

このオブジェクトは、対応していない型の演算を求められたとき、比較や二項演算から返されます。詳細は `comparisons` を参照してください。

`NotImplemented` と書き表されます。

5.12.10 ブール値

ブール値は二つの定数オブジェクト `False` および `True` です。これらは真理値を表すのに使われます (ただし他の値も偽や真とみなされます)。数値処理のコンテキスト (例えば算術演算子の引数として使われた場合) では、これらはそれぞれ 0 および 1 と同様に振舞います。任意の値に対して、真理値と解釈できる場合、組み込み関数 `bool()` は値をブール値に変換するのに使われます (上述の [真理値判定](#) の節を参照してください)。

それぞれ `False` および `True` と書き表されます。

5.12.11 内部オブジェクト

この情報は `types` を参照下さい。スタックフレームオブジェクト、トレースバックオブジェクト、スライスオブジェクトについて記述されています。

5.13 特殊属性

実装は、いくつかのオブジェクト型に対して、適切な場合には特殊な読み出し専用の属性を追加します。そのうちいくつかは `dir()` 組み込み関数で報告されません。

`object.__dict__`

オブジェクトの (書き込み可能な) 属性を保存するために使われる辞書またはその他のマッピングオブジェクトです。

`object.__methods__`

バージョン 2.2 で非推奨: オブジェクトの属性からなるリストを取得するには、組み込み関数 `dir()` を使ってください。この属性はもう利用できません。

`object.__members__`

バージョン 2.2 で非推奨: オブジェクトの属性からなるリストを取得するには、組み込み関数 `dir()` を使ってください。この属性はもう利用できません。

`instance.__class__`

クラスインスタンスが属しているクラスです。

`class.__bases__`

クラスオブジェクトの基底クラスのタプルです。

`definition.__name__`

クラス、型、関数、メソッド、デスク립タ、ジェネレータインスタンスの名前です。

以下の属性は、新スタイルクラス (*new-style class*) でのみサポートされます。

`class.__mro__`

この属性はメソッドの解決時に基底クラスを探索するときに考慮されるクラスのタプルです。

`class.mro()`

このメソッドは、メタクラスによって、そのインスタンスのメソッド解決の順序をカスタマイズするために、上書きされるかも知れません。このメソッドはクラスのインスタンス化時に呼ばれ、その結果は `__mro__` に格納されます。

`class.__subclasses__()`

それぞれの新スタイルクラスは、それ自身の直接のサブクラスへの弱参照を保持します。このメソッドはそれらの参照のうち、生存しているもののリストを返します。例:

```
>>> int.__subclasses__()
[<type 'bool'>]
```


第 6 章

組み込み例外

例外はクラスオブジェクトです。例外はモジュール `exceptions` で定義されています。このモジュールを明示的にインポートする必要はありません。例外は `exceptions` モジュールと同様に組み込み名前空間で与えられます。

`try` 文の中で、`except` 節を使って特定の例外クラスについて記述した場合、その節は指定した例外クラスから派生したクラスも扱います (指定した例外クラスの派生元のクラスは含みません)。サブクラス化の関係にない 2 つの例外クラスは、それらが同じ名前だったとしても等しくなることはありません。

以下に列挙した組み込み例外はインタプリタや組み込み関数によって生成されます。特に注記しないかぎり、これらの例外はエラーの詳しい原因を示している、”関連値 (associated value)” を持ちます。この値は文字列または複数の情報 (例えばエラーコードや、エラーコードを説明する文字列) を含むタプルです。この関連値は `raise` 文の 2 番目の引数です。例外が標準のルートクラスである `BaseException` の派生クラスであれば、関連値は例外インスタンスの `args` 属性中に置かれます。

ユーザによるコードも組み込み例外を送出できます。これを使って、例外ハンドラをテストしたり、インタプリタが同じ例外を送出する状況と ”ちょうど同じような” エラー条件であることを報告したりできます。しかし、ユーザのコードが適切でないエラーを送出するのを妨げる方法はないので注意してください。

組み込み例外クラスは新たな例外を定義するためにサブクラス化することができます。新しい例外は、`Exception` クラスかそのサブクラスの一つから派生することをお勧めします。`BaseException` からは派生しないで下さい。例外を定義する上での詳しい情報は、Python チュートリアル `tut-userexceptions` の項目にあります。

以下の例外クラスは他の例外クラスの基底クラスとしてのみ使われます。

`exception BaseException`

全ての組み込み例外のルートクラスです。ユーザ定義例外が直接このクラスを継承することは意図していません (そうした場合は `Exception` を使ってください)。このクラスに対して `str()` や `unicode()` が呼ばれた場合、引数の文字列表現か、引数が無い時には空文字列が返されます。

バージョン 2.5 で追加。

`args`

例外コンストラクタに与えられた引数のタプルです。組み込み例外は普通、エラーメッセージを与える一つの文字列だけを引数として呼ばれますが、中には (`IOError` など) いくつかの引数を必要

とし、このタプルの要素に特別な意味を込めるものもあります。

exception Exception

システム終了以外の全ての組み込み例外はこのクラスから導出されています。全てのユーザ定義例外もこのクラスの派生クラスであるべきです。

バージョン 2.5 で変更: *BaseException* を継承するように変更されました。

exception StandardError

StopIteration, *SystemExit*, *KeyboardInterrupt*, *SystemExit* 以外の、全ての組み込み例外の基底クラスです。 *StandardError* 自体は *Exception* の派生クラスです。

exception ArithmeticError

算術上の様々なエラーに対して送出される組み込み例外 *OverflowError*, *ZeroDivisionError*, *FloatingPointError* の基底クラスです。

exception BufferError

バッファ (buffer) に関連する演算が行えなかったときに送出されます。

exception LookupError

マッピングまたはシーケンスで使われたキーやインデックスが無効な場合に送出される例外 *IndexError* および *KeyError* の基底クラスです。 *codecs.lookup()* によって直接送出されることもあります。

exception EnvironmentError

Python システムの外部で起こる可能性のある例外 *IOError* および *OSError* の基底クラスです。この型の例外が 2 つの要素を持つタプルで生成された場合、1 番目の要素はインスタンスの *errno* 属性で得ることができます (この値はエラー番号と見なされます)。2 番目の要素は *strerror* 属性です (この値は通常、エラーに関連するメッセージです)。タプル自体は *args* 属性から得ることもできます。

バージョン 1.5.2 で追加。

EnvironmentError 例外が 3 要素のタプルで生成された場合、最初の 2 つの要素は上記と同様に値を得ることができ、さらに 3 番目の要素は *filename* 属性で得ることができます。しかしながら、以前のバージョンとの互換性のために、*args* 属性にはコンストラクタに渡した最初の 2 つの引数からなる 2 要素のタプルしか含みません。

この例外が 3 つ以外の引数で生成された場合、*filename* 属性は *None* になります。この例外が 2 つまたは 3 つ以外の引数で生成された場合、*errno* および *strerror* 属性も *None* になります。後者のケースでは、*args* がコンストラクタに与えた引数をそのままタプルの形で含んでいます。

以下の例外は実際に送出される例外です。

exception AssertionError

assert 文が失敗した場合に送出されます。

exception AttributeError

属性参照 (attribute-references を参照) や代入が失敗した場合に送出されます (オブジェクトが属性の参照や属性の代入をまったくサポートしていない場合には *TypeError* が送出されます)。

exception EOFError

組み込み関数 (`input()` または `raw_input()`) のいずれかで、データを全く読まないうちにファイルの終端 (EOF) に到達した場合に送出されます (注意: `file.read()` および `file.readline()` メソッドの場合、データを読まないうちに EOF にたどり着くと空の文字列を返します)。

exception FloatingPointError

浮動小数点演算が失敗した場合に送出されます。この例外は Python のどのバージョンでも常に定義されていますが、Python が `--with-fpectl` オプションを有効にしてコンパイルされているか、`pyconfig.h` ファイルにシンボル `WANT_SIGFPE_HANDLER` が定義されている場合にのみ送出されます。

exception GeneratorExit

ジェネレータ (*generator*) の `close()` メソッドが呼び出されたときに送出されます。この例外は厳密にはエラーではないので、`StandardError` ではなく `BaseException` を直接継承しています。

バージョン 2.5 で追加。

バージョン 2.6 で変更: `BaseException` を継承するように変更されました。

exception IOError

(`print` 文、組み込みの `open()` またはファイルオブジェクトに対するメソッドといった) I/O 操作が、例えば "ファイルが存在しません" や "ディスクの空き領域がありません" といった I/O に関連した理由で失敗した場合に送出されます。

このクラスは `EnvironmentError` の派生クラスです。この例外クラスのインスタンス属性に関する情報は上記の `EnvironmentError` に関する議論を参照してください。

バージョン 2.6 で変更: `socket.error` は、この例外を基底クラスとして使うように変更されました。

exception ImportError

`import` 文でモジュール定義を見つけられなかった場合や、`from ... import` 文で指定した名前をインポートすることができなかった場合に送出されます。

exception IndexError

シーケンスのインデックス指定がシーケンスの範囲を超えている場合に送出されます (スライスのインデックスはシーケンスの範囲に収まるように暗黙のうちに調整されます; インデックスが通常の整数でない場合、`TypeError` が送出されます)。

exception KeyError

マッピング (辞書) のキーが、存在するキーの集合内に見つからなかった場合に送出されます。

exception KeyboardInterrupt

ユーザが割り込みキー (通常は Control-C または Delete キー) を押した場合に送出されます。割り込みが起きたかどうかはインタプリタの実行中に定期的に調べられます。組み込み関数 `input()` や `raw_input()` がユーザの入力を待っている間に割り込みキーを押してもこの例外が送出されます。この例外は `Exception` を処理するコードに誤って捕捉されてインタプリタの終了が阻害されないように `BaseException` を継承しています。

バージョン 2.5 で変更: `BaseException` を継承するように変更されました。

exception `MemoryError`

ある操作中にメモリが不足したが、その状況は (オブジェクトをいくつか消去することで) まだ復旧可能なかもしれない場合に送出されます。例外の関連値は、どんな種類の (内部) 操作がメモリ不足になっているかを示す文字列です。背後にあるメモリ管理アーキテクチャ (C の `malloc()` 関数) のために、インタプリタが状況を完璧に復旧できるとはかぎらないので注意してください; プログラムの暴走が原因の場合にも、やはり実行スタックの追跡結果を出力できるようにするために例外が送出されます。

exception `NameError`

ローカルまたはグローバルの名前が見つからなかった場合に送出されます。これは非限定の名前のみに適用されます。関連値は見つからなかった名前を含むエラーメッセージです。

exception `NotImplementedError`

この例外は `RuntimeError` から派生しています。ユーザ定義の基底クラスにおいて、抽象メソッドが派生クラスでオーバーライドされることを要求する場合、この例外を送出しなくてはなりません。

バージョン 1.5.2 で追加.

exception `OSError`

このクラスは `EnvironmentError` から派生しています。関数がシステムに関連したエラーを返した場合に送出されます (引数の型が間違っている場合や、他の偶発的なエラーは除きます)。 `errno` 属性は `errno` に基づく数字のエラーコードで、`strerror` 属性は C の `perror()` 関数で表示されるような文字列です。オペレーティングシステムに依存したエラーコードの定義と名前については、 `errno` モジュールを参照して下さい。

ファイルシステムのパスに関係する例外 (`chdir()` や `unlink()` など) では、例外インスタンスは 3 番目の属性 `filename` を持ちます。これは関数に渡されたファイル名です。

バージョン 1.5.2 で追加.

exception `OverflowError`

算術演算の結果が表現できない大きな値になった場合に送出されます。これは long integer の演算と通常の整数に関するほとんどの操作では起こりません (long integer の演算ではむしろ `MemoryError` が送出されることになるでしょう)。整数に関するほとんどの操作では、代わりに long integer が返されます。C の浮動小数点演算の例外処理は標準化されていないので、ほとんどの浮動小数点演算もチェックされません。

exception `ReferenceError`

`weakref.proxy()` によって生成された弱参照 (weak reference) プロキシを使って、ガーベジコレクションによって回収された後の参照対象オブジェクトの属性にアクセスした場合に送出されます。弱参照については `weakref` モジュールを参照してください。

バージョン 2.2 で追加: 以前は `weakref.ReferenceError` 例外として知られていました。

exception `RuntimeError`

他のカテゴリに分類できないエラーが検出された場合に送出されます。関連値は、何が問題だったのかをより詳細に示す文字列です。

exception `StopIteration`

イテレータ (`iterator`) の `next()` メソッドにより、それ以上要素がないことを知らせるために送出され

ます。この例外は、通常の利用方法ではエラーとはみなされないため、`StandardError` ではなく `Exception` から派生しています。

バージョン 2.2 で追加。

exception SyntaxError

パーザが構文エラーに遭遇した場合に送出されます。この例外は `import` 文、`exec` 文、組み込み関数 `eval()` や `input()`、初期化スクリプトの読み込みや標準入力で (対話的な実行時にも) 起こる可能性があります。

このクラスのインスタンスは、例外の詳細に簡単にアクセスできるようにするために、属性 `filename`, `lineno`, `offset`, `text` を持ちます。例外インスタンスに対する `str()` はメッセージのみを返します。

exception IndentationError

正しくないインデントに関する構文エラーの基底クラスです。これは `SyntaxError` のサブクラスです。

exception TabError

タブとスペースを一貫しない方法でインデントに使っているときに送出されます。これは `IndentationError` のサブクラスです。

exception SystemError

インタプリタが内部エラーを発見したが、その状況は全ての望みを棄てさせるほど深刻ではないように思われる場合に送出されます。関連値は (下位層の言葉で) 何がまずいのかを示す文字列です。

Python の作者が、あなたの Python インタプリタを保守している人にこのエラーを報告してください。このとき、Python インタプリタのバージョン (`sys.version`; Python の対話的セッションを開始した際にも出力されます)、正確なエラーメッセージ (例外の関連値) を忘れずに報告してください。そしてもし可能ならエラーを引き起こしたプログラムのソースコードを報告してください。

exception SystemExit

この例外は `sys.exit()` 関数によって送出されます。この例外が処理されなかった場合、スタックのトレースバックを全く表示することなく Python インタプリタは終了します。関連値が通常の整数であれば、システム終了ステータスを表します (`exit()` 関数に渡されます)。値が `None` の場合、終了ステータスは 0 です。(文字列のような) 他の型の場合、そのオブジェクトの値が表示され、終了ステータスは 1 になります。

この例外のインスタンスは属性 `code` を持ちます。この値は終了ステータスまたはエラーメッセージ (標準では `None`) に設定されます。また、この例外は厳密にはエラーではないため、`StandardError` ではなく `BaseException` から派生しています。

`sys.exit()` は、クリーンアップのための処理 (`try` 文の `finally` 節) が実行されるようにするために、またデバッガが制御不能になるリスクを冒さずにスクリプトを実行できるようにするために例外に翻訳されます。即座に終了することが真に強く必要であるとき (例えば、`os.fork()` を呼んだ後の子プロセス内) には `os._exit()` 関数を使うことができます。

この例外は `Exception` を捕まえるコードに間違って捕まえないように、`StandardError` や `Exception` からではなく `BaseException` を継承しています。これにより、この例外は着実に呼出

し元の方に伝わっていったインタプリタを終了させます。

バージョン 2.5 で変更: `BaseException` を継承するように変更されました。

exception `TypeError`

組み込み演算または関数が適切でない型のオブジェクトに対して適用された際に送出されます。関連値は型の不整合に関して詳細を述べた文字列です。

exception `UnboundLocalError`

関数やメソッド内のローカルな変数に対して参照を行ったが、その変数には値が代入されていなかった場合に送出されます。 `NameError` のサブクラスです。

バージョン 2.0 で追加.

exception `UnicodeError`

Unicode に関するエンコードまたはデコードのエラーが発生した際に送出されます。 `ValueError` のサブクラスです。

`UnicodeError` はエンコードまたはデコードのエラーの説明を属性として持っています。例えば、`err.object[err.start:err.end]` は、無効な入力のうちコーデックが処理に失敗した箇所を表します。

encoding

エラーを送出したエンコーディングの名前です。

reason

そのコーデックエラーを説明する文字列です。

object

コーデックがエンコードまたはデコードしようとしたオブジェクトです。

start

`object` の最初の無効なデータのインデクスです。

end

`object` の最後の無効なデータの次のインデクスです。

バージョン 2.0 で追加.

exception `UnicodeEncodeError`

Unicode 関連のエラーがエンコード中に発生した際に送出されます。 `UnicodeError` のサブクラスです。

バージョン 2.3 で追加.

exception `UnicodeDecodeError`

Unicode 関連のエラーがデコード中に発生した際に送出されます。 `UnicodeError` のサブクラスです。

バージョン 2.3 で追加.

exception UnicodeTranslateError

Unicode 関連のエラーがコード翻訳に発生した際に送出されます。 *UnicodeError* のサブクラスです。

バージョン 2.3 で追加。

exception ValueError

演算子や関数が、正しい型だが適切でない値を持つ引数を受け取ったときや、 *IndexError* のようなより詳細な例外では記述できない状況で送出されます。

exception VMSError

VMS においてのみ利用可能です。 VMS 特有のエラーが起こったときに送出されます。

exception WindowsError

Windows 特有のエラーか、エラー番号が *errno* 値に対応しない場合に送出されます。 *winerrno* および *strerror* の値は Windows プラットフォーム API の関数 *GetLastError()* と *FormatMessage()* の戻り値から生成されます。 *errno* の値は *winerror* の値を対応する *errno.h* の値にマップしたものです。 *OSError* のサブクラスです。

バージョン 2.0 で追加。

バージョン 2.5 で変更: 以前のバージョンは *GetLastError()* のコードを *errno* に入れていました。

exception ZeroDivisionError

除算やモジュロ演算の第二引数が 0 であった場合に送出されます。 関連値は文字列で、その演算における被演算子と演算子の型を示します。

以下の例外は警告カテゴリとして使われます。 詳細については *warnings* モジュールを参照してください。

exception Warning

警告カテゴリの基底クラスです。

exception UserWarning

ユーザコードによって生成される警告の基底クラスです。

exception DeprecationWarning

廃止された機能に対する警告の基底クラスです。

exception PendingDeprecationWarning

将来廃止される予定の機能に対する警告の基底クラスです。

exception SyntaxWarning

曖昧な構文に対する警告の基底クラスです。

exception RuntimeWarning

あいまいなランタイム挙動に対する警告の基底クラスです。

exception FutureWarning

将来意味構成が変わることになっている文の構成に対する警告の基底クラスです。

exception `ImportWarning`

モジュールインポートの誤りと思われるものに対する警告の基底クラスです。

バージョン 2.5 で追加.

exception `UnicodeWarning`

Unicode に関連した警告の基底クラスです。

バージョン 2.5 で追加.

exception `BytesWarning`

Base class for warnings related to bytes and bytearray.

バージョン 2.6 で追加.

6.1 例外のクラス階層

組み込み例外のクラス階層は以下のとおりです:

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StandardError
        |   +-- BufferError
        |   +-- ArithmeticError
        |       |   +-- FloatingPointError
        |       |   +-- OverflowError
        |       |   +-- ZeroDivisionError
        |   +-- AssertionError
        |   +-- AttributeError
        |   +-- EnvironmentError
        |       |   +-- IOError
        |       |   +-- OSError
        |       |       +-- WindowsError (Windows)
        |       |       +-- VMSError (VMS)
        |   +-- EOFError
        |   +-- ImportError
        |   +-- LookupError
        |       |   +-- IndexError
        |       |   +-- KeyError
        |   +-- MemoryError
        |   +-- NameError
        |       |   +-- UnboundLocalError
        |   +-- ReferenceError
        |   +-- RuntimeError
        |       |   +-- NotImplementedError
        |   +-- SyntaxError
        |       |   +-- IndentationError
```

(次のページに続く)

(前のページからの続き)

```
| |         +-- TabError
| +-- SystemError
| +-- TypeError
| +-- ValueError
|     +-- UnicodeError
|         +-- UnicodeDecodeError
|         +-- UnicodeEncodeError
|         +-- UnicodeTranslateError
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
```


第 7 章

文字列処理

この章で解説されているモジュールは文字列を操作するさまざまな処理を提供します。

それに加えて、Python の組み込み文字列クラスたちは [シーケンス型](#) — *str*, *unicode*, *list*, *tuple*, *bytearray*, *buffer*, *xrange* で説明されたシーケンス型のメソッドをサポートし、また [文字列メソッド](#) で説明された文字列固有のメソッドもサポートします。フォーマットされた文字列の出力にはテンプレート文字列または [文字列フォーマット操作](#) で説明された `%` 演算子を使います。また、正規表現に基づいた文字列関数については *re* モジュールを参照してください。

7.1 string — 一般的な文字列操作

ソースコード: [Lib/string.py](#)

string モジュールには便利な定数やクラスが数多く入っています。また、現在は文字列のメソッドとして利用できる、すでに撤廃された古い関数も入っています。さらに、Python の組み込み文字列クラスは [シーケンス型](#) — *str*, *unicode*, *list*, *tuple*, *bytearray*, *buffer*, *xrange* 節に記載のシーケンス型のメソッドと、[文字列メソッド](#) 節に記載の文字列メソッドもサポートします。出力の書式指定には、テンプレート文字列、または、[文字列フォーマット操作](#) に記載の `%` 演算子を使用して下さい。正規表現に関する文字列操作の関数は *re* を参照してください。

7.1.1 文字列定数

このモジュールで定義されている定数は以下の通りです:

`string.ascii_letters`

後述の *ascii_lowercase* と *ascii_uppercase* を合わせたもの。この値はロケールに依存しません。

`string.ascii_lowercase`

小文字 'abcdefghijklmnopqrstuvwxyz'。この値はロケールに依存せず、固定です。

`string.ascii_uppercase`

大文字 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'。この値はロケールに依存せず、固定です。

`string.digits`

文字列 '0123456789' です。

`string.hexdigits`

文字列 '0123456789abcdefABCDEF' です。

`string.letters`

後述の *lowercase* と *uppercase* を合わせた文字列です。具体的な値はロケールに依存しており、`locale.setlocale()` が呼ばれたときに更新されます。

`string.lowercase`

小文字として扱われる文字全てを含む文字列です。ほとんどのシステムでは文字列 'abcdefghijklmnopqrstuvwxyz' です。具体的な値はロケールに依存しており、`locale.setlocale()` が呼ばれたときに更新されます。

`string.octdigits`

文字列 '01234567' です。

`string.punctuation`

C ロケールにおいて、句読点として扱われる ASCII 文字の文字列です。

`string.printable`

印刷可能な文字で構成される文字列です。 *digits*, *letters*, *punctuation* および *whitespace* を組み合わせたものです。

`string.uppercase`

大文字として扱われる文字全てを含む文字列です。ほとんどのシステムでは 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' です。具体的な値はロケールに依存しており、`locale.setlocale()` が呼ばれたときに更新されます。

`string.whitespace`

空白 (whitespace) として扱われる文字全てを含む文字列です。ほとんどのシステムでは、これはスペース (space)、タブ (tab)、改行 (linefeed)、復帰 (return)、改頁 (formfeed)、垂直タブ (vertical tab) です。

7.1.2 カスタムの文字列書式化

バージョン 2.6 で追加。

組み込みの `str`、および、`unicode` クラスは、[PEP 3101](#) に記載される `str.format()` メソッドによる、複雑な変数置換と値の書式指定を提供します。`string` モジュールの `Formatter` クラスは組み込みの `format()` メソッドと同じ実装で、文字列の書式指定の作成とカスタマイズを可能にします。

class `string.Formatter`

`Formatter` クラスは、以下のメソッドを持ちます：

format (*format_string*, **args*, ***kwargs*)

主要な API メソッドです。書式化文字列と、任意の位置引数およびキーワード引数のセットを取ります。これは、`vformat()` を呼び出す単なるラッパーです。

vformat (*format_string*, *args*, *kwargs*)

この関数はフォーマットの実際の仕事をします。この関数は、**args* および ***kwargs* シンタックスを使用して、辞書を個々の引数として unpack してから再度 pack するのではなく、引数としてあらかじめ用意した辞書を渡したい場合のために、独立した関数として公開されます。*vformat()* は、書式文字列を文字データと置換フィールドに分解する仕事をします。それは、以下に記述する様々なメソッドを呼び出します。

付け加えると、*Formatter* はサブクラスで置き換えるためのいくつかのメソッドを定義します:

parse (*format_string*)

format_string を探索し、タプル、(*literal_text*, *field_name*, *format_spec*, *conversion*) のイテラブルを返します。これは *vformat()* が文字列を文字としての文字データや置換フィールドに展開するために使用されます。

タプルの値は、概念的に文字としての文字データと、それに続く単一の置換フィールドを表現します。文字としての文字データが無い場合は (ふたつの置換フィールドが連続した場合などに起き得ます)、*literal_text* は長さが 0 の文字列となります。置換フィールドが無い場合は、*field_name*, *format_spec* および *conversion* が *None* となります。

get_field (*field_name*, *args*, *kwargs*)

引数として与えた *parse()* (上記参照) により返される *field_name* を書式指定対象オブジェクトに変換します。戻り値はタプル、(*obj*, *used_key*) です。デフォルトでは **PEP 3101** に規定される "0[name]" や "label.title" のような形式の文字列を引数としてとります。*args* と *kwargs* は *vformat()* に渡されます。戻り値 *used_key* は、*get_value()* の *key* 引数と同じ意味を持ちます。

get_value (*key*, *args*, *kwargs*)

与えられたフィールドの値を取り出します。*key* 引数は整数でも文字列でも構いません。整数の場合は、ポジション引数 *args* のインデックス番号を示します。文字列の場合は、名前付きの引数 *kwargs* を意味します。

args 引数は、*vformat()* へのポジション引数のリストに設定され、*kwargs* 引数は、キーワード引数の辞書に設定されます。

複合したフィールド名に対しては、これらの関数はフィールド名の最初の要素に対してのみ呼び出されます; あとに続く要素は通常の属性、および、インデックス処理へと渡されます。

つまり、例えば、フィールドが '0.name' と表現されるとき、*get_value()* は、*key* 引数が 0 として呼び出されます。属性 *name* は、組み込みの *getattr()* 関数が呼び出され、*get_value()* が返されたのちに検索されます。

もし、インデックス、もしくは、キーワードが存在しないアイテムを参照したら、*IndexError*、もしくは、*KeyError* が送出されます。

check_unused_args (*used_args*, *args*, *kwargs*)

希望に応じ、未使用の引数がないか確認する機能を実装します。この関数への引数は、書式指定文字列で参照される全てのキー引数の set、(ポジション引数への整数、名前付き引数への文字列)、そして *vformat* に渡される *args* と *kwargs* への参照です。使用されない引数の set は、それらのパ

ラメータから計算されます。 `check_unused_args()` は、確認の結果が偽であると、例外を送出するものとみなされます。

format_field(*value*, *format_spec*)

`format_field()` は単純に組み込みのグローバル関数 `format()` を呼び出します。このメソッドは、サブクラスをオーバーライドするために提供されます。

convert_field(*value*, *conversion*)

(`get_field()` が返す) 値を (`parse()` メソッドが返すタブルの形式で) 与えられた変換タイプとして変換します。デフォルトバージョンは 's' (str), 'r' (repr), 'a' (ascii) 変換タイプを理解します。

7.1.3 書式指定文字列の文法

`str.format()` メソッドと、`Formatter` クラスは、文字列の書式指定に同じ文法を共有します (しかしながら、`Formatter` サブクラスの場合、それ自身の書式指定文法を定義することが可能です)。

書式指定文字列は波括弧 `{}` に囲まれた "置換フィールド" を含みます。波括弧に囲まれた部分以外は全て単純な文字として扱われ、変更を加えることなく出力へコピーされます。波括弧を文字として扱う必要がある場合は、二重にすることでエスケープすることができます: `{{ および }}`。

置換フィールドの文法は以下です:

```
replacement_field ::= "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name         ::= arg_name ( "." attribute_name | "[" element_index "]" ) *
arg_name           ::= [identifier | integer]
attribute_name     ::= identifier
element_index      ::= integer | index_string
index_string       ::= <any source character except "]"> +
conversion         ::= "r" | "s"
format_spec        ::= <described in the next section>
```

もっと簡単にいうと、置換フィールドは *field_name* で始められます。これによって指定したオブジェクトの値が、置換フィールドの代わりに書式化され出力に挿入されます。*field_name* の後に、感嘆符 `!` を挟んで *conversion* フィールドを続けることができます。最後にコロン `:` を挟んで、*format_spec* を書くことができます。これは、置換される値の非デフォルトの書式を指定します。

書式指定ミニ言語仕様 節も参照して下さい。

field_name それ自身は、数がキーワードのいずれかである *arg_name* から始まります。それが数である場合、位置引数を参照します。また、それがキーワードである場合、指定されたキーワード引数を参照します。書式化文字列中で数の *arg_names* が順に 0, 1, 2, ... である場合、それらはすべて (いくつかではありません) 省略することができます。そして数 0, 1, 2, ... は、自動的にその順で挿入されます。*arg_name* は引用符で区切られていないので、書式化文字列内の任意の辞書キー (例えば文字列 `'10'` や `':-'` など) を指定することはできません。*arg_name* の後に任意の数のインデックス式または属性式を続けることができます。`' .name '` 形式の式は `getattr()` を使用して指定された属性を選択します。一方、`' [index] '` 形式の式は `__getitem__()`

を使用してインデックス参照を行います。

バージョン 2.7 で変更: The positional argument specifiers can be omitted for `str.format()` and `unicode.format()`, so `'{} {}'` is equivalent to `'{0} {1}'`, `u'{} {}'` is equivalent to `u'{0} {1}'`.

簡単な書式指定文字列の例を挙げます:

```
"First, thou shalt count to {0}" # References first positional argument
"Bring me a {}"                  # Implicitly references the first positional
↪argument
"From {} to {}"                  # Same as "From {0} to {1}"
"My quest is {name}"             # References keyword argument 'name'
"Weight in tons {0.weight}"      # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}"  # First element of keyword argument 'players'.
```

置換 (conversion) フィールドにより書式変換前に型の強制変換が実施されます。通常、値の書式変換は `__format__()` によって実施されます。しかしながら、場合によっては、文字列として変換することを強制したり、書式指定の定義をオーバーライドしたくなることもあります。 `__format__()` の呼び出し前に値を文字列に変換すると、通常の変換の処理は飛ばされます。

現時点では、二種類の変換フラグがサポートされています: 値に対して `str()` を呼び出す `'!s'` と、 `repr()` を呼び出す `'!r'` です。

いくつかの例です:

```
"Harold's a clever {0!s}"        # Calls str() on the argument first
"Bring out the holy {name!r}"    # Calls repr() on the argument first
```

`format_spec` フィールドは、フィールド幅、文字揃え、埋め方、精度などの、値を表現する仕様を含みます。それぞれの値の型は、"formatting mini-language"、または、`format_spec` の実装で定義されます。

ほとんどの組み込み型は、共通の次のセクションに記載の formatting mini-language をサポートします。

`format_spec` フィールド内には入れ子になった置換フィールドを含めることもできます。入れ子になった置換フィールドにはフィールド名、変換フラグ、書式指定を含めることができますが、さらに入れ子の階層を含めることはできません。 `format_spec` 中の置換フィールドは `format_spec` 文字列が解釈される前に置き換えられます。これにより、値の書式を動的に指定することができます。

書式指定例 のいくつかの例も参照して下さい。

書式指定ミニ言語仕様

書式指定 ("Format specifications") は書式指定文字列の個々の値を表現する方法を指定するための、置換フィールドで使用されます (書式指定文字列の文法 を参照下さい)。それらは、組み込み関数の `format()` 関数に直接渡されます。それぞれの書式指定可能な型について、書式指定がどのように解釈されるかが規定されます。

多くの組み込み型は、書式指定に関して以下のオプションを実装します。しかしながら、いくつかの書式指定オプションは数値型でのみサポートされます。

一般的な取り決めとして、空の書式指定文字列("") は、値に対して `str()` を呼び出したときと同じ結果を与えます。通常、空でない書式指定文字列はその結果を変更します。

一般的な書式指定子 (*standard format specifier*) の書式は以下です:

```
format_spec ::=  [[fill]align][sign][#][0][width][,][.precision][type]
```

```
fill        ::=  <any character>
```

```
align       ::=  "<" | ">" | "=" | "^"
```

```
sign        ::=  "+" | "-" | " "
```

```
width       ::=  integer
```

```
precision   ::=  integer
```

```
type        ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x"
```

有効な *align* 値を指定する場合、その前に *fill* 文字を付けることができます。この文字には任意の文字を指定でき、省略された場合はデフォルトの空白文字となります。`str.format()` メソッドを使う場合はリテラルの波括弧("{} と ") を *fill* 文字として使えないことに注意してください。ただし、波括弧を入れ子になった置換フィールド内に挿入することはできます。この制限は `format()` 関数には影響しません。

様々な *align* オプションの意味は以下のとおりです:

オプション	意味
'<'	利用可能なスペースにおいて、左詰めを強制します (ほとんどのオブジェクトにおいてのデフォルト)。
'>'	利用可能なスペースにおいて、右詰めを強制します (いくつかのオブジェクトにおいてのデフォルト)。
'= '	符号 (があれば) の後ろを埋めます。 '+000000120' のような形で表示されます。このオプションは数値型に対してのみ有効です。フィールド幅の直前が '0' の時はこれがデフォルトになります。
'^'	利用可能なスペースにおいて、中央寄せを強制します。

最小のフィールド幅が定義されない限り、フィールド幅はデータを表示するために必要な幅と同じになることに注意して下さい。そのため、その場合には、*align* オプションは意味を持ちません。

sign オプションは数値型に対してのみ有効であり、以下のうちのひとつとなります:

オプション	意味
'+'	符号の使用を、正数、負数の両方に対して指定します。
'- '	符号の使用を、負数に対してのみ指定します (デフォルトの挙動です)。
空白	空白を正数の前に付け、負号を負数の前に使用することを指定します。

'#' オプションは、整数、かつ、2 進数、8 進数、16 進数の出力に対してのみ有効です。指定されれば、出力

は、`'0b'`, `'0o'`, もしくは `'0x'`, のプリフィックスが付与されます。

`','` オプションは、千の位のセパレータにカンマを使うことを合図します。ロケール依存のセパレータには、代わりに `'n'` の整数表現形式を使ってください。

バージョン 2.7 で変更: `','` オプションが追加されました (PEP 378 も参照)。

`width` は 10 進数の整数で、最小のフィールド幅を規程します。もし指定されなければ、フィールド幅は内容により規程されます。

`alignment` が明示的に与えられない場合、`width` フィールドにゼロ (`'0'`) 文字を前置することは、数値型のための符号を意識した 0 パディングを可能にします。これは `fill` 文字に `'0'` を指定して、`alignment` タイプに `'='` を指定したことと等価です。

`precision` は 10 進数で、`'f'` および `'F'`、あるいは、`'g'` および `'G'` で指定される浮動小数点数の、小数点以下に続く桁数を指定します。非数型に対しては、最大フィールド幅を規程します。言い換えると、フィールドの内容から、何文字使用するかを規程します。`precision` は整数型に対しては、許されません。

最後に、`type` は、データがどのように表現されるかを規程します。

利用可能な文字列の表現型は以下です:

型	意味
<code>'s'</code>	文字列。これがデフォルトの値で、多くの場合省略されます。
<code>None</code>	<code>'s'</code> と同じです。

利用可能な整数の表現型は以下です:

型	意味
<code>'b'</code>	2 進数。出力される数値は 2 を基数とします。
<code>'c'</code>	文字。数値を対応する Unicode 文字に変換します。
<code>'d'</code>	10 進数。出力される数値は 10 を基数とします。
<code>'o'</code>	8 進数。出力される数値は 8 を基数とします。
<code>'x'</code>	16 進数。出力される数値は 16 を基数とします。10 進で 9 を越える数字には小文字が使われます。
<code>'X'</code>	16 進数。出力される数値は 16 を基数とします。10 進で 9 を越える数字には大文字が使われます。
<code>'n'</code>	数値。現在のロケールに従い、区切り文字を挿入することを除けば、 <code>'d'</code> と同じです。
<code>None</code>	<code>'d'</code> と同じです。

これらの表現型に加えて、整数は (`'n'` と `None` を除く) 以下の浮動小数点型の表現型で書式指定できます。そうすることで整数は書式変換される前に `float()` を使って浮動小数点数に変換されます。

利用可能な浮動小数点数と 10 進数の表現型は以下です:

型	意味
'e'	指数表記です。指数を示す 'e' を使って数値を表示します。デフォルトの精度は 6 です。
'E'	指数表記です。大文字の 'E' を使うことを除いては、'e' と同じです。
'f'	固定小数点数表記です。数値を固定小数点数として表示します。デフォルトの精度は 6 です。
'F'	Fixed point notation. Same as 'f'.
'g'	汎用フォーマットです。精度を $p \geq 1$ の数値で与えた場合、数値を有効桁 p で丸め、桁に応じて固定小数点か指数表記で表示します。 精度のルールは以下のように決まっています: 書式指定の結果が 'e' 型で $p-1$ の精度の場合、指数は exp になると仮定します。そうすると、 $-4 \leq exp < p$ のとき数値は表現型 'f' で精度 $p-1-exp$ に書式変換されます。それ以外の場合、数値は 'e' 型で精度 $p-1$ に書式指定されます。この両方の場合で重要でない、連続した 0 は取り除かれます、そして残った桁が無い場合小数点は取り除かれます。 正と負の無限大と 0 および NaN は精度に関係なくそれぞれ <code>inf</code> , <code>-inf</code> , <code>0</code> , <code>-0</code> および <code>nan</code> となります。 0 の精度は 1 の精度と同等に扱われます。デフォルトの精度は 6 です。
'G'	汎用フォーマットです。数値が大きくなったとき、'E' に切り替わることを除き、'g' と同じです。無限大と NaN の表示も大文字になります。
'n'	数値です。現在のロケールに合わせて、数値分割文字が挿入されることを除き、'g' と同じです。
'%'	パーセンテージです。数値は 100 倍され、固定小数点数フォーマット ('f') でパーセント記号付きで表示されます。
None	'g' と同じです。

書式指定例

この節では、`str.format()` 構文の例を紹介し、さらに従来の %-書式と比較します。

多くの場合、`{}` を加え % の代わりに `:` を使うことで新構文は古い %-書式に類似した書式になります。例えば、`'%03.2f'` は `'{:03.2f}'` に翻訳できます。

以下の例で示すように、新構文はさらに新たに様々なオプションもサポートしています。

位置引数を使ったアクセス:

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{} , {} , {}'.format('a', 'b', 'c') # 2.7+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc') # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad') # arguments' indices can be repeated
'abracadabra'
```

名前を使ったアクセス:

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-
↳115.81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

引数の属性へのアクセス:

```
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
... 'and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part
↳-5.0.'
```

```
>>> class Point(object):
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

引数の要素へのアクセス:

```
>>> coord = (3, 5)
>>> 'X: {0[0]}; Y: {0[1]}'.format(coord)
'X: 3; Y: 5'
```

%s と %r の置き換え:

```
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
'repr() shows quotes: 'test1'; str() doesn't: test2'
```

テキストの幅を指定した整列:

```
>>> '{:<30}'.format('left aligned')
'left aligned'
>>> '{:>30}'.format('right aligned')
'right aligned'
>>> '{:^30}'.format('centered')
'centered'
>>> '{:*^30}'.format('centered') # use '*' as a fill char
'*****centered*****'
```

%+f と %-f, % f の置換、そして符号の指定:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14) # show it always
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14) # show a space for positive numbers
```

(次のページに続く)

(前のページからの続き)

```
' 3.140000; -3.140000'
>>> '{:-f}; {:f}'.format(3.14, -3.14) # show only the minus -- same as '{:f};
↳{:f}'
'3.140000; -3.140000'
```

%x と %o の置換、そして値に対する異なる底の変換:

```
>>> # format also supports binary numbers
>>> "int: {0:d}; hex: {0:x}; oct: {0:o}; bin: {0:b}".format(42)
'int: 42; hex: 2a; oct: 52; bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d}; hex: {0:#x}; oct: {0:#o}; bin: {0:#b}".format(42)
'int: 42; hex: 0x2a; oct: 0o52; bin: 0b101010'
```

千の位のセパレータにカンマを使用する:

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

パーセントを表示する:

```
>>> points = 19.5
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 88.64%'
```

型特有の書式指定を使う:

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

引数をネストする、さらに複雑な例:

```
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...
'left<<<<<<<<<<<<<<<<<<'
'^^^^^center^^^^^'
'>>>>>>>>>>>>>>>>>>right'
>>>
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
>>> int(_, 16)
3232235521
>>>
>>> width = 5
>>> for num in range(5, 12):
```

(次のページに続く)

(前のページからの続き)

```

...     for base in 'dXob':
...         print '{0:{width}{base}}'.format(num, base=base, width=width),
...     print
...
5      5      5    101
6      6      6    110
7      7      7    111
8      8     10   1000
9      9     11   1001
10     A     12   1010
11     B     13   1011

```

7.1.4 テンプレート文字列

バージョン 2.4 で追加.

テンプレート (template) を使うと、[PEP 292](#) で解説されているように簡潔に文字列置換 (string substitution) を行えるようになります。通常の % ベースの置換に代わって、テンプレートでは以下のような規則に従った \$ ベースの置換をサポートしています:

- `$$` はエスケープ文字です; `$` 一つに置換されます。
- `$identifier` は置換プレースホルダの指定で、`"identifier"` というキーへの対応付けに相当します。デフォルトは、`"identifier"` の部分には Python の識別子が書かれていなければなりません。 `$` の後に識別子に使用できない文字が出現すると、そこでプレースホルダ名の指定が終わります。
- `${identifier}` は `$identifier` と同じです。プレースホルダ名の後ろに識別子として使える文字列が続いていて、それをプレースホルダ名の一部として扱いたくない場合、例えば `"${noun}ification"` のような場合に必要な書き方です。

上記以外の書き方で文字列中に `$` を使うと `ValueError` を送出します。

`string` モジュールでは、上記のような規則を実装した `Template` クラスを提供しています。 `Template` のメソッドを以下に示します:

```
class string.Template(template)
```

コンストラクタはテンプレート文字列になる引数を一つだけ取ります。

```
substitute(mapping[, **kws])
```

テンプレート置換を行い、新たな文字列を生成して返します。 `mapping` はテンプレート中のプレースホルダに対応するキーを持つような任意の辞書類似オブジェクトです。辞書を指定する代わりに、キーワード引数も指定でき、その場合にはキーワードをプレースホルダ名に対応させます。 `mapping` と `kws` の両方が指定され、内容が重複した場合には、 `kws` に指定したプレースホルダを優先します。

```
safe_substitute(mapping[, **kws])
```

`substitute()` と同じですが、プレースホルダに対応するものを `mapping` や `kws` から見つけれなかった場合に、 `KeyError` 例外を送出する代わりにもとのプレースホルダがそのまま入りま

す。また、`substitute()` とは違い、規則外の書き方で `$` を使った場合でも、`ValueError` を送出せず単に `$` を返します。

その他の例外も発生し得る一方で、このメソッドが「安全 (safe)」と呼ばれているのは、置換操作は常に、例外を送出する代わりに利用可能な文字列を返そうとするからです。別の見方をすれば、`safe_substitute()` は区切り間違いによるぶら下がり (dangling delimiter) や波括弧の非対応、Python の識別子として無効なプレースホルダ名を含むような不正なテンプレートをも何も警告せずに無視するため、安全とはいえないのです。

`Template` のインスタンスは、次のような public な属性を提供しています:

template

コンストラクタの引数 `template` に渡されたオブジェクトです。通常、この値を変更すべきではありませんが、読み込み専用アクセスを強制しているわけではありません。

Template の使い方の例を以下に示します:

```
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
...
ValueError: Invalid placeholder in string: line 1, col 11
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
...
KeyError: 'what'
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

さらに進んだ使い方: `Template` のサブクラスを派生して、プレースホルダの書式、区切り文字、テンプレート文字列の解釈に使われている正規表現全体をカスタマイズできます。こうした作業には、以下のクラス属性をオーバーライドします:

- *delimiter* – プレースホルダの開始を示すリテラル文字列です。デフォルトの値は `$` です。実装系はこの文字列に対して必要に応じて `re.escape()` を呼び出すので、正規表現を表すような文字列にしてはなりません。
- *idpattern* – 波括弧でくくらない形式のプレースホルダの表記パターンを示す正規表現です (波括弧は自動的に適切な場所に追加されます)。デフォルトの値は `[_a-z][_a-z0-9]*` という正規表現です。

他にも、クラス属性 `pattern` をオーバーライドして、正規表現パターン全体を指定できます。オーバーライドを行う場合、`pattern` の値は 4 つの名前つきキャプチャグループ (capturing group) を持った正規表現オブジェクトでなければなりません。これらのキャプチャグループは、上で説明した規則と、無効なプレースホルダに対する規則に対応しています:

- *escaped* – このグループはエスケープシーケンス、すなわちデフォルトパターンにおける `$$` に対応します。

- *named* – このグループは波括弧でくくらないプレースホルダ名に対応します; キャプチャグループに区切り文字を含めてはなりません。
- *braced* – このグループは波括弧でくくったプレースホルダ名に対応します; キャプチャグループに区切り文字を含めてはなりません。
- *invalid* – このグループはそのほかの区切り文字のパターン (通常は区切り文字一つ) に対応し、正規表現の末尾に出現しなければなりません。

7.1.5 文字列操作関数

以下の関数は文字列または Unicode オブジェクトを操作できます。これらの関数は文字列型のメソッドにはありません。

`string.capwords(s[, sep])`

`str.split()` を使って引数を単語に分割し、`str.capitalize()` を使ってそれぞれの単語の先頭の文字を大文字に変換し、`str.join()` を使ってつなぎ合わせます。オプションの第 2 引数 *sep* が与えられないか `None` の場合、この置換処理は文字列中の連続する空白文字をスペース一つに置き換え、先頭と末尾の空白を削除します、それ以外の場合には *sep* は `split` と `join` に使われます。

`string.maketrans(from, to)`

`translate()` に渡すのに適した変換テーブルを返します。このテーブルは、*from* 内の各文字を *to* の同じ位置にある文字に対応付けます; *from* と *to* は同じ長さでなければなりません。

注釈: `lowercase` と `uppercase` から取り出した文字列を引数に使ってはなりません; ロケールによっては、これらは同じ長さになりません。大文字小文字の変換には、常に `str.lower()` または `str.upper()` を使ってください。

7.1.6 撤廃された文字列関数

以下の一連の関数は、文字列型や Unicode 型のオブジェクトのメソッドとしても定義されています; 詳しくは、それらの **文字列メソッド** の項を参照してください。ここに挙げた関数は Python 3.0 で削除されることはありませんが、撤廃された関数とみなして下さい。このモジュールで定義されている関数は以下の通りです:

`string.atof(s)`

バージョン 2.0 で非推奨: 組み込み関数 `float()` を使ってください。

文字列を浮動小数点型の数値に変換します。文字列は Python における標準的な浮動小数点リテラルの文法に従っていなければなりません。先頭に符号 (+ または -) が付くのは構いません。この関数に文字列を渡した場合は、組み込み関数 `float()` と同じように振舞います。

注釈: 文字列を渡した場合、根底にある C ライブラリによって NaN や Infinity を返す場合があります。こうした値を返させるのがどんな文字列の集合であるかは、全て C ライブラリに依存しており、ライブ

ラリによって異なると知られています。

`string.atoi(s[, base])`

バージョン 2.0 で非推奨: 組み込み関数 `int()` を使ってください。

文字列 `s` を、`base` を基数とする整数に変換します。文字列は 1 桁またはそれ以上の数字からなっていない必要があります。先頭に符号 (+ または -) が付くのは構いません。`base` のデフォルト値は 10 です。`base` が 0 の場合、(符号を剥ぎ取った後の) 文字列の先頭にある文字列に従ってデフォルトの基数を決定します。0x か 0X なら 16、0 なら 8、その他の場合は 10 が基数になります。`base` が 16 の場合、先頭の 0x や 0X が付いていても受け付けますが、必須ではありません。文字列を渡す場合、この関数は組み込み関数 `int()` と同じように振舞います。(数値リテラルをより柔軟に解釈したい場合には、組み込み関数 `eval()` を使ってください。)

`string.atol(s[, base])`

バージョン 2.0 で非推奨: 組み込み関数 `long()` を使ってください。

文字列 `s` を、`base` を基数とする長整数に変換します。文字列は 1 桁またはそれ以上の数字からなっていない必要があります。先頭に符号 (+ または -) が付くのは構いません。`base` は `atoi()` と同じ意味です。基数が 0 の場合を除き、文字列末尾に `l` や `L` を付けてはなりません。`base` を指定しないか、10 を指定して文字列を渡した場合には、この関数は組み込み関数 `long()` と同じように振舞います。

`string.capitalize(word)`

先頭文字だけ大文字にした `word` のコピーを返します。

`string.expandtabs(s[, tabsize])`

現在のカラムと指定タブ幅に従って文字列中のタブを展開し、一つまたはそれ以上のスペースに置き換えます。文字列中に改行が出現するたびにカラム番号は 0 にリセットされます。この関数は、他の非表示文字やエスケープシーケンスを解釈しません。タブ幅のデフォルトは 8 です。

`string.find(s, sub[, start[, end]])`

`s[start:end]` の中で、部分文字列 `sub` が完全な形で入っている場所のうち、最初のものを `s` のインデックスで返します。見つからなかった場合は -1 を返します。`start` と `end` のデフォルト値、および、負の値を指定した場合の解釈は文字列のスライスと同じです。

`string.rfind(s, sub[, start[, end]])`

`find()` と同じですが、最後に見つかったもののインデックスを返します。

`string.index(s, sub[, start[, end]])`

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

`string.rindex(s, sub[, start[, end]])`

`rfind()` と同じですが、部分文字列が見つからなかったときに `ValueError` 送出します。

`string.count(s, sub[, start[, end]])`

`s[start:end]` における、部分文字列 `sub` の (重複しない) 出現回数を返します。`start` と `end` のデフォルト値、および、負の値を指定した場合の解釈は文字列のスライスと同じです。

`string.lower(s)`

`s` のコピーを大文字を小文字に変換して返します。

```
string.split(s[, sep[, maxsplit]])
```

文字列 *s* 内の単語からなるリストを返します。オプションの第二引数 *sep* を指定しないか、または *None* にした場合、空白文字 (スペース、タブ、改行、リターン、改頁) からなる任意の文字列で単語に区切ります。 *sep* を *None* 以外の値に指定した場合、単語の分割に使う文字列の指定になります。戻り値のリストには、文字列中に分割文字列が重複せずに出現する回数より一つ多い要素が入るはずで、*maxsplit* が与えられた場合には、最大でも *maxsplit* 回の分割しか行わず、リストの最後の要素は未分割の残りの文字列になります (従って、リスト中の要素数は最大でも *maxsplit*+1 です)。 *maxsplit* の指定がないか -1 の場合は、分割数は制限されません (可能な全ての分割を行います)。

空文字列に対する分割を行った場合の挙動は *sep* の値に依存します。 *sep* を指定しないか *None* にした場合、結果は空のリストになります。 *sep* に文字列を指定した場合、空文字列一つの入ったリストになります。

```
string.rsplit(s[, sep[, maxsplit]])
```

s 中の単語からなるリストを *s* の末尾から検索して生成し返します。関数の返す語のリストは全ての点で *split()* の返すものと同じになります。ただし、オプションの第三引数 *maxsplit* をゼロでない値に指定した場合には必ずしも同じにはなりません。 *maxsplit* が与えられた場合には、最大で *maxsplit* 個の分割を右端から行います - 未分割の残りの文字列はリストの最初の要素として返されます (従って、リスト中の要素数は最大でも *maxsplit*+1 です)。

バージョン 2.4 で追加。

```
string.splitfields(s[, sep[, maxsplit]])
```

この関数は *split()* と同じように振舞います。(以前は *split()* は単一引数の場合にのみ使い、*splitfields()* は引数 2 つの場合でのみ使っていました)。

```
string.join(words[, sep])
```

単語のリストやタプルを間に *sep* を入れて連結します。 *sep* のデフォルト値はスペース文字 1 つです。 *string.join(string.split(s, sep), sep)* は常に *s* になります。

```
string.joinfields(words[, sep])
```

この関数は *join()* と同じふるまいをします (以前は、 *join()* を使えるのは引数が 1 つの場合だけで、 *joinfields()* は引数 2 つの場合だけでした)。文字列オブジェクトには *joinfields()* メソッドがないので注意してください。代わりに *join()* メソッドを使ってください。

```
string.lstrip(s[, chars])
```

文字列の先頭から文字を取り除いたコピーを生成して返します。 *chars* を指定しない場合や *None* にした場合、先頭の空白を取り除きます。 *chars* を *None* 以外の値にする場合、 *chars* は文字列でなければなりません。

バージョン 2.2.3 で変更: *chars* パラメータを追加しました。初期の 2.2 バージョンでは、 *chars* パラメータを渡せませんでした。

```
string.rstrip(s[, chars])
```

文字列の末尾から文字を取り除いたコピーを生成して返します。 *chars* を指定しない場合や *None* にした場合、末尾の空白を取り除きます。 *chars* を *None* 以外の値にする場合、 *chars* は文字列でなければなりません。

バージョン 2.2.3 で変更: *chars* パラメータを追加しました。初期の 2.2 バージョンでは、*chars* パラメータを渡せませんでした。

`string.strip(s[, chars])`

文字列の先頭と末尾から文字を取り除いたコピーを生成して返します。 *chars* を指定しない場合や `None` にした場合、先頭と末尾の空白を取り除きます。 *chars* を `None` 以外に指定する場合、*chars* は文字列でなければなりません。

バージョン 2.2.3 で変更: *chars* パラメータを追加しました。初期の 2.2 バージョンでは、*chars* パラメータを渡せませんでした。

`string.swapcase(s)`

s の大文字と小文字を入れ替えたものを返します。

`string.translate(s, table[, deletechars])`

s の中から、(もし指定されていれば) *deletechars* に入っている文字を削除し、*table* を使って文字変換を行って返します。 *table* は 256 文字からなる文字列で、各文字はそのインデックスを序数とする文字に対する変換先の文字の指定になります。もし、*table* が `None` であれば、文字削除のみが行われます。

`string.upper(s)`

s に含まれる小文字を大文字に置換して返します。

`string.ljust(s, width[, fillchar])`

`string.rjust(s, width[, fillchar])`

`string.center(s, width[, fillchar])`

文字列を指定した文字幅のフィールド中でそれぞれ左寄せ、右寄せ、中央寄せします。これらの関数は指定幅になるまで文字列 *s* の左側、右側、および、両側のいずれかに *fillchar* (デフォルトでは空白) を追加して、少なくとも *width* 文字からなる文字列にして返します。文字列を切り詰めることはありません。

`string.zfill(s, width)`

数値を表現する文字列 *s* の左側に、指定の幅 *width* になるまでゼロを付加します。符号付きの数字も正しく処理します。

`string.replace(s, old, new[, maxreplace])`

s 内の部分文字列 *old* を全て *new* に置換したものを返します。 *maxreplace* を指定した場合、最初に見つかった *maxreplace* 個分だけ置換します。

7.2 re — 正規表現操作

このモジュールでは、Perl で見られるものと同様な正規表現マッチング操作を提供しています。パターンと検索対象文字列の両方について、8 ビット文字列と Unicode 文字列を同じように扱えます。

正規表現では、特殊な形式を表したり、特殊文字の持つ特別な意味を呼び出さずにその特殊な文字を使えるようにするために、バックスラッシュ文字 ('\') を使います。こうしたバックスラッシュの使い方は、Python の文字列リテラルにおける同じバックスラッシュ文字と衝突を起こします。例えば、バックスラッシュ自体にマッチさせるには、パターン文字列として '\\\\' と書かなければなりません、というのも、正規表現は \\

でなければならず、さらに正規な Python 文字列リテラルでは各々のバックスラッシュを `\\` と表現せねばならないからです。

正規表現パターンに Python の raw string 記法を使えばこの問題を解決できます。'r' を前置した文字列リテラル内ではバックスラッシュを特別扱いしません。従って、`"\n"` が改行一文字の入った文字列になるのに対して、`r"\n"` は `'\'` と `'n'` という二つの文字の入った文字列になります。通常、Python コード中では、パターンをこの raw string 記法を使って表現します。

大抵の正規表現操作がモジュールレベルの関数と、`RegexObject` のメソッドとして提供されることに注意して下さい。関数は正規表現オブジェクトのコンパイルを必要としない近道ですが、いくつかのチューニング変数を失います。

参考:

サードパーティの `regex` モジュールは、標準ライブラリの `re` モジュールと互換な API を持ち、追加の機能とより徹底した Unicode サポートを提供します。

7.2.1 正規表現のシンタクス

正規表現 (すなわち RE) は、表現にマッチ (match) する文字列の集合を表しています。このモジュールの関数を使えば、ある文字列が指定の正規表現にマッチするか (または指定の正規表現がある文字列にマッチするか、つまりは同じことですが) を検査できます。

正規表現を連結すると新しい正規表現を作れます。A と B がともに正規表現であれば AB も正規表現です。一般的に、文字列 *p* が A とマッチし、別の文字列 *q* が B とマッチすれば、文字列 *pq* は AB にマッチします。ただし、この状況が成り立つのは、A と B との間に境界条件がある場合や、番号付けされたグループ参照のような、優先度の低い演算を A や B が含まない場合だけです。かくして、ここで述べるような、より簡単にプリミティブな正規表現から、複雑な正規表現を容易に構築できます。正規表現に関する理論と実装の詳細については上記の Friedl 本か、コンパイラの構築に関する教科書を調べて下さい。

以下で正規表現の形式に関する簡単な説明をしておきます。より詳細な情報やよりやさしい説明に関しては、`regex-howto` を参照下さい。

正規表現には、特殊文字と通常文字の両方を含められます。'A'、'a'、あるいは'0'のようなほとんどの通常文字は最も簡単な正規表現になります。こうした文字は、単純にその文字自体にマッチします。通常の文字は連結できるので、`last` は文字列 `'last'` とマッチします。(この節の以降の説明では、正規表現を引用符を使わずにこの表示スタイル: `special style` で書き、マッチ対象の文字列は、' 引用符で括って' 書きます。)

'|' や '(' といったいくつかの文字は特殊文字です。特殊文字は通常の文字の種別を表したり、あるいは特殊文字の周辺にある通常の文字に対する解釈方法に影響します。正規表現パターン文字列には、null byte を含めることができませんが、`\number` 記法や、`'\x00'` などとして指定することができます。

繰り返しの修飾子 (`*`, `+`, `?`, `{m,n}` など) は直接入れ子にはできません。これによって、非貪欲な修飾子の接尾辞 `?` や他の実装での他の修飾子についての曖昧さを回避しています。繰り返しのある正規表現の外側にさらに繰り返しを適用するには丸括弧が使えます。例えば、正規表現 `(?:a{6})*` は 6 つの 'a' の 0 回以上の繰り返しに適合します。

特殊文字を以下に示します:

'.' (ドット) デフォルトのモードでは改行以外の任意の文字にマッチします。 *DOTALL* フラグが指定されていれば改行も含むすべての文字にマッチします。

'^' (キャレット) 文字列の先頭とマッチします。 *MULTILINE* モードでは各改行の直後にマッチします。

'\$' 文字列の末尾、あるいは文字列の末尾の改行の直前にマッチします。例えば、`foo` は `'foo'` と `'foobar'` の両方にマッチします。一方、正規表現 `foo$` は `'foo'` だけとマッチします。興味深いことに、`'foo1\nfoo2\n'` を `foo.$` で検索した場合、通常のモードでは `'foo2'` だけにマッチし、*MULTILINE* モードでは `'foo1'` にもマッチします。`$` だけで `'foo\n'` を検索した場合、2箇所 (内容は空) でマッチします: 1つは、改行の直前で、もう1つは、文字列の最後です。

'*' 直前にある RE に作用して、RE を 0 回以上できるだけ多く繰り返したものにマッチさせるようにします。例えば `ab*` は `'a'`、`'ab'`、あるいは `'a'` に任意個数の `'b'` を続けたものにマッチします。

'+' 直前にある RE に作用して、RE を、1 回以上繰り返したものにマッチさせるようにします。例えば `ab+` は `'a'` に一つ以上の `'b'` が続いたものにマッチし、`'a'` 単体にはマッチしません。

'?' 直前にある RE に作用して、RE を 0 回か 1 回繰り返したものにマッチさせるようにします。例えば `ab?` は `'a'` あるいは `'ab'` にマッチします。

`*?`, `+?`, `??` `'*'`、`'+'`、`'?'` といった修飾子は、すべて 貪欲 (*greedy*) マッチ、すなわちできるだけ多くのテキストにマッチするようになっています。時にはこの動作が望ましくない場合もあります。例えば正規表現 `<.*>` を `<a> b <c>` にマッチさせると、`<a>` だけにマッチするのではなく全文字列にマッチしてしまいます。`?` を修飾子の後に追加すると、非貪欲 (*non-greedy*) あるいは 最小一致 (*minimal*) のマッチになり、できるだけ少ない文字数のマッチになります。例えば正規表現 `<.*?>` を使うと `<a>` だけにマッチします。

{*m*} 前にある RE の *m* 回の正確なコピーとマッチすべきであることを指定します; マッチ回数が少なければ、RE 全体ではマッチしません。例えば、`a{6}` は、正確に 6 個の `'a'` 文字とマッチしますが、5 個ではマッチしません。

{*m*, *n*} 結果の RE は、前にある RE を、*m* 回から *n* 回まで繰り返したもので、できるだけ多く繰り返したものとマッチするように、マッチします。例えば、`a{3,5}` は、3 個から 5 個の `'a'` 文字とマッチします。*m* を省略するとマッチ回数の下限として 0 を指定した事になり、*n* を省略することは、上限が無限であることを指定します; `a{4,}b` は `aaaab` や、千個の `'a'` 文字に `b` が続いたものとマッチしますが、`aaab` とはマッチしません。コンマは省略できません、そうでないと修飾子が上で述べた形式と混同されてしまうからです。

{*m*, *n*}? 結果の RE は、前にある RE の *m* 回から *n* 回まで繰り返したもので、できるだけ少なく繰り返したものとマッチするように、マッチします。これは、前の修飾子の控え目バージョンです。例えば、6 文字文字列 `'aaaaaa'` では、`a{3,5}` は、5 個の `'a'` 文字とマッチしますが、`a{3,5}?` は 3 個の文字とマッチするだけです。

'\' 特殊文字をエスケープする (`'*'` や `'?'` 等のような文字とのマッチをできるようにする) か、あるいは、特殊シーケンスの合図です; 特殊シーケンスは後で議論します。

もしパターンを表現するのに raw string を使用していないのであれば、Python も、バックスラッシュを文字列リテラルでのエスケープシーケンスとして使っていることを覚えていて下さい; もしエスケ

ブシーケンスを Python の構文解析器が認識して処理しなければ、そのバックスラッシュとそれに続く文字は、結果の文字列にそのまま含まれます。しかし、もし Python が結果のシーケンスを認識するのであれば、バックスラッシュを 2 回繰り返さなければいけません。このことは複雑で理解しにくいので、最も簡単な表現以外は、すべて raw string を使うことをぜひ勧めます。

[] 文字の集合を指定するのに使用します。集合には以下のものが指定できます:

- 個別に指定できる文字。[amk] は 'a', 'm', 'k' とマッチします。
- 連続した文字の範囲を、先頭と最後の 2 文字とその間に '-' を挟んだ形で指定できます。[a-z] はすべての小文字の ASCII 文字とマッチします。[0-5][0-9] は 00 から 59 までの、すべての 2 桁の数字とマッチします。[0-9A-Fa-f] は任意の 16 進数の数字とマッチします。- が、エスケープされた場合 (例: [a\ -z])、あるいは先頭か末尾に置かれた場合 (例: [a-])、リテラル '-' とマッチします。
- 集合内では、特殊文字はその意味を失います。[(+*)] はリテラル文字 '(' '+'、'*'、あるいは ')' のいずれかとマッチします。
- \w や \s のような文字クラス (後述) も集合内に指定できますが、それらにマッチする文字は *LOCALE* か *UNICODE* のどちらかが有効にされているモードに依存します。
- 範囲内にない文字とは、その集合の 補集合 をとることでマッチできます。集合の最初の文字が '^' の時、集合に ない 文字すべてとマッチします。[^5] は '5' を除くあらゆる文字にマッチします。[^^] は '^' を除くあらゆる文字にマッチします。^ は集合の最初の文字でない限り特別の意味を持ちません。
- 集合内でリテラル '[' をマッチさせるには、その前にバックスラッシュをつけるか、集合の先頭に置きます。[() [\] {}] と [] () [{}] はどちらも '[' にマッチします。

'|' A|B は、ここで A と B は任意の RE ですが、A か B のどちらかとマッチする正規表現を作成します。任意個数の RE を、こういう風に '|' で分離することができます。これはグループ (以下参照) 内部でも同様に使えます。検査対象文字列をスキャンする中で、'|' で分離された RE は左から右への順に検査されます。一つでも完全にマッチしたパターンがあれば、そのパターン枝が受理されます。このことは、もし A がマッチすれば、たとえ B によるマッチが全体としてより長いマッチになったとしても、B を決して検査しないことを意味します。言いかえると、'|' 演算子は決して貪欲 (greedy) ではありません。文字通りの '|' とマッチするには、\\ を使うか、あるいはそれを [] のように文字クラス内に入れます。

(...) 丸括弧の中にどのような正規表現があってもマッチし、またグループの先頭と末尾を表します; グループの中身は、マッチが実行された後に検索され、後述する \number 特殊シーケンス付きの文字列内で、後でマッチされます。文字通りの '(' や ')' とマッチするには、\ (あるいは\) を使うか、それらを文字クラス内に入れます: [() []] 。

(?...) これは拡張記法です ('(' に続く '?' は他には意味がありません)。 '?' の後の最初の文字が、この構造の意味とこれ以上のシンタクスがどういうものであるかを決定します。拡張記法は普通新しいグループを作成しません; (?P<name>...) がこの規則の唯一の例外です。以下に現在サポートされている拡張記法を示します。

(?iLmsux) (集合 'i', 'L', 'm', 's', 'u', 'x' から 1 文字以上)。グループは空文字列ともマッチしま

す；文字は、正規表現全体の対応するフラグ (`re.I` (大文字・小文字を区別しない), `re.L` (ロケール依存), `re.M` (MULTILINE モード), `re.S` (DOTALL モード), `re.U` (Unicode 依存), `re.X` (冗長)) を設定します。(フラグについては、[モジュールコンテンツ](#) に記述があります) これは、もし `flag` 引数を `re.compile()` 関数に渡さずに、そのフラグを正規表現の一部として含めたいならば役に立ちます。

(`?x`) フラグは、式が構文解析される方法を変更することに注意して下さい。これは式文字列内の最初か、あるいは1つ以上の空白文字の後で使うべきです。もしこのフラグの前に非空白文字があると、その結果は未定義です。

(`?:...)` 正規表現の丸括弧の取り込まないバージョンです。どのような正規表現が丸括弧内にあってもマッチしますが、グループによってマッチされたサブ文字列は、マッチを実行したあと検索されることも、あるいは後でパターンで参照されることもできません。

(`?P<name>...`) 正規表現の丸括弧に似ていますが、グループによってマッチした部分文字列はシンボリックグループ名 `name` によってアクセス可能になります。グループ名は有効な Python 識別子でなければならず、グループ名は1個の正規表現内で一意でなければなりません。シンボリックグループは番号付けもされており、番号によるアクセスも可能です。

名前付きグループは3つのコンテキストで参照できます。パターンが (`?P<quote>['\"']`).`.*?`(`?P=quote`) (シングルまたはダブルクォートのどちらかにマッチ) の場合 ‘:

グループ "quote" を参照するコンテキスト	参照方法
同一パターンへの参照	<ul style="list-style-type: none"> (<code>?P=quote</code>) (そのまま) <code>\1</code>
マッチオブジェクト <code>m</code> の処理時	<ul style="list-style-type: none"> <code>m.group('quote')</code> <code>m.end('quote')</code> (etc.)
<code>re.sub()</code> の <code>repl</code> 引数へ渡される文字列	<ul style="list-style-type: none"> <code>\g<quote></code> <code>\g<1></code> <code>\1</code>

(`?P=name`) 名前付きグループへの後方参照です；既出のグループ名 `name` にマッチする文字列は何にでもマッチします。

(`?#...`) コメントです；括弧の内容は単純に無視されます。

(`?=...`) もし... が次に続くものとマッチすればマッチしますが、文字列をまったく消費しません。これは先読みアサーション (lookahead assertion) と呼ばれます。例えば、`Isaac (?=Asimov)` は、`'Isaac '` に `'Asimov'` が続く場合だけ、`'Isaac '` とマッチします。

(`?!...`) もし... が次に続くものとマッチしなければマッチします。これは否定先読みアサーション (negative lookahead assertion) です。例えば、`Isaac (?!Asimov)` は、`'Isaac '` に `'Asimov'` が続かない場合のみマッチします。

(?<=...) 文字列内の現在位置の前に、現在位置で終わる ... とのマッチがあれば、マッチします。これは 後読みアサーション と呼ばれます。(?<=abc)def は abcdef にマッチを見つけてます。後読みは 3 文字をバックアップし、含まれているパターンとマッチするかどうか検査します。含まれるパターンは、固定長の文字列にのみマッチしなければなりません。すなわち、abc や a|b は許されますが、a* や a{3,4} は許されません。グループ参照は固定長の文字列にマッチするときでさえサポートされません。肯定後読みアサーションで始まるパターンは、検索される文字列の先頭とは決してマッチしないことに注意して下さい; この表現を使用するのは、おそらく `match()` 関数より `search()` 関数の方が適しています:

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

この例ではハイフンに続く単語を探します:

```
>>> m = re.search('(?<=)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

(?!...) 文字列内の現在位置の前に ... とのマッチがない場合に、マッチします。これは 否定後読みアサーション (*negative lookbehind assertion*) と呼ばれます。肯定後読みアサーションと同様に、含まれるパターンは固定長さの文字列だけにマッチしなければならず、グループ参照を含んではなりません。否定後読みアサーションで始まるパターンは、検索される文字列の先頭とマッチできます。

(?(id/name)yes-pattern|no-pattern) グループに *id* が与えられている、もしくは *name* があるとき、yes-pattern とマッチします。存在しないときには no-pattern とマッチします。no-pattern はオプションで省略できます。例えば (<)?(\w+@\w+(?:\.\w+)+)(?(1)>) は email アドレスとマッチする最低限のパターンです。これは '<user@host.com>' や 'user@host.com' にはマッチしますが、 '<user@host.com' にはマッチしません。

バージョン 2.4 で追加.

特殊シーケンスは '\ ' と以下のリストにある文字から構成されます。もしリストにあるのが通常文字でないならば、結果の RE は 2 番目の文字とマッチします。例えば、\\$ は文字 '\$' とマッチします。

\number 同じ番号のグループの中身とマッチします。グループは 1 から始まる番号をつけられます。例えば、(.+)\1 は、'the the' あるいは '55 55' とマッチしますが、'thethe' とはマッチしません (グループの後のスペースに注意して下さい)。この特殊シーケンスは最初の 99 グループのうちの一つとマッチするのに使うことができます。もし *number* の最初の桁が 0 である、すなわち *number* が 3 桁の 8 進数であれば、それはグループのマッチとは解釈されず、8 進数値 *number* を持つ文字として解釈されます。文字クラスの '[' と ']' の中の数値エスケープは、文字として扱われます。

\A 文字列の先頭だけにマッチします。

\b 空文字列とマッチしますが、単語の先頭か末尾の時だけです。単語とは英数字またはアンダースコアからなるシーケンスで、単語の終わりは空白文字、あるいはアンダースコアを除く記号で表します。 \b は \w および \W の間 (およびその逆) あるいは \w と文字列の開始/終了との間の境界として定

義されていますので、文字の正確な集合は `UNICODE` と `LOCALE` フラグの値に依存します。例えば、`r'\bfoo\b'` は `'foo'`, `'foo.'`, `'(foo)'`, `'bar foo baz'` にマッチしますが、`'foobar'`, `'foo3'` にはマッチしません。文字範囲内では、`\b` は Python の文字列リテラルとの互換性のため、後退 (backspace) 文字を表します。

- `\B` 空文字列とマッチしますが、それが単語の先頭あるいは末尾に ない 時だけです。 `r'py\B'` は `'python'`, `'py3'`, `'py2'` にはマッチしますが、 `'py'`, `'py.'`, `'py!'` にはマッチしません。これは `\b` のちょうど反対ですので、同じように `LOCALE` と `UNICODE` の設定に影響されます。
- `\d` `UNICODE` フラグが指定されていない場合、任意の十進数とマッチします；これは集合 `[0-9]` と同じ意味です。 `UNICODE` がある場合、Unicode 文字特性データベースで十進数字と分類されているものにマッチします。
- `\D` `UNICODE` フラグが指定されていない場合、任意の非数字文字とマッチします；これは集合 `[^0-9]` と同じ意味です。 `UNICODE` がある場合、これは Unicode 文字特性データベースで数字とマーク付けされている文字以外にマッチします。
- `\s` `UNICODE` フラグが指定されていない場合、任意の空白文字とマッチし、これは集合 `[\t\n\r\f\v]` と同じ意味です。 `LOCALE` フラグは空白文字とのマッチについて特別な意味を持ちません。 `UNICODE` が指定されている場合、これは `[\t\n\r\f\v]` と Unicode 文字特性データベースで空白と分類されている全てにマッチします。
- `\S` `UNICODE` フラグが指定されていない場合、任意の非空白文字とマッチし、これは集合 `[^\t\n\r\f\v]` と同じ意味です。 `LOCALE` フラグは非空白文字とのマッチについて特別な意味を持ちません。 `UNICODE` が指定されている場合、これは Unicode 文字特性データベースで非空白と分類されている全てにマッチします。
- `\w` `LOCALE` と `UNICODE` フラグが指定されていない時は、任意の英数文字および下線とマッチします；これは、集合 `[a-zA-Z0-9_]` と同じ意味です。 `LOCALE` が設定されていると、集合 `[0-9_]` プラス現在のロケール用に英数字として定義されている任意の文字とマッチします。もし `UNICODE` が設定されていれば、文字 `[0-9_]` プラス Unicode 文字特性データベースで英数字として分類されているものとマッチします。
- `\W` `LOCALE` と `UNICODE` フラグが指定されていない時、任意の非英数文字とマッチします；これは集合 `[^a-zA-Z0-9_]` と同じ意味です。 `LOCALE` が指定されていると、集合 `[0-9_]` になく、現在のロケールで英数字として定義されていない任意の文字とマッチします。もし `UNICODE` がセットされていれば、これは `[0-9_]` 以外と、および Unicode 文字特性データベースで非英数字と分類されている文字とマッチします。
- `\Z` 文字列の末尾とのみマッチします。

特定のシーケンスに対し `LOCALE` と `UNICODE` のどちらも影響しう場合は、`LOCALE` が指定されていれば必ず `LOCALE` の効果に従います。

Python 文字列リテラルによってサポートされている標準エスケープのほとんども、正規表現パーザに認識されます：

<code>\a</code>	<code>\b</code>	<code>\f</code>	<code>\n</code>
<code>\r</code>	<code>\t</code>	<code>\v</code>	<code>\x</code>
<code>\\</code>			

(`\b` は単語の境界を表し、文字クラス内でのみ後退 (backspace) 文字を指すことに注意してください)

8 進エスケープは制限された形式で含まれています：もし第 1 桁が 0 であるか、もし 8 進 3 桁であれば、それは 8 進エスケープとみなされます。そうでなければ、それはグループ参照です。文字列リテラルについて、8 進エスケープはほとんどの場合 3 桁長になります。

参考:

Mastering Regular Expressions 詳説正規表現 Jeffrey Friedl 著、O'Reilly 刊の正規表現に関する本です。

この本の第 2 版では Python については触れていませんが、良い正規表現パターンの書き方を非常にくわしく説明しています。

7.2.2 モジュールコンテンツ

このモジュールは幾つかの関数、定数、例外を定義します。この関数のいくつかはコンパイル済み正規表現向けの完全版のメソッドを簡略化したバージョンです。それなりのアプリケーションのほとんどで、コンパイルされた形式が用いられるのが普通です。

`re.compile(pattern, flags=0)`

正規表現パターンを正規表現オブジェクトにコンパイルします。このオブジェクトは、以下で述べる `match()` と `search()` メソッドを使って、マッチングに使うことができます。

式の動作は、`flags` の値を指定することで加減することができます。値は以下の変数を、ビットごとの OR (| 演算子) を使って組み合わせることができます。

シーケンス

```
prog = re.compile(pattern)
result = prog.match(string)
```

は、

```
result = re.match(pattern, string)
```

と同じ意味ですが、`re.compile()` を使ってその結果の正規表現オブジェクトを再利用した方が、その式を一つのプログラムで何回も使う時にはより効率的です。

注釈: 最後に `re.match()`, `re.search()`, `re.compile()` に渡されたパターンのコンパイルされたものがキャッシュとして残ります。そのため、正規表現をひとつだけしか使わないプログラムは正規表現のコンパイルを気にする必要はありません。

re.DEBUG

コンパイルした表現に関するデバッグ情報を出力します。

re.I**re.IGNORECASE**

Perform case-insensitive matching; expressions like `[A-Z]` will match lowercase letters, too. This is not affected by the current locale. To get this effect on non-ASCII Unicode characters such as 端 and , add the `UNICODE` flag.

re.L**re.LOCALE**

`\w`、`\W`、`\b` および、`\B`、`\s` と `\S` を、現在のロケールに従わせます。

re.M**re.MULTILINE**

指定されると、パターン文字 '`^`' は、文字列の先頭および各行の先頭 (各改行の直後) とマッチします ; そしてパターン文字 '`$`' は文字列の末尾および各行の末尾 (改行の直前) とマッチします。デフォルトでは、'`^`' は、文字列の先頭とだけマッチし、'`$`' は、文字列の末尾および文字列の末尾の改行の直前 (がもしあれば) とマッチします。

re.S**re.DOTALL**

特殊文字 '`.`' を、改行を含む任意の文字と、とにかくマッチさせます ; このフラグがなければ、'`.`' は、改行以外の 任意の文字とマッチします。

re.U**re.UNICODE**

Make the `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` and `\S` sequences dependent on the Unicode character properties database. Also enables non-ASCII matching for `IGNORECASE`.

バージョン 2.0 で追加.

re.X**re.VERBOSE**

このフラグは正規表現を、パターンの論理的な節を視覚的に分割し、コメントを加えることで、見た目よく読みやすく書けるようにします。パターン中の空白は、文字クラス中にあるときと、エスケープされていないバックスラッシュの後にあるときと、`*?`、`(?:` や `(?P<...>` のようなトークン中を除いて無視されます。ある行が文字クラス中でもエスケープされていないバックスラッシュの後でもない `#` を含むなら、一番左のそのような `#` から行末までの全ての文字は無視されます。

つまり、数字にマッチする下記のふたつの正規表現オブジェクトは、機能的に等価です。:

```
a = re.compile(r"""\d + # the integral part
                \.    # the decimal point
                \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

re.search (*pattern*, *string*, *flags=0*)

string 全体を走査して、正規表現 *pattern* がマッチを発生する最初の位置を探して、対応する

`MatchObject` インスタンスを返します。もし文字列内に、そのパターンとマッチする位置がないならば、`None` を返します；これは、文字列内のある点で長さゼロのマッチを探すこととは異なることに注意して下さい。

`re.match(pattern, string, flags=0)`

もし `string` の先頭で 0 個以上の文字が正規表現 `pattern` とマッチすれば、対応する `MatchObject` インスタンスを返します。もし文字列がパターンとマッチしなければ、`None` を返します；これは長さゼロのマッチとは異なることに注意して下さい。

`MULTILINE` モードであっても、`re.match()` は文字列の先頭だけにマッチし、各行の先頭にはマッチしないことに注意してください。

`string` のどこにでもマッチさせたいければ、`search()` を使って下さい (`search()` vs. `match()` も参照してください)。

`re.split(pattern, string, maxsplit=0, flags=0)`

`string` を、`pattern` があるたびに分割します。もし括弧のキャプチャが `pattern` で使われていれば、パターン内のすべてのグループのテキストも結果のリストの一部として返されます。`maxsplit` がゼロでなければ、高々 `maxsplit` 個の分割が発生し、文字列の残りは、リストの最終要素として返されます。(非互換性ノート：オリジナルの Python 1.5 リリースでは、`maxsplit` は無視されていました。これはその後のリリースでは修正されました。)

```
>>> re.split('\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split('(\W+)', 'Words, words, words.')
['Words', ',', ' ', 'words', ',', ' ', 'words', ' ', '']
>>> re.split('\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

もし、捕捉するグループが分割パターンに含まれ、それが文字列の先頭にあるならば、分割結果は、空文字列から始まります。文字列最後においても同様です。

```
>>> re.split('(\W+)', '...words, words...')
['', '...', 'words', ',', ' ', 'words', '...', '']
```

その場合、常に、分割要素が、分割結果のリストの相対的なインデックスに現れます。(例えば、分割子の中に捕捉するグループが一つだけあれば、0 番目、2 番目、そして、4 番目です)

`split` は空のパターンマッチでは、文字列を分割しないことに注意して下さい。例えば:

```
>>> re.split('x*', 'foo')
['foo']
>>> re.split("(?m)^\$", "foo\n\nbar\n")
['foo\n\nbar\n']
```

バージョン 2.7 で変更: オプションの `flags` 引数が追加されました。

`re.findall(pattern, string, flags=0)`

string 中の *pattern* による全ての重複しないマッチを、文字列のリストとして返します。 *string* は左から右へ走査され、マッチは見つかった順で返されます。パターン中に 1 つ以上のグループがあれば、グループのリストを返します。パターンに複数のグループがあればタプルのリストになります。空マッチは結果に含まれます。

注釈: Due to the limitation of the current implementation the character following an empty match is not included in a next match, so `findall(r'^|\w+', 'two words')` returns `['', 'wo', 'words']` (note missed `"t"`). This is changed in Python 3.7.

バージョン 1.5.2 で追加.

バージョン 2.4 で変更: オプションの `flags` 引数が追加されました。

`re.finditer(pattern, string, flags=0)`

Return an *iterator* yielding *MatchObject* instances over all non-overlapping matches for the RE *pattern* in *string*. The *string* is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result. See also the note about *findall()*.

バージョン 2.2 で追加.

バージョン 2.4 で変更: オプションの `flags` 引数が追加されました。

`re.sub(pattern, repl, string, count=0, flags=0)`

string 内で、*pattern* と重複しないマッチの内、一番左にあるものを置換 *repl* で置換して得られた文字列を返します。もしパターンが見つからなければ、*string* を変更せずに返します。 *repl* は文字列でも関数でも構いません；もしそれが文字列であれば、それにある任意のバックスラッシュエスケープは処理されます。すなわち、`\n` は単一の改行文字に変換され、`\r` は、キャリッジリターンに変換されます、等々。`\j` のような未知のエスケープはそのままにされます。`\6` のような後方参照 (backreference) は、パターンのグループ 6 とマッチしたサブ文字列で置換されます。例えば:

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
...        r'static PyObject*\numpy_l(void)\n{',
...        'def myfunc():')
'static PyObject*\numpy_myfunc(void)\n{'
```

もし *repl* が関数であれば、重複しない *pattern* が発生するたびにその関数が呼ばれます。この関数は一つのマッチオブジェクト引数を取り、置換文字列を返します。例えば:

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro---gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
'Baked Beans & Spam'
```

パターンは、文字列でも RE オブジェクトでも構いません。

省略可能な引数 *count* は、置換されるパターンの出現回数の最大値です； *count* は非負の整数でなければなりません。もし省略されるかゼロであれば、出現したものがすべて置換されます。パターンのマッチが空であれば、以前のマッチと隣合わせでない時だけ置換されますので、`sub('x*', '-', 'abc')` は `'-a-b-c-'` を返します。

文字列タイプ *repl* 引数では、上で述べた文字エスケープや後方参照の他に、`\g<name>` は、`(?P<name>...)` シンタックスで定義された *name* グループによるマッチ部分文字列を使用することになりますし、`\g<number>` は対応するグループ番号への参照となります；`\g<2>` はつまり `\2` と等価ですが、`\g<2>0` のような置換においても曖昧になりません。`\20` は、グループ 20 への参照として解釈され、グループ 2 にリテラル文字 '0' が続いたものへの参照としては解釈されないかもしれません。後方参照 `\g<0>` は、RE とマッチするサブ文字列全体を置き換えます。

バージョン 2.7 で変更: オプションの *flags* 引数が追加されました。

`re.subn(pattern, repl, string, count=0, flags=0)`

`sub()` と同じ操作を行います、タプル (*new_string*, *number_of_subs_made*) を返します。

バージョン 2.7 で変更: オプションの *flags* 引数が追加されました。

`re.escape(pattern)`

Escape all the characters in *pattern* except ASCII letters and numbers. This is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it. For example:

```
>>> print re.escape('python.exe')
python\.exe

>>> legal_chars = string.ascii_lowercase + string.digits + "!#$%&'*+-.^_`|~:"
>>> print "[%s]+" % re.escape(legal_chars)
[abcdefghijklmnopqrstuvwxyz0123456789\!#\$\%\&\'\*\+\-\.^\_\`|\~\:]+

>>> operators = ['+', '-', '*', '/', '**']
>>> print '|'.join(map(re.escape, sorted(operators, reverse=True)))
\|\/\-\|\/\*\*\|\/\*
```

`re.purge()`

正規表現キャッシュをクリアします。

exception `re.error`

ここでの関数の一つに渡された文字列が、正しい正規表現ではない時 (例えば、その括弧が対になっていなかった)、あるいはコンパイルやマッチングの間になんらかのエラーが発生したとき、発生する例外です。たとえ文字列がパターンとマッチしなくても、決してエラーではありません。

7.2.3 正規表現オブジェクト

class `re.RegexObject`

RegexObject クラスは以下のメソッドと属性をサポートします:

`search(string[, pos[, endpos]])`

string を走査して、この正規表現がマッチする場所を探し、対応する *MatchObject* インスタンス

スを返します。string のどこにもマッチしない場合は `None` を返します。これは、string 内のどこかで長さ 0 でマッチした場合と異なることに注意してください。

省略可能な、2 つ目の引数 *pos* は、string のどこから探し始めるかを指定する index で、デフォルトでは 0 です。これは、文字列をスライスしてから検索するのと、完全には同じではありません。パターン文字 '`^`' は本当の文字列の先頭と、改行の直後にマッチしますが、検索を開始する index がマッチするとは限りません。

省略可能な引数 *endpos* は string のどこまでを検索するかを制限します。これは string の長さが *endpos* 文字だった場合と同じように動作します。つまり、*pos* から *endpos* - 1 の範囲の文字に対してパターンマッチします。*endpos* が *pos* よりも小さい場合は、マッチは見つかりません。それ以外の場合は、*rx* がコンパイルされた正規表現として、`rx.search(string, 0, 50)` は `rx.search(string[:50], 0)` と同じです。

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")          # Match at index 0
<_sre.SRE_Match object at ...>
>>> pattern.search("dog", 1)      # No match; search doesn't include the "d"
```

match(string[, pos[, endpos]])

もし *string* の先頭の 0 個以上の文字がこの正規表現とマッチすれば、対応する *MatchObject* インスタンスを返します。もし文字列がパターンとマッチしなければ、`None` を返します。これは長さゼロのマッチとは異なることに注意して下さい。

省略可能な引数 *pos* と *endpos* 引数は、*search()* メソッドと同じ意味を持ちます。

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")          # No match as "o" is not at the start of "dog"
<None>
>>> pattern.match("dog", 1)      # Match as "o" is the 2nd character of "dog".
<_sre.SRE_Match object at ...>
```

string のどこにでもマッチさせたいければ、代わりに *search()* を使って下さい (*search()* vs. *match()*) も参照してください)。

split(string, maxsplit=0)

split() 関数と同様で、コンパイルしたパターンを使います。ただし、*match()* と同じように、省略可能な *pos*, *endpos* 引数で検索範囲を指定することができます。

findall(string[, pos[, endpos]])

findall() 関数と同様で、コンパイルしたパターンを使います。ただし、*match()* と同じように、省略可能な *pos*, *endpos* 引数で検索範囲を指定することができます。

finditer(string[, pos[, endpos]])

finditer() 関数と同様で、コンパイルしたパターンを使います。ただし、*match()* と同じように、省略可能な *pos*, *endpos* 引数で検索範囲を指定することができます。

sub(repl, string, count=0)

sub() 関数と同様で、コンパイルしたパターンを使います。

subn (*repl*, *string*, *count=0*)

subn() 関数と同様で、コンパイルしたパターンを使います。

flags

正規表現のマッチングフラグです。これは *compile()* で指定されたフラグ、およびパターン内の (?...) インラインフラグとの組み合わせになります。

groups

パターンにあるキャプチャグループの数です。

groupindex

(?P<id>) で定義された任意の記号グループ名の、グループ番号への辞書マッピングです。もし記号グループがパターン内で何も使われていなければ、辞書は空です。

pattern

RE オブジェクトがそれからコンパイルされたパターン文字列です。

7.2.4 MatchObject オブジェクト

class `re.MatchObject`

マッチオブジェクトは常にブール値 `True` を持ちます。 *match()* と *search()* はマッチしなかった場合に `None` を返すので、単純な `if` ステートメントによってマッチしたかどうかをテストできます:

```
match = re.search(pattern, string)
if match:
    process(match)
```

マッチオブジェクトは以下のメソッドと属性をサポートしています:

expand (*template*)

テンプレート文字列 *template* に、 *sub()* メソッドがするようなバックスラッシュ置換をして得られる文字列を返します。 `\n` のようなエスケープは適当な文字に変換され、数値の後方参照 (`\1`, `\2`) と名前付きの後方参照 (`\g<1>`, `\g<name>`) は、対応するグループの内容で置き換えられます。

group ([*group1*, ...])

マッチした 1 個以上のサブグループを返します。もし引数で一つであれば、その結果は一つの文字列です。複数の引数があれば、その結果は、引数ごとに一項目を持つタプルです。引数がないければ、 *group1* はデフォルトでゼロです (マッチしたもののすべてが返されます)。もし *groupN* 引数がゼロであれば、対応する戻り値は、マッチする文字列全体です。もしそれが範囲 [1..99] 内であれば、それは、対応する丸括弧つきグループとマッチする文字列です。もしグループ番号が負であるか、あるいはパターンで定義されたグループの数より大きければ、 `IndexError` 例外が発生します。グループがマッチしなかったパターンの一部に含まれていれば、対応する結果は `None` です。グループが、複数回マッチしたパターンの一部に含まれていれば、最後のマッチが返されます。

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)           # The entire match
```

(次のページに続く)

(前のページからの続き)

```
'Isaac Newton'
>>> m.group(1)      # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)      # The second parenthesized subgroup.
'Newton'
>>> m.group(1, 2)   # Multiple arguments give us a tuple.
('Isaac', 'Newton')
```

もし正規表現が `(?P<name>...)` シンタックスを使うならば、`groupN` 引数は、それらのグループ名によってグループを識別する文字列であっても構いません。もし文字列引数がパターンのグループ名として使われていないものであれば、`IndexError` 例外が発生します。

適度に複雑な例題:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm ↵
↵Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

名前の付けられたグループは、そのインデックスによっても参照できます。

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

もし、グループが複数回マッチする場合、最後のマッチだけが利用可能となります。

```
>>> m = re.match(r"(..)+", "alb2c3") # Matches 3 times.
>>> m.group(1)                       # Returns only the last match.
'c3'
```

`groups([default])`

マッチの、1 からパターン内にある全グループ数までのすべてのサブグループを含むタプルを返します。`default` 引数は、マッチに加わらなかったグループ用に使われ、デフォルトでは `None` です。(非互換性ノート: オリジナルの Python 1.5 リリースでは、たとえタプルが一要素長であっても、その代わりに文字列を返していました。(1.5.1 以降の) 後のバージョンでは、そのような場合には、要素がひとつのタプルが返されます。)

例えば:

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

もし、整数部にのみ着目し、あとの部分をオプションとした場合、マッチの中に現れないグループがあるかも知れません。それらのグループは、`default` 引数が与えられていない場合、デフォルト

では `None` になります。

```
>>> m = re.match(r"(\d+)\.?(\\d+)?", "24")
>>> m.groups()           # Second group defaults to None.
('24', None)
>>> m.groups('0')       # Now, the second group defaults to '0'.
('24', '0')
```

`groupdict([default])`

マッチの、すべての 名前付きの サブグループを含む、サブグループ名でキー付けされた辞書を返します。 `default` 引数はマッチに加わらなかったグループに使われ、デフォルトでは `None` です。例えば、

```
>>> m = re.match(r"(?P<first_name>\\w+) (?P<last_name>\\w+)", "Malcolm ↵
↵Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

`start([group])`

`end([group])`

`group` とマッチした部分文字列の先頭と末尾のインデックスを返します。 `group` は、デフォルトでは (マッチした部分文字列全体を意味する) ゼロです。 `group` が存在してもマッチに寄与しなかった場合は、`-1` を返します。マッチオブジェクト `m` および、マッチに寄与しなかったグループ `g` があって、グループ `g` とマッチしたサブ文字列 (`m.group(g)` と同じ意味ですが) は:

```
m.string[m.start(g):m.end(g)]
```

です。もし `group` が空文字列とマッチすれば、`m.start(group)` が `m.end(group)` と等しくなることに注意して下さい。例えば、`m = re.search('b(c?)', 'cba')` とすると、`m.start(0)` は `1` で、`m.end(0)` は `2` であり、`m.start(1)` と `m.end(1)` はともに `2` であり、`m.start(2)` は `IndexError` 例外を発生します。

例として、電子メールのアドレスから `remove_this` を取り除く場合を示します。

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

`span([group])`

`MatchObject m` について、大きさ 2 のタプル (`m.start(group)`, `m.end(group)`) を返します。もし `group` がマッチに寄与しなかったら、これは `(-1, -1)` です。また `group` はデフォルトでゼロです。

`pos`

`RegexObject` の `search()` か `match()` に渡された `pos` の値です。これは RE エンジンがマッチを探し始める位置の文字列のインデックスです。

`endpos`

`RegexObject` の `search()` か `match()` に渡された `endpos` の値です。これは RE エンジンがそれ以上は進まない位置の文字列のインデックスです。

lastindex

最後にマッチした取り込みグループの整数インデックスです。もしどのグループも全くマッチしなければ `None` です。例えば、`(a)b`, `((a)(b))` や `((ab))` といった表現が `'ab'` に適用された場合、`lastindex == 1` となり、同じ文字列に `(a)(b)` が適用された場合には `lastindex == 2` となるでしょう。

lastgroup

最後にマッチした取り込みグループの名前です。もしグループに名前がないか、あるいはどのグループも全くマッチしなければ `None` です。

re

この `MatchObject` インスタンスを `match()` あるいは `search()` メソッドで生成した正規表現オブジェクトです。

string

`match()` あるいは `search()` に渡された文字列です。

7.2.5 例

ペアの確認

この例では、マッチオブジェクトの表示を少し美しくするために、下記の補助関数を使用します：

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

あなたがポーカープログラムを書いているとします。プレイヤーの持ち札はそれぞれの文字が 1 枚のカードを意味する 5 文字の文字列によって表現されます。`"a"` はエース、`"k"` はキング、`"q"` はクイーン、`"j"` はジャック、`"t"` は 10、そして `"2"` から `"9"` はそれぞれの数字のカードを表します。

与えられた文字列が、持ち札として有効かを確認するために、下記のようにするかも知れません。：

```
>>> valid = re.compile(r"^[a2-9tjqk]{5}$")
>>> displaymatch(valid.match("akt5q")) # Valid.
"<Match: 'akt5q', groups=()>"
>>> displaymatch(valid.match("akt5e")) # Invalid.
>>> displaymatch(valid.match("akt")) # Invalid.
>>> displaymatch(valid.match("727ak")) # Valid.
"<Match: '727ak', groups=()>"
```

最後の持ち札 `"727ak"` は、ペアを含んでいます。言い換えると同じ値のカードが 2 枚あります。これを正規表現にマッチさせるために、後方参照を使う場合もあります：

```
>>> pair = re.compile(r"*(.*)*\1")
>>> displaymatch(pair.match("717ak"))      # Pair of 7s.
"<Match: '717', groups=('7',)>"
>>> displaymatch(pair.match("718ak"))      # No pairs.
>>> displaymatch(pair.match("354aa"))      # Pair of aces.
"<Match: '354aa', groups=('a',)>"
```

どのカードのペアになっているかを調べるため、下記のように `MatchObject` の `group()` メソッドを使う場合があります。

```
>>> pair.match("717ak").group(1)
'7'

# Error because re.match() returns None, which doesn't have a group() method:
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r"*(.*)*\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'

>>> pair.match("354aa").group(1)
'a'
```

scanf() をシミュレートする

Python には現在のところ、`scanf()` に相当するものはありません。正規表現は、`scanf()` のフォーマット文字列よりも、一般的により強力であり、また冗長でもあります。以下の表に、`scanf()` のフォーマットトークンと正規表現の大体同等な対応付けを示します。

scanf() トークン	正規表現
%c	.
%5c	.{5}
%d	[−+]? \d+
%e, %E, %f, %g	[−+]? (\d+ (\.\d*)? \.\d+) ([eE] [−+]? \d+)?
%i	[−+]? (0[xX] [\dA-Fa-f] + 0[0-7] * \d+)
%o	[−+]? [0-7] +
%s	\S+
%u	\d+
%x, %X	[−+]? (0[xX])? [\dA-Fa-f] +

以下のような文字列からファイル名と数値を抽出することを考えます

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

このように `scanf()` フォーマットを使うでしょう

```
%s - %d errors, %d warnings
```

同等な正規表現はこのようなものとなります

```
(\S+) - (\d+) errors, (\d+) warnings
```

search() vs. match()

Python は正規表現ベースの 2 個の基本的な関数、文字列の先頭でのみのマッチを確認する `re.match()` および、文字列内の位置にかかわらずマッチを確認する `re.search()` (Perl でのデフォルトの挙動) を提供しています。

例えば:

```
>>> re.match("c", "abcdef")      # No match
>>> re.search("c", "abcdef")     # Match
<_sre.SRE_Match object at ...>
```

'^' で始まる正規表現は、`search()` において、マッチを文字列の先頭からに制限するために使用します:

```
>>> re.match("c", "abcdef")      # No match
>>> re.search("^c", "abcdef")    # No match
>>> re.search("a", "abcdef")     # Match
<_sre.SRE_Match object at ...>
```

ただし、`MULTILINE` モードの `match()` では文字列の先頭にのみマッチするのに対し、正規表現に '^' を使った `search()` では各行の先頭にもマッチします。

```
>>> re.match('X', 'A\nB\nX', re.MULTILINE) # No match
>>> re.search('X', 'A\nB\nX', re.MULTILINE) # Match
<_sre.SRE_Match object at ...>
```

電話帳の作成

`split()` は文字列を与えられたパターンで分割し、リストにして返します。下記の、電話帳作成の例のように、このメソッドはテキストデータを読みやすくしたり、Python で編集したりしやすくする際に、非常に役に立ちます。

最初に、入力を示します。通常、これはファイルからの入力になるでしょう。ここでは、3 重引用符の書式とします:

```
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street
...
... Ronald Heathmore: 892.345.3428 436 Finley Avenue
... Frank Burger: 925.541.7625 662 South Dogwood Way
...
...
... Heather Albrecht: 548.326.4584 919 Park Place"""
```


個々の記録は、1 つ以上の改行で区切られています。まずは、文字列から空行を除き、記録ごとのリストに変換しましょう。

```
>>> entries = re.split("\n+", text)
>>> entries
['Ross McFluff: 834.345.1254 155 Elm Street',
 'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
 'Frank Burger: 925.541.7625 662 South Dogwood Way',
 'Heather Albrecht: 548.326.4584 919 Park Place']
```

そして、各記録を、名、姓、電話番号、そして、住所に分割してリストにします。分割のためのパターンに使っている空白文字が、住所には含まれるため、`split()` の `maxsplit` 引数を使います。:

```
>>> [re.split("?: ", entry, 3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
 ['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],
 ['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],
 ['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

パターン、`?:` は姓に続くコロンにマッチします。そのため、コロンは分割結果のリストには現れません。`maxsplit` を 4 にすれば、ハウスナンバーと、ストリート名を分割することができます。:

```
>>> [re.split("?: ", entry, 4) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'],
 ['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue'],
 ['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood Way'],
 ['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]
```

テキストの秘匿

`sub()` はパターンにマッチした部分を文字列や関数の返り値で置き換えます。この例では、“秘匿”する文字列に、関数と共に `sub()` を適用する例を示します。言い換えると、最初と最後の文字を除く、単語中の文字の位置をランダム化します。

```
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + "".join(inner_word) + m.group(3)
>>> text = "Professor Abdolmalek, please report your absences promptly."
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Poefsrosr Aealmlobdk, pslae reorpt your abnseces plmrptoy.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reorpt yuor asnebces potlmpy.'
```

全ての副詞を見つける

`findall()` matches *all* occurrences of a pattern, not just the first one as `search()` does. For example, if a writer wanted to find all of the adverbs in some text, they might use `findall()` in the following manner:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly", text)
['carefully', 'quickly']
```

全ての副詞と、その位置を見つける

If one wants more information about all matches of a pattern than the matched text, `finditer()` is useful as it provides instances of `MatchObject` instead of strings. Continuing with the previous example, if a writer wanted to find all of the adverbs *and their positions* in some text, they would use `finditer()` in the following manner:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly", text):
...     print '%02d-%02d: %s' % (m.start(), m.end(), m.group(0))
07-16: carefully
40-47: quickly
```

Raw String 記法

Raw string 記法 (`r"text"`) により、バックスラッシュ (`'\'`) を個々にバックスラッシュでエスケープすることなしに、正規表現を正常な状態に保ちます。例えば、下記の 2 つのコードは機能的に等価です。:

```
>>> re.match(r"\W(.)\1\W", " ff ")
<_sre.SRE_Match object at ...>
>>> re.match("\\W(.)\\1\\W", " ff ")
<_sre.SRE_Match object at ...>
```

文字通りのバックスラッシュにマッチさせたいなら、正規表現中ではエスケープする必要があります。Raw string 記法では、`r"\\"` ということになります。Raw string 記法を用いない場合、`"\\\\"` としなくてはなりません。下記のコードは機能的に等価です。:

```
>>> re.match(r"\\", r"\\")
<_sre.SRE_Match object at ...>
>>> re.match("\\\\", r"\\")
<_sre.SRE_Match object at ...>
```

7.3 struct — 文字列データをパックされたバイナリデータとして解釈する

このモジュールは、Python の値と Python 上で文字列データとして表される C の構造体データとの間の変換を実現します。このモジュールは特に、ファイルに保存されたり、ネットワーク接続を経由したバイナリデータを扱うときに使われます。このモジュールでは、C 構造体のレイアウトおよび Python の値との間で行いたい変換をコンパクトに表現するために、`フォーマット文字列` を使います。

注釈: デフォルトでは、与えられた C の構造体をパックする際に、関連する C データ型を適切にアラインメ

ント (alignment) するために数バイトのパディングを行うことがあります。この挙動が選択されたのは、パックされた構造体のバイト表現に対応する C 構造体のメモリレイアウトに正確に対応させるためです。プラットフォーム独立のデータフォーマットを扱ったり、隠れたパディングを排除したりするには、サイズ及びアラインメントとして `native` の代わりに `standard` を使うようにします: 詳しくは [バイトオーダー](#)、[サイズ](#)、[アラインメント](#) を参照して下さい。

7.3.1 関数と例外

このモジュールは以下の例外と関数を定義しています:

exception `struct.error`

様々な状況で送出された例外です; 引数は何が問題かを記述する文字列です。

`struct.pack(fmt, v1, v2, ...)`

フォーマット文字列 `fmt` に従い値 `v1, v2, ...` をパックして、文字列で返します。引数は指定したフォーマットが要求する型と正確に一致していなければなりません。

`struct.pack_into(fmt, buffer, offset, v1, v2, ...)`

フォーマット文字列 `fmt` に従い値 `v1, v2, ...` をパックして文字列にし、書き込み可能な `buffer` のオフセット `offset` 位置より書き込みます。オフセットは省略出来ません。

バージョン 2.5 で追加.

`struct.unpack(fmt, string)`

(おそらく `pack(fmt, ...)` でパックされた) バッファ `buffer` から与えられた書式文字列 `fmt` に従ってアンパックします。値が一つしかない場合を含め、結果はタプルで返されます。文字列データにはフォーマットが要求するだけのデータが正確に含まれていなければなりません (`len(bytes)` が `calcsize(fmt)` と一致しなければなりません)。

`struct.unpack_from(fmt, buffer[, offset=0])`

バッファ `buffer` の `offset` の位置から、書式化文字列 `fmt` に従ってアンパックします。結果は要素が1つでもタプルとして返されます。`buffer` は少なくとも書式が要求するデータサイズを持っていないけません (`len(buffer[offset:])` は少なくとも `calcsize(fmt)` 以上でなくてはなりません)。

バージョン 2.5 で追加.

`struct.calcsize(fmt)`

与えられたフォーマットに対応する構造体のサイズ (すなわち文字列データのサイズ) を返します。

7.3.2 フォーマット文字列

フォーマット文字列はデータをパックしたりアンパックしたりするときの期待されるレイアウトを指定するためのメカニズムです。文字列はパック/アンパックされるデータの型を指定する [フォーマット文字](#) から組み立てられます。さらに、[バイトオーダー](#)、[サイズ](#)、[アラインメント](#) を制御するための特殊文字もあります。

バイトオーダー、サイズ、アラインメント

デフォルトでは、C での型はマシンのネイティブ (native) の形式およびバイトオーダー (byte order) で表され、適切にアラインメント (alignment) するために、必要に応じて数バイトのパディングを行ってスキップします (これは C コンパイラが用いるルールに従います)。

これに代わって、フォーマット文字列の最初の文字を使って、バイトオーダーやサイズ、アラインメントを指定することができます。指定できる文字を以下のテーブルに示します:

文字	バイトオーダー	サイズ	アラインメント
@	native	native	native
=	native	standard	none
<	リトルエンディアン	standard	none
>	ビッグエンディアン	standard	none
!	ネットワーク (= ビッグエンディアン)	standard	none

フォーマット文字列の最初の文字が上のいずれかでない場合、'@' であるとみなされます。

ネイティブのバイトオーダーはビッグエンディアンかリトルエンディアンで、ホスト計算機に依存します。例えば、Intel x86 および AMD64 (x86-64) はリトルエンディアンです。Motorola 68000 および PowerPC G5 はビッグエンディアンです。ARM および Intel Itanium はエンディアンを切り替えられる機能を備えています (バイエンディアン)。使っているシステムでのエンディアンは `sys.byteorder` を使って調べて下さい。

ネイティブのサイズおよびアラインメントは C コンパイラの `sizeof` 式で決定されます。ネイティブのサイズおよびアラインメントはネイティブのバイトオーダーと同時に使われます。

標準のサイズはフォーマット文字だけで決まります。 [フォーマット文字](#) の表を参照して下さい。

'@' と '=' の違いに注意してください: 両方ともネイティブのバイトオーダーですが、後者のバイトサイズとアラインメントは標準のものに合わせてあります。

'!' 表記法はネットワークバイトオーダーがビッグエンディアンかリトルエンディアンか忘れちゃったという熱意に乏しい人向けに用意されています。

バイトオーダーに関して、「(強制的にバイトスワップを行う) ネイティブの逆」を指定する方法はありません。'<' または '>' のうちふさわしい方を選んでください。

注釈:

- (1) パディングは構造体のメンバの並びの中にだけ自動で追加されます。最初や最後にパディングが追加されることはありません。
- (2) ネイティブでないサイズおよびアラインメントが使われる場合にはパディングは行われません (たとえば '<', '>', '=', '!' を使った場合です)。
- (3) 特定の型によるアラインメント要求に従うように構造体の末端をそろえるには、繰り返し回数をゼロにした特定の型でフォーマットを終端します。 [例](#) を参照して下さい。

フォーマット文字

フォーマット文字 (format character) は以下の意味を持っています; C と Python の間の変換では、値は正確に以下に指定された型でなくてはなりません: 「標準のサイズ」列は standard サイズ使用時にパックされた値が何バイトかを示します。つまり、フォーマット文字列が '<', '>', '!', '=' のいずれかで始まっている場合のものです。native サイズ使用時にはパックされた値の大きさはプラットフォーム依存です。

フォーマット	C の型	Python の型	標準のサイズ	注釈
x	パディングバイト	値なし		
c	char	長さ 1 の文字列	1	
b	signed char	integer	1	(3)
B	unsigned char	integer	1	(3)
?	_Bool	真偽値型 (bool)	1	(1)
h	short	integer	2	(3)
H	unsigned short	integer	2	(3)
i	int	integer	4	(3)
I	unsigned int	integer	4	(3)
l	long	integer	4	(3)
L	unsigned long	integer	4	(3)
q	long long	integer	8	(2), (3)
Q	unsigned long long	integer	8	(2), (3)
f	float	float	4	(4)
d	double	float	8	(4)
s	char[]	string		
p	char[]	string		
P	void *	integer		(5), (3)

注釈:

- (1) '?' 変換コードは C99 で定義された `_Bool` 型に対応します。その型が利用できない場合は、`char` で代用されます。標準モードでは常に 1 バイトで表現されます。

バージョン 2.6 で追加。

- (2) 変換コード 'q' および 'Q' は、ネイティブモードではプラットフォームの C コンパイラが C の `long long` 型をサポートする場合、または Windows では `_int64` をサポートする場合にのみ利用できます。標準モードでは常に利用できます。

バージョン 2.2 で追加。

- (3) 整数変換コードで非整数をパックしようとするとき、その非整数が `__index__()` メソッドを持っていると、パッキングの前に、そのメソッドが変数を整数に変換するために呼び出されます。`__index__()` メソッドが存在しないか、`__index__()` メソッドの呼び出しが `TypeError` を送出したら、次に `__int__()` メソッドが試されます。しかし、`__int__()` の使用は非推奨で、`DeprecationWarning` を送出します。

バージョン 2.7 で変更: 非整数に対して `__index__()` メソッドが使われるようになったのは 2.7 の新機能です。

バージョン 2.7 で変更: バージョン 2.7 以前では、すべての整数変換コードが変換に `__int__()` メソッドを使うわけではなく、浮動小数点の引数にのみ `DeprecationWarning` が送出されていました。

- (4) 'f' および 'd' 変換コードについて、パックされた表現は IEEE 754 binary32 ('f' の場合) または binary64 ('d' の場合) フォーマットが、プラットフォームにおける浮動小数点数のフォーマットに関係なく使われます。
- (5) 'P' フォーマット文字はネイティブバイトオーダーでのみ利用可能です (デフォルトのネットワークバイトオーダーに設定するか、'@' バイトオーダー指定文字を指定しなければなりません)。'=' を指定した場合、ホスト計算機のバイトオーダーに基づいてリトルエンディアンとビッグエンディアンのどちらを使うかを決めます。struct モジュールはこの設定をネイティブのオーダー設定として解釈しないので、'P' を使うことはできません。

フォーマット文字の前に整数をつけ、繰り返し回数 (count) を指定することができます。例えば、フォーマット文字列 '4h' は 'hhhh' と全く同じ意味です。

フォーマット文字間の空白文字は無視されます; count とフォーマット文字の間にはスペースを入れてはいけません。

フォーマット文字 's' では、count は文字列のサイズとして扱われます。他のフォーマット文字のように繰り返し回数ではありません; 例えば、'10c' が 10 個のキャラクタを表すのに対して、'10s' は 10 バイトの長さを持った 1 個の文字列です。count の指定がない場合のデフォルトは 1 です。文字列をパックする際には、指定した長さにフィットするように、必要に応じて切り詰められたりヌル文字で穴埋めされたりします。アンパック時は、取り出される文字列は正確に指定サイズのサイズを持ちます。また特殊なケースとして、('0c' が 0 個のキャラクタを表すのに対して) '0s' は 1 個の空文字列を意味します。

フォーマット文字 'p' は "Pascal 文字列 (pascal string)" をコードします。Pascal 文字列は count で与えられる固定長のバイト列に収められた短い可変長の文字列です。このデータの先頭の 1 バイトには文字列の長さか 255 のうち、小さい方の数が収められます。その後に文字列のバイトデータが続きます。pack() に渡された Pascal 文字列の長さが長すぎた (count-1 よりも長い) 場合、先頭の count-1 バイトが書き込まれます。文字列が count-1 よりも短い場合、指定した count バイトに達するまでの残りの部分はヌルで埋められます。unpack() では、フォーマット文字 'p' は指定された count バイトだけデータを読み込みますが、返される文字列は決して 255 文字を超えることはないので注意してください。

フォーマット文字 'P' では、返される値は Python 整数型または long 整数型で、これはポインタの値を Python での整数にキャストする際に、値を保持するために必要なサイズに依存します。NULL ポインタは常に Python 整数型の 0 になります。ポインタ型のサイズを持った値をパックする際には、Python 整数型か long 整数型オブジェクトのどちらかが使われます。例えば、Alpha および Merced プロセッサは 64 bit のポインタ値を使いますが、これはポインタを保持するために Python long 整数型が使われることを意味します; 32 bit ポインタを使う他のプラットフォームでは Python 整数型が使われます。

'?' フォーマット文字では、返り値は `True` または `False` です。パックするときには、引数オブジェクトの論理値としての値が使われます。0 または 1 のネイティブや標準の真偽値表現でパックされ、アンパックされるときはゼロでない値は `True` になります。

例

注釈: 全ての例は、ビッグエンディアンのマシンで、ネイティブのバイトオーダー、サイズおよびアラインメントを仮定します。

基本的な例として、三つの整数をパック/アンパックします:

```
>>> from struct import *
>>> pack('hhl', 1, 2, 3)
'\x00\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('hhl', '\x00\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('hhl')
8
```

アンパックした結果のフィールドは、変数に割り当てるか named tuple でラップすることによって名前を付けることができます:

```
>>> record = 'raymond \x32\x12\x08\x01\x08'
>>> name, serialnum, school, gradelevel = unpack('<10sHHb', record)

>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name serialnum school gradelevel')
>>> Student._make(unpack('<10sHHb', record))
Student(name='raymond ', serialnum=4658, school=264, gradelevel=8)
```

アラインメントの要求を満たすために必要なパディングが異なるという理由により、フォーマット文字の順番がサイズの違いを生み出すことがあります:

```
>>> pack('ci', '*', 0x12131415)
'*\x00\x00\x00\x12\x13\x14\x15'
>>> pack('ic', 0x12131415, '*')
'\x12\x13\x14\x15*'
>>> calcsize('ci')
8
>>> calcsize('ic')
5
```

以下のフォーマット 'llh01' は、long 型が 4 バイトを境界としてそろえられていると仮定して、末端に 2 バイトをパディングします:

```
>>> pack('llh01', 1, 2, 3)
'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

この例はネイティブのサイズとアラインメントが使われているときだけ思った通りに動きます。標準のサイズとアラインメントはアラインメントの設定ではいかなるアラインメントも行いません。

参考:

`array` モジュール 一様なデータ型からなるバイナリ記録データのパック。

`xdrlib` モジュール XDR データのパックおよびアンパック。

7.3.3 クラス

`struct` モジュールは次の型を定義します:

class `struct.Struct` (*format*)

フォーマット文字列 *format* に従ってバイナリデータを読み書きする、新しい Struct オブジェクトを返します。Struct オブジェクトを一度作ってからそのメソッドを使うと、フォーマット文字列のコンパイルが一度で済むので、`struct` モジュールの関数を同じフォーマットで何度も呼び出すよりも効率的です。

バージョン 2.5 で追加.

コンパイルされた Struct オブジェクトは以下のメソッドと属性をサポートします:

pack (*v1*, *v2*, ...)

`pack()` 関数と同じ、コンパイルされたフォーマットを利用するメソッドです。(`len(result)` は `self.size` と等しいでしょう)

pack_into (*buffer*, *offset*, *v1*, *v2*, ...)

`pack_into()` 関数と同じ、コンパイルされたフォーマットを利用するメソッドです。

unpack (*string*)

`unpack()` 関数と同じ、コンパイルされたフォーマットを利用するメソッドです。
(`len(string)` は `self.size` と等しくなければなりません)。

unpack_from (*buffer*, *offset*=0)

`unpack_from()` 関数と同じ、コンパイルされたフォーマットを利用するメソッドです。
(`len(buffer[offset:])` は `self.size` 以上でなければなりません)。

format

この Struct オブジェクトを作成する時に利用されたフォーマット文字列です。

size

format に対応する struct (とそれによる文字列) のサイズを計算したものです。

7.4 difflib — 差分の計算を助ける

バージョン 2.1 で追加.

このモジュールは、シーケンスを比較するためのクラスや関数を提供しています。例えば、ファイルの差分を計算して、それを HTML や context diff, unified diff などいろいろなフォーマットで出力するために、このモジュールを利用することができます。ディレクトリやファイル群を比較するためには、`filecmp` モジュールも参照してください。

class difflib.SequenceMatcher

柔軟性のあるクラスで、二つのシーケンスの要素は、ハッシュ化できる (*hashable*) 型である限り何でも比較可能です。基本的なアルゴリズムは、1980 年代の後半に発表された Ratcliff と Obershelp による”ゲシュタルトパターンマッチング”と大げさに名づけられたアルゴリズム以前から知られている、やや凝ったアルゴリズムです。その考え方は、”junk” 要素を含まない最も長い互いに隣接したマッチ列を探すことです (Ratcliff と Obershelp のアルゴリズムでは junk を示しません)。このアイデアは、マッチ列から左または右に伸びる断片に対して再帰的にあてはめられます。この方法では編集を最小にする列は取り出されませんが、人間の目からみて「正しい感じ」にマッチする傾向があります。

実行時間: 基本的な Ratcliff-Obershelp アルゴリズムは、最悪の場合 3 乗、期待値で 2 乗となります。*SequenceMatcher* オブジェクトでは、最悪のケースで 2 乗、期待値は比較されるシーケンス中に共通に現れる要素数に非常にややこしく依存しています。最良の場合は線形時間になります。

自動 junk ヒューリスティック: *SequenceMatcher* は、シーケンスの特定の要素を自動的に junk として扱うヒューリスティックをサポートしています。このヒューリスティックは、各個要素がシーケンス内に何回現れるかを数えます。ある要素の重複数が (最初のものは除いて) 合計でシーケンスの 1% 以上になり、そのシーケンスが 200 要素以上なら、その要素は ”popular” であるものとしてマークされ、シーケンスのマッチングの目的からは junk として扱われます。このヒューリスティックは、*SequenceMatcher* の作成時に *autojunk* パラメータを *False* に設定することで無効化できます。

バージョン 2.7.1 で追加: *autojunk* パラメータ。

class difflib.Differ

テキスト行からなるシーケンスを比較するクラスです。人が読むことのできる差分を作成します。*Differ* クラスは *SequenceMatcher* クラスを利用して、行からなるシーケンスを比較したり、(ほぼ) 同一の行内の文字を比較したりします。

Differ クラスによる差分の各行は、2 文字のコードで始まります:

コード	意味
'- '	行はシーケンス 1 にのみ存在する
'+' '	行はシーケンス 2 にのみ存在する
' ' '	行は両方のシーケンスで同一
'?' '	行は入力シーケンスのどちらにも存在しない

'?' で始まる行は、行内のどこに差異が存在するかに注意を向けようとします。その行は、入力されたシーケンスのどちらにも存在しません。シーケンスがタブ文字を含むとき、これらの行は判別しづらいものになることがあります。

class difflib.HtmlDiff

このクラスは、二つのテキストを左右に並べて比較表示し、行間あるいは行内の変更点を強調表示するような HTML テーブル (またはテーブルの入った完全な HTML ファイル) を生成するために使います。テーブルは完全差分モード、コンテキスト差分モードのいずれでも生成できます。

このクラスのコンストラクタは以下のようになっています:

```
__init__(tabsize=8, wrapcolumn=None, linejunk=None, charjunk=IS_CHARACTER_JUNK)
```

`HtmlDiff` のインスタンスを初期化します。

`tabsize` はオプションのキーワード引数で、タブストップ幅を指定します。デフォルトは 8 です。

`wrapcolumn` はオプションのキーワード引数で、テキストを折り返すカラム幅を指定します。デフォルトは `None` で折り返しを行いません。

`linejunk` および `charjunk` はオプションのキーワード引数で、`ndiff()` (`HtmlDiff` はこの関数を使って左右のテキストの差分を HTML で生成します) に渡されます。それぞれの引数のデフォルト値および説明は `ndiff()` のドキュメントを参照してください。

以下のメソッドが public になっています:

make_file (*fromlines*, *tolines* [, *fromdesc*][, *todesc*][, *context*][, *numlines*])

fromlines と *tolines* (いずれも文字列のリスト) を比較し、行間または行内の変更点が強調表示された行差分の入った表を持つ完全な HTML ファイルを文字列で返します。

fromdesc および *todesc* はオプションのキーワード引数で、差分表示テーブルにおけるそれぞれ差分元、差分先ファイルのカラムのヘッダになる文字列を指定します (いずれもデフォルト値は空文字列です)。

context および *numlines* はともにオプションのキーワード引数です。*context* を `True` にするとコンテキスト差分を表示し、デフォルトの `False` にすると完全なファイル差分を表示します。*numlines* のデフォルト値は 5 で、*context* が `True` の場合、*numlines* は強調部分の前後にあるコンテキスト行の数を制御します。*context* が `False` の場合、*numlines* は "next" と書かれたハイパーリンクをたどった時に到達する場所が次の変更部分より何行前にあるかを制御します (値をゼロにした場合、"next" ハイパーリンクを辿ると変更部分の強調表示がブラウザの最上部に表示されるようになります)。

make_table (*fromlines*, *tolines* [, *fromdesc*][, *todesc*][, *context*][, *numlines*])

fromlines と *tolines* (いずれも文字列のリスト) を比較し、行間または行内の変更点が強調表示された行差分の入った完全な HTML テーブルを文字列で返します。

このメソッドの引数は、`make_file()` メソッドの引数と同じです。

`Tools/scripts/diff.py` はこのクラスへのコマンドラインフロントエンドで、使い方を学ぶ上で格好の例題が入っています。

バージョン 2.4 で追加。

`difflib.context_diff` (*a*, *b* [, *fromfile*][, *tofile*][, *fromfiledate*][, *tofiledate*][, *n*][, *lineterm*])

a と *b* (文字列のリスト) を比較し、差分 (差分形式の行を生成するジェネレータ (*generator*)) を、`context diff` のフォーマット (以下「コンテキスト形式」) で返します。

コンテキスト形式は、変更があった行に前後数行を加えてある、コンパクトな表現方法です。変更箇所は、変更前/変更後に分けて表します。コンテキスト (変更箇所前後の行) の行数は *n* で指定し、デフォルト値は 3 です。

デフォルトでは、`diff` の制御行 (`***` や `---` を含む行) の最後には、改行文字が付加されます。この場合、入出力とも、行末に改行文字を持つので、`file.readlines()` で得た入力から生成した差分を、`file.writelines()` に渡す場合に便利です。

行末に改行文字を持たない入力に対しては、出力でも改行文字を付加しないように *lineterm* 引数に "" を渡してください。

コンテキスト形式は、通常、ヘッダにファイル名と変更時刻を持っています。この情報は、文字列 *fromfile*, *tofile*, *fromfiledate*, *tofiledate* で指定できます。変更時刻の書式は、通常、ISO 8601 フォーマットで表されます。指定しなかった場合のデフォルト値は、空文字列です。

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> for line in context_diff(s1, s2, fromfile='before.py', tofile='after.py'):
...     sys.stdout.write(line)
*** before.py
--- after.py
*****
*** 1,4 ****
! bacon
! eggs
! ham
! guido
--- 1,4 ----
! python
! eggy
! hamster
! guido
```

より詳細な例は、[difflib](#) のコマンドラインインタフェース を参照してください。

バージョン 2.3 で追加。

`difflib.get_close_matches(word, possibilities[, n][, cutoff])`

「十分」なマッチの上位のリストを返します。 *word* はマッチさせたいシーケンス (大概是文字列) です。 *possibilities* は *word* にマッチさせるシーケンスのリスト (大概是文字列のリスト) です。

オプションの引数 *n* (デフォルトでは 3) はメソッドの返すマッチの最大数です。 *n* は 0 より大きくなければなりません。

オプションの引数 *cutoff* (デフォルトでは 0.6) は、区間 [0, 1] に入る小数の値です。 *word* との一致率がそれ未満の *possibilities* の要素は無視されます。

possibilities の要素でマッチした上位 (多くても *n* 個) は、類似度のスコアに応じて (一番似たものを先頭に) ソートされたリストとして返されます。

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('apple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

`difflib.ndiff(a, b[, linejunk], charjunk)`

a と *b* (文字列のリスト) を比較し、差分 (差分形式の行を生成するジェネレータ (*generator*)) を、*Differ* のスタイルで返します。

オプションのキーワードパラメータ *linejunk* と *charjunk* は、フィルタ関数を渡します (使わないときは *None*):

linejunk: 文字列型の引数ひとつを受け取る関数で、文字列が junk ならば真を、違うときには偽を返します。Python 2.3 以降、デフォルトでは (*None*) になります。それまでは、モジュールレベルの関数 *IS_LINE_JUNK()* であり、それは高々ひとつのシャープ記号 ('#') を除いて可視のキャラクタを含まない行をフィルタリングするものです。Python 2.3 から、下位にある *SequenceMatcher* クラスが、雑音となるくらい頻繁に登場する行であるか否かを、動的に分析します。これは、バージョン 2.3 以前のデフォルト値よりたいていうまく動作します。

charjunk: 文字 (長さ 1 の文字列) を受け取る関数です。デフォルトでは、モジュールレベルの関数 *IS_CHARACTER_JUNK()* であり、これは空白文字類 (空白またはタブ、注: 改行文字をこれに含めるのは悪いアイデア!) をフィルタリングします。

`Tools/scripts/ndiff.py` は、この関数のコマンドラインのフロントエンド (インターフェイス) です。

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(1),
...              'ore\ntree\nemu\n'.splitlines(1))
>>> print ''.join(diff),
- one
?  ^
+ ore
?  ^
- two
- three
?  -
+ tree
+ emu
```

`difflib.restore(sequence, which)`

差分を生成した元の二つのシーケンスのうち一つを返します。

Differ.compare() または *ndiff()* によって生成された *sequence* を与えられると、行頭のプレフィクスを取りのぞいてファイル 1 または 2 (引数 *which* で指定される) に由来する行を復元します。

例:

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(1),
...              'ore\ntree\nemu\n'.splitlines(1))
>>> diff = list(diff) # materialize the generated delta into a list
>>> print ''.join	restore(diff, 1)),
one
two
three
>>> print ''.join	restore(diff, 2)),
```

(次のページに続く)

(前のページからの続き)

```
ore
tree
emu
```

`difflib.unified_diff(a, b[, fromfile][, tofile][, fromfiledate][, tofiledate][, n][, lineterm])`

a と *b* (文字列のリスト) を比較し、差分 (差分形式の行を生成するジェネレータ (*generator*)) を、unified diff フォーマット (以下「ユニファイド形式」) で返します。

ユニファイド形式は変更があった行にコンテキストとなる前後数行を加えた、コンパクトな表現方法です。変更箇所は (変更前/変更後を分離したブロックではなく) インラインスタイルで表されます。コンテキストの行数は、*n* で指定し、デフォルト値は 3 です。

デフォルトでは、diff の制御行 (---, +++, @@ を含む行) は行末の改行を含めて生成されます。このようにしてあると、入出力とも行末に改行文字を持つので、`file.readlines()` で得た入力进行处理して生成した差分を、`file.writelines()` に渡す場合に便利です。

行末に改行文字を持たない入力に対しては、出力でも改行文字を付加しないように *lineterm* 引数に "" を渡してください。

コンテキスト形式は、通常、ヘッダにファイル名と変更時刻を持っています。この情報は、文字列 *fromfile*, *tofile*, *fromfiledate*, *tofiledate* で指定できます。変更時刻の書式は、通常、ISO 8601 フォーマットで表されます。指定しなかった場合のデフォルト値は、空文字列です。

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> for line in unified_diff(s1, s2, fromfile='before.py', tofile='after.py'):
...     sys.stdout.write(line)
--- before.py
+++ after.py
@@ -1,4 +1,4 @@
-bacon
-eggs
-ham
+python
+eggy
+hamster
 guido
```

より詳細な例は、[difflib のコマンドラインインタフェース](#) を参照してください。

バージョン 2.3 で追加。

`difflib.IS_LINE_JUNK(line)`

無視できる行のとき真を返します。行 *line* は空白、または '#' ひとつのときに無視できます。それ以外のときには無視できません。Python 2.3 以前は `ndiff()` の引数 *linkjunk* にデフォルトで使用されました。

`difflib.IS_CHARACTER_JUNK(ch)`

無視できる文字のとき真を返します。文字 *ch* が空白、またはタブ文字のときには無視できます。それ

以外の時には無視できません。 `ndiff()` の引数 `charjunk` としてデフォルトで使用されます。

参考:

Pattern Matching: The Gestalt Approach John W. Ratcliff と D. E. Metzener による類似のアルゴリズムに関する議論。 *Dr. Dobbs's Journal* 1988 年 7 月号掲載。

7.4.1 SequenceMatcher オブジェクト

`SequenceMatcher` クラスには、以下のようなコンストラクタがあります:

```
class difflib.SequenceMatcher (isjunk=None, a="", b="", autojunk=True)
```

オプションの引数 `isjunk` は、`None` (デフォルトの値です) にするか、単一の引数をとる関数でなければなりません。後者の場合、関数はシーケンスの要素を受け取り、要素が junk であり、無視すべきである場合に限り真を返すようにしなければなりません。`isjunk` に `None` を渡すと、`lambda x: 0` を渡したのと同じになります; すなわち、いかなる要素も無視しなくなります。例えば以下のような引数を渡すと:

```
lambda x: x in " \t"
```

空白とタブ文字を無視して文字のシーケンスを比較します。

オプションの引数 `a` と `b` は、比較される文字列で、デフォルトでは空の文字列です。両方のシーケンスの要素は、ハッシュ化可能 (*hashable*) である必要があります。

オプションの引数 `autojunk` は、自動 junk ヒューリスティックを無効にするために使えます。

バージョン 2.7.1 で追加: `autojunk` パラメータ。

`SequenceMatcher` オブジェクトは以下のメソッドを持ちます:

```
set_seqs (a, b)
```

比較される 2 つの文字列を設定します。

`SequenceMatcher` オブジェクトは、2 つ目のシーケンスについての詳細な情報を計算し、キャッシュします。1 つのシーケンスをいくつかのシーケンスと比較する場合、まず `set_seq2()` を使って文字列を設定しておき、別の文字列を 1 つずつ比較するために、繰り返し `set_seq1()` を呼び出します。

```
set_seq1 (a)
```

比較を行う 1 つ目のシーケンスを設定します。比較される 2 つ目のシーケンスは変更されません。

```
set_seq2 (b)
```

比較を行う 2 つ目のシーケンスを設定します。比較される 1 つ目のシーケンスは変更されません。

```
find_longest_match (alo, ahi, blo, bhi)
```

`a[alo:ahi]` と `b[blo:bhi]` の中から、最長のマッチ列を探します。

`isjunk` が省略されたか `None` の時、`find_longest_match()` は `a[i:i+k]` が `b[j:j+k]` と等しいような `(i, j, k)` を返します。その値は `alo <= i <= i+k <= ahi` かつ `blo <=`

$j \leq j+k \leq bhi$ となります。(i', j', k') でも、同じようになります。さらに $k \geq k', i \leq i'$ が $i == i', j \leq j'$ でも同様です。言い換えると、いくつかのマッチ列すべてのうち、 a 内で最初に始まるものを返します。そしてその a 内で最初のマッチ列すべてのうち b 内で最初に始まるものを返します。

```
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

引数 *isjunk* が与えられている場合、上記の通り、はじめに最長のマッチ列を判定します。ブロック内に junk 要素が見当たらないような追加条件の際はこれに該当しません。次にそのマッチ列を、その両側の junk 要素にマッチするよう、できる限り広げていきます。そのため結果となる列は、探している列のたまたま直前にあった同一の junk 以外の junk にはマッチしません。

以下は前と同じサンプルですが、空白を junk とみなしています。これは ' abcd' が 2 つ目の列の末尾にある ' abcd' にマッチしないようにしています。代わりに 'abcd' にはマッチします。そして 2 つ目の文字列中、一番左の 'abcd' にマッチします:

```
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

どんな列にもマッチしない時は、(alo, blo, 0) を返します。

バージョン 2.6 で変更: このメソッドは *named tuple* Match(a, b, size) を返します。

get_matching_blocks()

マッチした互いに重複の無いシーケンスを表す、3 つ組の値のリストを返します。それぞれの値は (i, j, n) という形式で表され、 $a[i:i+n] == b[j:j+n]$ という関係を意味します。3 つの値は i と j の間で単調に増加します。

最後の 3 つ組はダミーで、(len(a), len(b), 0) という値を持ちます。これは $n == 0$ である唯一のタプルです。もし (i, j, n) と (i', j', n') がリストで並んでいる 3 つ組で、2 つ目が最後の 3 つ組でなければ、 $i+n < i'$ または $j+n < j'$ です。言い換えると並んでいる 3 つ組は常に隣接していない同じブロックを表しています。

バージョン 2.5 で変更: 隣接する 3 つ組は常に隣接しないブロックを表すと保証するようになりました。

```
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2), Match(a=5, b=4, size=0)]
```

get_opcodes()

a を b にするための方法を記述する 5 つのタプルを返します。それぞれのタプルは (tag, i1, i2, j1, j2) という形式であらわされます。最初のタプルは $i1 == j1 == 0$ であり、 $i1$ はその前にあるタプルの $i2$ と同じ値です。同様に $j1$ は前の $j2$ と同じ値になります。

tag の値は文字列であり、次のような意味です:

値	意味
'replace'	<code>a[i1:i2]</code> は <code>b[j1:j2]</code> に置き換えられる。
'delete'	<code>a[i1:i2]</code> は削除される。この時、 <code>j1 == j2</code> である。
'insert'	<code>b[j1:j2]</code> が <code>a[i1:i1]</code> に挿入される。この時 <code>i1 == i2</code> である。
'equal'	<code>a[i1:i2] == b[j1:j2]</code> (サブシーケンスは等しい)。

例えば:

```
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...     print ("%7s a[%d:%d] (%s) b[%d:%d] (%s)" %
...           (tag, i1, i2, a[i1:i2], j1, j2, b[j1:j2]))
delete a[0:1] (q) b[0:0] ()
equal a[1:3] (ab) b[0:2] (ab)
replace a[3:4] (x) b[2:3] (y)
equal a[4:6] (cd) b[3:5] (cd)
insert a[6:6] () b[5:6] (f)
```

`get_grouped_opcodes([n])`

最大 n 行までのコンテキストを含むグループを生成するような、ジェネレータ (*generator*) を返します。

このメソッドは、`get_opcodes()` で返されるグループの中から、似たような差異のかたまりに分け、間に挟まっている変更の無い部分を省きます。

グループは `get_opcodes()` と同じ書式で返されます。

バージョン 2.3 で追加.

`ratio()`

$[0, 1]$ の範囲の浮動小数点数で、シーケンスの類似度を測る値を返します。

T が 2 つのシーケンスの要素数の総計だと仮定し、 M をマッチした数とすると、この値は $2.0 * M / T$ であらわされます。もしシーケンスがまったく同じ場合、値は 1.0 となり、まったく異なる場合には 0.0 となります。

このメソッドは `get_matching_blocks()` または `get_opcodes()` がまだ呼び出されていない場合には非常にコストが高いです。この場合、上限を素早く計算するために、`quick_ratio()` もしくは `real_quick_ratio()` を最初に試してみる方がいいかもしれません。

`quick_ratio()`

`ratio()` の上界を、より高速に計算します。

`real_quick_ratio()`

`ratio()` の上界を、非常に高速に計算します。

この文字列全体のマッチ率を返す 3 つのメソッドは、精度の異なる近似値を返します。 `quick_ratio()` と `real_quick_ratio()` は、常に `ratio()` 以上の値を返します:

```
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

7.4.2 SequenceMatcher の例

この例は 2 つの文字列を比較します。空白を "junk" とします。

```
>>> s = SequenceMatcher(lambda x: x == " ",
...                       "private Thread currentThread;",
...                       "private volatile Thread currentThread;")
```

`ratio()` は、 $[0, 1]$ の範囲の値を返し、シーケンスの類似度を測ります。経験によると、`ratio()` の値が 0.6 を超えると、シーケンスがよく似ていることを示します:

```
>>> print round(s.ratio(), 3)
0.866
```

シーケンスのどこがマッチしているかにだけ興味のある時には `get_matching_blocks()` が手軽でしょう:

```
>>> for block in s.get_matching_blocks():
...     print "a[%d] and b[%d] match for %d elements" % block
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 21 elements
a[29] and b[38] match for 0 elements
```

`get_matching_blocks()` が返す最後のタプルが常にダミーであることに注目してください。このダミーは `(len(a), len(b), 0)` であり、これはタプルの最後の要素 (マッチする要素の数) が 0 となる唯一のケースです。

はじめのシーケンスがどのようにして 2 番目のものになるのかを知るには、`get_opcodes()` を使います:

```
>>> for opcode in s.get_opcodes():
...     print "%6s a[%d:%d] b[%d:%d]" % opcode
equal a[0:8] b[0:8]
insert a[8:8] b[8:17]
equal a[8:29] b[17:38]
```

参考:

- `SequenceMatcher` を使った、シンプルで使えるコードを知るには、このモジュールの関数 `get_close_matches()` を参照してください。
- Simple version control recipe `SequenceMatcher` で作った小規模アプリケーション。

7.4.3 Differ オブジェクト

Differ オブジェクトによって生成された差分が 最小 であるなどとは言いません。むしろ、最小の差分はしばしば直観に反しています。その理由は、どこでもできるとなれば一致を見いだしてしまうからで、ときには思いがけなく 100 ページも離れたマッチになってしまうのです。一致点を互いに隣接したマッチに制限することで、場合によって長めの差分を出力するというコストを掛けることにはなっても、ある種の局所性を保つことができるのです。

Differ は、以下のようなコンストラクタを持ちます：

```
class difflib.Differ([linejunk[, charjunk]])
```

オプションのキーワードパラメータ *linejunk* と *charjunk* は、フィルタ関数を渡します (使わないときは *None*)：

linejunk: ひとつの文字列引数を受け取る関数です。文字列が *junk* のときに真を返します。デフォルトでは、*None* であり、どんな行であっても *junk* とは見なされません。

charjunk: この関数は文字 (長さ 1 の文字列) を引数として受け取り、文字が *junk* であるときに真を返します。デフォルトは *None* であり、どんな文字も *junk* とは見なされません。

Differ オブジェクトは、以下の 1 つのメソッドを通して利用されます。(差分を生成します)：

```
compare(a, b)
```

文字列からなる 2 つのシーケンスを比較し、差分 (を表す文字列からなるシーケンス) を生成します。

各シーケンスの要素は、改行で終わる独立した単一行からなる文字列でなければなりません。そのようなシーケンスは、ファイル風オブジェクトの *readlines()* メソッドによって得ることができます。(得られる) 差分は改行文字で終了する文字列のシーケンスとして得られ、ファイル風オブジェクトの *writelines()* メソッドによって出力できる形になっています。

7.4.4 Differ の例

以下の例は 2 つのテキストを比較しています。最初に、テキストを行毎に改行で終わる文字列のシーケンスにセットアップします (そのようなシーケンスは、ファイル風オブジェクトの *readlines()* メソッドからも得ることができます)。

```
>>> text1 = ''' 1. Beautiful is better than ugly.
... 2. Explicit is better than implicit.
... 3. Simple is better than complex.
... 4. Complex is better than complicated.
... '''.splitlines(1)
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = ''' 1. Beautiful is better than ugly.
... 3. Simple is better than complex.
```

(次のページに続く)

(前のページからの続き)

```
... 4. Complicated is better than complex.
... 5. Flat is better than nested.
... ''' .splitlines(1)
```

次に Differ オブジェクトをインスタンス化します:

```
>>> d = Differ()
```

注意: *Differ* オブジェクトをインスタンス化するとき、行 junk と文字 junk をフィルタリングする関数を渡すことができます。詳細は *Differ()* コンストラクタを参照してください。

最後に、2 つを比較します:

```
>>> result = list(d.compare(text1, text2))
```

result は文字列のリストなので、pretty-print してみましょう:

```
>>> from pprint import pprint
>>> pprint(result)
[' 1. Beautiful is better than ugly.\n',
'- 2. Explicit is better than implicit.\n',
'- 3. Simple is better than complex.\n',
'+ 3.   Simple is better than complex.\n',
'?   ++\n',
'- 4. Complex is better than complicated.\n',
'?           ^           ---- ^\n',
'+ 4. Complicated is better than complex.\n',
'?           ++++ ^           ^\n',
'+ 5. Flat is better than nested.\n']
```

これは、複数行の文字列として、次のように出力されます:

```
>>> import sys
>>> sys.stdout.writelines(result)
1. Beautiful is better than ugly.
- 2. Explicit is better than implicit.
- 3. Simple is better than complex.
+ 3.   Simple is better than complex.
?   ++
- 4. Complex is better than complicated.
?           ^           ---- ^
+ 4. Complicated is better than complex.
?           ++++ ^           ^
+ 5. Flat is better than nested.
```

7.4.5 difflib のコマンドラインインタフェース

この例は、difflib を使って diff に似たユーティリティを作成する方法を示します。これは、Python のソース配布物にも、Tools/scripts/diff.py として含まれています。

```

""" Command line interface to difflib.py providing diffs in four formats:

* ndiff:      lists every line and highlights interline changes.
* context:    highlights clusters of changes in a before/after format.
* unified:    highlights clusters of changes in an inline format.
* html:       generates side by side comparison with change highlights.

"""

import sys, os, time, difflib, optparse

def main():
    # Configure the option parser
    usage = "usage: %prog [options] fromfile tofile"
    parser = optparse.OptionParser(usage)
    parser.add_option("-c", action="store_true", default=False,
                      help='Produce a context format diff (default)')
    parser.add_option("-u", action="store_true", default=False,
                      help='Produce a unified format diff')
    hlp = 'Produce HTML side by side diff (can use -c and -l in conjunction)'
    parser.add_option("-m", action="store_true", default=False, help=hlp)
    parser.add_option("-n", action="store_true", default=False,
                      help='Produce a ndiff format diff')
    parser.add_option("-l", "--lines", type="int", default=3,
                      help='Set number of context lines (default 3)')
    (options, args) = parser.parse_args()

    if len(args) == 0:
        parser.print_help()
        sys.exit(1)
    if len(args) != 2:
        parser.error("need to specify both a fromfile and tofile")

    n = options.lines
    fromfile, tofile = args # as specified in the usage string

    # we're passing these as arguments to the diff function
    fromdate = time.ctime(os.stat(fromfile).st_mtime)
    todater = time.ctime(os.stat(tofile).st_mtime)
    with open(fromfile, 'U') as f:
        fromlines = f.readlines()
    with open(tofile, 'U') as f:
        tolines = f.readlines()

    if options.u:
        diff = difflib.unified_diff(fromlines, tolines, fromfile, tofile,
                                    fromdate, todater, n=n)
    elif options.n:
        diff = difflib.ndiff(fromlines, tolines)
    elif options.m:
        diff = difflib.HtmlDiff().make_file(fromlines, tolines, fromfile,
                                             tofile, context=options.c,

```

(次のページに続く)

(前のページからの続き)

```

numlines=n)

else:
    diff = difflib.context_diff(fromlines, tolines, fromfile, tofile,
                                fromdate, todate, n=n)

    # we're using writelines because diff is a generator
    sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()

```

7.5 StringIO — ファイルのように文字列を読み書きする

このモジュールは、(メモリファイル としても知られている) 文字列のバッファに対して読み書きを行うファイルのようなクラス、*StringIO* を実装しています。操作方法についてはファイルオブジェクトの説明を参照してください(ファイルオブジェクト)。 (通常の文字列については *str* と *unicode* を参照してください)

class StringIO.*StringIO*(*[buffer]*)

StringIO オブジェクトを作る際に、コンストラクタに文字列を渡してオブジェクトを初期化できます。文字列を渡さない場合、最初は *StringIO* は空です。どちらの場合でも最初のファイル位置は 0 から始まります。

StringIO オブジェクトは Unicode も 8 ビットの文字列も受け付けますが、この 2 つを混ぜることに少し注意が必要です。この 2 つが一緒に使われると、(8 ビット目を使っていて) 7-bit ASCII として解釈できない 8-bit の文字列は、*getvalue()* が呼ばれたときに *UnicodeError* を引き起こします。

次にあげる *StringIO* オブジェクトのメソッドには特別な説明が必要です:

StringIO.*getvalue()*

StringIO オブジェクトの *close()* メソッドが呼ばれる前ならいつでも、"file" の中身全体を返します。Unicode と 8 ビット文字列を混ぜることの説明は、上の注意を参照してください; この 2 つの文字コードを混ぜると、このメソッドは *UnicodeError* を引き起こす可能性があります。

StringIO.*close()*

メモリバッファを解放します。close された後の *StringIO* オブジェクトを操作しようすると *ValueError* が送出されます。

使い方の例:

```

import StringIO

output = StringIO.StringIO()
output.write('First line.\n')
print >>output, 'Second line.'

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'

```

(次のページに続く)

(前のページからの続き)

```

contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()

```

7.6 cStringIO — 高速化された StringIO

cStringIO モジュールは *StringIO* モジュールと同様のインターフェースを提供しています。*StringIO.StringIO* オブジェクトを酷使する場合、このモジュールにある *StringIO()* 関数をかわりに使うと効果的です。

cStringIO.StringIO([s])

読み書きのための StringIO 類似ストリームを返します。

組み込み型オブジェクトを返す factory 関数なので、サブクラス化して独自の関数を組むことはできません。属性の追加もできません。これらをするにはオリジナルの *StringIO* モジュールを使ってください。

StringIO モジュールとは異なり、このモジュールで提供されているものは、ブレイン ASCII 文字列にエンコードできない Unicode を受け付けることができません。

もう一つ違いがあります。引数に文字列を指定して *StringIO()* 呼び出すと読み出し専用のオブジェクトが生成されますが、この場合 *cStringIO.StringIO()* では *write()* メソッドを持たないオブジェクトを生成します。これらのオブジェクトは普段は見えません。トレースバックに *StringI* と *StringO* として表示されます。

次にあげるデータオブジェクトも提供されています:

cStringIO.InputType

文字列をパラメーターに渡して *StringIO()* を呼ぶことで作られるオブジェクトのオブジェクト型。

cStringIO.OutputType

パラメーターを渡さずに *StringIO()* を呼ぶことで返されるオブジェクトのオブジェクト型。

このモジュールには C API もあります。詳しくはこのモジュールのソースを参照してください。

使い方の例:

```

import cStringIO

output = cStringIO.StringIO()
output.write('First line.\n')
print >>output, 'Second line.'

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'
contents = output.getvalue()

```

(次のページに続く)

(前のページからの続き)

```
# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

7.7 textwrap — テキストの折り返しと詰め込み

バージョン 2.3 で追加.

ソースコード: `Lib/textwrap.py`

`textwrap` モジュールは実際の処理を行う `TextWrapper` とともに、2 つの便利な関数として `wrap()` と `fill()` を提供しています。また、ユーティティー関数 `dedent()` も提供しています。1 つか 2 つの文字列を `wrap` あるいは `fill` したいだけなら便利関数で十分ですし、そうでないならば効率のために `TextWrapper` のインスタンスを使うべきでしょう。

`textwrap.wrap(text[, width[, ...]])`

`text` (文字列) 内の段落を一つだけ折り返しを行います。したがって、すべての行が高々 `width` 文字の長さになります。最後に改行が付かない出力行のリストを返します。

オプションのキーワード引数は、以下で説明する `TextWrapper` のインスタンス属性に対応しています。`width` はデフォルトで 70 です。

`wrap()` の動作についての詳細は `TextWrapper.wrap()` メソッドを参照してください。

`textwrap.fill(text[, width[, ...]])`

`text` 内の段落を一つだけ折り返しを行い、折り返しが行われた段落を含む一つの文字列を返します。`fill()` はこれの省略表現です

```
"\n".join(wrap(text, ...))
```

特に、`fill()` は `wrap()` とまったく同じ名前のキーワード引数を受け取ります。

`wrap()` と `fill()` の両方ともが `TextWrapper` インスタンスを作成し、その一つのメソッドを呼び出すことで機能します。そのインスタンスは再利用されません。したがって、たくさんのテキスト文字列を折り返し/詰め込みを行うアプリケーションのためには、あなた自身の `TextWrapper` オブジェクトを作成することでさらに効率が良くなるでしょう。

テキストはなるべく空白か、ハイフンを含む語のハイフンの直後で折り返されます。`TextWrapper.break_long_words` が偽に設定されていなければ、必要な場合に長い語が分解されます。

追加のユーティリティー関数である `dedent()` は、不要な空白をテキストの左側に持つ文字列からインデントを取り去ります。

`textwrap.dedent(text)`

`text` の各行に対し、共通して現れる先頭の空白を削除します。

この関数は通常、三重引用符で囲われた文字列をスクリーン/その他の左端にそろえ、なおかつソースコード中ではインデントされた形式を損なわないようにするために使われます。

タブとスペースはともにホワイトスペースとして扱われますが、同じではないことに注意してください: " hello" という行と "\thello" は、同じ先頭の空白文字をもっていないとみなされます。(これは Python 2.5 からの新しい振る舞いです。以前のバージョンではこのモジュールは共通の先頭空白文字を探す前に、不正にタブを展開していました。)

空白文字しか含まない行は入力の際に無視され、出力の際に単一の改行文字に正規化されます。

例えば:

```
def test():
    # end first line with \ to avoid the empty line!
    s = '''\
hello
    world
'''
    print repr(s)          # prints '    hello\n        world\n    '
    print repr(dedent(s))  # prints 'hello\n world\n'
```

class `textwrap.TextWrapper(...)`

`TextWrapper` コンストラクタはたくさんのオプションのキーワード引数を受け取ります。それぞれのキーワード引数は一つのインスタンス属性に対応します。したがって、例えば

```
wrapper = TextWrapper(initial_indent="* ")
```

はこれと同じです

```
wrapper = TextWrapper()
wrapper.initial_indent = "* "
```

あなたは同じ `TextWrapper` オブジェクトを何回も再利用できます。また、使用中にインスタンス属性へ代入することでそのオプションのどれでも変更できます。

`TextWrapper` インスタンス属性 (とコンストラクタのキーワード引数) は以下の通りです:

width

(デフォルト: 70) 折り返しが行われる行の最大の長さ。入力行に `width` より長い単一の語が無い限り、`TextWrapper` は `width` 文字より長い出力行が無いことを保証します。

expand_tabs

(デフォルト: True) もし真ならば、そのときは `text` 内のすべてのタブ文字は `text` の `expandtabs()` メソッドを用いて空白に展開されます。

replace_whitespace

(デフォルト: True) 真の場合、`wrap()` メソッドはタブの展開の後、`wrap` 処理の前に各種空白文字をスペース 1 文字に置換します。置換される空白文字は: TAB, 改行, 垂直 TAB, FF, CR ('`\t\n\v\f\r`') です。

注釈: `expand_tabs` が偽で `replace_whitespace` が真ならば、各タブ文字は 1 つの空白に置き換えられます。それはタブ展開と同じではありません。

注釈: `replace_whitespace` が偽の場合、改行が行の途中で現れることで出力がおかしくなることがあります。このため、テキストを (`str.splitlines()` などを使って) 段落ごとに分けて別々に wrap する必要があります。

drop_whitespace

(デフォルト: `True`) 真の場合、(wrap 処理のあとインデント処理の前に) 各行の最初と最後の空白文字を削除します。ただし、段落の最初の空白については、次の文字が空白文字でない場合は削除されません。削除される空白文字が行全体に及ぶ場合は、行自体を削除します。

バージョン 2.6 で追加: 過去のバージョンでは、空白は常に削除されていました。

initial_indent

(default: `' '`) wrap の出力の最初の行の先頭に付与する文字列。最初の行の長さに加算されます。空文字列の場合インデントされません。

subsequent_indent

(デフォルト: `' '`) 一行目以外の折り返しが行われる出力のすべての行の先頭に付けられる文字列。一行目以外の各行の折返しまでの長さにカウントされます。

fix_sentence_endings

(デフォルト: `False`) もし真ならば、`TextWrapper` は文の終わりを見つけようとし、確実に文がちょうど二つの空白で常に区切られているようにします。これは一般的に固定スペースフォントのテキストに対して望ましいです。しかし、文の検出アルゴリズムは完全ではありません: 文の終わりには、後ろに空白がある `'.'`, `','`, `':'` または `'?'` の中の一つ、ことによると `'"'` あるいは `"'"` が付随する小文字があると仮定しています。これに伴う一つの問題はアルゴリズムで下記の `"Dr."` と

```
[...] Dr. Frankenstein's monster [...]
```

下記の `"Spot."` の間の差異を検出できないことです

```
[...] See Spot. See Spot run [...]
```

`fix_sentence_endings` はデフォルトで偽です。

文検出アルゴリズムは `"小文字"` の定義のために `string.lowercase` に依存し、同一行の文を区切るためにピリオドの後に二つの空白を使う慣習に依存しているため、英文テキストに限定されたものです。

break_long_words

(デフォルト: `True`) もし真ならば、そのとき `width` より長い行が確実にないようにするために、`width` より長い語は切られます。偽ならば、長い語は切られないでしょう。そして、`width` より

長い行があるかもしれません。(*width* を超える分を最小にするために、長い語は単独で一行に置かれるでしょう。)

break_on_hyphens

(デフォルト: `True`) 真の場合、英語で一般的なように、ラップ処理は空白か合成語に含まれるハイフンの直後で行われます。偽の場合、空白だけが改行に適した位置として判断されます。ただし、本当に語の途中で改行が行われないようにするためには、 *break_long_words* 属性を真に設定する必要があります。過去のバージョンでのデフォルトの振る舞いは、常にハイフンの直後での改行を許していました。

バージョン 2.6 で追加.

TextWrapper はモジュールレベルの簡易関数に類似した二つの公開メソッドも提供します:

wrap (*text*)

1 段落の文字列 *text* を、各行が *width* 文字以下になるように wrap します。wrap のすべてのオプションは *TextWrapper* インスタンスの属性から取得します。結果の、行末に改行のない行のリストを返します。出力の内容が空になる場合は、返すリストも空になります。

fill (*text*)

text 内の段落を一つだけ折り返しを行い、折り返しが行われた段落を含む一つの文字列を返します。

7.8 codecs — codec レジストリと基底クラス

このモジュールでは、内部的な Python codec レジストリに対するアクセス手段を提供しています。codec レジストリは、標準の Python codec(エンコードとデコード)の基底クラスを定義し、codec およびエラー処理の検索手順を管理しています。

codecs では以下の関数を定義しています:

`codecs.encode(obj[, encoding[, errors]])`

encoding に記載された codec を使用して *obj* をエンコードします。デフォルトのエンコーディングは `'ascii'` です。

希望のエラー処理スキームを *errors* に設定することができます。デフォルトのエラーハンドラは `'strict'` です。これはエンコードエラーは *ValueError* (もしくは *UnicodeEncodeError* のような、より codec に固有のサブクラス) を送出することを意味します。codec エラー処理についてのより詳しい情報は *Codec 基底クラス* を参照してください。

バージョン 2.4 で追加.

`codecs.decode(obj[, encoding[, errors]])`

encoding に記載された codec を使用して *obj* をデコードします。デフォルトのエンコーディングは `'ascii'` です。

希望のエラー処理スキームを *errors* に設定することができます。デフォルトのエラーハンドラは `'strict'` です。これはデコードエラーは *ValueError* (もしくは *UnicodeDecodeError* のよう

な、より codec に固有のサブクラス) を送出することを意味します。codec エラー処理についてのより詳しい情報は [Codec 基底クラス](#) を参照してください。

バージョン 2.4 で追加.

`codecs.register(search_function)`

codec 検索関数を登録します。検索関数は第 1 引数にアルファベットの小文字から成るエンコーディング名を取り、以下の属性を持つ `CodecInfo` オブジェクトを返します:

- `name` エンコーディング名;
- `encode` 内部状態を持たないエンコード関数;
- `decode` 内部状態を持たないデコード関数;
- `incrementalencoder` 漸増的エンコーダクラスまたはファクトリ関数;
- `incrementaldecoder` 漸増的デコーダクラスまたはファクトリ関数;
- `streamwriter` ストリームライタクラスまたはファクトリ関数;
- `streamreader` ストリームリーダクラスまたはファクトリ関数。

種々の関数やクラスが以下の引数をとります:

encode と *decode*: これらの引数は、`Codec` インスタンスの `encode()` と `decode()` と同じインタフェースを持つ関数、またはメソッドでなければなりません ([Codec Interface](#) 参照)。これらの関数・メソッドは内部状態を持たずに動作する (stateless mode) と想定されています。

incrementalencoder と *incrementaldecoder*: これらは以下のインタフェースを持つファクトリ関数でなければなりません:

```
factory(errors='strict')
```

ファクトリ関数は、それぞれ基底クラスの `IncrementalEncoder` や `IncrementalDecoder` が定義しているインタフェースを提供するオブジェクトを返さなければなりません。漸増的 codecs は内部状態を維持できます。

streamreader と *streamwriter*: これらの引数は、次のようなインタフェースを持つファクトリ関数でなければなりません:

```
factory(stream, errors='strict')
```

ファクトリ関数は、それぞれ基底クラスの `StreamReader` や `StreamWriter` が定義しているインタフェースを提供するオブジェクトを返さねばなりません。ストリーム codecs は内部状態を維持できます。

errors が取り得る値は

- `'strict'` エンコーディングエラーの際に例外を発生
- `'replace'` 奇形データを `'?'` や `'\ufffd'` 等の適切な文字で置換
- `'ignore'` 奇形データを無視し何も通知せずに処理を継続

- 'xmlcharrefreplace' 適切な XML 文字参照で置換 (エンコーディングのみ))
- 'backslashreplace' (バックスラッシュ付きのエスケープシーケンス (エンコーディングのみ))

と `register_error()` で定義されたその他のエラー処理名になります。

検索関数は、与えられたエンコーディングを見つけられなかった場合、`None` を返さなければなりません。

`codecs.lookup(encoding)`

Python codec レジストリから codec 情報を探し、上で定義したような `CodecInfo` オブジェクトを返します。

エンコーディングの検索は、まずレジストリのキャッシュから行います。見つからなければ、登録されている検索関数のリストから探します。 `CodecInfo` オブジェクトが一つも見つからなければ `LookupError` を送出します。見つかったら、その `CodecInfo` オブジェクトはキャッシュに保存され、呼び出し側に返されます。

さまざまな codec へのアクセスを簡便化するために、このモジュールは以下のような関数を提供しています。これらの関数は、codec の検索に `lookup()` を使います:

`codecs.getencoder(encoding)`

与えられたエンコーディングに対する codec を検索し、エンコード関数を返します。

エンコーディングが見つからなければ `LookupError` を送出します。

`codecs.getdecoder(encoding)`

与えられたエンコーディングに対する codec を検索し、デコード関数を返します。

エンコーディングが見つからなければ `LookupError` を送出します。

`codecs.getincrementalencoder(encoding)`

与えられたエンコーディングに対する codec を検索し、漸増的エンコーダクラスまたはファクトリ関数を返します。

エンコーディングが見つからないか、codec が漸増的エンコーダをサポートしなければ `LookupError` を送出します。

バージョン 2.5 で追加.

`codecs.getincrementaldecoder(encoding)`

与えられたエンコーディングに対する codec を検索し、漸増的デコーダクラスまたはファクトリ関数を返します。

エンコーディングが見つからないか、codec が漸増的デコーダをサポートしなければ `LookupError` を送出します。

バージョン 2.5 で追加.

`codecs.getreader(encoding)`

与えられたエンコーディングに対する codec を検索し、`StreamReader` クラスまたはファクトリ関数を

返します。

エンコーディングが見つからなければ `LookupError` を送出します。

`codecs.getwriter(encoding)`

与えられたエンコーディングに対する codec を検索し、`StreamWriter` クラスまたはファクトリ関数を返します。

エンコーディングが見つからなければ `LookupError` を送出します。

`codecs.register_error(name, error_handler)`

エラー処理関数 `error_handler` を名前 `name` で登録します。エンコード中およびデコード中にエラーが送出された場合、`errors` パラメタに `name` を指定していれば `error_handler` を呼び出すようになります。

エンコード時の `error_handler` はエラーの場所に関する情報の入った `UnicodeEncodeError` インスタンスとともに呼び出されます。エラー処理関数はこの例外を送出するか、別の例外を送出するか、または入力のエンコードができなかった部分の代替文字列とエンコードを再開する場所の指定が入ったタプルを返すかしなければなりません。最後の場合、エンコーダは代替文字列をエンコードし、元の入力中の指定位置からエンコードを再開します。位置を負の値にすると、入力文字列の末端からの相対位置として扱われます。境界の外側にある位置を返した場合には `IndexError` が送出されます。

デコードと翻訳は同様に働きますが、エラー処理関数に渡されるのが `UnicodeDecodeError` か `UnicodeTranslateError` である点と、エラー処理関数の置換した内容が直接出力になる点が異なります。

`codecs.lookup_error(name)`

名前 `name` で登録済みのエラー処理関数を返します。

エラー処理関数が見つからなければ `LookupError` を送出します。

`codecs.strict_errors(exception)`

`strict` エラー処理の実装です: エンコード又はデコードエラーは各々 `UnicodeError` を送出します。

`codecs.replace_errors(exception)`

`replace` エラー処理の実装です: 奇形データは適切な文字列に置換されます。バイト文字列では '?'、Unicode 文字列では '\ufffd' に置換されます。

`codecs.ignore_errors(exception)`

`ignore` エラー処理の実装です: 奇形データは無視されエンコード又はデコードは何も通知せず、継続されます。

`codecs.xmlcharrefreplace_errors(exception)`

`xmlcharrefreplace` エラー処理の実装です (エンコードのみ): エンコードできなかった文字は適切な XML 文字参照に置き換えます。

`codecs.backslashreplace_errors(exception)`

`backslashreplace` エラー処理の実装です (エンコードのみ): エンコードできなかった文字はバックスラッシュ付きのエスケープシーケンスに置き換えられます。

エンコードされたファイルやストリームの処理を簡便化するため、このモジュールは次のようなユーティリティ関数を定義しています:

```
codecs.open(filename, mode[, encoding[, errors[, buffering]]])
```

エンコードされたファイルを *mode* を使って開き、透過的なエンコード/デコードを提供するラップされたバージョンを返します。デフォルトのファイルモードは 'r'、つまり、読み出しモードでファイルを開きます。

注釈: ラップされたバージョンは、該当する codec が定義している形式のオブジェクトだけを受け付けます。多くの組み込み codec では Unicode オブジェクトです。関数の戻り値も codec に依存し、通常は Unicode オブジェクトです。

注釈: 非バイナリモードが指定されても、ファイルは常にバイナリモードで開かれます。これは、8-bit の値を使うエンコーディングでデータが消失するのを防ぐためです。つまり、読み出しや書き込み時に、`'\n'` の自動変換はされないということです。

encoding にはファイルのエンコーディングを指定します。

エラーハンドリングのために *errors* を渡すことができます。これはデフォルトでは 'strict' で、エンコード時にエラーがあれば *ValueError* を送出します。

buffering は組み込み関数 *open()* の場合と同じ意味を持ちます。デフォルトでは行バッファリングです。

```
codecs.EncodedFile(file, input[, output[, errors]])
```

ラップしたファイルオブジェクトを返します。このオブジェクトは透過なエンコード変換を提供します。

ラップされたファイルに書かれた文字列は、*input* に指定したエンコーディングに従って変換され、*output* に指定したエンコーディングを使って string 型に変換され、元のファイルに書き込まれます。中間エンコーディングは指定された codecs に依存しますが、普通は Unicode です。

output が与えられなければ、*input* がデフォルトになります。

エラーハンドリングのために *errors* を渡すことができます。これはデフォルトでは 'strict' で、エンコード時にエラーがあれば *ValueError* を送出します。

```
codecs.iterencode(iterable, encoding[, errors])
```

漸増的エンコーダを使って、*iterable* から供給される入力を反復的にエンコードします。この関数は *generator* です。*errors* は (そして他のキーワード引数も同様に) 漸増的エンコーダにそのまま引き渡されます。

バージョン 2.5 で追加。

```
codecs.iterdecode(iterable, encoding[, errors])
```

漸増的デコーダを使って、*iterable* から供給される入力を反復的にデコードします。この関数は

generator です。 *errors* は (そして他のキーワード引数も同様に) 漸増的デコーダにそのまま引き渡されます。

バージョン 2.5 で追加。

このモジュールは以下のような定数も定義しています。プラットフォーム依存なファイルを読み書きするのに役立ちます:

```
codecs.BOM
codecs.BOM_BE
codecs.BOM_LE
codecs.BOM_UTF8
codecs.BOM_UTF16
codecs.BOM_UTF16_BE
codecs.BOM_UTF16_LE
codecs.BOM_UTF32
codecs.BOM_UTF32_BE
codecs.BOM_UTF32_LE
```

ここで定義された定数は、様々なエンコーディングの Unicode のバイトオーダーマーカ (BOM) で、UTF-16 と UTF-32 におけるデータストリームやファイルストリームのバイトオーダーを指定したり、UTF-8 における Unicode signature として使われます。 *BOM_UTF16* は *BOM_UTF16_BE* と *BOM_UTF16_LE* のいずれかで、プラットフォームのネイティブバイトオーダーに依存します。 *BOM* は *BOM_UTF16* の別名です。同様に *BOM_LE* は *BOM_UTF16_LE* の、 *BOM_BE* は *BOM_UTF16_BE* の別名です。他は UTF-8 と UTF-32 エンコーディングの BOM を表します。

7.8.1 Codec 基底クラス

codecs モジュールでは、codec のインタフェースを定義する一連の基底クラスを用意して、Python 用 codec を簡単に自作できるようにしています。

Python で何らかの codec を使えるようにするには、状態なしエンコーダ、状態なしデコーダ、ストリームリーダー、ストリームライタの 4 つのインタフェースを定義しなければなりません。通常は、状態なしエンコーダとデコーダを再利用してストリームリーダーとライタのファイル・プロトコルを実装します。

Codec クラスは、状態なしエンコーダ・デコーダのインタフェースを定義しています。

エラー処理の簡便化と標準化のため、*encode()* メソッドと *decode()* メソッドでは、*errors* 文字列引数を指定した場合に別のエラー処理を行うような仕組みを実装してもかまいません。全ての標準 Python codec では以下の文字列が定義され、実装されています。

値	意味
'strict'	<code>UnicodeError</code> (または、そのサブクラス) を送出します – デフォルトの動作です。
'ignore'	その文字を無視し、次の文字から変換を再開します。
'replace'	適当な文字で置換します – Python の組み込み Unicode codec のデコード時には公式の U+FFFD REPLACEMENT CHARACTER を、エンコード時には '?' を使います。
'xmlcharrefreplace'	適切な XML 文字参照で置換します (エンコードのみ)。
'backslashreplace'	バックスラッシュ付きのエスケープシーケンスで置換します (エンコードのみ)。

codecs がエラーハンドラとして受け入れる値は `register_error()` を使って追加できます。

Codec オブジェクト

Codec クラスは以下のメソッドを定義します。これらのメソッドは、内部状態を持たないエンコーダ / デコーダ関数のインタフェースを定義します:

`Codec.encode(input[, errors])`

オブジェクト `input` エンコードし、(出力オブジェクト, 消費した長さ) のタプルを返します。codecs は Unicode 専用ではありませんが、Unicode の文脈では、エンコーディングは Unicode オブジェクトを特定の文字集合エンコーディング (たとえば cp1252 や iso-8859-1) を使ってバイト文字列オブジェクトに変換します。

`errors` は適用するエラー処理を定義します。'strict' 処理がデフォルトです。

このメソッドは Codec に内部状態を保存してはなりません。効率よくエンコードするために状態を保持しなければならないような codecs には `StreamWriter` を使ってください。

エンコーダは長さが 0 の入力を処理できなければなりません。この場合、空のオブジェクトを出力オブジェクトとして返さなければなりません。

`Codec.decode(input[, errors])`

オブジェクト `input` をデコードし、(出力オブジェクト, 消費した長さ) のタプルを返します。Unicode の文脈では、デコードは特定の文字集合エンコーディングでエンコードされた文字列を Unicode オブジェクトに変換します。

`input` は `bf_getreadbuf` バッファスロットを提供するオブジェクトでなければなりません。バッファスロットを提供しているオブジェクトには Python 文字列オブジェクト、バッファオブジェクト、メモリマップファイルがあります。

`errors` は適用するエラー処理を定義します。'strict' 処理がデフォルトです。

このメソッドは、Codec インスタンスに内部状態を保存してはなりません。効率よくエンコード / デコードするために状態を保持しなければならないような codecs には `StreamReader` を使ってください。

デコーダは長さが 0 の入力を処理できなければなりません。この場合、空のオブジェクトを出力オブジェクトとして返さなければなりません。

IncrementalEncoder クラスおよび *IncrementalDecoder* クラスはそれぞれ漸増的エンコーディングおよびデコーディングのための基本的なインタフェースを提供します。エンコーディング/デコーディングは内部状態を持たないエンコーダ/デコーダを一度呼び出すことで行なわれるのではなく、漸増的エンコーダ/デコーダの *encode()/decode()* メソッドを複数回呼び出すことで行なわれます。漸増的エンコーダ/デコーダはメソッド呼び出しの間エンコーディング/デコーディング処理の進行を管理します。

encode()/decode() メソッド呼び出しの出力結果をまとめたものは、入力をひとまとめにして内部状態を持たないエンコーダ/デコーダでエンコード/デコードしたものと同じになります。

IncrementalEncoder オブジェクト

バージョン 2.5 で追加。

IncrementalEncoder クラスは入力を複数ステップでエンコードするのに使われます。全ての漸増的エンコーダが Python codec レジストリと互換性を持つために定義すべきメソッドとして、このクラスには以下のメソッドが定義されています。

```
class codecs.IncrementalEncoder (errors)
```

IncrementalEncoder インスタンスのコンストラクタ。

全ての漸増的エンコーダはこのコンストラクタインタフェースを提供しなければなりません。さらにキーワード引数を付け加えるのは構いませんが、Python codec レジストリで利用されるのはここで定義されているものだけです。

IncrementalEncoder は *errors* キーワード引数を提供して異なったエラー取扱方法を実装することもできます。あらかじめ定義されているパラメータは以下の通りです:

- 'strict' *ValueError* (またはそのサブクラス) を送出します。これがデフォルトです。
- 'ignore' 一文字無視して次に進みます。
- 'replace' 適当な代替文字で置き換えます。
- 'xmlcharrefreplace' 適切な XML 文字参照に置き換えます。
- 'backslashreplace' バックスラッシュ付きのエスケープシーケンスで置き換えます。

引数 *errors* は同名の属性に割り当てられます。属性に割り当てることで *IncrementalEncoder* オブジェクトが生きている間にエラー取扱戦略を違うものに切り替えることができますようになります。

errors 引数に許される値の集合は *register_error()* で拡張できます。

```
encode (object [, final])
```

object を (エンコーダの現在の状態を考慮に入れて) エンコードし、得られたエンコードされたオブジェクトを返します。 *encode()* 呼び出しがこれで最後という時には *final* は真でなければなりません (デフォルトは偽です)。

reset()

エンコーダを初期状態にリセットします。

IncrementalDecoder オブジェクト

IncrementalDecoder クラスは入力を複数ステップでデコードするのに使われます。全ての漸増的デコーダが Python codec レジストリと互換性を持つために定義すべきメソッドとして、このクラスには以下のメソッドが定義されています。

class codecs.*IncrementalDecoder* ([*errors*])

IncrementalDecoder インスタンスのコンストラクタ。

全ての漸増的デコーダはこのコンストラクタインタフェースを提供しなければなりません。さらにキーワード引数を付け加えるのは構いませんが、Python codec レジストリで利用されるのはここで定義されているものだけです。

IncrementalDecoder は *errors* キーワード引数を提供して異なったエラー取扱方法を実装することもできます。あらかじめ定義されているパラメータは以下の通りです:

- 'strict' *ValueError* (またはそのサブクラス) を送出します。これがデフォルトです。
- 'ignore' 一文字無視して次に進みます。
- 'replace' 適切な置換文字で置換します。

引数 *errors* は同名の属性に割り当てられます。属性に割り当てることで *IncrementalDecoder* オブジェクトが活着している間にエラー取扱戦略を違うものに切り替えることができます。

errors 引数に許される値の集合は *register_error()* で拡張できます。

decode (*object* [, *final*])

object を (デコーダの現在の状態を考慮に入れて) デコードし、得られたデコードされたオブジェクトを返します。 *decode()* 呼び出しがこれで最後という時には *final* は真でなければなりません (デフォルトは偽です)。もし *final* が真ならばデコーダは入力をデコードし切り全てのバッファをフラッシュしなければなりません。そうでない場合 (たとえば入力の最後に不完全なバイト列があるから)、デコーダは内部状態を持たない場合と同じようにエラーの取り扱いを開始しなければなりません (例外を送出するかもしれません)。

reset()

デコーダを初期状態にリセットします。

StreamWriter と *StreamReader* クラスは、新しいエンコーディングモジュールを、非常に簡単に実装するのに使用できる、一般的なインターフェイス提供します。実装例は `encodings.utf_8` をご覧ください。

StreamWriter オブジェクト

StreamWriter クラスは *Codec* のサブクラスで、以下のメソッドを定義しています。全てのストリームライタは、Python の codec レジストリとの互換性を保つために、これらのメソッドを定義する必要があります。

```
class codecs.StreamWriter (stream[, errors])
```

StreamWriter インスタンスのコンストラクタです。

全てのストリームライタはコンストラクタとしてこのインタフェースを提供しなければなりません。キーワード引数を追加しても構いませんが、Python の codec レジストリはここで定義されている引数だけを使います。

stream は、(バイナリで) 書き込み可能なファイル類似のオブジェクトでなくてはなりません。

StreamWriter は、*errors* キーワード引数を受けて、異なったエラー処理の仕組みを実装しても構いません。定義済みのパラメタを以下に示します:

- 'strict' *ValueError* (またはそのサブクラス) を送出します。これがデフォルトです。
- 'ignore' 一文字無視して次に進みます。
- 'replace' 適当な代替文字で置き換えます。
- 'xmlcharrefreplace' 適切な XML 文字参照に置き換えます。
- 'backslashreplace' バックスラッシュ付きのエスケープシーケンスで置き換えます。

errors 引数は、同名の属性に代入されます。この属性を変更すると、*StreamWriter* オブジェクトが生きている間に、異なるエラー処理に変更できます。

errors 引数に許される値の集合は *register_error()* で拡張できます。

write (*object*)

object の内容をエンコードしてストリームに書き出します。

writelines (*list*)

文字列からなるリストを連結して、(必要に応じて *write()* を何度も使って) ストリームに書き出します。

reset ()

状態保持に使われていた codec のバッファを強制的に出力してリセットします。

このメソッドが呼び出された場合、出力先データをきれいな状態にし、わざわざストリーム全体を再スキャンして状態を元に戻さなくても新しくデータを追加できるようにしなければなりません。

ここまでで挙げたメソッドの他にも、*StreamWriter* では背後にあるストリームの他の全てのメソッドや属性を継承しなければなりません。

StreamReader オブジェクト

StreamReader クラスは Codec のサブクラスで、以下のメソッドを定義しています。全てのストリームリーダは、Python の codec レジストリとの互換性を保つために、これらのメソッドを定義する必要があります。

```
class codecs.StreamReader (stream[, errors])
```

StreamReader インスタンスのコンストラクタです。

全てのストリームリーダはコンストラクタとしてこのインタフェースを提供しなければなりません。キーワード引数を追加しても構いませんが、Python の codec レジストリはここで定義されている引数だけを使います。

stream は、(バイナリで) 読み出し可能なファイル類似のオブジェクトでなくてはなりません。

StreamReader は、*errors* キーワード引数を受けて、異なったエラー処理の仕組みを実装しても構いません。定義済みのパラメタを以下に示します:

- 'strict' *ValueError* (またはそのサブクラス) を送出します。これがデフォルトです。
- 'ignore' 一文字無視して次に進みます。
- 'replace' 適切な置換文字で置換します。

errors 引数は、同名の属性に代入されます。この属性を変更すると、*StreamReader* オブジェクトが生きている間に、異なるエラー処理に変更できます。

errors 引数に許される値の集合は *register_error()* で拡張できます。

read(*[size[, chars[, firstline]]]*)

ストリームからのデータをデコードし、デコード済のオブジェクトを返します。

chars はストリームから読み込む文字数です。 *read()* は *chars* 以上の文字を返しません、それより少ない文字しか取得できない場合には *chars* 以下の文字を返します。

size は、デコードするためにストリームから読み込む、およその最大バイト数を意味します。デコードはこの値を適切な値に変更できます。デフォルト値 -1 にすると可能な限りたくさんのデータを読み込みます。 *size* の目的は、巨大なファイルの一括デコードを防ぐことにあります。

firstline は、1 行目さえ返せばその後の行でデコードエラーがあっても無視して十分だ、ということを示します。

このメソッドは貪欲な読み込み戦略を取るべきです。すなわち、エンコーディング定義と *size* の値が許す範囲で、できるだけ多くのデータを読むべきだということです。たとえば、ストリーム上にエンコーディングの終端や状態の目印があれば、それも読み込みます。

バージョン 2.4 で変更: *chars* 引数を追加しました。

バージョン 2.4.2 で変更: *firstline* 引数を追加しました。

readline(*[size[, keepends]]*)

入力ストリームから 1 行読み込み、デコード済みのデータを返します。

size が与えられた場合、ストリームの *read()* メソッドに *size* 引数として渡されます。

keepends が偽の場合には行末の改行が削除された行が返ります。

バージョン 2.4 で変更: *keepends* 引数を追加しました。

readlines(*[sizehint[, keepends]]*)

入力ストリームから全ての行を読み込み、行のリストとして返します。

keepends が真なら、改行は、codec のデコードメソッドを使って実装され、リスト要素の中に含まれます。

sizehint が与えられた場合、ストリームの *read()* メソッドに *size* 引数として渡されます。

reset()

状態保持に使われた codec のバッファをリセットします。

ストリームの読み位置を再設定してはならないので注意してください。このメソッドはデコードの際にエラーから復帰できるようにするためのものです。

ここまでで挙げたメソッドの他にも、*StreamReader* では背後にあるストリームの他の全てのメソッドや属性を継承しなければなりません。

次に挙げる 2 つの基底クラスは、利便性のために含まれています。codec レジストリは、これらを必要としませんが、実際のところ、あると有用なものでしょう。

StreamReaderWriter オブジェクト

StreamReaderWriter を使って、読み書き両方に使えるストリームをラップできます。

lookup() 関数が返すファクトリ関数を使って、インスタンスを生成するという設計です。

class `codecs.StreamReaderWriter` (*stream, Reader, Writer, errors*)

StreamReaderWriter インスタンスを生成します。 *stream* はファイル類似のオブジェクトです。 *Reader* と *Writer* は、それぞれ *StreamReader* と *StreamWriter* インタフェースを提供するファクトリ関数かファクトリクラスでなければなりません。エラー処理は、ストリームリーダとライタで定義したものと同じように行われます。

StreamReaderWriter インスタンスは、 *StreamReader* クラスと *StreamWriter* クラスを合わせたインタフェースを継承します。元になるストリームからは、他のメソッドや属性を継承します。

StreamRecoder オブジェクト

StreamRecoder はエンコーディングデータの、フロントエンド-バックエンドを観察する機能を提供します。異なるエンコーディング環境を扱うとき、便利な場合があります。

lookup() 関数が返すファクトリ関数を使って、インスタンスを生成するという設計です。

class `codecs.StreamRecoder` (*stream, encode, decode, Reader, Writer, errors*)

双方向変換を実装する *StreamRecoder* インスタンスを生成します。 *encode* と *decode* はフロントエンド (*read()* への入力と *write()* からの出力) を処理し、 *Reader* と *Writer* はバックエンド (ストリームに対する読み書き) を処理します。

これらのオブジェクトを使って、たとえば、Latin-1 から UTF-8、あるいは逆向きの変換を、透過に記録できます。

stream はファイル的オブジェクトでなくてはなりません。

`encode` と `decode` は Codec のインタフェースに忠実でなくてはならず、`Reader` と `Writer` は、それぞれ `StreamReader` と `StreamWriter` のインタフェースを提供するオブジェクトのファクトリ関数かクラスでなくてはなりません。

`encode` と `decode` はフロントエンドの変換に必要で、`Reader` と `Writer` はバックエンドの変換に必要です。中間のフォーマットはコーデックの組み合わせによって決定されます。たとえば、Unicode コデックは中間エンコーディングに Unicode を使います。

エラー処理はストリーム・リーダやライタで定義されている方法と同じように行われます。

`StreamRecoder` インスタンスは、`StreamReader` と `StreamWriter` クラスを合わせたインタフェースを定義します。また、元のストリームのメソッドと属性も継承します。

7.8.2 エンコーディングと Unicode

Unicode 文字列は内部的にはコードポイントのシーケンスとして格納されます (正確に言えば `Py_UNICODE` 配列です)。Python がどのようにコンパイルされたか (デフォルトである `--enable-unicode=ucs2` かまたは `--enable-unicode=ucs4` のどちらか) によって、`Py_UNICODE` は 16 ビットまたは 32 ビットのデータ型です。Unicode オブジェクトが CPU とメモリの外で使われることになると、CPU のエンディアンやこれらの配列がバイト列としてどのように格納されるかが問題になってきます。Unicode オブジェクトをバイト列に変換することをエンコーディングと呼び、バイト列から Unicode オブジェクトを再生することをデコーディングと呼びます。どのようにこの変換を行うかには多くの異なった方法があります (これらの方法のこともエンコーディングと言います)。最も単純な方法はコードポイント 0-255 をバイト `0x0-0xff` に写すことです。これは `U+00FF` より上のコードポイントを持つ Unicode オブジェクトはこの方法ではエンコードできないということを意味します (この方法を `'latin-1'` とか `'iso-8859-1'` と呼びます)。`unicode.encode()` は次のような `UnicodeEncodeError` を送出することになります: `UnicodeEncodeError: 'latin-1' codec can't encode character u'\u1234' in position 3: ordinal not in range(256)`

他のエンコーディングの一群 (`charmap` エンコーディングと呼ばれます) がありますが、Unicode コードポイントの別の部分集合とこれらがどのように `0x0-0xff` のバイトに写されるかを選んだものです。これがどのように行なわれるかを知るには、単にたとえば `encodings/cp1252.py` (主に Windows で使われるエンコーディングです) を開いてみてください。256 文字のひとつの文字列定数がありどの文字がどのバイト値に写されるかを示しています。

これらのエンコーディングはすべて、Unicode に定義された 1114112 のコードポイントのうちの 256 だけをエンコードすることができます。Unicode のすべてのコードポイントを格納するための単純で直接的な方法は、各コードポイントを連続する 4 バイトとして格納することです。これには 2 つの可能性があります: ビッグエンディアンまたはリトルエンディアンの順にバイトを格納することです。これら 2 つのエンコーディングはそれぞれ `UTF-32-BE` および `UTF-32-LE` と呼ばれます。それらのデメリットは、例えばリトルエンディアンのマシン上で `UTF-32-BE` を使用すると、エンコードでもデコードでも常にバイト順を交換する必要がありますことです。`UTF-32` はこの問題を回避します: バイト順は、常に自然なエンディアンに従います。しかし、これらのバイト順が異なるエンディアン性を持つ CPU によって読まれる場合、結局バイト順を交換しなければなりません。`UTF-16` あるいは `UTF-32` バイト列のエンディアン性を検出する目的で、いわゆる BOM (「バイト・オーダー・マーク」) があります。これは Unicode 文字 `U+FEFF` です。この文字はすべ

ての UTF-16 あるいは UTF-32 バイト列の前に置くことができます。この文字のバイトが交換されたバージョン (0xFFFE) は、Unicode テキストに現われてはならない不正な文字です。したがって、UTF-16 あるいは UTF-32 バイト列中の最初の文字が U+FFFE であるように見える場合、デコードの際にバイトを交換しなければなりません。不運にも文字 U+FEFF は ZERO WIDTH NO-BREAK SPACE として別の目的を持っていました: 幅を持たず、単語を分割することを許容しない文字。それは、例えばリガチャアルゴリズムにヒントを与えるために使用することができます。Unicode 4.0 で、ZERO WIDTH NO-BREAK SPACE としての U+FEFF の使用は廃止予定になりました (この役割は U+2060 (WORD JOINER) によって引き継がれました)。しかしながら、Unicode ソフトウェアは、依然として両方の役割の U+FEFF を扱うことができなければなりません: BOM として、エンコードされたバイトのメモリレイアウトを決定する手段であり、一旦バイト列が文字列にデコードされたならば消えます; ZERO WIDTH NO-BREAK SPACE として、他の任意の文字のようにデコードされる通常の文字です。

さらにもう一つ Unicode 文字全てをエンコードできるエンコーディングがあり、UTF-8 と呼ばれています。UTF-8 は 8 ビットエンコーディングで、したがって UTF-8 にはバイト順の問題はありません。UTF-8 バイト列の各バイトは二つのパートから成ります。二つはマーカ (上位数ビット) とペイロードです。マーカは 0 ビットから 4 ビットの 1 の列に 0 のビットが一つ続いたものです。Unicode 文字は次のようにエンコードされます (x はペイロードを表わし、連結されると一つの Unicode 文字を表わします):

範囲	エンコーディング
U-00000000 ... U-0000007F	0xxxxxxx
U-00000080 ... U-000007FF	110xxxxx 10xxxxxx
U-00000800 ... U-0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
U-00010000 ... U-0010FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Unicode 文字の最下位ビットとは最も右にある x のビットです。

UTF-8 は 8 ビットエンコーディングなので BOM は必要とせず、デコードされた Unicode 文字列中の U+FEFF は (たとえ最初の文字であったとしても) ZERO WIDTH NO-BREAK SPACE として扱われます。

外部からの情報無しには、Unicode 文字列のエンコーディングにどのエンコーディングが使われたのか信頼できる形で決定することは不可能です。どの charmap エンコーディングもどんなランダムなバイト列でもデコードできます。しかし UTF-8 ではそれは可能ではありません。任意のバイト列を許さないような構造を持っているからです。UTF-8 エンコーディングであることを検知する信頼性を向上させるために、Microsoft は Notepad プログラム用に UTF-8 の変種 (Python 2.5 では "utf-8-sig" と呼んでいます) を考案しました。Unicode 文字がファイルに書き込まれる前に UTF-8 でエンコードした BOM (バイト列では 0xef, 0xbb, 0xbf のように見えます) が書き込まれます。このようなバイト値で charmap エンコードされたファイルが始まることはほとんどあり得ない (たとえば iso-8859-1 では

LATIN SMALL LETTER I WITH DIAERESIS

RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK

INVERTED QUESTION MARK

のようになる) ので、utf-8-sig エンコーディングがバイト列から正しく推測される確率を高めます。つまりここでは BOM はバイト列を生成する際のバイト順を決定できるように使われているのではなく、エンコーディングを推測する助けになる印として使われているのです。utf-8-sig codec はエンコーディングの際

ファイルに最初の 3 文字として `0xef, 0xbb, 0xbf` を書き込みます。デコーディングの際はファイルの先頭に現れたこれら 3 バイトはスキップします。UTF-8 では BOM の使用は推奨されておらず、一般的には避けるべきです。

7.8.3 標準エンコーディング

Python には数多くの codec が組み込みで付属します。これらは C 言語の関数、対応付けを行うテーブルの両方で提供されています。以下のテーブルでは codec と、いくつかの良く知られている別名と、エンコーディングが使われる言語を列挙します。別名のリスト、言語のリストともしらみつぶしに網羅されているわけではありません。大文字と小文字、またはアンダースコアの代りにハイフンにただけの綴りも有効な別名です; そのため、例えば `'utf-8'` は `'utf_8'` codec の正当な別名です。

多くの文字セットは同じ言語をサポートしています。これらの文字セットは個々の文字 (例えば、EURO SIGN がサポートされているかどうか) や、文字のコード部分への割り付けが異なります。特に欧州言語では、典型的に以下の変種が存在します:

- ISO 8859 コードセット
- Microsoft Windows コードページで、8859 コード形式から導出されているが、制御文字を追加のグラフィック文字と置き換えたもの
- IBM EBCDIC コードページ
- ASCII 互換の IBM PC コードページ

Codec	別名	言語
ascii	646, us-ascii	英語
big5	big5-tw, csbig5	繁体字中国語
big5hkscs	big5-hkscs, hkscs	繁体字中国語
cp037	IBM037, IBM039	英語
cp424	EBCDIC-CP-HE, IBM424	ヘブライ語
cp437	437, IBM437	英語
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	西ヨーロッパ言語
cp720		アラビア語
cp737		ギリシャ語
cp775	IBM775	バルト沿岸国
cp850	850, IBM850	西ヨーロッパ言語
cp852	852, IBM852	中央および東ヨーロッパ
cp855	855, IBM855	ブルガリア、ベラルーシ、マケドニア、ロシア、セルビア
cp856		ヘブライ語
cp857	857, IBM857	トルコ語
cp858	858, IBM858	西ヨーロッパ言語

次のページに続く

表 3 – 前のページからの続き

Codec	別名	言語
cp860	860, IBM860	ポルトガル語
cp861	861, CP-IS, IBM861	アイスランド語
cp862	862, IBM862	ヘブライ語
cp863	863, IBM863	カナダ
cp864	IBM864	アラビア語
cp865	865, IBM865	デンマーク、ノルウェー
cp866	866, IBM866	ロシア語
cp869	869, CP-GR, IBM869	ギリシャ語
cp874		タイ語
cp875		ギリシャ語
cp932	932, ms932, mskanji, ms-kanji	日本語
cp949	949, ms949, uhc	韓国語
cp950	950, ms950	繁体字中国語
cp1006		Urdu
cp1026	ibm1026	トルコ語
cp1140	ibm1140	西ヨーロッパ言語
cp1250	windows-1250	中央および東ヨーロッパ
cp1251	windows-1251	ブルガリア、ベラルーシ、マケドニア、ロシア、セルビア
cp1252	windows-1252	西ヨーロッパ言語
cp1253	windows-1253	ギリシャ語
cp1254	windows-1254	トルコ語
cp1255	windows-1255	ヘブライ語
cp1256	windows-1256	アラビア語
cp1257	windows-1257	バルト沿岸国
cp1258	windows-1258	ベトナム
euc_jp	eucjp, ujis, u-jis	日本語
euc_jis_2004	jisx0213, eucjis2004	日本語
euc_jisx0213	eucjisx0213	日本語
euc_kr	euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001	韓国語
gb2312	chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58	簡体字中国語
gbk	936, cp936, ms936	Unified Chinese
gb18030	gb18030-2000	Unified Chinese
hz	hzgb, hz-gb, hz-gb-2312	簡体字中国語

次のページに続く

表 3 – 前のページからの続き

Codec	別名	言語
iso2022_jp	csiso2022jp, iso2022jp, iso-2022-jp	日本語
iso2022_jp_1	iso2022jp-1, iso-2022-jp-1	日本語
iso2022_jp_2	iso2022jp-2, iso-2022-jp-2	日本語, 韓国語, 簡体字中国語, 西欧, ギリシャ語
iso2022_jp_2004	iso2022jp-2004, iso-2022-jp-2004	日本語
iso2022_jp_3	iso2022jp-3, iso-2022-jp-3	日本語
iso2022_jp_ext	iso2022jp-ext, iso-2022-jp-ext	日本語
iso2022_kr	csiso2022kr, iso2022kr, iso-2022-kr	韓国語
latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	西ヨーロッパ
iso8859_2	iso-8859-2, latin2, L2	中央および東ヨーロッパ
iso8859_3	iso-8859-3, latin3, L3	エスペラント、マルタ
iso8859_4	iso-8859-4, latin4, L4	バルト沿岸国
iso8859_5	iso-8859-5, cyrillic	ブルガリア、ベラルーシ、マケドニア、ロシア、セルビア
iso8859_6	iso-8859-6, arabic	アラビア語
iso8859_7	iso-8859-7, greek, greek8	ギリシャ語
iso8859_8	iso-8859-8, hebrew	ヘブライ語
iso8859_9	iso-8859-9, latin5, L5	トルコ語
iso8859_10	iso-8859-10, latin6, L6	北欧語
iso8859_11	iso-8859-11, thai	タイ語
iso8859_13	iso-8859-13, latin7, L7	バルト沿岸国
iso8859_14	iso-8859-14, latin8, L8	ケルト語
iso8859_15	iso-8859-15, latin9, L9	西ヨーロッパ言語
iso8859_16	iso-8859-16, latin10, L10	南東ヨーロッパ
johab	cp1361, ms1361	韓国語
koi8_r		ロシア語
koi8_u		ウクライナ語
mac_cyrillic	maccyrillic	ブルガリア、ベラルーシ、マケドニア、ロシア、セルビア
mac_greek	macgreek	ギリシャ語
mac_iceland	maciceland	アイスランド語
mac_latin2	maclatin2, maccentraleurope	中央および東ヨーロッパ
mac_roman	macroman	西ヨーロッパ言語
mac_turkish	macturkish	トルコ語
ptcp154	csptcp154, pt154, cp154, cyrillic-asian	カザフ

次のページに続く

表 3 – 前のページからの続き

Codec	別名	言語
shift_jis	csshiftjis, shiftjis, sjis, s_jis	日本語
shift_jis.2004	shiftjis2004, sjis.2004, sjis2004	日本語
shift_jisx0213	shiftjisx0213, sjisx0213, s_jisx0213	日本語
utf_32	U32, utf32	全ての言語
utf_32.be	UTF-32BE	全ての言語
utf_32.le	UTF-32LE	全ての言語
utf_16	U16, utf16	全ての言語
utf_16.be	UTF-16BE	全ての言語 (BMP のみ)
utf_16.le	UTF-16LE	全ての言語 (BMP のみ)
utf_7	U7, unicode-1-1-utf-7	全ての言語
utf_8	U8, UTF, utf8	全ての言語
utf_8.sig		全ての言語

7.8.4 Python 特有のエンコーディング

予め定義された codec のいくつかは Python 特有のもので、それらの codec 名は Python の外では無意味なものとなります。以下に、想定されている入出力のタイプに基づいて、それらを表にしました（テキストエンコーディングは codec の最も一般的な使用例ですが、その根底にある codec 基盤は、ただのテキストエンコーディングというよりも、任意のデータの変換をサポートしていることに注意してください）。非対称的な codec については、その目的がエンコーディングの方向を説明しています。

以下の codec では Unicode から文字列へのエンコーディング^{*1}、文字列から Unicode へのデコーディング^{*2}を、Unicode テキストエンコーディング同様に提供しています。

^{*1} unicode オブジェクトを入力として受け容れる場所では str オブジェクトも許容されます。この場合デフォルトエンコーディングを使って暗黙に unicode に変換されます。この変換が失敗した場合はエンコード操作が `UnicodeDecodeError` を起こします。

^{*2} str オブジェクトを入力として受け容れる場所では unicode オブジェクトも許容されます。この場合デフォルトエンコーディングを使って暗黙に str に変換されます。この変換が失敗した場合はデコード操作が `UnicodeEncodeError` を起こします。

Codec	別名	目的
idna		RFC 3490 の実装です。 <code>encodings.idna</code> も参照してください。
mbcs	dbcs	Windows のみ: 被演算子を ANSI コードページ (CP_ACP) に従ってエンコードします
palms		PalmOS 3.5 のエンコーディングです
punycode		RFC 3492 を実装しています。
raw_unicode_escape		Python ソースコードにおける raw Unicode リテラルとして適切な文字列を生成します。
rot13	rot13	被演算子のシーザー暗号 (Caesar-cypher) を返します
undefined		全ての変換に対して例外を送出します。バイト列と Unicode 文字列との間で <i>coercion</i> (強制型変換) をおこないたくない時にシステムエンコーディングとして使うことができます。
unicode_escape		Python ソースコードにおける Unicode リテラルとして適切な文字列を生成します。
unicode_internal		被演算子の内部表現を返します。

バージョン 2.3 で追加: `idna`, `punycode` エンコーディングの追加。

以下の codec では文字列から文字列へのエンコーディングとデコーディング^{*2}を提供しています。

Codec	別名	目的	エンコーダ/デコーダ
base64_codec	base64, base-64	被演算子をマルチラインの MIME base64 に変換します (結果は常に末尾の '\n' を含みます)	<code>base64.encodestring()</code> , <code>base64.decodestring()</code>
bz2_codec	bz2	被演算子を bz2 を使って圧縮します	<code>bz2.compress()</code> , <code>bz2.decompress()</code>
hex_codec	hex	被演算子をバイトあたり 2 桁の 16 進数の表現に変換します	<code>binascii.b2a_hex()</code> , <code>binascii.a2b_hex()</code>
quopri_codec	quopri, quoted-printable, quotedprintable	被演算子を MIME quoted printable 形式に変換します	<code>quotetabs=True</code> を指定した <code>quopri.encode()</code> , <code>quopri.decode()</code>
string_escape		Python ソースコードにおける文字列リテラルとして適切な文字列を生成します。	
uu_codec	uu	被演算子を uuencode を用いて変換します	<code>uu.encode()</code> , <code>uu.decode()</code>
zlib_codec	zip, zlib	被演算子を gzip を用いて圧縮します	<code>zlib.compress()</code> , <code>zlib.decompress()</code>

7.8.5 `encodings.idna` — アプリケーションにおける国際化ドメイン名 (IDNA)

バージョン 2.3 で追加.

このモジュールでは [RFC 3490](#) (アプリケーションにおける国際化ドメイン名、IDNA: Internationalized Domain Names in Applications) および [RFC 3492](#) (Nameprep: 国際化ドメイン名 (IDN) のための stringprep プロファイル) を実装しています。このモジュールは `punycode` エンコーディングおよび `stringprep` の上に構築されています。

これらの RFC はともに、非 ASCII 文字の入ったドメイン名をサポートするためのプロトコルを定義しています。(www.Alliancefran 異 aise.nu のような) 非 ASCII 文字を含むドメイン名は、ASCII と互換性のあるエンコーディング (ACE、www.xn--alliancefranaise-npb.nu のような形式) に変換されます。ドメイン名の ACE 形式は、DNS クエリ、HTTP *Host* フィールドなどといった、プロトコル中で任意の文字を使えないような全ての局面で用いられます。この変換はアプリケーション内で行われます; 可能ならユーザからは不可視となります: アプリケーションは Unicode ドメインラベルをネットワークに載せる際に IDNA に、ACE ドメインラベルをユーザに提供する前に Unicode に、それぞれ透過的に変換しなければなりません。

Python ではこの変換をいくつかの方法でサポートします: `idna` codec は Unicode と ACE 間の変換を行い、入力文字列を [RFC 3490](#) の section 3.1 (1) で定義されている区切り文字に基づいてラベルに分解し、各ラベルを要求通りに ACE に変換します。逆に、入力のバイト文字列を、区切り文字でラベルに分解し、ACE ラベルを Unicode に変換します。さらに、`socket` モジュールは Unicode ホスト名を ACE に透過的に変換するため、アプリケーションはホスト名を `socket` モジュールに渡す際にホスト名の変換に煩わされることがありません。その上で、ホスト名を関数パラメタとして持つ、`httplib` や `ftplib` のようなモジュールでは Unicode ホスト名を受理します (`httplib` でもまた、`Host`: フィールドにある IDNA ホスト名を、フィールド全体を送信する場合に透過的に送信します)。

(逆引きなどによって) ネットワーク越しにホスト名を受信する際、Unicode への自動変換は行われません: こうしたホスト名をユーザに提供したいアプリケーションでは、Unicode にデコードしてやる必要があります。

`encodings.idna` ではまた、`nameprep` 手続きを実装しています。`nameprep` はホスト名に対してある正規化を行って、国際化ドメイン名で大小文字を区別しないようにするとともに、類似の文字を一元化します。`nameprep` 関数は必要なら直接使うこともできます。

`encodings.idna.nameprep(label)`

`label` を `nameprep` したバージョンを返します。現在の実装ではクエリ文字列を仮定しているので、`AllowUnassigned` は真です。

`encodings.idna.ToASCII(label)`

[RFC 3490](#) 仕様に従ってラベルを ASCII に変換します。`UseSTD3ASCIIRules` は偽であると仮定します。

`encodings.idna.ToUnicode(label)`

[RFC 3490](#) 仕様に従ってラベルを Unicode に変換します。

7.8.6 `encodings.utf_8_sig` — BOM 印付き UTF-8

バージョン 2.5 で追加.

このモジュールは UTF-8 codec の変種を実装します。エンコーディング時は、UTF-8 でエンコードしたバイト列の前に UTF-8 でエンコードした BOM を追加します。これは内部状態を持つエンコーダで、この動作は (バイトストリームの最初の書き込み時に) 一度だけ行なわれます。デコーディング時は、データの最初に UTF-8 でエンコードされた BOM があれば、それをスキップします。

7.9 unicodedata — Unicode データベース

このモジュールは、全ての Unicode 文字の属性を定義している Unicode 文字データベースへのアクセスを提供します。このデータベース内のデータは、<ftp://ftp.unicode.org/> で公開されている `UnicodeData.txt` ファイルのバージョン 5.2.0 に基づいています。

The module uses the same names and symbols as defined by the UnicodeData File Format 5.2.0 (see <https://www.unicode.org/reports/tr44/>). It defines the following functions:

`unicodedata.lookup(name)`

名前に対応する文字を探します。その名前の文字が見つかった場合、その Unicode 文字が返されます。見つからなかった場合には、`KeyError` を発生させます。

`unicodedata.name(unichr[, default])`

Unicode 文字 `unichr` に付いている名前を、文字列で返します。名前が定義されていない場合には `default` が返されますが、この引数が与えられていなければ `ValueError` を発生させます。

`unicodedata.decimal(unichr[, default])`

Unicode 文字 `unichr` に割り当てられている十進数を、整数で返します。この値が定義されていない場合には `default` が返されますが、この引数が与えられていなければ `ValueError` を発生させます。

`unicodedata.digit(unichr[, default])`

Unicode 文字 `unichr` に割り当てられている二進数を、整数で返します。この値が定義されていない場合には `default` が返されますが、この引数が与えられていなければ `ValueError` を発生させます。

`unicodedata.numeric(unichr[, default])`

Unicode 文字 `unichr` に割り当てられている数値を、float 型で返します。この値が定義されていない場合には `default` が返されますが、この引数が与えられていなければ `ValueError` を発生させます。

`unicodedata.category(unichr)`

Unicode 文字 `unichr` に割り当てられた、汎用カテゴリを返します。

`unicodedata.bidirectional(unichr)`

Unicode 文字 `unichr` に割り当てられた、双方向クラスを返します。そのような値が定義されていない場合、空の文字列が返されます。

`unicodedata.combining(unichr)`

Unicode 文字 `unichr` に割り当てられた正規結合クラスを返します。結合クラス定義されていない場合、0 が返されます。

`unicodedata.east_asian_width(unichr)`

Unicode 文字 `unichr` に割り当てられた east asian width を文字列で返します。

バージョン 2.4 で追加.

`unicodedata.mirrored(unichr)`

Unicode 文字 *unichr* に割り当てられた、鏡像化のプロパティを返します。その文字が双方向テキスト内で鏡像化された文字である場合には 1 を、それ以外の場合には 0 を返します。

`unicodedata.decomposition(unichr)`

Unicode 文字 *unichr* に割り当てられた、文字分解マッピングを、文字列型で返します。そのようなマッピングが定義されていない場合、空の文字列が返されます。

`unicodedata.normalize(form, unistr)`

Unicode 文字列 *unistr* の正規形 *form* を返します。 *form* の有効な値は、'NFC'、'NFKC'、'NFD'、'NFKD' です。

Unicode 規格は標準等価性 (canonical equivalence) と互換等価性 (compatibility equivalence) に基づいて、様々な Unicode 文字列の正規形を定義します。Unicode では、複数の方法で表現できる文字があります。たとえば、文字 U+00C7 (LATIN CAPITAL LETTER C WITH CEDILLA) は、U+0043 (LATIN CAPITAL LETTER C) U+0327 (COMBINING CEDILLA) というシーケンスとしても表現できます。

各文字には 2 つの正規形があり、それぞれ正規形 C と正規形 D といいます。正規形 D (NFD) は標準分解 (canonical decomposition) としても知られており、各文字を分解された形に変換します。正規形 C (NFC) は標準分解を適用した後、結合済文字を再構成します。

互換等価性に基づいて、2 つの正規形が加えられています。Unicode では、一般に他の文字との統合がサポートされている文字があります。たとえば、U+2160 (ROMAN NUMERAL ONE) は事実上 U+0049 (LATIN CAPITAL LETTER I) と同じものです。しかし、Unicode では、既存の文字集合 (たとえば gb2312) との互換性のために、これがサポートされています。

正規形 KD (NFKD) は、互換分解 (compatibility decomposition) を適用します。すなわち、すべての互換文字を、等価な文字で置換します。正規形 KC (NFKC) は、互換分解を適用してから、標準分解を適用します。

2 つの unicode 文字列が正規化されていて人間の目に同じに見えても、片方が結合文字を持っていたり片方が持っていない場合、それらは完全に同じではありません。

バージョン 2.3 で追加.

更に、本モジュールは以下の定数を公開します:

`unicodedata.unidata_version`

このモジュールで使われている Unicode データベースのバージョン。

バージョン 2.3 で追加.

`unicodedata.ucd_3_2_0`

これはモジュール全体と同じメソッドを具えたオブジェクトですが、Unicode データベースバージョン 3.2 を代わりに使っており、この特定のバージョンの Unicode データベースを必要とするアプリケーション (IDNA など) のためものです。

バージョン 2.5 で追加.

例:

```
>>> import unicodedata
>>> unicodedata.lookup('LEFT CURLY BRACKET')
u'{'
>>> unicodedata.name(u'/')
'SOLIDUS'
>>> unicodedata.decimal(u'9')
9
>>> unicodedata.decimal(u'a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not a decimal
>>> unicodedata.category(u'A') # 'L'etter, 'u'ppercase
'Lu'
>>> unicodedata.bidirectional(u'\u0660') # 'A'rabic, 'N'umber
'AN'
```

7.10 stringprep — インターネットのための文字列調製

バージョン 2.3 で追加.

(ホスト名のような) インターネット上にある存在に識別名をつける際、しばしば識別名間の ”等価性” 比較を行う必要があります。厳密には、例えば大小文字の区別をするかしないかといったように、比較をどのように行うかはアプリケーションの領域に依存します。また、例えば ”印字可能な” 文字で構成された識別名だけを許可するといったように、可能な識別名を制限することも必要となるかもしれません。

RFC 3454 では、インターネットプロトコル上で Unicode 文字列を ”調製 (prepare)” するためのプロシジャを定義しています。文字列は通信路に載せられる前に調製プロシジャで処理され、その結果ある正規化された形式になります。RFC ではあるテーブルの集合を定義しており、それらはプロファイルにまとめられています。各プロファイルでは、どのテーブルを使い、stringprep プロシジャのどのオプション部分がプロファイルの一部になっているかを定義しています。stringprep プロファイルの一つの例は nameprep で、国際化されたドメイン名に使われます。

stringprep は RFC 3453 のテーブルを公開しているに過ぎません。これらのテーブルは辞書やリストとして表現するにはバリエーションが大きすぎるので、このモジュールでは Unicode 文字データベースを内部的に利用しています。モジュールソースコード自体は mkstringprep.py ユーティリティを使って生成されました。

その結果、これらのテーブルはデータ構造体ではなく、関数として公開されています。RFC には 2 種類のテーブル: 集合およびマップ、が存在します。集合については、*stringprep* は ”特性関数 (characteristic function)”、すなわち引数が集合の一部である場合に真を返す関数を提供します。マップに対しては、マップ関数: キーが与えられると、それに関連付けられた値を返す関数、を提供します。以下はこのモジュールで利用可能な全ての関数を列挙したものです。

`stringprep.in_table.al` (*code*)

code がテーブル A.1 (Unicode 3.2 における未割り当てコードポイント: unassigned code point) かどうか判定します。

`stringprep.in_table_b1 (code)`

`code` がテーブル B.1 (一般には何にも対応付けられていない: commonly mapped to nothing) かどうか判定します。

`stringprep.map_table_b2 (code)`

テーブル B.2 (NFKC で用いられる大小文字の対応付け) に従って、`code` に対応付けられた値を返します。

`stringprep.map_table_b3 (code)`

テーブル B.3 (正規化を伴わない大小文字の対応付け) に従って、`code` に対応付けられた値を返します。

`stringprep.in_table_c11 (code)`

`code` がテーブル C.1.1 (ASCII スペース文字) かどうか判定します。

`stringprep.in_table_c12 (code)`

`code` がテーブル C.1.2 (非 ASCII スペース文字) かどうか判定します。

`stringprep.in_table_c11_c12 (code)`

`code` がテーブル C.1 (スペース文字、C.1.1 および C.1.2 の和集合) かどうか判定します。

`stringprep.in_table_c21 (code)`

`code` がテーブル C.2.1 (ASCII 制御文字) かどうか判定します。

`stringprep.in_table_c22 (code)`

`code` がテーブル C.2.2 (非 ASCII 制御文字) かどうか判定します。

`stringprep.in_table_c21_c22 (code)`

`code` がテーブル C.2 (制御文字、C.2.1 および C.2.2 の和集合) かどうか判定します。

`stringprep.in_table_c3 (code)`

`code` がテーブル C.3 (プライベート利用) かどうか判定します。

`stringprep.in_table_c4 (code)`

`code` がテーブル C.4 (非文字コードポイント: non-character code points) かどうか判定します。

`stringprep.in_table_c5 (code)`

`code` がテーブル C.5 (サロゲーションコード) かどうか判定します。

`stringprep.in_table_c6 (code)`

`code` がテーブル C.6 (平文: plain text として不適切) かどうか判定します。

`stringprep.in_table_c7 (code)`

`code` がテーブル C.7 (標準表現: canonical representation として不適切) かどうか判定します。

`stringprep.in_table_c8 (code)`

`code` がテーブル C.8 (表示プロパティの変更または撤廃) かどうか判定します。

`stringprep.in_table_c9 (code)`

`code` がテーブル C.9 (タグ文字) かどうか判定します。

`stringprep.in_table_d1 (code)`

`code` がテーブル D.1 (双方向プロパティ "R" または "AL" を持つ文字) かどうか判定します。

`stringprep.in_table.d2 (code)`

`code` がテーブル D.2 (双方向プロパティ "L" を持つ文字) かどうか判定します。

7.11 `fpformat` — 浮動小数点数の変換

バージョン 2.6 で非推奨: `fpformat` は Python 3 で削除されました。

`fpformat` モジュールは浮動小数点数の表示を 100% 純粋に Python だけで行うための関数を定義しています。

注釈: このモジュールは必要ありません: このモジュールのすべてのことは、[文字列フォーマット操作](#) 節で説明されている `%` を使った文字列の補間演算により実現可能です。

`fpformat` モジュールは次にあげる関数と例外を定義しています。

`fpformat.fix(x, digs)`

`x` を `[-]ddd.ddd` の形にフォーマットします。小数点の後ろに `digs` 桁と、小数点の前に少なくとも 1 桁です。 `digs <= 0` の場合、小数点以下は切り捨てられます。

`x` は数字か数字を表した文字列です。 `digs` は整数です。

返り値は文字列です。

`fpformat.sci(x, digs)`

`x` を `[-]d.dddE[+-]ddd` の形にフォーマットします。小数点の後ろに `digs` 桁と、小数点の前に 1 桁だけです。 `digs <= 0` の場合、1 桁だけ残され、小数点以下は切り捨てられます。

`x` は実数か実数を表した文字列です。 `digs` は整数です。

返り値は文字列です。

exception `fpformat.NotANumber`

`fix()` や `sci()` にパラメータとして渡された文字列 `x` が数字として認識できなかった場合、例外が発生します。標準の例外が文字列の頃から、この例外は `ValueError` のサブクラスです。例外値は、例外を発生させた不適切にフォーマットされた文字列です。

例:

```
>>> import fpformat
>>> fpformat.fix(1.23, 1)
'1.2'
```

第 8 章

データ型

この章で解説されるモジュールは日付や時間、型が固定された配列、ヒープキュー、同期キュー、集合のような種々の特殊なデータ型を提供します。

Python にはその他にもいくつかの組み込みデータ型があります。特に、`dict`、`list`、`set` (`frozenset` とともに古い `sets` モジュールを置き換えます)、そして `tuple` があります。`str` クラスはバイナリデータや 8 ビットテキストを扱うことができ、`unicode` クラスは Unicode テキストを扱うことができます。

この章では以下のモジュールが記述されています:

8.1 `datetime` — 基本的な日付型および時間型

バージョン 2.3 で追加.

`datetime` モジュールでは、日付や時間データを簡単な方法と複雑な方法の両方で操作するためのクラスを提供しています。日付や時刻を対象にした四則演算がサポートされている一方で、このモジュールの実装では出力の書式化や操作を目的とした属性の効率的な取り出しに焦点を絞っています。機能については、`time` および `calendar` も参照下さい。

”naive” と ”aware” の 2 種類の日付と時刻オブジェクトがあります。

aware オブジェクトは他の aware オブジェクトとの相対関係を把握出来るように、タイムゾーンや夏時間の情報のような、アルゴリズム的で政治的な適用可能な時間調節に関する知識を持っています。aware オブジェクトは解釈の余地のない特定の実時刻を表現するのに利用されます^{*1}。

naive オブジェクトには他の日付時刻オブジェクトとの相対関係を把握するのに足る情報が含まれません。あるプログラム内の数字がメートルを表わしているのか、マイルなのか、それとも質量なのかがプログラムによって異なるように、naive オブジェクトが協定世界時 (UTC) なのか、現地時間なのか、それとも他のタイムゾーンなのかはそのプログラムに依存します。Naive オブジェクトはいくつかの現実的な側面を無視してしまうというコストを無視すれば、簡単に理解でき、うまく利用することができます。

aware オブジェクトを必要とするアプリケーションのために、`datetime` と `time` オブジェクトは追加のタイムゾーン情報の属性 `tzinfo` を持ちます。`tzinfo` には抽象クラス `tzinfo` のサブクラスのインスタンス

^{*1} もし相対性理論の効果を無視するならば、ですが

スを設定することができます。これらの `tzinfo` オブジェクトは UTC 時間からのオフセットやタイムゾーンの名前、夏時間が実施されるかの情報を保持しています。 `datetime` モジュールには `tzinfo` の具象クラスは提供されていないので注意してください。より深く詳細までタイムゾーンをサポートするかはアプリケーションに依存します。世界中の時刻の調整を決めるルールは合理的というよりかは政治的なもので、頻繁に変わり、都合のよい基準というものはありません。

`datetime` モジュールでは以下の定数を公開しています:

`datetime.MINYEAR`

`date` や `datetime` オブジェクトで許されている、年を表現する最小の数字です。 `MINYEAR` は 1 です。

`datetime.MAXYEAR`

`date` や `datetime` オブジェクトで許されている、年を表現する最大の数字です。 `MAXYEAR` は 9999 です。

参考:

`calendar` モジュール 汎用のカレンダー関連関数。

`time` モジュール 時刻へのアクセスと変換。

8.1.1 利用可能なデータ型

class `datetime.date`

理想化された naive な日付表現で、実質的には、これまでもこれからも現在のグレゴリオ暦 (Gregorian calender) であると仮定しています。属性: `year`, `month`, および `day`。

class `datetime.time`

理想的な時刻で、特定の日から独立しており、毎日が厳密に 24*60*60 秒であると仮定しています (“うるう秒: leap seconds” の概念はありません)。属性は `hour`, `minute`, `second`, `microsecond`, および `tzinfo` です。

class `datetime.datetime`

日付と時刻を組み合わせたものです。属性は `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond`, および `tzinfo` です。

class `datetime.timedelta`

`date`, `time`, あるいは `datetime` クラスの二つのインスタンス間の時間差をマイクロ秒精度で表す経過時間値です。

class `datetime.tzinfo`

タイムゾーン情報オブジェクトの抽象基底クラスです。 `datetime` および `time` クラスで用いられ、カスタマイズ可能な時刻修正の概念 (たとえばタイムゾーンや夏時間の計算) を提供します。

これらの型のオブジェクトは変更不可能 (immutable) です。

`date` 型のオブジェクトは常に naive です。

`time` や `datetime` 型のオブジェクトは naive か aware のどちらかになります。 `datetime` オブジェクト `d` は `d.tzinfo` が `None` でなく `d.tzinfo.utcoffset(d)` が `None` を返さなければ、aware になります。 `d.tzinfo` が `None` であるか `d.tzinfo` が `None` でないが `d.tzinfo.utcoffset(d)` が `None` を返すと `d` は naive になります。 `time` のオブジェクト `t` は `t.tzinfo` が `None` でなく `t.tzinfo.utcoffset(None)` が `None` を返さないと aware になって、それ以外では `t` は naive になります。

naive なオブジェクトと aware なオブジェクトの区別は `timedelta` オブジェクトにはあてはまりません。

サブクラスの関係は以下のようになります:

```
object
  timedelta
  tzinfo
  time
  date
    datetime
```

8.1.2 timedelta オブジェクト

`timedelta` オブジェクトは経過時間、すなわち二つの日付や時刻間の差を表します。

```
class datetime.timedelta([days[, seconds[, microseconds[, milliseconds[, minutes[, hours[,
                                                                    weeks]]]]]]]])
```

全ての引数がオプションで、デフォルト値は 0 です。引数は整数、長整数、浮動小数点数にすることができ、正でも負でもかまいません。

`days`, `seconds`, `microseconds` だけが内部的に保持されます。引数は以下のようにして変換されます:

- 1 ミリ秒は 1000 マイクロ秒に変換されます。
- 1 分は 60 秒に変換されます。
- 1 時間は 3600 秒に変換されます。
- 1 週間は 7 日に変換されます。

さらに、値が一意に表されるように `days`, `seconds`, `microseconds` が以下のように正規化されます

- `0 <= microseconds < 1000000`
- `0 <= seconds < 3600*24` (一日中の秒数)
- `-999999999 <= days <= 999999999`

引数のいずれかが浮動小数点であり、小数のマイクロ秒が存在する場合、小数のマイクロ秒は全ての引数から一度取り置かれ、それらの和は最も近いマイクロ秒に丸められます。浮動小数点の引数がない場合、値の変換と正規化の過程は厳密な (失われる情報がない) ものとなります。

日の値を正規化した結果、指定された範囲の外側になった場合には、`OverflowError` が送出されます。

負の値を正規化すると、一見混乱するような値になります。例えば、

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

クラス属性を以下に示します:

`timedelta.min`

最小の値を表す *timedelta* オブジェクトで、`timedelta(-999999999)` です。

`timedelta.max`

最大の値を表す *timedelta* オブジェクトで、`timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)` です。

`timedelta.resolution`

timedelta オブジェクトが等しくない最小の時間差で、`timedelta(microseconds=1)` です。

正規化のために、`timedelta.max > -timedelta.min` となるので注意してください。 `-timedelta.max` は *timedelta* オブジェクトとして表現することができません。

インスタンスの属性 (読み込みのみ):

属性	値
<code>days</code>	両端値を含む -999999999 から 999999999 の間
<code>seconds</code>	両端値を含む 0 から 86399 の間
<code>microseconds</code>	両端値を含む 0 から 999999 の間

サポートされている操作を以下に示します:

演算	結果
<code>t1 = t2 + t3</code>	<i>t2</i> と <i>t3</i> の和。演算後、 <code>t1-t2 == t3</code> および <code>t1-t3 == t2</code> は真になります。(1)
<code>t1 = t2 - t3</code>	<i>t2</i> と <i>t3</i> の差。演算後、 <code>t1 == t2 - t3</code> および <code>t2 == t1 + t3</code> は真になります。(1)
<code>t1 = t2 * i</code> or <code>t1 = i * t2</code>	時間差と (長) 整数の積。演算後、 <code>t1 // i == t2</code> は <code>i != 0</code> であれば真となります。 一般的に、 <code>t1 * i == t1 * (i-1) + t1</code> は真となります。(1)
<code>t1 = t2 // i</code>	<code>floor</code> が計算され、余りは (もしあれば) 捨てられます。(3)
<code>+t1</code>	同じ値を持つ <i>timedelta</i> オブジェクトを返します。(2)
<code>-t1</code>	<code>timedelta(-t1.days, -t1.seconds, -t1.microseconds)</code> 、および <code>t1*-1</code> と同じです。(1)(4)
<code>abs(t)</code>	<code>t.days >= 0</code> のときには <code>+t</code> 、 <code>t.days < 0</code> のときには <code>-t</code> となります。(2)
<code>str(t)</code>	<code>[D day[s],][H]H:MM:SS[.UUUUUU]</code> という形式の文字列を返します。 <i>t</i> が負の値の場合は <i>D</i> は負の値となります。(5)
<code>repr(t)</code>	<code>datetime.timedelta(D[, S[, U]])</code> という形式の文字列を返します。 <i>t</i> が負の場合は <i>D</i> は負の値となります。(5)

注釈:

- (1) この演算は正確ですが、オーバーフローするかもしれません。
- (2) この演算は正確であり、オーバーフローしないはずですが。
- (3) 0 による除算は `ZeroDivisionError` を送出します。
- (4) `-timedelta.max` は `timedelta` オブジェクトで表現することができません。
- (5) `timedelta` オブジェクトの文字列表現は内部表現に類似した形に正規化されます。そのため負の `timedelta` はいくぶん珍しい結果となります。例えば:

```
>>> timedelta(hours=-5)
datetime.timedelta(-1, 68400)
>>> print(_)
-1 day, 19:00:00
```

上に列挙した操作に加えて、`timedelta` オブジェクトは `date` および `datetime` オブジェクトとの間で加減算をサポートしています (下を参照してください)。

`timedelta` オブジェクト間の比較はサポートされており、より小さい経過時間を表す `timedelta` オブジェクトがより小さい `timedelta` と見なされます。型混合の比較がデフォルトのオブジェクトアドレス比較となってしまうのを抑止するために、`timedelta` オブジェクトと異なる型のオブジェクトが比較されると、比較演算子が `==` または `!=` でないかぎり `TypeError` が送出されます。後者の場合、それぞれ `False` または `True` を返します。

`timedelta` オブジェクトはハッシュ可能 (*hashable*) つまり、辞書のキーとして利用可能) であり、効率的な pickle 化をサポートします。また、ブール演算コンテキストでは、`timedelta` オブジェクトは `timedelta(0)` に等しくない場合かつそのときに限り真となります。

インスタンスメソッド:

`timedelta.total_seconds()`

この期間に含まれるトータルの秒数を返します。true division が有効な場合の、`(td.microseconds + (td.seconds + td.days * 24 * 3600) * 10**6) / 10**6` と同じです。

非常に長い期間 (多くのプラットフォームでは 270 年以上) については、このメソッドはマイクロ秒の精度を失うことがあることに注意してください。

バージョン 2.7 で追加.

使用例:

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                           minutes=50, seconds=600) # adds up to 365 days
>>> year.total_seconds()
31536000.0
>>> year == another_year
True
```

(次のページに続く)

(前のページからの続き)

```

>>> ten_years = 10 * year
>>> ten_years, ten_years.days // 365
(datetime.timedelta(3650), 10)
>>> nine_years = ten_years - year
>>> nine_years, nine_years.days // 365
(datetime.timedelta(3285), 9)
>>> three_years = nine_years // 3;
>>> three_years, three_years.days // 365
(datetime.timedelta(1095), 3)
>>> abs(three_years - ten_years) == 2 * three_years + year
True

```

8.1.3 date オブジェクト

`date` オブジェクトは日付 (年、月、および日) を表します。日付は理想的なカレンダー、すなわち現在のグレゴリオ暦を過去と未来の両方向に無限に延長したもので表されます。1 年の 1 月 1 日は日番号 1, 1 年 1 月 2 日は日番号 2, となっていくます。この暦法は、全ての計算における基本カレンダーである、Dershowitz と Reingold の書籍 *Calendrical Calculations* における先発グレゴリオ暦 (proleptic Gregorian) の定義に一致します。

class `datetime.date` (*year*, *month*, *day*)

全ての引数が必要です。引数は整数または長整数で、以下の範囲に入らなければなりません:

- `MINYEAR <= year <= MAXYEAR`
- `1 <= month <= 12`
- `1 <= day <= 指定された月と年における日数`

範囲を超えた引数を与えた場合、`ValueError` が送出されます。

他のコンストラクタ、および全てのクラスメソッドを以下に示します:

classmethod `date.today()`

現在のローカルな日付を返します。`date.fromtimestamp(time.time())` と等価です。

classmethod `date.fromtimestamp(timestamp)`

`time.time()` で返されるような POSIX タイムスタンプに対応するローカルな日付を返します。`timestamp` がプラットフォームの C 関数 `localtime()` がサポートする値の範囲から外れていた場合、`ValueError` を送出するかもしれません。この範囲は通常は 1970 年から 2038 年までに制限されています。タイムスタンプの表記にうるう秒を含める非 POSIX なシステムでは、うるう秒は `fromtimestamp()` では無視されます。

classmethod `date.fromordinal(ordinal)`

先発グレゴリオ暦による序数に対応する日付を返します。1 年 1 月 1 日が序数 1 となります。 `1 <= ordinal <= date.max.toordinal()` でない場合、`ValueError` が送出されます。任意の日付 *d* に対し、`date.fromordinal(d.toordinal()) == d` となります。

以下にクラス属性を示します:

`date.min`
表現できる最も古い日付で、`date(MINYEAR, 1, 1)` です。

`date.max`
表現できる最も新しい日付で、`date(MAXYEAR, 12, 31)` です。

`date.resolution`
等しくない日付オブジェクト間の最小の差で、`timedelta(days=1)` です。

インスタンスの属性 (読み込みのみ):

`date.year`
両端値を含む `MINYEAR` から `MAXYEAR` までの値です。

`date.month`
両端値を含む 1 から 12 までの値です。

`date.day`
1 から与えられた月と年における日数までの値です。

サポートされている操作を以下に示します:

演算	結果
<code>date2 = date1 + timedelta</code>	<code>date2</code> はから <code>date1</code> から <code>timedelta.days</code> 日移動した日付です。 (1)
<code>date2 = date1 - timedelta</code>	<code>date2 + timedelta == date1</code> であるような日付 <code>date2</code> を計算します。 (2)
<code>timedelta = date1 - date2</code>	(3)
<code>date1 < date2</code>	<code>date1</code> が時刻として <code>date2</code> よりも前を表す場合に、 <code>date1</code> は <code>date2</code> よりも小さいと見なされます。 (4)

注釈:

- (1) `date2` は `timedelta.days > 0` の場合進む方向に、`timedelta.days < 0` の場合戻る方向に移動します。演算後は、`date2 - date1 == timedelta.days` となります。 `timedelta.seconds` および `timedelta.microseconds` は無視されます。 `date2.year` が `MINYEAR` になってしまったり、`MAXYEAR` より大きくなってしまう場合には `OverflowError` が送出されます。
- (2) この操作は `date1 + (-timedelta)` と等価ではありません。なぜならば、`date1 - timedelta` がオーバーフローしない場合でも、`-timedelta` 単体がオーバーフローする可能性があるからです。 `timedelta.seconds` および `timedelta.microseconds` は無視されます。
- (3) この演算は厳密で、オーバーフローしません。 `timedelta.seconds` および `timedelta.microseconds` は 0 で、演算後には `date2 + timedelta == date1` となります。

- (4) 別の言い方をすると、`date1.toordinal() < date2.toordinal()` であり、かつそのときに限り `date1 < date2` となります。型混合の比較がデフォルトのオブジェクトアドレス比較になってしまうのを抑止するために、`date` オブジェクトと異なる型のオブジェクトが比較されると `TypeError` が送出されます。しかしながら、被比較演算子のもう一方が `timetuple()` 属性を持つ場合には `NotImplemented` が返されます。このフックにより、他種の日付オブジェクトに型混合比較を実装するチャンスを与えています。そうでない場合、`date` オブジェクトと異なる型のオブジェクトが比較されると、比較演算子が `==` または `!=` でないかぎり `TypeError` が送出されます。後者の場合、それぞれ `False` または `True` を返します。

`date` オブジェクトは辞書のキーとして用いることができます。ブール演算コンテキストでは、全ての `date` オブジェクトは真であるとみなされます。

インスタンスメソッド:

`date.replace(year, month, day)`

キーワード引数で指定されたパラメタが置き換えられることを除き、同じ値を持つ `date` オブジェクトを返します。例えば、`d == date(2002, 12, 31)` とすると、`d.replace(day=26) == date(2002, 12, 26)` となります。

`date.timetuple()`

`time.localtime()` が返す形式の `time.struct_time` を返します。時間、分、および秒は 0 で、DST フラグは -1 になります。 `d.timetuple()` は次の値と同値です: `time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1))` ただし `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` が 1 月 1 日に 1 で始まる現在の年の日を表す。

`date.toordinal()`

先発グレゴリオ暦における日付序数を返します。1 年の 1 月 1 日が序数 1 となります。任意の `date` オブジェクト `d` について、`date.fromordinal(d.toordinal()) == d` となります。

`date.weekday()`

月曜日を 0、日曜日を 6 として、曜日を整数で返します。例えば、`date(2002, 12, 4).weekday() == 2` であり、水曜日を示します。 `isoweekday()` も参照してください。

`date.isoweekday()`

月曜日を 1、日曜日を 7 として、曜日を整数で返します。例えば、`date(2002, 12, 4).isoweekday() == 3` であり、水曜日を示します。 `weekday()`、`isocalendar()` も参照してください。

`date.isocalendar()`

3 要素のタプル (ISO 年、ISO 週番号、ISO 曜日) を返します。

ISO カレンダーはグレゴリオ暦の変種として広く用いられています。細かい説明については <https://www.staff.science.uu.nl/~gent0113/calendar/isocalendar.htm> を参照してください。

ISO 年は完全な週が 52 または 53 週あり、週は月曜から始まって日曜に終わります。ISO 年である年における最初の週は、その年の木曜日を含む最初の (グレゴリオ暦での) 週となります。この週は週番号 1 と呼ばれ、この木曜日での ISO 年はグレゴリオ暦における年と等しくなります。

例えば、2004 年は木曜日から始まるため、ISO 年の最初の週は 2003 年 12 月 29 日、月曜日から始まり、2004 年 1 月 4 日、日曜日に終わります。従って、`date(2003, 12, 29).isocalendar()` == (2004, 1, 1) であり、かつ `date(2004, 1, 4).isocalendar()` == (2004, 1, 7) となります。

`date.isoformat()`

ISO 8601 形式、'YYYY-MM-DD' の日付を表す文字列を返します。例えば、`date(2002, 12, 4).isoformat()` == '2002-12-04' となります。

`date.__str__()`

`date` オブジェクト *d* において、`str(d)` は `d.isoformat()` と等価です。

`date.ctime()`

日付を表す文字列を、例えば `date(2002, 12, 4).ctime()` == 'Wed Dec 4 00:00:00 2002' のようにして返します。ネイティブの C 関数 `ctime()` (`time.ctime()` はこの関数を呼び出しますが、`date.ctime()` は呼び出しません) が C 標準に準拠しているプラットフォームでは、`d.ctime()` は `time.ctime(time.mktime(d.timetuple()))` と等価です。

`date.strftime(format)`

明示的な書式文字列で制御された、日付を表現する文字列を返します。時間、分、秒を表す書式化コードは値 0 になります。完全な書式化ディレクティブのリストについては `strftime()` と `strptime()` の振る舞いを参照してください。

`date.__format__(format)`

`date.strftime()` と同等です。これのおかげで `str.format()` を使うときに、`date` のための書式文字列を指定できます。完全な書式化ディレクティブのリストについては `strftime()` と `strptime()` の振る舞いを参照してください。

イベントまでの日数を数える例を示します:

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year + 1)
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202
```

`date` と併用する例を示します:

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 0001
```

(次のページに続く)

(前のページからの続き)

```

>>> d
datetime.date(2002, 3, 11)
>>> t = d.timetuple()
>>> for i in t:
...     print i
2002                # year
3                   # month
11                  # day
0
0
0
0                   # weekday (0 = Monday)
70                  # 70th day in the year
-1
>>> ic = d.isocalendar()
>>> for i in ic:
...     print i
2002                # ISO year
11                  # ISO week number
1                   # ISO day number ( 1 = Monday )
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'

```

8.1.4 datetime オブジェクト

datetime オブジェクトは *date* オブジェクトおよび *time* オブジェクトの全ての情報が入っている単一のオブジェクトです。 *date* オブジェクトと同様に、 *datetime* は現在のグレゴリオ暦が両方向に延長されているものと仮定します; また、 *time* オブジェクトと同様に、 *datetime* は毎日が厳密に 3600*24 秒であると仮定します。

以下にコンストラクタを示します:

```
class datetime.datetime (year, month, day[, hour[, minute[, second[, microsecond[, tzinfo ]]]])
```

年、月、および日の引数は必須です。 *tzinfo* は *None* または *tzinfo* クラスのサブクラスのインスタンスにすることができます。残りの引数は整数または長整数で、以下のような範囲に入ります:

- MINYEAR <= year <= MAXYEAR
- 1 <= month <= 12
- 1 <= day <= 指定された月と年における日数
- 0 <= hour < 24

- `0 <= minute < 60`
- `0 <= second < 60`
- `0 <= microsecond < 1000000`

範囲を超えた引数を与えた場合、`ValueError` が送出されます。

他のコンストラクタ、および全てのクラスメソッドを以下に示します:

classmethod `datetime.today()`

現在のローカルな `datetime` を返します。`tzinfo` は `None` です。この関数は `datetime.fromtimestamp(time.time())` と等価です。`now()`、`fromtimestamp()` も参照してください。

classmethod `datetime.now([tz])`

現在のローカルな日付および時刻を返します。オプションの引数 `tz` が `None` であるか指定されていない場合、このメソッドは `today()` と同様ですが、可能ならば `time.time()` タイムスタンプを通じて得ることができる、より高い精度で時刻を提供します (例えば、プラットフォームが C 関数 `gettimeofday()` をサポートする場合には可能なことがあります)。

`tz` が `None` でない場合、`tz` は `tzinfo` のサブクラスのインスタンスでなければならず、現在の日付および時刻は `tz` のタイムゾーンに変換されます。この場合、結果は `tz.fromutc(datetime.utcnow().replace(tzinfo=tz))` と等価になります。`today()`、`utcnow()` も参照してください。

classmethod `datetime.utcnow()`

`tzinfo` が `None` である現在の UTC の日付および時刻を返します。これは `now()` と似ていますが、naive な `datetime` オブジェクトとして現在の UTC 日付および時刻を返します。`now()` も参照してください。

classmethod `datetime.fromtimestamp(timestamp[, tz])`

`time.time()` が返すような、POSIX タイムスタンプに対応するローカルな日付と時刻を返します。オプションの引数 `tz` が `None` であるか、指定されていない場合、タイムスタンプはプラットフォームのローカルな日付および時刻に変換され、返される `datetime` オブジェクトは naive なものになります。

`tz` が `None` でない場合、`tz` は `tzinfo` のサブクラスのインスタンスでなければならず、現在の日付および時刻は `tz` のタイムゾーンに変換されます。この場合、結果は `tz.fromutc(datetime.utcfrofromtimestamp(timestamp).replace(tzinfo=tz))` と等価になります。

タイムスタンプがプラットフォームの C 関数 `localtime()` や `gmtime()` でサポートされている範囲を超えた場合、`fromtimestamp()` は `ValueError` を送出することがあります。この範囲はよく 1970 年から 2038 年に制限されています。うるう秒がタイムスタンプの概念に含まれている非 POSIX システムでは、`fromtimestamp()` はうるう秒を無視します。このため、秒の異なる二つのタイムスタンプが同一の `datetime` オブジェクトとなることが起こり得ます。`utcfrofromtimestamp()` も参照してください。

classmethod `datetime.utcfrofromtimestamp(timestamp)`

POSIX タイムスタンプに対応する、`tzinfo` が `None` の UTC での `datetime` を返します。タイムスタンプがプラットフォームにおける C 関数 `localtime()` でサポートされている範囲を超えてい

る場合には `ValueError` を送出します。これはたいてい 1970 年から 2038 年に制限されています。
`fromtimestamp()` も参照してください。

classmethod `datetime.fromordinal(ordinal)`

1 年 1 月 1 日を序数 1 とする早期グレゴリオ暦序数に対応する `datetime` オブジェクトを返します。1 ≤ ordinal ≤ `datetime.max.toordinal()` でなければ `ValueError` が送出されます。返されるオブジェクトの時間、分、秒、およびマイクロ秒はすべて 0 で、`tzinfo` は `None` となっています。

classmethod `datetime.combine(date, time)`

日付部分が与えられた `date` オブジェクトと等しく、時刻部分と `tzinfo` 属性が与えられた `time` オブジェクトと等しい、新しい `datetime` オブジェクトを返します。任意の `datetime` オブジェクト `d` で `d == datetime.combine(d.date(), d.timetz())` が成立します。`date` が `datetime` オブジェクトだった場合、その時刻部分と `tzinfo` 属性は無視されます。

classmethod `datetime.strptime(date_string, format)`

`date_string` に対応した `datetime` を返します。`format` にしたがって構文解析されます。これは、`datetime(*(time.strptime(date_string, format)[0:6]))` と等価です。`date_string` と `format` が `time.strptime()` で構文解析できない場合や、この関数が時刻タプルを返してこない場合には `ValueError` を送出します。完全な書式化ディレクティブのリストについては `strptime()` と `strftime()` の振る舞いを参照してください。

バージョン 2.5 で追加。

以下にクラス属性を示します:

`datetime.min`

表現できる最も古い `datetime` で、`datetime(MINYEAR, 1, 1, tzinfo=None)` です。

`datetime.max`

表現できる最も新しい `datetime` で、`datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)` です。

`datetime.resolution`

等しくない `datetime` オブジェクト間の最小の差で、`timedelta(microseconds=1)` です。

インスタンスの属性 (読み込みのみ):

`datetime.year`

両端値を含む `MINYEAR` から `MAXYEAR` までの値です。

`datetime.month`

両端値を含む 1 から 12 までの値です。

`datetime.day`

1 から与えられた月と年における日数までの値です。

`datetime.hour`

in range(24) を満たします。。

`datetime.minute`

`in range(60)` を満たします。

`datetime.second`

`in range(60)` を満たします。

`datetime.microsecond`

`in range(1000000)` を満たします。

`datetime.tzinfo`

`datetime` コンストラクタに `tzinfo` 引数として与えられたオブジェクトになり、何も渡されなかった場合には `None` になります。

サポートされている操作を以下に示します:

演算	結果
<code>datetime2 = datetime1 + timedelta</code>	(1)
<code>datetime2 = datetime1 - timedelta</code>	(2)
<code>timedelta = datetime1 - datetime2</code>	(3)
<code>datetime1 < datetime2</code>	<code>datetime</code> を <code>datetime</code> と比較します。(4)

- (1) `datetime2` は `datetime1` から時間 `timedelta` 移動したもので、`timedelta.days > 0` の場合未来へ、`timedelta.days < 0` の場合過去へ移動します。結果は入力 of `datetime` と同じ `tzinfo` 属性を持ち、演算後には `datetime2 - datetime1 == timedelta` となります。`datetime2.year` が `MINYEAR` よりも小さいか、`MAXYEAR` より大きい場合には `OverflowError` が送出されます。入力が aware なオブジェクトの場合でもタイムゾーン修正は全く行われません。

- (2) `datetime2 + timedelta == datetime1` となるような `datetime2` を計算します。ちなみに、結果は入力の `datetime` と同じ `tzinfo` 属性を持ち、入力が aware でもタイムゾーン修正は全く行われません。この操作は `date1 + (-timedelta)` と等価ではありません。なぜならば、`date1 - timedelta` がオーバーフローしない場合でも、`-timedelta` 単体がオーバーフローする可能性があるからです。

- (3) `datetime` から `datetime` の減算は両方の被演算子が naive であるか、両方とも aware である場合にのみ定義されています。片方が aware でもう一方が naive の場合、`TypeError` が送出されます。

両方とも naive か、両方とも aware で同じ `tzinfo` 属性を持つ場合、`tzinfo` 属性は無視され、結果は `datetime2 + t == datetime1` であるような `timedelta` オブジェクト `t` となります。この場合タイムゾーン修正は全く行われません。

両方が aware で異なる `tzinfo` 属性を持つ場合、`a-b` は `a` および `b` をまず naive な UTC `datetime` オブジェクトに変換したかのようにして行います。演算結果は決してオーバーフローを起こさないことを除き、`(a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())` と同じになります。

- (4) `datetime1` が時刻として `datetime2` よりも前を表す場合に、`datetime1` は `datetime2` よりも小さいと見なされます。

native と aware を比較しようとする `TypeError` が送出されます。比較対象が両方とも aware であ

り、同じ `tzinfo` 属性を持つ場合、`tzinfo` は無視され `datetime` だけで比較が行われます。比較対象が両方とも `aware` であり、異なる `tzinfo` 属性を持つ場合、まず最初に (`self.utcoffset()` で取得できる) それぞれの UTC オフセットを引く調整が行われます。

注釈: 型混合の比較がデフォルトのオブジェクトアドレス比較となってしまうのを抑止するために、被演算子のもう一方が `datetime` オブジェクトと異なる型のオブジェクトの場合には `TypeError` が送出されます。しかしながら、被比較演算子のもう一方が `timetuple()` 属性を持つ場合には `NotImplemented` が返されます。このフックにより、他種の日付オブジェクトに型混合比較を実装するチャンスを与えています。そうでない場合、`datetime` オブジェクトと異なる型のオブジェクトが比較されると、比較演算子が `==` または `!=` でないかぎり `TypeError` が送出されます。後者の場合、それぞれ `False` または `True` を返します。

`datetime` オブジェクトは辞書のキーとして用いることができます。ブール演算コンテキストでは、全ての `datetime` オブジェクトは真であるとみなされます。

インスタンスメソッド:

`datetime.date()`

同じ年、月、日の `date` オブジェクトを返します。

`datetime.time()`

同じ時、分、秒、マイクロ秒を持つ `time` オブジェクトを返します。 `tzinfo` は `None` です。 `timetz()` も参照してください。

`datetime.timetz()`

同じ時、分、秒、マイクロ秒、および `tzinfo` 属性を持つ `time` オブジェクトを返します。 `time()` メソッドも参照してください。

`datetime.replace([year[, month[, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]]]])`

キーワード引数で指定した属性の値を除き、同じ属性をもつ `datetime` オブジェクトを返します。メンバに対する変換を行わずに `aware` な `datetime` オブジェクトから `naive` な `datetime` オブジェクトを生成するために、`tzinfo=None` を指定することもできます。

`datetime.astimezone(tz)`

`tz` を新たに `tzinfo` 属性として持つ `datetime` オブジェクトを返します。日付および時刻データを調整して、戻り値が `self` と同じ UTC 時刻を持ち、`tz` におけるローカルな時刻を表すようにします。

`tz` は `tzinfo` のサブクラスのインスタンスでなければならず、インスタンスの `utcoffset()` および `dst()` メソッドは `None` を返してはなりません。 `self` は `aware` でなくてはなりません (`self.tzinfo` が `None` であってはならず、かつ `self.utcoffset()` は `None` を返してはなりません)。

`self.tzinfo` が `tz` の場合、`self.astimezone(tz)` は `self` に等しくなります: 日付および時刻データメンバに対する調整は行われません。そうでない場合、結果はタイムゾーン `tz` におけるローカル時刻で、`self` と同じ UTC 時刻を表すようになります: `astz = dt.astimezone(tz)` とした後、`astz - astz.utcoffset()` は通常 `dt - dt.utcoffset()` と同じ日付および時刻データメンバを持ちます。 `tzinfo` クラスに関する議論では、夏時間 (Daylight Saving time) の遷移境界では上の

等価性が成り立たないことを説明しています (*tz* が標準時と夏時間の両方をモデル化している場合のみ)の問題です)。

単にタイムゾーンオブジェクト *tz* を *datetime* オブジェクト *dt* に追加したいだけで、日付や時刻データへの調整を行わないのなら、`dt.replace(tzinfo=tz)` を使ってください。単に *aware* な *datetime* オブジェクト *dt* からタイムゾーンオブジェクトを除去したいだけで、日付や時刻データの変換を行わないのなら、`dt.replace(tzinfo=None)` を使ってください。

デフォルトの `tzinfo.fromutc()` メソッドを *tzinfo* のサブクラスで上書きして、`astimezone()` が返す結果に影響を及ぼすことができます。エラーの場合を無視すると、`astimezone()` は以下のように動作します:

```
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new time zone object.
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```

`datetime.utcoffset()`

tzinfo が *None* の場合、*None* を返し、そうでない場合には `self.tzinfo.utcoffset(self)` を返します。後者の式が *None* か、1 日以下の大きさを持つ経過時間を表す *timedelta* オブジェクトのいずれかを返さない場合には例外を送出します。

`datetime.dst()`

tzinfo が *None* の場合 *None* を返し、そうでない場合には `self.tzinfo.dst(self)` を返します。後者の式が *None* もしくは、1 日未満の経過時間を表す *timedelta* オブジェクトのいずれかを返さない場合には例外を送出します。

`datetime.tzname()`

tzinfo が *None* の場合 *None* を返し、そうでない場合には `self.tzinfo.tzname(self)` を返します。後者の式が *None* か文字列オブジェクトのいずれかを返さない場合には例外を送出します。

`datetime.timetuple()`

`time.localtime()` が返す形式の `time.struct_time` を返します。`d.timetuple()` は `time.struct_time((d.year, d.month, d.day, d.hour, d.minute, d.second, d.weekday(), yday, dst))` と等価です。ここで `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` はその年の 1 月 1 日を 1 としたときのその日の位置です。返されるタプルの `tm_isdst` フラグは `dst()` メソッドに従って設定されます: *tzinfo* が *None* か `dst()` が *None* を返す場合、`tm_isdst` は -1 に設定されます; そうでない場合、`dst()` がゼロでない値を返すと `tm_isdst` は 1 となります; それ以外の場合には `tm_isdst` は 0 に設定されます。

`datetime.utctimetuple()`

datetime インスタンス *d* が *naive* の場合、このメソッドは `d.timetuple()` と同じであり、`d.dst()` の返す内容にかかわらず `tm_isdst` が 0 に強制される点だけが異なります。DST が UTC 時刻に影響を及ぼすことは決してありません。

d が *aware* の場合、*d* から `d.utcoffset()` が差し引かれて UTC 時刻に正規化され、正規化された

時刻の `time.struct_time` を返します。 `tm.isdst` は 0 に強制されます。 `d.year` が `MINYEAR` や `MAXYEAR` で、UTC への修正の結果表現可能な年の境界を越えた場合には、戻り値の `tm_year` メンバは `MINYEAR-1` または `MAXYEAR+1` になることがあります。

`datetime.toordinal()`

先発グレゴリオ暦における日付序数を返します。 `self.date().toordinal()` と同じです。

`datetime.weekday()`

月曜日を 0、日曜日を 6 として、曜日を整数で返します。 `self.date().weekday()` と同じです。
`isoweekday()` も参照してください。

`datetime.isoweekday()`

月曜日を 1、日曜日を 7 として、曜日を整数で返します。 `self.date().isoweekday()` と等価です。
`weekday()`、`isocalendar()` も参照してください。

`datetime.isocalendar()`

3 要素のタプル (ISO 年、ISO 週番号、ISO 曜日) を返します。 `self.date().isocalendar()` と等価です。

`datetime.isoformat([sep])`

日付と時刻を ISO 8601 形式、すなわち `YYYY-MM-DDTHH:MM:SS.mmmmmm` か、`microsecond` が 0 の場合には `YYYY-MM-DDTHH:MM:SS` で表した文字列を返します。

`utcoffset()` が `None` を返さない場合、UTC からのオフセットを時間と分を表した (符号付きの) 6 文字からなる文字列が追加されます: すなわち、`YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM` となるか、`microsecond` がゼロの場合には `YYYY-MM-DDTHH:MM:SS+HH:MM` となります。

オプションの引数 `sep` (デフォルトでは `'T'` です) は 1 文字のセパレータで、結果の文字列の日付と時刻の間に置かれます。例えば、

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     def utcoffset(self, dt): return timedelta(minutes=-399)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
```

`datetime.__str__()`

`datetime` オブジェクト `d` において、`str(d)` は `d.isoformat(' ')` と等価です。

`datetime.ctime()`

日付を表す文字列を、例えば `datetime(2002, 12, 4, 20, 30, 40).ctime() == 'Wed Dec 4 20:30:40 2002'` のようにして返します。ネイティブの C 関数 `ctime()` (`time.ctime()` はこの関数を呼び出しますが、`datetime.ctime()` は呼び出しません) が C 標準に準拠しているプラットフォームでは、`d.ctime()` は `time.ctime(time.mktime(d.timetuple()))` と等価です。

`datetime.strftime(format)`

明示的な書式化文字列で制御された、日付および時刻を表現する文字列を返します。完全な書式化ディ

レクティブのリストについては `strftime()` と `strptime()` の振る舞いを参照してください。

`datetime.__format__(format)`

`datetime.strftime()` と同等です。これのおかげで `str.format()` を使うときに、`datetime` のための書式文字列を指定できます。完全な書式化ディレクティブのリストについては `strftime()` と `strptime()` の振る舞いを参照してください。

`datetime` オブジェクトを使う例:

```
>>> from datetime import datetime, date, time
>>> # Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)
>>> # Using datetime.now() or datetime.utcnow()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043) # GMT +1
>>> datetime.utcnow()
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060)
>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)
>>> # Using datetime.timetuple() to get tuple of all attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print it
...
2006      # year
11        # month
21        # day
16        # hour
30        # minute
0         # second
1         # weekday (0 = Monday)
325       # number of days since 1st January
-1        # dst - method tzinfo.dst() returned None
>>> # Date in ISO format
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print it
...
2006      # ISO year
47        # ISO week
2         # ISO weekday
>>> # Formatting datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}.'.format(dt, "day
↪", "month", "time")
'The day is 21, the month is November, the time is 04:30PM.'
```

datetime を tzinfo と組み合わせて使う:

```
>>> from datetime import timedelta, datetime, tzinfo
>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1) + self.dst(dt)
...     def dst(self, dt):
...         # DST starts last Sunday in March
...         d = datetime(dt.year, 4, 1) # ends last Sunday in October
...         self.dston = d - timedelta(days=d.weekday() + 1)
...         d = datetime(dt.year, 11, 1)
...         self.dstoff = d - timedelta(days=d.weekday() + 1)
...         if self.dston <= dt.replace(tzinfo=None) < self.dstoff:
...             return timedelta(hours=1)
...         else:
...             return timedelta(0)
...     def tzname(self, dt):
...         return "GMT +1"
...
>>> class GMT2(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=2) + self.dst(dt)
...     def dst(self, dt):
...         d = datetime(dt.year, 4, 1)
...         self.dston = d - timedelta(days=d.weekday() + 1)
...         d = datetime(dt.year, 11, 1)
...         self.dstoff = d - timedelta(days=d.weekday() + 1)
...         if self.dston <= dt.replace(tzinfo=None) < self.dstoff:
...             return timedelta(hours=1)
...         else:
...             return timedelta(0)
...     def tzname(self, dt):
...         return "GMT +2"
...
>>> gmt1 = GMT1()
>>> # Daylight Saving Time
>>> dt1 = datetime(2006, 11, 21, 16, 30, tzinfo=gmt1)
>>> dt1.dst()
datetime.timedelta(0)
>>> dt1.utcoffset()
datetime.timedelta(0, 3600)
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=gmt1)
>>> dt2.dst()
datetime.timedelta(0, 3600)
>>> dt2.utcoffset()
datetime.timedelta(0, 7200)
>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(GMT2())
>>> dt3
datetime.datetime(2006, 6, 14, 14, 0, tzinfo=<GMT2 object at 0x...>)
>>> dt2
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=<GMT1 object at 0x...>)
>>> dt2.utctimetuple() == dt3.utctimetuple()
```

(次のページに続く)

(前のページからの続き)

True

8.1.5 `time` オブジェクト

`time` オブジェクトは (ローカルの) 日中時刻を表現します。この時刻表現は特定の日の影響を受けず、`tzinfo` オブジェクトを介した修正の対象となります。

```
class datetime.time([hour[, minute[, second[, microsecond[, tzinfo]]]])
```

全ての引数はオプションです。 `tzinfo` は `None` または `tzinfo` クラスのサブクラスのインスタンスにすることができます。残りの引数は整数または長整数で、以下のような範囲に入ります:

- `0 <= hour < 24`
- `0 <= minute < 60`
- `0 <= second < 60`
- `0 <= microsecond < 1000000.`

引数がこれらの範囲外にある場合、`ValueError` が送出されます。 `tzinfo` のデフォルト値が `None` である以外のデフォルト値は `0` です。

以下にクラス属性を示します:

`time.min`

表現できる最も古い `time` で、`time(0, 0, 0, 0)` です。

`time.max`

表現できる最も新しい `time` で、`time(23, 59, 59, 999999)` です。

`time.resolution`

等しくない `time` オブジェクト間の最小の差で、`timedelta(microseconds=1)` ですが、`time` オブジェクト間の四則演算はサポートされていないので注意してください。

インスタンスの属性 (読み込みのみ):

`time.hour`

`in range(24)` を満たします。。

`time.minute`

`in range(60)` を満たします。

`time.second`

`in range(60)` を満たします。

`time.microsecond`

`in range(1000000)` を満たします。

`time.tzinfo`

`time` コンストラクタに `tzinfo` 引数として与えられたオブジェクトになり、何も渡されなかった場合には `None` になります。

サポートされている操作を以下に示します:

- a が時間的に b に先行している場合に a は b より小さいとみなす、`time` と `time` の比較。native と aware を比較しようすると `TypeError` が送出されます。比較対象が両方とも aware であり、同じ `tzinfo` 属性を持つ場合、`tzinfo` は無視され datetime だけで比較が行われます。比較対象が両方とも aware であり、異なる `tzinfo` 属性を持つ場合、まず最初に (`self.utcoffset()`) で取得できる) それぞれの UTC オフセットを引く調整が行われます。型混合の比較がデフォルトのオブジェクトアドレス比較となってしまうのを抑止するために、`time` オブジェクトと異なる型のオブジェクトが比較されると、比較演算子が `==` または `!=` でないかぎり `TypeError` が送出されます。後者の場合、それぞれ `False` または `True` を返します。
- ハッシュ化、辞書のキーとしての利用
- 効率的な pickle 化
- ブール演算コンテキストでは、`time` オブジェクトは、分に変換し、`utcoffset()` (`None` を返した場合には 0) を差し引いて変換した後の結果がゼロでない場合、かつそのときに限って真とみなされます。

インスタンスメソッド:

`time.replace([hour[, minute[, second[, microsecond[, tzinfo]]]])`

キーワード引数で指定したメンバの値を除き、同じ値をもつ `time` オブジェクトを返します。データに対する変換を行わずに aware な `time` オブジェクトから naive な `time` オブジェクトを生成するために、`tzinfo=None` を指定することもできます。

`time.isoformat()`

日付と時刻を ISO 8601 形式、すなわち HH:MM:SS.mmmmmm か、`microsecond` が 0 の場合には HH:MM:SS で表した文字列を返します。`utcoffset()` が `None` を返さない場合、UTC からのオフセットを時間と分を表した (符号付きの) 6 文字からなる文字列が追加されます: すなわち、HH:MM:SS.mmmmmm+HH:MM となるか、`microsecond` が 0 の場合には HH:MM:SS+HH:MM となります

`time.__str__()`

`time` オブジェクト t において、`str(t)` は `t.isoformat()` と等価です。

`time.strftime(format)`

明示的な書式文字列で制御された、時刻を表現する文字列を返します。完全な書式化ディレクティブのリストについては `strftime()` と `strptime()` の振る舞いを参照してください。

`time.__format__(format)`

`time.strftime()` と同等です。これのおかげで `str.format()` を使うときに、`time` のための書式文字列を指定できます。完全な書式化ディレクティブのリストについては `strftime()` と `strptime()` の振る舞いを参照してください。

`time.utcoffset()`

`tzinfo` が `None` の場合 `None` を返し、そうでない場合には `self.tzinfo.utcoffset(None)`

を返します。後者の式が `None` もしくは、1 日未満の経過時間を表す `timedelta` オブジェクトのいずれかを返さない場合には例外を送出します。

`time.dst()`

`tzinfo` が `None` の場合 `None` を返し、そうでない場合には `self.tzinfo.dst(None)` を返します。後者の式が `None` もしくは、1 日未満の経過時間を表す `timedelta` オブジェクトのいずれかを返さない場合には例外を送出します。

`time.tzname()`

`tzinfo` が `None` の場合 `None` を返し、そうでない場合には `self.tzinfo.tzname(None)` を返します。後者の式が `None` か文字列オブジェクトのいずれかを返さない場合には例外を送出します。

例:

```
>>> from datetime import time, tzinfo, timedelta
>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self, dt):
...         return "Europe/Prague"
...
>>> t = time(12, 10, 30, tzinfo=GMT1())
>>> t
datetime.time(12, 10, 30, tzinfo=<GMT1 object at 0x...>)
>>> gmt = GMT1()
>>> t.isoformat()
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
>>> t.tzname()
'Europe/Prague'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 Europe/Prague'
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'
```

8.1.6 tzinfo オブジェクト

class `datetime.tzinfo`

これは抽象基底クラスです。つまり、このクラスは直接インスタンス化して利用しません。具体的なサブクラスを派生し、(少なくとも) 利用したい `datetime` のメソッドが必要とする `tzinfo` の標準メソッドを実装してやる必要があります。 `datetime` モジュールでは、 `tzinfo` の具体的なサブクラスは何ら提供していません。

`tzinfo` (の具体的なサブクラス) のインスタンスは `datetime` および `time` オブジェクトのコンストラクタに渡すことができます。後者のオブジェクトでは、データ属性をローカル時刻におけるものとして見ており、 `tzinfo` オブジェクトはローカル時刻の UTC からのオフセット、タイムゾーンの名前、

DST オフセットを、渡された日付および時刻オブジェクトからの相対で示すためのメソッドを提供します。

pickle 化についての特殊な要求事項: `tzinfo` のサブクラスは引数なしで呼び出すことのできる `__init__()` メソッドを持たなければなりません。そうでなければ、pickle 化することはできますがおそらく unpickle 化することはできないでしょう。これは技術的な側面からの要求であり、将来緩和されるかもしれません。

`tzinfo` の具体的なサブクラスでは、以下のメソッドを実装する必要があります。厳密にどのメソッドが必要なのかは、aware な `datetime` オブジェクトがこのサブクラスのインスタンスをどのように使うかに依存します。不確かならば、単に全てを実装してください。

`tzinfo.utcoffset(self, dt)`

ローカル時間の UTC からのオフセットを、UTC から東向きを正とした分で返します。ローカル時間が UTC の西側にある場合、この値は負になります。このメソッドは UTC からのオフセットの総計を返すように意図されているので注意してください; 例えば、`tzinfo` オブジェクトがタイムゾーンと DST 修正の両方を表現する場合、`utcoffset()` はそれらの合計を返さなければなりません。UTC オフセットが未知である場合、`None` を返してください。そうでない場合には、返される値は -1439 から 1439 の両端を含む値 ($1440 = 24 \times 60$; つまり、オフセットの大きさは 1 日より短くなくてはなりません) が分で指定された `timedelta` オブジェクトでなければなりません。ほとんどの `utcoffset()` 実装は、おそらく以下の二つのうちの一つに似たものになるでしょう:

```
return CONSTANT                # fixed-offset class
return CONSTANT + self.dst(dt) # daylight-aware class
```

`utcoffset()` が `None` を返さない場合、`dst()` も `None` を返してはなりません。

`utcoffset()` のデフォルトの実装は `NotImplementedError` を送出します。

`tzinfo.dst(self, dt)`

夏時間 (DST) 修正を、UTC から東向きを正とした分で返します。DST 情報が未知の場合、`None` が返されます。DST が有効でない場合には `timedelta(0)` を返します。DST が有効の場合、オフセットは `timedelta` オブジェクトで返します (詳細は `utcoffset()` を参照してください)。DST オフセットが利用可能な場合、この値は `utcoffset()` が返す UTC からのオフセットには既に加算されているため、DST を個別に取得する必要がない限り `dst()` を使って問い合わせる必要はないので注意してください。例えば、`datetime.timetuple()` は `tzinfo` 属性の `dst()` メソッドを呼んで `tm.isdst` フラグがセットされているかどうか判断し、`tzinfo.fromutc()` は `dst()` タイムゾーンを移動する際に DST による変更があるかどうかを調べます。

標準および夏時間の両方をモデル化している `tzinfo` サブクラスのインスタンス `tz` は以下の式:

$$tz.utcoffset(dt) - tz.dst(dt)$$

が、`dt.tzinfo == tz` 全ての `datetime` オブジェクト `dt` について常に同じ結果を返さなければならないという点で、一貫性を持っていなければなりません。正常に実装された `tzinfo` のサブクラスでは、この式はタイムゾーンにおける "標準オフセット (standard offset)" を表し、特定の日や時刻の事情ではなく地理的な位置にのみ依存してはなりません。`datetime.astimezone()` の実装はこの事実に依存していますが、違反を検出することができません; 正しく実装するのはプログラ

マの責任です。 `tzinfo` のサブクラスでこれを保証することができない場合、 `tzinfo.fromutc()` の実装をオーバーライドして、 `astimezone()` に関わらず正しく動作するようにしてもかまいません。

ほとんどの `dst()` 実装は、おそらく以下の二つのうちの一つに似たものになるでしょう:

```
def dst(self, dt):
    # a fixed-offset class: doesn't account for DST
    return timedelta(0)
```

もしくは

```
def dst(self, dt):
    # Code to set dston and dstoff to the time zone's DST
    # transition times based on the input dt.year, and expressed
    # in standard local time. Then

    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

デフォルトの `dst()` 実装は `NotImplementedError` を送出します。

`tzinfo.tzname(self, dt)`

`datetime` オブジェクト `dt` に対応するタイムゾーン名を文字列で返します。 `datetime` モジュールでは文字列名について何も定義しておらず、特に何かを意味するといった要求仕様もまったくありません。例えば、`"GMT"`、`"UTC"`、`"-500"`、`"-5:00"`、`"EDT"`、`"US/Eastern"`、`"America/New York"` は全て有効な応答となります。文字列名が未知の場合には `None` を返してください。 `tzinfo` のサブクラスでは、特に、`tzinfo` クラスが夏時間について記述している場合のように、渡された `dt` の特定の値によって異なった名前を返したい場合があるため、文字列値ではなくメソッドとなっていることに注意してください。

デフォルトの `tzname()` 実装は `NotImplementedError` を送出します。

以下のメソッドは `datetime` や `time` オブジェクトにおいて、同名のメソッドが呼び出された際に応じて呼び出されます。 `datetime` オブジェクトは自身を引数としてメソッドに渡し、 `time` オブジェクトは引数として `None` をメソッドに渡します。従って、 `tzinfo` のサブクラスにおけるメソッドは引数 `dt` が `None` の場合と、 `datetime` の場合を受理するように用意しなければなりません。

`None` が渡された場合、最良の応答方法を決めるのはクラス設計者次第です。例えば、このクラスが `tzinfo` プロトコルと関係をもたないということを表明させたいければ、 `None` が適切です。標準時のオフセットを見つける他の手段がない場合には、標準 UTC オフセットを返すために `utcoffset(None)` を使うのもっと便利かもしれません。

`datetime` オブジェクトが `datetime()` メソッドの応答として返された場合、 `dt.tzinfo` は `self` と同じオブジェクトになります。ユーザが直接 `tzinfo` メソッドを呼び出さないかぎり、 `tzinfo` メソッドは `dt.tzinfo` と `self` が同じであることに依存します。その結果 `tzinfo` メソッドは `dt` がローカル時間であると解釈するので、他のタイムゾーンでのオブジェクトの振る舞いについて心配する必要がありません。

サブクラスでオーバーライドすると良い、もう 1 つの `tzinfo` のメソッドがあります:

`tzinfo.fromutc(self, dt)`

デフォルトの `datetime.astimezone()` 実装で呼び出されます。 `datetime.astimezone()` から呼ばれた場合、 `dt.tzinfo` は `self` であり、 `dt` の日付および時刻データは UTC 時刻を表しているものとして見えます。 `fromutc()` の目的は、 `self` のローカル時刻に等しい `datetime` オブジェクトを返すことにより日付と時刻データメンバを修正することにあります。

ほとんどの `tzinfo` サブクラスではデフォルトの `fromutc()` 実装を問題なく継承できます。デフォルトの実装は、固定オフセットのタイムゾーンや、標準時と夏時間の両方について記述しているタイムゾーン、そして DST 移行時刻が年によって異なる場合でさえ、扱えるくらい強力なものです。デフォルトの `fromutc()` 実装が全ての場合に対して正しく扱うことができないような例は、標準時の (UTC からの) オフセットが引数として渡された特定の日や時刻に依存するもので、これは政治的な理由によって起きることがあります。デフォルトの `astimezone()` や `fromutc()` の実装は、結果が標準時オフセットの変化にまたがる何時間かの中にある場合、期待通りの結果を生成しないかもしれません。

エラーの場合のためのコードを除き、デフォルトの `fromutc()` の実装は以下のように動作します:

```
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst # this is self's standard offset
    if delta:
        dt += delta # convert to standard local time
        dtdst = dt.dst()
        # raise ValueError if dtdst is None
    if dtdst:
        return dt + dtdst
    else:
        return dt
```

以下に `tzinfo` クラスの使用例を示します:

```
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)

# A UTC class.

class UTC(tzinfo):
    """UTC"""

    def utcoffset(self, dt):
        return ZERO

    def tzname(self, dt):
        return "UTC"
```

(次のページに続く)

(前のページからの続き)

```

    def dst(self, dt):
        return ZERO

utc = UTC()

# A class building tzinfo objects for fixed-offset time zones.
# Note that FixedOffset(0, "UTC") is a different way to build a
# UTC tzinfo object.

class FixedOffset(tzinfo):
    """Fixed offset in minutes east from UTC."""

    def __init__(self, offset, name):
        self.__offset = timedelta(minutes = offset)
        self.__name = name

    def utcoffset(self, dt):
        return self.__offset

    def tzname(self, dt):
        return self.__name

    def dst(self, dt):
        return ZERO

# A class capturing the platform's idea of local time.

import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
if _time.daylight:
    DSTOFFSET = timedelta(seconds = -_time.altzone)
else:
    DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):

    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET

    def dst(self, dt):
        if self._isdst(dt):
            return DSTDIFF
        else:
            return ZERO

```

(次のページに続く)

(前のページからの続き)

```

def tzname(self, dt):
    return _time.tzname[self._isdst(dt)]

def _isdst(self, dt):
    tt = (dt.year, dt.month, dt.day,
          dt.hour, dt.minute, dt.second,
          dt.weekday(), 0, 0)
    stamp = _time.mktime(tt)
    tt = _time.localtime(stamp)
    return tt.tm_isdst > 0

Local = LocalTimezone()

# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt

# US DST Rules
#
# This is a simplified (i.e., wrong for a few cases) set of rules for US
# DST start and end times. For a complete and up-to-date set of DST rules
# and timezone definitions, visit the Olson Database (or try pytz):
# http://www.twinsun.com/tz/tz-link.htm
# http://sourceforge.net/projects/pytz/ (might not be up-to-date)
#
# In the US, since 2007, DST starts at 2am (standard time) on the second
# Sunday in March, which is the first Sunday on or after Mar 8.
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time; 1am standard time) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 1)
# From 1987 to 2006, DST used to start at 2am (standard time) on the first
# Sunday in April and to end at 2am (DST time; 1am standard time) on the last
# Sunday of October, which is the first Sunday on or after Oct 25.
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
DSTEND_1987_2006 = datetime(1, 10, 25, 1)
# From 1967 to 1986, DST used to start at 2am (standard time) on the last
# Sunday in April (the one on or after April 24) and to end at 2am (DST time;
# 1am standard time) on the last Sunday of October, which is the first Sunday
# on or after Oct 25.
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)
DSTEND_1967_1986 = DSTEND_1987_2006

class USTimeZone(tzinfo):

    def __init__(self, hours, reprname, stdname, dstname):

```

(次のページに続く)

(前のページからの続き)

```

self.stdoffset = timedelta(hours=hours)
self.reprname = reprname
self.stdname = stdname
self.dstname = dstname

def __repr__(self):
    return self.reprname

def tzname(self, dt):
    if self.dst(dt):
        return self.dstname
    else:
        return self.stdname

def utcoffset(self, dt):
    return self.stdoffset + self.dst(dt)

def dst(self, dt):
    if dt is None or dt.tzinfo is None:
        # An exception may be sensible here, in one or both cases.
        # It depends on how you want to treat them. The default
        # fromutc() implementation (called by the default astimezone()
        # implementation) passes a datetime with dt.tzinfo is self.
        return ZERO
    assert dt.tzinfo is self

    # Find start and end times for US DST. For years before 1967, return
    # ZERO for no DST.
    if 2006 < dt.year:
        dststart, dstend = DSTSTART_2007, DSTEND_2007
    elif 1986 < dt.year < 2007:
        dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
    elif 1966 < dt.year < 1987:
        dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
    else:
        return ZERO

    start = first_sunday_on_or_after(dststart.replace(year=dt.year))
    end = first_sunday_on_or_after(dstend.replace(year=dt.year))

    # Can't compare naive to aware objects, so strip the timezone from
    # dt first.
    if start <= dt.replace(tzinfo=None) < end:
        return HOUR
    else:
        return ZERO

Eastern = USTimeZone(-5, "Eastern", "EST", "EDT")
Central = USTimeZone(-6, "Central", "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific = USTimeZone(-8, "Pacific", "PST", "PDT")

```


標準時間 (standard time) および夏時間 (daylight time) の両方を記述している `tzinfo` のサブクラスでは、回避不能の難解な問題が年に 2 度あるので注意してください。具体的な例として、東部アメリカ時刻 (US Eastern, UTC -5000) を考えます。EDT は 3 月の第二日曜日の 1:59 (EST) 以後に開始し、11 月の最初の日曜日の 1:59 (EDT) に終了します:

UTC	3:MM	4:MM	5:MM	6:MM	7:MM	8:MM
EST	22:MM	23:MM	0:MM	1:MM	2:MM	3:MM
EDT	23:MM	0:MM	1:MM	2:MM	3:MM	4:MM
start	22:MM	23:MM	0:MM	1:MM	3:MM	4:MM
end	23:MM	0:MM	1:MM	1:MM	2:MM	3:MM

DST の開始の際 ("start" の並び) ローカルの壁時計は 1:59 から 3:00 に飛びます。この日は 2:MM の形式をとる時刻は実際には無意味となります。従って、`astimezone(Eastern)` は DST が開始する日には `hour == 2` となる結果を返すことはありません。 `astimezone()` がこのことを保証するようにするには、`tzinfo.dst()` メソッドは "失われた時間" (東部時刻における 2:MM) が夏時間に存在することを考えなければなりません。

DST が終了する際 ("end" の並び) では、問題はさらに悪化します: 1 時間の間、ローカルの壁時計ではっきりと時刻をいえなくなります: それは夏時間の最後の 1 時間です。東部時刻では、その日の UTC での 5:MM に夏時間は終了します。ローカルの壁時計は 1:59 (夏時間) から 1:00 (標準時) に再び巻き戻されます。ローカルの時刻における 1:MM はあいまいになります。 `astimezone()` は二つの UTC 時刻を同じローカルの時刻に対応付けることでローカルの時計の振る舞いをまねます。東部時刻の例では、5:MM および 6:MM の形式をとる UTC 時刻は両方とも、東部時刻に変換された際に 1:MM に対応づけられます。 `astimezone()` がこのことを保証するようにするには、`tzinfo.dst()` は "繰り返された時間" が標準時に存在することを考慮しなければなりません。このことは、例えばタイムゾーンの標準のローカルな時刻に DST への切り替え時刻を表現することで簡単に設定することができます。

このようなあいまいさを許容できないアプリケーションは、ハイブリッドな `tzinfo` サブクラスを使って問題を回避しなければなりません; UTC や、他のオフセットが固定された `tzinfo` のサブクラス (EST (-5 時間の固定オフセット) のみを表すクラスや、EDT (-4 時間の固定オフセット) のみを表すクラス) を使う限り、あいまいさは発生しません。

参考:

pytz 標準ライブラリには `tzinfo` インスタンスはありませんが、サードパーティーのライブラリで (Olson データベースとしても知られる) IANA タイムゾーンデータベースを Python に提供するものが存在します: それが `pytz` です。

`pytz` は最新の情報を含み、使用を推奨されています。

IANA タイムゾーンデータベース (しばしば `tz` や `zoneinfo` と呼ばれる) タイムゾーンデータベースはコードとデータを保持しており、それらは地球全体にわたる多くの代表的な場所のローカル時刻の履歴を表しています。政治団体によるタイムゾーンの境界、UTC オフセット、夏時間のルールの変更を反映するため、定期的にデータベースが更新されます。

8.1.7 `strftime()` と `strptime()` の振る舞い

`date`, `datetime`, および `time` オブジェクトは全て、明示的な書式文字列でコントロールして時刻表現文字列を生成するための `strftime(format)` メソッドをサポートしています。大雑把にいうと、`d.strftime(fmt)` は `time` モジュールの `time.strftime(fmt, d.timetuple())` のように動作します。ただし全てのオブジェクトが `timetuple()` メソッドをサポートしているわけではありません。

逆に `datetime.strptime()` クラスメソッドは日付や時刻に対応する書式文字列から `datetime` オブジェクトを生成します。`datetime.strptime(date_string, format)` は `datetime(*(time.strptime(date_string, format)[0:6]))` と等価です。ただし、その書式に小数秒以下の要素またはタイムゾーン情報が含まれる、“`datetime.strptime`”ではサポートされるが“`time.strptime`”ではサポートされないような場合はその限りではありません。

`time` オブジェクトでは、年、月、日の値がないため、それらの書式化コードを使うことができません。無理矢理使った場合、年は 1900 に置き換えられ、月と日は 1 に置き換えられます。

`date` オブジェクトでは、時、分、秒、マイクロ秒の値がないため、それらの書式化コードを使うことができません。無理矢理使った場合、これらの値は 0 に置き換えられます。

Python はプラットフォームの C ライブラリの `strftime()` 関数を呼び出していて、プラットフォームごとにその実装が異なるのはよくあることなので、サポートされるフォーマット記号全体はプラットフォームごとに様々です。プラットフォームでサポートされているフォーマット記号全体を見るには、`strftime(3)` のドキュメントを参照してください。

同じ理由で、現在のロケールの文字集合で表現できない Unicode コードポイントを含む書式文字列の対処もプラットフォーム依存です。あるプラットフォームではそういったコードポイントはそのまま出力に出される一方、他のプラットフォームでは `strftime` が `UnicodeError` を送出したり、その代わりに空文字列を返したりするかもしれません。

以下のリストは C 標準 (1989 年版) が要求する全ての書式化コードで、標準 C 実装があれば全ての環境で動作します。1999 年版の C 標準では書式化コードが追加されているので注意してください。

`strftime()` が正しく動作する年の厳密な範囲はプラットフォーム間で異なります。プラットフォームに関わらず、1900 年以前の年は使うことができません。

ディレクティブ	意味	例	注釈
%a	ロケールの曜日名を短縮形で表示します。	Sun, Mon, ..., Sat (en_US); So, Mo, ..., Sa (de_DE)	(1)
%A	ロケールの曜日名を表示します。	Sunday, Monday, ..., Saturday (en_US); Sonntag, Montag, ..., Samstag (de_DE)	(1)
%w	曜日を 10 進表記した文字列を表示します。0 が日曜日で、6 が土曜日を表します。	0, 1, ..., 6	
%d	0 埋めした 10 進数で表記した月中の日にち。	01, 02, ..., 31	
%b	ロケールの月名を短縮形で表示します。	Jan, Feb, ..., Dec (en_US); Jan, Feb, ..., Dez (de_DE)	(1)
%B	ロケールの月名を表示します。	January, February, ..., December (en_US); Januar, Februar, ..., Dezember (de_DE)	(1)
%m	0 埋めした 10 進数で表記した月。	01, 02, ..., 12	
%y	0 埋めした 10 進数で表記した世紀無しの年。	00, 01, ..., 99	
%Y	西暦 (4 桁) の 10 進表記を表します。	1970, 1988, 2001, 2013	
%H	0 埋めした 10 進数で表記した時 (24 時間表記)。	00, 01, ..., 23	
%I	0 埋めした 10 進数で表記した時 (12 時間表記)。	01, 02, ..., 12	
%p	ロケールの AM もしくは PM と等価な文字列になります。	AM, PM (en_US); am, pm (de_DE)	(1), (2)
204			第 8 章 データ型

注釈:

- (1) フォーマットは現在のロケールに依存するので、出力値に何か仮定を置くときは注意すべきです。フィールドの順序は様々で (例えば、"月/日/年" と "日/月/年")、出力はロケールのデフォルトエンコーディングでエンコードされた Unicode 文字列を含むかもしれません (例えば、現在のロケールが `ja_JP` だった場合、デフォルトエンコーディングは `eucJP`、`SJIS`、`utf-8` のいずれかです; `locale.getlocale()` を使って現在のロケールのエンコーディングを確認します)。
- (2) `strptime()` メソッドと共に使われた場合、`%p` 指定子は出力の時間フィールドのみに影響し、`%I` 指定子が使われたかのように振る舞います。
- (3) `time` モジュールと違い、`datetime` モジュールはうるう秒をサポートしていません。
- (4) `%f` は C 標準規格の書式化コードセットへの拡張です (ただし `datetime` オブジェクトとは分けて実装しているので、常に使えます)。 `strptime()` メソッドと共に使われた場合、`%f` 指定子は 1 桁から 6 桁の数字を受け付け、右側から 0 埋めされます。

バージョン 2.6 で追加。

- (5) `naive` オブジェクトでは、書式化コード `%z` および `%Z` は空文字列に置き換えられます。

`aware` オブジェクトでは以下のようになります:

`%z` `utcoffset()` は `+HHMM` あるいは `-HHMM` の形式をもった 5 文字の文字列に変換されます。 `HH` は UTC オフセット時間を与える 2 桁の文字列で、`MM` は UTC オフセット分を与える 2 桁の文字列です。例えば、`utcoffset()` が `timedelta(hours=-3, minutes=-30)` を返した場合、`%z` は文字列 `'-0330'` に置き換わります。

`%Z` `tzname()` が `None` を返した場合、`%Z` は空文字列に置き換わります。そうでない場合、`%Z` は返された値に置き換わりますが、これは文字列でなければなりません。

- (6) `strptime()` メソッドと共に使われた場合、`%U` と `%W` 指定子は、年と曜日が指定された場合の計算でのみ使われます。

8.2 calendar — 一般的なカレンダーに関する関数群

ソースコード: [Lib/calendar.py](#)

このモジュールは Unix の `cal` プログラムのようなカレンダー出力を行い、それに加えてカレンダーに関する有益な関数群を提供します。標準ではこれらのカレンダーは (ヨーロッパの慣例に従って) 月曜日を週の始まりとし、日曜日を最後の日としています。 `setfirstweekday()` を用いることで、日曜日 (6) や他の曜日を週の始まりに設定することができます。日付を表す引数は整数値で与えます。関連する機能として、`datetime` と `time` モジュールも参照してください。

このモジュールで提供する関数とクラスのほとんどは `datetime` に依存しており、過去も未来も現代のグレゴリオ暦を利用します。この方式は Dershowitz と Reingold の書籍「Calendrical Calculations」にある proleptic Gregorian 暦に一致しており、同書では全ての計算の基礎となる暦としています。

class `calendar.Calendar` (`[firstweekday]`)

`Calendar` オブジェクトを作ります。 `firstweekday` は整数で週の始まりの曜日を指定するものです。0 が月曜 (デフォルト)、6 が日曜です。

`Calendar` オブジェクトは整形されるカレンダーのデータを準備するために使えるいくつかのメソッドを提供しています。しかし整形機能そのものは提供していません。それはサブクラスの仕事なのです。

バージョン 2.5 で追加。

`Calendar` インスタンスには以下のメソッドがあります:

`iterweekdays` ()

曜日の数字を一週間分生成するイテレータを返します。イテレータから得られる最初の数字は `firstweekday` が返す数字と同じになります。

`itermonthdates` (`year`, `month`)

`year` 年 `month` (1-12) 月に対するイテレータを返します。このイテレータはその月の全ての日、およびその月が始まる前の日とその月が終わった後の日のうち、週の欠けを埋めるために必要な日 (`datetime.date` オブジェクトとして) を返します。

`itermonthdays2` (`year`, `month`)

`year` 年 `month` 月に対する `itermonthdates` () と同じようなイテレータを返します。生成されるのは日付の数字と曜日を表す数字のタプルです。

`itermonthdays` (`year`, `month`)

`year` 年 `month` 月に対する `itermonthdates` () と同じようなイテレータを返します。生成されるのは日付の数字だけです。

`monthdatescalendar` (`year`, `month`)

`year` 年 `month` 月の週のリストを返します。週は全て七つの `datetime.date` オブジェクトからなるリストです。

`monthdays2calendar` (`year`, `month`)

`year` 年 `month` 月の週のリストを返します。週は全て七つの日付の数字と曜日を表す数字のタプルからなるリストです。

`monthdayscalendar` (`year`, `month`)

`year` 年 `month` 月の週のリストを返します。週は全て七つの日付の数字からなるリストです。

`yeardatescalendar` (`year` [, `width`])

指定された年のデータを整形に向く形で返します。返される値は月の並びのリストです。月の並びは最大で `width` ヶ月 (デフォルトは 3 ヶ月) 分です。各月は 4 ないし 6 週からなり、各週は 1 ないし 7 日からなります。各日は `datetime.date` オブジェクトです。

`yeardays2calendar` (`year` [, `width`])

指定された年のデータを整形に向く形で返します (`yeardatescalendar` () と同様です)。週のリストの中が日付の数字と曜日の数字のタプルになります。月の範囲外の部分の日付はゼロです。

`yeardayscalendar` (`year` [, `width`])

指定された年のデータを整形に向く形で返します (`yeardatescalendar()` と同様です)。週のリストの中が日付の数字になります。月の範囲外の日付はゼロです。

```
class calendar.TextCalendar ([firstweekday])
```

このクラスはプレインテキストのカレンダーを生成するのに使えます。

バージョン 2.5 で追加.

`TextCalendar` インスタンスには以下のメソッドがあります:

```
formatmonth (theyear, themonth[, w[, l]])
```

ひと月分のカレンダーを複数行の文字列で返します。 `w` により日の列幅を変えることができ、それらはセンタリングされます。 `l` により各週の表示される行数を変えることができます。 `setfirstweekday()` メソッドでセットされた週の最初の曜日に依存します。

```
prmonth (theyear, themonth[, w[, l]])
```

`formatmonth()` で返されるひと月分のカレンダーを出力します。

```
formatyear (theyear[, w[, l[, c[, m]]]])
```

`m` 列からなる一年間のカレンダーを複数行の文字列で返します。任意の引数 `w`, `l`, `c` はそれぞれ、日付列の表示幅、各週の行数及び月と月の間のスペースの数を変更するためのものです。 `setfirstweekday()` メソッドでセットされた週の最初の曜日に依存します。カレンダーを出力できる最初の年はプラットフォームに依存します。

```
pryear (theyear[, w[, l[, c[, m]]]])
```

`formatyear()` で返される一年間のカレンダーを出力します。

```
class calendar.HTMLCalendar ([firstweekday])
```

このクラスは HTML のカレンダーを生成するのに使えます。

バージョン 2.5 で追加.

`HTMLCalendar` インスタンスには以下のメソッドがあります:

```
formatmonth (theyear, themonth[, withyear])
```

ひと月分のカレンダーを HTML のテーブルとして返します。 `withyear` が真であればヘッダには年も含まれます。そうでなければ月の名前だけが使われます。

```
formatyear (theyear[, width])
```

一年分のカレンダーを HTML のテーブルとして返します。 `width` の値 (デフォルトでは 3 です) は何ヶ月分を一行に収めるかを指定します。

```
formatyearpage (theyear[, width[, css[, encoding]]])
```

一年分のカレンダーを一つの完全な HTML ページとして返します。 `width` の値 (デフォルトでは 3 です) は何ヶ月分を一行に収めるかを指定します。 `css` は使われるカスケーディングスタイルシートの名前です。スタイルシートを使わないようにするために `None` を渡すこともできます。 `encoding` には出力に使うエンコーディングを指定します (デフォルトではシステムデフォルトのエンコーディングです)。

```
class calendar.LocaleTextCalendar ([firstweekday[, locale]])
```

この `TextCalendar` のサブクラスではコンストラクタにロケール名を渡すことができ、メソッドの

返り値で月や曜日が指定されたロケールのものになります。このロケールがエンコーディングを含む場合には、月や曜日の入った文字列はユニコードとして返されます。

バージョン 2.5 で追加.

class `calendar.LocaleHTMLCalendar` (`[firstweekday[, locale]]`)

この `HTMLCalendar` のサブクラスではコンストラクタにロケール名を渡すことができ、メソッドの返り値で月や曜日が指定されたロケールのものになります。このロケールがエンコーディングを含む場合には、月や曜日の入った文字列はユニコードとして返されます。

バージョン 2.5 で追加.

注釈: これら 2 つのクラスの `formatweekday()` と `formatmonthname()` メソッドは、一時的に現在の `locale` を指定された `locale` に変更します。現在の `locale` はプロセス全体に影響するので、これらはスレッドセーフではありません。

単純なテキストのカレンダーに関して、このモジュールには以下のような関数が提供されています。

`calendar.setfirstweekday` (`weekday`)

週の最初の曜日 (0 は月曜日, 6 は日曜日) を設定します。定数 `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` 及び `SUNDAY` は便宜上提供されています。例えば、日曜日を週の開始日に設定するときは:

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

バージョン 2.0 で追加.

`calendar.firstweekday` ()

現在設定されている週の最初の曜日を返します。

バージョン 2.0 で追加.

`calendar.isleap` (`year`)

`year` が閏年なら `True` を、そうでなければ `False` を返します。

`calendar.leapdays` (`y1`, `y2`)

範囲 (`y1 ... y2`) 指定された期間の閏年の回数を返します。ここで `y1` や `y2` は年を表します。

バージョン 2.0 で変更: この関数は、Python 1.5.2 で、世紀の境目をまたいだ範囲で正しく動作していませんでした。

`calendar.weekday` (`year`, `month`, `day`)

`year` (1970–...), `month` (1–12), `day` (1–31) で与えられた日の曜日 (0 は月曜日) を返します。

`calendar.weekheader` (`n`)

短縮された曜日名を含むヘッダを返します。`n` は各曜日を何文字で表すかを指定します。

`calendar.monthrange(year, month)`

`year` と `month` で指定された月の一日の曜日と日数を返します。

`calendar.monthcalendar(year, month)`

月のカレンダーを行列で返します。各行が週を表し、月の範囲外の日は 0 になります。それぞれの週は `setfirstweekday()` で設定をしていない限り月曜日から始まります。

`calendar.prmonth(theyear, themonth[, w[, l]])`

`month()` 関数によって返される月のカレンダーを出力します。

`calendar.month(theyear, themonth[, w[, l]])`

`TextCalendar` の `formatmonth()` メソッドを利用して、ひと月分のカレンダーを複数行の文字列で返します。

バージョン 2.0 で追加.

`calendar.prcal(year[, w[, l[c]]])`

`calendar()` 関数で返される一年間のカレンダーを出力します。

`calendar.calendar(year[, w[, l[c]]])`

`TextCalendar` の `formatyear()` メソッドを利用して、3 列からなる一年間のカレンダーを複数行の文字列で返します。

バージョン 2.0 で追加.

`calendar.timegm(tuple)`

カレンダーと直接は関係無いが、`time` モジュールの `gmtime()` 関数が返す形式の時刻を表すタプルを引数に取り、1970 を基点とするエポック時刻で POSIX エンコーディングであると仮定して、対応する Unix タイムスタンプの値を返します。実際には、`time.gmtime()` と `timegm()` はお互いの逆関数です。

バージョン 2.0 で追加.

`calendar` モジュールの以下のデータ属性を利用することができます:

`calendar.day_name`

現在のロケールでの曜日を表す配列です。

`calendar.day_abbr`

現在のロケールでの短縮された曜日を表す配列です。

`calendar.month_name`

現在のロケールでの月の名を表す配列です。この配列は通常の約束事に従って、1 月を数字の 1 で表しますので、長さが 13 ある代わりに `month_name[0]` が空文字列になります。

`calendar.month_abbr`

現在のロケールでの短縮された月の名を表す配列です。この配列は通常の約束事に従って、1 月を数字の 1 で表しますので、長さが 13 ある代わりに `month_abbr[0]` が空文字列になります。

参考:

`datetime` モジュール `time` モジュールと似た機能を持った日付と時間用のオブジェクト指向インタフェース。

`time` モジュール 低レベルの時間に関連した関数群。

8.3 collections — 高性能なコンテナ・データ型

バージョン 2.4 で追加。

Source code: [Lib/collections.py](#) and [Lib/_abcoll.py](#)

このモジュールは、汎用の Python 組み込みコンテナ `dict`, `list`, `set`, および `tuple` に代わる、特殊なコンテナデータ型を実装しています。

<code>namedtuple()</code>	名前付きフィールドを持つタプルのサブクラスを作成するファクトリ関数	バージョン 2.6 で追加。
<code>deque</code>	両端における <code>append</code> や <code>pop</code> を高速に行えるリスト風のコンテナ	バージョン 2.4 で追加。
<code>Counter</code>	ハッシュ可能なオブジェクトを数え上げる辞書のサブクラス	バージョン 2.7 で追加。
<code>OrderedDict</code>	項目が追加された順序を記憶する辞書のサブクラス	バージョン 2.7 で追加。
<code>defaultdict</code>	ファクトリ関数を呼び出して存在しない値を供給する辞書のサブクラス	バージョン 2.5 で追加。

コンテナ型の作成に加えて、`collections` モジュールは幾つかの ABC (*abstract base classes* = 抽象基底クラス) を提供しています。ABC はクラスが特定のインタフェース持っているかどうか、たとえばハッシュ可能であるかやマッピングであるかを判定するのに利用します。

8.3.1 Counter オブジェクト

便利で迅速な検数をサポートするカウンタツールが提供されています。例えば:

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
```

(次のページに続く)

(前のページからの続き)

```
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554), ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

class `collections.Counter` (`[iterable-or-mapping]`)

`Counter` はハッシュ可能なオブジェクトをカウントする `dict` のサブクラスです。これは、要素を辞書のキーとして保存し、そのカウントを辞書の値として保存する、順序付けされていないコレクションです。カウントは、0 や負のカウントを含む整数値をとれます。`Counter` クラスは、他の言語のバグや多重集合のようなものです。

要素は、`iterable` から数え上げられたり、他の `mapping` (やカウンタ) から初期化されます:

```
>>> c = Counter()                    # a new, empty counter
>>> c = Counter('gallahad')         # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2}) # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)      # a new counter from keyword args
```

カウンタオブジェクトは辞書のインタフェースを持ちますが、存在しない要素に対して `KeyError` を送出する代わりに 0 を返すという違いがあります:

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                        # count of a missing element is_
↪ zero
0
```

カウントを 0 に設定しても、要素はカウンタから取り除かれません。完全に取り除くには、`del` を使ってください:

```
>>> c['sausage'] = 0                 # counter entry with a zero count
>>> del c['sausage']                 # del actually removes the entry
```

バージョン 2.7 で追加.

カウンタオブジェクトは、すべての辞書に利用できるものに加え、3つのメソッドをサポートしています:

elements()

それぞれの要素を、そのカウント分の回数だけ繰り返すイテレータを返します。要素は任意の順序で返されます。ある要素のカウントが 1 未満なら、`elements()` はそれを無視します。

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common(`n`)

最も多い `n` 要素を、カウントが多いものから少ないものまで順に並べたリストを返します。`n` が省略されるか `None` であれば、`most_common()` はカウンタのすべての要素を返します。等しいカウントの要素は任意に並べられます:

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('r', 2), ('b', 2)]
```

subtract ([*iterable-or-mapping*])

要素から *iterable* の要素または *mapping* の要素が引かれます。 `dict.update()` に似ていますが、カウントを置き換えるのではなく引きます。入力も出力も、0 や負になりえます。

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

普通の辞書のメソッドは、以下の 2 つのメソッドがカウンタに対して異なる振る舞いをするのを除き、`Counter` オブジェクトにも利用できます。

fromkeys (*iterable*)

このクラスメソッドは `Counter` オブジェクトには実装されていません。

update ([*iterable-or-mapping*])

要素が *iterable* からカウントされるか、別の *mapping* (やカウンタ) が追加されます。 `dict.update()` に似ていますが、カウントを置き換えるのではなく追加します。また、*iterable* には (key, value) 対のシーケンスではなく、要素のシーケンスが求められます。

`Counter` オブジェクトを使ったよくあるパターン:

```
sum(c.values())           # total of all counts
c.clear()                 # reset all counts
list(c)                   # list unique elements
set(c)                    # convert to a set
dict(c)                   # convert to a regular dictionary
c.items()                 # convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs)) # convert from a list of (elem, cnt) pairs
c.most_common()[::-1]     # n least common elements
c += Counter()            # remove zero and negative counts
```

`Counter` オブジェクトを組み合わせると多重集合 (1 以上のカウントをもつカウンタ) を作るために、いくつかの数学演算が提供されています。足し算と引き算は、対応する要素を足したり引いたりすることによってカウンタを組み合わせます。共通部分と合併集合は、対応するカウントの最大値と最小値を返します。それぞれの演算はカウントに符号がついた入力を受け付けますが、カウントが 0 以下である結果は出力から除かれます。

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                               # add two counters together:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                               # subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d                               # intersection:  min(c[x], d[x])
Counter({'a': 1, 'b': 1})
```

(次のページに続く)

(前のページからの続き)

```
>>> c | d                                     # union: max(c[x], d[x])
Counter({'a': 3, 'b': 2})
```

注釈: カウンタはもともと、推移するカウントを正の整数で表すために設計されました。しかし、他の型や負の値を必要とするユースケースを不必要に排除することがないように配慮されています。このようなユースケースの助けになるように、この節で最低限の範囲と型の制限について記述します。

- `Counter` クラス自体は辞書のサブクラスで、キーと値に制限はありません。値はカウントを表す数であることを意図していますが、値フィールドに任意のものを保存 できます。
- `most_common()` メソッドが要求するのは、値が順序付け可能なことだけです。
- `c[key] += 1` のようなインブレース演算では、値の型に必要なのは足し算と引き算ができることだけです。よって分数、浮動小数点数、小数も使え、負の値がサポートされています。これと同じことが、負や 0 の値を入力と出力に許す `update()` と `subtract()` メソッドにも言えます。
- 多重集合メソッドは正の値を扱うユースケースに対してのみ設計されています。入力に負や 0 に出来ませんが、正の値の出力のみが生成されます。型の制限はありませんが、値の型は足し算、引き算、比較をサポートしている必要があります。
- `elements()` メソッドは整数のカウントを要求します。これは 0 と負のカウントを無視します。

参考:

- Python 2.5 向けの [Counter class](#) と、それより前の Python 2.4 向けの [Bag recipe](#)。
- Smalltalk の [Bag class](#)。
- Wikipedia の [Multisets](#) の項目。
- [C++ multisets](#) の例を交えたチュートリアル。
- 数学的な多重集合の演算とそのユースケースは、*Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19* を参照してください。
- 与えられた要素の集まりから与えられた大きさの別個の多重集合をすべて数え上げるには、[itertools.combinations_with_replacement\(\)](#) を参照してください。

```
map(Counter, combinations_with_replacement('ABC', 2)) -> AA AB AC BB BC CC
```

8.3.2 deque オブジェクト

```
class collections.deque([iterable[, maxlen]])
```

`iterable` で与えられるデータから、新しい deque オブジェクトを (`append()` をつかって) 左から右に初期化して返します。 `iterable` が指定されない場合、新しい deque オブジェクトは空になります。

Deque とは、スタックとキューを一般化したものです (この名前は「デック」と発音され、これは「double-ended queue」の省略形です)。Deque はどちらの側からも `append` と `pop` が可能で、スレッドセーフでメモリ効率がよく、どちらの方向からもおよそ $O(1)$ のパフォーマンスで実行できます。

`list` オブジェクトでも同様の操作を実現できますが、これは高速な固定長の操作に特化されており、内部のデータ表現形式のサイズと位置を両方変えるような `pop(0)` や `insert(0, v)` などの操作ではメモリ移動のために $O(n)$ のコストを必要とします。

バージョン 2.4 で追加.

`maxlen` が指定され無かったり `None` だった場合、`deque` は任意のサイズまで大きくなります。そうでない場合、`deque` のサイズは指定された最大長に制限されます。長さが制限された `deque` がいっぱいになると、新しい要素を追加するときに追加した要素数分だけ追加したのと反対側から要素が捨てられます。長さが制限された `deque` は Unix における `tail` フィルタと似た機能を提供します。トランザクションの tracking や最近使った要素だけを残したいデータプール (pool of data) などにも便利です。

バージョン 2.6 で変更: `maxlen` パラメータを追加しました。

Deque オブジェクトは以下のようなメソッドをサポートしています:

`append(x)`

`x` を deque の右側につけ加えます。

`appendleft(x)`

`x` を deque の左側につけ加えます。

`clear()`

deque からすべての要素を削除し、長さを 0 にします。

`count(x)`

`x` に等しい deque の要素を数え上げます。

バージョン 2.7 で追加.

`extend(iterable)`

イテレータ化可能な引数 `iterable` から得られる要素を deque の右側に追加し拡張します。

`extendleft(iterable)`

イテレータ化可能な引数 `iterable` から得られる要素を deque の左側に追加し拡張します。注意: 左から追加した結果は、イテレータ引数の順序とは逆になります。

`pop()`

deque の右側から要素をひとつ削除し、その要素を返します。要素がひとつも存在しない場合は `IndexError` を発生させます。

`popleft()`

deque の左側から要素をひとつ削除し、その要素を返します。要素がひとつも存在しない場合は `IndexError` を発生させます。

remove (*value*)

value の最初に現れるものを削除します。要素が見つからない場合は `ValueError` を送出します。

バージョン 2.5 で追加。

reverse ()

deque の要素をインプレースに逆転し、`None` を返します。

バージョン 2.7 で追加。

rotate (*n=1*)

deque の要素を全体で *n* ステップだけ右にローテートします。*n* が負の値の場合は、左にローテートします。

deque が空でないときは、deque をひとつ右にローテートすることは `d.appendleft(d.pop())` と同じで、deque をひとつ左にローテートすることは `d.append(d.popleft())` と同じです。

deque オブジェクトは読み取り専用属性も 1 つ提供しています:

maxlen

deque の最大長で、制限されていなければ `None` です。

バージョン 2.7 で追加。

上記の操作のほかにも、deque は次のような操作をサポートしています: イテレータ化、`pickle`、`len(d)`、`reversed(d)`、`copy.copy(d)`、`copy.deepcopy(d)`、`in` 演算子による包含検査、そして `d[-1]` などの添え字による参照。両端についてインデックスアクセスは $O(1)$ ですが、中央部分については $O(n)$ の遅さです。高速なランダムアクセスが必要なならリストを使ってください。

例:

```
>>> from collections import deque
>>> d = deque('ghi')                # make a new deque with three items
>>> for elem in d:                  # iterate over the deque's elements
...     print elem.upper()
G
H
I

>>> d.append('j')                   # add a new entry to the right side
>>> d.appendleft('f')               # add a new entry to the left side
>>> d                               # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                         # return and remove the rightmost item
'j'
>>> d.popleft()                     # return and remove the leftmost item
'f'
>>> list(d)                         # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                           # peek at leftmost item
```

(次のページに続く)

(前のページからの続き)

```
'g'
>>> d[-1]                                # peek at rightmost item
'i'

>>> list(reversed(d))                    # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                             # search the deque
True
>>> d.extend('jkl')                      # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                          # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                         # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))                  # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                            # empty the deque
>>> d.pop()                              # cannot pop from an empty deque
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')                  # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

deque のレシピ

この節では deque を使ったさまざまなアプローチを紹介します。

長さが制限された deque は Unix における tail フィルタに相当する機能を提供します:

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    return deque(open(filename), n)
```

別のアプローチとして deque を右に append して左に pop して使うことで追加した要素を維持するのに使えます:

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # http://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
```

(次のページに続く)

(前のページからの続き)

```
s = sum(d)
for elem in it:
    s += elem - d.popleft()
    d.append(elem)
yield s / float(n)
```

`rotate()` メソッドのおかげで、`deque` の一部を切り出したり削除したりできるようになります。たとえば `del d[n]` の純粋な Python 実装では `pop` したい要素まで `rotate()` します

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

`deque` の切り出しを実装するのにも、同様のアプローチを使います。まず対象となる要素を `rotate()` によって `deque` の左端までもってきってから、`popleft()` をつかって古い要素を消します。そして、`extend()` で新しい要素を追加したのち、逆のローテートでもとに戻せばよいのです。このアプローチをやや変えたものとして、Forth スタイルのスタック操作、つまり `dup`, `drop`, `swap`, `over`, `pick`, `rot`, および `roll` を実装するのも簡単です。

8.3.3 defaultdict オブジェクト

`class collections.defaultdict([default_factory[, ...]])`

新しいディクショナリ状のオブジェクトを返します。`defaultdict` は組み込みの `dict` のサブクラスです。メソッドをオーバーライドし、書き込み可能なインスタンス変数を 1 つ追加している以外は `dict` クラスと同じです。同じ部分については以下では省略されています。

1 つめの引数は `default_factory` 属性の初期値です。デフォルトは `None` です。残りの引数はキーワード引数もふくめ、`dict` のコンストラクタにあたえられた場合と同様に扱われます。

バージョン 2.5 で追加。

`defaultdict` オブジェクトは標準の `dict` に加えて、以下のメソッドを実装しています:

`__missing__(key)`

もし `default_factory` 属性が `None` であれば、このメソッドは `KeyError` 例外を、`key` を引数として発生させます。

もし `default_factory` 属性が `None` でなければ、このメソッドは `default_factory` を引数なしで呼び出し、あたえられた `key` に対応するデフォルト値を作ります。そしてこの値を `key` に対応する値を辞書に登録して返ります。

もし `default_factory` の呼出が例外を発生させた場合には、変更せずそのまま例外を投げます。

このメソッドは `dict` クラスの `__getitem__()` メソッドで、キーが存在しなかった場合によびだされます。値を返すか例外を発生させるのどちらにしても、`__getitem__()` からそのまま値が

返るか例外が発生します。

なお、`__missing__()` は `__getitem__()` 以外のいかなる演算に対しても呼び出されません。よって `get()` は、普通の辞書と同様に、`default_factory` を使うのではなくデフォルトとして `None` を返します。

`defaultdict` オブジェクトは以下のインスタンス変数をサポートしています：

default_factory

この属性は `__missing__()` メソッドによって使われます。これは存在すればコンストラクタの第 1 引数によって初期化され、そうでなければ `None` になります。

defaultdict の使用例

`list` を `default_factory` とすることで、キー=値ペアのシーケンスをリストの辞書へ簡単にグループ化できます。：

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> d.items()
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

それぞれのキーが最初に登場したとき、マッピングにはまだ存在しません。そのためエントリは `default_factory` 関数が返す空の `list` を使って自動的に作成されます。 `list.append()` 操作は新しいリストに紐付けられます。キーが再度出現下場合には、通常の参照動作が行われます（そのキーに対応するリストが返ります）。そして `list.append()` 操作で別の値をリストに追加します。このテクニックは `dict.setdefault()` を使った等価なものよりシンプルで速いです：

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> d.items()
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

`default_factory` を `int` にすると、`defaultdict` を（他の言語の `bag` や `multiset` のように）要素の数え上げに便利に使うことができます：

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> d.items()
[('i', 4), ('p', 2), ('s', 4), ('m', 1)]
```

最初に文字が出現したときは、マッピングが存在しないので `default_factory` 関数が `int()` を呼んでデフォルトのカウント 0 を生成します。インクリメント操作が各文字を数え上げます。

常に 0 を返す `int()` は特殊な関数でした。定数を生成するより速くて柔軟な方法は、0 に限らず何でも定数を生成する `itertools.repeat()` を使うことです。

```
>>> def constant_factory(value):
...     return itertools.repeat(value).next
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

`default_factory` を `set` に設定することで、`defaultdict` をセットの辞書を作るために利用することができます:

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
...
>>> d.items()
[('blue', set([2, 4])), ('red', set([1, 3]))]
```

8.3.4 `namedtuple()` 名前付きフィールドを持つタブルのファクトリ関数

名前付きタブルはタブルの中の場所に意味を割り当てて、より読みやすく自己解説的なコードを書けるようにします。通常のタブルが利用されていた場所で利用でき、場所に対するインデックスの代わりに名前を使ってフィールドにアクセスできます。

`collections.namedtuple(typename, field_names[, verbose=False][, rename=False])`

`typename` という名前の tuple の新しいサブクラスを返します。新しいサブクラスは、tuple に似ているけれどもインデックスやイテレータだけでなく属性名によるアクセスもできるオブジェクトを作るのに使います。このサブクラスのインスタンスは、わかりやすい docstring (型名と属性名が入っています) や、tuple の内容を `name=value` という形のリストで返す使いやすい `__repr__()` も持っています。

`field_names` は `['x', 'y']` のような文字列のシーケンスです。`field_names` には、代わりに各属性名を空白文字 (whitespace) および/またはカンマ (,) で区切った文字列を渡すこともできます。例えば、`'x y'` や `'x, y'` です。

アンダースコア (.) で始まる名前を除いて、Python の正しい識別子 (identifier) ならなんでも属性名として使うことができます。正しい識別子とはアルファベット (letters)、数字 (digits)、アンダースコア (.) を含みますが、数字やアンダースコアで始まる名前や、`class`, `for`, `return`, `global`, `pass`, `print`, `raise` などといった *keyword* は使えません。

`rename` が真なら、不適切なフィールド名は自動的に位置引数に置き換えられます。例えば `['abc', 'def', 'ghi', 'abc']` は、予約語 `def` と重複しているフィールド名 `abc` が除去され、`['abc', '_1', 'ghi', '_3']` に変換されます。

`verbose` が真なら、クラスを作る直前にクラス定義が表示されます。

名前付きタプルのインスタンスはインスタンスごとの辞書を持たないので、軽量で、普通のタプル以上のメモリを使用しません。

バージョン 2.6 で追加。

バージョン 2.7 で変更: `rename` のサポートを追加しました。

例:

```
>>> Point = namedtuple('Point', ['x', 'y'], verbose=True)
class Point(tuple):
    'Point(x, y)'

    __slots__ = ()

    _fields = ('x', 'y')

    def __new__(_cls, x, y):
        'Create new instance of Point(x, y)'
        return _tuple.__new__(_cls, (x, y))

    @classmethod
    def _make(cls, iterable, new=tuple.__new__, len=len):
        'Make a new Point object from a sequence or iterable'
        result = new(cls, iterable)
        if len(result) != 2:
            raise TypeError('Expected 2 arguments, got %d' % len(result))
        return result

    def __repr__(self):
        'Return a nicely formatted representation string'
        return 'Point(x=%r, y=%r)' % self

    def _asdict(self):
        'Return a new OrderedDict which maps field names to their values'
        return OrderedDict(zip(self._fields, self))

    def _replace(_self, **kwds):
        'Return a new Point object replacing specified fields with new values'
        result = _self._make(map(kwds.pop, ('x', 'y'), _self))
        if kwds:
            raise ValueError('Got unexpected field names: %r' % kwds.keys())
        return result

    def __getnewargs__(self):
        'Return self as a plain tuple.  Used by copy and pickle.'
        return tuple(self)

    __dict__ = _property(_asdict)

    def __getstate__(self):
```

(次のページに続く)

(前のページからの続き)

```

    'Exclude the OrderedDict from pickling'
    pass

    x = _property(_itemgetter(0), doc='Alias for field number 0')

    y = _property(_itemgetter(1), doc='Alias for field number 1')

>>> p = Point(11, y=22)      # instantiate with positional or keyword arguments
>>> p[0] + p[1]              # indexable like the plain tuple (11, 22)
33
>>> x, y = p                 # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y                # fields also accessible by name
33
>>> p                        # readable __repr__ with a name=value style
Point(x=11, y=22)

```

名前付きタプルは `csv` や `sqlite3` モジュールが返すタプルのフィールドに名前を付けるときにとても便利です:

```

EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, ↵
↵paygrade')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
    print emp.name, emp.title

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print emp.name, emp.title

```

タプルから継承したメソッドに加えて、名前付きタプルは 3 つの追加メソッドと一つの属性をサポートしています。フィールド名との衝突を避けるためにメソッド名と属性名はアンダースコアで始まります。

classmethod `somenamedtuple._make(iterable)`

既存の `sequence` や `Iterable` から新しいインスタンスを作るクラスメソッド。

```

>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)

```

`somenamedtuple._asdict()`

フィールド名を対応する値にマッピングする新しい順序付き辞書 (`OrderedDict`) を返します:

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
OrderedDict([('x', 11), ('y', 22)])
```

バージョン 2.7 で変更: 通常の *dict* の代わりに *OrderedDict* を返すようになりました。

`somenamedtuple._replace(**kwargs)`

指定されたフィールドを新しい値で置き換えた、新しい名前付きタプルを作って返します:

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)

>>> for partnum, record in inventory.items():
...     inventory[partnum] = record._replace(price=newprices[partnum],
... timestamp=time.now())
```

`somenamedtuple._fields`

フィールド名をリストにしたタプル. 内省 (introspection) したり、既存の名前付きタプルをもとに新しい名前付きタプルを作成する時に便利です。

```
>>> p._fields           # view the field names
('x', 'y')

>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

文字列に格納された名前を使って名前付きタプルから値を取得するには `getattr()` 関数を使います:

```
>>> getattr(p, 'x')
11
```

辞書を名前付きタプルに変換するには、`**` 演算子 (double-star-operator, `tut-unpacking-arguments` で説明しています) を使います。:

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

名前付きタプルは通常の Python クラスなので、継承して機能を追加したり変更するのは容易です。次の例では計算済みフィールドと固定幅の print format を追加しています:

```
>>> class Point(namedtuple('Point', 'x y')):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
```

(次のページに続く)

(前のページからの続き)

```

...     def __str__(self):
...         return 'Point: x=%6.3f y=%6.3f hypot=%6.3f' % (self.x, self.y, self.
↪hypot)
...
>>> for p in Point(3, 4), Point(14, 5/7.):
...     print p
Point: x= 3.000 y= 4.000 hypot= 5.000
Point: x=14.000 y= 0.714 hypot=14.018

```

このサブクラスは `__slots__` に空のタプルをセットしています。これにより、インスタンス辞書の作成を抑制してメモリ使用量を低く保つのに役立ちます。

サブクラス化は新しいフィールドを追加するのには適していません。代わりに、新しい名前付きタプルを `__fields__` 属性を元に作成してください:

```
>>> Point3D = namedtuple('Point3D', Point.__fields + ('z',))
```

`._replace()` でプロトタイプのインスタンスをカスタマイズする方法で、デフォルト値を実現できます:

```

>>> Account = namedtuple('Account', 'owner balance transaction_count')
>>> default_account = Account('<owner name>', 0.0, 0)
>>> johns_account = default_account._replace(owner='John')

```

列挙型定数は名前付きタプルでも実装できますが、クラス定義を利用した方がシンプルで効率的です:

```

>>> Status = namedtuple('Status', 'open pending closed')._make(range(3))
>>> Status.open, Status.pending, Status.closed
(0, 1, 2)
>>> class Status:
...     open, pending, closed = range(3)

```

参考:

Python 2.4 向けの [Named tuple recipe](#)。

8.3.5 OrderedDict オブジェクト

順序付き辞書 (ordered dictionary) は、ちょうど普通の辞書と同じようなものですが、項目が挿入された順序を記憶します。順序付き辞書に渡ってイテレートするとき、項目はそのキーが最初に追加された順序で返されます。

```
class collections.OrderedDict([items])
```

通常の *dict* メソッドをサポートする、辞書のサブクラスのインスタンスを返します。 *OrderedDict* は、キーが最初に追加された順序を記憶します。新しい項目が既存の項目を上書きしても、元の挿入位置は変わらないままです。項目を削除して再挿入するとそれが最後に移動します。

バージョン 2.7 で追加。

`OrderedDict.popitem(last=True)`

順序付き辞書の `popitem()` メソッドは、(key, value) 対を返して消去します。この対は `last` が真なら後入先出で、偽なら先入先出で返されます。

通常のマッピングのメソッドに加え、順序付き辞書は `reversed()` による逆順の反復もサポートしています。

`OrderedDict` 間の等価判定は順序が影響し、`list(od1.items())==list(od2.items())` のように実装されます。`OrderedDict` オブジェクトと他のマッピング (*Mapping*) オブジェクトの等価判定は、順序に影響されず、通常の辞書と同様です。これによって、`OrderedDict` オブジェクトは通常の辞書が使われるところならどこでも使用できます。

`OrderedDict` コンストラクタと `update()` メソッドは、どちらもキーワード引数を受け付けますが、その順序は失われます。これは、Python の関数呼び出しの意味づけにおいて、キーワード引数は順序付けされていない辞書を用いて渡されるからです。

参考:

Python 2.4 以降で動作する、[Equivalent OrderedDict recipe](#)。

OrderedDict の例とレシピ

順序付き辞書は挿入順序を記憶するので、ソートと組み合わせることで、ソートされた辞書を作れます:

```
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

>>> # dictionary sorted by value
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])

>>> # dictionary sorted by length of the key string
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2), ('banana', 3)])
```

この新しい順序付き辞書は、項目が削除されてもソートされた順序を保持します。しかし、キーが追加される時、そのキーは最後に追加され、ソートは保持されません。

キーが最後に挿入された順序を記憶するような、順序付き辞書の変種を作るのも簡単です。新しい項目が既存の項目を上書きしたら、元の挿入位置は最後に移動します:

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        if key in self:
            del self[key]
        OrderedDict.__setitem__(self, key, value)
```

順序付き辞書と `Counter` クラスを組み合わせると、要素が最初に現れた順を記憶するカウンタができます:

```
class OrderedCounter(Counter, OrderedDict):
    'Counter that remembers the order elements are first encountered'

    def __repr__(self):
        return '%s(%r)' % (self.__class__.__name__, OrderedDict(self))

    def __reduce__(self):
        return self.__class__, (OrderedDict(self),)
```

8.3.6 コレクション抽象基底クラス

`collections` モジュールは以下の *ABC* (抽象基底クラス) を提供します:

ABC	継承しているクラス	抽象メソッド	mixin メソッド
<code>Container</code>		<code>__contains__</code>	
<code>Hashable</code>		<code>__hash__</code>	
<code>Iterable</code>		<code>__iter__</code>	
<code>Iterator</code>	<code>Iterable</code>	<code>next</code>	<code>__iter__</code>
<code>Sized</code>		<code>__len__</code>	
<code>Callable</code>		<code>__call__</code>	
<code>Sequence</code>	<code>Sized</code> , <code>Iterable</code> , <code>Container</code>	<code>__getitem__</code> , <code>__len__</code>	<code>__contains__</code> , <code>__iter__</code> , <code>__reversed__</code> , <code>index</code> , <code>count</code>
<code>MutableSequence</code>	<code>Sequence</code>	<code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__len__</code> , <code>insert</code>	<code>Sequence</code> から継承したメソッドと、 <code>append</code> , <code>reverse</code> , <code>extend</code> , <code>pop</code> , <code>remove</code> , <code>__iadd__</code>
<code>Set</code>	<code>Sized</code> , <code>Iterable</code> , <code>Container</code>	<code>__contains__</code> , <code>__iter__</code> , <code>__len__</code>	<code>__le__</code> , <code>__lt__</code> , <code>__eq__</code> , <code>__ne__</code> , <code>__gt__</code> , <code>__ge__</code> , <code>__and__</code> , <code>__or__</code> , <code>__sub__</code> , <code>__xor__</code> , <code>isdisjoint</code>
<code>MutableSet</code>	<code>Set</code>	<code>__contains__</code> , <code>__iter__</code> , <code>__len__</code> , <code>add</code> , <code>discard</code>	<code>Set</code> から継承したメソッドと、 <code>clear</code> , <code>pop</code> , <code>remove</code> , <code>__ior__</code> , <code>__iand__</code> , <code>__ixor__</code> , <code>__isub__</code>
<code>Mapping</code>	<code>Sized</code> , <code>Iterable</code> , <code>Container</code>	<code>__getitem__</code> , <code>__iter__</code> , <code>__len__</code>	<code>__contains__</code> , <code>keys</code> , <code>items</code> , <code>values</code> , <code>get</code> , <code>__eq__</code> , <code>__ne__</code>
<code>MutableMapping</code>	<code>Mapping</code>	<code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__iter__</code> , <code>__len__</code>	<code>Mapping</code> から継承したメソッドと、 <code>pop</code> , <code>popitem</code> , <code>clear</code> , <code>update</code> , <code>setdefault</code>
<code>MappingView</code>	<code>Sized</code>		<code>__len__</code>
<code>ItemsView</code>	<code>MappingView</code> , <code>Set</code>		<code>__contains__</code> , <code>__iter__</code>
<code>KeysView</code>	<code>MappingView</code> , <code>Set</code>		<code>__contains__</code> , <code>__iter__</code>
<code>ValuesView</code>	<code>MappingView</code>		<code>__contains__</code> , <code>__iter__</code>

```
class collections.Container
```

```
class collections.Hashable
```

```
class collections.Sized
```

```
class collections.Callable
```

それぞれメソッド `__contains__()`, `__hash__()`, `__len__()`, `__call__()` を提供するクラスの ABC です。

```
class collections.Iterable
```

`__iter__()` メソッドを提供するクラスの ABC です。 [iterable](#) の定義も参照してください。

class collections.Iterator

`__iter__()` メソッドと `next()` メソッドを提供するクラスの ABC です。 [iterator](#) の定義も参照してください。

class collections.Sequence

class collections.MutableSequence

読み出し専用でMutableなシーケンスの ABC です。

class collections.Set

class collections.MutableSet

読み出し専用でMutableな集合の ABC です。

class collections.Mapping

class collections.MutableMapping

読み出し専用でMutableなマッピングの ABC です。

class collections.MappingView

class collections.ItemsView

class collections.KeysView

class collections.ValuesView

マッピング、要素、キー、値のビューの ABC です。

これらの ABC はクラスやインスタンスが特定の機能を提供しているかどうかを調べるのに使えます。例えば:

```
size = None
if isinstance(myvar, collections.Sized):
    size = len(myvar)
```

幾つかの ABC はコンテナ型 API を提供するクラスを開発するのを助ける mixin 型としても使えます。例えば、[Set](#) API を提供するクラスを作る場合、3 つの基本になる抽象メソッド `__contains__()`、`__iter__()`、`__len__()` だけが必要です。ABC が残りの `__and__()` や `isdisjoint()` といったメソッドを提供します:

```
class ListBasedSet(collections.Set):
    ''' Alternate set implementation favoring space over speed
        and not requiring the set elements to be hashable. '''
    def __init__(self, iterable):
        self.elements = lst = []
        for value in iterable:
            if value not in lst:
                lst.append(value)

    def __iter__(self):
        return iter(self.elements)

    def __contains__(self, value):
        return value in self.elements

    def __len__(self):
```

(次のページに続く)

(前のページからの続き)

```

        return len(self.elements)

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2           # The __and__() method is supported automatically

```

`Set` と `MutableSet` を mixin 型として利用するときの注意点:

- (1) 幾つかの set の操作は新しい set を作るので、デフォルトの mixin メソッドは iterable から新しいインスタンスを作成する方法を必要とします。クラスのコンストラクタは `ClassName(iterable)` の形のシグネチャを持つと仮定されます。内部の `_from_iterable()` というクラスメソッドが `cls(iterable)` を呼び出して新しい set を作る部分でこの仮定が使われています。コンストラクタのシグネチャが異なるクラスで `Set` を使う場合は、iterable 引数から新しいインスタンスを生成するように `_from_iterable()` をオーバーライドする必要があります。
- (2) (たぶん意味はそのままに速度を向上する目的で) 比較をオーバーライドする場合、`__le__()` と `__ge__()` だけを再定義すれば、その他の演算は自動的に追従します。
- (3) `Set` mixin 型は set のハッシュ値を計算する `_hash()` メソッドを提供しますが、すべての set が hashable や immutable とは限らないので、`__hash__()` は提供しません。mixin を使ってハッシュ可能な set を作る場合は、`Set` と `Hashable` の両方を継承して、`__hash__ = Set._hash` と定義してください。

参考:

- `MutableSet` を使った例として [OrderedSet recipe](#)。
- ABCs についての詳細は、[abc](#) モジュールと [PEP 3119](#) を参照してください。

8.4 heapq — ヒープキューアルゴリズム

バージョン 2.3 で追加.

ソースコード: [Lib/heapq.py](#)

このモジュールではヒープキューアルゴリズムの一実装を提供しています。優先度キューアルゴリズムとしても知られています。

ヒープとは、全ての親ノードの値が、その全ての子の値以下であるようなバイナリツリーです。この実装は、全ての k に対して、ゼロから要素を数えていった際に、 $\text{heap}[k] \leq \text{heap}[2*k+1]$ かつ $\text{heap}[k] \leq \text{heap}[2*k+2]$ となる配列を使っています。比較のために、存在しない要素は無限大として扱われます。ヒープの興味深い性質は、最小の要素が常にルート、つまり $\text{heap}[0]$ になることです。

以下の API は教科書におけるヒープアルゴリズムとは 2 つの側面で異なります: (a) ゼロベースのインデクス化を行っています。これにより、ノードに対するインデクスとその子ノードのインデクスの関係がやや明瞭でなくなりますが、Python はゼロベースのインデクス化を使っているのでよりしっくりきます。 (b) われわ

れの `pop` メソッドは最大の要素ではなく最小の要素 (教科書では "min heap: 最小ヒープ" と呼ばれています; 教科書では並べ替えをインプレースで行うのに適した "max heap: 最大ヒープ" が一般的です)。

これらの 2 点によって、ユーザに戸惑いを与えることなく、ヒープを通常の Python リストとして見ることができます: `heap[0]` が最小の要素となり、`heap.sort()` はヒープ不変式を保ちます!

ヒープを作成するには、`[]` に初期化されたリストを使うか、`heapify()` を用いて要素の入ったリストを変換します。

次の関数が用意されています:

`heapq.heappush(heap, item)`

`item` を `heap` に push します。ヒープ不変式を保ちます。

`heapq.heappop(heap)`

`pop` を行い、`heap` から最小の要素を返します。ヒープ不変式は保たれます。ヒープが空の場合、`IndexError` が送出されます。`pop` せずに最小の要素にアクセスするには、`heap[0]` を使ってください。

`heapq.heappushpop(heap, item)`

`item` を `heap` に push した後、`pop` を行って `heap` から最初の要素を返します。この一続きの動作を `heappush()` に引き続いて `heappop()` を別々に呼び出すよりも効率的に実行します。

バージョン 2.6 で追加。

`heapq.heapify(x)`

リスト `x` をインプレース処理し、線形時間でヒープに変換します。

`heapq.heapreplace(heap, item)`

`heap` から最小の要素を `pop` して返し、新たに `item` を push します。ヒープのサイズは変更されません。ヒープが空の場合、`IndexError` が送出されます。

この一息の演算は `heappop()` に次いで `heappush()` を送出するよりも効率的で、固定サイズのヒープを用いている場合にはより適しています。`pop/push` の組み合わせは必ずヒープから要素を一つ返し、それを `item` と置き換えます。

返される値は加えられた `item` よりも大きくなるかもしれません。それを望まないなら、代わりに `heappushpop()` を使うことを考えてください。この `push/pop` の組み合わせは二つの値の小さい方を返し、大きい方の値をヒープに残します。

このモジュールではさらに 3 つのヒープに基く汎用関数を提供します。

`heapq.merge(*iterables)`

複数のソートされた入力をマージ (merge) して一つのソートされた出力にします (たとえば、複数のログファイルの時刻の入ったエントリーをマージします)。ソートされた値にわたる `iterator` を返します。

`sorted(itertools.chain(*iterables))` と似ていますが、イテレータを返し、一度にはデータをメモリに読み込みまず、それぞれの入力 (最小から最大へ) ソートされていることを仮定します。

バージョン 2.6 で追加。

`heapq.nlargest(n, iterable[, key])`

`iterable` で定義されるデータセットのうち、最大値から降順に n 個の値のリストを返します。(あたえられた場合) `key` は、引数を一つとる、`iterable` のそれぞれの要素から比較キーを生成する関数を指定します: `key=str.lower` 以下のコードと同等です: `sorted(iterable, key=key, reverse=True)[:n]`

バージョン 2.4 で追加.

バージョン 2.5 で変更: オプションの `key` 引数が追加されました.

`heapq.nsmallest(n, iterable[, key])`

`iterable` で定義されるデータセットのうち、最小値から昇順に n 個の値のリストを返します。(あたえられた場合) `key` は、引数を一つとる、`iterable` のそれぞれの要素から比較キーを生成する関数を指定します: `key=str.lower` 以下のコードと同等です: `sorted(iterable, key=key)[:n]`

バージョン 2.4 で追加.

バージョン 2.5 で変更: オプションの `key` 引数が追加されました.

後ろ二つの関数は n の値が小さな場合に最適動作をします。大きな値の時には `sorted()` 関数の方が効率的です。さらに、 $n==1$ の時には `min()` および `max()` 関数の方が効率的です。この関数を繰り返し使うことが必要なら、`iterable` を実際のヒープに変えることを考えてください。

8.4.1 基本的な例

すべての値をヒープに push してから最小値を 1 つずつ pop することで、ヒープソートを実装できます:

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

これは `sorted(iterable)` に似ていますが、`sorted()` とは異なり、この実装はステブルソートではありません。

ヒープの要素はタプルに出来ます。これは、追跡される主レコードとは別に (タスクの優先度のような) 比較値を指定するときに便利です:

```
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

8.4.2 優先度キュー実装の注釈

優先度つきキュー は、ヒープの一般的な使い方で、実装にはいくつか困難な点があります:

- ソート安定性: 優先度が等しい二つのタスクが、もともと追加された順序で返されるためにはどうしたらいいでしょうか?
- Python 3 では、(priority, task) ペアに対するタプルの比較は、priority が同じで task がデフォルトの比較順を持たないときに破綻します。
- あるタスクの優先度が変化したら、どうやってそれをヒープの新しい位置に移動させるのでしょうか?
- 未解決のタスクが削除される必要があるとき、どのようにそれをキューから探して削除するのでしょうか?

最初の二つの困難の解決策は、項目を優先度、項目番号、そしてタスクを含む 3 要素のリストとして保存することです。この項目番号は、同じ優先度の二つのタスクが、追加された順序で返されるようにするための同点決勝戦として働きます。そして二つの項目番号が等しくなることはありませんので、タプルの比較が二つのタスクを直接比べようとすることはありません。

残りの困難は主に、未解決のタスクを探して、その優先度を変更したり、完全に削除することです。タスクを探すことは、キュー内の項目を指し示す辞書によってなされます。

項目を削除したり、優先度を変更することは、ヒープ構造の不変関係を壊すことになるので、もっと難しいです。ですから、可能な解決策は、その項目が無効であるものとしてマークし、必要なら変更された優先度の項目を加えることです:

```

pq = []                                # list of entries arranged in a heap
entry_finder = {}                      # mapping of tasks to entries
REMOVED = '<removed-task>'             # placeholder for a removed task
counter = itertools.count()            # unique sequence count

def add_task(task, priority=0):
    'Add a new task or update the priority of an existing task'
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
    'Mark an existing task as REMOVED. Raise KeyError if not found.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    'Remove and return the lowest priority task. Raise KeyError if empty.'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:

```

(次のページに続く)

(前のページからの続き)

```

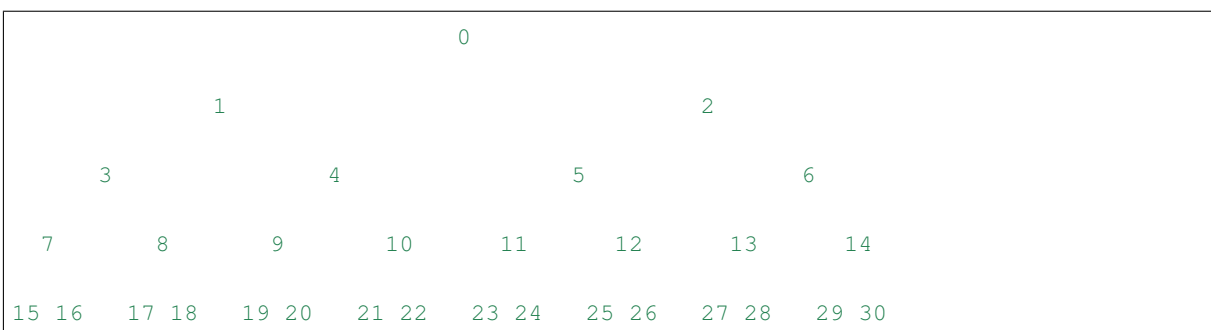
del entry_finder[task]
return task
raise KeyError('pop from an empty priority queue')

```

8.4.3 理論

ヒープとは、全ての k について、要素を 0 から数えたときに、 $a[k] \leq a[2*k+1]$ かつ $a[k] \leq a[2*k+2]$ となる配列です。比較のために、存在しない要素を無限大と考えます。ヒープの興味深い属性は $a[0]$ が常に最小の要素になることです。

上記の奇妙な不変式は、勝ち抜き戦判定の際に効率的なメモリ表現を行うためのものです。以下の番号は $a[k]$ ではなく k とします：



上の木構造では、各セル k は $2*k+1$ および $2*k+2$ を最大値としています。スポーツに見られるような通常の 2 つ組勝ち抜き戦では、各セルはその下にある二つのセルに対する勝者となっていて、個々のセルの勝者を追跡していくことにより、そのセルに対する全ての相手を見ることができます。しかしながら、このような勝ち抜き戦を使う計算機アプリケーションの多くでは、勝歴を追跡する必要はありません。メモリ効率をより高めるために、勝者が上位に進級した際、下のレベルから持ってきて置き換えることにすると、あるセルとその下位にある二つのセルは異なる三つの要素を含み、かつ上位のセルは二つの下位のセルに対して”勝者”となります。

このヒープ不変式が常に守られれば、インデクス 0 は明らかに最勝者となります。最勝者の要素を除去し、”次の”勝者を見つけるための最も単純なアルゴリズム的手法は、ある敗者要素（ここでは上図のセル 30 とします）を 0 の場所に持っていく、この新しい 0 を濾過するようにしてツリーを下らせて値を交換してゆきます。不変関係が再構築されるまでこれを続けます。この操作は明らかに、ツリー内の全ての要素数に対して対数的な計算量となります。全ての要素について繰り返すと、 $O(n \log n)$ のソート（並べ替え）になります。

このソートの良い点は、新たに挿入する要素が、その最に取り出す 0 番目の要素よりも”良い値”でない限り、ソートを行っている最中に新たな要素を効率的に追加できるということです。この性質は、シミュレーション的な状況で、ツリーで全ての入力イベントを保持し、”勝者となる状況”を最小のスケジュール時刻にするような場合に特に便利です。あるイベントが他のイベント群の実行をスケジュールする際、それらは未来にスケジュールされることになるので、それらのイベント群を容易にヒープに積むことができます。すなわち、ヒープはスケジューラを実装する上で良いデータ構造であるといえます（私は MIDI シーケンサで使っているものです :-）。

これまでスケジューラを実装するための様々なデータ構造が広範に研究されています。ヒープは十分高速で、

速度もおおむね一定であり、最悪の場合でも平均的な速度とさほど変わらないため良いデータ構造といえます。しかし、最悪の場合がひどい速度になることを除き、たいいていより効率の高い他のデータ構造表現も存在します。

ヒープはまた、巨大なディスクのソートでも非常に有用です。おそらくご存知のように、巨大なソートを行うと、複数の”ラン (run)” (予めソートされた配列で、そのサイズは通常 CPU メモリの量に関係しています) が生成され、続いて統合処理 (merging) がこれらのランを判定します。この統合処理はしばしば非常に巧妙に組織されています^{*1}。重要なのは、最初のソートが可能な限り長いランを生成することです。勝ち抜き戦はこれを達成するための良い方法です。もし利用可能な全てのメモリを使って勝ち抜き戦を行い、要素を置換および濾過処理して現在のランに収めれば、ランダムな入力に対してメモリの二倍のサイズのランを生成することになり、大体順序づけがなされている入力に対してはもっと高い効率になります。

さらに、ディスク上の 0 番目の要素を出力して、現在の勝ち抜き戦に (最後に出力した値に ”勝って” しまうために) 収められない入力を得たなら、ヒープには収まらないため、ヒープのサイズは減少します。解放されたメモリは二つ目のヒープを段階的に構築するために巧妙に再利用することができ、この二つ目のヒープは最初のヒープが崩壊していくのと同じ速度で成長します。最初のヒープが完全に消滅したら、ヒープを切り替えて新たなランを開始します。なんと巧妙で効率的なのでしょう！

一言で言うと、ヒープは知って得するメモリ構造です。私はいくつかのアプリケーションでヒープを使っていて、’ヒープ’ モジュールを常備するのはいい事だと考えています。:-)

8.5 bisect — 配列二分法アルゴリズム

バージョン 2.1 で追加。

ソースコード: [Lib/bisect.py](#)

このモジュールは、挿入の度にリストをソートすることなく、リストをソートされた順序に保つことをサポートします。大量の比較操作を伴うような、アイテムがたくさんあるリストでは、より一般的なアプローチに比べて、パフォーマンスが向上します。動作に基本的な二分法アルゴリズムを使っているため、*bisect* と呼ばれています。ソースコードはこのアルゴリズムの実例として一番役に立つかもしれません (境界条件はすでに正しいです!)

次の関数が用意されています:

`bisect.bisect_left(a, x, lo=0, hi=len(a))`

ソートされた順序を保ったまま *x* を *a* に挿入できる点を探し当てます。リストの中から検索する部分集合を指定するには、パラメーターの *lo* と *hi* を使います。デフォルトでは、リスト全体が使われます。*x* がすでに *a* に含まれている場合、挿入点は既存のどのエントリーよりも前 (左) になります。戻り値

^{*1} 現在使われているディスクバランス化アルゴリズムは、最近ではもはや巧妙というよりも目障りであり、このためにディスクに対するシーク機能が重要になっています。巨大な容量を持つテープのようにシーク不能なデバイスでは、事情は全く異なり、個々のテープ上の移動が可能な限り効率的に行われるように非常に巧妙な処理を (相当前もって) 行わなければなりません (すなわち、もっとも統合処理の ”進行” に関係があります)。テープによっては逆方向に読むことさえでき、巻き戻しに時間を取られるのを避けるために使うこともできます。正直、本当に良いテープソートは見えていて素晴らしく驚異的なものです！ソートというのは常に偉大な芸術なのです！:-)

は、`list.insert()` の第一引数として使うのに適しています。*a* はすでにソートされているものとします。

返された挿入点 *i* は、配列 *a* を二つに分け、`all(val < x for val in a[lo:i])` が左側に、`all(val >= x for val in a[i:hi])` が右側になるようにします。

```
bisect.bisect_right(a, x, lo=0, hi=len(a))
```

```
bisect.bisect(a, x, lo=0, hi=len(a))
```

`bisect_left()` と似ていますが、*a* に含まれる *x* のうち、どのエントリーよりも後ろ (右) にくるような挿入点を返します。

返された挿入点 *i* は、配列 *a* を二つに分け、`all(val <= x for val in a[lo:i])` が左側に、`all(val > x for val in a[i:hi])` が右側になるようにします。

```
bisect.insort_left(a, x, lo=0, hi=len(a))
```

x を *a* にソートされた順序で挿入します。これは、`a.insert(bisect.bisect_left(a, x, lo, hi), x)` と等価です。*a* はすでにソートされているものとします。なお、 $O(\log n)$ の探索は遅い $O(n)$ の挿入の段階に支配されます。

```
bisect.insort_right(a, x, lo=0, hi=len(a))
```

```
bisect.insort(a, x, lo=0, hi=len(a))
```

`insort_left()` と似ていますが、*a* に含まれる *x* のうち、どのエントリーよりも後ろに *x* を挿入します。

参考:

`bisect` を利用して、直接の探索ができ、キー関数をサポートする、完全な機能を持つコレクションクラスを組み立てる [SortedCollection recipe](#)。キーは、探索中に不必要な呼び出しをさせないために、予め計算しておきます。

8.5.1 ソート済みリストの探索

上記の `bisect()` 関数群は挿入点を探索するのには便利ですが、普通の探索タスクに使うのはトリッキーだったり不器用だったりします。以下の 5 関数は、これらをどのように標準の探索やソート済みリストに変換するかを説明します:

```
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
```

(次のページに続く)

(前のページからの続き)

```

    raise ValueError

def find_le(a, x):
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

def find_ge(a, x):
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

```

8.5.2 その他の使用例

`bisect()` 関数は数値テーブルの探索に役に立ちます。この例では、`bisect()` を使って、(たとえば) 順序のついた数値の区切り点の集合に基づいて、試験の成績の等級を表す文字を調べます。区切り点は 90 以上は 'A'、80 から 89 は 'B'、などです:

```

>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
    i = bisect(breakpoints, score)
    return grades[i]

>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']

```

`sorted()` 関数と違い、`bisect()` 関数に `key` や `reversed` 引数を用意するのは、設計が非効率になるので、非合理的です (連続する `bisect` 関数の呼び出しは前回の `key` 参照の結果を ”記憶” しません)。

代わりに、事前に計算しておいたキーのリストから検索して、レコードのインデックスを見つけます:

```

>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])
>>> keys = [r[1] for r in data]           # precomputed list of keys
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect_left(keys, 5)]

```

(次のページに続く)

(前のページからの続き)

```
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```

8.6 array — 効率のよい数値アレイ

このモジュールでは、基本的な値 (文字、整数、浮動小数点数) のアレイ (array、配列) をコンパクトに表現できるオブジェクト型を定義しています。アレイはシーケンス (sequence) 型であり、中に入れるオブジェクトの型に制限があることを除けば、リストとまったく同じように振る舞います。オブジェクト生成時に一文字の型コードを用いて型を指定します。次の型コードが定義されています:

型コード	C の型	Python の型	最小サイズ (バイト単位)
'c'	char	文字 (str 型)	1
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	Unicode 文字 (unicode 型)	2 (ノートを参照)
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	long	2
'l'	signed long	int	4
'L'	unsigned long	long	4
'f'	float	float	4
'd'	double	float	8

注釈: 'u' 型コードは Python の Unicode 文字列に対応します。一文字幅の文字は 2 バイトで二文字幅の文字は 4 バイトです。

値の実際の表現はマシンアーキテクチャ (厳密に言うと C の実装) によって決まります。値の実際のサイズは `itemsize` 属性から得られます。Python の通常の整数型では C の unsigned (long) 整数の最大範囲を表せないため、'L' と 'I' で表現されている要素に入る値は Python では長整数として表されます。

このモジュールでは次の型を定義しています:

class `array.array` (*typecode* [, *initializer*])

要素のデータ型が *typecode* に限定される新しいアレイで、オプションの値 *initializer* を渡すと初期値になりますが、リスト、文字列または適当な型のイテレーション可能オブジェクトでなければなりません。

バージョン 2.4 で変更: 以前はリストか文字列しか受け付けませんでした。

リストか文字列を渡した場合、新たに作成されたアレイの `fromlist()`、`fromstring()` あるいは `fromunicode()` メソッド (以下を参照して下さい) に渡され、初期値としてアレイに追加されます。

それ以外の場合には、イテレーション可能オブジェクト *initializer* は新たに作成されたオブジェクトの *extend()* メソッドに渡されます。

`array.ArrayType`

`array` の別名です。撤廃されました。

アレイオブジェクトでは、インデックス指定、スライス、連結および反復といった、通常のシーケンスの演算をサポートしています。スライス代入を使うときは、代入値は同じ型コードのアレイオブジェクトでなければなりません。それ以外のオブジェクトを指定すると *TypeError* を送出します。アレイオブジェクトはバッファインタフェースを実装しており、バッファオブジェクトをサポートしている場所ならどこでも利用できます。

次のデータ要素やメソッドもサポートされています:

`array.typecode`

アレイを作るときに使う型コード文字です。

`array.itemsize`

アレイの要素 1 つの内部表現に使われるバイト長です。

`array.append(x)`

値 *x* の新たな要素をアレイの末尾に追加します。

`array.buffer_info()`

アレイの内容を記憶するために使っているバッファの、現在のメモリアドレスと要素数の入ったタプル (*address*, *length*) を返します。バイト単位で表したメモリバッファの大きさは `array.buffer_info()[1] * array.itemsize` で計算できます。例えば `ioctl()` 操作のような、メモリアドレスを必要とする低レベルな (そして、本質的に危険な) I/O インタフェースを使って作業する場合に、ときどき便利です。アレイ自体が存在し、長さを変えるような演算を適用しない限り、有効な値を返します。

注釈: C や C++ で書いたコードからアレイオブジェクトを使う場合 (`buffer_info()` の情報を使う意味のある唯一の方法です) は、アレイオブジェクトでサポートしているバッファインタフェースを使う方がより理にかなっています。このメソッドは後方互換性のために保守されており、新しいコードでの使用は避けるべきです。バッファインタフェースの説明は `bufferobjects` にあります。

`array.byteswap()`

アレイのすべての要素に対して「バイトスワップ」(リトルエンディアンとビッグエンディアンの変換)を行います。このメソッドは大きさが 1、2、4 および 8 バイトの値にのみをサポートしています。他の型の値に使うと *RuntimeError* を送出します。異なるバイトオーダをもつ計算機で書かれたファイルからデータを読み込むときに役に立ちます。

`array.count(x)`

シーケンス中の *x* の出現回数を返します。

`array.extend(iterable)`

iterable から要素を取り出し、アレイの末尾に要素を追加します。 *iterable* が別のアレイ型である場合、

二つのアレイは全く同じ型コードでなければなりません。それ以外の場合には `TypeError` を送出します。 `iterable` がアレイでない場合、アレイに値を追加できるような正しい型の要素からなるイテレーション可能オブジェクトでなければなりません。

バージョン 2.4 で変更: 以前は他のアレイ型しか引数に指定できませんでした。

`array.fromfile(f, n)`

ファイルオブジェクト `f` から (マシン依存のデータ形式そのまま) `n` 個の要素を読み出し、アレイの末尾に要素を追加します。 `n` 個の要素を読めなかったときは `EOFError` を送出しますが、それまでに読み出した値はアレイに追加されています。 `f` は本当の組み込みファイルオブジェクトでなければなりません。 `read()` メソッドをもつ他の型では動作しません。

`array.fromlist(list)`

リストから要素を追加します。型に関するエラーが発生した場合にアレイが変更されないことを除き、`for x in list: a.append(x)` と同じです。

`array.fromstring(s)`

文字列から要素を追加します。文字列は、(ファイルから `fromfile()` メソッドを使って値を読み込んだときのように) マシン依存のデータ形式で表された値の配列として解釈されます。

`array.fromunicode(s)`

指定した Unicode 文字列のデータを使ってアレイを拡張します。アレイの型コードは 'u' でなければなりません。それ以外の場合には、 `ValueError` を送出します。他の型のアレイに Unicode 型のデータを追加するには、 `array.fromstring(unicodestring.encode(enc))` を使ってください。

`array.index(x)`

アレイ中で `x` が出現するインデックスのうち最小の値 `i` を返します。

`array.insert(i, x)`

アレイ中の位置 `i` の前に値 `x` をもつ新しい要素を挿入します。 `i` の値が負の場合、アレイの末尾からの相対位置として扱います。

`array.pop([i])`

アレイからインデックスが `i` の要素を取り除いて返します。オプションの引数はデフォルトで `-1` になっていて、最後の要素を取り除いて返すようになっています。

`array.read(f, n)`

バージョン 1.5.1 で非推奨: `fromfile()` メソッドを使ってください。

ファイルオブジェクト `f` から (マシン依存のデータ形式そのまま) `n` 個の要素を読み出し、アレイの末尾に要素を追加します。 `n` 個の要素を読めなかったときは `EOFError` を送出しますが、それまでに読み出した値はアレイに追加されています。 `f` は本当の組み込みファイルオブジェクトでなければなりません。 `read()` メソッドをもつ他の型では動作しません。

`array.remove(x)`

アレイ中の `x` のうち、最初に現れたものを取り除きます。

`array.reverse()`

アレイの要素の順番を逆にします。

`array.tofile(f)`

アレイのすべての要素をファイルオブジェクト *f* に (マシン依存のデータ形式そのまま) 書き込みます。

`array.tolist()`

アレイを同じ要素を持つ普通のリストに変換します。

`array.tostring()`

アレイをマシン依存のデータアレイに変換し、文字列表現 (`tofile()` メソッドによってファイルに書き込まれるものと同じバイト列) を返します。

`array.tounicode()`

アレイを Unicode 文字列に変換します。アレイの型コードは 'u' でなければなりません。それ以外の場合には `ValueError` を送出します。他の型のアレイから Unicode 文字列を得るには、`array.tostring().decode(enc)` を使ってください。

`array.write(f)`

バージョン 1.5.1 で非推奨: `tofile()` メソッドを使ってください。

アレイのすべての要素をファイルオブジェクト *f* に (マシン依存のデータ形式そのまま) 書き込みます。

アレイオブジェクトを表示したり文字列に変換したりすると、`array(typecode, initializer)` という形式で表現されます。アレイが空の場合は `initializer` の表示を省略します。アレイが空でなければ、`typecode` が 'c' の場合には文字列に、それ以外の場合には数値のリストになります。`array` クラスが `from array import array` というふうにインポートされている限り、変換後の文字列に `eval()` を用いると元のアレイオブジェクトと同じデータ型と値を持つアレイに逆変換できることが保証されています。文字列表現の例を以下に示します:

```
array('l')
array('c', 'hello world')
array('u', u'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

参考:

`struct` モジュール 異なる種類のバイナリデータのパックおよびアンパック。

`xdrlib` モジュール 遠隔手続き呼び出しシステムで使われる外部データ表現仕様 (External Data Representation, XDR) のデータのパックおよびアンパック。

Numerical Python ドキュメント Numeric Python 拡張モジュール (NumPy) では、別の方法でシーケンス型を定義しています。Numerical Python に関する詳しい情報は <http://www.numpy.org/> を参照してください。

8.7 sets — 重複のない要素の順序なしコレクション

バージョン 2.3 で追加.

バージョン 2.6 で非推奨: 組み込みの `set/frozenset` 型がこのモジュールを置き換えます。

`sets` モジュールは、ユニークな要素の順序なしコレクションを構築し、操作するためのクラスを提供します。帰属関係のテストやシーケンスから重複を取り除いたり、積集合・和集合・差集合・対称差集合のような標準的な数学操作などを含みます。

集合は、他のコレクションと同様、`x in set`, `len(set)`, `for x in set` をサポートします。コレクションには順序がないので、集合は挿入の順序や要素の位置を記録しません。従って、集合はインデクシング、スライシング、その他のシーケンス的な振舞いをサポートしません。

ほとんどの集合のアプリケーションは、`__hash__()` を除いてすべての集合のメソッドを提供する `Set` クラスを使用します。ハッシュを要求する高度なアプリケーションについては、`ImmutableSet` クラスが `__hash__()` メソッドを加えているが、集合の内容を変更するメソッドは省略されます。`Set` と `ImmutableSet` は、何が集合であるか決めるのに役立つ (`isinstance(obj, BaseSet)`) 抽象クラス `BaseSet` から派生します。

集合クラスは辞書を使用して実装されます。このことから、集合の要素にするには辞書のキーと同様の要件を満たさなければなりません。具体的には、要素になるものには `__eq__()` と `__hash__()` が定義されているという条件です。その結果、集合はリストや辞書のような変更可能な要素を含むことができません。しかしそれらは、タプルや `ImmutableSet` のインスタンスのような不変コレクションを含むことができます。集合の集合の実装中の便宜については、内部集合が自動的に変更不可能な形式に変換されます。例えば、`Set([Set(['dog'])])` は `Set([ImmutableSet(['dog'])])` へ変換されます。

class `sets.Set` (`[iterable]`)

新しい空の `Set` オブジェクトを構築します。もしオプション `iterable` が与えられたら、イテレータから得られた要素を備えた集合として更新します。`iterable` 中の全ての要素は、変更不可能であるか、または [不変に自動変換するためのプロトコル](#) で記述されたプロトコルを使って変更不可能なものに変換可能であるべきです。

class `sets.ImmutableSet` (`[iterable]`)

新しい空の `ImmutableSet` オブジェクトを構築します。もしオプション `iterable` が与えられたら、イテレータから得られた要素を備えた集合として更新します。`iterable` 中の全ての要素は、変更不可能であるか、または [不変に自動変換するためのプロトコル](#) で記述されたプロトコルを使って変更不可能なものに変換可能であるべきです。

`ImmutableSet` オブジェクトは `__hash__()` メソッドを備えているので、集合要素または辞書キーとして使用することができます。`ImmutableSet` オブジェクトは要素を加えたり取り除いたりするメソッドを持っていません。したがって、コンストラクタが呼ばれたとき要素はすべて知られていなければなりません。

8.7.1 Set オブジェクト

`Set` と `ImmutableSet` のインスタンスはともに、以下の操作を備えています:

演算	等価な演算	結果
<code>len(s)</code>		集合 s の要素数 (濃度)
<code>x in s</code>		x が s に帰属していれば真を返す
<code>x not in s</code>		x が s に帰属していなければ真を返す
<code>s.issubset(t)</code>	$s \leq t$	s のすべての要素が t に帰属していれば真を返す
<code>s.issuperset(t)</code>	$s \geq t$	t のすべての要素が s に帰属していれば真を返す
<code>s.union(t)</code>	$s \mid t$	s と t の両方の要素からなる新しい集合
<code>s.intersection(t)</code>	$s \& t$	s と t で共通する要素からなる新しい集合
<code>s.difference(t)</code>	$s - t$	s にあるが t にない要素からなる新しい集合
<code>s.symmetric_difference(t)</code>	$s \wedge t$	s と t のどちらか一方だけに属する要素からなる集合
<code>s.copy()</code>		s の浅いコピーからなる集合

演算子を使わない書き方である `union()`, `intersection()`, `difference()`, および `symmetric_difference()` は任意のイテレータ可能オブジェクトを引数として受け取るのに対し、演算子を使った書き方の方では引数は集合型でなければならないので注意してください。これはエラーの元となる `Set('abc') & 'cbs'` のような書き方を排除し、より可読性のある `Set('abc').intersection('cbs')` を選ばせるための仕様です。

バージョン 2.3.1 で変更: 以前は全ての引数が集合型でなければなりませんでした。

加えて、`Set` と `ImmutableSet` は集合間の比較をサポートしています。二つの集合は、各々の集合のすべての要素が他方に含まれて (各々が他方の部分集合) いる場合、かつその場合に限り等価になります。ある集合は、他方の集合の真の部分集合 (proper subset、部分集合であるが非等価) である場合、かつその場合に限り、他方の集合より小さくなります。ある集合は、他方の集合の真の上位集合 (proper superset、上位集合であるが非等価) である場合、かつその場合に限り、他方の集合より大きくなります。

部分集合比較やと等値比較では、完全な順序決定関数を一般化できません。たとえば、互いに素な2つの集合は等しくありませんし、互いの部分集合でもないので、 $a < b$, $a == b$, $a > b$ はすべて `False` を返します。したがって集合は `__cmp__()` メソッドを実装しません。

集合は半順序 (部分集合関係) しか定義しないので、集合のリストにおける `list.sort()` メソッドの出力は未定義です。

以下は `ImmutableSet` で利用可能であるが `Set` にはない操作です:

演算	結果
<code>hash(s)</code>	s のハッシュ値を返す

以下は `Set` で利用可能であるが `ImmutableSet` にはない操作です:

演算	等価な演算	結果
<code>s.update(t)</code>	$s \mid= t$	t を加えた要素からなる集合 s を返します
<code>s.intersection_update(t)</code>	$s \&= t$	t でも見つかった要素だけを持つ集合 s を返します
<code>s.difference_update(t)</code>	$s -= t$	t にあった要素を取り除いた後の集合 s を返します
<code>s.symmetric_difference_update(t)</code>	$s \hat{=} t$	s と t のどちらか一方だけに属する要素からなる集合 s を返します
<code>s.add(x)</code>		要素 x を集合 s に加えます
<code>s.remove(x)</code>		要素 x を集合 s から取り除きます; x がなければ <code>KeyError</code> を送出します
<code>s.discard(x)</code>		要素 x が存在すれば、集合 s から取り除きます
<code>s.pop()</code>		s から任意に要素を取り除き、それを返します; 集合が空なら <code>KeyError</code> を送出します
<code>s.clear()</code>		集合 s からすべての要素を取り除きます

演算子を使わない書き方である `update()`, `intersection_update()`, `difference_update()`, および `symmetric_difference_update()` は任意のイテレータ可能オブジェクトを引数として受け取ること注目してください。

バージョン 2.3.1 で変更: 以前は全ての引数が集合型でなければなりませんでした。

もう一つ注意を述べますが、このモジュールでは `union_update()` が `update()` の別名として含まれています。このメソッドは後方互換性のために残されているものです。プログラマは組み込みの `set()` および `frozenset()` でサポートされている `update()` を選ぶべきです。

8.7.2 使用例

```
>>> from sets import Set
>>> engineers = Set(['John', 'Jane', 'Jack', 'Janice'])
>>> programmers = Set(['Jack', 'Sam', 'Susan', 'Janice'])
>>> managers = Set(['Jane', 'Jack', 'Susan', 'Zack'])
>>> employees = engineers | programmers | managers           # union
>>> engineering_management = engineers & managers           # intersection
>>> fulltime_management = managers - engineers - programmers # difference
>>> engineers.add('Marvin')                                   # add element
>>> print engineers
Set(['Jane', 'Marvin', 'Janice', 'John', 'Jack'])
>>> employees.issuperset(engineers)                          # superset test
False
>>> employees.update(engineers)                               # update from another set
>>> employees.issuperset(engineers)
True
>>> for group in [engineers, programmers, managers, employees]:
...     group.discard('Susan')                               # unconditionally remove element
```

(次のページに続く)

(前のページからの続き)

```

...     print group
...
Set(['Jane', 'Marvin', 'Janice', 'John', 'Jack'])
Set(['Janice', 'Jack', 'Sam'])
Set(['Jane', 'Zack', 'Jack'])
Set(['Jack', 'Sam', 'Jane', 'Marvin', 'Janice', 'John', 'Zack'])

```

8.7.3 不変に自動変換するためのプロトコル

集合は変更不可能な要素だけを含むことができます。都合上、変更可能な `Set` オブジェクトは、集合要素として加えられる前に、自動的に `ImmutableSet` へコピーします。

そのメカニズムはハッシュ可能な (*hashable*) 要素を常に加えることですが、もしハッシュ不可能な場合は、その要素は変更不可能な等価物を返す `__as_immutable__()` メソッドを持っているかどうかチェックされます。

`Set` オブジェクトは、`ImmutableSet` のインスタンスを返す `__as_immutable__()` メソッドを持っているので、集合の集合を構築することが可能です。

集合内のメンバーであることをチェックするために、要素をハッシュする必要がある `__contains__()` メソッドと `remove()` メソッドが、同様のメカニズムを必要としています。これらのメソッドは要素がハッシュできるかチェックします。もし出来なければ `__hash__()`, `__eq__()`, `__ne__()` のための一時的なメソッドを備えたクラスによってラップされた要素を返すメソッド `__as_temporarily_immutable__()` メソッドをチェックします。

代理メカニズムは、オリジナルの可変オブジェクトから分かれたコピーを組み上げる手間を助けてくれます。

`Set` オブジェクトは、新しいクラス `TemporarilyImmutableSet` によってラップされた `Set` オブジェクトを返す、`__as_temporarily_immutable__()` メソッドを実装します。

ハッシュ可能を与えるための 2 つのメカニズムは通常ユーザーに見えません。しかしながら、マルチスレッド環境下においては、`TemporarilyImmutableSet` によって一時的にラップされたものを持っているスレッドがあるときに、もう一つのスレッドが集合を更新することで、衝突を発生させることができます。言い換えれば、変更可能な集合の集合はスレッドセーフではありません。

8.7.4 組み込み `set` 型との比較

組み込みの `set` および `frozenset` 型はこの *sets* で学んだことを生かして設計されています。主な違いは次の通りです。

- `Set` と `ImmutableSet` は `set` と `frozenset` に改名されました。
- `BaseSet` に相当するものではありません。代わりに `isinstance(x, (set, frozenset))` を使って下さい。

- 組み込みのものに使われているハッシュアルゴリズムは、多くのデータ集合に対してずっと良い性能(少ない衝突)を実現します。
- 組み込みのものはより空間効率良く pickle 化できます。
- 組み込みのものには `union_update()` メソッドがありません。代わりに同じ機能の `update()` メソッドを使って下さい。
- 組み込みのものには `_repr(sorted=True)` メソッドがありません。代わりに組み込み関数の `repr()` と `sorted()` を使って `repr(sorted(s))` として下さい。
- 組み込みのものは変更不可能なものに自動で変換するプロトコルがありません。この機能は多くの人が困惑を覚えるわりに、コミュニティの誰からも実例的な使用例の報告がありませんでした。

8.8 sched — イベントスケジューラ

ソースコード: [Lib/sched.py](#)

`sched` モジュールは一般的な目的のためのイベントスケジューラを実装するクラスを定義します:

class `sched.scheduler` (*timefunc*, *delayfunc*)

`scheduler` クラスはイベントをスケジュールするための一般的なインタフェースを定義します。それは "外の世界" を実際に扱うための 2 つの関数を必要とします — *timefunc* は引数なしで呼ばれて 1 つの数値を返す callable オブジェクトでなければなりません (戻り値は任意の単位で「時間」を表します)。*delayfunc* は 1 つの引数を持つ callable オブジェクトでなければならず、その時間だけ遅延する必要があります (引数は *timefunc* の出力と互換)。*delayfunc* は、各々のイベントが実行された後に引数 0 で呼ばれることがあります。これは、マルチスレッドアプリケーションの中で他のスレッドが実行する機会を与えるためです。

例:

```
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(): print "From print_time", time.time()
...
>>> def print_some_times():
...     print time.time()
...     s.enter(5, 1, print_time, ())
...     s.enter(10, 1, print_time, ())
...     s.run()
...     print time.time()
...
>>> print_some_times()
930343690.257
From print_time 930343695.274
From print_time 930343700.273
930343700.276
```

マルチスレッド環境において、`scheduler` クラスにはスレッドセーフのための制限があります。イベントキューが空になるまでは、実行中のスケジューラで現在中断中でメインスレッドを足止めしているタスクの前に、新しいタスクを挿入することはできません。代わりに、より推奨される方法として、`threading.Timer` クラスを利用してください。

例:

```
>>> import time
>>> from threading import Timer
>>> def print_time():
...     print "From print_time", time.time()
...
>>> def print_some_times():
...     print time.time()
...     Timer(5, print_time, ()).start()
...     Timer(10, print_time, ()).start()
...     time.sleep(11) # sleep while time-delay events execute
...     print time.time()
...
>>> print_some_times()
930343690.257
From print_time 930343695.274
From print_time 930343700.273
930343701.301
```

8.8.1 スケジューラオブジェクト

`scheduler` インスタンスは以下のメソッドと属性を持っています:

`scheduler.enterabs(time, priority, action, argument)`

新しいイベントをスケジュールします。引数 `time` は、コンストラクタへ渡された `timefunc` の戻り値と互換な数値型でなければいけません。同じ `time` によってスケジュールされたイベントは、それらの `priority` によって実行されます。数値の小さい方が高い優先度となります。

イベントを実行することは、`action(*argument)` を実行することを意味します。`argument` は `action` のためのパラメータを保持するシーケンスでなければいけません。

戻り値は、イベントのキャンセル後に使われるかもしれないイベントです (`cancel()` を見よ)。

`scheduler.enter(delay, priority, action, argument)`

時間単位以上の `delay` でイベントをスケジュールします。相対的時間以外の、引数、効果、戻り値は、`enterabs()` に対するものと同じです。

`scheduler.cancel(event)`

キューからイベントを消去します。もし `event` がキューにある現在のイベントでないならば、このメソッドは `ValueError` を送出します。

`scheduler.empty()`

もしイベントキューが空ならば、`True` を返します。

`scheduler.run()`

すべてのスケジュールされたイベントを実行します。この関数は次のイベントを (コンストラクタへ渡された関数 `delayfunc()` を使うことで) 待ち、そしてそれを実行し、スケジュールされたイベントがなくなるまで同じことを繰り返します。

`action` あるいは `delayfunc` は例外を投げることができます。いずれの場合も、スケジューラは一貫した状態を維持し、例外を伝播するでしょう。例外が `action` によって投げられる場合、イベントは `run()` への呼出しを未来に行なわないでしょう。

イベントのシーケンスが、次イベントの前に、利用可能時間より実行時間が長いと、スケジューラは単に遅れることになるでしょう。イベントが落ちることはありません; 呼出しコードはもはや適切でないキャンセルイベントに対して責任があります。

`scheduler.queue`

読み取り専用の属性で、これから起こるイベントが実行される順序で格納されたリストを返します。各イベントは、次の属性 `time`, `priority`, `action`, `argument` を持った *named tuple* の形式になります。

バージョン 2.6 で追加。

8.9 mutex — 排他制御

バージョン 2.6 で非推奨: `mutex` モジュールは Python 3 で削除されました。

`mutex` モジュールでは、ロック (lock) の獲得と解除によって排他制御を可能にするクラスを定義しています。排他制御は `threading` やマルチタスクを使う上で便利かもしれませんが、このクラスがそうした機能を必要として (いたり、想定して) いるわけではありません。

`mutex` モジュールでは以下のクラスを定義しています:

class `mutex.mutex`

新しい (ロックされてない) `mutex` を作ります。

`mutex` には 2 つの状態変数 — ”ロック” ビット (locked bit) とキュー (queue) があります。 `mutex` がロックされていなければ、キューは空です。それ以外の場合、キューは空になっているか、 (`function`, `argument`) のペアが一つ以上入っています。このペアはロックを獲得しようと待機している関数 (またはメソッド) を表しています。キューが空でないときに `mutex` をロック解除すると、キューの先頭のエントリをキューから除去し、そのエントリのペアに基づいて `function(argument)` を呼び出します。これによって、先頭にあったエントリが新たなロックを獲得します。

当然のことながらマルチスレッドの制御には利用できません — というのも、 `lock()` が、ロックを獲得したら関数を呼び出すという変なインタフェースだからです。

8.9.1 mutex オブジェクト

`mutex` には以下のメソッドがあります:

`mutex.test()`

`mutex` がロックされているかどうか調べます。

`mutex.testandset()`

「原子的 (Atomic)」な Test-and-Set 操作です。ロックがセットされていなければ獲得して `True` を返します。それ以外の場合には `False` を返します。

`mutex.lock(function, argument)`

`mutex` がロックされていなければ `function(argument)` を実行します。`mutex` がロックされている場合、関数とその引数をキューに置きます。キューに置かれた `function(argument)` がいつ実行されるかについては `unlock()` を参照してください。

`mutex.unlock()`

キューが空ならば `mutex` をロック解除します。そうでなければ、キューの最初の要素を実行します。

8.10 Queue — 同期キュークラス

注釈: `Queue` モジュールは、Python 3 では `queue` にリネームされました。2to3 ツールが自動的にソースコードの `import` を修正します。

ソースコード: [Lib/Queue.py](#)

`Queue` モジュールは、複数プロデューサ-複数コンシューマ (multi-producer, multi-consumer) キューを実装します。これは、複数のスレッドの間で情報を安全に交換しなければならないときのマルチスレッドプログラミングで特に有益です。このモジュールの `Queue` クラスは、必要なすべてのロックセマンティクスを実装しています。これは Python のスレッドサポートの状況に依存します。 `threading` モジュールを参照してください。

このモジュールでは 3 種類のキューが実装されています。それらはキューから取り出されるエントリの順番だけが違います。FIFO キューでは、最初に追加されたエントリが最初に取り出されます。LIFO キューでは、最後に追加されたエントリが最初に取り出されます (スタックのように振る舞います)。優先順位付きキュー (priority queue) では、エントリは (`heapq` モジュールを利用して) ソートされ、最も低い値のエントリが最初に取り出されます。

`Queue` モジュールは以下のクラスと例外を定義します:

class `Queue.Queue(maxsize=0)`

FIFO キューのコンストラクタです。`maxsize` はキューに置くことのできる要素数の上限を設定する整数です。いったんこの大きさに達したら、挿入はキューの要素が消費されるまでブロックされます。もし `maxsize` が 0 以下であるならば、キューの大きさは無限です。

class `Queue.LifoQueue(maxsize=0)`

LIFO キューのコンストラクタです。`maxsize` はキューに置くことのできる要素数の上限を設定する整

数です。いったんこの大きさに達したら、挿入はキューの要素が消費されるまでブロックされます。もし `maxsize` が 0 以下であるならば、キューの大きさは無限です。

バージョン 2.6 で追加.

class `Queue.PriorityQueue` (`maxsize=0`)

優先順位付きキューのコンストラクタです。 `maxsize` はキューに置くことのできる要素数の上限を設定する整数です。いったんこの大きさに達したら、挿入はキューの要素が消費されるまでブロックされます。もし `maxsize` が 0 以下であるならば、キューの大きさは無限です。

最小の値を持つ要素が最初に検索されます (最小の値を持つ値は、 `sorted(list(entries))[0]` によって返されるものです)。典型的な要素のパターンは、 `(priority_number, data)` 形式のタプルです。

バージョン 2.6 で追加.

exception `Queue.Empty`

空な `Queue` オブジェクトで、非ブロックメソッドとして `get()` (または `get_nowait()`) が呼ばれたとき、送出される例外です。

exception `Queue.Full`

満杯な `Queue` オブジェクトで、非ブロックメソッドとして `put()` (または `put_nowait()`) が呼ばれたとき、送出される例外です。

参考:

`collections.deque` は、ロックなしで `popleft()` や `append()` といったアトミック操作が可能なキューの実装です。

8.10.1 キューオブジェクト

キューオブジェクト (`Queue`, `LifoQueue`, `PriorityQueue`) は、以下の public メソッドを提供しています。

`Queue.qsize()`

キューの近似サイズを返します。ここで、 `qsize() > 0` は後続の `get()` がブロックしないことを保証しないこと、また `qsize() < maxsize` が `put()` がブロックしないことを保証しないことに注意してください。

`Queue.empty()`

キューが空の場合は `True` を返し、そうでなければ `False` を返します。 `empty()` が `True` を返しても、後続の `put()` の呼び出しがブロックしないことは保証されません。同様に、 `empty()` が `False` を返しても、後続の `get()` の呼び出しがブロックしないことは保証されません。

`Queue.full()`

キューが一杯の場合は `True` を返し、そうでなければ `False` を返します。 `full()` が `True` を返しても、後続の `get()` の呼び出しがブロックしないことは保証されません。同様に、 `full()` が `False` を返しても、後続の `put()` の呼び出しがブロックしないことは保証されません。

`Queue.put(item[, block[, timeout]])`

`item` をキューに入れます。もしオプション引数 `block` が真で `timeout` が `None` (デフォルト) の場合は、必要であればフリースロットが利用可能になるまでブロックします。`timeout` が正の数の場合は、最大で `timeout` 秒間ブロックし、その時間内に空きスロットが利用可能にならないければ、例外 `Full` を送出します。そうでない場合 (`block` が偽) は、空きスロットが直ちに利用できるならば、キューにアイテムを置きます。できないならば、例外 `Full` を送出します (この場合 `timeout` は無視されます)。

バージョン 2.3 で追加: `timeout` パラメータが追加されました。

`Queue.put_nowait(item)`

`put(item, False)` と等価です。

`Queue.get([block[, timeout]])`

キューからアイテムを取り除き、それを返します。オプション引数 `block` が真で `timeout` が `None` (デフォルト) の場合は、必要であればアイテムが取り出せるようになるまでブロックします。もし `timeout` が正の数の場合は、最大で `timeout` 秒間ブロックし、その時間内でアイテムが取り出せるようにならないければ、例外 `Empty` を送出します。そうでない場合 (`block` が偽) は、直ちにアイテムが取り出せるならば、それを返します。できないならば、例外 `Empty` を送出します (この場合 `timeout` は無視されます)。

バージョン 2.3 で追加: `timeout` パラメータが追加されました。

`Queue.get_nowait()`

`get(False)` と等価です。

キューに入れられたタスクが全てコンシューマスレッドに処理されたかどうかを追跡するために 2 つのメソッドが提供されます。

`Queue.task_done()`

過去にキューに入れられたタスクが完了した事を示します。キューのコンシューマスレッドに利用されます。タスクの取り出しに使われた、各 `get()` に対して、それに続く `task_done()` の呼び出しは、取り出したタスクに対する処理が完了した事をキューに教えます。

`join()` がブロックされていた場合、全 item が処理された (キューに `put()` された全ての item に対して `task_done()` が呼び出されたことを意味します) 時に復帰します。

キューにある要素より多く呼び出された場合 `ValueError` が発生します。

バージョン 2.5 で追加。

`Queue.join()`

キューの中の全アイテムが処理される間でブロックします。

キューに item が追加される度に、未完了タスクカウントが増やされます。コンシューマスレッドが `task_done()` を呼び出して、item を受け取ってそれに対する処理が完了した事を知らせる度に、未完了タスクカウントが減らされます。未完了タスクカウントが 0 になったときに、`join()` のブロックが解除されます。

バージョン 2.5 で追加。

キューに入れたタスクが完了するのを待つ例:

```
def worker():
    while True:
        item = q.get()
        do_work(item)
        q.task_done()

q = Queue()
for i in range(num_worker_threads):
    t = Thread(target=worker)
    t.daemon = True
    t.start()

for item in source():
    q.put(item)

q.join()          # block until all tasks are done
```

8.11 weakref — 弱参照

バージョン 2.1 で追加.

ソースコード: [Lib/weakref.py](#)

`weakref` モジュールは、Python プログラマがオブジェクトへの弱参照 (*weak reference*) を作成できるようにします。

以下では、リファレント (*referent*) という用語は弱参照が参照するオブジェクトを意味します。

オブジェクトに対する弱参照は、そのオブジェクトを生かしておく十分条件にはなりません: あるリファレントに対する参照が弱参照しか残っていない場合、ガベージコレクション (*garbage collection*) 機構は自由にリファレントを破壊し、そのメモリを別の用途に再利用できます。弱参照の主な用途は、巨大なオブジェクトを保持するキャッシュやマップ型の実装において、キャッシュやマップ型にあるという理由だけオブジェクトを存続させたくない場合です。

例えば、巨大なバイナリ画像のオブジェクトがたくさんあり、それぞれに名前を関連付けたいとします。Python の辞書型を使って名前を画像に対応付けたり画像を名前に対応付けたりすると、画像オブジェクトは辞書内のキーや値に使われているため存続しつづけることになります。 `weakref` モジュールが提供している `WeakKeyDictionary` や `WeakValueDictionary` クラスはその代用で、対応付けを構築するのに弱参照を使い、キャッシュやマッピングに存在するという理由だけでオブジェクトを存続させないようにします。例えば、もしある画像オブジェクトが `WeakValueDictionary` の値になっていた場合、最後に残った画像オブジェクトへの参照を弱参照マッピングが保持していれば、ガベージコレクションはこのオブジェクトを再利用でき、画像オブジェクトに対する弱参照内の対応付けは削除されます。

`WeakKeyDictionary` や `WeakValueDictionary` は弱参照を使って実装されていて、キーや値がガベージコレクションによって回収されたことを弱参照辞書に知らせるような弱参照オブジェクトのコールバック関数を設定しています。ほとんどのプログラムが、いずれかの弱参照辞書型を使うだけで必要を満たせるはずで

す — 自作の弱参照辞書を直接作成する必要は普通はありません。とはいえ、高度な使い方をするために、弱参照辞書の実装に使われている低水準の機構は `weakref` モジュールで公開されています。

すべてのオブジェクトを弱参照できるわけではありません。弱参照できるオブジェクトは、クラスインスタンス、(C ではなく) Python で書かれた関数、(束縛および非束縛の両方の) メソッド、`set` と `frozenset` 型、ファイルオブジェクト、ジェネレータ (*generator*)、type オブジェクト、`bsddb` モジュールの `DBcursor` 型、ソケット型、`array` 型、`deque` 型、正規表現パターンオブジェクト、code オブジェクトです。

バージョン 2.4 で変更: ファイル、ソケット、`array`、および正規表現パターンのサポートを追加しました。

バージョン 2.7 で変更: `thread.lock`, `threading.Lock`, code オブジェクトのサポートが追加されました。

`list` や `dict` など、いくつかの組み込み型は弱参照を直接サポートしませんが、以下のようにサブクラス化を行えばサポートを追加できます:

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)    # this object is weak referenceable
```

`tuple` や `long` など、他の組み込み型はサブクラス化しても弱参照をサポートしません。

拡張型は、簡単に弱参照をサポートできます。詳細については、`weakref-support` を参照してください。

`class weakref.ref(object[, callback])`

`object` への弱参照を返します。リファレントがまだ生きているならば、元のオブジェクトは参照オブジェクトの呼び出しで取り出せず。リファレントがもはや生きていないならば、参照オブジェクトを呼び出したときに `None` を返します。 `callback` に `None` 以外の値を与えた場合、オブジェクトをまさに後始末処理しようとするときに呼び出します。このとき弱参照オブジェクトは `callback` の唯一のパラメータとして渡されます。リファレントはもはや利用できません。

同じオブジェクトに対してたくさんの弱参照を作れます。それぞれの弱参照に対して登録されたコールバックは、もっとも新しく登録されたコールバックからもっとも古いものへと呼び出されます。

コールバックが発生させた例外は標準エラー出力に書き込まれますが、伝播されません。それらはオブジェクトの `__del__()` メソッドが発生させる例外と完全に同じ方法で処理されます。

`object` がハッシュ可能 (*hashable*) ならば、弱参照はハッシュ可能です。それらは `object` が削除された後もそれらのハッシュ値を保持します。 `object` が削除されてから初めて `hash()` が呼び出された場合に、その呼び出しは `TypeError` を発生させます。

弱参照は等価性のテストをサポートしていますが、順序をサポートしていません。参照がまだ生きているならば、`callback` に関係なく二つの参照はそれらのリファレントと同じ等価関係を持ちます。リファレントのどちらか一方が削除された場合、参照オブジェクトが同一である場合に限り、その参照は等価です。

バージョン 2.4 で変更: 以前はファクトリでしたが、サブクラス化可能な型になりました。 `object` 型から派生しています。

`weakref.proxy(object[, callback])`

弱参照を使う `object` へのプロキシを返します。弱参照オブジェクトを明示的な参照外しをしながら利用

する代わりに、多くのケースでプロキシを利用することができます。返されるオブジェクトは、*object* が呼び出し可能かどうかによって、*ProxyType* または *CallableProxyType* のどちらかの型を持ちます。プロキシオブジェクトはリファレントに関係なくハッシュ可能 (*hashable*) ではありません。これによって、それらの基本的な変更可能という性質による多くの問題を避けています。そして、辞書のキーとしての利用を妨げます。*callback* は *ref()* 関数の同じ名前のパラメータと同じものです。(—訳注: リファレントが変更不能型であっても、プロキシはリファレントが消えるという状態の変更があるために、変更可能型です。—)

`weakref.getweakrefcount(object)`

object を参照する弱参照とプロキシの数を返します。

`weakref.getweakrefs(object)`

object を参照するすべての弱参照とプロキシオブジェクトのリストを返します。

class `weakref.WeakKeyDictionary([dict])`

キーを弱参照するマッピングクラス。キーへの強参照がなくなったときに、辞書のエントリは捨てられます。アプリケーションの他の部分が所有するオブジェクトへ属性を追加することもなく、それらのオブジェクトに追加データを関連づけるために使うことができます。これは属性へのアクセスをオーバーライドするオブジェクトに特に便利です。

注釈: 警告: *WeakKeyDictionary* は Python 辞書型の上に作られているので、反復処理を行うときにはサイズ変更してはなりません。*WeakKeyDictionary* の場合、反復処理の最中にプログラムが行った操作が、(ガベージコレクションの副作用として)「魔法のように」辞書内の要素を消し去ってしまうため、確実なサイズ変更は困難なのです。

WeakKeyDictionary オブジェクトは、以下の追加のメソッドを持ちます。これらのメソッドは、内部の参照を直接公開します。その参照は、利用される時に生存しているとは限りません。なので、参照を利用する前に、その参照をチェックする必要があります。これにより、必要なくなったキーの参照が残っているために、ガベージコレクタがそのキーを削除できなくなる事態を避ける事ができます。

`WeakKeyDictionary.iterkeyrefs()`

キーへの弱参照を持つ iterable オブジェクトを返します。

バージョン 2.5 で追加。

`WeakKeyDictionary.keyrefs()`

キーへの弱参照のリストを返します。

バージョン 2.5 で追加。

class `weakref.WeakValueDictionary([dict])`

値を弱参照するマッピングクラス。値への強参照が存在しなくなったときに、辞書のエントリは捨てられます。

注釈: 警告: *WeakValueDictionary* は Python 辞書型の上に作られているので、反復処理を行うときにはサイズ変更してはなりません。*WeakValueDictionary* の場合、反復処理の最中にプログラ

ムが行った操作が、(ガベージコレクションの副作用として)「魔法のように」辞書内の要素を消し去ってしまうため、確実なサイズ変更は困難なのです。

`WeakValueDictionary` オブジェクトは次のメソッドも備えています。これらのメソッドは、`WeakKeyDictionary` オブジェクトの `iterkeyrefs()` メソッドや `keyrefs()` メソッドと同じ問題を抱えています。

`WeakValueDictionary.itervaluerefs()`

値への弱参照を持つ iterable オブジェクトを返します。

バージョン 2.5 で追加.

`WeakValueDictionary.valuerefs()`

値への弱参照のリストを返します。

バージョン 2.5 で追加.

`class weakref.WeakSet([elements])`

要素への弱参照を持つ集合型。要素への強参照が無くなったときに、その要素は削除されます。

バージョン 2.7 で追加.

`weakref.ReferenceType`

弱参照オブジェクトのための型オブジェクト。

`weakref.ProxyType`

呼び出し可能でないオブジェクトのプロキシのための型オブジェクト。

`weakref.CallableProxyType`

呼び出し可能なオブジェクトのプロキシのための型オブジェクト。

`weakref.ProxyTypes`

プロキシのためのすべての型オブジェクトを含むシーケンス。これは両方のプロキシ型の名前付けに依存しないで、オブジェクトがプロキシかどうかのテストをより簡単にできます。

`exception weakref.ReferenceError`

プロキシオブジェクトが使われても、元のオブジェクトがガベージコレクションされてしまっているときに発生する例外。これは標準の `ReferenceError` 例外と同じです。

参考:

PEP 205 - 弱参照 この機能の提案と理論的根拠。初期の実装と他の言語における類似の機能についての情報へのリンクを含んでいます。

8.11.1 弱参照オブジェクト

弱参照オブジェクトは属性あるいはメソッドを持ちません。しかし、リファレントがまだ存在するならば、呼び出すことでそのリファレントを取得できるようにします:

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

リファレントがもはや存在しないならば、参照オブジェクトの呼び出しは `None` を返します:

```
>>> del o, o2
>>> print r()
None
```

弱参照オブジェクトがまだ生きているかどうかのテストは、式 `ref() is not None` を用いて行われます。通常、参照オブジェクトを使う必要があるアプリケーションコードはこのパターンに従います:

```
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print "Object has been deallocated; can't frobnicate."
else:
    print "Object is still live!"
    o.do_something_useful()
```

”生存性 (liveness)”のテストを分割すると、スレッド化されたアプリケーションにおいて競合状態を作り出します。(訳注: `if r() is not None: r().do_something()` では、2 度目の `r()` が `None` を返す可能性があります) 弱参照が呼び出される前に、他のスレッドは弱参照が無効になる原因となり得ます。上で示したイディオムは、シングルスレッドのアプリケーションと同じくマルチスレッド化されたアプリケーションにおいても安全です。

サブクラス化を行えば、`ref` オブジェクトの特殊なバージョンを作成できます。これは `WeakValueDictionary` の実装で使われており、マップ内の各エントリによるメモリのオーバーヘッドを減らしています。こうした実装は、ある参照に追加情報を関連付けたい場合に便利です、リファレントを取り出すための呼び出し時に何らかの追加処理を行いたい場合にも使えます。

以下の例では、`ref` のサブクラスを使って、あるオブジェクトに追加情報を保存し、リファレントがアクセスされたときにその値に作用をできるようにするための方法を示しています:

```
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, **annotations):
        super(ExtendedRef, self).__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.iteritems():
```

(次のページに続く)

(前のページからの続き)

```

        setattr(self, k, v)

    def __call__(self):
        """Return a pair containing the referent and the number of
        times the reference has been called.
        """
        ob = super(ExtendedRef, self).__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob

```

8.11.2 例

この簡単な例では、アプリケーションが以前に参照したオブジェクトを取り出すためにオブジェクト ID を利用する方法を示します。オブジェクトに生きたままであることを強制することなく、オブジェクトの ID を他のデータ構造の中で使うことができ、必要に応じて ID からオブジェクトを取り出せます。

```

import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]

```

8.12 UserDict — 辞書オブジェクトのためのクラスラッパー

ソースコード: [Lib/UserDict.py](#)

このモジュールは最小限のマッピングインターフェイスをすでに持っているクラスのために、すべての辞書メソッドを定義している mixin、`DictMixin` を定義しています。これによって、(shelve モジュールのような) 辞書の代わりをする必要があるクラスを書くことが非常に簡単になります。

このモジュールでは `UserDict` クラスも定義しています。これは辞書オブジェクトのラッパーとして動作します。これの必要性については既に `dict` を直接的にサブクラス化出来るようになった (Python 2.2 から利用可能な機能です) ことによって大幅に取って代わられています。`dict` の導入以前には、`UserDict` クラスはメソッドをオーバーライドしたり新しいメソッドを追加して、辞書風のサブクラスを作成するために使われていました。

`UserDict` モジュールは `UserDict` クラスと `DictMixin` を定義しています:

```
class UserDict.UserDict([initialdata])
```

辞書をシミュレートするクラスです。インスタンスの内容は通常の辞書に保存され、`UserDict` インスタンスの `data` 属性を通してアクセスできます。`initialdata` が与えられた場合は、`data` はその内容で初期化されます; 他の目的のために使えるように、`initialdata` への参照が保存されないことがあるということに注意してください。

注釈: 後方互換性のために、`UserDict` のインスタンスはイテレート可能ではありません。

```
class UserDict.IterableUserDict([initialdata])
```

`UserDict` のイテレーションをサポートするサブクラス (使用例: `for key in myDict`).

マッピングのメソッドと演算 (マッピング型 — `dict` を参照) に加えて、`UserDict`、`IterableUserDict` インスタンスは次の属性を提供します:

`IterableUserDict.data`

`UserDict` クラスの内容を保存するために使われる実際の辞書です。

```
class UserDict.DictMixin
```

`__getitem__()`、`__setitem__()`、`__delitem__()` および `keys()` といった最小の辞書インタフェースを既に持っているクラスのために、全ての辞書メソッドを定義する mixin です。

この mixin はスーパークラスとして使われるべきです。上のそれぞれのメソッドを追加することで、より多くの機能がだんだん追加されます。例えば、`__delitem__()` 以外の全てのメソッドを定義すると、使えないのは `pop()` と `popitem()` だけになります。

4 つの基底メソッドに加えて、`__contains__()`、`__iter__()` および `iteritems()` を定義すれば、順次能力を増やして頂けます。

mixin はサブクラスのコンストラクタについて何も知らないので、`__init__()` や `copy()` は定義していません。

Python 2.6 からは、`DictMixin` の代わりに、`collections.MutableMapping` を利用することが推奨されています。

`DictMixin` は `viewkeys()`、`viewvalues()`、`viewitems()` メソッドを実装していないことに注意してください。

8.13 UserList — リストオブジェクトのためのクラスラッパー

注釈: Python 2.2 がリリースされた際、`list` から直接的にサブクラス化することがこのクラスの多くのユースケースを包含することとなりました。ですがわずかばかりのユースケースが健在です。

このモジュールは下層のデータストアを包むリストインターフェイスを提供します。デフォルトではデータストアは `list` ですが、(例えば永続化記憶のような) 他のオブジェクトに対してリストのようなインターフェイスで包むのに使うことが出来ます。

加えて、このクラスは多重継承を使って、組み込みクラスに `mixin` できます。これが役に立つときがあって、例えば `UserList` と `str` の両方を同時に継承できます。本物の `list` と本物の `str` ではこれはいけません。

このモジュールはリストオブジェクトのラッパーとして働くクラスを定義します。独自のリストに似たクラスのために役に立つ基底クラスで、これを継承し既存のメソッドをオーバーライドしたり、あるいは、新しいものを追加したりすることができます。このような方法で、リストに新しい振る舞いを追加できます。

`UserList` モジュールは `UserList` クラスを定義しています:

```
class UserList.UserList ([list])
```

リストをシミュレートするクラスです。インスタンスの内容は通常のリストに保存され、`UserList` インスタンスの `data` 属性を通してアクセスできます。インスタンスの内容は最初に `list` のコピーに設定されますが、デフォルトでは空リスト `[]` です。`list` は何らかのイテラブル、例えば通常の Python リストや `UserList` オブジェクトです。

注釈: `UserList` モジュールは、Python 3 では `collections` に移動しました。`2to3` ツールは、ソースコードの `import` を自動的に Python 3 向けに修正します。

可変シーケンスのメソッドと演算 (シーケンス型 — `str`, `unicode`, `list`, `tuple`, `bytearray`, `buffer`, `xrange` を参照) に加えて、`UserList` インスタンスは次の属性を提供します:

`UserList.data`

`UserList` クラスの内容を保存するために使われる実際の Python リストオブジェクト。

サブクラス化の要件: `UserList` のサブクラスは引数なしか、あるいは一つの引数のどちらかとともに呼び出せるコンストラクタを提供することが期待されています。新しいシーケンスを返すリスト演算は現在の実装クラスのインスタンスを作成しようとしています。そのために、データ元として使われるシーケンスオブジェクトである一つのパラメータとともにコンストラクタを呼び出せると想定しています。

派生クラスがこの要求に従いたくないならば、このクラスがサポートしているすべての特殊メソッドはオーバーライドされる必要があります。その場合に提供される必要のあるメソッドについての情報は、ソースを参考にしてください。

バージョン 2.0 で変更: Python バージョン 1.5.2 と 1.6 では、コンストラクタが引数なしで呼び出し可能であることと変更可能な `data` 属性を提供するということも要求されていました。Python の初期のバージョンでは、派生クラスのインスタンスを作成しようとはしません。

8.14 UserString — 文字列オブジェクトのためのクラスラッパー

注釈: このモジュールの `UserString` クラスは後方互換性のためだけに残されています。書いているコードが Python 2.2 より前のバージョンの Python で動作する必要があるのならば、`UserString` を使う代わりに

組み込み `str` 型から直接サブクラス化することを検討してください (組み込みの `MutableString` と等価なものはありません)。

このモジュールは文字列オブジェクトのラッパーとして働くクラスを定義します。独自の文字列に似たクラスのために役に立つ基底クラスで、これを継承し既存のメソッドをオーバーライドしたり、あるいは、新しいものを追加したりすることができます。このような方法で、文字列に新しい振る舞いを追加できます。

これらのクラスは本物の文字列や Unicode オブジェクトに比べてとてつもなく非効率であるということに注意すべきです。これは特に `MutableString` に対して当てはまります。

`UserString` モジュールは次のクラスを定義しています:

```
class UserString.UserString ([sequence])
```

文字列または Unicode 文字列オブジェクトをシミュレートするクラスです。インスタンスの内容は通常の文字列または Unicode 文字列オブジェクトに保存され、`UserString` インスタンスの `data` 属性を通してアクセスできます。インスタンスの内容は最初に `sequence` のコピーに設定されます。`sequence` は通常の Python 文字列または Unicode 文字列、`UserString` (またはサブクラス) のインスタンス、あるいは組み込み `str()` 関数を使って文字列に変換できる任意のシーケンスのいずれかです。

注釈: `UserString` クラスは、Python 3 では `collections` に移動しました。`2to3` ツールは、ソースコードの `import` を自動的に Python 3 向けに修正します。

```
class UserString.MutableString ([sequence])
```

このクラスは上の `UserString` から派生し、可変になるように文字列を再定義します。可変な文字列は辞書のキーとして使うことができません。なぜなら、辞書はキーとして不変なオブジェクトを要求するからです。このクラスの主な目的は、辞書のキーとして可変なオブジェクトを使うという試みを捕捉するために、継承と `__hash__()` メソッドを取り除く (オーバーライドする) 必要があることを示す教育的な例を提供することです。そうしなければ、非常にエラーになりやすく、その原因を突き止めることが困難でしょう。

バージョン 2.6 で非推奨: `MutableString` クラスは Python 3 で削除されました。

文字列および Unicode オブジェクトのメソッドと演算 (文字列メソッドを参照) に加えて、`UserString` インスタンスは次の属性を提供します:

`MutableString.data`

`UserString` クラスの内容を保存するために使われる実際の Python 文字列または Unicode オブジェクト。

8.15 types — 組み込み型の名前

ソースコード: `Lib/types.py`

このモジュールは標準の Python インタプリタで使われているオブジェクトの型について、名前を定義しています (拡張モジュールで定義されている型を除く)。また、このモジュールには `listiterator` 型のような処理中に例外が起きる型は含まれません。 `from types import *` のように使っても安全です – モジュールはここでリストされている以外の名前をエクスポートしません。このモジュールの将来のバージョンで追加される名前は、 `Type` で終わる予定です。

典型的な利用方法は、関数が以下のように引数の型によって異なる動作をする場合です:

```
from types import *
def delete(mylist, item):
    if type(item) is IntType:
        del mylist[item]
    else:
        mylist.remove(item)
```

Python 2.2 以降では、 `int()` や `str()` のようなファクトリ関数は、型の名前となりましたので、 `types` を使用する必要はなくなりました。上記のサンプルは、以下のように記述する事が推奨されています。

```
def delete(mylist, item):
    if isinstance(item, int):
        del mylist[item]
    else:
        mylist.remove(item)
```

このモジュールは以下の名前を定義しています。

`types.NoneType`

`None` の型です。

`types.TypeType`

`type` オブジェクトの型です (`type()` などによって返されます)。組み込みの `type` のエイリアスになります。

`types.BooleanType`

`bool` の `True` と `False` の型です。これは組み込みの `bool` のエイリアスです。

バージョン 2.3 で追加。

`types.IntType`

整数の型です (e.g. 1)。組み込みの `int` のエイリアスになります。

`types.LongType`

長整数の型です (e.g. 1L)。組み込みの `long` のエイリアスになります。

`types.FloatType`

浮動小数点数の型です (e.g. 1.0)。組み込みの `float` のエイリアスになります。

`types.ComplexType`

複素数の型です (e.g. 1.0j)。Python が複素数のサポートなしでコンパイルされていた場合には定義されません。

types.StringType

文字列の型です (e.g. 'Spam')。組み込みの *str* のエイリアスになります。

types.UnicodeType

Unicode 文字列の型です (e.g. u'Spam')。Python が Unicode のサポートなしでコンパイルされていた場合には定義されません。組み込みの Unicode のエイリアスになります。

types.TupleType

タプルの型です (e.g. (1, 2, 3, 'Spam'))。組み込みの *tuple* のエイリアスになります。

types.ListType

リストの型です (e.g. [0, 1, 2, 3])。組み込みの *list* のエイリアスになります。

types.DictType

辞書の型です (e.g. {'Bacon': 1, 'Ham': 0})。組み込みの *dict* のエイリアスになります。

types.DictionaryType

DictType の別名です。

types.FunctionType

types.LambdaType

ユーザー定義の関数または lambda 式によって作成された関数の型です。

types.GeneratorType

ジェネレータ (*generator*) 関数の呼び出しによって生成されたイテレータオブジェクトの型です。

バージョン 2.2 で追加。

types.CodeType

compile() 関数などによって返されるコードオブジェクトの型です。

types.ClassType

ユーザー定義の、旧形式クラスの型です。

types.InstanceType

ユーザー定義の旧形式クラスのインスタンスの型です。

types.MethodType

ユーザー定義のクラスのインスタンスのメソッドの型です。

types.UnboundMethodType

MethodType の別名です。

types.BuiltinFunctionType

types.BuiltinMethodType

len() や *sys.exit()* のような組み込み関数や、組み込み型のメソッドの型です。(ここでは、“組み込み”という単語を、“C で書かれた”という意味で使っています)

types.ModuleType

モジュールの型です。

types.FileType

`sys.stdout` のような open されたファイルオブジェクトの型です。組み込みの `file` のエイリアスになります。

types.XRangeType

`xrange()` 関数によって返される range オブジェクトの型です。組み込みの `xrange` のエイリアスになります。

types.SliceType

`slice()` 関数によって返されるオブジェクトの型です。組み込みの `slice` のエイリアスになります。

types.EllipsisType

Ellipsis の型です。

types.TracebackType

`sys.exc_traceback` に含まれるようなトレースバックオブジェクトの型です。

types.FrameType

フレームオブジェクトの型です。トレースバックオブジェクト `tb` の `tb.tb_frame` などです。

types.BufferType

`buffer()` 関数によって作られるバッファオブジェクトの型です。

types.DictProxyType

`TypeType.__dict__` のような dict へのプロキシ型です。

types.NotImplementedType

NotImplemented の型です。

types.GetSetDescriptorType

`FrameType.f_locals` や `array.array.typecode` のような、拡張モジュールにおいて `PyGetSetDef` によって定義されたオブジェクトの型です。この型はオブジェクト属性のディスクリプタとして利用されます。`property` 型と同じ目的を持った型ですが、こちらは拡張モジュールで定義された型のためのものです。

バージョン 2.5 で追加。

types.MemberDescriptorType

`datetime.timedelta.days` のような、拡張モジュールにおいて `PyMemberDef` によって定義されたオブジェクトの型です。この型は、標準の変換関数を利用するような、C のシンプルなデータメンバで利用されます。`property` 型と同じ目的を持った型ですが、こちらは拡張モジュールで定義された型のためのものです。

Python の他の実装では、この型は `GetSetDescriptorType` と同じかもしれません。

バージョン 2.5 で追加。

types.StringTypes

文字列型のチェックを簡単にするための `StringType` と `UnicodeType` を含むシーケンスです。`UnicodeType` は実行中の版の Python に含まれている場合にだけ含まれるので、2 つの文字列型

のシーケンスを使うよりこれを使う方が移植性が高くなります。例: `isinstance(s, types.StringTypes)`。

バージョン 2.2 で追加。

8.16 `new` — ランタイム内部オブジェクトの作成

バージョン 2.6 で非推奨: `new` モジュールは Python 3 で削除されました。代わりに、`types` モジュールのクラスを利用してください。

`new` モジュールはインタプリタオブジェクト作成関数へのインターフェイスを与えます。新しいオブジェクトを ”魔法を使ったように” 作り出す必要がある、通常の作成関数が使えないときに、これは主にマーシャル型関数で使われます。このモジュールはインタプリタへの低レベルインターフェイスを提供します。したがって、このモジュールを使うときには注意しなければなりません。オブジェクトが利用される時にインタプリタをクラッシュさせるような引数を与えることもできてしまいます。

`new` モジュールは次の関数を定義しています:

`new.instance(class[, dict])`

この関数は `__init__()` コンストラクタを呼び出さずに辞書 `dict` をもつ `class` のインスタンスを作り出します。 `dict` が省略されるか、`None` である場合は、新しいインスタンスのために新しい空の辞書が作られます。オブジェクトが一貫した状態であるという保証はないことに注意してください。

`new.instancemethod(function, instance, class)`

この関数は `instance` に束縛されたメソッドオブジェクトか、あるいは `instance` が `None` の場合に束縛されていないメソッドオブジェクトを返します。 `function` は呼び出し可能でなければなりません。

`new.function(code, globals[, name[, argdefs[, closure]]])`

与えられたコードとグローバル変数をもつ (Python) 関数を返します。 `name` を与えるならば、文字列か `None` でなければなりません。文字列の場合は、関数は与えられた名前をもちます。そうでなければ、関数名は `code.co_name` から取られます。 `argdefs` を与える場合はタプルでなければならず、パラメータのデフォルト値を決めるために使われます。 `closure` を与える場合は `None` または名前を `code.co_freevars` に束縛するセルオブジェクトのタプルである必要があります。

`new.code(argcount, nlocals, stacksize, flags, codestring, constants, names, varnames, filename, name, firstlineno, lnotab)`

この関数は `PyCode_New()` という C 関数へのインターフェイスです。

`new.module(name[, doc])`

この関数は `name` という名前の新しいモジュールオブジェクトを返します。 `name` は文字列でなければなりません。省略可能な `doc` 引数は任意の型を取ることができます。

`new.classobj(name, baseclasses, dict)`

この関数は新しいクラスオブジェクトを返します。そのクラスオブジェクトは (クラスのタプルであるべき) `baseclasses` から派生し、名前空間 `dict` を持ち、 `name` という名前です。

8.17 copy — 浅いコピーおよび深いコピー操作

Python において代入文はオブジェクトをコピーしません。代入はターゲットとオブジェクトの間に束縛を作ります。ミュータブルなコレクションまたはミュータブルなアイテムを含むコレクションについては、元のオブジェクトを変更せずにコピーを変更できるように、コピーが必要になることが時々あります。このモジュールは、汎用的な浅い (shallow) コピーと深い (deep) コピーの操作 (以下で説明されます) を提供します。

以下にインタフェースをまとめます:

`copy.copy(x)`

x の浅い (shallow) コピーを返します。

`copy.deepcopy(x)`

x の深い (deep) コピーを返します。

exception `copy.error`

モジュール特有のエラーを送出します。

浅い (shallow) コピーと深い (deep) コピーの違いが関係するのは、複合オブジェクト (リストやクラスインスタンスのような他のオブジェクトを含むオブジェクト) だけです:

- 浅いコピー (*shallow copy*) は新たな複合オブジェクトを作成し、その後 (可能な限り) 元のオブジェクト中に見つかったオブジェクトに対する 参照 を挿入します。
- 深いコピー (*deep copy*) は新たな複合オブジェクトを作成し、その後元のオブジェクト中に見つかったオブジェクトの コピー を挿入します。

深いコピー操作には、しばしば浅いコピー操作の時には存在しない 2 つの問題がついてまわります:

- 再帰的なオブジェクト (直接、間接に関わらず、自分自身に対する参照を持つ複合オブジェクト) は再帰ループを引き起こします。
- 深いコピーは何もかもコピーしてしまうため、例えば複数のコピー間で共有するつもりだったデータも余分にコピーしてしまいます。

`deepcopy()` 関数では、これらの問題を以下のようにして回避しています:

- 現在のコピー過程ですでにコピーされたオブジェクトからなる、“メモ” 辞書を保持します; かつ
- ユーザ定義のクラスでコピー操作やコピーされる内容の集合を上書きできるようにします。

このモジュールでは、モジュール、メソッド、スタックトレース、スタックフレーム、ファイル、ソケット、ウィンドウ、アレイ、その他これらに類似の型をコピーしません。このモジュールでは元のオブジェクトを変更せずに返すことで関数とクラスを (浅くまたは深く) 「コピー」します。これは `pickle` モジュールでの扱われかたと同じです。

辞書型の浅いコピーは `dict.copy()` で、リストの浅いコピーはリスト全体を指すスライス (例えば `copied_list = original_list[:]`) でできます。

バージョン 2.5 で変更: 関数コピーの追加。

クラスでは、pickle 化を制御するためのインタフェースと同じインタフェースをコピーの制御に使うことができます。これらのメソッドに関する情報は `pickle` モジュールの記述を参照してください。 `copy` モジュールは pickle 用関数登録モジュール `copy_reg` を使いません。

クラス独自のコピー実装を定義するために、特殊メソッド `__copy__()` および `__deepcopy__()` を定義することができます。前者は浅いコピー操作を実装するために使われます; 追加の引数はありません。後者は深いコピー操作を実現するために呼び出されます; この関数には単一の引数としてメモ辞書が渡されます。 `__deepcopy__()` の実装で、内容のオブジェクトに対して深いコピーを生成する必要がある場合、 `deepcopy()` を呼び出し、最初の引数にそのオブジェクトを、メモ辞書を二つ目の引数に与えなければなりません。

参考:

`pickle` モジュール オブジェクト状態の取得と復元をサポートするために使われる特殊メソッドについて議論されています。

8.18 pprint — データ出力の整然化

ソースコード: [Lib/pprint.py](#)

`pprint` モジュールを使うと、Python の任意のデータ構造をインタープリタへの入力で使われる形式にして”pretty-print”できます。フォーマット化された構造の中に Python の基本的なタイプではないオブジェクトがあるなら、表示できないかもしれません。Python の定数として表現できない多くの組み込みオブジェクトと同様、ファイル、ソケット、クラスあるいはインスタンスのようなオブジェクトが含まれていた場合は出力できません。

可能であればオブジェクトをフォーマット化して 1 行に出力しますが、与えられた幅に合わないなら複数行に分けて出力します。無理に幅を設定したいなら、 `PrettyPrinter` オブジェクトを作成して明示してください。

バージョン 2.5 で変更: 辞書は出力を計算する前にキーでソートされます。2.5 以前では、辞書は 1 行以上の出力が必要な場合にのみソートされていましたがドキュメントには書かれていませんでした。

バージョン 2.6 で変更: `set` と `frozenset` がサポートされました。

`pprint` モジュールには 1 つのクラスが定義されています:

```
class pprint.PrettyPrinter(indent=1, width=80, depth=None, stream=None)
```

`PrettyPrinter` インスタンスを作ります。このコンストラクタにはいくつかのキーワードパラメータを設定できます。 `stream` キーワードで出力ストリームを設定できます; このストリームに対して呼び出されるメソッドはファイルプロトコルの `write()` メソッドだけです。もし設定されなければ、 `PrettyPrinter` は `sys.stdout` を使用します。さらに 3 つのパラメータで出力フォーマットをコントロールできます。そのキーワードは `indent`、 `depth` と `width` です。再帰的なレベルごとに加えるインデントの量は `indent` で設定できます; デフォルト値は 1 です。他の値にすると出力が少しおかしく見えますが、ネスト化されたところが見分け易くなります。出力されるレベルは `depth` で設定できます; 出力されるデータ構造が深いなら、指定以上の深いレベルのものは ... で置き換えられて表示されま

す。デフォルトでは、オブジェクトの深さを制限しません。 *width* パラメータを使うと、出力する幅を望みの文字数に設定できます; デフォルトでは 80 文字です。もし指定した幅にフォーマットできない場合は、できるだけ近づけます。

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[  ['spam', 'eggs', 'lumberjack', 'knights', 'ni'],
   'spam',
   'eggs',
   'lumberjack',
   'knights',
   'ni']
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
... ('parrot', ('fresh fruit',))))))))
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead', (...)))))))
```

PrettyPrinter クラスにはいくつかの派生する関数が提供されています:

`pprint.pformat(object, indent=1, width=80, depth=None)`

object をフォーマット化して文字列として返します。 *indent*、 *width* と、 *depth* は *PrettyPrinter* コンストラクタにフォーマット指定引数として渡されます。

バージョン 2.4 で変更: 引数 *indent*、 *width* と、 *depth* が追加されました。

`pprint.pprint(object, stream=None, indent=1, width=80, depth=None)`

stream 上に *object* のフォーマットされた表現を出力して、その後に newline を続けます。 *stream* が None の場合、 `sys.stdout` が使用されます。これは、対話型インタプリタの中で値を調査するために `print` 文の代わりに使用されることがあります。 *indent*, *width*, *depth* は、フォーマットパラメータとして *PrettyPrinter* コンストラクタに渡されます。

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
[<Recursion on list with id=...>,
  'spam',
  'eggs',
  'lumberjack',
  'knights',
  'ni']
```

バージョン 2.4 で変更: 引数 *indent*、 *width* と、 *depth* が追加されました。

`pprint.isreadable(object)`

object をフォーマット化して出力できる ("readable") か、あるいは `eval()` を使って値を再構成できるかを返します。再帰的なオブジェクトに対しては常に `False` を返します。

```
>>> pprint.isreadable(stuff)
False
```

`pprint.isrecursive(object)`

object が再帰的な表現かどうかを返します。

さらにもう 1 つ、関数が定義されています:

`pprint.saferepr(object)`

object の文字列表現を、再帰的なデータ構造から保護した形式で返します。もし *object* の文字列表現が再帰的な要素を持っているなら、再帰的な参照は `<Recursion on typename with id=number>` で表示されます。出力は他と違ってフォーマット化されません。

```
>>> pprint.saferepr(stuff)
" [<Recursion on list with id=...>, 'spam', 'eggs', 'lumberjack', 'knights',
  ↪ 'ni'] "
```

8.18.1 PrettyPrinter オブジェクト

PrettyPrinter インスタンスには以下のメソッドがあります:

`PrettyPrinter.pformat(object)`

object のフォーマット化した表現を返します。これは *PrettyPrinter* のコンストラクタに渡されたオプションを考慮してフォーマット化されます。

`PrettyPrinter.pprint(object)`

object のフォーマット化した表現を指定したストリームに出力し、最後に改行します。

以下のメソッドは、対応する同じ名前の関数と同じ機能を持っています。以下のメソッドをインスタンスに対して使うと、新たに *PrettyPrinter* オブジェクトを作る必要がないのでちょっぴり効果的です。

`PrettyPrinter.isreadable(object)`

object をフォーマット化して出力できる ("readable") か、あるいは `eval()` を使って値を再構成できるかを返します。これは再帰的なオブジェクトに対して `False` を返すことに注意して下さい。もし *PrettyPrinter* の `depth` パラメータが設定されていて、オブジェクトのレベルが設定よりも深かったら、`False` を返します。

`PrettyPrinter.isrecursive(object)`

オブジェクトが再帰的な表現かどうかを返します。

このメソッドをフックとして、サブクラスがオブジェクトを文字列に変換する方法を修正するのが可能になっています。デフォルトの実装では、内部で `saferepr()` を呼び出しています。

`PrettyPrinter.format(object, context, maxlevels, level)`

次の 3 つの値を返します。*object* をフォーマット化して文字列にしたもの、その結果が読み込み可能かどうかを示すフラグ、再帰が含まれているかどうかを示すフラグ。最初の引数は表示するオブジェクトです。2 つめの引数はオブジェクトの `id()` をキーとして含むディクショナリで、オブジェクトを含んでいる現在の (直接、間接に *object* のコンテナとして表示に影響を与える) 環境です。ディクショナリ

`context` の中でどのオブジェクトが表示されたか表示する必要があるなら、3 つめの返り値は `True` になります。 `format()` メソッドの再帰呼び出しではこのディクショナリのコンテナに対してさらにエントリを加えます。3 つめの引数 `maxlevels` で再帰呼び出しのレベルを制限します。制限しない場合、0 になります。この引数は再帰呼び出しでそのまま渡されます。4 つめの引数 `level` で現在のレベルを設定します。再帰呼び出しでは、現在の呼び出しより小さい値が渡されます。

バージョン 2.3 で追加。

8.18.2 pprint の例

この例は `pprint()` 関数とその引数の幾つかの使い方を例示しています。

```
>>> import pprint
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
... ('parrot', ('fresh fruit',)))))))
>>> stuff = ['a' * 10, tup, ['a' * 30, 'b' * 30], ['c' * 20, 'd' * 20]]
>>> pprint.pprint(stuff)
['aaaaaaaaaa',
 ('spam',
 ('eggs',
 ('lumberjack',
 ('knights', ('ni', ('dead', ('parrot', ('fresh fruit',))))))),
 ['aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa', 'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb'],
 ['cccccccccccccccccccc', 'dddddddddddddddddddd']]
>>> pprint.pprint(stuff, depth=3)
['aaaaaaaaaa',
 ('spam', ('eggs', (...))),
 ['aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa', 'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb'],
 ['cccccccccccccccccccc', 'dddddddddddddddddddd']]
>>> pprint.pprint(stuff, width=60)
['aaaaaaaaaa',
 ('spam',
 ('eggs',
 ('lumberjack',
 ('knights',
 ('ni', ('dead', ('parrot', ('fresh fruit',))))))),
 ['aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa',
 'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb'],
 ['cccccccccccccccccccc', 'dddddddddddddddddddd']]
```

8.19 repr — もう一つの repr() の実装

注釈: `repr` モジュールは、Python 3 では `reprlib` にリネームされました。 `2to3` ツールが自動的にソースコードの `import` を修正します。

ソースコード: `Lib/repr.py`

`repr` モジュールは結果の文字列の大きさを制限したオブジェクト表現を作り出すための方法を提供します。これは Python デバッガで使われていますが、他の状況でも同じように役に立つかもしれません。

このモジュールはクラスとインスタンス、それに関数を提供します:

class `repr.Repr`

組み込みクラス `repr()` によく似た関数を実装するために役に立つ書式化サービスを提供します。過度に長い表現を作り出さないように、異なるオブジェクト型に対する大きさの制限が追加されます。

`repr.aRepr`

これは下で説明される `repr()` 関数を提供するために使われる `Repr` のインスタンスです。このオブジェクトの属性を変更すると、`repr()` と Python デバッガが使うサイズ制限に影響します。

`repr.repr(obj)`

これは `aRepr` の `repr()` メソッドです。同じ名前の組み込み関数が返す文字列と似ていますが、最大サイズに制限のある文字列を返します。

8.19.1 Repr オブジェクト

`Repr` インスタンスは様々なオブジェクト型の表現にサイズ制限を与えるために使えるいくつかの属性と、特定のオブジェクト型を書式化するメソッドを提供します。

`Repr.maxlevel`

再帰的な表現を作る場合の深さ制限。デフォルトは 6 です。

`Repr.maxdict`

`Repr.maxlist`

`Repr.maxtuple`

`Repr.maxset`

`Repr.maxfrozenset`

`Repr.maxdeque`

`Repr.maxarray`

指定されたオブジェクト型に対するエントリ表現の数についての制限。 `maxdict` に対するデフォルトは 4 で、 `maxarray` は 5、その他に対しては 6 です。

バージョン 2.4 で追加: `maxset`, `maxfrozenset`, `set` の追加。

`Repr.maxlong`

長整数の表現における文字数の最大値。中央の数字が抜け落ちます。デフォルトは 40 です。

`Repr.maxstring`

文字列の表現における文字数の制限。文字列の”通常の”表現は文字の材料だということに注意してください: 表現にエスケープシーケンスが必要とされる場合は、表現が短縮されたときにこれらはマングルされます。デフォルトは 30 です。

Repr.maxother

この制限は *Repr* オブジェクトに利用できる特定の書式化メソッドがないオブジェクト型のサイズをコントロールするために使われます。 *maxstring* と同じようなやり方で適用されます。デフォルトは 20 です。

Repr.repr(obj)

インスタンスが強制する書式化を使う組み込み *repr()* と等価なもの。

Repr.repr1(obj, level)

repr() が使う再帰的な実装。これはどの書式化メソッドを呼び出すかを決定するために *obj* の型を使い、それを *obj* と *level* に渡します。再帰呼び出しにおいて *level* の値に対して *level - 1* を与える再帰的な書式化を実行するために、型に固有のメソッドは *repr1()* を呼び出します。

Repr.repr_TYPE(obj, level)

型名に基づく名前をもつメソッドとして、特定の型に対する書式化メソッドは実装されます。メソッド名では、**TYPE** は `string.join(string.split(type(obj).__name__, '_'), '_')` に置き換えられます。これらのメソッドへのディスパッチは *repr1()* によって処理されます。再帰的に値の書式を整える必要がある型固有のメソッドは、`self.repr1(subobj, level - 1)` を呼び出します。

8.19.2 Repr オブジェクトをサブクラス化する

更なる組み込みオブジェクト型へのサポートを追加するためや、すでにサポートされている型の扱いを変更するために、*Repr.repr1()* による動的なディスパッチを使って *Repr* をサブクラス化することができます。この例はファイルオブジェクトのための特別なサポートを追加する方法を示しています：

```
import repr as reprlib
import sys

class MyRepr(reprlib.Repr):
    def repr_file(self, obj, level):
        if obj.name in ['<stdin>', '<stdout>', '<stderr>']:
            return obj.name
        else:
            return repr(obj)

aRepr = MyRepr()
print aRepr.repr(sys.stdin)           # prints '<stdin>'
```


第 9 章

数値と数学モジュール

この章で解説されるモジュールは数値と数学関連の関数とデータ型を提供します。 `numbers` モジュールは数値型の抽象的階層を定義します。 `math` と `cmath` はさまざまな浮動小数点数および複素数向け数学関数を含みます。速度より 10 進法での正確さに興味があるユーザには、 `decimal` モジュールが真の 10 進表現をサポートしています。

この章では以下のモジュールが記述されています:

9.1 `numbers` — 数の抽象基底クラス

バージョン 2.6 で追加.

`numbers` モジュール ([PEP 3141](#)) は数の [抽象基底クラス](#) の、順により多くの演算を定義していく階層を定義します。このモジュールで定義される型はどれもインスタンス化できません。

class `numbers.Number`

数の階層の根。引数 `x` が、種類は何であれ、数であるということだけチェックしたい場合、`isinstance(x, Number)` が使えます。

9.1.1 数値塔

class `numbers.Complex`

この型のサブクラスは複素数を表し、組み込みの `complex` 型を受け付ける演算を含みます。それらは: `complex` および `bool` への変換、`real`, `imag`, `+`, `-`, `*`, `/`, `abs()`, `conjugate()`, `==`, `!=` です。 `-` と `!=` 以外の全てのものは抽象メソッドや抽象プロパティです。

real

抽象プロパティ。この数の実部を取り出します。

imag

抽象プロパティ。この数の虚部を取り出します。

conjugate()

抽象プロパティ。複素共役を返します。たとえば、`(1+3j).conjugate() == (1-3j)` です。

class numbers.Real

Complex の上に、*Real* は実数で意味を成す演算を加えます。

簡潔に言うとそれらは: *float* への変換, *math.trunc()*, *round()*, *math.floor()*, *math.ceil()*, *divmod()*, *//*, *%*, *<*, *<=*, *>* および *>=* です。

Real はまた *complex()*, *real*, *imag* および *conjugate()* のデフォルトを提供します。

class numbers.Rational

Real をサブタイプ化し *numerator* と *denominator* のプロパティを加えたものです。これらは既約分数のものでなければなりません。この他に *float()* のデフォルトも提供します。

numerator

抽象プロパティ。

denominator

抽象プロパティ。

class numbers.Integral

Rational をサブタイプ化し *int* への変換が加わります。 *float()*, *numerator*, *denominator* のデフォルトを提供します。 **** に対する抽象メソッドと、ビット列演算 *<<*, *>>*, *&*, *^*, *|*, *~* を追加します。

9.1.2 型実装者のための注意事項

実装する人は等しい数が等しく扱われるように同じハッシュを与えるように気を付けねばなりません。これは二つの異なった実数の拡張があるような場合にはややこしいことになるかもしれません。たとえば、*fractions.Fraction* は *hash()* を以下のように実装しています:

```
def __hash__(self):
    if self.denominator == 1:
        # Get integers right.
        return hash(self.numerator)
    # Expensive check, but definitely correct.
    if self == float(self):
        return hash(float(self))
    else:
        # Use tuple's hash to avoid a high collision rate on
        # simple fractions.
        return hash((self.numerator, self.denominator))
```

さらに数の ABC を追加する

もちろん、他にも数に対する ABC が有り得ますし、そういったものを付け加える可能性を閉ざしてしまうとすれば貧相な階層でしかありません。たとえば *MyFoo* を *Complex* と *Real* の間に付け加えるには:

```
class MyFoo(Complex): ...
MyFoo.register(Real)
```

算術演算の実装

私たちは、混在型 (mixed-mode) 演算について作者が両方の引数の型について知っているような実装を呼び出すか、両方を最も近い組み込み型に変換してそこで演算するか、どちらかを行うように算術演算を実装したいのです。 `Integral` のサブタイプに対して、このことは `__add__()` と `__radd__()` が次のように定義されるべきであることを意味します:

```
class MyIntegral(Integral):

    def __add__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(self, other)
        else:
            return NotImplemented

    def __radd__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(other, self)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(other, self)
        elif isinstance(other, Integral):
            return int(other) + int(self)
        elif isinstance(other, Real):
            return float(other) + float(self)
        elif isinstance(other, Complex):
            return complex(other) + complex(self)
        else:
            return NotImplemented
```

ここには 5 つの異なる `Complex` のサブクラス間の混在型の演算があります。上のコードの中で `MyIntegral` と `OtherTypeIKnowAbout` に触れない部分を "ボイラープレート" と呼ぶことにしましょう。 `a` を `Complex` のサブタイプである `A` のインスタンス (`a : A <: Complex`)、同様に `b : B <: Complex` として、`a + b` を考えます:

1. `A` が `b` を受け付ける `__add__()` を定義している場合、何も問題はありません。
2. `A` でボイラープレート部分に落ち込み、その結果 `__add__()` が値を返すならば、`B` に良く考えられた `__radd__()` が定義されている可能性を見逃してしまいますので、ボイラープレートは `__add__()` から `NotImplemented` を返すのが良いでしょう。(若しくは、`A` はまったく `__add__()` を実装すべきではなかったかもしれません。)
3. そうすると、`B` の `__radd__()` にチャンスが巡ってきます。ここで `a` が受け付けられるならば、結果は上々です。
4. ここでボイラープレートに落ち込むならば、もう他に試すべきメソッドはありませんので、デフォルト実装の出番です。
5. もし `B <: A` ならば、Python は `A.__add__` の前に `B.__radd__` を試します。これで良い理由は、`A` についての知識を持って実装しており、`Complex` に委ねる前にこれらのインスタンスを扱えるはずだ

からです。

もし `A <: Complex` かつ `B <: Real` で他に共有された知識が無いならば、適切な共通の演算は組み込みの `complex` を使ったものになり、どちらの `__radd__()` とともに着地するでしょうから、`a+b == b+a` です。

ほとんどの演算はどのような型についても非常に良く似ていますので、与えられた演算子について順結合 (forward) および逆結合 (reverse) のメソッドを生成する支援関数を定義することは役に立ちます。たとえば、`fractions.Fraction` では次のようなものを利用しています:

```
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, long, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
        else:
            return NotImplemented
    forward.__name__ = '__' + fallback_operator.__name__ + '__'
    forward.__doc__ = monomorphic_operator.__doc__

    def reverse(b, a):
        if isinstance(a, Rational):
            # Includes ints.
            return monomorphic_operator(a, b)
        elif isinstance(a, numbers.Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, numbers.Complex):
            return fallback_operator(complex(a), complex(b))
        else:
            return NotImplemented
    reverse.__name__ = '__r' + fallback_operator.__name__ + '__'
    reverse.__doc__ = monomorphic_operator.__doc__

    return forward, reverse

def _add(a, b):
    """a + b"""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(_add, operator.add)

# ...
```

9.2 math — 数学関数

このモジュールはいつでも利用できます。標準 C で定義されている数学関数にアクセスすることができます。

これらの関数で複素数を使うことはできません。複素数に対応する必要があるならば、`cmath` モジュールにある同じ名前の関数を使ってください。ほとんどのユーザーは複素数を理解するのに必要なだけの数学を勉強したくないので、複素数に対応した関数と対応していない関数の区別がされています。これらの関数では複素数が利用できないため、引数に複素数を渡されると、複素数の結果が返るのではなく例外が発生します。その結果、こういった理由で例外が送出されたかに早い段階で気づく事ができます。

このモジュールでは次の関数を提供しています。明示的な注記のない限り、戻り値は全て浮動小数点数になります。

9.2.1 数論および数表現にまつわる関数です

`math.ceil(x)`

x の天井値 (ceil)、すなわち x 以上の最も小さい整数を float 型で返します。

`math.copysign(x, y)`

x に y の符号を付けて返します。符号付きのゼロをサポートしているプラットフォームでは、`copysign(1.0, -0.0)` は `-1.0` を返します。

バージョン 2.6 で追加。

`math.fabs(x)`

x の絶対値を返します。

`math.factorial(x)`

x の階乗を返します。 x が整数値でなかったり負であったりするときは、`ValueError` を送出します。

バージョン 2.6 で追加。

`math.floor(x)`

x の床値 (floor)、すなわち x 以下の最も大きい整数を float 型で返します。

`math.fmod(x, y)`

プラットフォームの C ライブラリで定義されている `fmod(x, y)` を返します。Python の `x % y` という式は必ずしも同じ結果を返さないということに注意してください。C 標準の要求では、`fmod()` は除算の結果が x と同じ符号になり、大きさが `abs(y)` より小さくなるような整数 n については `fmod(x, y)` が厳密に (数学的に、つまり限りなく高い精度で) $x - n*y$ と等価であるよう求めています。Python の `x % y` は、 y と同じ符号の結果を返し、浮動小数点の引数に対して厳密な解を出せないことがあります。例えば、`fmod(-1e-100, 1e100)` は `-1e-100` ですが、Python の `-1e-100 % 1e100` は `1e100-1e-100` になり、浮動小数点型で厳密に表現できず、ややこしいことに `1e100` に丸められます。このため、一般には浮動小数点の場合には関数 `fmod()`、整数の場合には `x % y` を使う方がよいでしょう。

`math.frexp(x)`

x の仮数と指数を (m, e) のペアとして返します。 m は float 型で、 e は厳密に `x == m * 2**e` であ

るような整数型です。 x がゼロの場合は、 $(0.0, 0)$ を返し、それ以外の場合は、 $0.5 \leq \text{abs}(m) < 1$ を返します。これは浮動小数点型の内部表現を可搬性を保ったまま ”分解 (pick apart)” するためです。

`math.fsum(iterable)`

`iterable` 中の値の浮動小数点数の正確な和を返します。複数の部分和を追跡することで桁落ちを防ぎます:

```
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>> fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
1.0
```

アルゴリズムの正確性は IEEE-754 演算の保証と丸めモードが偶数丸め (half-even) である典型的な場合に依存します。Windows 以外の幾つかのビルドでは、依存する C ライブラリが、拡張精度の加算と時々時々合計の中間値を `double` 型へ丸めを行ってしまい、最下位ビットの消失が発生します。

より詳細な議論と代替となる二つのアプローチについては、[ASPN cookbook recipes for accurate floating point summation](#) をご覧下さい。

バージョン 2.6 で追加.

`math.isinf(x)`

浮動小数点数 x が正または負の無限大であるかチェックします。

バージョン 2.6 で追加.

`math.isnan(x)`

浮動小数点数 x が NaN (not a number) であるかチェックします。NaN についての詳しい情報は、IEEE 754 標準を参照してください。

バージョン 2.6 で追加.

`math.ldexp(x, i)`

$x * (2^{**i})$ を返します。これは本質的に `frexp()` の逆関数です。

`math.modf(x)`

x の小数部分と整数部分を返します。両方の結果は x の符号を受け継ぎます。整数部は `float` 型で返されます。

`math.trunc(x)`

x の *Integral* 値 (たいてい長整数) へ切り捨てられた *Real* 値を返します。`x.__trunc__()` メソッドに処理を委譲します。

バージョン 2.6 で追加.

`frexp()` と `modf()` は C のものとは異なった呼び出し/返しパターンを持っていることに注意してください。引数を 1 つだけ受け取り、1 組のペアになった値を返すので、2 つ目の戻り値を '出力用の引数' 経由で返したりはしません (Python には出力用の引数はありません)。

`ceil()`、`floor()`、および `modf()` 関数については、非常に大きな浮動小数点数が全て 整数そのもの

になるということに注意してください。通常、Python の浮動小数点型は 53 ビット以上の精度をもたない (プラットフォームにおける C double 型と同じ) ので、結果的に $\text{abs}(x) \geq 2^{52}$ であるような浮動小数点型 x は小数部分を持たなくなるのです。

9.2.2 指数関数と対数関数

`math.exp(x)`

e^x を返します。

`math.expm1(x)`

$e^x - 1$ を返します。小さな浮動小数点数 x について $\text{exp}(x) - 1$ を計算すると、引き算により桁落ちする可能性があります。この `expm1()` 関数は、完全な精度でこの値を計算します:

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1 # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5) # result accurate to full precision
1.0000050000166668e-05
```

バージョン 2.7 で追加.

`math.log(x[, base])`

引数が 1 つの場合、 x の (e を底とする) 自然対数を返します。

引数が 2 つの場合、 $\log(x) / \log(\text{base})$ として求められる base を底とした x の対数を返します。

バージョン 2.3 で変更: 引数 base を追加

`math.log1p(x)`

$1+x$ の自然対数 (つまり底 e の対数) を返します。結果はゼロに近い x に対して正確になるような方法で計算されます。

バージョン 2.6 で追加.

`math.log10(x)`

x の 10 を底とした対数 (常用対数) を返します。この関数は通常、 $\log(x, 10)$ よりも高精度です。

`math.pow(x, y)`

x の y 乗を返します。例外的な場合については、C99 標準の付録 'F' に可能な限り従います。特に、`pow(1.0, x)` と `pow(x, 0.0)` は、たとえ x が零や NaN でも、常に 1.0 を返します。もし x と y の両方が有限の値で、 x が負、 y が整数でない場合、`pow(x, y)` は未定義で、`ValueError` を送出します。

組み込みの `**` 演算子と違って、`math.pow()` は両方の引数を `float` 型に変換します。正確な整数の冪乗を計算するには `**` もしくは組み込みの `pow()` 関数を使ってください。

バージョン 2.6 で変更: 以前は `1**nan` や `nan**0` の結果は未定義でした。

`math.sqrt(x)`

x の平方根を返します。

9.2.3 三角関数

`math.acos(x)`

x の逆余弦を、ラジアンで返します。

`math.asin(x)`

x の逆正弦を、ラジアンで返します。

`math.atan(x)`

x の逆正接を、ラジアンで返します。

`math.atan2(y, x)`

`atan(y / x)` を、ラジアンで返します。戻り値は $-\pi$ から π の間になります。この角度は、極座標平面において原点から (x, y) へのベクトルが X 軸の正の方向となす角です。`atan2()` のポイントは、両方の入力の符号が既知であるために、位相角の正しい象限を計算できることにあります。例えば、`atan(1)` と `atan2(1, 1)` はいずれも $\pi/4$ ですが、`atan2(-1, -1)` は $-3\pi/4$ になります。

`math.cos(x)`

x ラジアンの余弦を返します。

`math.hypot(x, y)`

ユークリッドノルム ($\sqrt{x*x + y*y}$) を返します。これは原点から点 (x, y) のベクトルの長さです。

`math.sin(x)`

x ラジアンの正弦を返します。

`math.tan(x)`

x ラジアンの正接を返します。

9.2.4 角度変換

`math.degrees(x)`

角 x をラジアンから度に変換します。

`math.radians(x)`

角 x を度からラジアンに変換します。

9.2.5 双曲線関数

`math.acosh(x)`

x の逆双曲線余弦を返します。

バージョン 2.6 で追加.

`math.asinh(x)`

x の逆双曲線正弦を返します。

バージョン 2.6 で追加.

`math.atanh(x)`

x の逆双曲線正接を返します。

バージョン 2.6 で追加.

`math.cosh(x)`

x の双曲線余弦を返します。

`math.sinh(x)`

x の双曲線正弦を返します。

`math.tanh(x)`

x の双曲線正接を返します。

9.2.6 特殊な関数

`math.erf(x)`

x の誤差関数を返します。

バージョン 2.7 で追加.

`math.erfc(x)`

x の相補誤差関数を返します。

バージョン 2.7 で追加.

`math.gamma(x)`

x のガンマ関数を返します。

バージョン 2.7 で追加.

`math.lgamma(x)`

x のガンマ関数の絶対値の自然対数を返します。

バージョン 2.7 で追加.

9.2.7 定数

`math.pi`

利用可能な精度の、数学定数 $\pi = 3.141592\dots$ (円周率)。

`math.e`

利用可能な精度の、数学定数 $e = 2.718281\dots$ (自然対数の底)。

`math` モジュールは、ほとんどが実行プラットフォームにおける C 言語の数学ライブラリ関数に対する薄いラップでできています。例外的な場合での挙動は、適切である限り C99 標準の Annex F に従います。現在の実装では、(C99 Annex F で不正な演算やゼロ除算を通知することが推奨されている) `sqrt(-1.0)` や `log(0.0)` といった不正な操作に対して `ValueError` を発生させ、(例えば `exp(1000.0)` のような) 演算結果がオーバーフローする場合には `OverflowError` を発生させます。上記の関数群は、1 つ以上の引数が NaN であった場合を除いて、NaN を返しません。引数に NaN が与えられた場合は、殆どの関数は NaN を返しますが、(C99 Annex F に従って) 別の動作をする場合があります。例えば、`pow(float('nan'), 0.0)` や `hypot(float('nan'), float('inf'))` といった場合です。訳注: 例外が発生せずに結果が返ると、計算結果がおかしくなった原因が複素数を渡したためだということに気づくのが遅れる可能性があります。

Python は signaling NaN と quiet NaN を区別せず、signaling NaN に対する挙動は未定義とされていることに注意してください。典型的な挙動は、全ての NaN を quiet NaN として扱うことです。

バージョン 2.6 で変更: 特別なケースにおける挙動は、C99 Annex F に従おうとするようになりました。以前のバージョンの Python では、特別なケースでの挙動は曖昧にしか定義されていませんでした。

参考:

`cmath` モジュール これらの多くの関数の複素数版。

9.3 cmath — 複素数のための数学関数

このモジュールは常に利用できます。提供しているのは複素数を扱う数学関数へのアクセス手段です。このモジュール中の関数は整数、浮動小数点数または複素数を引数にとります。また、`--complex--()` または `--float--()` どちらかのメソッドを提供している Python オブジェクトも受け付けます。これらのメソッドはそのオブジェクトを複素数または浮動小数点数に変換するのにそれぞれ使われ、呼び出された関数はそうして変換された結果を利用します。

注釈: ハードウェア及びシステムレベルでの符号付きゼロのサポートがあるプラットフォームでは、分枝切断 (branch cut) の関わる関数において切断された 両側 の分枝で連続になります。ゼロの符号でどちらの分枝であるかを区別するのはです。符号付きゼロがサポートされないプラットフォームでは連続性は以下の仕様で述べるようになります。

9.3.1 極座標変換

Python の複素数 z は内部的には 直交座標 もしくは デカルト座標 と呼ばれる座標を使って格納されています。この座標はその複素数の 実部 `z.real` と 虚部 `z.imag` で決まります。言い換えると:

```
z == z.real + z.imag*1j
```

極座標 は複素数を表現する別の方法です。極座標では、複素数 z は半径 r と位相角 ϕ で定義されます。半径 r は z から原点までの距離です。位相 ϕ は x 軸の正の部分から原点と z を結んだ線分までの角度を反時計回

りにラジアンで測った値です。

次の関数はネイティブの直交座標を極座標に変換したりその逆を行うのに使えます。

`cmath.phase(x)`

x の位相 (x の 偏角 と呼びます) を浮動小数点数で返します。`phase(x)` は `math.atan2(x.imag, x.real)` と同等です。返り値は $[-\pi, \pi]$ の範囲にあり、この演算の分枝切断は負の実軸に沿って延びていて、上から連続です。(現在のほとんどのシステムはそうですが) 符号付きゼロをサポートしているシステムでは、結果の符号は `x.imag` がゼロであってさえ `x.imag` の符号と等しくなります:

```
>>> phase(complex(-1.0, 0.0))
3.1415926535897931
>>> phase(complex(-1.0, -0.0))
-3.1415926535897931
```

バージョン 2.6 で追加.

注釈: 複素数 x のモジュラス (絶対値) は組み込みの `abs()` 関数で計算できます。この演算を行う `cmath` モジュールの関数はありません。

`cmath.polar(x)`

x の極座標表現を返します。 x の半径 r と x の位相 ϕ の組 (r, ϕ) を返します。`polar(x)` は $(\text{abs}(x), \text{phase}(x))$ に等しいです。

バージョン 2.6 で追加.

`cmath.rect(r, phi)`

極座標 r, ϕ を持つ複素数 x を返します。値は $r * (\text{math.cos}(\phi) + \text{math.sin}(\phi)*1j)$ に等しいです。

バージョン 2.6 で追加.

9.3.2 指数関数と対数関数

`cmath.exp(x)`

指数 e^{**x} を返します。

`cmath.log(x[, base])`

`base` を底とする x の対数を返します。もし `base` が指定されていない場合には、 x の自然対数を返します。分枝切断を一つもち、0 から負の実数軸に沿って $-\infty$ へと延びており、上から連続しています。

バージョン 2.4 で変更: 引数 `base` を追加

`cmath.log10(x)`

x の底を 10 とする対数を返します。`log()` と同じ分枝切断を持ちます。

`cmath.sqrt(x)`

x の平方根を返します。 `log()` と同じ分枝切断を持ちます。

9.3.3 三角関数

`cmath.acos(x)`

x の逆余弦を返します。この関数には二つの分枝切断 (branch cut) があります: 一つは 1 から右側に実数軸に沿って ∞ へと延びていて、下から連続しています。もう一つは -1 から左側に実数軸に沿って $-\infty$ へと延びていて、上から連続しています。

`cmath.asin(x)`

x の逆正弦を返します。 `acos()` と同じ分枝切断を持ちます。

`cmath.atan(x)`

x の逆正接を返します。二つの分枝切断があります: 一つは $1j$ から虚数軸に沿って ∞j へと延びており、右から連続です。もう一つは $-1j$ から虚数軸に沿って $-\infty j$ へと延びており、左から連続です。

バージョン 2.6 で変更: 上側の分割での連続な方向が逆転しました。

`cmath.cos(x)`

x の余弦を返します。

`cmath.sin(x)`

x の正弦を返します。

`cmath.tan(x)`

x の正接を返します。

9.3.4 双曲線関数

`cmath.acosh(x)`

x の逆双曲線余弦を返します。分枝切断が一つあり、1 の左側に実数軸に沿って $-\infty$ へと延びていて、上から連続しています。

`cmath.asinh(x)`

x の逆双曲線正弦を返します。二つの分枝切断があります: 一つは $1j$ から虚数軸に沿って ∞j へと延びており、右から連続です。もう一つは $-1j$ から虚数軸に沿って $-\infty j$ へと延びており、左から連続です。

バージョン 2.6 で変更: 分枝切断が C99 標準で推奨されたものに合わせて動かされました。

`cmath.atanh(x)`

x の逆双曲線正接を返します。二つの分枝切断があります: 一つは 1 から実数軸に沿って ∞ へと延びており、下から連続です。もう一つは -1 から実数軸に沿って $-\infty$ へと延びており、上から連続です。

バージョン 2.6 で変更: 右側の分割での連続な方向が逆転しました。

`cmath.cosh(x)`

x の双曲線余弦を返します。

`cmath.sinh(x)`

x の双曲線正弦を返します。

`cmath.tanh(x)`

x の双曲線正接を返します。

9.3.5 類別関数

`cmath.isinf(x)`

x の実数部または虚数部が正または負の無限大であれば `True` を返します。

バージョン 2.6 で追加。

`cmath.isnan(x)`

x の実数部または虚数部が非数 (NaN) であれば `True` を返します。

バージョン 2.6 で追加。

9.3.6 定数

`cmath.pi`

定数 π (円周率) で、浮動小数点数です。

`cmath.e`

定数 e (自然対数の底) で、浮動小数点数です。

`math` と同じような関数が選ばれていますが、全く同じではないので注意してください。機能を二つのモジュールに分けているのは、複素数に興味が無かったり、もしかすると複素数とは何かすら知らないようなユーザーがいるからです。そういった人たちはむしろ、`math.sqrt(-1)` が複素数を返すよりも例外を送出してほしいと考えます。また、`cmath` で定義されている関数は、たとえ結果が実数で表現可能な場合 (虚数部がゼロの複素数) でも、常に複素数を返すので注意してください。

分枝切断 (branch cut) に関する注釈: 分枝切断をもつ曲線上では、与えられた関数は連続でありえなくなります。これらは多くの複素関数における必然的な特性です。複素関数を計算する必要がある場合、これらの分枝に関して理解しているものと仮定しています。悟りに至るために何らかの (到底基礎的とはいえない) 複素数に関する書をひもといてください。数値計算を目的とした分枝切断の正しい選択方法についての情報としては、以下がよい参考文献となります:

参考:

Kahan, W: Branch cuts for complex elementary functions; or, Much ado about nothings's sign bit. In Iserles, A., and Powell, M. (eds.), The state of the art in numerical analysis. Clarendon Press (1987) pp165–211.

9.4 decimal — 10 進固定及び浮動小数点数の算術演算

バージョン 2.4 で追加.

`decimal` モジュールは 10 進の浮動小数点算術をサポートします。 `decimal` には、 `float` データ型に比べて、以下のような利点があります:

- Decimal 「は人々を念頭にデザインされた浮動小数点モデルを元にしており、必然的に最も重要な指針があります – コンピュータは人々が学校で習った算術と同じように動作する算術を提供しなければならない」 – 10 進数演算仕様より。
- 10 進数を正確に表現できます。 `1.1` や `2.2` のような数は、2 進数の浮動小数点型では正しく表現できません。エンドユーザは普通、2 進数における `1.1 + 2.2` の近似値が `3.3000000000000003` だからといって、そのように表示してほしいとは思えないものです。
- 値の正確さは算術にも及びます。10 進の浮動小数点による計算では、`0.1 + 0.1 + 0.1 - 0.3` は厳密にゼロに等しくなります。2 進浮動小数点では `5.5511151231257827e-017` になってしまいます。ゼロに近い値とはいえ、この誤差は数値間の等価性テストの信頼性を阻害します。また、誤差が蓄積されることもあります。こうした理由から、数値間の等価性を厳しく保たなければならないようなアプリケーションを考えるなら、10 進数による数値表現が望ましいということになります。
- `decimal` モジュールでは、有効桁数の表記が取り入れられており、例えば `1.30 + 1.20` は `2.50` になります。すなわち、末尾のゼロは有効数字を示すために残されます。こうした仕様は通貨計算を行うアプリケーションでは慣例です。乗算の場合、「教科書的な」アプローチでは、乗算の被演算子すべての桁数を使います。例えば、`1.3 * 1.2` は `1.56` になり、`1.30 * 1.20` は `1.5600` になります。
- ハードウェアによる 2 進浮動小数点表現と違い、`decimal` モジュールでは計算精度をユーザが変更できます (デフォルトでは 28 桁です)。この桁数はほとんどの問題解決に十分な大きさです:

```
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

- 2 進と 10 進の浮動小数点は、いずれも広く公開されている標準仕様のもとに実装されています。組み込みの浮動小数点型では、標準仕様で提唱されている機能のほんのささやかな部分を利用できるにすぎませんが、`decimal` では標準仕様が要求している全ての機能を利用できます。必要に応じて、プログラマは値の丸めやシグナル処理を完全に制御できます。この中には全ての不正確な操作を例外でブロックして正確な算術を遵守させるオプションもあります。
- `decimal` モジュールは「偏見なく、正確な丸めなしの十進算術 (固定小数点算術と呼ばれることもある) と丸めありの浮動小数点数算術」(10 進数演算仕様より引用) をサポートするようにデザインされました。

このモジュールは、10 進数型、算術コンテキスト (context for arithmetic)、そしてシグナル (signal) という三つの概念を中心に設計されています。

10 進数型は変更不可能な型です。これは符号、係数部、そして指数を持ちます。有効桁数を残すために、仮数部の末尾にあるゼロは切り詰められません。 `decimal` では、 `Infinity`、`-Infinity`、および `NaN` といった特殊な値も定義されています。標準仕様では `-0` と `+0` も区別します。

算術コンテキストとは、精度や値丸めの規則、指数部の制限を決めている環境です。この環境では、演算結果を表すためのフラグや、演算上発生した特定のシグナルを例外として扱うかどうかを決めるトラップイネーブラも定義しています。丸め規則には `ROUND_CEILING`、`ROUND_DOWN`、`ROUND_FLOOR`、`ROUND_HALF_DOWN`、`ROUND_HALF_EVEN`、`ROUND_HALF_UP`、`ROUND_UP`、および `ROUND_05UP` があります。

シグナルとは、演算の過程で生じる例外的条件です。個々のシグナルは、アプリケーションそれぞれの要求に従って、無視されたり、単なる情報とみなされたり、例外として扱われたりします。 `decimal` モジュールには、 `Clamped`、`InvalidOperation`、`DivisionByZero`、`Inexact`、`Rounded`、`Subnormal`、`Overflow`、`Underflow` といったシグナルがあります。

各シグナルには、フラグとトラップイネーブラがあります。演算上何らかのシグナルに遭遇すると、フラグは 1 にセットされます。このとき、もしトラップイネーブラが 1 にセットされていれば、例外を送出します。フラグの値は膠着型 (sticky) なので、演算によるフラグの変化をモニタしたければ、予めフラグをリセットしておかなければなりません。

参考:

- IBM による汎用 10 進演算仕様、 [The General Decimal Arithmetic Specification](#)。

9.4.1 クイックスタートチュートリアル

普通、 `decimal` を使うときには、モジュールを `import` し、現在の演算コンテキストを `getcontext()` で調べ、必要なら、精度、丸め、有効なトラップを設定します:

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999,
        capitals=1, flags=[], traps=[Overflow, DivisionByZero,
        InvalidOperation])

>>> getcontext().prec = 7           # Set a new precision
```

`Decimal` インスタンスは、整数、文字列、浮動小数点数、またはタプルから構成できます。整数や浮動小数点数からの構成は、整数や浮動小数点数の値を正確に変換します。 `Decimal` は ”数値ではない (Not a Number)” を表す `NaN` や正負の `Infinity` (無限大)、 `-0` といった特殊な値も表現できます:

```
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.140000000000000124344978758017532527446746826171875')
>>> Decimal((0, (3, 1, 4), -2))
```

(次のページに続く)

(前のページからの続き)

```
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.41421356237')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.414213562373095048801688724')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

新たな *Decimal* の有効桁数は入力桁数だけで決まります。演算コンテキストにおける精度や値丸めの設定が影響するのは算術演算の間だけです。

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

decimal はほとんどの場面で Python の他の機能とうまくやりとりできます。decimal 浮動小数点小劇場 (flying circus) をお見せしましょう:

```
>>> data = map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split())
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
 Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
>>> sum(data)
Decimal('19.29')
>>> a,b,c = data[:3]
>>> str(a)
'1.34'
>>> float(a)
1.34
>>> round(a, 1)      # round() first converts to binary floating point
1.3
>>> int(a)
1
>>> a * 5
Decimal('6.70')
>>> a * b
Decimal('2.5058')
```

(次のページに続く)

(前のページからの続き)

```
>>> c % a
Decimal('0.77')
```

いくつかの数学的関数も `Decimal` には用意されています:

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.414213562373095048801688724')
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal('10').ln()
Decimal('2.302585092994045684017991455')
>>> Decimal('10').log10()
Decimal('1')
```

`quantize()` メソッドは位を固定して数値を丸めます。このメソッドは、結果を固定の桁数で丸めることがよくある、金融アプリケーションで便利です:

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

前述のように、`getcontext()` 関数を使うと現在の演算コンテキストにアクセスでき、設定を変更できます。ほとんどのアプリケーションはこのアプローチで十分です。

より高度な作業を行う場合、`Context()` コンストラクタを使って別の演算コンテキストを作っておくと便利ことがあります。別の演算コンテキストをアクティブにしたければ、`setcontext()` を使います。

`decimal` モジュールでは、標準仕様に従って、すぐ利用できる二つの標準コンテキスト、`BasicContext` および `ExtendedContext` を提供しています。前者はほとんどのトラップが有効になっており、とりわけデバッグの際に便利です:

```
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999,
       capitals=1, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')

>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
```

(次のページに続く)

(前のページからの続き)

```
File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

演算コンテキストには、演算中に遭遇した例外的状況をモニタするためのシグナルフラグがあります。フラグが一度セットされると、明示的にクリアするまで残り続けます。そのため、フラグのモニタを行いたいような演算の前には `clear_flags()` メソッドでフラグをクリアしておくのがベストです。

```
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal('3.14159292')
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999,
        capitals=1, flags=[Rounded, Inexact], traps=[])
```

`flags` エントリから、`Pi` の有理数による近似値が丸められた (コンテキスト内で決められた精度を超えた桁数が捨てられた) ことと、計算結果が厳密でない (無視された桁の値に非ゼロのものがあつた) ことがわかります。

コンテキストの `traps` フィールドに入っている辞書を使うと、個々のトラップをセットできます:

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

ほとんどのプログラムでは、開始時に一度だけ現在の演算コンテキストを修正します。また、多くのアプリケーションでは、データから `Decimal` への変換はループ内で一度だけキャストして行います。コンテキストを設定し、`Decimal` オブジェクトを生成できたら、ほとんどのプログラムは他の Python 数値型と全く変わらないかのように `Decimal` を操作できます。

9.4.2 Decimal オブジェクト

class `decimal.Decimal` (`[value[, context]]`)

`value` に基づいて新たな `Decimal` オブジェクトを構築します。

`value` は整数、文字列、タプル、`float` および他の `Decimal` オブジェクトにできます。 `value` を指定しない場合、`Decimal('0')` を返します。 `value` が文字列の場合、先頭と末尾の空白を取り除いた後には以下の 10 進数文字列の文法に従わなければなりません:

```
sign      ::= '+' | '-'
digit     ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

(次のページに続く)

(前のページからの続き)

```

indicator      ::= 'e' | 'E'
digits         ::= digit [digit]...
decimal-part   ::= digits '.' [digits] | ['.'] digits
exponent-part  ::= indicator [sign] digits
infinity       ::= 'Infinity' | 'Inf'
nan            ::= 'NaN' [digits] | 'sNaN' [digits]
numeric-value  ::= decimal-part [exponent-part] | infinity
numeric-string ::= [sign] numeric-value | [sign] nan

```

value を Unicode 文字列にした場合、他の Unicode 数字も上の *digit* の場所に使うことができます。つまり各書記体系における (アラビア-インド系やデーヴァナーガリーなど) の数字や、全角数字 0 (u'\uff10') から 9 (u'\uff19') までなどです。

value を *tuple* にする場合、タプルは三つの要素を持ち、それぞれ符号 (正なら 0、負なら 1)、仮数部を表す数字の *tuple*、そして指数を表す整数でなければなりません。例えば、`Decimal((0, (1, 4, 1, 4), -3))` は `Decimal('1.414')` を返します。

value を *float* にする場合、2 進浮動小数点数値が損失なく正確に等価な *Decimal* に変換されます。この変換はしばしば 53 桁以上の精度を要求します。例えば、`Decimal(float('1.1'))` は `Decimal('1.1000000000000000088817841970012523233890533447265625')` に変換されます。

context の精度 (precision) は、記憶される桁数には影響しません。桁数は *value* に指定した桁数だけから決定されます。例えば、演算コンテキストに指定された精度が 3 桁しかなくても、`Decimal('3.00000')` は 5 つのゼロを全て記憶します。

context 引数の目的は、*value* が正しくない形式の文字列であった場合に行う処理を決めることにあります; 演算コンテキストが *InvalidOperation* をトラップするようになっていれば、例外を送出します。それ以外の場合には、コンストラクタは値が NaN の *Decimal* を返します。

一度生成すると、*Decimal* オブジェクトは変更不能 (immutable) になります。

バージョン 2.6 で変更: 文字列から *Decimal* インスタンスを生成する際に先頭と末尾の空白が許されることになりました。

バージョン 2.7 で変更: コンストラクタに対する引数に *float* インスタンスも許されるようになりました。

10 進浮動小数点オブジェクトは、*float* や *int* のような他の組み込み型と多くの点で似ています。通常の数学演算や特殊メソッドを適用できます。また、*Decimal* オブジェクトはコピーでき、pickle 化でき、print で出力でき、辞書のキーにでき、集合の要素にでき、比較、保存、他の型 (*float* や *long*) への型強制を行えます。

10 進オブジェクトの算術演算と整数や浮動小数点数の算術演算には少々違いがあります。10 進オブジェクトに対して剰余演算を適用すると、計算結果の符号は除数の符号ではなく 被除数の符号と一致します:

```
>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
```

整数除算演算子 `//` も同様に、実際の商の切り捨てではなく (0 に近付くように丸めた) 整数部分を返します。そうすることで通常の恒等式 $x == (x // y) * y + x \% y$ が保持されます:

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

演算子 `%` と演算子 `//` は (それぞれ) 仕様にあるような 剰余 操作と 整数除算 操作を実装しています。

`Decimal` オブジェクトは一般に、算術演算で浮動小数点数と組み合わせることができません。例えば、`Decimal` に `float` を足そうとすると、`TypeError` が送出されます。ただしこの規則には例外があります。 `float` インスタンス `x` と `Decimal` インスタンス `y` を比較する比較演算子です。この例外がなかったとすると、`Decimal` と `float` インスタンスの比較は、リファレンスマニュアルの `expressions` 節で記述されている、異なる型のオブジェクトを比較するときの一般の規則に従うことになり、紛らわしい結果につながります。

バージョン 2.7 で変更: `float` インスタンス `x` と `Decimal` インスタンス `y` の比較は、`x` と `y` の値に基づく結果を返すようになりました。以前のバージョンでは、どんな `float` インスタンス `x` とどんな `Decimal` インスタンス `y` に対しても、`x < y` は同じ (任意の) 結果を返していました。

こうした標準的な数値型の特性の他に、10 進浮動小数点オブジェクトには様々な特殊メソッドがあります:

`adjusted()`

仮数の先頭の一桁だけが残るように右側の数字を追いつす桁シフトを行い、その結果の指数部を返します: `Decimal('321e+5').adjusted()` は 7 を返します。最上桁の小数点からの相対位置を調べる際に使います。

`as_tuple()`

数値を表現するための名前付きタプル (*named tuple*): `DecimalTuple(sign, digittuple, exponent)` を返します。

バージョン 2.6 で変更: 名前付きタプル (*named tuple*) を使うようになりました。

`canonical()`

引数の標準的 (*canonical*) エンコーディングを返します。現在のところ、`Decimal` インスタンスのエンコーディングは常に標準的なので、この操作は引数に手を加えずに返します。

バージョン 2.6 で追加.

`compare(other[, context])`

二つの `Decimal` インスタンスの値を比較します。この演算は普通の比較メソッド `__cmp__()` と同じように振舞いますが、`compare()` は整数ではなく `Decimal` インスタンスを返し、また、被演

算子のどちらかが NaN ならば NaN を返します:

```
a or b is a NaN ==> Decimal('NaN')
a < b           ==> Decimal('-1')
a == b          ==> Decimal('0')
a > b           ==> Decimal('1')
```

compare_signal(*other*[, *context*])

この演算は *compare()* とほとんど同じですが、全ての NaN がシグナルを送るところが異なります。すなわち、どちらの比較対象も発信 (signaling) NaN でないならば無言 (quiet) NaN である比較対象があたかも発信 NaN であるかのように扱われます。

バージョン 2.6 で追加.

compare_total(*other*)

二つの対象を数値によらず抽象表現によって比較します。 *compare()* に似ていますが、結果は *Decimal* に全順序を与えます。この順序づけによると、数値的に等しくても異なった表現を持つ二つの *Decimal* インスタンスの比較は等しくなりません:

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

無言 NaN と発信 NaN もこの全順序に位置付けられます。この関数の結果は、もし比較対象が同じ表現を持つならば *Decimal('0')* であり、一つめの比較対象が二つめより下位にあれば *Decimal('-1')*、上位にあれば *Decimal('1')* です。全順序の詳細については仕様を参照してください。

バージョン 2.6 で追加.

compare_total_mag(*other*)

二つの対象を *compare_total()* のように数値によらず抽象表現によって比較しますが、両者の符号を無視します。 *x.compare_total_mag(y)* は *x.copy_abs().compare_total(y.copy_abs())* と等価です。

バージョン 2.6 で追加.

conjugate()

self を返すだけです。このメソッドは十進演算仕様に適合するためだけのものです。

バージョン 2.6 で追加.

copy_abs()

引数の絶対値を返します。この演算はコンテキストに影響されず、静かです。すなわち、フラグは変更されず、丸めは行われません。

バージョン 2.6 で追加.

copy_negate()

引数の符号を変えて返します。この演算はコンテキストに影響されず、静かです。すなわち、フラグは変更されず、丸めは行われません。

バージョン 2.6 で追加.

copy_sign (*other*)

最初の演算対象のコピーに二つめと同じ符号を付けて返します。たとえば:

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

この演算はコンテキストに影響されず、静かです。すなわち、フラグは変更されず、丸めは行われません。

バージョン 2.6 で追加.

exp ([*context*])

与えられた数での (自然) 指数関数 e^{**x} の値を返します。結果は ROUND_HALF_EVEN 丸めモードで正しく丸められます。

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

バージョン 2.6 で追加.

from_float (*f*)

浮動小数点数を正確に小数に変換するクラスメソッドです。

なお、*Decimal.from_float(0.1)* は *Decimal('0.1')* と同じではありません。0.1 は二進浮動小数点数で正確に表せないなので、その値は表現できる最も近い値、 $0x1.999999999999ap-4$ として記憶されます。浮動小数点数での等価な値は *0.1000000000000000055511151231257827021181583404541015625* です。

注釈: Python 2.7 以降では、*Decimal* インスタンスは *float* から直接構築できるようになりました。

```
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

バージョン 2.7 で追加.

fma (*other*, *third* [, *context*])

融合積和 (fused multiply-add) です。self*other+third を途中結果の積 self*other で丸めを行わずに計算して返します。


```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

バージョン 2.6 で追加.

`is_canonical()`

引数が標準的 (canonical) ならば *True* を返し、そうでなければ *False* を返します。現在のところ、*Decimal* のインスタンスは常に標準的なのでこのメソッドの結果はいつでも *True* です。

バージョン 2.6 で追加.

`is_finite()`

引数が有限の数値ならば *True* を、無限大か NaN ならば *False* を返します。

バージョン 2.6 で追加.

`is_infinite()`

引数が正または負の無限大ならば *True* を、そうでなければ *False* を返します。

バージョン 2.6 で追加.

`is_nan()`

引数が (無言か発信かは問わず) NaN であれば *True* を、そうでなければ *False* を返します。

バージョン 2.6 で追加.

`is_normal()`

引数が 正規 (*normal*) のゼロでない有限数値で調整された指数が *Emin* 以上ならば *True* を返します。引数がゼロ、非正規 (*subnormal*)、無限大または NaN であれば *False* を返します。ここでの 正規 という用語は標準的な (canonical) 値を作り出すために使われる *normalize()* メソッドにおける意味合いとは異なりますので注意して下さい。

バージョン 2.6 で追加.

`is_qnan()`

引数が無言 NaN であれば *True* を、そうでなければ *False* を返します。

バージョン 2.6 で追加.

`is_signed()`

引数に負の符号がついていれば *True* を、そうでなければ *False* を返します。注意すべきはゼロや NaN なども符号を持ち得ることです。

バージョン 2.6 で追加.

`is_snan()`

引数が発信 NaN であれば *True* を、そうでなければ *False* を返します。

バージョン 2.6 で追加.

`is_subnormal()`

引数が非正規数 (*subnormal*) であれば *True* を、そうでなければ *False* を返します。非正規な数

値とは、ゼロでなく、有限で、調整された指数が *Emin* 未満のものを指します。

バージョン 2.6 で追加.

`is_zero()`

引数が (正または負の) ゼロであれば *True* を、そうでなければ *False* を返します。

バージョン 2.6 で追加.

`ln([context])`

演算対象の自然対数 (底 *e* の対数) を返します。結果は `ROUND_HALF_EVEN` 丸めモードで正しく丸められます。

バージョン 2.6 で追加.

`log10([context])`

演算対象の底 10 の対数を返します。結果は `ROUND_HALF_EVEN` 丸めモードで正しく丸められます。

バージョン 2.6 で追加.

`logb([context])`

非零の数値については、*Decimal* インスタンスとして調整された指数を返します。演算対象がゼロだった場合、`Decimal('-Infinity')` が返され *DivisionByZero* フラグが送出されます。演算対象が無限大だった場合、`Decimal('Infinity')` が返されます。

バージョン 2.6 で追加.

`logical_and(other[, context])`

`logical_and()` は二つの 論理引数 (論理引数 参照) を取る論理演算です。結果は二つの引数の数字ごとの `and` です。

バージョン 2.6 で追加.

`logical_invert([context])`

`logical_invert()` は論理演算です。結果は引数の数字ごとの反転です。

バージョン 2.6 で追加.

`logical_or(other[, context])`

`logical_or()` は二つの 論理引数 (論理引数 参照) を取る論理演算です。結果は二つの引数の数字ごとの `or` です。

バージョン 2.6 で追加.

`logical_xor(other[, context])`

`logical_xor()` は二つの 論理引数 (論理引数 参照) を取る論理演算です。結果は二つの引数の数字ごとの排他的論理和です。

バージョン 2.6 で追加.

`max(other[, context])`

`max(self, other)` と同じですが、値を返す前に現在のコンテキストに即した丸め規則を適用

します。また、NaN に対して、(コンテキストの設定と、発信が無言どちらのタイプであるかに応じて) シグナルを発行するか無視します。

max_mag(*other*[, *context*])

max() メソッドに似ていますが、比較は絶対値で行われます。

バージョン 2.6 で追加.

min(*other*[, *context*])

min(*self*, *other*) と同じですが、値を返す前に現在のコンテキストに即した丸め規則を適用します。また、NaN に対して、(コンテキストの設定と、発信が無言どちらのタイプであるかに応じて) シグナルを発行するか無視します。

min_mag(*other*[, *context*])

min() メソッドに似ていますが、比較は絶対値で行われます。

バージョン 2.6 で追加.

next_minus([*context*])

与えられたコンテキスト (またはコンテキストが渡されなければ現スレッドのコンテキスト) において表現可能な、操作対象より小さい最大の数を返します。

バージョン 2.6 で追加.

next_plus([*context*])

与えられたコンテキスト (またはコンテキストが渡されなければ現スレッドのコンテキスト) において表現可能な、操作対象より大きい最小の数を返します。

バージョン 2.6 で追加.

next_toward(*other*[, *context*])

二つの比較対象が等しくなければ、一つめの対象に最も近く二つめの対象へ近付く方向の数を返します。もし両者が数値的に等しければ、二つめの対象の符号を採った一つめの対象のコピーを返します。

バージョン 2.6 で追加.

normalize([*context*])

数値を正規化 (normalize) して、右端に連続しているゼロを除去し、`Decimal('0')` と同じ結果はすべて `Decimal('0e0')` に変換します。等価クラスの属性から基準表現を生成する際に用います。たとえば、`Decimal('32.100')` と `Decimal('0.321000e+2')` の正規化は、いずれも同じ値 `Decimal('32.1')` になります。

number_class([*context*])

操作対象の クラス を表す文字列を返します。返されるのは以下の 10 種類のいずれかです。

- `"-Infinity"`, 負の無限大であることを示します。
- `"-Normal"`, 負の通常数であることを示します。
- `"-Subnormal"`, 負の非正規数であることを示します。

- `"-Zero"`, 負のゼロであることを示します。
- `"+Zero"`, 正のゼロであることを示します。
- `"+Subnormal"`, 正の非正規数であることを示します。
- `"+Normal"`, 正の通常数であることを示します。
- `"+Infinity"`, 正の無限大であることを示します。
- `"NaN"`, 無言 (quiet) NaN (Not a Number) であることを示します。
- `"sNaN"`, 発信 (signaling) NaN であることを示します。

バージョン 2.6 で追加.

quantize(*exp*[, *rounding*[, *context*[, *watchexp*]]])

二つめの操作対象と同じ指数を持つように丸めを行った、一つめの操作対象と等しい値を返します。

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

他の操作と違い、打ち切り (quantize) 操作後の係数の長さが精度を越えた場合には、`InvalidOperation` がシグナルされます。これによりエラー条件がない限り打ち切られた指数が常に右側の引数と同じになることが保証されます。

同様に、他の操作と違い、quantize は Underflow を、たとえ結果が非正規になったり不正確になったとしても、シグナルしません。

二つ目の演算対象の指数が一つ目のそれよりも大きければ丸めが必要かもしれません。この場合、丸めモードは以下のように決められます。rounding 引数が与えられていればそれが使われます。そうでなければ context 引数で決まります。どちらの引数も渡されなければ現在のスレッドのコンテキストの丸めモードが使われます。

watchexp が設定されている場合 (デフォルト)、処理結果の指数が `Emax` よりも大きい場合や `Etiny` よりも小さい場合にエラーを返します。

radix()

`Decimal(10)` つまり `Decimal` クラスがその全ての算術を実行する基数を返します。仕様との互換性のために取り入れられています。

バージョン 2.6 で追加.

remainder_near(*other*[, *context*])

self を *other* で割った剰余を返します。これは `self % other` とは違って、剰余の絶対値を小さくするように符号が選ばれます。より詳しく言うと、*n* を `self / other` の正確な値に最も近い整数としたときの `self - n * other` が返り値になります。最も近い整数が 2 つある場合には偶数のものが選ばれます。

結果が 0 になる場合の符号は *self* の符号と同じになります。

```
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

rotate(*other*[, *context*])

一つめの演算対象の数字を二つめので指定された量だけ巡回 (rotate) した結果を返します。二つめの演算対象は `-precision` から `precision` までの範囲の整数でなければなりません。この二つめの演算対象の絶対値が何桁ずらすかを決めます。そしてもし正の数ならば巡回の方向は左に、そうでなければ右になります。一つめの演算対象の仮数部は必要ならば精度いっぱいまでゼロで埋められます。符号と指数は変えられません。

バージョン 2.6 で追加。

same_quantum(*other*[, *context*])

self と *other* が同じ指数を持っているか、あるいは双方とも NaN である場合に真を返します。

scaleb(*other*[, *context*])

二つめの演算対象で調整された指数の一つめの演算対象を返します。同じことですが、一つめの演算対象を `10**other` 倍したものを返します。二つめの演算対象は整数でなければなりません。

バージョン 2.6 で追加。

shift(*other*[, *context*])

一つめの演算対象の数字を二つめので指定された量だけシフトした結果を返します。二つめの演算対象は `-precision` から `precision` までの範囲の整数でなければなりません。この二つめの演算対象の絶対値が何桁ずらすかを決めます。そしてもし正の数ならばシフトの方向は左に、そうでなければ右になります。一つめの演算対象の係数は必要ならば精度いっぱいまでゼロで埋められます。符号と指数は変えられません。

バージョン 2.6 で追加。

sqrt([, *context*])

平方根を精度いっぱいまで求めます。

to_eng_string([, *context*])

文字列に変換します。指数が必要なら工学表記が使われます。

工学表記法では指数は 3 の倍数になります。これにより、基数の小数部には最大で 3 桁までの数字が残されるとともに、末尾に 1 つまたは 2 つの 0 の付加が必要とされるかもしれません。

たとえば、`Decimal('123E+1')` は `Decimal('1.23E+3')` に変換されます。

to_integral([, *rounding*[, *context*]])

`to_integral_value()` メソッドと同じです。to_integral の名前は古いバージョンとの互換性のために残されています。

to_integral_exact([, *rounding*[, *context*]])

最近傍の整数に値を丸め、丸めが起こった場合には *Inexact* または *Rounded* のシグナルを適切に出します。丸めモードは以下のように決められます。 `rounding` 引数が与えられていればそれが使われます。そうでなければ `context` 引数で決まります。どちらの引数も渡されなければ現在のスレッドのコンテキストの丸めモードが使われます。

バージョン 2.6 で追加.

`to_integral_value([rounding[, context]])`

Inexact や *Rounded* といったシグナルを出さずに最近傍の整数に値を丸めます。 `rounding` が指定されていれば適用されます; それ以外の場合、値丸めの方法は `context` の設定が現在のコンテキストの設定になります。

バージョン 2.6 で変更: `to_integral` から `to_integral_value` に改名されました。古い名前も互換性のために残されています。

論理引数

`logical_and()`, `logical_invert()`, `logical_or()`, および `logical_xor()` メソッドはその引数が論理引数であると想定しています。論理引数とは *Decimal* インスタンスで指数と符号は共にゼロであり、各桁の数字が 0 か 1 であるものです。

9.4.3 Context オブジェクト

コンテキスト (context) とは、算術演算における環境設定です。コンテキストは計算精度を決定し、値丸めの方法を設定し、シグナルのどれが例外になるかを決め、指数の範囲を制限しています。

多重スレッドで処理を行う場合には各スレッドごとに現在のコンテキストがあり、 `getcontext()` や `setcontext()` といった関数でアクセスしたり設定変更できます:

`decimal.getcontext()`

アクティブなスレッドの現在のコンテキストを返します。

`decimal.setcontext(c)`

アクティブなスレッドのコンテキストを `c` に設定します。

Python 2.5 から、`with` 文と `localcontext()` 関数を使って実行するコンテキストを一時的に変更することもできるようになりました。

`decimal.localcontext([c])`

`with` 文の入口でアクティブなスレッドのコンテキストを `c` のコピーに設定し、`with` 文を抜ける時に元のコンテキストに復旧する、コンテキストマネージャを返します。コンテキストが指定されなければ、現在のコンテキストのコピーが使われます。

バージョン 2.5 で追加.

たとえば、以下のコードでは精度を 42 桁に設定し、計算を実行し、そして元のコンテキストに復帰します:

```

from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42    # Perform a high precision calculation
    s = calculate_something()
s = +s    # Round the final result back to the default precision

with localcontext(BasicContext):    # temporarily use the BasicContext
    print Decimal(1) / Decimal(7)
    print Decimal(355) / Decimal(113)

```

新たなコンテキストは、以下で説明する *Context* コンストラクタを使って生成できます。その他にも、*decimal* モジュールでは作成済みのコンテキストを提供しています:

class decimal.BasicContext

汎用 10 進演算仕様で定義されている標準コンテキストの一つです。精度は 9 桁に設定されています。丸め規則は `ROUND_HALF_UP` です。すべての演算結果フラグはクリアされています。 *Inexact*, *Rounded*, *Subnormal* を除く全ての演算エラートラップが有効 (例外として扱う) になっています。

多くのトラップが有効になっているので、デバッグの際に便利なコンテキストです。

class decimal.ExtendedContext

汎用 10 進演算仕様で定義されている標準コンテキストの一つです。精度は 9 桁に設定されています。丸め規則は `ROUND_HALF_EVEN` です。すべての演算結果フラグはクリアされています。トラップは全て無効 (演算中に一切例外を送出しない) になっています。

トラップが無効になっているので、エラーの伴う演算結果を `NaN` や `Infinity` にし、例外を送出しないようにしたいアプリケーションに向けたコンテキストです。このコンテキストを使うと、他の場合にはプログラムが停止してしまうような状況があっても実行を完了させられます。

class decimal.DefaultContext

Context コンストラクタが新たなコンテキストを作成するさいに雛形にするコンテキストです。このコンテキストのフィールド (精度の設定など) を変更すると、*Context* コンストラクタが生成する新たなコンテキストに影響を及ぼします。

このコンテキストは、主に多重スレッド環境で便利です。スレッドを開始する前に何らかのフィールドを変更しておく、システム全体のデフォルト設定に効果を及ぼすことができます。スレッドを開始した後にフィールドを変更すると、競合条件を抑制するためにスレッドを同期化しなければならないので、推奨しません。

単一スレッドの環境では、このコンテキストを使わないよう薦めます。下で述べるように明示的にコンテキストを作成してください。

デフォルトの値は精度 28 桁、丸め規則 `ROUND_HALF_EVEN` で、トラップ *Overflow*, *InvalidOperation*, および *DivisionByZero* が有効になっています。

上に挙げた三つのコンテキストに加え、*Context* コンストラクタを使って新たなコンテキストを生成できます。

class decimal.Context (*prec=None, rounding=None, traps=None, flags=None, Emin=None, Emax=None, capitals=1*)

新たなコンテキストを生成します。あるフィールドが定義されていないか *None* であれば、*DefaultContext* からデフォルト値をコピーします。*flags* フィールドが設定されているか *None* の場合には、全てのフラグがクリアされます。

prec フィールドは正の整数で、コンテキストにおける算術演算の計算精度を設定します。

rounding オプションは以下のどれか一つです:

- ROUND_CEILING (Infinity 寄りの値に丸める),
- ROUND_DOWN (ゼロ寄りの値に丸める),
- ROUND_FLOOR (-Infinity 寄りの値に丸める),
- ROUND_HALF_DOWN (近い方に、引き分けはゼロ方向に向けて丸める),
- ROUND_HALF_EVEN (近い方に、引き分けは偶数整数方向に向けて丸める),
- ROUND_HALF_UP (近い方に、引き分けはゼロから遠い方向に向けて丸める),
- ROUND_UP (ゼロから遠い値に丸める), または
- ROUND_05UP (ゼロ方向に丸めた後の最後の桁が 0 または 5 ならばゼロから遠い方向に、そうでなければゼロ方向に丸める)

traps および *flags* フィールドには、セットしたいシグナルを列挙します。一般的に、新たなコンテキストを作成するときにはトラップだけを設定し、フラグはクリアしておきます。

Emin および *Emax* フィールドには、指数の外限を整数で指定します。

capitals フィールドは 0 または 1 (デフォルト) にします。1 に設定すると、指数記号を大文字 E で出力します。それ以外の場合には *Decimal('6.02e+23')* のように e を使います。

バージョン 2.6 で変更: ROUND_05UP 丸めモードが追加されました。

Context クラスでは、いくつかの汎用のメソッドの他、現在のコンテキストで算術演算を直接行うためのメソッドを数多く定義しています。加えて、*Decimal* の各メソッドについて (*adjusted()* および *as_tuple()* メソッドを例外として) 対応する *Context* のメソッドが存在します。たとえば、*Context* インスタンス *C* と *Decimal* インスタンス *x* に対して、*C.exp(x)* は *x.exp(context=C)* と等価です。それぞれの *Context* メソッドは、*Decimal* インスタンスが受け付けられるところならどこでも、Python の整数 (*int* または *long* のインスタンス) を受け付けます。

clear_flags()

フラグを全て 0 にリセットします。

copy()

コンテキストの複製を返します。

copy_decimal(num)

Decimal インスタンス *num* のコピーを返します。

create_decimal(*num*)

self をコンテキストとする新たな Decimal インスタンスを *num* から生成します。Decimal コンストラクタと違い、数値を変換する際にコンテキストの精度、値丸め方法、フラグ、トラップを適用します。

定数値はしばしばアプリケーションの要求よりも高い精度を持っているため、このメソッドが役に立ちます。また、値丸めを即座に行うため、例えば以下のように、入力値に値丸めを行わないために合計値にゼロの加算を追加するだけで結果が変わってしまうといった、現在の精度よりも細かい値の影響が紛れ込む問題を防げるという恩恵もあります。以下の例は、丸められていない入力を使うということは和にゼロを加えると結果が変わり得るという見本です:

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

このメソッドは IBM 仕様の to-number 演算を実装したものです。引数が文字列の場合、前や後ろに余計な空白を付けることは許されません。

create_decimal_from_float(*f*)

浮動小数点数 *f* から新しい Decimal インスタンスを生成しますが、*self* をコンテキストとして丸めます。Decimal.from_float() クラスメソッドとは違い、変換にコンテキストの精度、丸めメソッド、フラグ、そしてトラップが適用されます。

```
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(math.pi)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create_decimal_from_float(math.pi)
Traceback (most recent call last):
...
Inexact: None
```

バージョン 2.7 で追加.

Etiny()

$E_{min} - prec + 1$ に等しい値を返します。演算結果の劣化が起こる桁の最小値です。アンダーフローが起きた場合、指数は *Etiny* に設定されます。

Etop()

$E_{max} - prec + 1$ に等しい値を返します。

Decimal を使った処理を行う場合、通常は Decimal インスタンスを生成して、算術演算を適用するというアプローチをとります。演算はアクティブなスレッドにおける現在のコンテキストの下で行われます。もう一つのアプローチは、コンテキストのメソッドを使った特定のコンテキスト下での計算です。コンテキストのメソッドは Decimal クラスのメソッドに似ているので、ここでは簡単な説明にとどめます。

abs (*x*)

x の絶対値を返します。

add (*x*, *y*)

x と *y* の和を返します。

canonical (*x*)

同じ Decimal オブジェクト *x* を返します。

compare (*x*, *y*)

x と *y* を数値として比較します。

compare_signal (*x*, *y*)

二つの演算対象の値を数値として比較します。

compare_total (*x*, *y*)

二つの演算対象を抽象的な表現を使って比較します。

compare_total_mag (*x*, *y*)

二つの演算対象を抽象的な表現を使い符号を無視して比較します。

copy_abs (*x*)

x のコピーの符号を 0 にセットして返します。

copy_negate (*x*)

x のコピーの符号を反転して返します。

copy_sign (*x*, *y*)

y から *x* に符号をコピーします。

divide (*x*, *y*)

x を *y* で除算した値を返します。

divide_int (*x*, *y*)

x を *y* で除算した値を整数に切り捨てて返します。

divmod (*x*, *y*)

二つの数値間の除算を行い、結果の整数部を返します。

exp (*x*)

e^{**x} を返します。

fma (*x*, *y*, *z*)

x を *y* 倍したものに *z* を加えて返します。

is_canonical (*x*)

x が標準的 (canonical) ならば True を返します。そうでなければ False です。

is_finite (*x*)

x が有限ならば True を返します。そうでなければ False です。

is.infinite(*x*)

x が無限ならば `True` を返します。そうでなければ `False` です。

is.nan(*x*)

x が `qNaN` か `sNaN` であれば `True` を返します。そうでなければ `False` です。

is.normal(*x*)

x が通常の数ならば `True` を返します。そうでなければ `False` です。

is.qnan(*x*)

x が無言 NaN であれば `True` を返します。そうでなければ `False` です。

is.signed(*x*)

x が負の数であれば `True` を返します。そうでなければ `False` です。

is.snan(*x*)

x が発信 NaN であれば `True` を返します。そうでなければ `False` です。

is.subnormal(*x*)

x が非正規数であれば `True` を返します。そうでなければ `False` です。

is.zero(*x*)

x がゼロであれば `True` を返します。そうでなければ `False` です。

ln(*x*)

x の自然対数 (底 *e* の対数) を返します。

log10(*x*)

x の底 10 の対数を返します。

logb(*x*)

演算対象の MSD の大きさの指数部を返します。

logical.and(*x*, *y*)

それぞれの桁に論理演算 *and* を当てはめます。

logical.invert(*x*)

x の全ての桁を反転させます。

logical.or(*x*, *y*)

それぞれの桁に論理演算 *or* を当てはめます。

logical.xor(*x*, *y*)

それぞれの桁に論理演算 *xor* を当てはめます。

max(*x*, *y*)

二つの値を数値として比較し、大きいほうを返します。

max.mag(*x*, *y*)

値を符号を無視して数値として比較します。

min(*x*, *y*)

二つの値を数値として比較し、小さいほうを返します。

min_mag(*x*, *y*)

値を符号を無視して数値として比較します。

minus(*x*)

Python における単項マイナス演算子に対応する演算です。

multiply(*x*, *y*)

x と *y* の積を返します。

next_minus(*x*)

x より小さい最大の表現可能な数を返します。

next_plus(*x*)

x より大きい最小の表現可能な数を返します。

next_toward(*x*, *y*)

x に *y* の方向に向かって最も近い数を返します。

normalize(*x*)

x をもっとも単純な形にします。

number_class(*x*)

x のクラスを指し示すものを返します。

plus(*x*)

Python における単項のプラス演算子に対応する演算です。コンテキストにおける精度や値丸めを適用するので、等値 (identity) 演算とは違います。

power(*x*, *y*[, *modulo*])

x の *y* 乗を計算します。*modulo* が指定されていればモジュロを取ります。

二引数であれば *x**y* を計算します。*x* が負であれば *y* は整数でなければなりません。結果は *y* が整数であって結果が有限になり 'precision' 桁で正確に表現できるのでなければ不正確になります。その結果は現スレッドのコンテキストの丸めモードを使って正しく丸められます。

三引数であれば (*x**y*) % *modulo* を計算します。この形式の場合、以下の制限が引数に掛かります:

- 全ての引数は整数
- *y* は非負でなければならない
- *x* と *y* の少なくともどちらかはゼロでない
- *modulo* は非零で大きくても 'precision' 桁

`Context.power(x, y, modulo)` で得られる値は (*x**y*) % *modulo* を精度無制限で計算して得られるものと同じ値ですが、より効率的に計算されます。結果の指数は *x*, *y*, *modulo* の指数に関係なくゼロです。この計算は常に正確です。

バージョン 2.6 で変更: $x**y$ 形式で y が非整数で構わないことになりました。三引数バージョンがより厳格な要求をするようになりました。

quantize (x, y)

x に値丸めを適用し、指数を y にした値を返します。

radix ()

単に 10 を返します。何せ十進ですから :)

remainder (x, y)

整数除算の剰余を返します。

剰余がゼロでない場合、符号は割られる数の符号と同じになります。

remainder_near (x, y)

$x - y * n$ を返します。ここで n は x / y の正確な値に一番近い整数です (この結果が 0 ならばその符号は x の符号と同じです)。

rotate (x, y)

x の y 回巡回したコピーを返します。

same_quantum (x, y)

2 つの演算対象が同じ指数を持っている場合に True を返します。

scaleb (x, y)

一つめの演算対象の指数部に二つめの値を加えたものを返します。

shift (x, y)

x を y 回シフトしたコピーを返します。

sqrt (x)

x の平方根を精度いっぱいまで求めます。

subtract (x, y)

x と y の間の差を返します。

to_eng_string (x)

文字列に変換します。指数が必要なら工学表記が使われます。

工学表記法では指数は 3 の倍数になります。これにより、基数の小数部には最大で 3 桁までの数字が残されるとともに、末尾に 1 つまたは 2 つの 0 の付加が必要とされるかもしれません。

to_integral_exact (x)

最近傍の整数に値を丸めます。

to_sci_string (x)

数値を科学表記で文字列に変換します。

9.4.4 シグナル

シグナルは、計算中に生じた様々なエラー条件を表現します。各々のシグナルは一つのコンテキストフラグと一つのトラップイネーブラに対応しています。

コンテキストフラグは、該当するエラー条件に遭遇するたびにセットされます。演算後にフラグを調べれば、演算に関する情報（例えば計算が厳密だったかどうか）がわかります。フラグを調べたら、次の計算を始める前にフラグを全てクリアするようにしてください。

あるコンテキストのトラップイネーブラがあるシグナルに対してセットされている場合、該当するエラー条件が生じると Python の例外を送出します。例えば、`DivisionByZero` が設定されていると、エラー条件が生じた際に `DivisionByZero` 例外を送出します。

class `decimal.Clamped`

値の表現上の制限に沿わせるために指数部が変更されたことを通知します。

通常、クランプ (clamp) は、指数部がコンテキストにおける指数桁の制限値 `Emin` および `Emax` を越えた場合に発生します。可能な場合には、係数部にゼロを加えた表現に合わせて指数部を減らします。

class `decimal.DecimalException`

他のシグナルの基底クラスで、`ArithmeticError` のサブクラスです。

class `decimal.DivisionByZero`

有限値をゼロで除算したときのシグナルです。

除算やモジュロ除算、数を負の値で累乗した場合に起きることがあります。このシグナルをトラップしない場合、演算結果は `Infinity` または `-Infinity` になり、その符号は演算に使った入力に基づいて決まります。

class `decimal.Inexact`

値の丸めによって演算結果から厳密さが失われたことを通知します。

このシグナルは値丸め操作中にゼロでない桁を無視した際に生じます。演算結果は値丸め後の値です。シグナルのフラグやトラップは、演算結果の厳密さが失われたことを検出するために使えるだけです。

class `decimal.InvalidOperation`

無効な演算が実行されたことを通知します。

ユーザが有意な演算結果にならないような操作を要求したことを示します。このシグナルをトラップしない場合、`NaN` を返します。このシグナルの発生原因として考えられるのは、以下のような状況です:

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
x._rescale( non-integer )
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```


class decimal.Overflow

数値オーバーフローを示すシグナルです。

このシグナルは、値丸めを行った後の指数部が `Emax` より大きいことを示します。シグナルをトラップしない場合、演算結果は値丸めのモードにより、表現可能な最大の数値になるように内側へ引き込んで丸めを行った値か、`Infinity` になるように外側に丸めた値のいずれかになります。いずれの場合も、*Inexact* および *Rounded* が同時にシグナルされます。

class decimal.Rounded

情報が全く失われていない場合も含み、値丸めが起きたときのシグナルです。

このシグナルは、値丸めによって桁がなくなると常に発生します。なくなった桁がゼロ (例えば `5.00` を丸めて `5.0` になった場合) であってもです。このシグナルをトラップしなければ、演算結果をそのまま返します。このシグナルは有効桁数の減少を検出する際に使います。

class decimal.Subnormal

値丸めを行う前に指数部が `Emin` より小さかったことを示すシグナルです。

演算結果が微小である場合 (指数が小さすぎる場合) に発生します。このシグナルをトラップしなければ、演算結果をそのまま返します。

class decimal.Underflow

演算結果が値丸めによってゼロになった場合に生じる数値アンダフローです。

演算結果が微小なため、値丸めによってゼロになった場合に発生します。 *Inexact* および *Subnormal* シグナルも同時に発生します。

これらのシグナルの階層構造をまとめると、以下の表のようになります:

```
exceptions.ArithmeticError(exceptions.StandardError)
    DecimalException
        Clamped
        DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
        Inexact
            Overflow(Inexact, Rounded)
            Underflow(Inexact, Rounded, Subnormal)
        InvalidOperation
        Rounded
        Subnormal
```

9.4.5 浮動小数点数に関する注意

精度を上げて丸め誤差を抑制する

10 進浮動小数点数を使うと、10 進数表現による誤差を抑制できます (`0.1` を正確に表現できるようになります); しかし、ゼロでない桁が一定の精度を越えている場合には、演算によっては依然として値丸めによる誤差を引き起こします。

値丸めによる誤差の影響は、桁落ちを生じるような、ほとんど相殺される量での加算や減算によって増幅されます。Knuth は、十分でない計算精度の下で値丸めを伴う浮動小数点演算を行った結果、加算の結合則や分配

則における恒等性が崩れてしまう例を二つ示しています:

```
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.5111111')
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
>>> u * (v+w)
Decimal('0.0060000')
```

`decimal` モジュールでは、最下桁を失わないように十分に計算精度を広げることで、上で問題にしたような恒等性をとりもどせます:

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
>>> u * (v+w)
Decimal('0.0060000')
```

特殊値

`decimal` モジュールの数体系では、NaN, sNaN, `-Infinity`, `Infinity`, および二つのゼロ、`+0` と `-0` といった特殊な値を提供しています。

無限大 (`Infinity`) は `Decimal('Infinity')` で直接構築できます。また、`DivisionByZero` をトラップせずにゼロで除算を行った場合にも出てきます。同様に、`Overflow` シグナルをトラップしなければ、表現可能な最大の数値の制限を越えた値を丸めたときに出てきます。

無限大には符号があり (アフィン: affine であり)、算術演算に使用でき、非常に巨大で不確定の (indeterminate) 値として扱われます。例えば、無限大に何らかの定数を加算すると、演算結果は別の無限大になります。

演算によっては結果が不確定になるものがあり、NaN を返します。ただし、`InvalidOperation` シグナルをトラップするようになっていれば例外を送出します。例えば、`0/0` は NaN を返します。NaN は「非数値 (not a number)」を表します。このような NaN は暗黙のうちに生成され、一度生成されるとそれを他の計算にも流れてゆき、関係する個々の演算全てが個別の NaN を返すようになります。この挙動は、たまに入力値

が欠けるような状況で一連の計算を行う際に便利です — 特定の計算に対しては無効な結果を示すフラグを立てつつ計算を進められるからです。

一方、NaN の変種である sNaN は関係する全ての演算で演算後にシグナルを送出します。sNaN は、無効な演算結果に対して特別な処理を行うために計算を停止する必要がある場合に便利です。

Python の比較演算は NaN が関わってくると少し驚くようなことがあります。等価性のテストの一方の対象が無言または発信 NaN である場合いつでも `False` を返し (たとえば `Decimal('NaN')==Decimal('NaN')` でも)、一方で不等価をテストするといつでも `True` を返します。二つの `Decimal` を `<`, `<=`, `>` または `>=` を使って比較する試みは一方が NaN である場合には `InvalidOperation` シグナルを送出し、このシグナルをトラップしなければ結果は `False` に終わります。汎用 10 進演算仕様は直接の比較の振る舞いについて定めていないことに注意しておきましょう。ここでの NaN が関係する比較ルールは IEEE 854 標準から持ってきました (section 5.7 の Table 3 を見て下さい)。厳格に標準遵守を貫くなら、`compare()` および `compare-signal()` メソッドを代わりに使いましょう。

アンダフローの起きた計算は、符号付きのゼロ (signed zero) を返すことがあります。符号は、より高い精度で計算を行った結果の符号と同じになります。符号付きゼロの大きさはやはりゼロなので、正のゼロと負のゼロは等しいとみなされ、符号は単なる参考になります。

二つの符号付きゼロが区別されているのに等価であることに加えて、異なる精度におけるゼロの表現はまちまちなのに、値は等価とみなされるということがあります。これに慣れるには多少時間がかかります。正規化浮動小数点表現に目が慣れてしまうと、以下の計算でゼロに等しい値が返っているとは即座に分かりません:

```
>>> 1 / Decimal('Infinity')
Decimal('0E-10000000026')
```

9.4.6 スレッドを使った処理

関数 `getcontext()` は、スレッド毎に別々の `Context` オブジェクトにアクセスします。別のスレッドコンテキストを持つということは、複数のスレッドが互いに影響を及ぼさずに (`getcontext().prec=10` のような) 変更を適用できるということです。

同様に、`setcontext()` 関数は自動的に引数のコンテキストを現在のスレッドのコンテキストに設定します。

`getcontext()` を呼び出す前に `setcontext()` が呼び出されていなければ、現在のスレッドで使うための新たなコンテキストを生成するために `getcontext()` が自動的に呼び出されます。

新たなコンテキストは、`DefaultContext` と呼ばれる雛形からコピーされます。アプリケーションを通じて全てのスレッドに同じ値を使うようにデフォルトを設定したければ、`DefaultContext` オブジェクトを直接変更します。`getcontext()` を呼び出すスレッド間で競合条件が生じないようにするため、`DefaultContext` への変更はいかなるスレッドを開始するよりも 前に行わなければなりません。以下に例を示します:

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
```

(次のページに続く)

(前のページからの続き)

```

DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()

. . .

```

9.4.7 レシピ

`Decimal` クラスの利用を実演している例をいくつか示します。これらはユーティリティ関数としても利用できます:

```

def moneyfmt(value, places=2, curr='', sep=',', dp='.',
             pos='', neg='-', trailneg=''):
    """Convert Decimal to a money formatted string.

places:  required number of places after the decimal point
curr:    optional currency symbol before the sign (may be blank)
sep:     optional grouping separator (comma, period, space, or blank)
dp:      decimal point indicator (comma or period)
         only specify as blank when places is zero
pos:     optional sign for positive numbers: '+', space or blank
neg:     optional sign for negative numbers: '-', '(', space or blank
trailneg: optional trailing minus indicator:  '-', ')', space or blank

    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
    '-$1,234,567.89'
    >>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
    '1.234.568-'
    >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
    '($1,234,567.89)'
    >>> moneyfmt(Decimal(123456789), sep=' ')
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
    '<0.02>'

    """
    q = Decimal(10) ** -places          # 2 places --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = map(str, digits)
    build, next = result.append, digits.pop
    if sign:
        build(trailneg)
    for i in range(places):
        build(next() if digits else '0')

```

(次のページに続く)

(前のページからの続き)

```

build(dp)
if not digits:
    build('0')
i = 0
while digits:
    build(next())
    i += 1
    if i == 3 and digits:
        i = 0
        build(sep)
build(curr)
build(neg if sign else pos)
return ''.join(reversed(result))

def pi():
    """Compute Pi to the current precision.

    >>> print pi()
    3.141592653589793238462643383

    """
    getcontext().prec += 2 # extra digits for intermediate steps
    three = Decimal(3)     # substitute "three=3.0" for regular floats
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2
    return +s                # unary plus applies the new precision

def exp(x):
    """Return e raised to the power of x.  Result type matches input type.

    >>> print exp(Decimal(1))
    2.718281828459045235360287471
    >>> print exp(Decimal(2))
    7.389056098930650227230427461
    >>> print exp(2.0)
    7.38905609893
    >>> print exp(2+0j)
    (7.38905609893+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 1

```

(次のページに続く)

```

        fact *= i
        num *= x
        s += num / fact
    getcontext().prec -= 2
    return +s

def cos(x):
    """Return the cosine of x as measured in radians.

    >>> print cos(Decimal('0.5'))
    0.8775825618903727161162815826
    >>> print cos(0.5)
    0.87758256189
    >>> print cos(0.5+0j)
    (0.87758256189+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

def sin(x):
    """Return the sine of x as measured in radians.

    >>> print sin(Decimal('0.5'))
    0.4794255386042030002732879352
    >>> print sin(0.5)
    0.479425538604
    >>> print sin(0.5+0j)
    (0.479425538604+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

```

9.4.8 Decimal FAQ

Q. `decimal.Decimal('1234.5')` などと打ち込むのは煩わしいのですが、対話式インタプリタを使う際にタイプ量を少なくする方法はありませんか？

A. コンストラクタを 1 文字に縮める人もいます:

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

Q. 小数点以下 2 桁の固定小数点数のアプリケーションの中で、いくつかの入力が余計な桁を保持しているのでこれを丸めなければなりません。その他のものに余計な桁はなくそのまま使えます。どのメソッドを使うのがいいでしょうか？

A. `quantize()` メソッドで固定した桁に丸められます。 *Inexact* トラップを設定しておけば、確認にも有用です:

```
>>> TWOPLACES = Decimal(10) ** -2          # same as Decimal('0.01')
```

```
>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')
```

```
>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')
```

```
>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
...
Inexact: None
```

Q. 正当な 2 桁の入力が得られたとして、その正当性をアプリケーション実行中も変わらず保ち続けるにはどうすればいいでしょうか？

A. 加減算あるいは整数との乗算のような演算は自動的に固定小数点を守ります。その他の除算や整数以外の乗算などは小数点以下の桁を変えてしまいますので実行後は `quantize()` ステップが必要です:

```
>>> a = Decimal('102.72')          # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                          # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42                         # So does integer multiplication
Decimal('4314.24')
>>> (a * b).quantize(TWOPLACES)    # Must quantize non-integer multiplication
Decimal('325.62')
```

(次のページに続く)

(前のページからの続き)

```
>>> (b / a).quantize(TWOPLACES)      # And quantize division
Decimal('0.03')
```

固定小数点のアプリケーションを開発する際は、`quantize()` の段階を扱う関数を定義しておくとう便利です:

```
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)
```

```
>>> mul(a, b)                                # Automatically preserve fixed-point
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')
```

Q. 一つの値に対して多くの表現方法があります。200 と 200.000 と 2E2 と 02E+4 は全て同じ値で違った精度の数です。これらをただ一つの正規化された値に変換することはできますか?

A. `normalize()` メソッドは全ての等しい値をただ一つの表現に直します:

```
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

Q. ある種の 10 進数値はいつも指数表記で表示されます。指数表記以外の表示にする方法がありますか?

A. 値によっては、指数表記だけが有効桁数を表せる表記法なのです。たとえば、5.0E+3 を 5000 と表してしまうと、値は変わりませんが元々の 2 桁という有効数字が反映されません。

もしアプリケーションが有効数字の追跡を等閑視するならば、指数部や末尾のゼロを取り除き、有効数字を忘れ、しかし値を変えずに置くことは容易です:

```
def remove_exponent(d):
    '''Remove exponent and trailing zeros.

    >>> remove_exponent(Decimal('5E+3'))
    Decimal('5000')

    '''
    return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()
```

Q. 普通の float を `Decimal` に変換できますか?

A. はい。どんな 2 進浮動小数点数も `Decimal` として正確に表現できます。ただし、正確な変換は直感的に考えたよりも多い桁になることがあります:

```
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

Q. 複雑な計算の中で、精度不足や丸めの異常で間違った結果になっていないことをどうやって保証すれば良いでしょうか。

A. decimal モジュールでは検算は容易です。一番良い方法は、大きめの精度や様々な丸めモードで再計算してみることです。大きく異なった結果が出てきたら、精度不足や丸めの問題や悪条件の入力、または数値計算的に不安定なアルゴリズムを示唆しています。

Q. コンテキストの精度は計算結果には適用されていますが入力には適用されていないようです。様々な異なる精度の入力値を混ぜて計算する時に注意すべきことはありますか？

A. はい。原則として入力値は正確であると見做しておりそれらの値を使った計算も同様です。結果だけが丸められます。入力の強みは "what you type is what you get" (打ち込んだ値が得られる値) という点にあります。入力が丸められないということを忘れていると結果が奇妙に見えるというのは弱点です：

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

解決策は、精度を増やすか、単項プラス演算子を使って入力の丸めを強制することです：

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')           # unary plus triggers rounding
Decimal('1.23')
```

もしくは、入力を `Context.create_decimal()` を使って生成時に丸めてしまうこともできます：

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

9.5 fractions — 有理数

バージョン 2.6 で追加.

ソースコード: [Lib/fractions.py](#)

`fractions` モジュールは有理数計算のサポートを提供します。

Fraction インスタンスは一对の整数から、他の有理数から、または文字列から組み立てられます。

```
class fractions.Fraction (numerator=0, denominator=1)
class fractions.Fraction (other_fraction)
class fractions.Fraction (float)
class fractions.Fraction (decimal)
class fractions.Fraction (string)
```

最初のバージョンは `numerator` と `denominator` が `numbers.Rational` のインスタンスであるこ

とを要求し、`numerator/denominator` の値を持つ新しい *Fraction* インスタンスを返します。`denominator` が 0 ならば、*ZeroDivisionError* を送出します。二番目のバージョンは *other-fraction* が *numbers.Rational* のインスタンスであることを要求し、同じ値を持つ新しい *Fraction* インスタンスを返します。その次の二つのバージョンは、*float* と *decimal.Decimal* インスタンスを受け付け、それとちょうど同じ値を持つ *Fraction* インスタンスを返します。なお、二進浮動小数点数にお決まりの問題 (tut-fp-issues 参照) のため、`Fraction(1.1)` の引数は `11/10` と正確に等しいとは言えないので、`Fraction(1.1)` は予期した通りの `Fraction(11, 10)` を返しません。(しかし *limit_denominator()* メソッドのドキュメントを参照してください。)最後のバージョンは、文字列またはユニコードのインスタンスを渡されるとしています。一つめの形式の通常の形式は:

```
[sign] numerator ['/' denominator]
```

で、ここにオプションの `sign` は '+' か '-' のどちらかであり、`numerator` および `denominator` は (もしあるならば) 十進数の数字の並びです。さらに、*float* コンストラクタで受け付けられる、有限の値を表す文字列は、必ず *Fraction* コンストラクタでも受け付けられます。どちらの形式でも入力される文字列は前後に空白がなくて構いません。例を見ましょう:

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)
```

Fraction クラスは抽象基底クラス *numbers.Rational* を継承し、その全てのメソッドと演算を実装します。*Fraction* インスタンスはハッシュ可能で、不変 (immutable) であるものとして扱われます。加えて、*Fraction* には以下のメソッドがあります:

バージョン 2.7 で変更: *Fraction* のコンストラクタが *float* および *decimal.Decimal* インスタンスを受け付けるようになりました。

from_float (flt)

このクラスメソッドは `float` である `flt` の正確な値を表す `Fraction` を構築します。`Fraction.from_float(0.3)` と `Fraction(3, 10)` の値は同じでないことに注意してください。

注釈: Python 2.7 以降では、`float` から直接 `Fraction` インスタンスを構築できるようになりました。

`from_decimal(dec)`

このクラスメソッドは `decimal.Decimal` インスタンスである `dec` の正確な値を表す `Fraction` を構築します。

注釈: Python 2.7 以降では、`decimal.Decimal` インスタンスから直接 `Fraction` インスタンスを構築できるようになりました。

`limit_denominator(max_denominator=1000000)`

分母が高々 `max_denominator` である、`self` に最も近い `Fraction` を見付けて返します。このメソッドは与えられた浮動小数点数の有理数近似を見つけるのに役立ちます:

```
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

あるいは `float` で表された有理数を元に戻すのにも使えます:

```
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
Fraction(4503599627370497, 9007199254740992)
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
>>> Fraction(1.1).limit_denominator()
Fraction(11, 10)
```

`fractions.gcd(a, b)`

整数 a と b の最大公約数を返します。 a も b もゼロでないとする、`gcd(a, b)` の絶対値は a と b の両方を割り切る最も大きな整数です。`gcd(a, b)` は b がゼロでなければ b と同じ符号になります。そうでなければ a の符号を取ります。`gcd(0, 0)` は 0 を返します。

参考:

`numbers` モジュール 数値の塔を作り上げる抽象基底クラス。

9.6 random — 擬似乱数を生成する

ソースコード: [Lib/random.py](#)

このモジュールでは様々な分布をもつ擬似乱数生成器を実装しています。

整数用では、ある値域内の数の選択を一様にします。シーケンス用には、シーケンスからのランダムな要素の一樣な選択、リストの要素の順列をランダムに置き換える関数、順列を入れ替えずにランダムに取り出す関数があります。

実数用としては、一樣分布、正規分布 (ガウス分布)、対数正規分布、負の指数分布、ガンマおよびベータ分布を計算する関数があります。角度分布の生成用には、フォンミーゼス分布が利用できます。

ほとんど全てのモジュール関数は、基礎となる関数 `random()` に依存します。この関数はランダムな浮動小数点数を半開区間 $[0.0, 1.0)$ 内に一樣に生成します。Python は中心となる乱数生成器としてメルセンヌツイスタを使います。これは 53 ビット精度の浮動小数点を生成し、周期は $2^{19937}-1$ です、本体は C で実装されていて、高速でスレッドセーフです。メルセンヌツイスタは、現存する中で、最も広範囲にテストされた乱数生成器のひとつです。しかし、完全に決定論的であるため、この乱数生成器は全ての目的に合致しているわけではなく、暗号化の目的には全く向いていません。

このモジュールで提供されている関数は、実際には `random.Random` クラスの隠蔽されたインスタンスのメソッドにバインドされています。内部状態を共有しない生成器を取得するため、自分で `Random` のインスタンスを生成することができます。異なる `Random` のインスタンスを各スレッド毎に生成し、`jumpahead()` メソッドを使うことで各々のスレッドにおいて生成された乱数列ができるだけ重複しないようにすれば、マルチスレッドプログラムを作成する上で特に便利になります。

自分で考案した基本乱数生成器を使いたい場合、クラス `Random` をサブクラス化することもできます。この場合、メソッド `random()`、`seed()`、`getstate()`、`setstate()`、`jumpahead()` をオーバーライドしてください。オプションとして、新しいジェネレータは `getrandbits()` メソッドを提供することができます。これにより `randrange()` メソッドが任意に大きな範囲から選択を行えるようになります。

バージョン 2.4 で追加: `getrandbits()` メソッド。

サブクラス化の例として、`random` モジュールは `WichmannHill` クラスを提供します。このクラスは Python だけで書かれた代替生成器を実装しています。このクラスは、乱数生成器に Wichmann-Hill 法を使っていた古いバージョンの Python から得られた結果を再現するための、後方互換の手段になります。ただし、この Wichmann-Hill 生成器はもはや推奨することができないということに注意してください。現在の水準では生成される周期が短すぎ、また厳密な乱数性試験に合格しないことが知られています。こうした欠点を修正した最近の改良についてはページの最後に挙げた参考文献を参照してください。

バージョン 2.3 で変更: メルセンヌツイスタが Wichmann-Hill の代わりにデフォルト生成器になりました。

`random` モジュールは `SystemRandom` クラスも提供していて、このクラスは OS が提供している乱数発生源を利用して乱数を生成するシステム関数 `os.urandom()` を使うものです。

警告: このモジュールの擬似乱数生成器はセキュリティ上の手段に使うべきではありません。暗号的にセキュアな疑似乱数生成器が必要ななら、`os.urandom()` や `SystemRandom` を使ってください。

保守 (bookkeeping) 関数:

`random.seed(a=None)`

乱数生成器の内部状態を初期化します。

引数が `None` または無い場合は、現在時刻もしくは利用可能ならオペレーティングシステム固有の乱数発生源をシードとします (利用可能かどうかについての詳細は `os.urandom()` 関数を参照してください)。

`a` が `None` や `int` や `long` でない場合は `hash(a)` を代わりに使います。PYTHONHASHSEED が有効なときは、ある型のハッシュ値は非決定的であることに注意してください。

バージョン 2.4 で変更: 以前は、オペレーティングシステムのリソースは使われませんでした。

`random.getstate()`

乱数生成器の現在の内部状態を記憶したオブジェクトを返します。このオブジェクトを `setstate()` に渡して内部状態を復帰することができます。

バージョン 2.1 で追加.

バージョン 2.6 で変更: Python 2.6 が作り出す状態オブジェクトは以前のバージョンには読み込めません。

`random.setstate(state)`

`state` は予め `getstate()` を呼び出して得ておかなくてはなりません。 `setstate()` は `getstate()` が呼び出された時の乱数生成器の内部状態を復帰します。

バージョン 2.1 で追加.

`random.jumpahead(n)`

内部状態を、現在の状態から、非常に離れているであろう状態に変更します。 `n` は非負の整数です。これはマルチスレッドのプログラムが複数の `Random` クラスのインスタンスと結合されている場合に非常に便利です: `setstate()` や `seed()` は全てのインスタンスを同じ内部状態にするのに使うことができ、その後 `jumpahead()` を使って各インスタンスの内部状態を引き離すことができます。

バージョン 2.1 で追加.

バージョン 2.3 で変更: `n` ステップ先の特定の状態になるのではなく、 `jumpahead(n)` は何ステップも離れているであろう別の状態にする。

`random.getrandbits(k)`

`k` ビット分の乱数ビットを納めた Python の `long` 整数を返します。このメソッドは MersenneTwister 生成器で提供されており、その他の乱数生成器でもオプションの API として提供されているかもしれません。このメソッドが使えるとき、 `randrange()` メソッドは大きな範囲を扱えるようになります。

バージョン 2.4 で追加.

整数用の関数:

`random.randrange(stop)`

`random.randrange(start, stop[, step])`

`range(start, stop, step)` の要素からランダムに選ばれた要素を返します。この関数は

`choice(range(start, stop, step))` と等価ですが、実際には `range` オブジェクトを生成しません。

バージョン 1.5.2 で追加.

`random.randint(a, b)`

$a \leq N \leq b$ であるようなランダムな整数 N を返します。

シーケンス用の関数:

`random.choice(seq)`

空でないシーケンス `seq` からランダムに要素を返します。 `seq` が空のときは、`IndexError` が送出されます。

`random.shuffle(x[, random])`

シーケンス `x` をインプレースにシャッフルします。オプションの引数 `random` は、引数を持たず、`[0.0, 1.0)` のランダムな浮動小数点数を返すような関数です。デフォルトでは、これは関数 `random()` です。

やや小さい `len(x)` であっても、`x` の順列の総数はほとんどの乱数生成器の周期よりも大きくなるので注意してください; このことは長いシーケンスに対してはほとんどの順列は生成されないことを意味します。

`random.sample(population, k)`

母集団のシーケンスから選ばれた長さ k の一意な要素からなるリストを返します。値の置換を行わないランダムサンプリングに用いられます。

バージョン 2.3 で追加.

母集団自体を変更せずに、母集団内の要素を含む新たなリストを返します。返されたリストは選択された順に並んでいるので、このリストの部分スライスもランダムなサンプルになります。これにより、くじの当選者 (サンプル) を 1 等賞と 2 等賞 (の部分スライス) に分けることも可能です。

母集団の要素はハッシュ可能 (*hashable*) でなくても、ユニークでなくても、かまいません。母集団が繰り返しを含む場合、出現するそれぞれがサンプルに選択されえます。

整数の並びからサンプルを選ぶには、引数に `xrange()` オブジェクトを使いましょう。特に、巨大な母集団からサンプルを取るとき、速度と空間効率が上がります。 `sample(xrange(10000000), 60)`

以下の関数は特定の実数値分布を生成します。関数の引数の名前は、一般的な数学の慣例で使われている分布の公式の対応する変数から取られています; これらの公式のほとんどはどんな統計学のテキストにも載っています。

`random.random()`

値域 `[0.0, 1.0)` の次のランダムな浮動小数点数を返します。

`random.uniform(a, b)`

$a \leq b$ であれば $a \leq N \leq b$ 、 $b < a$ であれば $b \leq N \leq a$ であるようなランダムな浮動小数点数 N を返します。

端点の値 b が範囲に含まれるかどうかは、等式 $a + (b-a) * \text{random}()$ における浮動小数点の丸めに依存します。

`random.triangular(low, high, mode)`

$low \leq N \leq high$ でありこれら境界値の間に指定された最頻値 *mode* を持つようなランダムな浮動小数点数 N を返します。境界 *low* と *high* のデフォルトは 0 と 1 です。最頻値 *mode* 引数のデフォルトは両境界値の中点になり、対称な分布を与えます。

バージョン 2.6 で追加。

`random.betavariate(alpha, beta)`

ベータ分布です。引数の満たすべき条件は $\alpha > 0$ および $\beta > 0$ です。範囲 0 から 1 の値を返します。

`random.expovariate(lambd)`

指数分布です。*lambd* は平均にしたい値の逆数です。(この引数は "lambda" と呼ぶべきなのですが、Python の予約語なので使えません。) 返される値の範囲は *lambd* が正なら 0 から正の無限大、*lambd* が負なら負の無限大から 0 です。

`random.gammavariate(alpha, beta)`

ガンマ分布です。(ガンマ関数 ではありません！) 引数の満たすべき条件は $\alpha > 0$ および $\beta > 0$ です。

確率分布関数は:

$$\text{pdf}(x) = \frac{x^{(\alpha - 1)} * \text{math.exp}(-x / \beta)}{\text{math.gamma}(\alpha) * \beta^{**} \alpha}$$

`random.gauss(mu, sigma)`

ガウス分布です。*mu* は平均であり、*sigma* は標準偏差です。この関数は後で定義する関数 `normalvariate()` より少しだけ高速です。

`random.lognormvariate(mu, sigma)`

対数正規分布です。この分布を自然対数を用いた分布にした場合、平均 *mu* で標準偏差 *sigma* の正規分布になります。*mu* は任意の値を取ることができ、*sigma* はゼロより大きくなければなりません。

`random.normalvariate(mu, sigma)`

正規分布です。*mu* は平均で、*sigma* は標準偏差です。

`random.vonmisesvariate(mu, kappa)`

mu は平均の角度で、0 から 2π までのラジアンで表されます。*kappa* は濃度パラメタで、ゼロ以上でなければなりません。*kappa* がゼロに等しい場合、この分布は範囲 0 から 2π の一様でランダムな角度の分布に退化します。

`random.paretovariate(alpha)`

パレート分布です。*alpha* は形状パラメタです。

`random.weibullvariate(alpha, beta)`

ワイブル分布です。*alpha* は尺度パラメタで、*beta* は形状パラメタです。

代替の乱数生成器:

class random.WichmannHill([seed])

乱数生成器として Wichmann-Hill アルゴリズムを実装するクラスです。Random クラスと同じメソッド全てと、下で説明する `whseed()` メソッドを持ちます。このクラスは、Python だけで実装されているので、スレッドセーフではなく、呼び出しと呼び出しの間にロックが必要です。また、周期が 6,953,607,871,644 と短く、独立した 2 つの乱数列が重複しないように注意が必要です。

random.whseed([x])

これは obsolete で、バージョン 2.1 以前の Python と、ビット・レベルの互換性のために提供されています。詳細は `seed()` を参照してください。 `whseed()` は、引数に与えた整数が異なっても、内部状態が異なることを保障しません。取り得る内部状態の個数が 2^{24} 以下になる場合もあります。

class random.SystemRandom([seed])

オペレーティングシステムの提供する発生源によって乱数を生成する `os.urandom()` 関数を使うクラスです。すべてのシステムで使えるメソッドではありません。ソフトウェアの状態に依存してはいけませんし、一連の操作は再現不能です。それに応じて、`seed()` と `jumpahead()` メソッドは何の影響も及ぼさず、無視されます。 `getstate()` と `setstate()` メソッドが呼び出されると、例外 `NotImplementedError` が送出されます。

バージョン 2.4 で追加。

基本使用例:

```
>>> random.random()           # Random float x, 0.0 <= x < 1.0
0.37444887175646646
>>> random.uniform(1, 10)     # Random float x, 1.0 <= x < 10.0
1.1800146073117523
>>> random.randint(1, 10)     # Integer from 1 to 10, endpoints included
7
>>> random.randrange(0, 101, 2) # Even integer from 0 to 100
26
>>> random.choice('abcdefghij') # Choose a random element
'c'

>>> items = [1, 2, 3, 4, 5, 6, 7]
>>> random.shuffle(items)
>>> items
[7, 3, 2, 5, 6, 4, 1]

>>> random.sample([1, 2, 3, 4, 5], 3) # Choose 3 elements
[4, 1, 5]
```

参考:

M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3–30 1998.

Wichmann, B. A. & Hill, I. D., "Algorithm AS 183: An efficient and portable pseudo-random number generator", Applied Statistics 31 (1982) 188-190.

Complementary-Multiply-with-Carry recipe 長い周期と比較的シンプルな更新操作を備えた互換性のある別の乱数生成器。

9.7 itertools — 効率的なループ実行のためのイテレータ生成関数

バージョン 2.3 で追加.

このモジュールではイテレータ (*iterator*) を構築する部品を実装しています。プログラム言語 APL, Haskell, SML からアイデアを得ていますが、Python に適した形に修正されています。

このモジュールは、高速でメモリ効率に優れ、単独でも組合せても使用することのできるツールを標準化したものです。同時に、このツール群は ”イテレータの代数” を構成していて、pure Python で簡潔かつ効率的なツールを作れるようにしています。

例えば、SML の作表ツール `tabulate(f)` は `f(0)`, `f(1)`, ... のシーケンスを作成します。同じことを Python では `imap()` と `count()` を組合せて `imap(f, count())` という形で実現できます。

これらのツールと組み込み関数は `operator` モジュール内の高速な関数とともに使うことで見事に動作します。例えば、乗算演算子を 2 つのベクタにまたがってマップして効率的な内積計算が実現できます: `sum(imap(operator.mul, vector1, vector2))`。

無限イテレータ:

イテレータ	引数	結果	例
<code>count()</code>	<code>start</code> , <code>[step]</code>	<code>start</code> , <code>start+step</code> , <code>start+2*step</code> , ...	<code>count(10) --> 10 11 12 13 14</code> ...
<code>cycle()</code>	<code>p</code>	<code>p0</code> , <code>p1</code> , ... <code>plast</code> , <code>p0</code> , <code>p1</code> , ...	<code>cycle('ABCD') --> A B C D A B C D</code> ...
<code>repeat()</code>	<code>elem [n]</code>	<code>elem</code> , <code>elem</code> , <code>elem</code> , ... 無限もしくは <code>n</code> 回	<code>repeat(10, 3) --> 10 10 10</code>

一番短い入力シーケンスで止まるイテレータ:

イテレータ	引数	結果	例
<code>chain()</code>	p, q, ...	p0, p1, ... plast, q0, q1, ...	<code>chain('ABC', 'DEF')</code> --> A B C D E F
<code>compress()</code>	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ...	<code>compress('ABCDEF', [1,0,1,0,1,1])</code> --> A C E F
<code>dropwhile()</code>	(pred, seq	seq[n], seq[n+1], pred が偽の場所から始まる	<code>dropwhile(lambda x: x<5, [1,4,6,4,1])</code> --> 6 4 1
<code>groupby()</code>	iterable[, keyfunc]	keyfunc(v) の値でグループ化したサブイテレータ	
<code>ifilter()</code>	pred, seq	pred(elem) が真になる seq の要素	<code>ifilter(lambda x: x%2, range(10))</code> --> 1 3 5 7 9
<code>ifilterfalse()</code>	pred, seq	pred(elem) が偽になる seq の要素	<code>ifilterfalse(lambda x: x%2, range(10))</code> --> 0 2 4 6 8
<code>islice()</code>	seq, [start, stop [, step]	seq[start:stop:step]	<code>islice('ABCDEFG', 2, None)</code> --> C D E F G
<code>imap()</code>	func, p, q, ...	func(p0, q0), func(p1, q1), ...	<code>imap(pow, (2,3,10), (5,2,3))</code> --> 32 9 1000
<code>starmap()</code>	func, seq	func(*seq[0]), func(*seq[1]), ...	<code>starmap(pow, [(2,5), (3,2), (10,3)])</code> --> 32 9 1000
<code>tee()</code>	it, n	it1, it2, ... itn 一つのイテレータを n 個に分ける	
<code>takewhile()</code>	(pred, seq	seq[0], seq[1], pred が偽になるまで	<code>takewhile(lambda x: x<5, [1,4,6,4,1])</code> --> 1 4
<code>izip()</code>	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	<code>izip('ABCD', 'xy')</code> --> Ax By
<code>izip_longest()</code>	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	<code>izip_longest('ABCD', 'xy', fillvalue='-')</code> --> Ax By C- D-

組合せジェネレータ:

イテレータ	引数	結果
<code>product()</code>	<code>p, q, ... [repeat=1]</code>	デカルト積、ネストした for ループと等価
<code>permutations()</code>	<code>p[, r]</code>	長さ <code>r</code> のタプル列, 繰り返しを許さない順列
<code>combinations()</code>	<code>p, r</code>	長さ <code>r</code> のタプル列, 繰り返しを許さない組合せ
<code>combinations_with_replacement()</code>	<code>p, r</code>	長さ <code>r</code> のタプル列, 繰り返しを許した組合せ
<code>product('ABCD', repeat=2)</code>		AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
<code>permutations('ABCD', 2)</code>		AB AC AD BA BC BD CA CB CD DA DB DC
<code>combinations('ABCD', 2)</code>		AB AC AD BC BD CD
<code>combinations_with_replacement('ABCD', 2)</code>		AA AB AC AD BB BC BD CC CD DD

9.7.1 Itertool 関数

以下の関数は全て、イテレータを作成して返します。無限長のストリームのイテレータを返す関数もあり、この場合にはストリームを中断するような関数がループ処理から使用しなければなりません。

`itertools.chain(*iterables)`

先頭の `iterable` の全要素を返し、次に 2 番目の `iterable` の全要素を返し、と全 `iterable` の要素を返すイテレータを作成します。連続したシーケンスを一つのシーケンスとして扱う場合に使用します。およそ次と等価です:

```
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

`classmethod chain.from_iterable(iterable)`

`chain()` のためのもう一つのコンストラクタです。遅延評価される唯一のイテラブル引数から連鎖した入力を受け取ります。この関数は、以下のコードとほぼ等価です:

```
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

バージョン 2.6 で追加.

`itertools.combinations(iterable, r)`

入力 `iterable` の要素からなる長さ `r` の部分列を返します。

組合せ (combination) は辞書式順序で出力されます。したがって、入力 `iterable` がソートされていれば、

組合せのタプルは整列された形で生成されます。

各要素は場所に基づいて一意に取り扱われ、値には依りません。入力された要素がバラバラならば、各組合せの中に重複した値は現れません。

およそ次と等価です:

```
def combinations(iterable, r):
    # combinations('ABCD', 2) --> AB AC AD BC BD CD
    # combinations(range(4), 3) --> 012 013 023 123
    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = range(r)
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

`combinations()` のコードは `permutations()` のシーケンスから (入力プールでの位置に応じた順序で) 要素がソートされていないものをフィルターしたようにも表現できます:

```
def combinations(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in permutations(range(n), r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

返される要素の数は、 $0 \leq r \leq n$ の場合は、 $n! / r! / (n-r)!$ で、 $r > n$ の場合は 0 です。

バージョン 2.6 で追加.

`itertools.combinations_with_replacement(iterable, r)`

入力 *iterable* から、それぞれの要素が複数回現れることを許して、長さ *r* の要素の部分列を返します。

組合せ (combination) は辞書式順序で出力されます。したがって、入力 *iterable* がソートされていれば、組合せのタプルは整列された形で生成されます。

要素は、値ではなく位置に基づいて一意に扱われます。ですから、入力の要素が一意であれば、生成された組合せも一意になります。

およそ次と等価です:

```
def combinations_with_replacement(iterable, r):
    # combinations_with_replacement('ABC', 2) --> AA AB AC BB BC CC
    pool = tuple(iterable)
    n = len(pool)
    if not n and r:
        return
    indices = [0] * r
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != n - 1:
                break
        else:
            return
        indices[i:] = [indices[i] + 1] * (r - i)
        yield tuple(pool[i] for i in indices)
```

`combinations_with_replacement()` のコードは、`product()` の部分列から、要素が (入力プールの位置に従って) ソートされた順になっていない項目をフィルタリングしたものとしても表せます:

```
def combinations_with_replacement(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in product(range(n), repeat=r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

返される要素の数は、 $n > 0$ のとき $(n+r-1)! / r! / (n-1)!$ です。

バージョン 2.7 で追加.

`itertools.compress(data, selectors)`

`data` の要素から `selectors` の対応する要素が `True` と評価されるものだけをフィルタしたイテレータを作ります。 `data` と `selectors` のいずれかが尽きたときに止まります。およそ次と等価です:

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
    return (d for d, s in izip(data, selectors) if s)
```

バージョン 2.7 で追加.

`itertools.count(start=0, step=1)`

数値 n で始まる、等間隔の値を返すイテレータを作成します。 `imap()` で連続したデータの生成によく使われます。また、 `izip()` にシーケンス番号を追加するのにも使われます。この関数は以下のスクリプトと同等です:

```
def count(start=0, step=1):
    # count(10) --> 10 11 12 13 14 ...
    # count(2.5, 0.5) -> 2.5 3.0 3.5 ...
```

(次のページに続く)

(前のページからの続き)

```
n = start
while True:
    yield n
    n += step
```

浮動小数点数で数えるときは、`(start + step * i for i in count())` のように、掛け算を使ったコードに置き換えたほうが正確にできることがあります。

バージョン 2.7 で変更: `step` 引数が追加され、非整数の引数が可能になりました。

`itertools.cycle(iterable)`

イテラブルから要素を取得し、そのコピーを保存するイテレータを作成します。イテラブルの全要素を返すと、セーブされたコピーから要素を返します。これを無限に繰り返します。およそ次と等価です:

```
def cycle(iterable):
    # cycle('ABCD') --> A B C D A B C D A B C D ...
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
    while saved:
        for element in saved:
            yield element
```

`cycle()` は大きなメモリ領域を使用します。使用するメモリ量は `iterable` の大きさに依存します。

`itertools.dropwhile(predicate, iterable)`

`predicate` (述語) が真である間は要素を飛ばし、その後は全ての要素を返すイテレータを作成します。このイテレータは、`predicate` が最初に偽になるまで 全く 要素を返さないため、要素を返し始めるまでに長い時間がかかる場合があります。およそ次と等価です:

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

`itertools.groupby(iterable[, key])`

同じキーをもつような要素からなる `iterable` 中のグループに対して、キーとグループを返すようなイテレータを作成します。`key` は各要素に対するキー値を計算する関数です。キーを指定しない場合や `None` にした場合、`key` 関数のデフォルトは恒等関数になり要素をそのまま返します。通常、`iterable` は同じキー関数で並べ替え済みである必要があります。

`groupby()` の操作は Unix の `uniq` フィルターと似ています。 `key` 関数の値が変わるたびに休止または新しいグループを生成します (このために通常同じ `key` 関数でソートしておく必要があります)。こ

の動作は SQL の入力順に関係なく共通の要素を集約する GROUP BY とは違います。

返されるグループはそれ自体がイテレータで、`groupby()` と `iterable` を共有しています。もともとなる `iterable` を共有しているため、`groupby()` オブジェクトの要素取り出しを先に進めると、それ以前の要素であるグループは見えなくなってしまう。従って、データが後で必要な場合にはリストの形で保存しておく必要があります：

```
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))      # Store group iterator as a list
    uniquekeys.append(k)
```

`groupby()` はおよそ次と等価です：

```
class groupby(object):
    # [k for k, g in groupby('AAAABBBCCDAABBB')] --> A B C D A B
    # [list(g) for k, g in groupby('AAAABBBCCD')] --> AAAA BBB CC D
    def __init__(self, iterable, key=None):
        if key is None:
            key = lambda x: x
        self.keyfunc = key
        self.it = iter(iterable)
        self.tgtkey = self.currkey = self.currvalue = object()
    def __iter__(self):
        return self
    def next(self):
        while self.currkey == self.tgtkey:
            self.currvalue = next(self.it)    # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)
            self.tgtkey = self.currkey
        return (self.currkey, self._grouper(self.tgtkey))
    def _grouper(self, tgtkey):
        while self.currkey == tgtkey:
            yield self.currvalue
            self.currvalue = next(self.it)    # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)
```

バージョン 2.4 で追加。

`itertools.ifilter(predicate, iterable)`

`iterable` から `predicate` が `True` となる要素だけを返すイテレータを作成します。`predicate` が `None` の場合、真の要素だけを返します。およそ次と等価です：

```
def ifilter(predicate, iterable):
    # ifilter(lambda x: x%2, range(10)) --> 1 3 5 7 9
    if predicate is None:
        predicate = bool
    for x in iterable:
        if predicate(x):
            yield x
```

`itertools.ifilterfalse` (*predicate, iterable*)

`iterable` から `predicate` が `False` となる要素だけを返すイテレータを作成します。`predicate` が `None` の場合、偽の要素だけを返します。およそ次と等価です:

```
def ifilterfalse(predicate, iterable):
    # ifilterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
    if predicate is None:
        predicate = bool
    for x in iterable:
        if not predicate(x):
            yield x
```

`itertools.imap` (*function, *iterables*)

`iterables` の各要素を引数として `function` を呼び出すイテレータを作成します。`function` が `None` の場合、`imap()` は引数のタプルを返します。`map()` に似ていますが、最短の `iterable` の末尾まで到達した後はそこで終了します。短い `iterable` のところを `None` で補ったりはしません。この違いは、`map()` に無限長のイテレータを指定するのはたいていは誤り (全出力が評価されてしまうため) なのですが、`imap()` の場合には一般的で役に立つ方法であるためです。この関数はおよそ以下のスクリプトと同等です:

```
def imap(function, *iterables):
    # imap(pow, (2,3,10), (5,2,3)) --> 32 9 1000
    iterables = map(iter, iterables)
    while True:
        args = [next(it) for it in iterables]
        if function is None:
            yield tuple(args)
        else:
            yield function(*args)
```

`itertools.islice` (*iterable, stop*)

`itertools.islice` (*iterable, start, stop[, step]*)

`iterable` から要素を選択して返すイテレータを作成します。`start` が 0 でない場合、`iterable` の要素は `start` に達するまでスキップされます。その後、要素が順に返されます。

```
def islice(iterable, *args):
    # islice('ABCDEFGH', 2) --> A B
    # islice('ABCDEFGH', 2, 4) --> C D
    # islice('ABCDEFGH', 2, None) --> C D E F G
    # islice('ABCDEFGH', 0, None, 2) --> A C E G
    s = slice(*args)
    start, stop, step = s.start or 0, s.stop or sys.maxint, s.step or 1
    it = iter(xrange(start, stop, step))
    try:
        nexti = next(it)
    except StopIteration:
        # Consume *iterable* up to the *start* position.
        for i, element in izip(xrange(start), iterable):
            pass
    return
```

(次のページに続く)

(前のページからの続き)

```

try:
    for i, element in enumerate(iterable):
        if i == nexti:
            yield element
            nexti = next(it)
except StopIteration:
    # Consume to *stop*.
    for i, element in izip(xrange(i + 1, stop), iterable):
        pass

```

`start` が `None` ならば、繰返しは 0 から始まります。`step` が `None` ならば、ステップは 1 となります。

バージョン 2.5 で変更: `start` と `step` のデフォルト値として `None` を受け付けるようにしました。

`itertools.izip(*iterables)`

各 `iterable` の要素をまとめるイテレータを作成します。`zip()` に似ていますが、リストではなくイテレータを返します。複数のイテレート可能オブジェクトに対して、一度に lock-step (横並び) イテレーションを行う場合に使用します。この関数はおよそ以下のスクリプトと同等です:

```

def izip(*iterables):
    # izip('ABCD', 'xy') --> Ax By
    iterators = map(iter, iterables)
    while iterators:
        yield tuple(map(next, iterators))

```

バージョン 2.4 で変更: イテレート可能オブジェクトを指定しない場合、`TypeError` 例外を送出する代わりに長さゼロのイテレータを返します。

イテラブルの左から右への評価順序が保証されます。そのため `izip(*[iter(s)]*n)` を使ってデータ系列を `n` 長のグループにクラスタリングできます。

`izip()` は、等しくない長さの入力に対しては、長い方のイテラブルの、終端の対にならない値を気にしないのであれば、使うべきではありません。そのような値が重要なら、かわりに `izip_longest()` を使ってください。

`itertools.izip_longest(*iterables[, fillvalue])`

各 `iterable` の要素をまとめるイテレータを作成します。`iterable` の長さが違う場合、足りない値は `fillvalue` で埋められます。最も長い `iterable` が尽きるまでイテレーションします。およそ次と等価です:

```

class ZipExhausted(Exception):
    pass

def izip_longest(*args, **kwargs):
    # izip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-
    fillvalue = kwargs.get('fillvalue')
    counter = [len(args) - 1]
    def sentinel():
        if not counter[0]:
            raise ZipExhausted

```

(次のページに続く)

(前のページからの続き)

```

    counter[0] -= 1
    yield fillvalue
    fillers = repeat(fillvalue)
    iterators = [chain(it, sentinel()), fillers) for it in args]
    try:
        while iterators:
            yield tuple(map(next, iterators))
    except ZipExhausted:
        pass

```

iterables の 1 つが無限になりうる場合 `izip.longest()` は呼び出し回数を制限するような何かでラップしなければいけません (例えば `islice()` or `takewhile()`)。 `fillvalue` は指定しない場合のデフォルトは `None` です。

バージョン 2.6 で追加.

`itertools.permutations(iterable[, r])`

`iterable` の要素からなる長さ `r` の置換 (permutation) を次々と返します。

`r` が指定されないかまたは `None` であるならば、`r` のデフォルトは `iterable` の長さとなり全ての可能な最長の置換が生成されます。

置換は辞書式にソートされた順序で吐き出されます。したがって入力 `iterable` がソートされていたならば、置換のタプルはソートされた状態で出力されます。

要素は位置に基づいて一意的に扱われ、値に基づいてではありません。したがって入力された要素が全て異なっているならば、それぞれの置換に重複した要素が現れないことになります。

およそ次と等価です:

```

def permutations(iterable, r=None):
    # permutations('ABCD', 2) --> AB AC AD BA BC BD CA CB CD DA DB DC
    # permutations(range(3)) --> 012 021 102 120 201 210
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return
    indices = range(n)
    cycles = range(n, n-r, -1)
    yield tuple(pool[i] for i in indices[:r])
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
            yield tuple(pool[i] for i in indices[:r])

```

(次のページに続く)

(前のページからの続き)

```

        break
    else:
        return

```

`permutations()` のコードは `product()` の列から重複のあるもの (それらは入力プールの同じ位置から取られたものです) を除外するようにフィルタを掛けたものとしても表現できます:

```

def permutations(iterable, r=None):
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    for indices in product(range(n), repeat=r):
        if len(set(indices)) == r:
            yield tuple(pool[i] for i in indices)

```

返される要素の数は、 $0 \leq r \leq n$ の場合 $n! / (n-r)!$ で、 $r > n$ の場合は 0 です。

バージョン 2.6 で追加.

`itertools.product(*iterables[, repeat])`

入力イテラブルの直積 (Cartesian product) です。

ジェネレータ式の入れ子になった for ループとおおよそ等価です。たとえば `product(A, B)` は `((x, y) for x in A for y in B)` と同じものを返します。

入れ子ループは走行距離計と同じように右端の要素がイテレーションごとに更新されていきます。このパターンは辞書式順序を作り出し、入力のイテレート可能オブジェクトたちがソートされていれば、直積タプルもソートされた順に吐き出されます。

イテラブル自身との直積を計算するためには、オプションの `repeat` キーワード引数に繰り返し回数を指定します。たとえば `product(A, repeat=4)` は `product(A, A, A, A)` と同じ意味です。

この関数は以下のコードとおおよそ等価ですが、実際の実装ではメモリ中に中間結果を作りません:

```

def product(*args, **kwargs):
    # product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) --> 000 001 010 011 100 101 110 111
    pools = map(tuple, args) * kwargs.get('repeat', 1)
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    for prod in result:
        yield tuple(prod)

```

バージョン 2.6 で追加.

`itertools.repeat(object[, times])`

繰り返し `object` を返すイテレータを作成します。 `times` を指定しない場合、無限に値を返し続けます。

`imap()` で常に同じオブジェクトを関数の引数として指定する場合に使用します。また、`izip()` で

作成するタプルの定数部分を指定する場合にも使用することもできます。この関数はおよそ以下のスクリプトと同等です:

```
def repeat(object, times=None):
    # repeat(10, 3) --> 10 10 10
    if times is None:
        while True:
            yield object
    else:
        for i in xrange(times):
            yield object
```

`repeat` は `imap` や `zip` に定数のストリームを与えるためによく利用されます:

```
>>> list(imap(pow, xrange(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`itertools.starmap` (*function, iterable*)

`iterable` の要素を引数として `function` を計算するイテレータを作成します。 `function` の引数が一つの `iterable` からタプルに既にグループ化されている (データが "zip 済み") 場合、 `imap()` の代わりに使います。 `imap()` と `starmap()` の違いは `function(a,b)` と `function(*c)` の差に似ています。 およそ次と等価です:

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
    for args in iterable:
        yield function(*args)
```

バージョン 2.6 で変更: 以前のバージョンでは、 `starmap()` は関数の引数がタプルであることが必要でした。 今では任意のイテレート可能オブジェクトを使えます。

`itertools.takewhile` (*predicate, iterable*)

`predicate` が真である限り `iterable` から要素を返すイテレータを作成します。 およそ次と等価です:

```
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

`itertools.tee` (*iterable[, n=2]*)

一つの `iterable` から `n` 個の独立したイテレータを生成して返します。 以下のコードとおよそ等価になります:

```
def tee(iterable, n=2):
    it = iter(iterable)
    deques = [collections.deque() for i in range(n)]
```

(次のページに続く)

(前のページからの続き)

```
def gen(mydeque):
    while True:
        if not mydeque:           # when the local deque is empty
            newval = next(it)     # fetch a new value and
            for d in deques:      # load it to all the deques
                d.append(newval)
        yield mydeque.popleft()
    return tuple(gen(d) for d in deques)
```

一度 `tee()` でイテレータを分割すると、もとの *iterable* を他で使ってはいけません。さもなければ、`tee()` オブジェクトの知らない間に *iterable* が先の要素に進んでしまうことになります。

`tee` iterators are not threadsafe. A `RuntimeError` may be raised when using simultaneously iterators returned by the same `tee()` call, even if the original *iterable* is threadsafe.

`tee()` はかなり大きなメモリ領域を使用するかもしれません (使用するメモリ量は *iterable* の大きさに依存します)。一般には、一つのイテレータが他のイテレータよりも先にほとんどまたは全ての要素を消費するような場合には、`tee()` よりも `list()` を使った方が高速です。

バージョン 2.4 で追加。

9.7.2 レシピ

この節では、既存の `itertools` を素材としてツールセットを拡張するためのレシピを示します。

iterable 全体を一度にメモリ上に置くよりも、要素を一つずつ処理する方がメモリ効率上の有利さを保てます。関数形式のままツールをリンクしてゆくと、コードのサイズを減らし、一時変数を減らす助けになります。インタプリタのオーバーヘッドをもたらす `for` ループやジェネレータ (*generator*) を使わずに、“ベクトル化された”ビルディングブロックを使うと、高速な処理を実現できます。

```
def take(n, iterable):
    "Return first n items of the iterable as a list"
    return list(islice(iterable, n))

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return imap(function, count(start))

def consume(iterator, n=None):
    "Advance the iterator n-steps ahead. If n is None, consume entirely."
    # Use functions that consume iterators at C speed.
    if n is None:
        # feed the entire iterator into a zero-length deque
        collections.deque(iterator, maxlen=0)
    else:
        # advance to the empty slice starting at position n
        next(islice(iterator, n, n), None)

def nth(iterable, n, default=None):
```

(次のページに続く)

(前のページからの続き)

```

    "Returns the nth item or a default value"
    return next(islice(iterable, n, None), default)

def all_equal(iterable):
    "Returns True if all the elements are equal to each other"
    g = groupby(iterable)
    return next(g, True) and not next(g, False)

def quantify(iterable, pred=bool):
    "Count how many times the predicate is true"
    return sum(imap(pred, iterable))

def padnone(iterable):
    """Returns the sequence elements and then returns None indefinitely.

    Useful for emulating the behavior of the built-in map() function.
    """
    return chain(iterable, repeat(None))

def ncycles(iterable, n):
    "Returns the sequence elements n times"
    return chain.from_iterable(repeat(tuple(iterable), n))

def dotproduct(vec1, vec2):
    return sum(imap(operator.mul, vec1, vec2))

def flatten(listOfLists):
    "Flatten one level of nesting"
    return chain.from_iterable(listOfLists)

def repeatfunc(func, times=None, *args):
    """Repeat calls to func with specified arguments.

    Example:  repeatfunc(random.random)
    """
    if times is None:
        return starmap(func, repeat(args))
    return starmap(func, repeat(args, times))

def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    next(b, None)
    return izip(a, b)

def grouper(iterable, n, fillvalue=None):
    "Collect data into fixed-length chunks or blocks"
    # grouper('ABCDEFG', 3, 'x') --> ABC DEF Gxx
    args = [iter(iterable)] * n
    return izip_longest(fillvalue=fillvalue, *args)

```

(次のページに続く)

(前のページからの続き)

```

def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    # Recipe credited to George Sakkis
    pending = len(iterables)
    nexts = cycle(iter(it).next for it in iterables)
    while pending:
        try:
            for next in nexts:
                yield next()
        except StopIteration:
            pending -= 1
            nexts = cycle(islice(nexts, pending))

def powerset(iterable):
    "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

def unique_everseen(iterable, key=None):
    "List unique elements, preserving order. Remember all elements ever seen."
    # unique_everseen('AAAABBBCCDAABBB') --> A B C D
    # unique_everseen('ABBCcAD', str.lower) --> A B C D
    seen = set()
    seen_add = seen.add
    if key is None:
        for element in ifilterfalse(seen.__contains__, iterable):
            seen_add(element)
            yield element
    else:
        for element in iterable:
            k = key(element)
            if k not in seen:
                seen_add(k)
                yield element

def unique_justseen(iterable, key=None):
    "List unique elements, preserving order. Remember only the element just seen."
    # unique_justseen('AAAABBBCCDAABBB') --> A B C D A B
    # unique_justseen('ABBCcAD', str.lower) --> A B C A D
    return imap(next, imap(itemgetter(1), groupby(iterable, key)))

def iter_except(func, exception, first=None):
    """ Call a function repeatedly until an exception is raised.

    Converts a call-until-exception interface to an iterator interface.
    Like __builtin__.iter(func, sentinel) but uses an exception instead
    of a sentinel to end the loop.

    Examples:
        bsddbiter = iter_except(db.next, bsddb.error, db.first)
        heapiter = iter_except(functools.partial(heapop, h), IndexError)

```

(次のページに続く)

```

dictiter = iter_except(d.popitem, KeyError)
dequeiter = iter_except(d.popleft, IndexError)
queueiter = iter_except(q.get_nowait, Queue.Empty)
setiter = iter_except(s.pop, KeyError)

"""
try:
    if first is not None:
        yield first()
    while 1:
        yield func()
except exception:
    pass

def random_product(*args, **kwds):
    "Random selection from itertools.product(*args, **kwds)"
    pools = map(tuple, args) * kwds.get('repeat', 1)
    return tuple(random.choice(pool) for pool in pools)

def random_permutation(iterable, r=None):
    "Random selection from itertools.permutations(iterable, r)"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "Random selection from itertools.combinations(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(xrange(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with_replacement(iterable, r):
    "Random selection from itertools.combinations_with_replacement(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.randrange(n) for i in xrange(r))
    return tuple(pool[i] for i in indices)

def tee_lookahead(t, i):
    """Inspect the i-th upcoming value from a tee object
    while leaving the tee object at its current position.

    Raise an IndexError if the underlying iterator doesn't
    have enough values.

    """
    for value in islice(t.__copy__(), i, None):
        return value
    raise IndexError(i)

```

上記のレシピはデフォルト値を指定してグローバルな名前検索をローカル変数の検索に変えることで、より効

率を上げることができます。例えば、*dotproduct* のレシピを書き換えるとすればこんな具合です:

```
def dotproduct(vec1, vec2, sum=sum, imap=imap, mul=operator.mul):
    return sum(imap(mul, vec1, vec2))
```

9.8 functools — 高次関数と呼び出し可能オブジェクトの操作

バージョン 2.5 で追加.

ソースコード: [Lib/functools.py](#)

functools モジュールは高次関数、つまり関数に影響を及ぼしたり他の関数を返したりする関数、のためのものです。一般に、どんな呼び出し可能オブジェクトでもこのモジュールの目的には関数として扱えます。

モジュール *functools* は以下の関数を定義します:

`functools.cmp_to_key(func)`

古いスタイルの比較関数を *key function* に変換します。key 関数を受け取る関数 (*sorted()*, *min()*, *max()*, *heapq.nlargest()*, *heapq.nsmallest()*, *itertools.groupby()* など) と共に使用します。この関数は、主に比較関数をサポートしていない Python 3 への移行のためのツールとして用意されています。

比較関数は 2 つの引数を受け取り、それらを比較し、”より小さい” 場合は負の数を、同値の場合には 0 を、”より大きい” 場合には正の数を返す、あらゆる呼び出し可能オブジェクトです。key 関数は呼び出し可能オブジェクトで、1 つの引数を受け取り、ソートキーとして使われる値を返します。

例:

```
sorted(iterable, key=cmp_to_key(locale.strcoll)) # locale-aware sort order
```

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

バージョン 2.7 で追加.

`functools.total_ordering(cls)`

ひとつ以上の拡張順序比較メソッド (rich comparison ordering methods) を定義したクラスを受け取り、残りを実装するクラスデコレータです。このデコレータは全ての拡張順序比較演算をサポートするための労力を軽減します:

引数のクラスは、`__lt__()`, `__le__()`, `__gt__()`, `__ge__()` の中からどれか 1 つと、`__eq__()` メソッドを定義する必要があります。

例えば:

```
@total_ordering
class Student:
    def __eq__(self, other):
        return ((self.lastname.lower(), self.firstname.lower()) ==
```

(次のページに続く)

(前のページからの続き)

```

        (other.lastname.lower(), other.firstname.lower()))
def __lt__(self, other):
    return ((self.lastname.lower(), self.firstname.lower()) <
            (other.lastname.lower(), other.firstname.lower()))

```

バージョン 2.7 で追加.

`functools.reduce(function, iterable[, initializer])`

これは `reduce()` 関数と同じものです。このモジュールからも使えるようにしたのは Python 3 と前方向互換なコードを書けるようにするためです。

バージョン 2.6 で追加.

`functools.partial(func[, *args][, **keywords])`

新しい *partial* オブジェクトを返します。このオブジェクトは呼び出されると位置引数 *args* とキーワード引数 *keywords* 付きで呼び出された *func* のように振る舞います。呼び出しに際してさらなる引数が渡された場合、それらは *args* に付け加えられます。追加のキーワード引数が渡された場合には、それらで *keywords* を拡張または上書きします。おおよそ次のコードと等価です:

```

def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*(args + fargs), **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc

```

関数 `partial()` は、関数の位置引数・キーワード引数の一部を「凍結」した部分適用として使われ、簡素化された引数形式をもった新たなオブジェクトを作り出します。例えば、`partial()` を使って `base` 引数のデフォルトが 2 である `int()` 関数のように振る舞う呼び出し可能オブジェクトを作ることができます:

```

>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18

```

`functools.update_wrapper(wrapper, wrapped[, assigned][, updated])`

wrapper 関数を *wrapped* 関数に見えるようにアップデートします。オプション引数はタプルで、元の関数のどの属性がラップ関数の対応する属性に直接代入されるか、またラップ関数のどの属性が元の関数の対応する属性でアップデートされる (*updated*) か、を指定します。これらの引数のデフォルト値はモジュールレベル定数 `WRAPPER_ASSIGNMENTS` (ラップ関数の `__name__`、`__module__`、そしてドキュメンテーション文字列 `__doc__` に代入します) と `WRAPPER_UPDATES` (ラップ関数の `__dict__`、すなわちインスタンス辞書をアップデートします) です。

この関数は主に関数を包んでラッパを返すデコレータ (*decorator*) 関数の中で使われるよう意図されています。もしラッパ関数がアップデートされないとすると、返される関数のメタデータは元の関数の定義ではなくラッパ関数の定義を反映してしまい、これは通常あまり有益ではありません。

`functools.wraps(wrapped[, assigned][, updated])`

これはラッパ関数を定義するときに `update_wrapper()` を関数デコレータとして呼び出す便宜関数です。これは `partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)` と等価です。例えば:

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwargs):
...         print 'Calling decorated function'
...         return f(*args, **kwargs)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print 'Called example function'
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

このデコレータ・ファクトリーを使わなければ、上の例中の関数の名前は 'wrapper' となり、元々の `example()` のドキュメンテーション文字列は失われたところです。

9.8.1 partial オブジェクト

partial オブジェクトは、`partial()` 関数によって作られる呼び出し可能オブジェクトです。オブジェクトには読み取り専用の属性が三つあります:

`partial.func`

呼び出し可能オブジェクトまたは関数です。 *partial* オブジェクトの呼び出しは新しい引数とキーワードと共に *func* に転送されます。

`partial.args`

最左の位置引数で、 *partial* オブジェクトの呼び出し時にその呼び出しの際の位置引数の前に追加されます。

`partial.keywords`

partial オブジェクトの呼び出し時に渡されるキーワード引数です。

partial オブジェクトは `function` オブジェクトのように呼び出し可能で、弱参照可能で、属性を持つこ

とができます。重要な相違点もあります。例えば、`__name__` と `__doc__` 両属性は自動では作られません。また、クラス中で定義された *partial* オブジェクトはスタティックメソッドのように振る舞い、インスタンスの属性問い合わせの中で束縛メソッドに変換されません。

9.9 operator — 関数形式の標準演算子

operator モジュールは、Python の組み込み演算子に対応する効率的な関数群を提供します。例えば、`operator.add(x, y)` は式 `x+y` と等価です。関数の名前は、特殊クラスメソッドに使われている名前と同じです; 便宜上、先頭と末尾の `_` を取り除いたものも提供されています。

これらの関数はそれぞれ、オブジェクト比較、論理演算、数学演算、シーケンス操作、および抽象型テストに分類されます。

オブジェクト比較関数は全てのオブジェクトで有効で、関数の名前はサポートする拡張比較演算子からとられています:

```
operator.lt(a, b)
operator.le(a, b)
operator.eq(a, b)
operator.ne(a, b)
operator.ge(a, b)
operator.gt(a, b)
operator.__lt__(a, b)
operator.__le__(a, b)
operator.__eq__(a, b)
operator.__ne__(a, b)
operator.__ge__(a, b)
operator.__gt__(a, b)
```

a と *b* の ”拡張比較 (rich comparisons)” を行います。具体的には、`lt(a, b)` は `a < b`、`le(a, b)` は `a <= b`、`eq(a, b)` は `a == b`、`ne(a, b)` は `a != b`、`gt(a, b)` は `a >= b`、そして `ge(a, b)` は `a > b` と等価です。組み込み関数 `cmp()` と違って、これらの関数はどのような値を返してもよく、ブール値として解釈できてもできなくてもかまいません。拡張比較の詳細については `comparisons` を参照してください。

バージョン 2.2 で追加。

論理演算もまた全てのオブジェクトに対して適用でき、真理値判定、同一性判定およびブール演算をサポートします:

```
operator.not_(obj)
operator.__not__(obj)
```

`not obj` の結果を返します。(オブジェクトインスタンスには `__not__()` メソッドは無いので注意してください; インタプリタコアがこの演算を定義しているだけです。結果は `__nonzero__()` および `__len__()` メソッドに影響されます。)

```
operator.truth(obj)
```

obj が真の場合 *True* を返し、そうでない場合 *False* を返します。この関数は *bool* のコンストラクタ呼び出しと同等です。

`operator.is_(a, b)`

a is b を返します。オブジェクトの同一性を判定します。

バージョン 2.3 で追加.

`operator.is_not(a, b)`

a is not b を返します。オブジェクトの同一性を判定します。

バージョン 2.3 で追加.

演算子で最も多いのは数学演算およびビット単位の演算です:

`operator.abs(obj)`

`operator.__abs__(obj)`

obj の絶対値を返します。

`operator.add(a, b)`

`operator.__add__(a, b)`

数値 *a* および *b* について *a + b* を返します。

`operator.and_(a, b)`

`operator.__and__(a, b)`

a と *b* のビット単位論理積を返します。

`operator.div(a, b)`

`operator.__div__(a, b)`

`__future__.division` が有効でなければ、*a / b* は *a // b* と同じ結果を返します。これは "古典的な (classic)" 除算とも呼ばれます。

`operator.floordiv(a, b)`

`operator.__floordiv__(a, b)`

a // b を返します。

バージョン 2.2 で追加.

`operator.index(a)`

`operator.__index__(a)`

整数に変換された *a* を返します。*a.__index__()* と同等です。

バージョン 2.5 で追加.

`operator.inv(obj)`

`operator.invert(obj)`

`operator.__inv__(obj)`

`operator.__invert__(obj)`

obj のビット単位反転を返します。*~obj* と同じです。

バージョン 2.0 で追加: 名前 *invert()* および *__invert__()* が追加されました。

`operator.lshift(a, b)`

`operator.__lshift__(a, b)`

a の *b* ビット左シフトを返します。

`operator.mod(a, b)`

`operator.__mod__(a, b)`

a % *b* を返します。

`operator.mul(a, b)`

`operator.__mul__(a, b)`

数値 *a* および *b* について *a* * *b* を返します。

`operator.neg(obj)`

`operator.__neg__(obj)`

obj の符号反転 ($-obj$) を返します。

`operator.or_(a, b)`

`operator.__or__(a, b)`

a と *b* のビット単位論理和を返します。

`operator.pos(obj)`

`operator.__pos__(obj)`

obj の符号非反転 ($+obj$) を返します。

`operator.pow(a, b)`

`operator.__pow__(a, b)`

数値 *a* および *b* について *a* ** *b* を返します。

バージョン 2.3 で追加.

`operator.rshift(a, b)`

`operator.__rshift__(a, b)`

a の *b* ビット右シフトを返します。

`operator.sub(a, b)`

`operator.__sub__(a, b)`

a - *b* を返します。

`operator.truediv(a, b)`

`operator.__truediv__(a, b)`

`__future__.division` が有効な場合 *a* / *b* を返します。 ”真の” 除算としても知られています。

バージョン 2.2 で追加.

`operator.xor(a, b)`

`operator.__xor__(a, b)`

a および *b* のビット単位排他的論理和を返します。

シーケンスを扱う演算子 (いくつかの演算子はマッピングも扱います) には以下のようなものがあります:

`operator.concat(a, b)`

`operator.__concat__(a, b)`

シーケンス *a* および *b* について $a + b$ を返します。

`operator.contains(a, b)`

`operator.__contains__(a, b)`

$b \text{ in } a$ の判定結果を返します。被演算子が左右反転しているので注意してください。

バージョン 2.0 で追加: 関数名 `__contains__()` が追加されました。

`operator.countOf(a, b)`

a の中に *b* が出現する回数を返します。

`operator.delitem(a, b)`

`operator.__delitem__(a, b)`

a でインデクスが *b* の値を削除します。

`operator.delslice(a, b, c)`

`operator.__delslice__(a, b, c)`

a でインデクスが *b* から *c-1* のスライス要素を削除します。

バージョン 2.6 で非推奨: この関数は Python 3.x で削除されます。 `delitem()` をスライスインデクスで使って下さい。

`operator.getitem(a, b)`

`operator.__getitem__(a, b)`

a でインデクスが *b* の値を返します。

`operator.getslice(a, b, c)`

`operator.__getslice__(a, b, c)`

a でインデクスが *b* から *c-1* のスライス要素を返します。

バージョン 2.6 で非推奨: この関数は Python 3.x で削除されます。 `getitem()` をスライスインデクスで使って下さい。

`operator.indexOf(a, b)`

a で最初に *b* が出現する場所のインデクスを返します。

`operator.repeat(a, b)`

`operator.__repeat__(a, b)`

バージョン 2.7 で非推奨: 代わりに `__mul__()` を使ってください。

シーケンス *a* と整数 *b* について $a * b$ を返します。

`operator.sequenceIncludes(...)`

バージョン 2.0 で非推奨: 代わりに `contains()` を使ってください。

`contains()` の別名です。

`operator.setitem(a, b, c)`

`operator.__setitem__(a, b, c)`

a でインデクスが *b* の値を *c* に設定します。

```
operator.setslice(a, b, c, v)
```

```
operator.__setslice__(a, b, c, v)
```

`a` でインデクスが `b` から `c-1` のスライス要素の値をシーケンス `v` に設定します。

バージョン 2.6 で非推奨: この関数は Python 3.x で削除されます。 `setitem()` をスライスインデクスで使って下さい。

operator の関数を使う例を挙げます:

```
>>> # Elementwise multiplication
>>> map(mul, [0, 1, 2, 3], [10, 20, 30, 40])
[0, 20, 60, 120]

>>> # Dot product
>>> sum(map(mul, [0, 1, 2, 3], [10, 20, 30, 40]))
200
```

多くの演算に「インプレース」版があります。以下の関数はそうした演算子の通常の文法に比べてより素朴な呼び出し方を提供します。たとえば、文 (*statement*) `x += y` は `x = operator.iadd(x, y)` と等価です。別の言い方をすると、`z = operator.iadd(x, y)` は複合文 `z = x; z += y` と等価です。

```
operator.iadd(a, b)
```

```
operator.__iadd__(a, b)
```

`a = iadd(a, b)` は `a += b` と等価です。

バージョン 2.5 で追加.

```
operator.iand(a, b)
```

```
operator.__iand__(a, b)
```

`a = iand(a, b)` は `a &= b` と等価です。

バージョン 2.5 で追加.

```
operator.iconcat(a, b)
```

```
operator.__iconcat__(a, b)
```

`a = iconcat(a, b)` は二つのシーケンス `a` と `b` に対し `a += b` と等価です。

バージョン 2.5 で追加.

```
operator.idiv(a, b)
```

```
operator.__idiv__(a, b)
```

`a = idiv(a, b)` は `__future__.division` が有効でないときに `a /= b` と等価です。

バージョン 2.5 で追加.

```
operator.ifloordiv(a, b)
```

```
operator.__ifloordiv__(a, b)
```

`a = ifloordiv(a, b)` は `a //= b` と等価です。

バージョン 2.5 で追加.

```
operator.ilshift(a, b)
```

`operator.__ilshift__(a, b)`

`a = ilshift(a, b)` は `a <<= b` と等価です。

バージョン 2.5 で追加.

`operator.imod(a, b)`

`operator.__imod__(a, b)`

`a = imod(a, b)` は `a %= b` と等価です。

バージョン 2.5 で追加.

`operator.imul(a, b)`

`operator.__imul__(a, b)`

`a = imul(a, b)` は `a *= b` と等価です。

バージョン 2.5 で追加.

`operator.ior(a, b)`

`operator.__ior__(a, b)`

`a = ior(a, b)` は `a |= b` と等価です。

バージョン 2.5 で追加.

`operator.ipow(a, b)`

`operator.__ipow__(a, b)`

`a = ipow(a, b)` は `a **= b` と等価です。

バージョン 2.5 で追加.

`operator.irepeat(a, b)`

`operator.__irepeat__(a, b)`

バージョン 2.7 で非推奨: 代わりに `__imul__()` を使ってください。

`a = irepeat(a, b)` は `a` がシーケンスで `b` が整数であるとき `a *= b` と等価です。

バージョン 2.5 で追加.

`operator.irshift(a, b)`

`operator.__irshift__(a, b)`

`a = irshift(a, b)` は `a >>= b` と等価です。

バージョン 2.5 で追加.

`operator.isub(a, b)`

`operator.__isub__(a, b)`

`a = isub(a, b)` は `a -= b` と等価です。

バージョン 2.5 で追加.

`operator.itruediv(a, b)`

`operator.__itruediv__(a, b)`

`a = itruediv(a, b)` は `__future__.division` が有効なときに `a /= b` と等価です。

バージョン 2.5 で追加.

```
operator.ixor(a, b)
```

```
operator.__ixor__(a, b)
```

`a = ixor(a, b)` は `a ^= b` と等価です。

バージョン 2.5 で追加.

`operator` モジュールでは、オブジェクトの型を調べるための述語演算子も定義しています。しかしながらこれらはいつでも信頼できるというわけではありません。代わりに抽象基底クラスをテストするのが望ましい方法です (詳しくは `collections` や `numbers` を参照して下さい)。

```
operator.isCallable(obj)
```

バージョン 2.0 で非推奨: 代わりに `isinstance(x, collections.Callable)` を使ってください。

オブジェクト `obj` を関数のように呼び出すことができる場合真を返し、それ以外の場合偽を返します。関数、束縛および非束縛メソッド、クラスオブジェクト、および `__call__()` メソッドをサポートするインスタンスオブジェクトに対しては真を返します。

```
operator.isMappingType(obj)
```

バージョン 2.7 で非推奨: 代わりに `isinstance(x, collections.Mapping)` を使ってください。

オブジェクト `obj` がマップ型インタフェースをサポートする場合に真を返します。辞書および `__getitem__()` メソッドが定義された全てのインスタンスオブジェクトに対しては、この値は真になります。

```
operator.isNumberType(obj)
```

バージョン 2.7 で非推奨: 代わりに `isinstance(x, numbers.Number)` を使ってください。

オブジェクト `obj` が数値を表現している場合に真を返します。C で実装された全ての数値型に対して、この値は真になります。

```
operator.isSequenceType(obj)
```

バージョン 2.7 で非推奨: 代わりに `isinstance(x, collections.Sequence)` を使ってください。

`obj` がシーケンス型プロトコルをサポートする場合に真を返します。シーケンス型メソッドを C で定義している全てのオブジェクトおよび `__getitem__()` メソッドが定義された全てのインスタンスオブジェクトに対して、この値は真になります。

`operator` モジュールは属性とアイテムの汎用的な検索のための道具も定義しています。 `map()`, `sorted()`, `itertools.groupby()`, や関数を引数に取るその他の関数に対して高速にフィールドを抽出する際に引数として使うと便利です。

```
operator.attrgetter(attr)
```

```
operator.attrgetter(*attrs)
```

演算対象から `attr` を取得する呼び出し可能なオブジェクトを返します。二つ以上の属性を要求された場合には、属性のタプルを返します。属性名はドットを含むこともできます。例えば:

- `f = attrgetter('name')` とした後で、`f(b)` を呼び出すと `b.name` を返します。

- `f = attrgetter('name', 'date')` とした後で、`f(b)` を呼び出すと `(b.name, b.date)` を返します。
- `f = attrgetter('name.first', 'name.last')` とした後で、`f(b)` を呼び出すと `(b.name.first, b.name.last)` を返します。

次と等価です:

```
def attrgetter(*items):
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
        def g(obj):
            return tuple(resolve_attr(obj, attr) for attr in items)
    return g

def resolve_attr(obj, attr):
    for name in attr.split("."):
        obj = getattr(obj, name)
    return obj
```

バージョン 2.4 で追加.

バージョン 2.5 で変更: 複数のアトリビュートがサポートされました。

バージョン 2.6 で変更: ドット付きアトリビュートがサポートされました。

`operator.itemgetter(item)`

`operator.itemgetter(*items)`

演算対象からその `--getitem--()` メソッドを使って `item` を取得する呼び出し可能なオブジェクトを返します。二つ以上のアイテムを要求された場合には、アイテムのタプルを返します。例えば:

- `f = itemgetter(2)` とした後で、`f(r)` を呼び出すと `r[2]` を返します。
- `g = itemgetter(2, 5, 3)` とした後で、`g(r)` を呼び出すと `(r[2], r[5], r[3])` を返します。

次と等価です:

```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

アイテムは被演算子の `--getitem--()` メソッドが受け付けるとんな型でも構いません。辞書ならば任

意のハッシュ可能な値を受け付けます。リスト、タプル、文字列などはインデックスかスライスを受け付けます:

```
>>> itemgetter(1)('ABCDEFGH')
'B'
>>> itemgetter(1,3,5)('ABCDEFGH')
('B', 'D', 'F')
>>> itemgetter(slice(2,None))('ABCDEFGH')
'CDEFGH'
```

バージョン 2.4 で追加.

バージョン 2.5 で変更: 複数アイテム抽出がサポートされました.

`itemgetter()` を使って特定のフィールドをタプルレコードから取り出す例:

```
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> map(getcount, inventory)
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

`operator.methodcaller` (`name`[, `args...`])

引数の `name` メソッドを呼び出す呼び出し可能オブジェクトを返します。追加の引数および/またはキーワード引数が与えられると、これらもそのメソッドに引き渡されます。例えば:

- `f = methodcaller('name')` とした後で、`f(b)` を呼び出すと `b.name()` を返します。
- `f = methodcaller('name', 'foo', bar=1)` とした後で、`f(b)` を呼び出すと `b.name('foo', bar=1)` を返します。

次と等価です:

```
def methodcaller(name, *args, **kwargs):
    def caller(obj):
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

バージョン 2.6 で追加.

9.9.1 演算子から関数への対応表

下のテーブルでは、個々の抽象的な操作が、どのように Python 構文上の各演算子や `operator` モジュールの関数に対応しているかを示しています。

演算	操作	関数
加算	<code>a + b</code>	<code>add(a, b)</code>

[次のページに続く](#)

表 1 – 前のページからの続き

演算	操作	関数
結合	<code>seq1 + seq2</code>	<code>concat(seq1, seq2)</code>
包含判定	<code>obj in seq</code>	<code>contains(seq, obj)</code>
除算	<code>a / b</code>	<code>div(a, b)</code> (<code>__future__.division</code> を使用しない場合)
除算	<code>a / b</code>	<code>truediv(a, b)</code> (<code>__future__.division</code> 使用時)
除算	<code>a // b</code>	<code>floordiv(a, b)</code>
ビット単位論理積	<code>a & b</code>	<code>and_(a, b)</code>
ビット単位排他的論理和	<code>a ^ b</code>	<code>xor(a, b)</code>
ビット単位反転	<code>~ a</code>	<code>invert(a)</code>
ビット単位論理和	<code>a b</code>	<code>or_(a, b)</code>
べき乗	<code>a ** b</code>	<code>pow(a, b)</code>
同一性	<code>a is b</code>	<code>is_(a, b)</code>
同一性	<code>a is not b</code>	<code>is_not(a, b)</code>
インデックス指定の代入	<code>obj[k] = v</code>	<code>setitem(obj, k, v)</code>
インデックス指定の削除	<code>del obj[k]</code>	<code>delitem(obj, k)</code>
インデックス指定	<code>obj[k]</code>	<code>getitem(obj, k)</code>
左シフト	<code>a << b</code>	<code>lshift(a, b)</code>
剰余	<code>a % b</code>	<code>mod(a, b)</code>
乗算	<code>a * b</code>	<code>mul(a, b)</code>
(算術) 負	<code>- a</code>	<code>neg(a)</code>
(論理) 否	<code>not a</code>	<code>not_(a)</code>
正	<code>+ a</code>	<code>pos(a)</code>
右シフト	<code>a >> b</code>	<code>rshift(a, b)</code>
シーケンスの反復	<code>seq * i</code>	<code>repeat(seq, i)</code>
スライス指定の代入	<code>seq[i:j] = values</code>	<code>setitem(seq, slice(i, j), values)</code>
スライス指定の削除	<code>del seq[i:j]</code>	<code>delitem(seq, slice(i, j))</code>
スライス指定	<code>seq[i:j]</code>	<code>getitem(seq, slice(i, j))</code>
文字列書式化	<code>s % obj</code>	<code>mod(s, obj)</code>
減算	<code>a - b</code>	<code>sub(a, b)</code>
真理値判定	<code>obj</code>	<code>truth(obj)</code>
順序付け	<code>a < b</code>	<code>lt(a, b)</code>
順序付け	<code>a <= b</code>	<code>le(a, b)</code>
等価性	<code>a == b</code>	<code>eq(a, b)</code>
不等性	<code>a != b</code>	<code>ne(a, b)</code>
順序付け	<code>a >= b</code>	<code>ge(a, b)</code>
順序付け	<code>a > b</code>	<code>gt(a, b)</code>

第 10 章

ファイルとディレクトリへのアクセス

この章で説明されるモジュールはディスクのファイルやディレクトリを扱います。たとえば、ファイルの属性を読むためのモジュール、ファイルパスを移植可能な方式で操作する、テンポラリファイルを作成するためのモジュールです。この章の完全な一覧は:

10.1 `os.path` — 共通のパス名操作

このモジュールには、パス名を操作する便利な関数が定義されています。ファイルの読み書きに関しては、`open()`、ファイルシステムへのアクセスに関しては、`os` モジュールを参照下さい。

注釈: これらの関数の多くは Windows の一律命名規則 (UNC パス名) を正しくサポートしていません。`splitunc()` と `ismount()` は正しく UNC パス名を操作できます。

Unix シェルとは異なり、Python はあらゆるパス展開を 自動的に は 行いません。アプリケーションがシェルのようなパス展開を必要とした場合は、`expanduser()` や `expandvars()` といった関数を明示的に呼び出すことで行えます。(`glob` モジュールも参照してください)

注釈: OS によって異なるパス名の決まりがあるため、標準ライブラリにはこのモジュールのいくつかのバージョンが含まれています。`os.path` モジュールは常に現在 Python が動作している OS に適したパスモジュールであるため、ローカルのパスを扱うのに適しています。各々のモジュールをインポートして 常に 一つのフォーマットを利用することも可能です。これらはすべて同じインタフェースを持っています:

- `posixpath` UNIX スタイルのパス用
 - `ntpath` Windows パス用
 - `macpath` 古いスタイルの MacOS パス用
 - `os2emxpath` OS/2 EMX パス用
-

`os.path.abspath(path)`

パス名 *path* の正規化された絶対パスを返します。ほとんどのプラットフォームでは、これは関数 `normpath()` を次のように呼び出した時と等価です: `normpath(join(os.getcwd(), path))`。

バージョン 1.5.2 で追加.

`os.path.basename(path)`

パス名 *path* の末尾のファイル名部分を返します。これは関数 `split()` に *path* を渡した時に返されるペアの 2 番目の要素です。この関数が返すのは Unix の **basename** とは異なります; Unix の **basename** は `'/foo/bar/'` に対して `'bar'` を返しますが、関数 `basename()` は空文字列 `('')` を返します。

`os.path.commonprefix(list)`

パスの *list* の中に共通する最長の接頭辞を (パス名の 1 文字 1 文字を判断して) 返します。もし *list* が空なら、空文字列 `('')` を返します。これは一度に 1 文字を扱うため、不正なパスを返すことがあるかもしれないので注意して下さい。

`os.path.dirname(path)`

パス名 *path* のディレクトリ名を返します。これは関数 `split()` に *path* を渡した時に返されるペアの 1 番目の要素です。

`os.path.exists(path)`

path が存在するなら、`True` を返します。壊れたシンボリックリンクについては `False` を返します。いくつかのプラットフォームでは、たとえ *path* が物理的に存在していたとしても、リクエストされたファイルに対する `os.stat()` の実行が許可されなければこの関数が `False` を返すことがあります。

`os.path.lexists(path)`

path が実在するパスなら `True` を返します。壊れたシンボリックリンクについては `True` を返します。`os.lstat()` がない環境では `exists()` と等価です。

バージョン 2.4 で追加.

`os.path.expanduser(path)`

Unix および Windows では、与えられた引数の先頭のパス要素 `~`、または `~user` を、*user* のホームディレクトリのパスに置き換えて返します。

Unix では、先頭の `~` は、環境変数 `HOME` が設定されているならその値に置き換えられます。設定されていない場合は、現在のユーザのホームディレクトリをビルトインモジュール `pwd` を使ってパスワードディレクトリから探して置き換えます。先頭の `~user` については、直接パスワードディレクトリから探します。

Windows では、`HOME` と `USERPROFILE` が設定されていればそれを使用します。設定されていない場合は、環境変数 `HOME` と `HOMEDRIVE` の組み合わせで置き換えられます。先頭の `~user` は `~` で得られるユーザパスの最後のディレクトリ要素を除去したものを利用します。

置き換えに失敗したり、引数のパスがチルダで始まっていなかった場合は、パスをそのまま返します。

`os.path.expandvars(path)`

引数のパスの環境変数を展開して返します。引数の中の `$name` または `${name}` のような形式の文字

列は環境変数、*name* の値に置き換えられます。不正な変数名や存在しない変数名の場合には変換されず、そのまま返します。

Windows では、`$name` や `${name}` の形式に加えて、`%name%` の形式もサポートされています。

`os.path.getatime(path)`

path に最後にアクセスした時刻を、エポック (*time* モジュールを参照下さい) からの経過時間を示す秒数で返します。ファイルが存在しなかったりアクセスできない場合は `os.error` を送出します。

バージョン 1.5.2 で追加.

バージョン 2.3 で変更: `os.stat_float_times()` が `True` を返す場合、この関数の返り値は浮動小数点数になります。

`os.path.getmtime(path)`

path の最終更新時刻を、エポック (*time* モジュールを参照下さい) からの経過時間を示す秒数で返します。ファイルが存在しなかったりアクセスできない場合は `os.error` を送出します。

バージョン 1.5.2 で追加.

バージョン 2.3 で変更: `os.stat_float_times()` が `True` を返す場合、この関数の返り値は浮動小数点数になります。

`os.path.getctime(path)`

システムの ctime、Unix 系など一部のシステムでは最後にメタデータが変更された時刻、Windows などその他のシステムでは *path* の作成時刻を返します。返り値はエポック (*time* モジュールを参照) からの経過時間を示す秒数になります。ファイルが存在しない、あるいはアクセスできなかった場合は `os.error` を送出します。

バージョン 2.3 で追加.

`os.path.getsize(path)`

ファイル *path* のサイズをバイト数で返します。ファイルが存在しなかったりアクセスできない場合は `os.error` を送出します。

バージョン 1.5.2 で追加.

`os.path.isabs(path)`

path が絶対パスなら `True` を返します。すなわち、Unix ではスラッシュで始まり、Windows ではドライブレターに続く (バック) スラッシュで始まる場合です。

`os.path.isfile(path)`

path が実在する一般ファイルなら `True` を返します。シンボリックリンクの場合にはその実体をチェックするので、同じパスに対して `islink()` と `isfile()` の両方が `True` を返すことがあります。

`os.path.isdir(path)`

path が実在するディレクトリなら `True` を返します。シンボリックリンクの場合にはその実体をチェックするので、同じパスに対して `islink()` と `isdir()` の両方が `True` を返すことがあります。

`os.path.islink(path)`

path がシンボリックリンクなら `True` を返します。Python ランタイムでシンボリックリンクがサポー

トされていないプラットフォームでは、常に `False` を返します。

`os.path.ismount (path)`

パス名 *path* がマウントポイント *mount point* (ファイルシステムの中で異なるファイルシステムがマウントされているところ) なら `True` を返します。この関数は *path* の親ディレクトリである *path/..* が *path* と異なるデバイス上にあるか、あるいは *path/..* と *path* が同じデバイス上の同じ i-node を指しているかをチェックします — これによってすべての Unix と POSIX 系システムでマウントポイントが検出できます。

`os.path.join (path, *paths)`

1 つあるいはそれ以上のパスの要素を賢く結合します。戻り値は *path*、ディレクトリの区切り文字 (`os.sep`) を **paths* の各パートの (末尾でない場合の空文字列を除いて) 頭に付けたもの、これらの結合になります。最後の部分が空文字列の場合に限り区切り文字で終わる文字列になります。付け加える要素に絶対パスがあれば、それより前の要素は全て破棄され、以降の要素を結合します。

Windows の場合は、絶対パスの要素 (たとえば `r'\foo'`) が見つかった場合はドライブレターはリセットされません。要素にドライブレターが含まれていれば、それより前の要素は全て破棄され、ドライブレターがリセットされます。各ドライブに対してカレントディレクトリがあるので、`os.path.join("c:", "foo")` によって、`c:\foo` ではなく、ドライブ `C:` 上のカレントディレクトリからの相対パス (`c:foo`) が返されることに注意してください。

`os.path.normcase (path)`

パス名の大文字、小文字をシステムの標準にします。Unix と Mac OS X ではそのまま返します。大文字、小文字を区別しないファイルシステムではパス名を小文字に変換します。Windows では、スラッシュをバックスラッシュに変換します。

`os.path.normpath (path)`

パスを正規化します。余分な区切り文字や上位レベル参照を除去し、`A//B`、`A/B/`、`A/. /B` や `A/foo/.. /B` などはすべて `A/B` になります。この文字列操作は、シンボリックリンクを含むパスの意味を変えてしまう場合があります。Windows では、スラッシュをバックスラッシュに変換します。大文字小文字の正規化には `normcase()` を使用してください。

`os.path.realpath (path)`

パスの中のシンボリックリンク (もしそれが当該オペレーティングシステムでサポートされていれば) を取り除いて、指定されたファイル名を正規化したパスを返します。

バージョン 2.2 で追加。

`os.path.relpath (path[, start])`

カレントディレクトリあるいはオプションの *start* ディレクトリからの *path* への相対パスを返します。これは経路計算で行っており、ファイルシステムにアクセスして *path* や *start* の存在や性質を確認することはありません。

start のデフォルト値は `os.curdir` です。

利用できる環境 : Windows、Unix

バージョン 2.6 で追加。

`os.path.samefile(path1, path2)`

2つの引数であるパス名が同じファイルあるいはディレクトリを指していれば(同じデバイスナンバーと i-node ナンバーで示されていれば)、`True` を返します。どちらかのパス名で `os.stat()` の呼び出しに失敗した場合には、例外が発生します。

利用できる環境: Unix。

`os.path.sameopenfile(fp1, fp2)`

ファイル記述子 `fp1` と `fp2` が同じファイルを参照していたら `True` を返します。

利用できる環境: Unix。

`os.path.samestat(stat1, stat2)`

`stat` タプル `stat1` と `stat2` が同じファイルを参照していれば `True` を返します。これらのタプルは `os.fstat()`、`os.lstat()` あるいは `os.stat()` の返り値で構いません。この関数は `samefile()` と `sameopenfile()` を使用した比較に基いて実装しています。

利用できる環境: Unix。

`os.path.split(path)`

パス名 `path` を (`head`, `tail`) のペアに分割します。`tail` はパス名の構成要素の末尾で、`head` はそれより前の部分です。`tail` はスラッシュを含みません; もし `path` がスラッシュで終わっていれば `tail` は空文字列になります。もし `path` にスラッシュがなければ、`head` は空文字列になります。`path` が空文字なら、`head` と `tail` の両方が空文字列になります。`head` の末尾のスラッシュは `head` がルートディレクトリ(または1個以上のスラッシュだけ)でない限り取り除かれます。`join(head, tail)` は常に `path` と同じ場所を返しますが、文字列としては異なるかもしれません。関数 `dirname()`、`basename()` も参照してください。

`os.path.splitdrive(path)`

パス名 `path` を (`drive`, `tail`) のペアに分割します。`drive` はドライブ名か、空文字列です。ドライブ名を使用しないシステムでは、`drive` は常に空文字列です。全ての場合に `drive + tail` は `path` と等しくなります。

バージョン 1.3 で追加。

`os.path.splitext(path)`

パス名 `path` を (`root`, `ext`) のペアに分割します。`root + ext == path` になります。`ext` は空文字列か1つのピリオドで始まり、多くても1つのピリオドを含みます。ベースネームを導出するピリオドは無視されます; `splitext('.cshrc')` は `('.cshrc', '')` を返します。

バージョン 2.6 で変更: 以前のバージョンでは、最初の文字がピリオドであった場合、空の `root` を生成していました。

`os.path.splitunc(path)`

パス名 `path` をペア (`unc`, `rest`) に分割します。ここで `unc` は `(r'\\host\mount')` のような UNC マウントポイント、そして `rest` は `(r'path\file.ext')` のようなパスの残りの部分です。ドライブレターを含むパスでは常に `unc` が空文字列になります。

利用できる環境: Windows。

`os.path.walk(path, visit, arg)`

`path` をルートとする各ディレクトリに対して (もし `path` がディレクトリなら `path` も含みます)、`(arg, dirname, names)` を引数として関数 `visit` を呼び出します。引数 `dirname` は訪れたディレクトリを示し、引数 `names` はそのディレクトリ内のファイルのリスト (`os.listdir(dirname)` で得られる) です。関数 `visit` によって `names` を変更して、`dirname` 以下の対象となるディレクトリのセットを変更することもできます。例えば、あるディレクトリツリーだけ関数を適用しないなど。 (`names` で参照されるオブジェクトは、`del` あるいはスライスを使って正しく変更しなければなりません。)

注釈: ディレクトリへのシンボリックリンクはサブディレクトリとして扱われないので、`walk()` による操作対象とはされません。ディレクトリへのシンボリックリンクを操作対象とするには、`os.path.islink(file)` と `os.path.isdir(file)` で識別して、`walk()` で必要な操作を実行しなければなりません。

注釈: この関数は廃止予定で、Python 3 では `os.walk()` を採用し、こちらは削除されます。

`os.path.supports_unicode_filenames`

ファイル名に任意の Unicode 文字列を (システムの制限内で) 使用できる場合は `True` になります。

バージョン 2.3 で追加。

10.2 fileinput — 複数の入力ストリームをまたいだ行の繰り返し処理をサポートする

ソースコード: [Lib/fileinput.py](#)

このモジュールは標準入力やファイルの並びにまたがるループを素早く書くためのヘルパークラスと関数を提供しています。単一のファイルを読み書きしたいだけなら、`open()` を参照してください。

典型的な使い方は以下の通りです:

```
import fileinput
for line in fileinput.input():
    process(line)
```

このプログラムは `sys.argv[1:]` に含まれる全てのファイルをまたいで繰り返します。もし該当するものがなければ、`sys.stdin` がデフォルトとして扱われます。ファイル名として `'-'` が与えられた場合も、`sys.stdin` に置き換えられます。別のファイル名リストを使いたい時には、`input()` の最初の引数にリストを与えます。単一ファイル名の文字列も受け付けます。

全てのファイルはデフォルトでテキストモードでオープンされます。しかし、`input()` や `FileInput` をコールする際に `mode` パラメータを指定すれば、これをオーバーライドすることができます。オープン中あるいは読み込み中に I/O エラーが発生した場合には、`IOError` が発生します。

`sys.stdin` が 2 回以上使われた場合は、2 回目以降は行を返しません。ただしインタラクティブに利用している時や明示的にリセット (`sys.stdin.seek(0)` を使う) を行った場合はその限りではありません。

空のファイルは開いた後すぐ閉じられます。空のファイルはファイル名リストの最後にある場合にしか外部に影響を与えません。

ファイルの各行は、各種改行文字まで含めて返されます。ファイルの最後が改行文字で終わっていない場合には、改行文字で終わらない行が返されます。

ファイルのオープン方法を制御するためのオープン時フックは、`fileinput.input()` あるいは `FileInput()` の `openhook` パラメータで設定します。このフックは、ふたつの引数 `filename` と `mode` をとる関数でなければなりません。そしてその関数の返り値はオープンしたファイルオブジェクトとなります。このモジュールには、便利なフックが既に用意されています。

以下の関数がこのモジュールの基本的なインタフェースです:

```
fileinput.input([files[, inplace[, backup[, bufsize[, mode[, openhook]]]]])
```

`FileInput` クラスのインスタンスを作ります。生成されたインスタンスは、このモジュールの関数群が利用するグローバルな状態として利用されます。この関数への引数は `FileInput` クラスのコンストラクタへ渡されます。

バージョン 2.5 で変更: パラメータ `mode` および `openhook` が追加されました。

バージョン 2.7.12 で変更: `bufsize` 引数はもう使われません。

以下の関数は `fileinput.input()` 関数によって作られたグローバルな状態を利用します。アクティブな状態が無い場合には、`RuntimeError` が発生します。

```
fileinput.filename()
```

現在読み込み中のファイル名を返します。一行目が読み込まれる前は `None` を返します。

```
fileinput.fileeno()
```

現在のファイルの "ファイルデスク립タ" を整数値で返します。ファイルがオープンされていない場合 (最初の行の前、ファイルとファイルの間) は `-1` を返します。

バージョン 2.5 で追加。

```
fileinput.lineno()
```

最後に読み込まれた行の、累積した行番号を返します。1 行目が読み込まれる前は `0` を返します。最後のファイルの最終行が読み込まれた後には、その行の行番号を返します。

```
fileinput.filelineno()
```

現在のファイル中での行番号を返します。1 行目が読み込まれる前は `0` を返します。最後のファイルの最終行が読み込まれた後には、その行のファイル中での行番号を返します。

```
fileinput.isfirstline()
```

最後に読み込まれた行がファイルの 1 行目なら `True`、そうでなければ `False` を返します。

```
fileinput.isstdin()
```

最後に読み込まれた行が `sys.stdin` から読まれていれば `True`、そうでなければ `False` を返します。

```
fileinput.nextfile()
```

現在のファイルを閉じます。次の繰り返しでは (存在すれば) 次のファイルの最初の行が読み込まれます。閉じたファイルの読み込まれなかった行は、累積の行数にカウントされません。ファイル名は次のファイルの最初の行が読み込まれるまで変更されません。最初の行の読み込みが行われるまでは、この関数は呼び出されても何もみませんので、最初のファイルをスキップするために利用することはできません。最後のファイルの最終行が読み込まれた後にも、この関数は呼び出されても何もみません。

```
fileinput.close()
```

シーケンスを閉じます。

このモジュールのシーケンスの振舞いを実装しているクラスのサブクラスを作こともできます:

```
class fileinput.FileInput ([files[, inplace[, backup[, bufsize[, mode[, openhook]]]]])
```

FileInput クラスはモジュールの関数に対応するメソッド *filename()*、*fileno()*、*lineno()*、*filelineno()*、*isfirstline()*、*isstdin()*、*nextfile()* および *close()* を実装しています。それに加えて、次の入力行を返す *readline()* メソッドと、シーケンスの振舞いの実装をしている *__getitem__()* メソッドがあります。シーケンスはシーケンシャルに読み込むことしかできません。つまりランダムアクセスと *readline()* を混在させることはできません。

mode を使用すると、*open()* に渡すファイルモードを指定することができます。これは 'r'、'rU'、'U' および 'rb' のうちのいずれかとなります。

openhook を指定する場合は、ふたつの引数 *filename* と *mode* をとる関数でなければなりません。この関数の返り値は、オープンしたファイルオブジェクトとなります。*inplace* と *openhook* を同時に使うことはできません。

バージョン 2.5 で変更: パラメータ *mode* および *openhook* が追加されました。

バージョン 2.7.12 で変更: *bufsize* 引数はもう使われません。

インプレース (in-place) フィルタオプション: キーワード引数 *inplace=1* が *fileinput.input()* か *FileInput* クラスのコンストラクタに渡された場合には、入力ファイルはバックアップファイルに移動され、標準出力が入力ファイルに向けられます (バックアップファイルと同じ名前のファイルが既に存在していた場合には、警告無しに置き換えられます)。これによって入力ファイルをその場で書き替えるフィルタを書くことができます。キーワード引数 *backup* (通常は *backup='.<拡張子>'* という形で利用します) が与えられていた場合、バックアップファイルの拡張子として利用され、バックアップファイルは削除されずに残ります。デフォルトでは、拡張子は *'.bak'* になっていて、出力先のファイルが閉じられればバックアップファイルも消されます。インプレースフィルタ機能は、標準入力を読み込んでいる間は無効にされます。

注釈: 現在の実装は MS-DOS の 8+3 ファイルシステムでは動作しません。

このモジュールには、次のふたつのオープン時フックが用意されています:

```
fileinput.hook_compressed(filename, mode)
```

gzip や *bzip2* で圧縮された (拡張子が *'.gz'* や *'.bz2'* の) ファイルを、*gzip* モジュールや *bz2* モジュールを使って透過的にオープンします。ファイルの拡張子が *'.gz'* や *'.bz2'* でない場合は、通常通りファイルをオープンします (つまり、*open()* をコールする際に伸長を行いません)。

使用例: `fi = fileinput.FileInput(openhook=fileinput.hook_compressed)`

バージョン 2.5 で追加.

`fileinput.hook_encoded(encoding)`

各ファイルを `io.open()` でオープンするフックを返します。指定した `encoding` でファイルを読み込みます。

使用例: `fi = fileinput.FileInput(openhook=fileinput.hook_encoded("iso-8859-1"))`

注釈: このフックでは、指定した `encoding` によっては `FileInput` が Unicode 文字列を返す可能性があります。

バージョン 2.5 で追加.

10.3 stat — stat() の結果を解釈する

ソースコード: [Lib/stat.py](#)

`stat` モジュールでは、`os.stat()`、`os.lstat()` および `os.fstat()` (存在すれば) の返す内容を解釈するための定数や関数を定義しています。`stat()`、`fstat()`、および `lstat()` の関数呼び出しについての完全な記述はシステムのドキュメントを参照してください。

`stat` モジュールでは、特殊なファイル型を判別するための以下の関数を定義しています:

`stat.S_ISDIR(mode)`

ファイルのモードがディレクトリの場合にゼロでない値を返します。

`stat.S_ISCHR(mode)`

ファイルのモードがキャラクタ型の特殊デバイスファイルの場合にゼロでない値を返します。

`stat.S_ISBLK(mode)`

ファイルのモードがブロック型の特殊デバイスファイルの場合にゼロでない値を返します。

`stat.S_ISREG(mode)`

ファイルのモードが通常ファイルの場合にゼロでない値を返します。

`stat.S_ISFIFO(mode)`

ファイルのモードが FIFO (名前つきパイプ) の場合にゼロでない値を返します。

`stat.S_ISLNK(mode)`

ファイルのモードがシンボリックリンクの場合にゼロでない値を返します。

`stat.S_ISSOCK(mode)`

ファイルのモードがソケットの場合にゼロでない値を返します。

より一般的なファイルのモードを操作するための二つの関数が定義されています:

`stat.S_IMODE(mode)`

`os.chmod()` で設定することのできる一部のファイルモード — すなわち、ファイルの許可ビット (permission bits) に加え、(サポートされているシステムでは) スティッキービット (sticky bit)、実行グループ ID 設定 (set-group-id) および実行ユーザ ID 設定 (set-user-id) ビット — を返します。

`stat.S_IFMT(mode)`

ファイルの形式を記述しているファイルモードの一部 (上記の `S_IS*`() 関数で使われます) を返します。

通常、ファイルの形式を調べる場合には `os.path.is*`() 関数を使うことになります; ここで挙げた関数は同じファイルに対して複数のテストを同時に行いたい、`stat()` システムコールを何度も呼び出してオーバーヘッドが生じるのを避けたい場合に便利です。これらはまた、ブロック型およびキャラクタ型デバイスに対するテストのように、`os.path` で扱うことのできないファイルの情報を調べる際にも便利です。

例:

```
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted at top,
       calling the callback function for each regular file'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.stat(pathname).st_mode
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print 'Skipping %s' % pathname

def visitfile(file):
    print 'visiting', file

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

以下の全ての変数は、`os.stat()`、`os.fstat()`、または `os.lstat()` が返す 10 要素のタプルにおけるインデックスを単にシンボル定数化したものです。

`stat.ST_MODE`

I ノードの保護モード。

`stat.ST_INO`

I ノード番号。

`stat.ST_DEV`

I ノードが存在するデバイス。

`stat.ST_NLINK`

該当する I ノードへのリンク数。

`stat.ST_UID`

ファイルの所持者のユーザ ID。

`stat.ST_GID`

ファイルの所持者のグループ ID。

`stat.ST_SIZE`

通常ファイルではバイトサイズ; いくつかの特殊ファイルでは処理待ちのデータ量。

`stat.ST_ATIME`

最後にアクセスした時刻。

`stat.ST_MTIME`

最後に変更された時刻。

`stat.ST_CTIME`

オペレーティングシステムから返される”ctime”。ある OS(Unix など) では最後にメタデータが更新された時間となり、別の OS(Windows など) では作成時間となります (詳細については各プラットフォームのドキュメントを参照してください)。

”ファイルサイズ”の解釈はファイルの型によって異なります。通常のファイルの場合、サイズはファイルの大きさをバイトで表したものです。ほとんどの Unix 系 (特に Linux) における FIFO やソケットの場合、”サイズ”は `os.stat()`、`os.fstat()`、あるいは `os.lstat()` を呼び出した時点で読み出し待ちであったデータのバイト数になります; この値は時に有用で、特に上記の特殊なファイルを非ブロックモードで開いた後にポーリングを行いたいといった場合に便利です。他のキャラクタ型およびブロック型デバイスにおけるサイズフィールドの意味はさらに異なっていて、背後のシステムコールの実装によります。

以下の変数は、`ST_MODE` フィールドで使用されるフラグを定義しています。

最初に挙げる、以下のフラグを使うよりは、上記の関数を使うほうがポータブルです:

`stat.S_IFSOCK`

ソケット。

`stat.S_IFLNK`

シンボリックリンク。

`stat.S_IFREG`

通常のファイル。

`stat.S_IFBLK`

ブロックデバイス。

`stat.S_IFDIR`

ディレクトリ。

`stat.S_IFCHR`

キャラクターデバイス。

`stat.S_IFIFO`

FIFO。

以下のフラグは、`os.chmod()` の *mode* 引数に使うこともできます:

`stat.S_ISUID`

UID ビットを設定する。

`stat.S_ISGID`

グループ ID ビットを設定する。このビットには幾つかの特殊ケースがあります。ディレクトリに対して設定されていた場合、BSD のセマンティクスが利用される事を示しています。すなわち、そこに作成されるファイルは、作成したプロセスの有効グループ ID (effective group ID) ではなくそのディレクトリのグループ ID を継承し、そこに作成されるディレクトリにも `S_ISGID` ビットが設定されます。グループ実行ビット (`S_IXGRP`) が設定されていないファイルに対してこのビットが設定されていた場合、強制ファイル/レコードロックを意味します (`S_ENFMT` も参照してください)。

`stat.S_ISVTX`

スティッキービット。このビットがディレクトリに対して設定されているとき、そのディレクトリ内のファイルは、そのファイルのオーナー、あるいはそのディレクトリのオーナーが特権プロセスのみが、リネームや削除をすることが出来ることを意味しています。

`stat.S_IRWXU`

ファイルオーナーの権限に対するマスク。

`stat.S_IRUSR`

オーナーがリード権限を持っている。

`stat.S_IWUSR`

オーナーがライト権限を持っている。

`stat.S_IXUSR`

オーナーが実行権限を持っている。

`stat.S_IRWXG`

グループの権限に対するマスク。

`stat.S_IRGRP`

グループがリード権限を持っている。

`stat.S_IWGRP`

グループがライト権限を持っている。

`stat.S_IXGRP`

グループが実行権限を持っている。

`stat.S_IRWXO`

その他 (グループ外) の権限に対するマスク。

`stat.S_IROTH`

その他はリード権限を持っている。

`stat.S_IWOTH`

その他はライト権限を持っている。

`stat.S_IXOTH`

その他は実行権限を持っている。

`stat.S_ENFMT`

System V ファイルロック強制。このフラグは `S_ISGID` と共有されています。グループ実行ビット (`S_IXGRP`) が設定されていないファイルでは、ファイル/レコードのロックが強制されます。

`stat.S_IREAD`

`S_IRUSR` の、Unix V7 のシノニム。

`stat.S_IWRITE`

`S_IWUSR` の、Unix V7 のシノニム。

`stat.S_IEXEC`

`S_IXUSR` の、Unix V7 のシノニム。

以下のフラグを `os.chflags()` の *flags* 引数として利用できます:

`stat.UF_NODUMP`

ファイルをダンプしない。

`stat.UF_IMMUTABLE`

ファイルは変更されない。

`stat.UF_APPEND`

ファイルは追記しかされない。

`stat.UF_OPAQUE`

ユニオンファイルシステムのスタックを通したとき、このディレクトリは不透明です。

`stat.UF_NOUNLINK`

ファイルはリネームや削除されない。

`stat.UF_COMPRESSED`

ファイルは圧縮して保存される (Mac OS X 10.6+)。

`stat.UF_HIDDEN`

ファイルは GUI で表示されるべきでない (Mac OS X 10.5+)。

`stat.SF_ARCHIVED`

ファイルはアーカイブされているかもしれません。

`stat.SF_IMMUTABLE`

ファイルは変更されない。

`stat.SF_APPEND`

ファイルは追記しかされない。

`stat.SF_NOUNLINK`

ファイルはリネームや削除されない。

`stat.SF_SNAPSHOT`

このファイルはスナップショットファイルです。

詳しい情報は *BSD か Mac OS システムの man page *chflags(2)* を参照してください。

10.4 statvfs — `os.statvfs()` で使われる定数群

バージョン 2.6 で非推奨: *statvfs* モジュールは Python 3 で削除されました。

statvfs モジュールでは、`os.statvfs()` の返す値を解釈するための定数を定義しています。`os.statvfs()` は "マジックナンバ" を記憶せずにタプルを生成して返します。このモジュールで定義されている各定数は `os.statvfs()` が返すタプルにおいて、特定の情報が収められている各エントリへのインデクスです。

`statvfs.F_BSIZE`

選択されているファイルシステムのブロックサイズです。

`statvfs.F_FRSIZE`

ファイルシステムの基本ブロックサイズです。

`statvfs.F_BLOCKS`

ブロック数の総計です。

`statvfs.F_BFREE`

空きブロック数の総計です。

`statvfs.F_BAVAIL`

非スーパーユーザが利用できる空きブロック数です。

`statvfs.F_FILES`

ファイルノード数の総計です。

`statvfs.F_FFREE`

空きファイルノード数の総計です。

`statvfs.F_FAVAIL`

非スーパーユーザが利用できる空きノード数です。

`statvfs.F_FLAG`

フラグで、システム依存です: `statvfs()` マニュアルページを参照してください。

`statvfs.F_NAMEMAX`

ファイル名の最大長です。

10.5 filecmp — ファイルおよびディレクトリの比較

ソースコード: [Lib/filecmp.py](#)

`filecmp` モジュールでは、ファイルおよびディレクトリを比較するため、様々な時間 / 正確性のトレードオフに関するオプションを備えた関数を定義しています。ファイルの比較については、`difflib` モジュールも参照してください。

`filecmp` モジュールでは以下の関数を定義しています:

`filecmp.cmp(f1, f2[, shallow])`

名前が `f1` および `f2` のファイルを比較し、二つのファイルが同じらしければ `True` を返し、そうでなければ `False` を返します。

`shallow` が真の場合、同一の `os.stat()` シグニチャを持つファイルは等しいとみなされます。

`os.stat()` シグニチャが変わらない限り、この関数を用いて比較されたファイルが再び比較されることはありません。

可搬性と効率のために、この関数は外部プログラムを一切呼び出さないで注意してください。

`filecmp.cmpfiles(dir1, dir2, common[, shallow])`

`dir1` と `dir2` ディレクトリの中の、`common` で指定されたファイルを比較します。

ファイル名からなる 3 つのリスト: `match`, `mismatch`, `errors` を返します。`match` には双方のディレクトリで一致したファイルのリストが含まれ、`mismatch` にはそうでないファイル名のリストが入ります。そして `errors` は比較されなかったファイルが列挙されます。`errors` になるのは、片方あるいは両方のディレクトリに存在しなかった、ユーザーにそのファイルを読む権限がなかった、その他何らかの理由で比較を完了することができなかった場合です。

引数 `shallow` はその意味も標準の設定も `filecmp.cmp()` と同じです。

例えば、`cmpfiles('a', 'b', ['c', 'd/e'])` は `a/c` を `b/c` と、`a/d/e` を `b/d/e` と、それぞれ比較します。`'c'` と `'d/e'` はそれぞれ、返される 3 つのリストのいずれかに登録されます。

例:

```
>>> import filecmp
>>> filecmp.cmp('undoc.rst', 'undoc.rst')
True
>>> filecmp.cmp('undoc.rst', 'index.rst')
False
```

10.5.1 dircmp クラス

`dircmp` インスタンスはこのコンストラクタを用いて構築されます:

`class filecmp.dircmp(a, b[, ignore[, hide]])`

ディレクトリ `a` および `b` を比較するための新しいディレクトリ比較オブジェクトを生成します。 `ignore`

は比較の際に無視するファイル名のリストで、標準の設定では ['RCS', 'CVS', 'tags'] です。
hide は表示しない名前のリストで、標準の設定では [os.curdir, os.pardir] です。

dircmp クラスは、*filecmp.cmp()* で説明されているような 浅い 比較を行うことによりファイルを比較します。

dircmp クラスは以下のメソッドを提供しています:

report()

a と *b* の比較を (*sys.stdout* に) 表示します。

report_partial_closure()

a および *b* およびそれらの直下にある共通のサブディレクトリ間での比較結果を出力します。

report_full_closure()

a および *b* およびそれらの共通のサブディレクトリ間での比較結果を (再帰的に比較して) 出力します。

dircmp クラスは、比較されているディレクトリ階層に関する様々な情報のビットを得るために使用することのできる、興味深い属性を数多く提供しています。

__getattr__() フックを経由すると、全ての属性をのろのろと計算するため、速度上のペナルティを受けないのは計算処理の軽い属性を使ったときだけなので注意してください。

left

ディレクトリ *a* です。

right

ディレクトリ *b* です。

left_list

a にあるファイルおよびサブディレクトリです。 *hide* および *ignore* でフィルタされています。

right_list

b にあるファイルおよびサブディレクトリです。 *hide* および *ignore* でフィルタされています。

common

a および *b* の両方にあるファイルおよびサブディレクトリです。

left_only

a だけにあるファイルおよびサブディレクトリです。

right_only

b だけにあるファイルおよびサブディレクトリです。

common_dirs

a および *b* の両方にあるサブディレクトリです。

common_files

a および *b* の両方にあるファイルです。

common.funny

a および *b* の両方にあり、ディレクトリ間でタイプが異なるか、`os.stat()` がエラーを報告するような名前です。

same_files

クラスのファイル比較オペレータを用いて *a* と *b* の両方において同一のファイルです。

diff_files

a と *b* の両方に存在し、クラスのファイル比較オペレータに基づいて内容が異なるファイルです。

funny_files

a および *b* 両方にあるが、比較されなかったファイルです。

subdirs

`common.dirs` のファイル名を `dircmp` オブジェクトに対応付けた辞書です。

これは `subdirs` 属性を使用して 2 つのディレクトリを再帰的に探索して、共通の異なるファイルを示すための単純化された例です:

```
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...     for name in dcmp.diff_files:
...         print "diff_file %s found in %s and %s" % (name, dcmp.left,
...             dcmp.right)
...     for sub_dcmp in dcmp.subdirs.values():
...         print_diff_files(sub_dcmp)
...
>>> dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)
```

10.6 tempfile — 一時的なファイルやディレクトリの生成

ソースコード: [Lib/tempfile.py](#)

このモジュールを使うと、一時的なファイルやディレクトリを生成できます。このモジュールはサポートされている全てのプラットフォームで利用可能です。

バージョン 2.3 の Python では、このモジュールに対してセキュリティを高める為の見直しが行われました。現在では新たに 3 つの関数、`NamedTemporaryFile()`、`mkstemp()`、および `mkdtemp()` が提供されており、安全でない `mktemp()` を使いつづける必要をなくしました。このモジュールで生成されるテンポラリファイルはもはやプロセス番号を含みません; その代わりに、6 桁のランダムな文字からなる文字列が使われます。

また、ユーザから呼び出し可能な関数は全て、テンポラリファイルの場所や名前を直接操作できるようにするための追加の引数をとるようになりました。もはや変数 `tempdir` および `template` を使う必要はありません。以前のバージョンとの互換性を維持するために、引数の順番は多少変です; 明確さのためにキーワード引数を使うことをお勧めします。

このモジュールではユーザから呼び出し可能な以下の関数を定義しています:

```
tempfile.TemporaryFile([mode='w+b', bufsize=-1, suffix='', prefix='tmp', dir=None])
```

一時的な記憶領域として使うことができるファイルライク (file-like) オブジェクトを返します。ファイルは `mkstemp()` を使って生成されます。このファイルは閉じられると (オブジェクトがガーベジコレクションされた際に、暗黙のうちに閉じられる場合を含みます) すぐに消去されます。Unix 環境では、ファイルが生成されるとすぐにそのファイルのディレクトリエントリは除去されてしまいます。一方、他のプラットフォームではこの機能はサポートされていません; 従って、コードを書くときには、この関数で作成したテンポラリファイルをファイルシステム上で見ることができる、あるいはできないということをあてにすべきではありません。

生成されたファイルを一旦閉じなくてもファイルを読み書きできるようにするために、`mode` パラメータは標準で `'w+b'` に設定されています。ファイルに記録するデータが何であるかに関わらず全てのプラットフォームで一貫性のある動作をさせるために、バイナリモードが使われています。`bufsize` の値は標準で `-1` で、これはオペレーティングシステムにおける標準の値を使うことを意味しています。

`dir`、`prefix` および `suffix` パラメータは `mkstemp()` に渡されます。

返されるオブジェクトは、POSIX プラットフォームでは本物の `file` オブジェクトです。それ以外のプラットフォームではファイルライクオブジェクトが返され、`file` 属性に本物の `file` オブジェクトがあります。このファイルライクオブジェクトは、通常のファイルと同じように `with` 文で利用することができます。

```
tempfile.NamedTemporaryFile([mode='w+b', bufsize=-1, suffix='', prefix='tmp',
                             dir=None, delete=True])
```

この関数はファイルがファイルシステム上で見ることができるよう保証されている点を除き、`TemporaryFile()` と全く同じに働きます (Unix では、ディレクトリエントリは `unlink` されます)。ファイル名は返されるファイルライクオブジェクトの `name` メンバから取得することができます。このファイル名を使ってテンポラリファイルをもう一度開くことができるかどうかは、プラットフォームによって異なります (Unix では可能です; Windows NT 以降では開けません)。`delete` が `true` (デフォルト) の場合、ファイルは閉じられるとすぐに削除されます。

返されるオブジェクトは、常にファイルライクオブジェクトです。このオブジェクトの `file` 属性が本物の `file` オブジェクトになります。このファイルライクオブジェクトは、通常のファイルと同じように `with` 文を利用することができます。

バージョン 2.3 で追加。

バージョン 2.6 で追加: `delete` 引数

```
tempfile.SpooledTemporaryFile([max_size=0, mode='w+b', bufsize=-1, suffix='',
                               fix='tmp', dir=None])
```

この関数は、ファイルサイズが `max_size` を超えるか、`fileno()` メソッドが呼ばれるまでの間メモリ上で処理される以外は、`TemporaryFile()` と同じです。`max_size` を超えるか `fileno()` が呼ばれたとき、テンポラリファイルの内容がディスクに書き込まれ、その後の処理は `TemporaryFile()` で行われます。また、これの `truncate` メソッドは `size` 引数を受け付けません。

この関数が返すファイルは、追加で 1 つのメソッド `rollover()` を持っています。このメソッドが呼

ばれると、(サイズに関係なく) メモリからディスクへのロールオーバーが実行されます。

返されるオブジェクトはファイルライクオブジェクトで、その `_file` 属性は、`rollover()` が呼ばれたかどうかによって、`StringIO` オブジェクトか、本物のファイルオブジェクトになります。このファイルライクオブジェクトは、通常のファイルオブジェクトと同じように、`with` 文で利用することができます。

バージョン 2.6 で追加。

```
tempfile.mkstemp([suffix="", prefix='tmp', dir=None, text=False])
```

可能な限り最も安全な手段でテンポラリファイルを作成します。使用するプラットフォームで `os.open()` の `os.O_EXCL` フラグが正しく実装されている限り、ファイルの生成で競合状態が起こることはありません。このファイルは、ファイルを作成したユーザのユーザ ID からのみ読み書き可能です。使用するプラットフォームにおいて、ファイルを実行可能かどうかを示す許可ビットが使われている場合、ファイルは誰からも実行不可なように設定されます。このファイルのファイル記述子は子プロセスに継承されません。

`TemporaryFile()` と違って、`mkstemp()` で生成されたファイルが用済みになったときにファイルを消去するのはユーザの責任です。

`suffix` が指定された場合、ファイル名は指定された `suffix` で終わります。そうでない場合には `suffix` は付けられません。`mkstemp()` はファイル名と `suffix` の間にドットを追加しません; 必要なら、`suffix` の先頭につけてください。

`prefix` が指定された場合、ファイル名は指定されたプレフィクス (接頭文字列) で始まります; そうでない場合、標準のプレフィクスが使われます。

`dir` が指定された場合、テンポラリファイルは指定されたディレクトリ下に作成されます; そうでない場合、標準のディレクトリが使われます。デフォルトのディレクトリは、プラットフォームごとに異なるリストから選ばれます。しかし、アプリケーションのユーザは `TMPDIR`, `TEMP`, `TMP` 環境変数を設定することで、その場所を設定することができます。そのため、生成されたファイル名について、クォート無しで `os.popen()` を使って外部コマンドに渡せるかどうかなどの保証はありません。

`text` が指定された場合、ファイルをバイナリモード (標準の設定) かテキストモードで開くかを示します。使用するプラットフォームによってはこの値を設定しても変化はありません。

`mkstemp()` は開かれたファイルを扱うための OS レベル (`os.open()` が返すものと同じ) とファイルの絶対パス名が順番に並んだタプルを返します。

バージョン 2.3 で追加。

```
tempfile.mkdtemp([suffix="", prefix='tmp', dir=None])
```

可能な限り安全な方法でテンポラリディレクトリを作成します。ディレクトリの生成で競合状態は発生しません。ディレクトリを作成したユーザ ID だけが、このディレクトリに対して内容を読み出ししたり、書き込んだり、検索したりすることができます。

`mkdtemp()` によって作られたディレクトリとその内容が用済みになった時にそれを消去するのはユーザの責任です。

`prefix`, `suffix`, `dir` 引数は `mkstemp()` 関数のものと同じです。

`mkdtemp()` は新たに生成されたディレクトリの絶対パス名を返します。

バージョン 2.3 で追加。

`tempfile.mktemp([suffix="", prefix='tmp', dir=None])`

バージョン 2.3 で非推奨: 代わりに `mkstemp()` を使って下さい。

テンポラリファイルの絶対パス名を返します。このパス名は少なくともこの関数が呼び出された時点ではファイルシステム中に存在しなかったパス名です。 `prefix`、`prefix`、`suffix`、および `dir` 引数は `mkstemp()` のものと同じです。

警告: この関数を使うとプログラムのセキュリティホールになる可能性があります。この関数がファイル名を返した後、あなたがそのファイル名を使って次に何かをしようとする段階に至る前に、誰か他の人間があなたを出し抜くことができます。 `mktemp()` の利用は、`NamedTemporaryFile()` に `delete=False` 引数を渡すことで、簡単に置き換えることができます:

```
>>> f = NamedTemporaryFile(delete=False)
>>> f
<open file '<fdopen>', mode 'w+b' at 0x384698>
>>> f.name
'/var/folders/5q/5qTPn6xq2RaWqk+1Ytw3-U+++TI/-Tmp-/tmpG7V1Y0'
>>> f.write("Hello World!\n")
>>> f.close()
>>> os.unlink(f.name)
>>> os.path.exists(f.name)
False
```

このモジュールは一時的な名前の作成法を指定するグローバル変数を使います。それらの変数は上記のいずれかの関数を最初に呼び出した際に初期化されます。関数呼び出しを行うユーザはこれらの値を変更することができますが、推奨されていません。その代わりに関数に適切な引数を使ってください。

`tempfile.tempdir`

この値が `None` 以外に設定された場合、このモジュールで定義されている関数全ての `dir` 引数に対する標準の設定値となります。

`tempdir` が設定されていないか `None` の場合、上記のいずれかの関数を呼び出した際は常に、Python は標準的なディレクトリ候補のリストを検索し、関数を呼び出しているユーザの権限でファイルを作成できる最初のディレクトリ候補を `tempdir` に設定します。リストは以下のようになっています:

1. 環境変数 `TMPDIR` で与えられているディレクトリ名。
2. 環境変数 `TEMP` で与えられているディレクトリ名。
3. 環境変数 `TMP` で与えられているディレクトリ名。
4. プラットフォーム依存の場所:
 - RiscOS では環境変数 `Wimp$ScrapDir` で与えられているディレクトリ名。

- Windows ではディレクトリ `C:\TEMP`、`C:\TMP`、`\TEMP`、および `\TMP` の順。
- その他の全てのプラットフォームでは、`/tmp`、`/var/tmp`、および `/usr/tmp` の順。

5. 最後の手段として、現在の作業ディレクトリ。

`tempfile.gettempdir()`

現在選択されている、テンポラリファイルを作成するためのディレクトリを返します。 `tempdir` が `None` でない場合、単にその内容を返します; そうでない場合には上で記述されている検索が実行され、その結果が返されます。

バージョン 2.3 で追加。

`tempfile.template`

バージョン 2.0 で非推奨: 代わりに `gettempprefix()` を使ってください。

この値に `None` 以外の値を設定した場合、`mktemp()` が返すファイル名のディレクトリ部を含まない先頭部分 (プレフィクス) を定義します。ファイル名を一意にするために、6 つのランダムな文字および数字がこのプレフィクスの後に追加されます。デフォルトのプレフィックスは `tmp` です。

このモジュールの古いバージョンでは、`os.fork()` を呼び出した後に `template` を `None` に設定することが必要でした; この仕様はバージョン 1.5.2 からは必要なくなりました。

`tempfile.gettempprefix()`

テンポラリファイルを生成する際に使われるファイル名の先頭部分を返します。この先頭部分にはディレクトリ部は含まれません。変数 `template` を直接読み出すよりもこの関数を使うことを勧めます。

バージョン 1.5.2 で追加。

10.7 glob — Unix 形式のパス名のパターン展開

ソースコード: [Lib/glob.py](#)

`glob` モジュールは Unix シェルで使われているルールに従い指定されたパターンに一致するすべてのパス名を見つけ出します。返される結果の順序は不定です。チルダは展開されませんが、`*`、`?`、および `[]` で表現される文字範囲については正しくマッチされます。これは、関数 `os.listdir()` および `fnmatch.fnmatch()` を使用して行われており、実際にサブシェルを呼び出しているわけではありません。 `fnmatch.fnmatch()` と異なり、`glob` はドット (`.`) で始まるファイル名は特別扱いする点に注意してください。(チルダおよびシェル変数の展開を利用したい場合は `os.path.expanduser()` および `os.path.expandvars()` を使用してください。)

文字通りにマッチさせる場合はメタ文字を括弧に入れてください。例えば、`'[?]'` は文字 `'?'` にマッチします。

`glob.glob(pathname)`

`pathname` (パスの指定を含んだ文字列でなければいけません。) にマッチする空の可能性のあるパス名のリストを返します。 `pathname` は (`/usr/src/Python-1.5/Makefile` のように) 絶対パスでも

いいし、(`../../Tools/*/*.gif` のように) 相対パスでもよくて、シェル形式のワイルドカードを含んでいてもかまいません。結果には (シェルと同じく) 壊れたシンボリックリンクも含まれます。

`glob.iglob(pathname)`

実際には一度に全てを格納せずに、`glob()` と同じ値を順に生成する **イテレータ** を返します。

バージョン 2.5 で追加。

たとえば、次の三つのファイルだけがあるディレクトリを考えてください: `1.gif`、`2.txt`、`card.gif`。`glob()` は次のような結果になります。パスに接頭するどの部分が保たれているかに注意してください。

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
```

ディレクトリが `.` で始まるファイルを含んでいる場合、デフォルトでそれらはマッチしません。例えば、`card.gif` と `.card.gif` を含むディレクトリを考えてください:

```
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

参考:

`fnmatch` モジュール シェル形式の (パスではない) ファイル名展開

10.8 fnmatch — Unix ファイル名のパターンマッチ

ソースコード: [Lib/fnmatch.py](#)

このモジュールは Unix のシェル形式のワイルドカードへの対応を提供しますが、(`re` モジュールでドキュメント化されている) 正規表現と同じではありません。シェル形式のワイルドカードで使われる特別な文字は:

Pattern	意味
<code>*</code>	すべてにマッチします
<code>?</code>	任意の一文字にマッチします
<code>[seq]</code>	<code>seq</code> にある任意の文字にマッチします
<code>[!seq]</code>	<code>seq</code> にない任意の文字にマッチします

文字通りにマッチさせる場合はメタ文字を括弧に入れてください。例えば、`'[?]'` は文字 `'?'` にマッチします。

ファイル名の区切り文字 (Unix では '/') はこのモジュールに固有なものではないことに注意してください。パス名展開については、`glob` モジュールを参照してください (`glob` はパス名の部分にマッチさせるのに `filter()` を使っています)。同様に、ピリオドで始まるファイル名はこのモジュールに固有ではなくて、`*` と `?` のパターンでマッチします。

`fnmatch.fnmatch(filename, pattern)`

`filename` の文字列が `pattern` の文字列にマッチするかテストして、`True`、`False` のいずれかを返します。どちらの引数とも `os.path.normcase()` を使って、大小文字が正規化されます。オペレーティングシステムが標準でどうなっているかに関係なく、大小文字を区別して比較する場合には、`fnmatchcase()` が使えます。

次の例では、カレントディレクトリにある、拡張子が `.txt` である全てのファイルを表示しています:

```
import fnmatch
import os

for file in os.listdir('.'):
    if fnmatch.fnmatch(file, '*.txt'):
        print file
```

`fnmatch.fnmatchcase(filename, pattern)`

`filename` が `pattern` にマッチするかテストして、`True`、`False` を返します。比較は大文字、小文字を区別し、`os.path.normcase()` は適用しません。

`fnmatch.filter(names, pattern)`

`pattern` にマッチする `names` のリストの部分集合を返します。[`n for n in names if fnmatch(n, pattern)`] と同じですが、もっと効率よく実装しています。

バージョン 2.2 で追加。

`fnmatch.translate(pattern)`

シェルスタイルの `pattern` を、`re.match()` で使用するための正規表現に変換して返します。

例:

```
>>> import fnmatch, re
>>>
>>> regex = fnmatch.translate('*.txt')
>>> regex
'.*\\.txt\\Z(?ms) '
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<_sre.SRE_Match object at 0x...>
```

参考:

`glob` モジュール Unix シェル形式のパス展開。

10.9 linecache — テキストラインにランダムアクセスする

ソースコード: [Lib/linecache.py](#)

`linecache` モジュールは、キャッシュ (一つのファイルから何行も読んでおくのが一般的です) を使って、内部で最適化を図りつつ、任意のファイルの任意の行を取得するのを可能にします。 `traceback` モジュールは、整形されたトレースバックにソースコードを含めるためにこのモジュールを利用しています。

`linecache` モジュールでは次の関数が定義されています:

`linecache.getline(filename, lineno[, module_globals])`

`filename` という名前のファイルから `lineno` 行目を取得します。この関数は決して例外を発生させません — エラーの際には `''` を返します (行末の改行文字は、見つかった行に含まれます)。

`filename` という名前のファイルが見つからなかった場合、モジュールの、つまり、`sys.path` でそのファイルを探します。 `zipfile` やその他のファイルシステムでない `import` 元に対応するためまず `module_globals` の `PEP 302` `__loader__` をチェックし、そのあと `sys.path` を探索します。

バージョン 2.5 で追加: パラメータ `module_globals` の追加。

`linecache.clearcache()`

キャッシュをクリアします。それまでに `getline()` を使って読み込んだファイルの行が必要でなくなったら、この関数を使ってください。

`linecache.checkcache([filename])`

キャッシュが有効かチェックします。キャッシュしたファイルにディスク上で変更があったかもしれなくて、更新が必要なときにこの関数を使ってください。もし `filename` がなければ、全てのキャッシュエントリをチェックします。

例:

```
>>> import linecache
>>> linecache.getline('/etc/passwd', 4)
'sys:x:3:3:sys:/dev:/bin/sh\n'
```

10.10 shutil — 高水準のファイル操作

ソースコード: [Lib/shutil.py](#)

`shutil` モジュールはファイルやファイルの集まりに対する高水準の操作方法を多数提供します。特にファイルのコピーや削除のための関数が用意されています。個別のファイルに対する操作については、`os` モジュールも参照してください。

警告: 高レベルなファイルコピー関数 (`shutil.copy()`, `shutil.copy2()`) でも、全てのファイルのメタデータをコピーできるわけではありません。

POSIX プラットフォームでは、これは ACL やファイルのオーナー、グループが失われることを意味しています。Mac OS では、リソースフォーク (resource fork) やその他のメタデータが利用されません。これは、リソースが失われ、ファイルタイプや生成者コード (creator code) が正しくなくなることを意味しています。Windows では、ファイルオーナー、ACL、代替データストリームがコピーされません。

10.10.1 ディレクトリとファイルの操作

`shutil.copyfileobj(fsrc, fdst[, length])`

ファイル形式のオブジェクト `fsrc` の内容を `fdst` へコピーします。整数値 `length` は与えられた場合バッファサイズを表します。特に `length` が負の場合、チャンク内のソースデータを繰り返し操作することなくデータをコピーします。デフォルトでは、制御不能なメモリ消費を避けるためにデータはチャンク内に読み込まれます。`fsrc` オブジェクトの現在のファイル位置が 0 でない場合、現在の位置からファイル終端までの内容のみがコピーされることに注意してください。

`shutil.copyfile(src, dst)`

`src` で指定されたファイルの内容を `dst` で指定されたファイルへとコピーします。(メタデータはコピーされません) `dst` は完全なターゲットファイル名である必要があります。コピー先にディレクトリ名を使用したい場合は、`shutil.copy()` を参照してください。もし、`src` と `dst` が同じファイルであれば、`Error` 例外が発生します。コピー先は書き込み可能である必要があります。そうでなければ `IOError` を発生します。もし `dst` が存在したら、置き換えられます。キャラクタやブロックデバイス、パイプ等の特別なファイルはこの関数ではコピーできません。`src` と `dst` にはパス名を文字列で与えられます。

`shutil.copymode(src, dst)`

`src` から `dst` へパーミッションをコピーします。ファイル内容や所有者、グループは影響を受けません。`src` と `dst` には文字列としてパス名を与えられます。

`shutil.copystat(src, dst)`

`src` から `dst` へ、パーミッション、最終アクセス時間、最終更新時間、フラグをコピーします。ファイル内容や所有者、グループは影響を受けません。`src` と `dst` には文字列としてパス名を与えられます。

`shutil.copy(src, dst)`

ファイル `src` をファイルまたはディレクトリ `dst` へコピーします。もし、`dst` がディレクトリであればファイル名は `src` と同じものが指定されたディレクトリ内に作成 (または上書き) されます。パーミッションはコピーされます。`src` と `dst` には文字列としてパス名を与えられます。

`shutil.copy2(src, dst)`

`copy2()` はファイルのメタデータを保持しようとするのを除けば `copy()` と等価です。

`copy2()` uses `copystat()` to copy the file metadata. Please see `copystat()` for more information.

`shutil.ignore_patterns(*patterns)`

このファクトリ関数は、`copytree()` 関数の `ignore` 引数に渡すための呼び出し可能オブジェクトを

作成します。glob 形式の *patterns* にマッチするファイルやディレクトリが無視されます。下の例を参照してください。

バージョン 2.6 で追加。

`shutil.copytree(src, dst, symlinks=False, ignore=None)`

src を起点としたディレクトリツリーをコピーします。*dst* で指定されたターゲットディレクトリは、既存のもので無い必要があります。存在しない親ディレクトリも含めて作成されます。パーミッションと時刻は `copystat()` 関数でコピーされます。個々のファイルは `shutil.copy2()` によってコピーされます。

symlinks が真であれば、元のディレクトリ内のシンボリックリンクはコピー先のディレクトリ内へシンボリックリンクとしてコピーされます。偽が与えられたり省略された場合は元のディレクトリ内のリンクの対象となっているファイルがコピー先のディレクトリ内へコピーされます。

ignore は `copytree()` が走査しているディレクトリと `os.listdir()` が返すその内容のリストを引数として受け取ることで呼出し可能オブジェクトでなければなりません。`copytree()` は再帰的に呼び出されるので、*ignore* はコピーされる各ディレクトリ毎に呼び出されます。*ignore* の戻り値はカレントディレクトリに相対的なディレクトリ名およびファイル名のシーケンス（すなわち第二引数の項目のサブセット）でなければなりません。それらの名前はコピー中に無視されます。`ignore_patterns()` を用いて glob 形式のパターンによって無視する呼出し可能オブジェクトを作成することが出来ます。

例外が発生した場合、理由のリストとともに `Error` を送出します。

この関数は、究極の道具としてではなく、ソースコードが利用例になっていると捉えるべきでしょう。

バージョン 2.3 で変更: コピー中にエラーが発生した場合、メッセージを出力するのではなく `Error` を起こす。

バージョン 2.5 で変更: *dst* を作成する際に中間のディレクトリ作成が必要な場合、エラーを起こすのではなく作成する。ディレクトリのパーミッションと時刻を `copystat()` を利用してコピーする。

バージョン 2.6 で変更: 何がコピーされるかを制御するための *ignore* 引数

`shutil.rmtree(path[, ignore_errors[, onerror]])`

ディレクトリツリー全体を削除します。*path* はディレクトリを指しているなければなりません（ただしディレクトリに対するシンボリックリンクではいけません）。もし *ignore_errors* が真であれば削除に失敗したことによるエラーは無視されます。偽や省略された場合はこれらのエラーは *onerror* で与えられたハンドラを呼び出して処理され、*onerror* が省略された場合は例外を送出します。

onerror が与えられた場合、それは 3 つのパラメータ *function*, *path* および *excinfo* を受け入れて呼出し可能のものでなくてはなりません。最初のパラメータ *function* は例外を引き起こした関数で `os.listdir()`, `os.remove()`, `os.rmdir()` のいずれかでしょう。2 番目のパラメータ *path* は *function* へ渡されたパス名です。3 番目のパラメータ *excinfo* は `sys.exc_info()` で返されるような例外情報になるでしょう。*onerror* が引き起こす例外はキャッチできません。

バージョン 2.6 で変更: *path* を明示的にチェックして、シンボリックリンクだった場合は `OSError` を返すようになりました。

`shutil.move(src, dst)`

再帰的にファイルやディレクトリ (*src*) を別の場所 (*dst*) へ移動します。

移動先が存在するディレクトリの場合、*src* はそのディレクトリの中へ移動します。移動先が存在していてそれがディレクトリでない場合、`os.rename()` の動作によっては上書きされることがあります。

もし移動先が現在のファイルシステム上であれば `os.rename()` を使います。そうでない場合は *src* を (`shutil.copy2()` で) *dst* にコピーし、その後コピー元は削除されます。

バージョン 2.3 で追加。

exception `shutil.Error`

この例外は複数ファイルの操作を行っているときに生じる例外をまとめたものです。`copytree()` に対しては例外の引数は 3 つのタプル (*srcname*, *dstname*, *exception*) からなるリストです。

バージョン 2.3 で追加。

copytree の例

以下は前述の `copytree()` 関数のドキュメント文字列を省略した実装例です。本モジュールで提供される他の関数の使い方を示しています。

```
def copytree(src, dst, symlinks=False, ignore=None):
    names = os.listdir(src)
    if ignore is not None:
        ignored_names = ignore(src, names)
    else:
        ignored_names = set()

    os.makedirs(dst)
    errors = []
    for name in names:
        if name in ignored_names:
            continue
        srcname = os.path.join(src, name)
        dstname = os.path.join(dst, name)
        try:
            if symlinks and os.path.islink(srcname):
                linkto = os.readlink(srcname)
                os.symlink(linkto, dstname)
            elif os.path.isdir(srcname):
                copytree(srcname, dstname, symlinks, ignore)
            else:
                copy2(srcname, dstname)
            # XXX What about devices, sockets etc.?
        except (IOError, os.error) as why:
            errors.append((srcname, dstname, str(why)))
        # catch the Error from the recursive copytree so that we can
        # continue with other files
    except Error as err:
        errors.extend(err.args[0])
```

(次のページに続く)

(前のページからの続き)

```
try:
    copystat(src, dst)
except WindowsError:
    # can't copy file access times on Windows
    pass
except OSError as why:
    errors.extend((src, dst, str(why)))
if errors:
    raise Error(errors)
```

`ignore_patterns()` ヘルパ関数を利用する、もう 1 つの例です。

```
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*'))
```

この例では、`.pyc` ファイルと、`tmp` で始まる全てのファイルやディレクトリを除いて、全てをコピーします。

`ignore` 引数にロギングさせる別の例です。

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s' % path)
    return [] # nothing will be ignored

copytree(source, destination, ignore=_logpath)
```

10.10.2 アーカイブ化操作

圧縮とアーカイブ化されているファイルの読み書きの高水準なユーティリティも提供されています。これらは `zipfile`、`tarfile` モジュールに依拠しています。

`shutil.make_archive(base_name, format[, root_dir[, base_dir[, verbose[, dry_run[, owner[, group[, logger]]]]]])`
アーカイブファイル (zip や tar など) を作成し、その名前を返します。

`base_name` は作成するファイルの名前で、パスを含み、フォーマットごとの拡張子を抜いたものです。`format` はアーカイブフォーマットで "zip" (`zlib` モジュールもしくは外部の zip 実行可能ファイルが利用可能な場合), "tar", "gztar" (`zlib` モジュールが利用可能な場合), "bztar" (`bz2` モジュールが利用可能な場合) のいずれかです。

`root_dir` は、アーカイブのルートディレクトリとなるディレクトリです。すなわち、一般的には、アーカイブを作成する前に `root_dir` をカレントディレクトリにします。

`base_dir` は、アーカイブを開始するディレクトリです。すなわち、`base_dir` は、アーカイブのすべてのファイルとディレクトリに共通する接頭辞になります。

`root_dir` と `base_dir` のどちらも、デフォルトはカレントディレクトリです。

`owner` と `group` は、tar アーカイブを作成するときに使われます。デフォルトでは、カレントのオーナーとグループを使います。

`logger` は [PEP 282](#) に互換なオブジェクトでなければなりません。これは普通は `logging.Logger` のインスタンスです。

バージョン 2.7 で追加。

`shutil.get_archive_formats()`

アーカイブ化をサポートしているフォーマットのリストを返します。返されるシーケンスのそれぞれの要素は、タプル (`name`, `description`) です。

デフォルトでは、`shutil` はこれらのフォーマットを提供しています：

- `zip`: ZIP ファイル (`zlib` モジュールもしくは外部の `zip` 実行可能ファイルが利用可能な場合)。
- `tar`: 圧縮されていない tar ファイル。
- `gztar`: gzip で圧縮された tar ファイル (`zlib` モジュールが利用可能な場合)。
- `bztar`: bzip2 で圧縮された tar ファイル (`bz2` モジュールが利用可能な場合)。

`register_archive_format()` を使って、新しいフォーマットを登録したり、既存のフォーマットに独自のアーカイバを提供したりできます。

バージョン 2.7 で追加。

`shutil.register_archive_format(name, function[, extra_args[, description]])`

フォーマット `name` のアーカイバを登録します。 `function` は、アーカイバを呼び出すのに使われる呼び出し可能オブジェクトです。

`extra_args` は、与えられるなら (`name`, `value`) のシーケンスで、アーカイバ呼び出し可能オブジェクトが使われるときの追加キーワード引数に使われます。

`description` は、アーカイバのリストを返す `get_archive_formats()` で使われます。デフォルトでは、空のリストです。

バージョン 2.7 で追加。

`shutil.unregister_archive_format(name)`

アーカイブフォーマット `name` を、サポートされているフォーマットのリストから取り除きます。

バージョン 2.7 で追加。

アーカイブ化の例

この例では、ユーザの `.ssh` ディレクトリにあるすべてのファイルを含む、gzip された tar ファイルアーカイブを作成します：

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
>>> make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

結果のアーカイブは、以下のものを含まます:

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
drwx----- tarek/staff      0 2010-02-01 16:23:40 ./
-rw-r--r-- tarek/staff    609 2008-06-09 13:26:54 ./authorized_keys
-rwxr-xr-x tarek/staff     65 2008-06-09 13:26:54 ./config
-rwx----- tarek/staff    668 2008-06-09 13:26:54 ./id_dsa
-rwxr-xr-x tarek/staff    609 2008-06-09 13:26:54 ./id_dsa.pub
-rw----- tarek/staff   1675 2008-06-09 13:26:54 ./id_rsa
-rw-r--r-- tarek/staff    397 2008-06-09 13:26:54 ./id_rsa.pub
-rw-r--r-- tarek/staff  37192 2010-02-06 18:23:10 ./known_hosts
```

10.11 dircache — キャッシュされたディレクトリ一覧の生成

バージョン 2.6 で非推奨: *dircache* モジュールは Python 3 で削除されました。

dircache モジュールはキャッシュされた情報を使ってディレクトリ一覧を読み出すための関数を定義しています。キャッシュはディレクトリの *mtime* に応じて無効化されます。さらに、一覧中のディレクトリにスラッシュ ('/') を追加することでディレクトリであると分かるようにするための関数も定義しています。

dircache モジュールは以下の関数を定義しています:

`dircache.reset()`

ディレクトリキャッシュをリセットします。

`dircache.listdir(path)`

`os.listdir()` によって得た *path* のディレクトリ一覧を返します。 *path* を変えない限り、以降の `listdir()` を呼び出してもディレクトリ構造を読み込みなおすことはしないので注意してください。

返されるリストは読み出し専用であると見なされるので注意してください (おそらく将来のバージョンではタプルを返すように変更されるはず ? です)。

`dircache.opendir(path)`

`listdir()` と同じです。以前のバージョンとの互換性のために定義されています。

`dircache.annotate(head, list)`

list を *head* の相対パスからなるリストとして、各パスがディレクトリを指す場合には '/' をパス名の後ろに追加したものに置き換えます。

```
>>> import dircache
>>> a = dircache.listdir('/')
```

(次のページに続く)

(前のページからの続き)

```
>>> a = a[:] # Copy the return value so we can change 'a'
>>> a
['bin', 'boot', 'cdrom', 'dev', 'etc', 'floppy', 'home', 'initrd', 'lib', 'lost+
found', 'mnt', 'proc', 'root', 'sbin', 'tmp', 'usr', 'var', 'vmlinuz']
>>> dircache.annotate('/', a)
>>> a
['bin/', 'boot/', 'cdrom/', 'dev/', 'etc/', 'floppy/', 'home/', 'initrd/', 'lib/
', 'lost+found/', 'mnt/', 'proc/', 'root/', 'sbin/', 'tmp/', 'usr/', 'var/', 'vm
linuz']
```

10.12 macpath — Mac OS 9 のパス操作関数

このモジュールは `os.path` モジュールの Macintosh 9 (およびそれ以前) 用の実装です。これを使用すると、古い形式の Macintosh のパス名を Mac OS X (あるいはその他の任意のプラットフォーム) 上で扱うことができます。

次の関数がこのモジュールで利用できます。 `normcase()`、`normpath()`、`isabs()`、`join()`、`split()`、`isdir()`、`isfile()`、`walk()`、`exists()`。 `os.path` で利用できる他の関数については、ダミーの関数として相当する物が利用できます。

参考:

[ファイルオブジェクト](#) 節 Python 組み込みのファイルオブジェクト。

`os` モジュール オペレーティングシステムのインタフェース、Python の [ファイルオブジェクト](#) より低レベルでのファイル操作を含みます。

第 11 章

データの永続化

この章で解説されるモジュール群は Python データをディスクに永続的な形式で保存します。モジュール `pickle` とモジュール `marshal` は多くの Python データ型をバイト列に変換し、バイト列から再生成します。様々な DBM 関連モジュールはハッシュを基にした、文字列から他の文字列へのマップを保存するファイルフォーマット群をサポートします。モジュール `bsddb` はディスクベースの文字列から文字列へのマッピングを、ハッシュ、B-Tree、レコードを基にしたフォーマットで提供します。

この章で解説されるモジュールのリスト:

11.1 `pickle` — Python オブジェクトの直列化

`pickle` モジュールでは、Python オブジェクトデータ構造を直列化 (serialize) したり非直列化 (de-serialize) するための基礎的ですが強力なアルゴリズムを実装しています。"Pickle 化 (Pickling)" は Python のオブジェクト階層をバイトストリームに変換する過程を指します。"非 Pickle 化 (unpickling)" はその逆の操作で、バイトストリームをオブジェクト階層に戻すように変換します。Pickle 化 (及び非 Pickle 化) は、別名 "直列化 (serialization)" や "整列化 (marshalling)"^{*1}、"平坦化 (flattening)" として知られていますが、ここでは混乱を避けるため、用語として "Pickle 化" および "非 Pickle 化" を使います。

このドキュメントでは `pickle` モジュールおよび `cPickle` モジュールの両方について記述します。

警告: `pickle` モジュールはエラーや不正に生成されたデータに対して安全ではありません。信頼できない、あるいは認証されていないソースから受け取ったデータを unpickle してはいけません。

11.1.1 他の Python モジュールとの関係

`pickle` モジュールには `cPickle` と呼ばれる最適化のなされた親類モジュールがあります。名前が示すように、`cPickle` は C で書かれており、このため `pickle` より 1000 倍くらいまで高速になる可能性があります。しかしながら `cPickle` では `Pickler()` および `Unpickler()` クラスのサブクラス化をサポートしていません。これは `cPickle` では、これらは関数であってクラスではないからです。ほとんどのアプリ

^{*1} `marshal` モジュールと間違えないように注意してください。

ケーションではこの機能は不要であり、`cPickle` の持つ高いパフォーマンスの恩恵を受けることができます。その他の点では、二つのモジュールにおけるインタフェースはほとんど同じです; このマニュアルでは共通のインタフェースを記述しており、必要に応じてモジュール間の相違について指摘します。以下の議論では、`pickle` と `cPickle` の総称として ”pickle” という用語を使うことにします。

これら二つのモジュールが生成するデータストリームは相互交換できることが保証されています。

Python には `marshal` と呼ばれるより原始的な直列化モジュールがありますが、一般的に Python オブジェクトを直列化する方法としては `pickle` を選ぶべきです。 `marshal` は基本的に `.pyc` ファイルをサポートするために存在しています。

`pickle` モジュールはいくつかの点で `marshal` と明確に異なります:

- `pickle` モジュールでは、同じオブジェクトが再度直列化されることのないよう、すでに直列化されたオブジェクトについて追跡情報を保持します。 `marshal` はこれを行いません。

この機能は再帰的オブジェクトと共有オブジェクトの両方に重要な関わりをもっています。再帰的オブジェクトとは自分自身に対する参照を持っているオブジェクトです。再帰的オブジェクトは `marshal` で扱うことができず、実際、再帰的オブジェクトを `marshal` 化しようとする Python インタプリタをクラッシュさせてしまいます。共有オブジェクトは、直列化しようとするオブジェクト階層の異なる複数の場所で同じオブジェクトに対する参照が存在する場合に生じます。共有オブジェクトを共有のままにしておくことは、変更可能なオブジェクトの場合には非常に重要です。

- `marshal` はユーザ定義クラスやそのインスタンスを直列化するために使うことができません。 `pickle` はクラスインスタンスを透過的に保存したり復元したりすることができますが、クラス定義をインポートすることが可能で、かつオブジェクトが保存された際と同じモジュールで定義されていなければなりません。
- `marshal` の直列化フォーマットは Python の異なるバージョンで可搬性があることを保証していません。 `marshal` の本来の仕事は `.pyc` ファイルのサポートなので、Python を実装する人々には、必要に応じて直列化フォーマットを以前のバージョンと互換性のないものに変更する権限が残されています。 `pickle` 直列化フォーマットには、全ての Python リリース間で以前のバージョンとの互換性が保証されています。

直列化は永続化 (persistence) よりも原始的な概念です; `pickle` はファイルオブジェクトを読み書きしますが、永続化されたオブジェクトの名前付け問題や、(より複雑な) オブジェクトに対する競合アクセスの問題を扱いません。 `pickle` モジュールは複雑なオブジェクトをバイトストリームに変換することができ、バイトストリームを変換前と同じ内部構造をオブジェクトに変換することができます。このバイトストリームの最も明白な用途はファイルへの書き込みですが、その他にもネットワークを介して送信したり、データベースに記録したりすることができます。モジュール `shelve` はオブジェクトを DBM 形式のデータベースファイル上で pickle 化したり unpickle 化したりするための単純なインタフェースを提供しています。

11.1.2 データストリームの形式

`pickle` が使うデータ形式は Python 特有です。そうすることで、XDR のような外部の標準が持つ制限 (例えば XDR ではポインタの共有を表現できません) を課せられることがないという利点があります; しかしこれは

Python で書かれていないプログラムが pickle 化された Python オブジェクトを再構築できない可能性があることを意味します。

標準では、*pickle* データ形式では印字可能な ASCII 表現を使います。これはバイナリ表現よりも少しかさばるデータになります。印字可能な ASCII の利用 (とその他の *pickle* 表現形式が持つ特徴) の大きな利点は、デバッグやリカバリーを目的とした場合に、pickle 化されたファイルを標準的なテキストエディタで読めるということです。

現在、pickle 化に使われるプロトコルは、以下の 3 種類です。

- バージョン 0 のプロトコルは、最初の ASCII プロトコルで、以前のバージョンの Python と後方互換です。
- バージョン 1 のプロトコルは、古いバイナリ形式で、以前のバージョンの Python と後方互換です。
- バージョン 2 のプロトコルは、Python 2.3 で導入されました。 *new-style class* を、より効率よく pickle 化します。

詳細は [PEP 307](#) を参照してください。

protocol を指定しない場合、プロトコル 0 が使われます。 *protocol* に負値か *HIGHEST_PROTOCOL* を指定すると、有効なプロトコルの内、もっとも高いバージョンのものが使われます。

バージョン 2.3 で変更: *protocol* パラメータが導入されました。

protocol version ≥ 1 を指定することで、少しだけ効率の高いバイナリ形式を選ぶことができます。

11.1.3 使用法

オブジェクト階層を直列化するには、まず pickler を生成し、続いて pickler の *dump()* メソッドを呼び出します。データストリームから非直列化するには、まず unpickler を生成し、続いて unpickler の *load()* メソッドを呼び出します。 *pickle* モジュールでは以下の定数を提供しています:

`pickle.HIGHEST_PROTOCOL`

有効なプロトコルのうち、最も大きいバージョン。この値は、*protocol* として渡せます。

バージョン 2.3 で追加。

注釈: *protocols* ≥ 1 で作られた pickle ファイルは、常にバイナリモードでオープンするようにしてください。古い ASCII ベースの pickle プロトコル 0 では、矛盾しない限りにおいてテキストモードとバイナリモードのいずれも利用することができます。

プロトコル 0 で書かれたバイナリの pickle ファイルは、行ターミネータとして単独の改行 (LF) を含んでいて、ですのでこの形式をサポートしない、Notepad や他のエディタで見るときに「おかしく」見えるかもしれません。

この pickle 化の手続きを便利にするために、*pickle* モジュールでは以下の関数を提供しています:

`pickle.dump(obj, file[, protocol])`

すでに開かれているファイルオブジェクト *file* に、*obj* を pickle 化したものを表現する文字列を書き込みます。 `Pickler(file, protocol).dump(obj)` と同じです。

protocol を指定しない場合、プロトコル 0 が使われます。 *protocol* に負値か `HIGHEST_PROTOCOL` を指定すると、有効なプロトコルの内、もっとも高いバージョンのものが使われます。

バージョン 2.3 で変更: *protocol* パラメータが導入されました。

file は、単一の文字列引数を受理する `write()` メソッドを持たなければなりません。従って、*file* としては、書き込みのために開かれたファイルオブジェクト、 `StringIO` オブジェクト、その他前述のインタフェースに適合する他のカスタムオブジェクトをとることができます。

`pickle.load(file)`

すでに開かれているファイルオブジェクト *file* から文字列を読み出し、読み出された文字列を pickle 化されたデータ列として解釈して、もとのオブジェクト階層を再構築して返します。 `Unpickler(file).load()` と同じです。

file は、整数引数をとる `read()` メソッドと、引数の必要ない `readline()` メソッドを持たなければなりません。これらのメソッドは両方とも文字列を返さなければなりません。従って、*file* としては、読み出しのために開かれたファイルオブジェクト、 `StringIO` オブジェクト、その他前述のインタフェースに適合する他のカスタムオブジェクトをとることができます。

この関数はデータ列の書き込まれているモードがバイナリかそうでないかを自動的に判断します。

`pickle.dumps(obj[, protocol])`

obj の pickle 化された表現を、ファイルに書き込む代わりに文字列で返します。

protocol を指定しない場合、プロトコル 0 が使われます。 *protocol* に負値か `HIGHEST_PROTOCOL` を指定すると、有効なプロトコルの内、もっとも高いバージョンのものが使われます。

バージョン 2.3 で変更: *protocol* パラメータが追加されました。

`pickle.loads(string)`

pickle 化されたオブジェクト階層を文字列から読み出します。文字列中で pickle 化されたオブジェクト表現よりも後に続く文字列は無視されます。

`pickle` モジュールでは、以下の 3 つの例外も定義しています:

exception `pickle.PickleError`

下で定義されている他の例外で共通の基底クラスです。 `Exception` を継承しています。

exception `pickle.PicklingError`

この例外は unpickle 不可能なオブジェクトが `dump()` メソッドに渡された場合に送出されます。

exception `pickle.UnpicklingError`

この例外は、オブジェクトを unpickle 化する際に問題が発生した場合に送出されます。 unpickle 化中には `AttributeError`、`EOFError`、`ImportError`、および `IndexError` といった他の例外 (これだけとは限りません) も発生する可能性があるので注意してください。

`pickle` モジュールでは、2つの呼び出し可能オブジェクト^{*2}として、`Pickler` および `Unpickler` を提供しています:

```
class pickle.Pickler(file[, protocol])
```

`pickle` 化されたオブジェクトのデータ列を書き込むためのファイル類似のオブジェクトを引数にとります。

`protocol` を指定しない場合、プロトコル 0 が使われます。 `protocol` に負値が `HIGHEST_PROTOCOL` を指定すると、有効なプロトコルの内、もっとも高いバージョンのものが使われます。

バージョン 2.3 で変更: `protocol` パラメータが導入されました。

`file` は単一の文字列引数を受理する `write()` メソッドを持たなければなりません。従って、`file` としては、書き込みのために開かれたファイルオブジェクト、`StringIO` オブジェクト、その他前述のインタフェースに適合する他のカスタムオブジェクトをとることができます。

`Pickler` オブジェクトでは、一つ (または二つ) の public なメソッドを定義しています:

```
dump(obj)
```

コンストラクタで与えられた、すでに開かれているファイルオブジェクトに `obj` の `pickle` 化された表現を書き込みます。コンストラクタに渡された `protocol` 引数の値に応じて、バイナリおよび ASCII 形式が使われます。

```
clear_memo()
```

`pickler` の "メモ" を消去します。メモとは、共有オブジェクトまたは再帰的なオブジェクトが値ではなく参照で記憶されるようにするために、`pickler` がこれまでどのオブジェクトに遭遇してきたかを記憶するデータ構造です。このメソッドは `pickler` を再利用する際に便利です。

注釈: Python 2.3 以前では、`clear_memo()` は `cPickle` で生成された `pickler` でのみ利用可能でした。`pickle` モジュールでは、`pickler` は `memo` と呼ばれる Python 辞書型のインスタンス変数を持ちます。従って、`pickle` モジュールにおける `pickler` のメモを消去は、以下のようにしてできます:

```
mypickler.memo.clear()
```

以前のバージョンの Python での動作をサポートする必要のないコードでは、単に `clear_memo()` を使ってください。

同じ `Pickler` のインスタンスに対し、`dump()` メソッドを複数回呼び出すことは可能です。この呼び出しは、対応する `Unpickler` インスタンスで同じ回数だけ `load()` を呼び出す操作に対応します。同じオブジェクトが `dump()` を複数回呼び出して `pickle` 化された場合、`load()` は全て同じオブジェクトに対して参照を行います^{*3}。

^{*2} `pickle` では、これらの呼び出し可能オブジェクトはクラスであり、サブクラス化してその動作をカスタマイズすることができます。しかし、`cPickle` モジュールでは、これらの呼び出し可能オブジェクトはファクトリ関数であり、サブクラス化することができません。サブクラスを作成する共通の理由の一つは、どのオブジェクトを実際に `unpickle` するかを制御することです。詳細については `Unpickler` をサブクラス化する を参照してください。

^{*3} 警告: これは、複数のオブジェクトを `pickle` 化する際に、オブジェクトやそれらの一部に対する変更を妨げないようにするための

`Unpickler` オブジェクトは以下のように定義されています:

class `pickle.Unpickler` (*file*)

`pickle` データ列を読み出すためのファイル類似のオブジェクトを引数に取ります。このクラスはデータ列がバイナリモードかどうかを自動的に判別します。従って、`Pickler` のファクトリメソッドのようなフラグを必要としません。

file は、整数引数をとる `read()` メソッドと、引数の必要ない `readline()` メソッドを持たなければなりません。これらのメソッドは両方とも文字列を返さなければなりません。従って、*file* としては、読み出しのために開かれたファイルオブジェクト、`StringIO` オブジェクト、その他前述のインタフェースに適合する他のカスタムオブジェクトをとることができます。

`Unpickler` オブジェクトは 1 つ (または 2 つ) の public なメソッドを持っています:

load()

コンストラクタで渡されたファイルオブジェクトからオブジェクトの `pickle` 化表現を読み出し、中に収められている再構築されたオブジェクト階層を返します。

このメソッドは自動的にデータストリームがバイナリモードで書き出されているかどうかを判別します。

loads()

`load()` に似ていますが、実際には何もオブジェクトを生成しないという点が違います。この関数は第一に `pickle` 化データ列中で参照されている、“永続化 id” と呼ばれている値を検索する上で便利です。詳細は以下の [pickle 化プロトコル](#) を参照してください。

注意: `loads()` メソッドは現在 `cPickle` モジュールで生成された `Unpickler` オブジェクトのみで利用可能です。`pickle` モジュールの `Unpickler` には、`loads()` メソッドがありません。

11.1.4 何を `pickle` 化したり `unpickle` 化できるのか?

以下の型は `pickle` 化できます:

- `None`、`True`、および `False`
- 整数、長整数、浮動小数点数、複素数
- 通常文字列および `Unicode` 文字列
- `pickle` 化可能なオブジェクトからなるタプル、リスト、集合および辞書
- モジュールのトップレベルで定義されている関数
- モジュールのトップレベルで定義されている組み込み関数

仕様です。あるオブジェクトに変更を加えて、その後同じ `Pickler` を使って再度 `pickle` 化しようとしても、そのオブジェクトは `pickle` 化しなおされません — そのオブジェクトに対する参照が `pickle` 化され、`Unpickler` は変更された値ではなく、元の値を返します。これには 2 つの問題点: (1) 変更の検出、そして (2) 最小限の変更を整列化すること、があります。ガーベジコレクションもまた問題になります。

- モジュールのトップレベルで定義されているクラス
- `__dict__` 属性を持つクラス、あるいは `__getstate__()` メソッドの返り値が pickle 化可能なクラス (詳細は [pickle 化プロトコル](#) を参照)。

pickle 化できないオブジェクトを pickle 化しようとすると、`PicklingError` 例外が送出されます; この例外が起きた場合、背後のファイルには未知の長さのバイト列が書き込まれてしまいます。極端に再帰的なデータ構造を pickle 化しようとした場合には再帰の深さ制限を越えてしまうかもしれませんが、この場合には `RuntimeError` が送出されます。この制限は、`sys.setrecursionlimit()` で慎重に上げていくことは可能です。

(組み込みおよびユーザ定義の) 関数は、値ではなく ”完全記述された” 参照名として pickle 化されるので注意してください。これは、関数の定義されているモジュールの名前と一緒に併せ、関数名だけが pickle 化されることを意味します。関数のコードや関数の属性は何も pickle 化されません。従って、定義しているモジュールは unpickle 化環境で import 可能でなければならず、そのモジュールには指定されたオブジェクトが含まれていなければなりません。そうでない場合、例外が送出されます^{*4}。

クラスも同様に名前参照で pickle 化されるので、unpickle 化環境には同じ制限が課せられます。クラス中のコードやデータは何も pickle 化されないので、以下の例ではクラス属性 `attr` が unpickle 化環境で復元されないことに注意してください

```
class Foo:
    attr = 'a class attr'

picklestring = pickle.dumps(Foo)
```

pickle 化可能な関数やクラスがモジュールのトップレベルで定義されていないのはこれらの制限のためです。

同様に、クラスのインスタンスが pickle 化された際、そのクラスのコードおよびデータはオブジェクトと一緒に pickle 化されることはありません。インスタンスのデータのみが pickle 化されます。この仕様は、クラス内のバグを修正したりメソッドを追加した後でも、そのクラスの以前のバージョンで作られたオブジェクトを読み出せるように意図的に行われています。あるクラスの多くのバージョンで使われるような長命なオブジェクトを作ろうと計画しているなら、そのクラスの `__setstate__()` メソッドによって適切な変換が行われるようにオブジェクトのバージョン番号を入れておくともいかもしれません。

11.1.5 pickle 化プロトコル

この節では pickler/unpickler と直列化対象のオブジェクトとの間のインタフェースを定義する ”pickle 化プロトコル” について記述します。このプロトコルは自分のオブジェクトがどのように直列化されたり非直列化されたりするかを定義し、カスタマイズし、制御するための標準的な方法を提供します。この節での記述は、unpickle 化環境を不信な pickle 化データに対して安全にするために使う特殊なカスタマイズ化についてはカバーしていません; 詳細は [Unpickler をサブクラス化する](#) を参照してください。

^{*4} 送出される例外は `ImportError` や `AttributeError` になるはずですが、他の例外も起こりえます。

通常のクラスインスタンスの **pickle** 化および **unpickle** 化`object.__getinitargs__()`

pickle 化されたクラスインスタンスが unpickle 化されたとき、`__init__()` メソッドは通常呼び出されません。unpickle 化の際に `__init__()` が呼び出される方が望ましい場合、旧スタイルクラスではメソッド `__getinitargs__()` を定義することができます。このメソッドはクラスコンストラクタ (例えば `__init__()`) に渡されるべき位置引数からなる タプル を返さなければなりません。`__getinitargs__()` メソッドは pickle 時に呼び出されます; この関数が返すタプルはインスタンスの pickle 化データに組み込まれます。

`object.__getnewargs__()`

新スタイルクラスでは、プロトコル 2 で呼び出される `__getnewargs__()` を定義する事ができます。インスタンス生成時に内部的な不変条件が成立する必要があるあたり、(タプルや文字列のように) 型の `__new__()` メソッドに指定する引数によってメモリの割り当てを変更する必要がある場合には `__getnewargs__()` を定義してください。新スタイルクラス C のインスタンスは、次のように生成されます。:

```
obj = C.__new__(C, *args)
```

ここで `args` は元のオブジェクトの `__getnewargs__()` メソッドを呼び出した時の戻り値となります。`__getnewargs__()` を定義していない場合、`args` は空のタプルとなります。

`object.__getstate__()`

クラスは、インスタンスの pickle 化方法にさらに影響を与えることができます; クラスが `__getstate__()` メソッドを定義している場合、このメソッドが呼び出され、返された状態値はインスタンスの内容として、インスタンスの辞書の代わりに pickle 化されます。`__getstate__()` メソッドが定義されていない場合、インスタンスの `__dict__` の内容が pickle 化されます。

`object.__setstate__(state)`

unpickle 化では、クラスが `__setstate__()` も定義していた場合、unpickle 化された状態値とともに呼び出されます。^{*5} `__setstate__()` メソッドが定義されていない場合、pickle 化された状態は辞書型でなければならず、その要素は新たなインスタンスの辞書に代入されます。クラスが `__getstate__()` と `__setstate__()` の両方を定義している場合、状態値オブジェクトは辞書である必要はなく、これらのメソッドは期待通りの動作を行います。^{*6}

注釈: 新しいスタイルのクラスにおいて `__getstate__()` が偽値を返す場合、`__setstate__()` メソッドは呼ばれません。

注釈: unpickle するとき、`__getattr__()`, `__getattribute__()`, `__setattr__()` といったメソッドがインスタンスに対して呼ばれます。これらのメソッドが何か内部の不変条件に依存しているのであれば、その型は `__getinitargs__()` か `__getnewargs__()` のどちらかを実装してその不変条件を満たせるようにする

^{*5} これらのメソッドはクラスインスタンスのコピーを実装する際にも用いられます。

^{*6} このプロトコルはまた、`copy` で定義されている浅いコピーや深いコピー操作でも用いられます。

べきです。それ以外の場合、`__new__()` も `__init__()` も呼ばれません。

拡張型の `pickle` 化および `unpickle` 化

`object.__reduce__()`

`Pickler` が全く未知の型の — 拡張型のような — オブジェクトに遭遇した場合、`pickle` 化方法のヒントとして 2 箇所を探します。第一は `__reduce__()` メソッドを実装しているかどうかです。もし実装されていれば、`pickle` 化時に `__reduce__()` メソッドが引数なしで呼び出されます。メソッドはこの呼び出しに対して文字列またはタプルのどちらかを返さねばなりません。

文字列を返す場合、その文字列は通常通りに `pickle` 化されるグローバル変数の名前を指しています。`__reduce__()` の返す文字列は、モジュールにからみてオブジェクトのローカルな名前でなければなりません; `pickle` モジュールはモジュールの名前空間を検索して、オブジェクトの属するモジュールを決定します。

タプルを返す場合、タプルの要素数は 2 から 5 でなければなりません。オプションの要素は省略したり `None` を指定したりできます。各要素の意味づけは以下の通りです:

- オブジェクトの初期バージョンを生成するために呼び出される呼び出し可能オブジェクトです。この呼び出し可能オブジェクトへの引数はタプルの次の要素で与えられます。それ以降の要素では `pickle` 化されたデータを完全に再構築するために使われる付加的な状態情報が与えられます。

逆 `pickle` 化的环境下では、このオブジェクトはクラスか、“安全なコンストラクタ (safe constructor, 下記参照)” として登録されていたり属性 `__safe_for_unpickling__` の値が真であるような呼び出し可能オブジェクトでなければなりません。そうでない場合、逆 `pickle` 化を行う環境で `UnpicklingError` が送出されます。通常通り、callable は名前だけで `pickle` 化されるので注意してください。

- 呼び出し可能なオブジェクトのための引数からなるタプル

バージョン 2.5 で変更: 以前は、この引数には `None` もあり得ました。

- オプションとして、通常のクラスインスタンスの `pickle` 化および `unpickle` 化 節で記述されているようにオブジェクトの `__setstate__()` メソッドに渡される、オブジェクトの状態。オブジェクトが `__setstate__()` メソッドを持たない場合、上記のように、この値は辞書でなくてはならず、オブジェクトの `__dict__` に追加されます。
- オプションとして、リスト中の連続する要素を返すイテレータ (シーケンスではありません)。このリストの要素は `pickle` 化され、`obj.append(item)` または `obj.extend(list_of_items)` のいずれかを使って追加されます。主にリストのサブクラスで用いられていますが、他のクラスでも、適切なシグネチャの `append()` や `extend()` を備えている限り利用できます。(`append()` と `extend()` のいずれを使うかは、どのバージョンの `pickle` プロトコルを使っているか、そして追加する要素の数で決まります。従って両方のメソッドをサポートしていなければなりません。)
- オプションとして、辞書中の連続する要素を返すイテレータ (シーケンスではありません)。このリストの要素は `(key, value)` という形式でなければなりません。要素は `pickle` 化され、

`obj[key] = value` を使ってオブジェクトに格納されます。主に辞書のサブクラスで用いられていますが、他のクラスでも、`__setitem__()` を備えている限り利用できます。

`object.__reduce_ex__(protocol)`

`__reduce__()` を実装する場合、プロトコルのバージョンを知っておくと便利ことがあります。これは `__reduce__()` の代わりに `__reduce_ex__()` を使って実現できます。 `__reduce_ex__()` が定義されている場合、`__reduce__()` よりも優先して呼び出されます (以前のバージョンとの互換性のために `__reduce__()` を残しておいてもかまいません)。 `__reduce_ex__()` はプロトコルのバージョンを表す整数の引数の一つ伴って呼び出されます。

`object` クラスでは `__reduce__()` と `__reduce_ex__()` の両方を定義しています。とはいえ、サブクラスで `__reduce__()` をオーバーライドしており、`__reduce_ex__()` をオーバーライドしていない場合には、`__reduce_ex__()` の実装がそれを検出して `__reduce__()` を呼び出すようになっています。

pickle 化するオブジェクト上で `__reduce__()` メソッドを実装する代わりに、`copy_reg` モジュールを使って呼び出し可能オブジェクトを登録する方法もあります。このモジュールはプログラムに ”縮小化関数 (reduction function)” とユーザ定義型のためのコンストラクタを登録する方法を提供します。縮小化関数は、単一の引数として pickle 化するオブジェクトをとることを除き、上で述べた `__reduce__()` メソッドと同じ意味とインタフェースを持ちます。

登録されたコンストラクタは上で述べたような unpickle 化については ”安全なコンストラクタ” であると考えられます。

外部オブジェクトの pickle 化および unpickle 化

オブジェクトの永続化を便利にするために、`pickle` は pickle 化されたデータ列上にないオブジェクトに対して参照を行うという概念をサポートしています。これらのオブジェクトは ”永続化 id (persistent id)” で参照されており、この id は単に印字可能な ASCII 文字からなる任意の文字列です。これらの名前の解決方法は `pickle` モジュールでは定義されていません; オブジェクトはこの名前解決を pickler および unpickler 上のユーザ定義関数にゆだねます^{*7}。

外部永続化 id の解決を定義するには、pickler オブジェクトの `persistent_id` 属性と、unpickler オブジェクトの `persistent_load` 属性を設定する必要があります。

外部永続化 id を持つオブジェクトを pickle 化するには、pickler は自作の `persistent_id()` メソッドを持たなければなりません。このメソッドは一つの引数を取り、`None` とオブジェクトの永続化 id のどちらかを返さなければなりません。`None` が返された場合、pickler は単にオブジェクトを通常のように pickle 化だけです。永続化 id 文字列が返された場合、pickler はその文字列に対して、unpickler がこの文字列を永続化 id として認識できるように、マーカと共に pickle 化します。

外部オブジェクトを unpickle 化するには、unpickler は自作の `persistent_load()` 関数を持たなければなりません。この関数は永続化 id 文字列を引数にとり、参照されているオブジェクトを返します。

多分 より理解できるようになるようなちょっとした例を以下に示します:

^{*7} ユーザ定義関数に関連付けを行うための実際のメカニズムは、`pickle` および `cPickle` では少し異なります。`pickle` のユーザは、サブクラス化を行い、`persistend_id()` および `persistent_load()` メソッドを上書きすることで同じ効果を得ることができます。

```

import pickle
from cStringIO import StringIO

src = StringIO()
p = pickle.Pickler(src)

def persistent_id(obj):
    if hasattr(obj, 'x'):
        return 'the value %d' % obj.x
    else:
        return None

p.persistent_id = persistent_id

class Integer:
    def __init__(self, x):
        self.x = x
    def __str__(self):
        return 'My name is integer %d' % self.x

i = Integer(7)
print i
p.dump(i)

datastream = src.getvalue()
print repr(datastream)
dst = StringIO(datastream)

up = pickle.Unpickler(dst)

class FancyInteger(Integer):
    def __str__(self):
        return 'I am the integer %d' % self.x

def persistent_load(persid):
    if persid.startswith('the value '):
        value = int(persid.split()[2])
        return FancyInteger(value)
    else:
        raise pickle.UnpicklingError, 'Invalid persistent id'

up.persistent_load = persistent_load

j = up.load()
print j

```

`cPickle` モジュール内では、`unpickler` の `persistent_load` 属性は Python リスト型として設定することができます。この場合、`unpickler` が永続化 id に遭遇しても、永続化 id 文字列は単にリストに追加されるだけです。この仕様は、pickle データ中の全てのオブジェクトを実際にインスタンス化しなくても、pickle データ列中でオブジェクトに対する参照を”嗅ぎ回る”ことができるようにするために存在しています^{*8}。リストに

^{*8} Guide と Jim が居間に座り込んでピクルス (pickles) を嗅いでいる光景を想像してください。

`persistent_load` を設定するやり方は、よく `Unpickler` クラスの `no_load()` メソッドと共に使われます。

11.1.6 Unpickler をサブクラス化する

デフォルトでは、逆 pickle 化は pickle 化されたデータ中に見つかったクラスを import することになります。自前の unpickler をカスタマイズすることで、何が unpickle 化されて、どのメソッドが呼び出されるかを厳密に制御することはできます。しかし不運なことに、厳密になにを行うべきかは `pickle` と `cPickle` のどちらを使うかで異なります^{*9}。

`pickle` モジュールでは、`Unpickler` からサブクラスを派生し、`load_global()` メソッドを上書きする必要があります。`load_global()` は pickle データ列から最初の 2 行を読まなければならない、ここで最初の行はそのクラスを含むモジュールの名前、2 行目はそのインスタンスのクラス名になるはずで、次にこのメソッドは、例えばモジュールをインポートして属性を掘り起こすなどしてクラスを探し、発見されたものを unpickler のスタックに置きます。その後、このクラスは空のクラスの `__class__` 属性に代入する方法で、クラスの `__init__()` を使わずにインスタンスを魔法のように生成します。あなたの作業は (もしその作業を受け入れるなら)、unpickler のスタックの上に push された `load_global()` を、unpickle しても安全だと考えられる何らかのクラスの既知の安全なバージョンにすることです。あるいは全てのインスタンスに対して unpickling を許可したくないならエラーを送出してください。このからくりがハックのように思えるなら、あなたは間違っていない。このからくりを動かすには、ソースコードを参照してください。

`cPickle` では事情は多少すっきりしていますが、十分というわけではありません。何を unpickle 化するかを制御するには、unpickler の `find_global` 属性を関数が `None` に設定します。属性が `None` の場合、インスタンスを unpickle しようとする試みは全て `UnpicklingError` を送出します。属性が関数の場合、この関数はモジュール名またはクラス名を受け取り、対応するクラスオブジェクトを返さなくてはなりません。このクラスが行わなくてはならないのは、クラスの探索、必要な import のやり直しです。そしてそのクラスのインスタンスが unpickle 化されるのを防ぐためにエラーを送出することもできます。

以上の話から言えることは、アプリケーションが unpickle 化する文字列の発信元については非常に高い注意をはらわなくてはならないということです。

11.1.7 例

いちばん単純には、`dump()` と `load()` を使用してください。自己参照リストが正しく pickle 化およびリストアされることに注目してください。

```
import pickle

data1 = {'a': [1, 2.0, 3, 4+6j],
         'b': ('string', u'Unicode string'),
         'c': None}

selfref_list = [1, 2, 3]
```

(次のページに続く)

^{*9} 注意してください: ここで記述されている機構は内部の属性とメソッドを使っており、これらは Python の将来のバージョンで変更される対象になっています。われわれは将来、この挙動を制御するための、`pickle` および `cPickle` の両方で動作する、共通のインタフェースを提供するつもりです。

(前のページからの続き)

```
selfref_list.append(selfref_list)

output = open('data.pkl', 'wb')

# Pickle dictionary using protocol 0.
pickle.dump(data1, output)

# Pickle the list using the highest protocol available.
pickle.dump(selfref_list, output, -1)

output.close()
```

以下の例は pickle 化された結果のデータを読み込みます。pickle を含むデータを読み込む場合、ファイルはバイナリモードでオープンしなければいけません。これは ASCII 形式とバイナリ形式のどちらが使われているかは分からないからです。

```
import pprint, pickle

pkl_file = open('data.pkl', 'rb')

data1 = pickle.load(pkl_file)
pprint.pprint(data1)

data2 = pickle.load(pkl_file)
pprint.pprint(data2)

pkl_file.close()
```

より大きな例で、クラスを pickle 化する挙動を変更するやり方を示します。TextReader クラスはテキストファイルを開き、readline() メソッドが呼ばれるたびに行番号と行の内容を返します。TextReader インスタンスが pickle 化された場合、ファイルオブジェクト以外の全ての属性が保存されます。インスタンスが unpickle 化された際、ファイルは再度開かれ、以前のファイル位置から読み出しを再開します。上記の動作を実装するために、__setstate__() および __getstate__() メソッドが使われています。

```
#!/usr/local/bin/python

class TextReader:
    """Print and number lines in a text file."""
    def __init__(self, file):
        self.file = file
        self.fh = open(file)
        self.lineno = 0

    def readline(self):
        self.lineno = self.lineno + 1
        line = self.fh.readline()
        if not line:
            return None
        if line.endswith("\n"):

```

(次のページに続く)

(前のページからの続き)

```

        line = line[:-1]
        return "%d: %s" % (self.lineno, line)

    def __getstate__(self):
        odict = self.__dict__.copy() # copy the dict since we change it
        del odict['fh']               # remove filehandle entry
        return odict

    def __setstate__(self, dict):
        fh = open(dict['file'])       # reopen file
        count = dict['lineno']        # read from file...
        while count:                 # until line count is restored
            fh.readline()
            count = count - 1
        self.__dict__.update(dict)   # update attributes
        self.fh = fh                 # save the file object

```

使用例は以下になるでしょう:

```

>>> import TextReader
>>> obj = TextReader.TextReader("TextReader.py")
>>> obj.readline()
'1: #!/usr/local/bin/python'
>>> obj.readline()
'2: '
>>> obj.readline()
'3: class TextReader:'
>>> import pickle
>>> pickle.dump(obj, open('save.p', 'wb'))

```

`pickle` が Python プロセス間でうまく働くことを見たいなら、先に進む前に他の Python セッションを開始してください。以下の振る舞いは同じプロセスでも新たなプロセスでも起こります。

```

>>> import pickle
>>> reader = pickle.load(open('save.p', 'rb'))
>>> reader.readline()
'4:      """Print and number lines in a text file."""'

```

参考:

`copy-reg` モジュール 拡張型を登録するための Pickle インタフェース構成機構。

`shelve` モジュール オブジェクトのインデックス付きデータベース; `pickle` を使います。

`copy` モジュール オブジェクトの浅いコピーおよび深いコピー。

`marshal` モジュール 組み込み型の高性能な直列化。

11.2 cPickle — より高速な pickle

`cPickle` モジュールは Python オブジェクトの直列化および非直列化をサポートし、`pickle` モジュールとほとんど同じインタフェースと機能を提供します。いくつか相違点がありますが、最も重要な違いはパフォーマンスとサブクラス化が可能かどうかです。

第一に、`cPickle` は C で実装されているため、`pickle` よりも最大で 1000 倍高速です。第二に、`cPickle` モジュール内では、呼び出し可能オブジェクト `Pickler()` および `Unpickler()` は関数で、クラスではありません。つまり、pickle 化や unpickle 化を行うカスタムのサブクラスを派生することができないということです。多くのアプリケーションではこの機能は不要なので、`cPickle` モジュールによる大きなパフォーマンス向上の恩恵を受けられるはずです。

`pickle` と `cPickle` で作られた pickle データ列は同じなので、既存の pickle データに対して `pickle` と `cPickle` を互換に使用することができます。^{*10}

`cPickle` と `pickle` の API 間には他にも些細な相違がありますが、ほとんどのアプリケーションで互換性があります。より詳細なドキュメンテーションは `pickle` のドキュメントにあり、そこでドキュメント化されている相違点について挙げています。

脚注

11.3 copy_reg — pickle サポート関数を登録する

注釈: `copy_reg` モジュールは Python 3 で `copyreg` に変更されました。2to3 ツールが自動的にソースコードの `import` を変換します。

`copy_reg` モジュールは特定のオブジェクトを pickle 化するのに使われる関数を定義する手段を提供します。`pickle`、`cPickle`、`copy` モジュールが、そのオブジェクトを pickle 化したりコピーする際にそれら登録関数を使います。このモジュールは、クラスでないオブジェクトコンストラクタについての設定情報を提供します。クラスでないオブジェクトコンストラクタ、とは、おそらくファクトリ関数かクラスのインスタンスです。

`copy_reg.constructor(object)`

`object` を有効なコンストラクタであると宣言します。`object` が呼び出し可能でなければ (したがってコンストラクタとして有効でなければ)、`TypeError` を発生します。

`copy_reg.pickle(type, function[, constructor])`

`function` を、型 `type` のオブジェクトに対する "リダクション" 関数 として使うことを宣言します。`type` は旧スタイルクラスオブジェクトであってはいけません。(旧スタイルクラスは異なった扱われ方をします。詳細は、`pickle` モジュールのドキュメンテーションを参照してください。) `function` は文字列または二ないし三つの要素を含むタプルを返さなければなりません。

^{*10} `pickle` データ形式は実際には小規模なスタック指向のプログラム言語であり、またあるオブジェクトをエンコードする際に多少の自由度があるため、二つのモジュールが同じ入力オブジェクトに対して異なるデータ列を生成することもあります。しかし、常に互いに他のデータ列を読み出せることが保証されています。

オプションの *constructor* パラメータが与えられた場合、それは呼び出し可能オブジェクトで、*function* が返した引数のタプルとともに pickle 化時に呼ばれてオブジェクトを再構築するために使われます。*object* がクラスの場合、または *constructor* が呼び出し可能でない場合には *TypeError* が発生します。

function と *constructor* の求められるインターフェイスについての詳細は、*pickle* モジュールを参照してください。

11.3.1 例

下記の例は、pickle 関数を登録する方法と、それがどのように使用されるかを示そうとしています：

```
>>> import copy_reg, copy, pickle
>>> class C(object):
...     def __init__(self, a):
...         self.a = a
...
>>> def pickle_c(c):
...     print("pickling a C instance...")
...     return C, (c.a,)
...
>>> copy_reg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)
pickling a C instance...
>>> p = pickle.dumps(c)
pickling a C instance...
```

11.4 shelve — Python オブジェクトの永続化

ソースコード: [Lib/shelve.py](#)

”シェルフ (shelf, 棚)” は辞書に似た永続性を持つオブジェクトです。 ”dbm” データベースとの違いは、シェルフの値 (キーではありません！) は実質上どんな Python オブジェクトにも — *pickle* モジュールが扱えるなら何でも — できるということです。これにはほとんどのクラスインスタンス、再帰的なデータ型、沢山の共有されたサブオブジェクトを含むオブジェクトが含まれます。キーは通常の文字列です。

`shelve.open(filename, flag='c', protocol=None, writeback=False)`

永続的な辞書を開きます。指定された *filename* は、根底にあるデータベースの基本ファイル名となります。副作用として、*filename* には拡張子がつけられる場合があり、ひとつ以上のファイルが生成される可能性もあります。デフォルトでは、根底にあるデータベースファイルは読み書き可能なように開かれます。オプションの *flag* パラメータは *anydbm.open()* における *flag* パラメータと同様に解釈されます。

デフォルトでは、値を整列化する際にはバージョン 0 の pickle 化が用いられます。pickle 化プロトコルのバージョンは *protocol* パラメータで指定することができます。pickle 化プロトコルについては *pickle* のドキュメントを参照してください。

バージョン 2.3 で変更: *protocol* パラメータが追加されました。

Python の意味論により、シェルフには永続的な辞書の可変エントリがいつ変更されたかを知る術がありません。デフォルトでは、変更されたオブジェクトはシェルフに代入されたとき だけ 書き込まれます (例 参照)。オプションの *writeback* パラメータが `True` に設定されている場合は、アクセスされたすべてのエントリはメモリ上にキャッシュされ、*sync()* および *close()* を呼び出した際に書き戻されます; この機能は永続的な辞書上の可変の要素に対する変更を容易にしますが、多数のエントリがアクセスされた場合、膨大な量のメモリがキャッシュのために消費され、アクセスされた全てのエントリを書き戻す (アクセスされたエントリが可変であるか、あるいは実際に変更されたかを決定する方法は存在しないのです) ために、ファイルを閉じる操作が非常に低速になります。

ファイルオブジェクト同様に、*shelve* オブジェクトは永続データが確実にディスクにフラッシュされるように、明示的にクローズすべきです。

警告: *shelve* モジュールは裏で *pickle* を使っているので、信頼できないソースからシェルフを読み込むのは危険です。 *pickle* と同じく、*shelf* の読み込みでも任意のコードを実行できるからです。

シェルフオブジェクトは辞書がサポートするほとんどのメソッドをサポートしています。これにより、辞書ベースのスクリプトから永続的な記憶媒体を必要とするスクリプトに容易に移行できるようになります。

Python 3 への移行用メソッド (*viewkeys()*, *viewvalues()*, *viewitems()*) はサポートされていないことに注意してください。

追加でサポートされるメソッドが二つあります:

`Shelf.sync()`

シェルフが *writeback* を `True` にセットして開かれている場合に、キャッシュ中の全てのエントリを書き戻します。また可能な場合は、キャッシュを空にしてディスク上の永続的な辞書を同期します。このメソッドはシェルフを *close()* によって閉じるとき自動的に呼び出されます。

`Shelf.close()`

永続的な 辞書 オブジェクトを同期して閉じます。既に閉じられているシェルフに対して呼び出すと *ValueError* を出し失敗します。

参考:

通常の辞書に近い速度をもち、いろいろなストレージフォーマットに対応した、[永続化辞書のレシピ](#)。

11.4.1 制限事項

- どのデータベースパッケージが使われるか (例えば *dbm*、*gdbm*、*bsddb*) は、どのインタフェースが利用可能かに依存します。従って、データベースを *dbm* を使って直接開く方法は安全ではありません。データベースはまた、*dbm* が使われた場合 (不幸なことに) その制約に縛られます — これはデー

データベースに記録されたオブジェクト (の pickle 化された表現) はかなり小さくなくてはならず、キー衝突が生じた場合に、稀にデータベースを更新することができなくなることを意味します。

- `shelve` モジュールは、シェルフに置かれたオブジェクトの 並列した 読み出し/書き込みアクセスをサポートしません (複数の同時読み出しアクセスは安全です)。あるプログラムが書き込みのために開かれたシェルフを持っているとき、他のプログラムはそのシェルフを読み書きのために開いてはいけません。この問題を解決するために Unix のファイルロック機構を使うことができますが、この機構は Unix のバージョン間で異なり、使われているデータベースの実装について知識が必要となります。

class `shelve.Shelf` (*dict*, *protocol=None*, *writeback=False*)

`UserDict.DictMixin` のサブクラスで、pickle 化された値を *dict* オブジェクトに保存します。

デフォルトでは、値を整列化するにはバージョン 0 の pickle 化が用いられます。pickle 化プロトコルのバージョンは *protocol* パラメータで指定することができます。pickle 化プロトコルについては [pickle](#) のドキュメントを参照してください。

バージョン 2.3 で変更: *protocol* パラメータが追加されました。

writeback パラメータが `True` に設定されていれば、アクセスされたすべてのエントリはメモリ上にキャッシュされ、ファイルを閉じる際に *dict* に書き戻されます; この機能により、可変のエントリに対して自然な操作が可能になりますが、さらに多くのメモリを消費し、辞書をファイルと同期して閉じる際に長い時間がかかるようになります。

class `shelve.BsdDbShelf` (*dict*, *protocol=None*, *writeback=False*)

`Shelf` のサブクラスで、`first()`, `next()`, `previous()`, `last()`, `set_location()` メソッドを公開しています。これらのメソッドは `bsddb` モジュールでは利用可能ですが、他のデータベースモジュールでは利用できません。コンストラクタに渡された *dict* オブジェクトは上記のメソッドをサポートしていなくてはなりません。通常は、`bsddb.hashopen()`, `bsddb.btopen()` または `bsddb.rnopen()` のいずれかを呼び出して得られるオブジェクトが条件を満たしています。オプションの *protocol* および *writeback* パラメータは `Shelf` クラスにおけるパラメータと同様に解釈されます。

class `shelve.DbfilenameShelf` (*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

`Shelf` のサブクラスで、辞書に似たオブジェクトの代わりに *filename* を受理します。根底にあるファイルは `anydbm.open()` を使って開かれます。デフォルトでは、ファイルは読み書き可能な状態で開かれます。オプションの *flag* パラメータは `open()` 関数におけるパラメータと同様に解釈されます。オプションの *protocol* および *writeback* パラメータは `Shelf` クラスにおけるパラメータと同様に解釈されます。

11.4.2 例

インタフェースは以下のコードに集約されています (key は文字列で、data は任意のオブジェクトです):

```
import shelve

d = shelve.open(filename) # open -- file may get suffix added by low-level
                           # library
```

(次のページに続く)

(前のページからの続き)

```

d[key] = data      # store data at key (overwrites old data if
                  # using an existing key)
data = d[key]      # retrieve a COPY of data at key (raise KeyError if no
                  # such key)
del d[key]         # delete data stored at key (raises KeyError
                  # if no such key)
flag = d.has_key(key)  # true if the key exists
klist = d.keys()    # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = range(4)   # this works as expected, but...
d['xx'].append(5)    # *this doesn't!* -- d['xx'] is STILL range(4)!

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']       # extracts the copy
temp.append(5)       # mutates the copy
d['xx'] = temp       # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()           # close it

```

参考:*anydbm* モジュール dbm スタイルの汎用的なインタフェース*bsddb* モジュール BSD db データベースインタフェース。*dbhash* モジュール *bsddb* をラップする薄いレイヤで、他のデータベースモジュールのように関数 *open()* を提供しています。*dbm* モジュール 標準の Unix データベースインタフェース。*dumbdbm* モジュール dbm インタフェースの移植性のある実装。*gdbm* モジュール dbm インタフェースに基づいた GNU データベースインタフェース。*pickle* モジュール *shelve* によって使われるオブジェクト整列化機構。*cPickle* モジュール *pickle* の高速版。

11.5 marshal — 内部使用向けの Python オブジェクト整列化

このモジュールには Python 値をバイナリ形式で読み書きできるような関数が含まれています。このバイナリ形式は Python 特有のものです。マシンアーキテクチャ非依存のものです (つまり、Python の値を PC 上でファイルに書き込み、Sun に転送し、そこで読み戻すことができます)。バイナリ形式の詳細は意図的にドキュメント化されていません; この形式は (稀にしかないので) Python のバージョン間で変更される可能性

があるからです。^{*1}

このモジュールは汎用の”永続化 (persistence)” モジュールではありません。汎用的な永続化や、RPC 呼び出しを通じた Python オブジェクトの転送については、モジュール `pickle` および `shelve` を参照してください。 `marshal` モジュールは主に、”擬似コンパイルされた (pseudo-compiled)” コードの `.pyc` ファイルへの読み書きをサポートするために存在します。したがって、Python のメンテナは、必要が生じれば `marshal` 形式を後方互換性のないものに変更する権利を有しています。Python オブジェクトを直列化 (シリアライズ) および非直列化 (デシリアライズ) したい場合には、 `pickle` モジュールを使ってください。 `pickle` は速度は同等で、バージョン間の互換性が保証されていて、 `marshal` より広範囲のオブジェクトをサポートしています。

警告: `marshal` モジュールは、誤ったデータや悪意を持って作成されたデータに対する安全性を考慮していません。信頼できない、もしくは認証されていない出所からのデータを非整列化してはなりません。

すべての Python オブジェクト型がサポートされるわけではありません。一般に、その値が Python の特定の起動から独立しているオブジェクトだけが、このモジュールによって読み書きすることができます。次の型がサポートされます: 真偽値、整数、長整数、浮動小数点数、複素数、文字列、Unicode 文字列、タプル、リスト、`set`、`frozensets`、辞書、コードオブジェクト。ここで、タプル、リスト、`set`、`frozenset`、辞書は、その中に含まれる値がそれ自身サポートされる場合に限りサポートされることを理解すべきです; また、再帰的なリスト、`set`、辞書は書かれるべきではありません (無限ループを起こします)。シングルトン `None`、`Ellipsis`、`StopIteration` も読み書きできます。

警告: (DEC Alpha のように) C 言語の `long int` が 32 ビットよりも長いビット長を持つ場合、32 ビットよりも長い Python 整数を作成することが可能です。そのような整数が整列化された後、C 言語の `long int` のビット長が 32 ビットしかないマシン上で読み戻された場合、通常整数の代わりに Python 長整数が返されます。型は異なりますが、数値は同じです。(この動作は Python 2.2 で新たに追加されたものです。それ以前のバージョンでは、値のうち最小桁から 32 ビット以外の情報は失われ、警告メッセージが出力されていました。)

文字列を操作する関数と同様に、ファイルの読み書きを行う関数が提供されています。

このモジュールにはこれらの関数が定義されています:

`marshal.dump (value, file[, version])`

開かれたファイルに値を書き込みます。値はサポートされている型でなくてはなりません。ファイルは `sys.stdout` か、`open()` や `os.popen()` が返すようなファイルオブジェクトでなくてはなりません。Windows では `TemporaryFile` のようなラッパーは使えません。またファイルはバイナリモード ('wb' または 'w+b') で開かれていなければなりません。

値 (または値に含まれるオブジェクト) がサポートされていない型の場合、`ValueError` 例外が送出されます — しかし、同時にごみのデータがファイルに書き込まれます。このオブジェクトは `load()`

^{*1} このモジュールの名前は (特に) Modula-3 の設計者の間で使われていた用語の一つに由来しています。彼らはデータを自己充足的な形式で輸送する操作に ”整列化 (marshalling)” という用語を使いました。厳密に言えば、”整列させる (to marshal)” とは、あるデータを (例えば RPC パッファのように) 内部表現形式から外部表現形式に変換することを意味し、”非整列化 (unmarshalling)” とはその逆を意味します。

で適切に読み出されることはありません。

バージョン 2.4 で追加: `version` 引数は `dump` が使用するデータフォーマットを指定します (下記を参照してください)。

`marshal.load(file)`

開かれたファイルから値を一つ読んで返します。(異なるバージョンの Python の互換性のない marshal フォーマットだったなどの理由で) 有効な値が読み出せなかった場合、`EOFError`、`ValueError`、または `TypeError` を送出します。file はバイナリモード ('rb' または 'r+b') で開かれたファイルオブジェクトでなければなりません。

注釈: サポートされていない型を含むオブジェクトが `dump()` で整列化されている場合、`load()` は整列化不能な値を `None` で置き換えます。

`marshal.dumps(value[, version])`

`dump(value, file)` でファイルに書き込まれるような文字列を返します。値はサポートされている型でなければなりません。値がサポートされていない型のオブジェクト (またはサポートされていない型のオブジェクトを含むようなオブジェクト) の場合、`ValueError` 例外が送出されます。

バージョン 2.4 で追加: `version` 引数は `dumps` が使用するデータフォーマットを指定します (下記を参照してください)。

`marshal.loads(string)`

データ文字列を値に変換します。有効な値が見つからなかった場合、`EOFError`、`ValueError`、または `TypeError` が送出されます。文字列中の余分な文字は無視されます。

これに加えて、以下の定数が定義されています:

`marshal.version`

このモジュールが利用するバージョンを表します。バージョン 0 は歴史的なフォーマットです。バージョン 1 は文字列の再利用をします (Python 2.4 で追加されました)。バージョン 2 は浮動小数点数にバイナリフォーマットを使用します (Python 2.5 で追加されました)。現在のバージョンは 2 です。

バージョン 2.4 で追加。

脚注

11.6 anydbm — DBM 形式のデータベースへの汎用アクセスインタフェース

注釈: `anydbm` モジュールは Python 3 では `dbm` に名前が変更されました。2to3 ツールが自動的に import を変換します。

`anydbm` は種々の DBM データベース — `dbhash` (要 `bsddb`)、`gdbm`、および `dbm` — への汎用インタ

フェースです。これらのモジュールがどれもインストールされていない場合、`dumbdbm` モジュールの低速で単純な DBM 実装が使われます。

`anydbm.open(filename[, flag[, mode]])`

データベースファイル `filename` を開いて対応するオブジェクトを返します。

データベースファイルが既に存在する場合、その種類を決定するために `whichdb` モジュールが使用され、適切なモジュールが使用されます; データベースファイルが存在しない場合、上記のリストの中でインポート可能な最初のモジュールが使用されます。

オプションの `flag` は以下の値のいずれかです:

値	意味
'r'	既存のデータベースを読み込み専用で開く (デフォルト)
'w'	既存のデータベースを読み書き用に開く
'c'	データベースを読み書き用に開く。ただし存在しない場合には新たに作成する
'n'	常に新たに読み書き用の新規のデータベースを作成する

この引数が指定されない場合、標準の値は 'r' になります。

オプションの `mode` 引数は、新たにデータベースを作成しなければならない場合に使われる Unix のファイルモードです。標準の値は 8 進数の 0666 です (この値は現在有効な `umask` で修飾されます)。

exception `anydbm.error`

サポートされているモジュールそれぞれによって送出される可能性のある例外を含むタプル。これにはユニークな例外があり、最初の要素として同じく `anydbm.error` という名前の例外が含まれます — `anydbm.error` が送出される場合、後者 (訳注: タプルの `anydbm.error` ではなく例外 `anydbm.error`) が使用されます。

`open()` によって返されたオブジェクトは辞書とほとんど同じ機能をサポートします; キーとそれに対応付けられた値を記憶し、取り出し、削除することができ、`has_key()` メソッドと `keys()` メソッドが使えます。キーと値は常に文字列でなければなりません。

以下の例ではホスト名と対応するタイトルがいくつか登録し、データベースの内容を表示します:

```
import anydbm

# Open database, creating it if necessary.
db = anydbm.open('cache', 'c')

# Record some values
db['www.python.org'] = 'Python Website'
db['www.cnn.com'] = 'Cable News Network'

# Loop through contents. Other dictionary methods
# such as .keys(), .values() also work.
for k, v in db.iteritems():
    print k, '\t', v
```

(次のページに続く)

(前のページからの続き)

```
# Storing a non-string key or value will raise an exception (most
# likely a TypeError).
db['www.yahoo.com'] = 4

# Close when done.
db.close()
```

辞書型形式のメソッドに加えて、`anydbm` オブジェクトには以下のメソッドがあります:

`anydbm.close()`

`anydbm` データベースをクローズします。

参考:

`dbhash` モジュール BSD db データベースインタフェース。

`dbm` モジュール 標準の Unix データベースインタフェース。

`dumbdbm` モジュール `dbm` インタフェースの移植性のある実装。

`gdbm` モジュール `dbm` インタフェースに基づいた GNU データベースインタフェース。

`shelve` モジュール Python `dbm` インタフェース上に構築された汎用オブジェクト永続化機構。

`whichdb` モジュール 既存のデータベースがどの形式のデータベースか判定するユーティリティモジュール。

11.7 whichdb — どの DBM モジュールがデータベースを作ったかを推測する

注釈: `whichdb` モジュールが持っていた唯一の関数は、Python 3 では `dbm` モジュールに移動されました。`2to3` ツールは自動的に `import` を修正します。

The single function in this module attempts to guess which of the several simple database modules available `dbm`, `gdbm`, or `dbhash` should be used to open a given file.

`whichdb.whichdb(filename)`

次の値のうち 1 つを返します: ファイルが読み取れないか存在しないために開くことができない場合は `None`; ファイルのフォーマットを推測することができない場合は空文字列 (''); それ以外は `'dbm'` や `'gdbm'` のような、必要なモジュール名を含む文字列。

11.8 dbm — UNIX dbm のシンプルなインタフェース

注釈: `dbm` モジュールは、Python 3 では `dbm.ndbm` に変更されました。2to3 ツールは、自動的に `import` を修正します。

`dbm` モジュールは Unix の "(n)dbm" ライブラリのインタフェースを提供します。dbm オブジェクトは、キーと値が必ず文字列である以外はマッピング (辞書) のような振る舞いをします。print 文などで dbm インスタンスを出力してもキーと値は出力されません。また、`items()` と `values()` メソッドはサポートされません。

このモジュールは、BSD DB、GNU GDBM 互換インタフェースを持ったクラシックな `ndbm` インタフェースを使うことができます。Unix において、このモジュールは `configure` スクリプトで適切なヘッダファイルの特定が試みられてビルドされます。

このモジュールは以下を定義します:

exception `dbm.error`

I/O エラーのような dbm 特有のエラーが起ったときに上げられる値です。また、正しくないキーが与えられた場合に通常のマッピングエラーのような `KeyError` が発生します。

dbm.library

`ndbm` が使用している実装ライブラリ名です。

dbm.open (`filename`[, `flag`[, `mode`]])

dbm データベースを開いて dbm オブジェクトを返します。引数 `filename` はデータベースのファイル名を指定します。(拡張子 `.dir` や `.pag` は付けません。このインターフェイスの BSD DB 実装は拡張子 `.db` を追加して、単一のファイルのみ作ることに注意してください。)

オプションの `flag` は以下の値のいずれかです:

値	意味
'r'	既存のデータベースを読み込み専用で開く (デフォルト)
'w'	既存のデータベースを読み書き用に開く
'c'	データベースを読み書き用に開く。ただし存在しない場合には新たに作成する
'n'	常に新たに読み書き用の新規のデータベースを作成する

オプションの `mode` 引数は、新たにデータベースを作成しなければならない場合に使われる Unix のファイルモードです。標準の値は 8 進数の `0666` です (この値は現在有効な `umask` で修飾されます)。

辞書型形式のメソッドに加えて、dbm オブジェクトには以下のメソッドがあります:

dbm.close ()

dbm データベースをクローズします。

参考:

`anydbm` モジュール `dbm` スタイルの汎用的なインタフェース

`gdbm` モジュール GNU GDBM ライブラリに対する同様のインタフェース。

`whichdb` モジュール 既存のデータベースがどの形式のデータベースか判定するユーティリティモジュール。

11.9 gdbm — GNU による dbm の再実装

注釈: `gdbm` モジュールは、Python 3 では `dbm.gnu` にリネームされました。 `2to3` ツールが自動的にソースコードの `import` を修正します。

このモジュールは `dbm` モジュールによく似ていますが、`gdbm` を使っていくつかの追加機能を提供しています。`gdbm` と `dbm` では生成されるファイル形式に互換性がないので注意してください。

`gdbm` モジュールでは GNU DBM ライブラリへのインタフェースを提供します。`gdbm` オブジェクトはキーと値が常に文字列であることを除き、マップ型 (辞書型) と同じように動作します。`gdbm` オブジェクトに対して `print` を適用してもキーや値を印字することはなく、`items()` 及び `values()` メソッドはサポートされていません。

このモジュールでは以下の定数および関数を定義しています:

exception `gdbm.error`

I/O エラーのような `gdbm` 特有のエラーで送出されます。誤ったキーの指定のように、一般的なマップ型のエラーに対しては `KeyError` が送出されます。

`gdbm.open(filename[, flag[, mode]])`

`gdbm` データベースを開いて `gdbm` オブジェクトを返します。`filename` 引数はデータベースファイルの名前です。

オプションの `flag` は:

値	意味
'r'	既存のデータベースを読み込み専用で開く (デフォルト)
'w'	既存のデータベースを読み書き用に開く
'c'	データベースを読み書き用に開く。ただし存在しない場合には新たに作成する
'n'	常に新たに読み書き用の新規のデータベースを作成する

以下の追加の文字を `flag` に追加して、データベースの開きかたを制御することができます:

値	意味
'f'	データベースを高速モードで開きます。書き込みが同期されません。
's'	同期モード。データベースへの変更がすぐにファイルに書き込まれます。
'u'	データベースをロックしません。

全てのバージョンの `gdbm` で全てのフラグが有効とは限りません。モジュール定数 `open_flags` はサポートされているフラグ文字からなる文字列です。無効なフラグが指定された場合、例外 `error` が送出されます。

オプションの `mode` 引数は、新たにデータベースを作成しなければならない場合に使われる Unix のファイルモードです。標準の値は 8 進数の `0666` です。

辞書型形式のメソッドに加えて、`gdbm` オブジェクトには以下のメソッドがあります:

`gdbm.firstkey()`

このメソッドと `nextkey()` メソッドを使って、データベースの全てのキーにわたってループ処理を行うことができます。探索は `gdbm` の内部ハッシュ値の順番に行われ、キーの値に順に並んでいるとは限りません。このメソッドは最初のキーを返します。

`gdbm.nextkey(key)`

データベースの順方向探索において、`key` よりも後に来るキーを返します。以下のコードはデータベース `db` について、キー全てを含むリストをメモリ上に生成することなく全てのキーを出力します:

```
k = db.firstkey()
while k != None:
    print k
    k = db.nextkey(k)
```

`gdbm.reorganize()`

大量の削除を実行した後、`gdbm` ファイルの占めるスペースを削減したい場合、このルーチンはデータベースを再組織化します。この再組織化を使う以外に `gdbm` はデータベースファイルの大きさを短くすることはありません; そうでない場合、削除された部分のファイルスペースは保持され、新たな (キー、値の) ペアが追加される際に再利用されます。

`gdbm.sync()`

データベースが高速モードで開かれていた場合、このメソッドはディスクにまだ書き込まれていないデータを全て書き込ませます。

`gdbm.close()`

`gdbm` データベースをクローズします。

参考:

[`anydbm` モジュール](#) `dbm` スタイルの汎用的なインタフェース

[`whichdb` モジュール](#) 既存のデータベースがどの形式のデータベースか判定するユーティリティモジュール。

11.10 `dbhash` — BSD データベースライブラリへの `DBM` 形式のインタフェース

バージョン 2.6 で非推奨: `dbhash` モジュールは Python 3 で削除されました。

`dbhash` モジュールでは BSD db ライブラリを使ってデータベースを開くための関数を提供します。このモジュールは、DBM 形式のデータベースへのアクセスを提供する他の Python データベースモジュールのインタフェースをそのまま反映しています。`dbhash` を使うには `bsddb` モジュールが必要です。

このモジュールは 1 つの例外と 1 つの関数を提供しています:

exception `dbhash.error`

`KeyError` 以外のデータベースのエラーで送出されます。`bsddb.error` の別名です。

`dbhash.open(path[, flag[, mode]])`

データベース `db` を開き、データベースオブジェクトを返します。引数 `path` はデータベースファイルの名前です。

`flag` 引数は、次のいずれかの値になります:

値	意味
'r'	既存のデータベースを読み込み専用で開く (デフォルト)
'w'	既存のデータベースを読み書き用に開く
'c'	データベースを読み書き用に開く。ただし存在しない場合には新たに作成する
'n'	常に新たに読み書き用の新規のデータベースを作成する

BSD db ライブラリがロックをサポートしているプラットフォームでは、`flag` に 'l' を追加して、ロックを利用することを指定できます。

オプションの `mode` 引数は、新たにデータベースを作成しなければならないときにデータベースファイルに設定すべき Unix ファイル権限ビットを表すために使われます; この値はプロセスの現在の `umask` 値でマスクされます。

参考:

`anydbm` モジュール dbm スタイルの汎用的なインタフェース

`bsddb` モジュール BSD db ライブラリへの低レベルインタフェース。

`whichdb` モジュール 既存のデータベースがどの形式のデータベースか判定するユーティリティモジュール。

11.10.1 データベースオブジェクト

`open()` によって返されるデータベースオブジェクトは、全ての DBM 形式データベースやマップ型オブジェクトで共通のメソッドを提供します。それら標準のメソッドに加えて、以下のメソッドが利用可能です。

`dbhash.first()`

このメソッドと `next()` メソッドを使って、データベースの全てのキー/値のペアにわたってループ処理を行えます。探索はデータベースの内部ハッシュ値の順番に行われ、キーの値に順に並んでいるとは限りません。このメソッドは最初のキーを返します。

`dbhash.last()`

データベース探索における最後のキー/値を返します。逆順探索を開始する際に使うことができます;

`previous()` を参照してください。

`dbhash.next()`

データベースの順方向探索において、次のキー/値のペアを返します。以下のコードはデータベース `db` について、キー全てを含むリストをメモリ上に生成することなく全てのキーを出力します。

```
print db.first()
for i in xrange(1, len(db)):
    print db.next()
```

`dbhash.previous()`

データベースの順方向探索において、一つ前のキー/値のペアを返します。`last()` と併せて、逆方向の探索に用いられます。

`dbhash.sync()`

このメソッドはディスクにまだ書き込まれていないデータを全て書き込ませます。

11.11 bsddb — Berkeley DB ライブラリへのインタフェース

バージョン 2.6 で非推奨: `bsddb` モジュールは、Python 3 で削除されました。

`bsddb` モジュールは Berkeley DB ライブラリへのインタフェースを提供します。ユーザは適当な `open()` 呼び出しを使うことで、ハッシュ、B-Tree、またはレコードに基づくデータベースファイルを生成することができます。`bsddb` オブジェクトは辞書と大体同じように振る舞います。しかし、キー及び値は文字列でなければならないので、他のオブジェクトをキーとして使ったり、他の種のオブジェクトを記録したい場合、それらのデータを何らかの方法で直列化しなければなりません。これには通常 `marshal.dumps()` や `pickle.dumps()` が使われます。

`bsddb` モジュールは、バージョン 4.0 から 4.7 までの間の Berkeley DB ライブラリを必要とします。

参考:

<http://www.jcea.es/programacion/pybsddb.htm> Python Berkeley DB インターフェース `bsddb.db` のドキュメント。インターフェースは、Berkeley DB 4.x が提供しているオブジェクト指向インターフェースとほぼ同じインターフェースとなっています。

<http://www.oracle.com/database/berkeley-db/> The Berkeley DB library.

より新しい DB である DBEnv や DBSequence オブジェクトのインターフェースも `bsddb.db` モジュールで使用できます。これは、上の URL で説明されている Berkeley DB C API によりマッチしています。`bsddb.db` API が提供する追加機能には、チューニングやトランザクション、ログ出力、マルチプロセス環境でのデータベースへの同時アクセスなどがあります。

以下では、従来の `bsddb` モジュールと互換性のある、古いインターフェースを解説しています。Python 2.5 以降、このインターフェースはマルチスレッドに対応しています。マルチスレッドを使用する場合は `bsddb.db` API を推奨します。こちらのほうがスレッドをよりうまく制御できるからです。

`bsddb` モジュールでは、適切な形式の Berkeley DB ファイルにアクセスするオブジェクトを生成する以下の関数を定義しています。各関数の最初の二つの引数は同じです。可搬性のために、ほとんどのインスタンスで

は最初の二つの引数だけが使われているはずです。

```
bsddb.hashopen(filename[, flag[, mode[, pgsz[, ffactor[, nelem[, cachesize[, lorder[, hflags ]]]]]]])
```

filename と名づけられたハッシュ形式のファイルを開きます。 *filename* に `None` を指定することで、ディスクに保存するつもりがないファイルを作成することもできます。オプションの *flag* には、ファイルを開くためのモードを指定します。このモードは 'r' (読み出し専用)、'w' (読み書き可能)、'c' (読み書き可能 - 必要ならファイルを作成...これがデフォルトです) または 'n' (読み書き可能 - ファイル長を 0 に切り詰め)、にすることができます。他の引数はほとんど使われることはなく、下位レベルの `dbopen()` 関数に渡されるだけです。他の引数の使い方およびその解釈については Berkeley DB のドキュメントを読んで下さい。

```
bsddb.btopen(filename[, flag[, mode[, btflags[, cachesize[, maxkeypage[, minkeypage[, pgsz[, lorder ]]]]]]]]])
```

filename と名づけられた B-Tree 形式のファイルを開きます。 *filename* に `None` を指定することで、ディスクに保存するつもりがないファイルを作成することもできます。オプションの *flag* には、ファイルを開くためのモードを指定します。このモードは 'r' (読み出し専用)、'w' (読み書き可能)、'c' (読み書き可能 - 必要ならファイルを作成...これがデフォルトです)、または 'n' (読み書き可能 - ファイル長を 0 に切り詰め)、にすることができます。他の引数はほとんど使われることはなく、下位レベルの `dbopen()` 関数に渡されるだけです。他の引数の使い方およびその解釈については Berkeley DB のドキュメントを読んで下さい。

```
bsddb.rnopen(filename[, flag[, mode[, rnflags[, cachesize[, pgsz[, lorder[, rlen[, delim[, source[, pad ]]]]]]]]]]])
```

filename と名づけられた DB レコード形式のファイルを開きます、 *filename* に `None` を指定することで、ディスクに保存するつもりがないファイルを作成することもできます、オプションの *flag* には、ファイルを開くためのモードを指定します、このモードは 'r' (読み出し専用)、'w' (読み書き可能)、'c' (読み書き可能 - 必要ならファイルを作成...これがデフォルトです)、または 'n' (読み書き可能 - ファイル長を 0 に切り詰め)、にすることができます。他の引数はほとんど使われることはなく、下位レベルの `dbopen()` 関数に渡されるだけです、他の引数の使い方およびその解釈については Berkeley DB のドキュメントを読んで下さい。

注釈: 2.3 以降の Unix 版 Python には、`bsddb185` モジュールが存在する場合があります。このモジュールは古い Berkeley DB 1.85 データベースライブラリを持つシステムをサポートするためだけに存在しています。新規に開発するコードでは、`bsddb185` を直接使用しないで下さい。このモジュールは Python 3 で削除されます。必要であれば、PyPI にあるかもしれません。

参考:

[dbhash](#) モジュール [bsddb](#) への DBM 形式のインタフェース

11.11.1 ハッシュ、BTree、およびレコードオブジェクト

インスタンス化したハッシュ、B-Tree、およびレコードオブジェクトは辞書型と同じメソッドをサポートするようになります。加えて、以下に列挙したメソッドもサポートします。

バージョン 2.3.1 で変更: 辞書型メソッドを追加しました。

`bsddbobject.close()`

データベースの背後にあるファイルを閉じます。オブジェクトはアクセスできなくなります。これらのオブジェクトには `open()` メソッドがないため、再度ファイルを開くためには、新たな `bsddb` モジュールを開く関数を呼び出さなくてはなりません。

`bsddbobject.keys()`

DB ファイルに収められているキーからなるリストを返します。リスト内のキーの順番は決まっておらず、あてにはなりません。特に、異なるファイル形式の DB 間では返されるリストの順番が異なります。

`bsddbobject.has_key(key)`

引数 `key` が DB ファイルにキーとして含まれている場合 `1` を返します。

`bsddbobject.set_location(key)`

カーソルを `key` で示される要素に移動し、キー及び値からなるタプルを返します。`(bopen())` を使って開かれる) B-Tree データベースでは、`key` が実際にはデータベース内に存在しなかった場合、カーソルは並び順が `key` の次に来るような要素を指し、その場所のキー及び値が返されます。他のデータベースでは、データベース中に `key` が見つからなかった場合 `KeyError` が送出されます。

`bsddbobject.first()`

カーソルを DB ファイルの最初の要素に設定し、その要素を返します。B-Tree データベースの場合を除き、ファイル中のキーの順番は決まっています。データベースが空の場合、このメソッドは `bsddb.error` を発生させます。

`bsddbobject.next()`

カーソルを DB ファイルの次の要素に設定し、その要素を返します。B-Tree データベースの場合を除き、ファイル中のキーの順番は決まっています。

`bsddbobject.previous()`

カーソルを DB ファイルの直前の要素に設定し、その要素を返します。B-Tree データベースの場合を除き、ファイル中のキーの順番は決まっています。`(hashopen())` で開かれるような) ハッシュ表データベースではサポートされていません。

`bsddbobject.last()`

カーソルを DB ファイルの最後の要素に設定し、その要素を返します。ファイル中のキーの順番は決まっています。`(hashopen())` で開かれるような) ハッシュ表データベースではサポートされていません。データベースが空の場合、このメソッドは `bsddb.error` を発生させます。

`bsddbobject.sync()`

ディスク上のファイルをデータベースに同期させます。

以下はプログラム例です:

```
>>> import bsddb
>>> db = bsddb.btopen('spam.db', 'c')
>>> for i in range(10): db['%d'%i] = '%d'% (i*i)
...
>>> db['3']
```

(次のページに続く)

(前のページからの続き)

```

'9'
>>> db.keys()
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
>>> db.first()
('0', '0')
>>> db.next()
('1', '1')
>>> db.last()
('9', '81')
>>> db.set_location('2')
('2', '4')
>>> db.previous()
('1', '1')
>>> for k, v in db.iteritems():
...     print k, v
0 0
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
>>> '8' in db
True
>>> db.sync()
0

```

11.12 dumbdbm — 可搬性のある DBM 実装

注釈: `dumbdbm` モジュールは、Python 3 では `dbm.dumb` にリネームされました。 `2to3` ツールが自動的にソースコードの `import` を修正します。

注釈: `dumbdbm` モジュールは、`anydbm` が安定なモジュールを他に見つけることができなかった際の最後の手段とされています。 `dumbdbm` モジュールは速度を重視して書かれているわけではなく、他のデータベースモジュールのように重い使い方をするためのものではありません。

`dumbdbm` モジュールは永続的で辞書に類似したインタフェースを提供し、全て Python で書かれています。 `gdbm` や `bsddb` といったモジュールと異なり、外部ライブラリは必要ありません。他の永続性マップ型のように、キーおよび値は常に文字列でなければなりません。

以下はこのモジュールの定義:

exception `dumbdbm.error`

I/O エラーのような `dumbdbm` 特有のエラーの際に送出されます。不正なキーを指定したときのような、一般的な対応付けエラーの際には `KeyError` が送出されます。

`dumbdbm.open(filename[, flag[, mode]])`

`dumbdbm` データベースを開き、`dumbdbm` オブジェクトを返します。 *filename* 引数はデータベースファイル名のベース名 (特定の拡張子をもたないもの) です。 `dumbdbm` データベースが生成される際、 `.dat` および `.dir` の拡張子を持ったファイルが生成されます。

オプションの *flag* 引数は現状では無視されます; データベースは常に更新のために開かれ、存在しない場合には新たに作成されます。

オプションの *mode* 引数は、新たにデータベースを作成しなければならない場合に使われる Unix のファイルモードです。標準の値は 8 進数の `0666` です (この値は現在有効な `umask` で修飾されます)。

バージョン 2.2 で変更: *mode* 引数は以前のバージョンでは無視されます。

辞書型形式のメソッドに加えて、`dumbdbm` オブジェクトには以下のメソッドがあります:

`dumbdbm.close()`

`dumbdbm` データベースをクローズします。

参考:

anydbm モジュール `dbm` スタイルの汎用的なインタフェース

dbm モジュール `DBM/NDBM` ライブラリに対する同様のインタフェース。

gdbm モジュール `GNU GDBM` ライブラリの類似したインタフェース

shelve モジュール 非文字列データを記録する永続化モジュール。

whichdb モジュール 既存のデータベースがどの形式のデータベースか判定するユーティリティモジュール。

11.12.1 Dumbdbm オブジェクト

`UserDict.DictMixin` クラスで提供されているメソッドに加え、`dumbdbm` オブジェクトでは以下のメソッドを提供しています。

`dumbdbm.sync()`

ディスク上の辞書とデータファイルを同期します。このメソッドは `Shelve` オブジェクトの `sync()` メソッドから呼び出されます。

11.13 sqlite3 — SQLite データベースに対する DB-API 2.0 インタフェース

バージョン 2.5 で追加.

SQLite は、別にサーバプロセスは必要とせずデータベースのアクセスに SQL 問い合わせ言語の非標準的な一種を使える軽量のディスク上のデータベースを提供する C ライブラリです。ある種のアプリケーションは内部データ保存に SQLite を使えます。また、SQLite を使ってアプリケーションのプロトタイプを作りその後そのコードを PostgreSQL や Oracle のような大規模データベースに移植するということも可能です。

sqlite3 モジュールの著者は Gerhard Hring です。 [PEP 249](#) で記述されている DB-API 2.0 に準拠した SQL インターフェイスを提供します。

このモジュールを使うには、最初にデータベースを表す *Connection* オブジェクトを作ります。ここではデータはファイル `example.db` に格納されているものとします:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

特別な名前である `:memory:` を使うと RAM 上にデータベースを作ることができます。

Connection があれば、*Cursor* オブジェクトを作りその *execute()* メソッドを呼んで SQL コマンドを実行することができます:

```
c = conn.cursor()

# Create table
c.execute('''CREATE TABLE stocks
            (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)")

# Save (commit) the changes
conn.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
conn.close()
```

保存されたデータは永続的であり、次のセッションでもそのまま使用できます:

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
```

たいてい、SQL 操作では Python 変数の値を使う必要があります。この時、クエリーを Python の文字列操作を使って構築することは安全とは言えないので、すべきではありません。そのようなことをするとプログラムが SQL インジェクション攻撃に対し脆弱になります (<https://xkcd.com/327/> ではどうなってしまうかをユーモラスに描いています)。

代わりに、DB-API のパラメータ割り当てを使います。? を変数の値を使いたいところに埋めておきます。その上で、値のタプルをカーソルの *execute()* メソッドの第 2 引数として引き渡します。(他のデータベースモジュールでは変数の場所を示すのに `%s` や `:1` などの異なった表記を用いることがあります。) 例を示します:

```
# Never do this -- insecure!
symbol = 'RHAT'
c.execute("SELECT * FROM stocks WHERE symbol = '%s'" % symbol)

# Do this instead
t = ('RHAT',)
c.execute('SELECT * FROM stocks WHERE symbol=?', t)
print c.fetchone()

# Larger example that inserts many records at a time
purchases = [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
              ('2006-04-05', 'BUY', 'MSFT', 1000, 72.00),
              ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
              ]
c.executemany('INSERT INTO stocks VALUES (?, ?, ?, ?, ?)', purchases)
```

SELECT 文を実行した後データを取得する方法は3つありどれを使っても構いません。一つはカーソルをイテレータ (*iterator*) として扱う、一つはカーソルの `fetchone()` メソッドを呼んで一致した内の一行を取得する、もう一つは `fetchall()` メソッドを呼んで一致した全ての行のリストとして受け取る、という3つです。

以下の例ではイテレータの形を使います:

```
>>> for row in c.execute('SELECT * FROM stocks ORDER BY price'):
      print row

(u'2006-01-05', u'BUY', u'RHAT', 100, 35.14)
(u'2006-03-28', u'BUY', u'IBM', 1000, 45.0)
(u'2006-04-06', u'SELL', u'IBM', 500, 53.0)
(u'2006-04-05', u'BUY', u'MSFT', 1000, 72.0)
```

参考:

<https://github.com/ghaering/pysqlite> pysqlite のウェブページ – sqlite3 は「pysqlite」という名の下、外部で開発されています。

<https://www.sqlite.org> SQLite のウェブページ。この文書ではサポートされる SQL 方言の文法と使えるデータ型を説明しています。

<http://www.w3schools.com/sql/> SQL 学習に効くチュートリアル、リファレンス、実例集。

PEP 249 - Database API Specification 2.0 Marc-Andre Lemburg により書かれた PEP。

11.13.1 モジュールの関数と定数

`sqlite3.version`

文字列で表現されたモジュールのバージョン番号です。これは SQLite ライブラリのバージョンではありません。

`sqlite3.version_info`

整数のタプルで表現されたモジュールのバージョン番号です。これは SQLite ライブラリのバージョンではありません。

`sqlite3.sqlite_version`

文字列で表現された SQLite ランタイムライブラリのバージョン番号です。

`sqlite3.sqlite_version_info`

整数のタプルで表現された SQLite ランタイムライブラリのバージョン番号です。

`sqlite3.PARSE_DECLTYPES`

この定数は `connect()` 関数の `detect_types` パラメータとして使われます。

この定数を設定すると `sqlite3` モジュールは戻り値のカラムの宣言された型を読み取るようになります。意味を持つのは宣言の最初の単語です。すなわち、`"integer primary key"` においては `"integer"` が読み取られます。また、`"number(10)"` では、`"number"` が読み取られます。そして、そのカラムに対して、変換関数の辞書を探してその型に対して登録された関数を使うようにします。

`sqlite3.PARSE_COLNAMES`

この定数は `connect()` 関数の `detect_types` パラメータとして使われます。

この定数を設定すると SQLite のインタフェースは戻り値のそれぞれのカラムの名前を読み取るようになります。文字列の中の `[mytype]` といった形の部分を探し、`'mytype'` がそのカラムの名前であると判断します。そして `'mytype'` のエントリを変換関数辞書の中から見つけ、見つかった変換関数を値を返す際に用います。`Cursor.description` で見つかるカラム名はその最初の単語だけです。すなわち、もし `'as "x [datetime]"'` のようなものを SQL の中で使っていたとすると、読み取るのはカラム名の中の最初の空白までの全てですので、カラム名として使われるのは単純に `"x"` ということになります。

`sqlite3.connect(database[, timeout, detect_types, isolation_level, check_same_thread, factory, cached_statements])`

ファイル `database` の SQLite データベースへの接続を開きます。`":memory:"` という名前を使うことでディスクの代わりに RAM 上のデータベースへの接続を開くこともできます。

データベースが複数の接続からアクセスされている状況で、その内の一つがデータベースに変更を加えたとき、SQLite データベースはそのトランザクションがコミットされるまでロックされます。`timeout` パラメータで、例外を送出するまで接続がロックが解除されるのをどれだけ待つかを決めます。デフォルトは 5.0 (5 秒) です。

`isolation_level` パラメータについては、`Connection` オブジェクトの、`Connection.isolation_level` 属性を参照してください。

SQLite はネイティブで TEXT、INTEGER、REAL、BLOB および NULL のみをサポートしています。その他のタイプを使用したい場合はあなた自身で追加しなければなりません。`detect_types` パラメータおよび、`register_converter()` 関数でモジュールレベルで登録できるカスタム 変換関数 を使用することで簡単に追加できます。

パラメータ `detect_types` のデフォルトは 0 (つまりオフ、型検知無し) です。型検知を有効にするためには、`PARSE_DECLTYPES` と `PARSE_COLNAMES` の適当な組み合わせをこのパラメータにセットします。

デフォルトでは、`sqlite3` モジュールは `connect` の呼び出しの際にモジュールの `Connection` クラスを使います。しかし、`Connection` クラスを継承したクラスを `factory` パラメータに渡して `connect()` にそのクラスを使わせることもできます。

詳しくはこのマニュアルの [SQLite と Python の型](#) 節を参考にしてください。

`sqlite3` モジュールは SQL 解析のオーバーヘッドを避けるために内部で文キャッシュを使っています。接続に対してキャッシュされる文の数を自分で指定したいならば、`cached_statements` パラメータに設定してください。現在の実装ではデフォルトでキャッシュされる SQL 文の数を 100 にしています。

`sqlite3.register_converter(typename, callable)`

Registers a callable to convert a bytestring from the database into a custom Python type. The callable will be invoked for all database values that are of the type *typename*. Confer the parameter *detect_types* of the `connect()` function for how the type detection works. Note that *typename* and the name of the type in your query are matched in case-insensitive manner.

`sqlite3.register_adapter(type, callable)`

自分が使いたい Python の型 *type* を SQLite がサポートしている型に変換する呼び出し可能オブジェクト (callable) を登録します。その呼び出し可能オブジェクト *callable* はただ一つの引数に Python の値を受け取り、int, long, float, (UTF-8 でエンコードされた) str, unicode または buffer のいずれかの型の値を返さなければなりません。

`sqlite3.complete_statement(sql)`

もし文字列 *sql* がセミコロンで終端された一つ以上の完全な SQL 文を含んでいれば、`True` を返します。判定は SQL 文として文法的に正しいかではなく、閉じられていない文字列リテラルが無いことおよびセミコロンで終端されていることだけで行なわれます。

この関数は以下の例にあるような SQLite のシェルを作る際に使われます:

```
# A minimal SQLite shell for experiments

import sqlite3

con = sqlite3.connect(":memory:")
con.isolation_level = None
cur = con.cursor()

buffer = ""

print "Enter your SQL commands to execute in sqlite3."
print "Enter a blank line to exit."

while True:
    line = raw_input()
    if line == "":
        break
    buffer += line
    if sqlite3.complete_statement(buffer):
        try:
            buffer = buffer.strip()
```

(次のページに続く)

(前のページからの続き)

```

        cur.execute(buffer)

        if buffer.lstrip().upper().startswith("SELECT"):
            print cur.fetchall()
        except sqlite3.Error as e:
            print "An error occurred:", e.args[0]
        buffer = ""

con.close()

```

`sqlite3.enable_callback_tracebacks(flag)`

デフォルトでは、ユーザ定義の関数、集計関数、変換関数、認可コールバックなどはトレースバックを出力しません。デバッグの際にはこの関数を *flag* に `True` を指定して呼び出します。そうした後は先に述べたような関数のトレースバックが `sys.stderr` に出力されます。元に戻すには `False` を使います。

11.13.2 Connection オブジェクト

class `sqlite3.Connection`

SQLite データベースコネクション。以下の属性やメソッドを持ちます:

isolation_level

現在の分離レベルを取得または設定します。 `None` で自動コミットモードまたは "DEFERRED", "IMMEDIATE", "EXCLUSIVE" のどれかです。より詳しい説明は [トランザクション制御](#) 節を参照してください。

cursor (*factory=Cursor*)

`cursor` メソッドはオプション引数 *factory* を 1 つだけ受け付けます。渡された場合は、 `Cursor` またはそのサブクラスのインスタンスを返す呼び出し可能オブジェクトでなければなりません。

commit()

このメソッドは現在のトランザクションをコミットします。このメソッドを呼ばないと、前回 `commit()` を呼び出してから行ったすべての変更は、他のデータベースコネクションから見ることはできません。もし、データベースに書き込んだはずのデータが見えなくて悩んでいる場合は、このメソッドの呼び出しを忘れていないかチェックしてください。

rollback()

このメソッドは最後に行った `commit()` 後の全ての変更をロールバックします。

close()

このメソッドはデータベースコネクションを閉じます。このメソッドが自動的に `commit()` を呼び出さないことに注意してください。 `commit()` をせずにコネクションを閉じると、変更が消えてしまいます!

execute (*sql[, parameters]*)

このメソッドは非標準のショートカットで、`cursor` メソッドを呼び出して中間的なカーソルオブ

ジェクトを作り、そのカーソルの `execute` メソッドを与えられたパラメータと共に呼び出します。

executemany (*sql* [, *parameters*])

このメソッドは非標準のショートカットで、`cursor` メソッドを呼び出して中間的なカーソルオブジェクトを作り、そのカーソルの `executemany` メソッドを与えられたパラメータと共に呼び出します。

executescript (*sql_script*)

このメソッドは非標準のショートカットで、`cursor` メソッドを呼び出して中間的なカーソルオブジェクトを作り、そのカーソルの `executescript` メソッドを与えられたパラメータと共に呼び出します。

create_function (*name*, *num_params*, *func*)

後から SQL 文中で *name* という名前の関数として使えるユーザ定義関数を作成します。*num_params* は関数が受け付ける引数の数、*func* は SQL 関数として使われる Python の呼び出し可能オブジェクトです。

関数は SQLite でサポートされている任意の型を返すことができます。具体的には unicode, str, int, long, float, buffer, None です。

例:

```
import sqlite3
import md5

def md5sum(t):
    return md5.md5(t).hexdigest()

con = sqlite3.connect(":memory:")
con.create_function("md5", 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", ("foo",))
print cur.fetchone()[0]
```

create_aggregate (*name*, *num_params*, *aggregate_class*)

ユーザ定義の集計関数を作成します。

集計クラスにはパラメータ *num_params* で指定される個数の引数を取る `step` メソッドおよび最終的な集計結果を返す `finalize` メソッドを実装しなければなりません。

`finalize` メソッドは SQLite でサポートされている任意の型を返すことができます。具体的には unicode, str, int, long, float, buffer, None です。

例:

```
import sqlite3

class MySum:
    def __init__(self):
```

(次のページに続く)

(前のページからの続き)

```

        self.count = 0

    def step(self, value):
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
cur = con.cursor()
cur.execute("create table test(i)")
cur.execute("insert into test(i) values (1)")
cur.execute("insert into test(i) values (2)")
cur.execute("select mysum(i) from test")
print cur.fetchone()[0]

```

create_collation (*name*, *callable*)

name と *callable* で指定される照合順序を作成します。呼び出し可能オブジェクトには二つの文字列が渡されます。一つめのものが二つめのものより低く順序付けられるならば -1 を返し、等しければ 0 を返し、一つめのものが二つめのものより高く順序付けられるならば 1 を返すようにしなければなりません。この関数はソート (SQL での ORDER BY) をコントロールするもので、比較を行なうことは他の SQL 操作には影響を与えないことに注意しましょう。

また、呼び出し可能オブジェクトに渡される引数は Python のバイト文字列として渡されますが、それは通常 UTF-8 で符号化されたものになります。

以下の例は「間違った方法で」ソートする自作の照合順序です:

```

import sqlite3

def collate_reverse(string1, string2):
    return -cmp(string1, string2)

con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)

cur = con.cursor()
cur.execute("create table test(x)")
cur.executemany("insert into test(x) values (?)", [("a",), ("b",)])
cur.execute("select x from test order by x collate reverse")
for row in cur:
    print row
con.close()

```

照合順序を取り除くには *callable* に `None` を指定して `create_collation` を呼び出します:

```
con.create_collation("reverse", None)
```

interrupt ()

このメソッドを別スレッドから呼び出して接続上で現在実行中であるクエリを中断させられます。クエリが中断されると呼び出し元は例外を受け取ります。

set_authorizer (*authorizer_callback*)

このルーチンはコールバックを登録します。コールバックはデータベースのテーブルのカラムにアクセスしようとするたびに呼び出されます。コールバックはアクセスが許可されるならば `SQLITE_OK` を、SQL 文全体がエラーとともに中断されるべきならば `SQLITE_DENY` を、カラムが `NULL` 値として扱われるべきなら `SQLITE_IGNORE` を返さなければなりません。これらの定数は `sqlite3` モジュールに用意されています。

コールバックの第一引数はどの種類の操作が許可されるかを決めます。第二第三引数には第一引数に依存して本当に使われる引数が `None` かが渡されます。第四引数はもし適用されるならばデータベースの名前 ("main", "temp", etc.) です。第五引数はアクセスを試みる要因となった最も内側のトリガまたはビューの名前、またはアクセスの試みが入力された SQL コードに直接起因するものならば `None` です。

第一引数に与えることができる値や、その第一引数によって決まる第二第三引数の意味については、SQLite の文書を参考にしてください。必要な定数は全て `sqlite3` モジュールに用意されています。

set_progress_handler (*handler*, *n*)

このメソッドはコールバックを登録します。コールバックは SQLite 仮想マシン上の *n* 個の命令を実行するごとに呼び出されます。これは、GUI 更新などのために、長時間かかる処理中に SQLite からの呼び出しが欲しい場合に便利です。

以前登録した progress handler をクリアしたい場合は、このメソッドを、*handler* 引数に `None` を渡して呼び出してください。

バージョン 2.6 で追加.

enable_load_extension (*enabled*)

このメソッドは SQLite エンジンが共有ライブラリから SQLite 拡張を読み込むのを許可したり、禁止したりします。SQLite 拡張は新しい関数や集計関数や仮想テーブルの実装を定義できます。1 つの有名な拡張は SQLite によって頒布されている全テキスト検索拡張です。

SQLite 拡張はデフォルトで無効にされています。^{*1} をご覧ください。

バージョン 2.7 で追加.

```
import sqlite3

con = sqlite3.connect(":memory:")

# enable extension loading
con.enable_load_extension(True)
```

(次のページに続く)

^{*1} `sqlite3` モジュールはデフォルトで SQLite 拡張サポートなしで構築されます。いくつかのプラットフォーム (特に Mac OS X) でこの機能なしでコンパイルされる SQLite ライブラリがあるためです。SQLite 拡張サポートを有効にしてビルドするには、`setup.py` を編集して `SQLITE_OMIT_LOAD_EXTENSION` をセットしている行を削除してください。

(前のページからの続き)

```
# Load the fulltext search extension
con.execute("select load_extension('./fts3.so')")

# alternatively you can load the extension using an API call:
# con.load_extension("./fts3.so")

# disable extension loading again
con.enable_load_extension(False)

# example from SQLite wiki
con.execute("create virtual table recipe using fts3(name, ingredients)")
con.executescript("""
    insert into recipe (name, ingredients) values ('broccoli stew',
↪'broccoli peppers cheese tomatoes');
    insert into recipe (name, ingredients) values ('pumpkin stew',
↪'pumpkin onions garlic celery');
    insert into recipe (name, ingredients) values ('broccoli pie',
↪'broccoli cheese onions flour');
    insert into recipe (name, ingredients) values ('pumpkin pie', 'pumpkin_
↪sugar flour butter');
""")
for row in con.execute("select rowid, name, ingredients from recipe where_
↪name match 'pie'"):
    print row
```

load_extension(path)

このメソッドは共有ライブラリから SQLite 拡張を読み込みます。このメソッドを使う前に `enable_load_extension()` で拡張の読み込みを許可しておかなくてはなりません。

SQLite 拡張はデフォルトで無効にされています。^{*1} を見てください。

バージョン 2.7 で追加。

row_factory

この属性を、カーソルとタプルの形での元の行のデータを受け取り最終的な行を表すオブジェクトを返す呼び出し可能オブジェクトに、変更することができます。これによって、より進んだ結果の返し方を実装することができます。例えば、カラムの名前で各データにアクセスできるようなオブジェクトを返したりできます。

例:

```
import sqlite3

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d

con = sqlite3.connect(":memory:")
```

(次のページに続く)

(前のページからの続き)

```

con.row_factory = dict_factory
cur = con.cursor()
cur.execute("select 1 as a")
print cur.fetchone() ["a"]

```

タプルを返すのでは物足りず、名前に基づいたカラムへのアクセスが行ないたい場合は、高度に最適化された `sqlite3.Row` 型を `row_factory` にセットすることを考えてはいかがでしょうか。 `Row` クラスでは添字でも大文字小文字を無視した名前でもカラムにアクセスでき、しかもほとんどメモリーを浪費しません。おそらく、辞書を使うような独自実装のアプローチよりも、もしかすると db の行に基づいた解法よりも良いものかもしれません。

`text_factory`

この属性を使って TEXT データ型をどのオブジェクトで返すかを制御できます。デフォルトではこの属性は `unicode` に設定されており、`sqlite3` モジュールは TEXT を Unicode オブジェクトで返します。もしバイト列で返したいならば、`str` に設定してください。

効率化のため、非 ASCII データに対してのみ Unicode オブジェクトを返し、それ以外に対してバイト文字列を返す方法もあります。これを有効化するには、この属性を `sqlite3.OptimizedUnicode` にセットしてください。

バイト列を受け取って望みの型のオブジェクトを返すような呼び出し可能オブジェクトを何でも設定して構いません。

以下の説明用のコード例を参照してください:

```

import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()

AUSTRIA = u"\xd6sterreich"

# by default, rows are returned as Unicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert row[0] == AUSTRIA

# but we can make sqlite3 always return bytestrings ...
con.text_factory = str
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is str
# the bytestrings will be encoded in UTF-8, unless you stored garbage in
↳ the
# database ...
assert row[0] == AUSTRIA.encode("utf-8")

# we can also implement a custom text_factory ...
# here we implement one that will ignore Unicode characters that cannot be

```

(次のページに続く)

(前のページからの続き)

```
# decoded from UTF-8
con.text_factory = lambda x: unicode(x, "utf-8", "ignore")
cur.execute("select ?", ("this is latin1 and would normally create errors"
↳+
                                u"\xe4\xfc".encode("latin1"),))

row = cur.fetchone()
assert type(row[0]) is unicode

# sqlite3 offers a built-in optimized text_factory that will return
↳bytestring
# objects, if the data is in ASCII only, and otherwise return unicode
↳objects
con.text_factory = sqlite3.OptimizedUnicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is unicode

cur.execute("select ?", ("Germany",))
row = cur.fetchone()
assert type(row[0]) is str
```

total_changes

データベース接続が開始されて以来の行の変更・挿入・削除がなされた行の総数を返します。

iterdump

データベースを SQL test フォーマットでダンプするためのイテレータを返します。 in-memory データベースの内容を、後でリストアするための保存する場合に便利です。この関数は `sqlite3` シェルの中の `.dump` コマンドと同じ機能を持っています。

バージョン 2.6 で追加。

例:

```
# Convert file existing_db.db to SQL dump file dump.sql
import sqlite3, os

con = sqlite3.connect('existing_db.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
```

11.13.3 カーソルオブジェクト

class sqlite3.Cursor

`Cursor` インスタンスは以下の属性やメソッドを持ちます。

execute (*sql* [, *parameters*])

SQL 文を実行します。SQL 文はパラメータ化できます (すなわち SQL リテラルの代わりの場所確保文字 (placeholder) を入れておけます)。 `sqlite3` モジュールは 2 種類の場所確保記法をサポート

トします。一つは疑問符 (qmark スタイル)、もう一つは名前 (named スタイル) です。

両方のスタイルの例です:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table people (name_last, age)")

who = "Yeltsin"
age = 72

# This is the qmark style:
cur.execute("insert into people values (?, ?)", (who, age))

# And this is the named style:
cur.execute("select * from people where name_last=:who and age=:age", {"who": who, "age": age})

print cur.fetchone()
```

`execute()` は一つの SQL 文しか実行しません。二つ以上の文を実行しようとする、Warning を発生させます。複数の SQL 文を一つの呼び出しで実行したい場合は `executescript()` を使ってください。

executemany (*sql*, *seq_of_parameters*)

SQL 文 *sql* を *seq_of_parameters* シーケンス (またはマッピング) に含まれる全てのパラメータに対して実行します。 `sqlite3` モジュールでは、シーケンスの代わりにパラメータの組を作り出すイテレータ (*iterator*) を使うことが許されています。

```
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')

    def __iter__(self):
        return self

    def next(self):
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),) # this is a 1-tuple

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)
```

(次のページに続く)

(前のページからの続き)

```
cur.execute("select c from characters")
print cur.fetchall()
```

もう少し短いジェネレータ (*generator*) を使った例です:

```
import sqlite3
import string

def char_generator():
    for c in string.lowercase:
        yield (c,)

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

cur.executemany("insert into characters(c) values (?)", char_generator())

cur.execute("select c from characters")
print cur.fetchall()
```

executescript (*sql_script*)

これは非標準の便宜メソッドで、一度に複数の SQL 文を実行することができます。メソッドは最初に COMMIT 文を発行し、次いで引数として渡された SQL スクリプトを実行します。

sql_script はバイト文字列または Unicode 文字列です。

例:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript("""
    create table person(
        firstname,
        lastname,
        age
    );

    create table book(
        title,
        author,
        published
    );

    insert into book(title, author, published)
    values (
        'Dirk Gently''s Holistic Detective Agency',
        'Douglas Adams',

```

(次のページに続く)

(前のページからの続き)

```
1987
);
"""
```

fetchone()

クエリ結果から次の row をフェッチして、1 つのシーケンスを返します。これ以上データがない場合は *None* を返します。

fetchmany([size=cursor.arraysize])

クエリ結果から次の幾つかの row をフェッチして、リストを返します。これ以上データがない場合は空のリストを返します。

一回の呼び出しで返される row の数は、*size* 引数で指定できます。この引数が与えられない場合、*cursor* の *arraysize* 属性が利用されます。このメソッドは可能な限り指定された *size* の数の row を fetch しようとするべきです。もし、指定された数の row が利用可能でない場合、それより少ない数の row が返されます。

size 引数とパフォーマンスの関係についての注意です。パフォーマンスを最適化するためには、大抵、*arraysize* 属性を利用するのがベストです。*size* 引数を利用したのであれば、次の *fetchmany()* の呼び出しでも同じ数を利用するのがベストです。

fetchall()

全ての (残りの) クエリ結果の row をフェッチして、リストを返します。*cursor* の *arraysize* 属性がこの操作のパフォーマンスに影響することに気をつけてください。これ以上の row がない場合は、空のリストが返されます。

rowcount

一応 *sqlite3* モジュールの *Cursor* クラスはこの属性を実装していますが、データベースエンジン自身の「影響を受けた行」/「選択された行」の決定方法は少し風変わりです。

executemany() では、変更数が *rowcount* に合計されます。

Python DB API 仕様で要求されるように、*rowcount* 属性は「カーソルに対して *executeXX()* が行なわれていないか、最後の操作の *rowcount* がインターフェースによって決定できなかった場合は -1 」です。これには *SELECT* 文も含まれます。すべての列を取得するまでクエリによって生じた列の数を決定できないからです。

SQLite のバージョン 3.6.5 以前は、条件なしで *DELETE FROM table* を実行すると *rowcount* が 0 にセットされます。

lastrowid

この読み込み専用の属性は、最後に変更した row の *rowid* を提供します。この属性は、*execute()* メソッドを利用して *INSERT* 文を実行したときのみ設定されます。*INSERT* 以外の操作や、*executemany()* メソッドを呼び出した場合は、*lastrowid* は *None* に設定されます。

description

この読み込み専用の属性は、最後のクエリの結果のカラム名を提供します。Python DB API との

互換性を維持するために、各カラムに対して 7 つのタプルを返しますが、タプルの後ろ 6 つの要素は全て *None* です。

この属性は SELECT 文にマッチする row が 1 つもなかった場合でもセットされます。

connection

この読み込み専用の属性は、*Cursor* オブジェクトが使用する SQLite データベースの *Connection* を提供します。*con.cursor()* を呼び出すことにより作成される *Cursor* オブジェクトは、*con* を参照する *connection* 属性を持ちます:

```
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
```

11.13.4 Row オブジェクト

class sqlite3.Row

Row インスタンスは、*Connection* オブジェクトの *row_factory* として高度に最適化されています。タプルによく似た機能を持つ row を作成します。

カラム名とインデックスによる要素へのアクセス、イテレーション、*repr()*、同値テスト、*len()* をサポートしています。

もし、2 つの *Row* オブジェクトが完全に同じカラムと値を持っていた場合、それらは同値になります。

バージョン 2.6 で変更: イテレーションと同値性、ハッシュ可能

keys()

このメソッドはカラム名のリストを返します。クエリ直後から、これは *Cursor.description* の各タプルの最初のメンバになります。

バージョン 2.6 で追加.

Row の例のために、まずサンプルのテーブルを初期化します:

```
conn = sqlite3.connect(":memory:")
c = conn.cursor()
c.execute('''create table stocks
(date text, trans text, symbol text,
qty real, price real)''')
c.execute("""insert into stocks
            values ('2006-01-05', 'BUY', 'RHAT', 100, 35.14) """)
conn.commit()
c.close()
```

そして、*Row* を使ってみます:

```
>>> conn.row_factory = sqlite3.Row
>>> c = conn.cursor()
>>> c.execute('select * from stocks')
<sqlite3.Cursor object at 0x7f4e7dd8fa80>
>>> r = c.fetchone()
>>> type(r)
<type 'sqlite3.Row'>
>>> r
(u'2006-01-05', u'BUY', u'RHAT', 100.0, 35.14)
>>> len(r)
5
>>> r[2]
u'RHAT'
>>> r.keys()
['date', 'trans', 'symbol', 'qty', 'price']
>>> r['qty']
100.0
>>> for member in r:
...     print member
...
2006-01-05
BUY
RHAT
100.0
35.14
```

11.13.5 SQLite と Python の型

はじめに

SQLite は以下の型をネイティブにサポートします: NULL, INTEGER, REAL, TEXT, BLOB。

したがって、次の Python の型は問題なく SQLite に送り込めます:

Python の型	SQLite の型
<i>None</i>	NULL
<i>int</i>	INTEGER
<i>long</i>	INTEGER
<i>float</i>	REAL
<i>str</i> (UTF8 でエンコードされたバイト文字列)	TEXT
<i>unicode</i>	TEXT
<i>buffer</i>	BLOB

SQLite の型から Python の型へのデフォルトでの変換は以下の通りです:

SQLite の型	Python の型
NULL	<i>None</i>
INTEGER	<i>int</i> または <i>long</i> (サイズによる)
REAL	<i>float</i>
TEXT	<i>text_factory</i> に依存する。デフォルトでは <i>unicode</i> 。
BLOB	<i>buffer</i>

sqlite3 モジュールの型システムは二つの方法で拡張できます。一つはオブジェクト適合 (adaptation) を通じて追加された Python の型を SQLite に格納することです。もう一つは変換関数 (converter) を通じて *sqlite3* モジュールに SQLite の型を違った Python の型に変換させることです。

追加された **Python** の型を **SQLite** データベースに格納するために適合関数を使う

既に述べたように、SQLite が最初からサポートする型は限られたものだけです。それ以外の Python の型を SQLite で使うには、その型を *sqlite3* モジュールがサポートしている型の一つに 適合 させなくてはなりません。サポートしている型というのは、NoneType, int, long, float, str, unicode, buffer です。

sqlite3 モジュールで望みの Python の型をサポートされている型の一つに適合させる方法は二つあります。

オブジェクト自身で適合するようにする

自分でクラスを書いているならばこの方法が良いでしょう。次のようなクラスがあるとします:

```
class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y
```

さてこの点を SQLite の一つのカラムに収めたいと考えたとしましょう。最初にしなければならないのはサポートされている型の中から点を表現するのに使えるものを選ぶことです。ここでは単純に文字列を使うことにして、座標を区切るのにはセミコロンを使いましょう。次に必要なのはクラスに変換された値を返す `__conform__(self, protocol)` メソッドを追加することです。引数 *protocol* は `PrepareProtocol` になります。

```
import sqlite3

class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return "%f;%f" % (self.x, self.y)

con = sqlite3.connect(":memory:")
cur = con.cursor()
```

(次のページに続く)

(前のページからの続き)

```
p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print cur.fetchone()[0]
```

適合関数を登録する

もう一つの可能性は型を文字列表現に変換する関数を作り `register_adapter()` でその関数を登録することです。

注釈: 適合させる型/クラスは新スタイルクラス (*new-style class*) でなければなりません。すなわち、`object` を基底クラスの一つとしていなければなりません。

```
import sqlite3

class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)

sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print cur.fetchone()[0]
```

`sqlite3` モジュールには二つの Python 標準型 `datetime.date` と `datetime.datetime` に対するデフォルト適合関数があります。いま `datetime.datetime` オブジェクトを ISO 表現でなく Unix タイムスタンプとして格納したいとしましょう。

```
import sqlite3
import datetime, time

def adapt_datetime(ts):
    return time.mktime(ts.timetuple())

sqlite3.register_adapter(datetime.datetime, adapt_datetime)

con = sqlite3.connect(":memory:")
cur = con.cursor()

now = datetime.datetime.now()
```

(次のページに続く)

(前のページからの続き)

```
cur.execute("select ?", (now,))
print cur.fetchone()[0]
```

SQLite の値を好きな Python 型に変換する

適合関数を書くことで好きな Python 型を SQLite に送り込めるようになりました。しかし、本当に使い物になるようにするには Python から SQLite さらに Python へという往還 (roundtrip) の変換ができる必要があります。

そこで変換関数 (converter) です。

Point クラスの例に戻りましょう。x, y 座標をセミコロンで区切った文字列として SQLite に格納したのでした。

まず、文字列を引数として取り Point オブジェクトをそれから構築する変換関数を定義します。

注釈: 変換関数は SQLite に送り込んだデータ型に関係なく 常に 文字列を渡されます。

```
def convert_point(s):
    x, y = map(float, s.split(";"))
    return Point(x, y)
```

次に `sqlite3` モジュールにデータベースから取得したものが本当に点であることを教えなければなりません。二つの方法があります:

- 宣言された型を通じて暗黙的に
- カラム名を通じて明示的に

どちらの方法も **モジュールの関数と定数** 節の中で説明されています。それぞれ `PARSE_DECLTYPES` 定数と `PARSE_COLNAMES` 定数の項目です。

以下の例で両方のアプローチを紹介します。

```
import sqlite3

class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return("(%f;%f)" % (self.x, self.y))

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)

def convert_point(s):
```

(次のページに続く)

(前のページからの続き)

```

    x, y = map(float, s.split(";"))
    return Point(x, y)

# Register the adapter
sqlite3.register_adapter(Point, adapt_point)

# Register the converter
sqlite3.register_converter("point", convert_point)

p = Point(4.0, -3.2)

#####
# 1) Using declared types
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(p point)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute("select p from test")
print "with declared types:", cur.fetchone()[0]
cur.close()
con.close()

#####
# 1) Using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(p)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute('select p as "p [point]" from test')
print "with column names:", cur.fetchone()[0]
cur.close()
con.close()

```

デフォルトの適合関数と変換関数

`datetime` モジュールの `date` 型および `datetime` 型のためのデフォルト適合関数があります。これらの型は ISO 日付 / ISO タイムスタンプとして SQLite に送られます。

デフォルトの変換関数は `datetime.date` 用が "date" という名前で、`datetime.datetime` 用が "timestamp" という名前で登録されています。

これにより、多くの場合特別な細工無しに Python の日付 / タイムスタンプを使えます。適合関数の書式は実験的な SQLite の date/time 関数とも互換性があります。

以下の例でこのことを確かめます。

```

import sqlite3
import datetime

```

(次のページに続く)

(前のページからの続き)

```

con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES|sqlite3.
↳PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(d date, ts timestamp)")

today = datetime.date.today()
now = datetime.datetime.now()

cur.execute("insert into test(d, ts) values (?, ?)", (today, now))
cur.execute("select d, ts from test")
row = cur.fetchone()
print today, "=>", row[0], type(row[0])
print now, "=>", row[1], type(row[1])

cur.execute('select current_date as "d [date]", current_timestamp as "ts.'
↳[timestamp]"')
row = cur.fetchone()
print "current_date", row[0], type(row[0])
print "current_timestamp", row[1], type(row[1])

```

SQLite に格納されているタイムスタンプが 6 桁より長い小数部を持っている場合、タイムスタンプの変換関数によってマイクロ秒精度に丸められます。

11.13.6 トランザクション制御

デフォルトでは、`sqlite3` モジュールはデータ変更言語 (DML) 文 (すなわち INSERT/UPDATE/DELETE/REPLACE) の前に暗黙のうちにトランザクションを開始し、非 DML、非クエリ文 (すなわち SELECT や上記のいずれでもないもの。) の前にトランザクションをコミットします。

そのため、トランザクション内で CREATE TABLE ..., VACUUM, PRAGMA のようなコマンドを発行した場合、`sqlite3` モジュールはそのコマンドを実行する前に暗黙にコミットします。これには 2 つの理由があります。1 番目は、これらのコマンドのうちのいくつかがトランザクション内では動かないということです。2 番目の理由は、`sqlite3` がトランザクション状態 (トランザクションがアクティブかどうか) を追跡する必要があるためです。

`sqlite3` が暗黙のうちに実行する BEGIN 文の種類 (またはそういうものを使わないこと) を `connect()` 呼び出しの `isolation_level` パラメータを通じて、または接続の `isolation_level` プロパティを通じて、制御することができます。

自動コミットモードを使いたい場合は、`isolation_level` は None にしてください。

そうでなければデフォルトのまま BEGIN 文を使い続けるか、SQLite がサポートする分離レベル "DEFERRED", "IMMEDIATE" または "EXCLUSIVE" を設定してください。

11.13.7 sqlite3 の効率的な使い方

ショートカットメソッドを使う

Connection オブジェクトの非標準的なメソッド `execute()`, `executemany()`, `executescript()` を使うことで、(しばしば余計な) *Cursor* オブジェクトをわざわざ作り出さずに済むので、コードをより簡潔に書くことができます。 *Cursor* オブジェクトは暗黙裡に生成されショートカットメソッドの戻り値として受け取ることができます。この方法を使えば、`SELECT` 文を実行してその結果について反復することが、*Connection* オブジェクトに対する呼び出し一つで行なえます。

```
import sqlite3

persons = [
    ("Hugo", "Boss"),
    ("Calvin", "Klein")
]

con = sqlite3.connect(":memory:")

# Create the table
con.execute("create table person(firstname, lastname)")

# Fill the table
con.executemany("insert into person(firstname, lastname) values (?, ?)", persons)

# Print the table contents
for row in con.execute("select firstname, lastname from person"):
    print row

print "I just deleted", con.execute("delete from person").rowcount, "rows"
```

位置ではなく名前でカラムにアクセスする

sqlite3 モジュールの有用な機能の一つに、行生成関数として使われるための *sqlite3.Row* クラスがあります。

このクラスでラップされた行は、位置インデクス (タブルのような) でも大文字小文字を区別しない名前でもアクセスできます:

```
import sqlite3

con = sqlite3.connect(":memory:")
con.row_factory = sqlite3.Row

cur = con.cursor()
cur.execute("select 'John' as name, 42 as age")
for row in cur:
    assert row[0] == row["name"]
    assert row["name"] == row["nAmE"]
    assert row[1] == row["age"]
    assert row[1] == row["AgE"]
```

コネクションをコンテキストマネージャーとして利用する

バージョン 2.6 で追加.

Connection オブジェクトはコンテキストマネージャーとして利用して、トランザクションを自動的にコミットしたりロールバックすることができます。例外が発生したときにトランザクションはロールバックされ、それ以外の場合、トランザクションはコミットされます:

```
import sqlite3

con = sqlite3.connect(":memory:")
con.execute("create table person (id integer primary key, firstname varchar unique)
↪")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("insert into person(firstname) values (?)", ("Joe",))

# con.rollback() is called after the with block finishes with an exception, the
# exception is still raised and must be caught
try:
    with con:
        con.execute("insert into person(firstname) values (?)", ("Joe",))
except sqlite3.IntegrityError:
    print "couldn't add Joe twice"
```

11.13.8 既知の問題

マルチスレッド

古いバージョンの SQLite はスレッド間でのコネクションの共有に問題がありました。その理由は、Python のモジュールではスレッド間のコネクションとカーソルの共有ができないためです。依然としてそのようなことをしようとすると、実行時に例外を受け取るでしょう。

唯一の例外は `interrupt()` メソッドで、これだけが異なるスレッドから呼び出せます。

第 12 章

データ圧縮とアーカイブ

この章で説明されるモジュールは `zlib`, `gzip`, `bzip2` アルゴリズムによるデータの圧縮と、ZIP, tar フォーマットのアーカイブ作成をサポートします。 `shutil` モジュールで提供されている [アーカイブ化操作](#) も参照してください。

12.1 `zlib` — `gzip` 互換の圧縮

このモジュールは、データ圧縮を必要とするアプリケーションが `zlib` ライブラリを使って圧縮および展開を行えるようにします。 `zlib` ライブラリ自身の Web ページは <http://www.zlib.net> です。Python モジュールと `zlib` ライブラリの 1.1.3 より前のバージョンには互換性のない部分があることが知られています。1.1.3 にはセキュリティホールが存在するため、1.1.4 以降のバージョンを利用することを推奨します。

`zlib` の関数にはたくさんのオプションがあり、場合によっては特定の順番で使わなければなりません。このドキュメントではそれら順番についてすべてを説明しようとはしていません。詳細は公式サイト <http://www.zlib.net/manual.html> にある `zlib` のマニュアルを参照してください。

`.gz` ファイルの読み書きのためには、 `gzip` モジュールを参照してください。

このモジュールで利用可能な例外と関数を以下に示します：

exception `zlib.error`

圧縮および展開時のエラーによって送出される例外です。

`zlib.adler32` (`data` [, `value`])

`data` の Adler-32 チェックサムを計算します (Adler-32 チェックサムは、おおむね CRC32 と同等の信頼性を持ちながら、はるかに高速に計算できます)。 `value` が与えられていれば、チェックサム計算の初期値として使われます。与えられていなければ固定のデフォルト値が使われます。 `value` を与えることで、複数の入力を結合したデータ全体にわたり、通しのチェックサムを計算できます。このアルゴリズムは暗号論的には強力ではなく、認証やデジタル署名などに用いるべきではありません。また、チェックサムアルゴリズムとして設計されているため、汎用のハッシュアルゴリズムには向きません。

この関数は常に整数オブジェクトを返します。

注釈: 全ての Python のバージョンとプラットフォームで共通な数値を生成するには、 `adler32(data) &`

0xffffffff を利用してください。もしチェックサムをパックされたバイナリフォーマットのためにしか利用しないのであれば、符号が関係なくなり、32bit のバイナリ値としては戻り値は正しいので、この処理は必要ありません。

バージョン 2.6 で変更: 戻り値の範囲は、プラットフォームに関係なく $[-2^{31}, 2^{31}-1]$ になりました。古いバージョンでは、この値は幾つかのプラットフォームでは符号付き、別のプラットフォームでは符号なしになっていました。

バージョン 3.0 で変更: 戻り値の範囲は、プラットフォームに関係なく $[0, 2^{32}-1]$ の範囲の符号無しです。

`zlib.compress(string[, level])`

data で与えられたバイト列を圧縮し、圧縮されたデータを含む文字列オブジェクトを返します。 *level* は 0 から 9 の整数を取り、圧縮レベルを制御します; 1 は最も高速で最小限の圧縮を行い、9 は最も低速ですが最大限の圧縮を行います。0 は圧縮しません。デフォルト値は 6 です。何らかのエラーが発生した場合は `error` 例外を送出します。

`zlib.compressobj([level[, method[, wbits[, memlevel[, strategy]]]])`

Returns a compression object, to be used for compressing data streams that won't fit into memory at once. *level* is an integer from 0 to 9 or -1, controlling the level of compression; 1 is fastest and produces the least compression, 9 is slowest and produces the most. 0 is no compression. The default value is -1 (Z_DEFAULT_COMPRESSION). Z_DEFAULT_COMPRESSION represents a default compromise between speed and compression (currently equivalent to level 6).

method は圧縮アルゴリズムです。現在、DEFLATED のみサポートされています。

The *wbits* argument controls the size of the history buffer (or the "window size") used when compressing data, and whether a header and trailer is included in the output. It can take several ranges of values. The default is 15.

- +9 to +15: The base-two logarithm of the window size, which therefore ranges between 512 and 32768. Larger values produce better compression at the expense of greater memory usage. The resulting output will include a zlib-specific header and trailer.
- - 9 to - 15: Uses the absolute value of *wbits* as the window size logarithm, while producing a raw output stream with no header or trailing checksum.
- +25 to +31 = 16 + (9 to 15): Uses the low 4 bits of the value as the window size logarithm, while including a basic **gzip** header and trailing checksum in the output.

memlevel は内部圧縮状態用に使用されるメモリ量を制御します。有効な値は 1 から 9 です。大きい値ほど多くのメモリを消費しますが、より速く、より小さな出力を作成します。デフォルトは 8 です。

strategy は圧縮アルゴリズムの調整に使用されます。指定可能な値は、Z_DEFAULT_STRATEGY、Z_FILTERED、および Z_HUFFMAN_ONLY です。デフォルトは Z_DEFAULT_STRATEGY です。

`zlib.crc32(data[, value])`

data の CRC (Cyclic Redundancy Check, 巡回冗長検査) チェックサムを計算します。 *value* が与えられていれば、チェックサム計算の初期値として使われます。与えられていなければ固定のデフォルト値が使われます。 *value* を与えることで、複数の入力を結合したデータ全体にわたり、通しのチェックサム

を計算できます。このアルゴリズムは暗号論的には強力ではなく、認証やデジタル署名などに用いるべきではありません。また、チェックサムアルゴリズムとして設計されているため、汎用のハッシュアルゴリズムには向きません。

この関数は常に整数オブジェクトを返します。

注釈: 全ての Python のバージョン、全てのプラットフォームに渡って同じ数値を生成しようとするならば、`crc32(data) & 0xffffffff` を使って下さい。チェックサムをバイナリ形式そのままだけで扱うならばこのような細工は必要ありません。返値は符号に関係なく正しい 32 ビットのバイナリ表現だからです。

バージョン 2.6 で変更: 戻り値の範囲は、プラットフォームに関係なく $[-2^{31}, 2^{31}-1]$ になりました。古いバージョンでは、この値は幾つかのプラットフォームでは符号付き、別のプラットフォームでは符号なしになっていました。

バージョン 3.0 で変更: 戻り値の範囲は、プラットフォームに関係なく $[0, 2^{32}-1]$ の範囲の符号無しです。

`zlib.decompress (string[, wbits[, bufsize]])`

Decompresses the data in *string*, returning a string containing the uncompressed data. The *wbits* parameter depends on the format of *string*, and is discussed further below. If *bufsize* is given, it is used as the initial size of the output buffer. Raises the *error* exception if any error occurs.

The *wbits* parameter controls the size of the history buffer (or "window size"), and what header and trailer format is expected. It is similar to the parameter for *compressobj()*, but accepts more ranges of values:

- +8 to +15: The base-two logarithm of the window size. The input must include a zlib header and trailer.
- 0: Automatically determine the window size from the zlib header. Only supported since zlib 1.2.3.5.
- - 8 to - 15: Uses the absolute value of *wbits* as the window size logarithm. The input must be a raw stream with no header or trailer.
- +24 to +31 = 16 + (8 to 15): Uses the low 4 bits of the value as the window size logarithm. The input must include a gzip header and trailer.
- +40 to +47 = 32 + (8 to 15): Uses the low 4 bits of the value as the window size logarithm, and automatically accepts either the zlib or gzip format.

When decompressing a stream, the window size must not be smaller than the size originally used to compress the stream; using a too-small value may result in an *error* exception. The default *wbits* value is 15, which corresponds to the largest window size and requires a zlib header and trailer to be included.

bufsize は展開されたデータを保持するためのバッファサイズの初期値です。バッファの空きは必要に応じて必要なだけ増加するので、必ずしも正確な値を指定する必要はありません。この値のチューニングでできることは、`malloc()` が呼ばれる回数を数回減らすことぐらいです。デフォルトサイズは 16384 です。

`zlib.decompressobj ([wbits])`

一度にメモリ上に置くことができないようなデータストリームを展開するための展開オブジェクトを返

します。

The *whits* parameter controls the size of the history buffer (or the "window size"), and what header and trailer format is expected. It has the same meaning as *described for decompress()*.

圧縮オブジェクトは以下のメソッドをサポートしています:

`Compress.compress(string)`

string を圧縮し、圧縮されたデータを含む文字列を返します。この文字列は少なくとも *string* の一部分のデータに対する圧縮データを含みます。このデータは以前に呼んだ *compress()* が返した出力と結合することができます。入力の一部は以後の処理のために内部バッファに保存されることもあります。

`Compress.flush([mode])`

未処理の入力データが処理され、この未処理部分を圧縮したデータを含む文字列が返されます。 *mode* は定数 `Z_SYNC_FLUSH`、`Z_FULL_FLUSH`、または `Z_FINISH` のいずれかをとり、デフォルト値は `Z_FINISH` です。 `Z_SYNC_FLUSH` および `Z_FULL_FLUSH` ではこれ以後にもデータ文字列を圧縮できるモードです。一方、 `Z_FINISH` は圧縮ストリームを閉じ、これ以後のデータの圧縮を禁止します。 *mode* に `Z_FINISH` を設定して *flush()* メソッドを呼び出した後は、 *compress()* メソッドを再び呼ぶべきではありません。唯一の現実的な操作はこのオブジェクトを削除することだけです。

`Compress.copy()`

圧縮オブジェクトのコピーを返します。これを使うと先頭部分が共通している複数のデータを効率的に圧縮することができます。

バージョン 2.5 で追加。

展開オブジェクトは以下のメソッドと属性をサポートしています:

`Decompress.unused_data`

圧縮データの末尾より後のバイト列が入った文字列です。すなわち、この値は圧縮データの入っているバイト列の最後の文字が利用可能になるまでは "" のままとなります。入力文字列全てが圧縮データを含んでいた場合、この属性は ""、すなわち空文字列になります。

圧縮データ文字列がどこで終了しているかを決定する唯一の方法は、実際にそれを展開することです。つまり、大きなファイルの一部分に圧縮データが含まれているときに、その末端を調べるためには、データをファイルから読み出し、空でない文字列を後ろに続けて、 *unused_data* が空文字列でなくなるまで、展開オブジェクトの *decompress()* メソッドに入力しつづけるしかありません。

`Decompress.unconsumed_tail`

展開されたデータを収めるバッファの長さ制限を超えたために、直近の *decompress()* 呼び出しで処理しきれなかったデータを含む文字列オブジェクトです。このデータはまだ `zlib` 側からは見えていないので、正しい展開出力を得るには以降の *decompress()* メソッド呼び出しに (場合によっては後続のデータが追加された) データを差し戻さなければなりません。

`Decompress.decompress(string[, max_length])`

string を展開し、少なくとも *string* の一部分に対応する展開されたデータを含む文字列オブジェクトを返します。このデータは以前に *decompress()* メソッドを呼んだ時に返された出力と結合することができます。入力データの一部分が以後の処理のために内部バッファに保存されることもあります。

オプションパラメータ `max_length` が非ゼロの場合、返される展開データの長さが `max_length` 以下に制限されます。このことは入力した圧縮データの全てが処理されるとは限らないことを意味し、処理されなかったデータは `unconsumed_tail` 属性に保存されます。展開処理を継続したいならば、この保存されたデータを以降の `decompress()` 呼び出しに渡さなくてはなりません。`max_length` が与えられなかった場合、全ての入力が展開され、`unconsumed_tail` 属性は空文字列になります。

`Decompress.flush([length])`

未処理の入力データを全て処理し、最終的に圧縮されなかった残りの出力文字列オブジェクトを返します。`flush()` を呼んだ後、`decompress()` を再度呼ぶべきではありません。このときできる唯一の現実的な操作はオブジェクトの削除だけです。

オプション引数 `length` には出力バッファの初期サイズを指定します。

`Decompress.copy()`

展開オブジェクトのコピーを返します。これを使うとデータストリームの途中にある展開オブジェクトの状態を保存でき、未来のある時点で行なわれるストリームのランダムなシークをスピードアップするのに利用できます。

バージョン 2.5 で追加。

参考:

`gzip` モジュール `gzip` 形式ファイルへの読み書きを行うモジュール。

<http://www.zlib.net> `zlib` ライブラリホームページ。

<http://www.zlib.net/manual.html> `zlib` ライブラリの多くの関数の意味と使い方を解説したマニュアル。

12.2 gzip — gzip ファイルのサポート

ソースコード: `Lib/gzip.py`

このモジュールは、GNU の `gzip` や `gunzip` のようにファイルを圧縮、展開するシンプルなインターフェイスを提供しています。

データ圧縮は `zlib` モジュールで提供されています。

`gzip` モジュールは、Python のファイルオブジェクトに似た `GzipFile` クラスを提供しています。`GzipFile` クラスは `gzip` フォーマットのファイルを読み書きします。自動的にデータを圧縮・伸張するので、外からは通常のファイルオブジェクトのように見えます。

`compress` や `pack` 等によって作成され、`gzip` や `gunzip` が展開できる他のファイル形式についてはこのモジュールは対応していないので注意してください。

このモジュールは以下の項目を定義しています:

```
class gzip.GzipFile([filename[, mode[, compresslevel[, fileobj[, mtime]]]])
```

`GzipFile` クラスのコンストラクタです。`GzipFile` オブジェクトは `readinto()` メソッドと

`truncate()` メソッドを除くほとんどの `ファイルオブジェクト` のメソッドをシミュレートします。少なくとも `fileobj` および `filename` は有効な値でなければなりません。

クラスの新しいインスタンスは、`fileobj` に基づいて作成されます。`fileobj` は通常のファイル、`StringIO` オブジェクト、そしてその他ファイルをシミュレートできるオブジェクトでかまいません。値はデフォルトでは `None` で、ファイルオブジェクトを生成するために `filename` を開きます。

`fileobj` が `None` でない場合、`filename` 引数は `gzip` ファイルヘッダにインクルードされることのみに使用されます。`gzip` ファイルヘッダは圧縮されていないファイルの元の名前をインクルードするかもしれませんが、認識可能な場合、規定値は `fileobj` のファイル名です。そうでない場合、規定値は空の文字列で、元のファイル名はヘッダにはインクルードされません。

`mode` 引数は、ファイルを読み出すのか、書き込むのかによって、`'r'`、`'rb'`、`'a'`、`'ab'`、`'w'`、そして `'wb'` のいずれかになります。`fileobj` のファイルモードが認識可能な場合、`mode` はデフォルトで `fileobj` のモードと同じになります。そうでない場合、デフォルトのモードは `'rb'` です。`'b'` フラグがついていなくても、ファイルがバイナリモードで開かれることを保証するために `'b'` フラグが追加されます。これはプラットフォーム間での移植性のためです。

引数 `compresslevel` は 0 から 9 の整数を取り、圧縮レベルを制御します; 1 は最も高速で最小限の圧縮を行い、9 は最も低速ですが最大限の圧縮を行います。0 は圧縮しません。デフォルトは 9 です。

引数 `mtime` はオプションで、圧縮時にストリームに書かれる数値型のタイムスタンプです。`gzip` で圧縮されたすべてのストリームはタイムスタンプを必要とします。省略された場合や `None` が渡された場合は、現在の時刻が使用されます。このモジュールは展開時にはタイムスタンプを無視しますが、`gunzip` などの一部のプログラムはタイムスタンプを利用します。タイムスタンプのフォーマットは `time.time()` の戻り値や、`os.stat()` の戻り値となるオブジェクトの `st_mtime` 属性と同じです。

圧縮したデータの後ろにさらに何か追記したい場合もあるので、`GzipFile` オブジェクトの `close()` メソッド呼び出しは `fileobj` をクローズしません。この機能によって、書き込み用にオープンした `StringIO` オブジェクトを `fileobj` として渡し、`StringIO` オブジェクトの `getvalue()` メソッドを使って書き込んだデータの入っているメモリバッファを取得することもできます。

`GzipFile` はイテレーションと `with` 文をサポートします。

バージョン 2.7 で変更: `with` 文をサポートしました。

バージョン 2.7 で変更: zero-pad されたファイルのサポートが追加されました。

バージョン 2.7 で追加: `mtime` 引数が追加されました。

```
gzip.open(filename[, mode[, compresslevel]])
```

`GzipFile(filename, mode, compresslevel)` の短縮形です。引数 `filename` は必須です。デフォルトで `mode` は `'rb'` に、`compresslevel` は 9 に設定されています。

12.2.1 使い方の例

圧縮されたファイルを読み込む例:


```
import gzip
with gzip.open('file.txt.gz', 'rb') as f:
    file_content = f.read()
```

GZIP 圧縮されたファイルを作成する例:

```
import gzip
content = "Lots of content here"
with gzip.open('file.txt.gz', 'wb') as f:
    f.write(content)
```

既存のファイルを GZIP 圧縮する例:

```
import gzip
import shutil
with open('file.txt', 'rb') as f_in, gzip.open('file.txt.gz', 'wb') as f_out:
    shutil.copyfileobj(f_in, f_out)
```

参考:

`zlib` モジュール `gzip` ファイル形式のサポートを行うために必要な基本ライブラリモジュール。

12.3 bz2 — bzip2 互換の圧縮ライブラリ

バージョン 2.3 で追加。

このモジュールでは bz2 圧縮ライブラリのためのわかりやすいインタフェースを提供します。モジュールでは完全なファイルインタフェース、データを一括して圧縮/解凍する関数、データを逐次的に圧縮/解凍するための型を実装しています。

bz2 モジュールが提供している機能を以下にまとめます。

- `BZ2File` クラスは、`readline()`、`readlines()`、`writelines()`、`seek()` 等を含む、完全なファイルインタフェースを実装します。
- `BZ2File` クラスは `seek()` をエミュレーションでサポートします。
- `BZ2File` クラスは広範囲の改行文字バリエーション (universal newline) をサポートします。
- `BZ2File` クラスは、ファイルオブジェクトの先読みアルゴリズムを元にして実装されている、最適化された行単位のイテレーション機能を提供します。
- `BZ2Compressor` および `BZ2Decompressor` クラスでは逐次的圧縮 (解凍) をサポートしています。
- `compress()` および `decompress()` 関数は、一括圧縮/解凍機能を提供しています。
- 個別のロックメカニズムによってスレッド安全性を持っています。

注釈: Handling of multi-stream bzip2 files is not supported. Modules such as [bz2file](#) let you overcome this.

12.3.1 ファイルの圧縮/解凍

[BZ2File](#) クラスは圧縮ファイルの操作機能を提供しています。

```
class bz2.BZ2File(filename[, mode[, buffering[, compresslevel]]])
```

bz2 ファイルを開きます。 *mode* は 'r' (デフォルト) または 'w' で、それぞれ読み込みと書き込みに対応します。書き込み用に開いた場合、ファイルが存在しないなら新しく作成し、そうでない場合ファイルを切り詰めます。 *buffering* パラメタを与えた場合、0 はバッファリングなしを表し、それよりも大きい値はバッファサイズになります。デフォルトでは 0 です。圧縮レベル (*compresslevel*) を与える場合、値は 1 から 9 までの整数値でなければなりません。デフォルトの値は 9 です。ファイルを読み込む際に [universal newlines](#) モードを使う場合は 'U' を *mode* に追加します。このモードで開かれた入力ファイルの行末はどれも、Python からは '\n' として見えます。また、開かれているファイルオブジェクトは *newlines* 属性を持ち、None (まだ改行文字を読み込んでいない時), '\r', '\n', '\r\n' またはそれまでに読み込んだ全ての改行文字バリエーションを含むタプルになります。 [universal newlines](#) が利用できるのは読み込みだけです。 [BZ2File](#) が生成するインスタンスは通常のファイルインスタンスと同様のイテレーション操作をサポートしています。

[BZ2File](#) は `with` 構文をサポートしています。

バージョン 2.7 で変更: `with` 構文のサポートが追加されました。

注釈: This class does not support input files containing multiple streams (such as those produced by the [pbzip2](#) tool). When reading such an input file, only the first stream will be accessible. If you require support for multi-stream files, consider using the third-party [bz2file](#) module (available from [PyPI](#)). This module provides a backport of Python 3.3's [BZ2File](#) class, which does support multi-stream files.

close()

ファイルを閉じます。オブジェクトのデータ属性 `closed` を真にします。閉じたファイルはそれ以後入出力操作の対象にできません。 `close()` 自体の呼び出しはエラーを引き起こすことなく何度でも実行できます。

read([size])

最大で *size* バイトの解凍されたデータを読み出し、文字列として返します。 *size* 引数を負の値にした場合や省略した場合、EOF まで読み出します。

readline([size])

ファイルから次の 1 行を読み出し、改行文字も含めて文字列を返します。負でない *size* 値は、返される文字列の最大バイト長を制限します (その場合不完全な行を返すこともあります)。EOF の時には空文字列を返します。

readlines([size])

ファイルから読み取った各行の文字列からなるリストを返します。オプション引数 *size* を与えた場合、文字列リストの合計バイト数の大まかな上限の指定になります。

xreadlines()

前のバージョンとの互換性のために用意されています。 *BZ2File* オブジェクトはかつて *xreadlines* モジュールで提供されていたパフォーマンス最適化を含んでいます。

バージョン 2.3 で非推奨: このメソッドは *file* オブジェクトの同名のメソッドとの互換性のために用意されていますが、現在は推奨されないメソッドです。代わりに `for line in file` を使ってください。

seek(offset[, whence])

ファイルの読み書き位置を移動します。引数 *offset* はバイト数で指定したオフセット値です。オプション引数 *whence* はデフォルトで `os.SEEK_SET` もしくは 0 (ファイルの先頭からのオフセットで、`offset >= 0` になるはず) です。他にとり得る値は 1 (現在のファイル位置からの相対位置で、正負どちらの値もととり得る)、および 2 (ファイルの終末端からの相対位置で、通常は負の値になるが、多くのプラットフォームではファイルの終末端を越えて `seek` できる) です。

bz2 ファイルの `seek` はエミュレーションであり、パラメタの設定によっては処理が非常に低速になるかもしれないので注意してください。

tell()

現在のファイル位置を整数 (long 整数になるかもしれませんが) で返します。

write(data)

ファイルに文字列 *data* を書き込みます。バッファリングのため、ディスク上のファイルに書き込まれたデータを反映させるには `close()` が必要になるかもしれないので注意してください。

writelines(sequence_of_strings)

複数の文字列からなるシーケンスをファイルに書き込みます。それぞれの文字列を書き込む際に改行文字を追加することはありません。シーケンスはイテレーション処理で文字列を取り出せる任意のオブジェクトにできます。この操作はそれぞれの文字列を `write()` を呼んで書き込むのと同じ操作です。

12.3.2 逐次的な圧縮/解凍

逐次的な圧縮および解凍は *BZ2Compressor* および *BZ2Decompressor* クラスを用いて行います。

class bz2.BZ2Compressor([compresslevel])

新しい圧縮オブジェクトを作成します。このオブジェクトはデータを逐次的に圧縮できます。一括してデータを圧縮したいのなら、代わりに `compress()` 関数を使ってください。 *compresslevel* パラメタを与える場合、この値は 1 と 9 の間の整数でなければなりません。デフォルトの値は 9 です。

compress(data)

圧縮オブジェクトに追加のデータを入力します。圧縮データのチャンクを生成できた場合にはチャンクを返します。圧縮データの入力を終えた後は圧縮処理を終えるために `flush()` を呼んでください。内部バッファに残っている未処理のデータを返します。

flush()

圧縮処理を終え、内部バッファに残されているデータを返します。このメソッドの呼び出し以降は同じ圧縮オブジェクトを使ってはなりません。

class bz2.BZ2Decompressor

新しい解凍オブジェクトを生成します。このオブジェクトは逐次的にデータを解凍できます。一括してデータを解凍したいのなら、代わりに `decompress()` 関数を使ってください。

decompress(data)

解凍オブジェクトに追加のデータを入力します。可能な限り、解凍データのチャンクを生成できた場合にはチャンクを返します。ストリームの末端に到達した後に解凍処理を行おうとした場合には、例外 `EOFError` を送出します。ストリームの終末端の後ろに何らかのデータがあった場合、解凍処理はこのデータを無視し、オブジェクトの `unused_data` 属性に収めます。

12.3.3 一括圧縮/解凍

一括での圧縮および解凍を行うための関数、`compress()` および `decompress()` が提供されています。

bz2.compress(data[, compresslevel])

`data` を一括して圧縮します。データを逐次的に圧縮したいのなら、代わりに `BZ2Compressor` を使ってください。もし `compresslevel` パラメタを与えるなら、この値は 1 から 9 の間の値をとらなくてはなりません。デフォルトの値は 9 です。

bz2.decompress(data)

`data` を一括して解凍します。データを逐次的に解凍したいのなら、代わりに `BZ2Decompressor` を使ってください。

12.4 zipfile — ZIP アーカイブの処理

バージョン 1.6 で追加.

ソースコード: [Lib/zipfile.py](#)

ZIP は一般によく知られているアーカイブ (書庫化) および圧縮の標準ファイルフォーマットです。このモジュールでは ZIP 形式のファイルの作成、読み書き、追記、書庫内のファイル一覧の作成を行うためのツールを提供します。より高度な使い方でのこのモジュールを利用したいのであれば、[PKZIP Application Note](#) に定義されている ZIP ファイルフォーマットの理解が必要になるでしょう。

このモジュールは現在マルチディスク ZIP ファイルを扱うことはできません。ZIP64 拡張を利用する ZIP ファイル (サイズが 4 GiB を超えるような ZIP ファイル) は扱えます。このモジュールは暗号化されたアーカイブの復号をサポートしますが、現在暗号化ファイルを作成することはできません。C 言語ではなく、Python で実装されているため、復号は非常に遅くなっています。

このモジュールは以下の項目を定義しています:

exception `zipfile.BadZipfile`

正常ではない ZIP ファイルに対して送出されるエラーです (旧名称: `zipfile.error`)。

exception `zipfile.LargeZipFile`

ZIP ファイルが ZIP64 の機能を必要としているが、その機能が有効化されていない場合に送出されるエラーです。

class `zipfile.ZipFile`

ZIP ファイルの読み書きのためのクラスです。コンストラクタの詳細については、[ZipFile オブジェクト](#) 節を参照してください。

class `zipfile.PyZipFile`

Python ライブラリを含む、ZIP アーカイブを作成するためのクラスです。

class `zipfile.ZipInfo` (`[filename[, date_time]]`)

アーカイブ内の 1 個のメンバの情報を取得するために使うクラスです。このクラスのインスタンスは `ZipFile` オブジェクトの `getinfo()` および `infolist()` メソッドを返します。ほとんどの `zipfile` モジュールの利用者はこれらを作成する必要はなく、このモジュールによって作成されたものを使用できます。`filename` はアーカイブメンバのフルネームでなければならず、`date_time` はファイルが最後に変更された日時を表す 6 個のフィールドのタプルでなければなりません; フィールドは [ZipInfo オブジェクト](#) 節で説明されています。

`zipfile.is_zipfile(filename)`

`filename` が正しいマジックナンバをもつ ZIP ファイルの時に `True` を返し、そうでない場合 `False` を返します。`filename` にはファイルやファイルライクオブジェクトを渡すこともできます。

バージョン 2.7 で変更: ファイルおよびファイルライクオブジェクトをサポートしました。

`zipfile.ZIP_STORED`

アーカイブメンバを圧縮しない (複数ファイルを一つにまとめるだけ) ことを表す数値定数です。

`zipfile.ZIP_DEFLATED`

通常の ZIP 圧縮方法を表す数値定数です。これには `zlib` モジュールが必要です。現在のところ他の圧縮手法はサポートされていません。

参考:

[PKZIP Application Note](#) ZIP ファイルフォーマットおよびアルゴリズムを作成した Phil Katz によるドキュメント。

[Info-ZIP Home Page](#) Info-ZIP プロジェクトによる ZIP アーカイブプログラムおよびプログラム開発ライブラリに関する情報。

12.4.1 ZipFile オブジェクト

class `zipfile.ZipFile` (`file[, mode[, compression[, allowZip64]]]`)

ZIP ファイルを開きます。`file` はファイルへのパス名 (文字列) またはファイルのように振舞うオブジェクトのどちらでもかまいません。`mode` パラメタは、既存のファイルを読むためには `'r'`、既存のフ

イルを切り詰めたり新しいファイルに書き込むためには 'w'、追記を行うためには 'a' でなくてはなりません。mode が 'a' で file が既存の ZIP ファイルを参照している場合、追加するファイルは既存のファイル中の ZIP アーカイブに追加されます。file が ZIP を参照していない場合、新しい ZIP アーカイブが生成され、既存のファイルの末尾に追加されます。このことは、ある ZIP ファイルを他のファイル (例えば python.exe) に追加することを意味しています。

バージョン 2.6 で変更: mode が 'a' でファイルが存在しない場合、ファイルが作られるようになりました。

compression はアーカイブを書き出すときの ZIP 圧縮法で、ZIP_STORED または ZIP_DEFLATED でなくてはなりません。不正な値を指定すると `RuntimeError` が送出されます。また、ZIP_DEFLATED 定数が指定されているのに `zlib` モジュールを利用することができない場合も、`RuntimeError` が送出されます。デフォルト値は ZIP_STORED です。allowZip64 が True ならば 2GB より大きな ZIP ファイルの作成時に ZIP64 拡張を使用します。これが False (デフォルト) ならば、zipfile モジュールは ZIP64 拡張が必要になる場面で例外を送出します。ZIP64 拡張はデフォルトでは無効にされていますが、これは Unix の `zip` および `unzip` (InfoZIP ユーティリティ) コマンドがこの拡張をサポートしていないからです。

バージョン 2.7.1 で変更: ファイルがモード 'a' または 'w' で作成され、その後そのアーカイブにファイルを追加することなく クローズ された場合、空のアーカイブのための適切な ZIP 構造がファイルに書き込まれます。

ZipFile はコンテキストマネージャにもなっているので、with 文をサポートしています。次の例では、myzip は with 文のブロックが終了したときに、(たとえ例外が発生したとしても) クローズされます:

```
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

バージョン 2.7 で追加: ZipFile をコンテキストマネージャとして使用できるようになりました。

`ZipFile.close()`

アーカイブファイルをクローズします。close() はプログラムを終了する前に必ず呼び出さなければなりません。さもないとアーカイブ上の重要なレコードが書き込まれません。

`ZipFile.getinfo(name)`

アーカイブメンバ name に関する情報を持つ ZipInfo オブジェクトを返します。アーカイブに含まれないファイル名に対して getinfo() を呼び出すと、`KeyError` が送出されます。

`ZipFile.infolist()`

アーカイブに含まれる各メンバの ZipInfo オブジェクトからなるリストを返します。既存のアーカイブファイルを開いている場合、リストの順番は実際の ZIP ファイル中のメンバの順番と同じになります。

`ZipFile.namelist()`

アーカイブメンバの名前のリストを返します。

`ZipFile.open(name[, mode[, pwd]])`

ファイルライクオブジェクト (ZipExtFile) としてのアーカイブからメンバを 1 個展開します。name はアーカイブ内のファイルか、または ZipInfo オブジェクトの名前です。mode パラメータを指定する

場合、次の中のどれか一つでなければなりません: 'r' (デフォルト)、'U'、または 'rU'。'U' または 'rU' を選択すると、読み込み専用オブジェクトでは [ユニバーサル改行](#) サポートが有効になります。 `pwd` は暗号化ファイルで使用されているパスワードです。クローズしたファイルに対して `open()` が呼び出されると、`RuntimeError` が送出されます。

注釈: file-like オブジェクトは読み出し専用で、以下のメソッドを提供します: `read()`, `readline()`, `readlines()`, `__iter__()`, `next()`

注釈: ファイルライクオブジェクトをコンストラクタの第一引数として `ZipFile` ファイルが作成された場合、`open()` メソッドが返したオブジェクトは `ZipFile` のファイルポインタを共有します。この場合、`open()` が返したオブジェクトを、`ZipFile` オブジェクトに対する追加操作後に使用してはいけません。もし、`ZipFile` が文字列 (ファイル名) をコンストラクタに対する第一引数として作成された場合は、`open()` は `ZipExtFile` に含まれる、新しいファイルオブジェクトを作成します。これは `ZipFile` と独立して操作できます。

注釈: `open()`、`read()`、および `extract()` メソッドには、ファイル名または `ZipInfo` オブジェクトを指定できます。これは重複する名前のメンバを含む ZIP ファイルを読み込むときにそのメリットを享受できるでしょう。

バージョン 2.6 で追加。

`ZipFile.extract(member[, path[, pwd]])`

メンバをアーカイブから現在の作業ディレクトリに展開します。 `member` は展開するファイルのフルネームまたは `ZipInfo` オブジェクトでなければなりません。ファイル情報は可能な限り正確に展開されます。 `path` は展開先のディレクトリを指定します。 `member` はファイル名または `ZipInfo` オブジェクトです。 `pwd` は暗号化ファイルに使われるパスワードです。

作成された (ディレクトリか新ファイルの) 正規化されたパスを返します。

バージョン 2.6 で追加。

注釈: メンバのファイル名が絶対パスなら、ドライブ/UNC sharepoint および先頭の (バック) スラッシュは取り除かれます。例えば、Unix で `///foo/bar` は `foo/bar` となり、Window で `C:\foo\bar` は `foo\bar` となります。また、メンバのファイル名に含まれる全ての `".."` は取り除かれます。例えば、`../../foo../../ba..r` は `foo../ba..r` となります。Windows では、不正な文字 (`:`, `<`, `>`, `|`, `"`, `?`, および `*`) はアンダースコア (`_`) で置き換えられます。

`ZipFile.extractall([path[, members[, pwd]]])`

すべてのメンバをアーカイブから現在の作業ディレクトリに展開します。 `path` は展開先のディレクトリを指定します。 `members` は、オプションで、`namelist()` で返されるリストの部分集合でなければ

なりません。 *pwd* は、暗号化ファイルに使われるパスワードです。

警告: 内容を信頼できない tar アーカイブを、事前の内部チェック前に展開してはいけません。ファイルが *path* の外側に作られる可能性があります。例えば、`" / "` で始まる絶対パスのファイル名や、2 重ドット `" .. "` で始まるパスのファイル名です。

バージョン 2.7.4 で変更: `zipfile` モジュールは、それを防ごうと試みます。 `extract()` の注を参照してください。

バージョン 2.6 で追加.

`ZipFile.printdir()`

アーカイブの内容の一覧を `sys.stdout` に出力します。

`ZipFile.setpassword(pwd)`

pwd を展開する圧縮ファイルのデフォルトパスワードとして指定します。

バージョン 2.6 で追加.

`ZipFile.read(name[, pwd])`

アーカイブ中のファイル名 *name* の内容をバイト列にして返します。 *name* はアーカイブに含まれるファイル、もしくは、 `ZipInfo` オブジェクトの名前です。アーカイブは読み込みまたは追記モードでオープンされていなくてはなりません。 *pwd* は暗号化されたファイルのパスワードで、指定された場合、 `setpassword()` で指定されたデフォルトのパスワードを上書きします。クローズした `ZipFile` に対し `read()` を呼び出すと、 `RuntimeError` が送出されます。

バージョン 2.6 で変更: *pwd* が追加され、 *name* に `ZipInfo` オブジェクトを指定できるようになりました。

`ZipFile.testzip()`

アーカイブ中のすべてのファイルを読み、CRC チェックサムとヘッダが正常か調べます。最初に見つかった不正なファイルの名前を返します。不正なファイルがなければ `None` を返します。クローズした `ZipFile` に対して `testzip()` メソッドを呼び出すと、 `RuntimeError` が送出されます。

`ZipFile.write(filename[, arcname[, compress_type]])`

ファイル *filename* を、アーカイブ名 *arcname* (デフォルトは *filename* からドライブレターとパスセパレータを取り除いたもの) へ書き込みます。 *compress_type* を指定した場合、コンストラクタを使って新たなアーカイブエントリを生成した際に使った *compression* パラメタを上書きします。アーカイブのモードは `'w'` または `'a'` でなくてはなりません。モードが `'r'` で作成された `ZipFile` に対し `write()` メソッドを呼び出すと `RuntimeError` が送出されます。クローズした `ZipFile` に対し `write()` メソッドを呼び出すと `RuntimeError` が送出されます。

注釈: ZIP ファイル中のファイル名に関する公式なエンコーディングはありません。Unicode のファイル名が付けられている場合は、それを `write()` に渡す前に望ましいエンコーディングでバイト列に変換しなければなりません。WinZip はすべてのファイル名を DOS Latin としても知られる CP437 で解

積します。

注釈: アーカイブ名はアーカイブルートに対する相対パスでなければなりません。言い換えると、アーカイブ名はパスセパレータで始まってはいけません。

注釈: もし、`arcname` (`arcname` が与えられない場合は、`filename`) が `null byte` を含むなら、アーカイブ中のファイルのファイル名は、`null byte` までで切り詰められます。

`ZipFile.writestr(zinfo_or_arcname, bytes[, compress_type])`

文字列 `bytes` をアーカイブに書き込みます。`zinfo_or_arcname` には、アーカイブ中で指定するファイル名か、または `ZipInfo` インスタンスを指定します。`zinfo_or_arcname` に `ZipInfo` インスタンスを指定する場合、`zinfo` インスタンスには少なくともファイル名、日付および時刻を指定しなければなりません。ファイル名を指定した場合、日付と時刻には現在の日付と時間が設定されます。アーカイブはモード `'w'` または `'a'` でオープンされていなければなりません。モード `'r'` で作成された `ZipFile` に対し `writestr()` メソッドを呼び出すと `RuntimeError` が送出されます。クローズした `ZipFile` に対し `writestr()` メソッドを呼び出すと `RuntimeError` が送出されます。

`compress_type` が指定された場合、その値はコンストラクタに与えられた `compression` の値か、`zinfo_or_arcname` が `ZipInfo` のインスタンスだったときはその値をオーバーライドします。

注釈: `ZipInfo` インスタンスを引数 `zinfo_or_arcname` として与えた場合、与えられた `ZipInfo` インスタンスのメンバーである `compress_type` で指定された圧縮方法が使われます。デフォルトでは、`ZipInfo` コンストラクターが、このメンバーを `ZIP_STORED` に設定します。

バージョン 2.7 で変更: 引数 `compress_type` を追加しました。

以下のデータ属性も利用することができます:

`ZipFile.debug`

使用するデバッグ出力レベルです。この属性は 0 (デフォルト、何も出力しない) から 3 (最も多く出力する) までの値に設定することができます。デバッグ情報は `sys.stdout` に出力されます。

`ZipFile.comment`

ZIP ファイルに関連付けられたコメント文字列です。モード `'a'` または `'w'` で作成された `ZipFile` インスタンスへコメントを割り当てする場合、文字列長は 65535 バイトまでで、それを超えた場合は `close()` が呼び出されてアーカイブへ書き込む際に切り捨てられます。

12.4.2 PyZipFile オブジェクト

`PyZipFile` コンストラクタは `ZipFile` コンストラクタと同じパラメータを取ります。インスタンスは

`ZipFile` のメソッドの他に、追加のメソッドを一つ持ちます。

`PyZipFile.writepy(pathname[, basename])`

`*.py` ファイルを探し、`*.py` ファイルに対応するファイルをアーカイブに追加します。対応するファイルとは、もしあれば `*.pyo` であり、そうでなければ `*.pyc` で、必要に応じて `*.py` からコンパイルします。もし `pathname` がファイルなら、ファイル名は `.py` で終わっていなければなりません。また、(`*.py` に対応する `*.py[co]`) ファイルはアーカイブのトップレベルに (パス情報なしで) 追加されます。もし `pathname` が `.py` で終わらないファイル名なら `RuntimeError` を送出します。もし `pathname` がディレクトリで、ディレクトリがパッケージディレクトリでないなら、全ての `*.py[co]` ファイルはトップレベルに追加されます。もしディレクトリがパッケージディレクトリなら、全ての `*.py[co]` ファイルはパッケージ名の名前をもつファイルパスの下に追加されます。サブディレクトリがパッケージディレクトリなら、それらは再帰的に追加されます。 `basename` はクラス内部での呼び出しに使用するためのものです。 `writepy()` メソッドは以下のようなファイル名を持ったアーカイブを生成します。

<code>string.pyc</code>	<code># Top level name</code>
<code>test/__init__.pyc</code>	<code># Package directory</code>
<code>test/test_support.pyc</code>	<code># Module test.test_support</code>
<code>test/bogus/__init__.pyc</code>	<code># Subpackage directory</code>
<code>test/bogus/myfile.pyc</code>	<code># Submodule test.bogus.myfile</code>

12.4.3 ZipInfo オブジェクト

`ZipInfo` クラスのインスタンスは、`ZipFile` オブジェクトの `getinfo()` および `infolist()` メソッドによって返されます。各オブジェクトは ZIP アーカイブ内の 1 個のメンバに関する情報を格納します。

インスタンスは以下の属性を持ちます:

`ZipInfo.filename`

アーカイブ中のファイル名。

`ZipInfo.date_time`

アーカイブメンバの最終更新日時。6 つの値からなるタプルになります:

インデックス	値
0	西暦年 (>= 1980)
1	月 (1 から始まる)
2	日 (1 から始まる)
3	時 (0 から始まる)
4	分 (0 から始まる)
5	秒 (0 から始まる)

注釈: ZIP ファイルフォーマットは 1980 年より前のタイムスタンプをサポートしていません。

`ZipInfo.compress_type`

アーカイブメンバの圧縮形式。

`ZipInfo.comment`

各アーカイブメンバに対するコメント。

`ZipInfo.extra`

拡張フィールドデータ。この文字列に含まれているデータの内部構成については、[PKZIP Application Note](#) でコメントされています。

`ZipInfo.create_system`

ZIP アーカイブを作成したシステムを記述する文字列。

`ZipInfo.create_version`

このアーカイブを作成した PKZIP のバージョン。

`ZipInfo.extract_version`

このアーカイブを展開する際に必要な PKZIP のバージョン。

`ZipInfo.reserved`

予約領域。ゼロでなくてはなりません。

`ZipInfo.flag_bits`

ZIP フラグビット列。

`ZipInfo.volume`

ファイルヘッダのボリューム番号。

`ZipInfo.internal_attr`

内部属性。

`ZipInfo.external_attr`

外部ファイル属性。

`ZipInfo.header_offset`

ファイルヘッダへのバイトオフセット。

`ZipInfo.CRC`

圧縮前のファイルの CRC-32 チェックサム。

`ZipInfo.compress_size`

圧縮後のデータのサイズ。

`ZipInfo.file_size`

圧縮前のファイルのサイズ。

12.4.4 コマンドラインインターフェイス

`zipfile` モジュールは、ZIP アーカイブを操作するための簡単なコマンドラインインターフェイスを提供しています。

ZIP アーカイブを新規に作成したい場合、`-c` オプションの後にまとめたいファイルを列挙してください:

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

ディレクトリを渡すこともできます:

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

ZIP アーカイブを特定のディレクトリに展開したい場合、`-e` オプションを使用してください:

```
$ python -m zipfile -e monty.zip target-dir/
```

ZIP アーカイブ内のファイル一覧を表示するには `-l` を使用してください:

```
$ python -m zipfile -l monty.zip
```

コマンドラインオプション

`-l <zipfile>`

zipfile 内のファイル一覧を表示します。

`-c <zipfile> <source1> ... <sourceN>`

ソースファイルから zipfile を作成します。

`-e <zipfile> <output_dir>`

zipfile を対象となるディレクトリに展開します。

`-t <zipfile>`

zipfile が有効かどうか調べます。

12.5 tarfile — tar アーカイブファイルの読み書き

バージョン 2.3 で追加.

ソースコード: [Lib/tarfile.py](#)

`tarfile` モジュールは、gzip、bz2 圧縮されたものを含む、tar アーカイブを読み書きできます。`.zip` ファイルの読み書きには `zipfile` モジュールか、あるいは `shutil` の高レベル関数を使用してください。

いくつかの事実と形態:

- モジュールが利用可能な場合、`gzip`、`bz2` で圧縮されたアーカイブを読み書きします。
- POSIX.1-1988 (ustar) フォーマットの読み書きをサポートしています。
- `longname`、`longlink` 拡張を含む GNU tar フォーマットの読み書きをサポートしています。`sparse` 拡張は読み込みのみサポートしています。

- POSIX.1-2001 (pax) フォーマットの読み書きをサポートしています。

バージョン 2.6 で追加。

- ディレクトリ、一般ファイル、ハードリンク、シンボリックリンク、fifo、キャラクターデバイスおよびブロックデバイスを処理します。また、タイムスタンプ、アクセス許可や所有者のようなファイル情報の取得および保存が可能です。

注釈: Handling of multi-stream bzip2 files is not supported. Modules such as [bz2file](#) let you overcome this.

`tarfile.open(name=None, mode='r', fileobj=None, bufsize=10240, **kwargs)`

パス名 *name* の [TarFile](#) オブジェクトを返します。[TarFile](#) オブジェクトと、利用出来るキーワード引数に関する詳細な情報については、[TarFile オブジェクト](#) 節を参照してください。

mode は `'filemode[:compression]'` の形式をとる文字列でなければなりません。デフォルトの値は `'r'` です。以下に *mode* のとりうる組み合わせすべてを示します:

mode	action
'r' または 'r:*	圧縮方法に関して透過的に、読み込み用にオープンします (推奨)。
'r:'	非圧縮で読み込み用に排他的にオープンします。
'r:gzip'	gzip 圧縮で読み込み用にオープンします。
'r:bzip2'	bzip2 圧縮で読み込み用にオープンします。
'a' または 'a:'	非圧縮で追記用にオープンします。ファイルが存在しない場合は新たに作成されます。
'w' または 'w:'	非圧縮で書き込み用にオープンします。
'w:gzip'	gzip 圧縮で書き込み用にオープンします。
'w:bzip2'	bzip2 圧縮で書き込み用にオープンします。

`'a:gzip'`、`'a:bzip2'` は利用できないことに注意して下さい。もし *mode* が、ある (圧縮した) ファイルを読み込み用にオープンするのに適していないなら、[ReadError](#) が送出されます。これを防ぐには *mode* `'r'` を使って下さい。もし圧縮方式がサポートされていないならば、[CompressionError](#) が送出されます。

もし *fileobj* が指定されていれば、それは *name* でオープンされた [ファイルオブジェクト](#) の代替として使うことができます。そのファイルオブジェクトの位置が 0 であることを前提に動作します。

`'w:gzip'`、`'r:gzip'`、`'w:bzip2'`、`'r:bzip2'` モードの場合、[tarfile.open\(\)](#) はファイルの圧縮レベルを指定するキーワード引数 *compresslevel* (デフォルトは 9) を受け付けます。

特別な目的のために、*mode* には 2 番目の形式: `'ファイルモード|[圧縮方式]'` があります。この形式を使うと、[tarfile.open\(\)](#) が返すのは、データをブロックからなるストリームとして扱う [TarFile](#) オブジェクトになります。この場合、ファイルに対してランダムなシークが行えなくなります。*fileobj* を指定する場合、`read()` および `write()` メソッドを持つ (*mode* に依存した) 任意のオ

プロジェクトにできます。 *bufsize* にはブロックサイズを指定します。デフォルトは 20 * 512 バイトです。 `sys.stdin`、ソケットファイルオブジェクト、あるいはテープデバイスと組み合わせる場合にはこの形式を使ってください。ただし、このような *TarFile* オブジェクトにはランダムアクセスを行えないという制限があります。例節を参照してください。現在可能なモードは以下のとおりです。

モード	動作
'r *'	tar ブロックの <i>stream</i> を圧縮方法に関して透過的に読み込み用にオープンします。
'r '	非圧縮 tar ブロックの <i>stream</i> を読み込み用にオープンします。
'r gz'	gzip 圧縮の <i>stream</i> を読み込み用にオープンします。
'r bz2'	bzip2 圧縮の <i>stream</i> を読み込み用にオープンします。
'w '	非圧縮の <i>stream</i> を書き込み用にオープンします。
'w gz'	gzip 圧縮の <i>stream</i> を書き込み用にオープンします。
'w bz2'	bzip2 圧縮の <i>stream</i> を書き込み用にオープンします。

class `tarfile.TarFile`

tar アーカイブを読み書きするためのクラスです。このクラスを直接使わず、代わりに `tarfile.open()` を使ってください。 *TarFile* オブジェクトを参照してください。

`tarfile.is_tarfile(name)`

もし *name* が tar アーカイブファイルであり、 `tarfile` モジュールで読み込める場合に `True` を返します。

class `tarfile.TarFileCompat(filename, mode='r', compression=TAR_PLAIN)`

zipfile 風なインターフェースを持つ tar アーカイブへの制限されたアクセスのためのクラスです。詳細は *zipfile* のドキュメントを参照してください。 *compression* は、以下の定数のどれかでなければなりません:

TAR_PLAIN

非圧縮 tar アーカイブのための定数。

TAR_GZIPPED

gzip 圧縮 tar アーカイブのための定数。

バージョン 2.6 で非推奨: *TarFileCompat* クラスは Python 3 で削除されました。

exception `tarfile.TarError`

すべての `tarfile` 例外のための基本クラスです。

exception `tarfile.ReadError`

tar アーカイブがオープンされた時、 `tarfile` モジュールで操作できないか、あるいは何か無効であるとき送出されます。

exception `tarfile.CompressionError`

圧縮方法がサポートされていないか、あるいはデータを正しくデコードできない時に送出されます。

exception `tarfile.StreamError`

ストリームのような *TarFile* オブジェクトで典型的な制限のために送出されます。

exception `tarfile.ExtractError`

`TarFile.extract()` を使った時に 致命的でないエラーに対して送出されます。ただし `TarFile.errorlevel == 2` の場合に限ります。

モジュールレベルで以下の定数が利用出来ます。

`tarfile.ENCODING`

既定の文字エンコーディング。Windows では `'utf-8'`、それ以外では `sys.getfilesystemencoding()` の返り値です。

exception `tarfile.HeaderError`

`TarInfo.frombuf()` メソッドが取得したバッファーが不正だったときに送出されます。

バージョン 2.6 で追加。

以下の各定数は、`tarfile` モジュールが作成できる tar アーカイブフォーマットを定義しています。詳細は、サポートしている `tar` フォーマット を参照してください。

`tarfile.USTAR_FORMAT`

POSIX.1-1988 (ustar) フォーマット。

`tarfile.GNU_FORMAT`

GNU tar フォーマット。

`tarfile.PAX_FORMAT`

POSIX.1-2001 (pax) フォーマット。

`tarfile.DEFAULT_FORMAT`

アーカイブを作成する際のデフォルトのフォーマット。現在は `GNU_FORMAT` です。

参考:

`zipfile` モジュール `zipfile` 標準モジュールのドキュメント。

アーカイブ化操作 `shutil` が提供するより高水準のアーカイブ機能についてのドキュメント。

GNU tar manual, Basic Tar Format GNU tar 拡張機能を含む、tar アーカイブファイルのためのドキュメント。

12.5.1 TarFile オブジェクト

`TarFile` オブジェクトは、tar アーカイブへのインターフェースを提供します。tar アーカイブは一連のブロックです。アーカイブメンバー (保存されたファイル) は、ヘッダーブロックとそれに続くデータブロックで構成されています。一つの tar アーカイブにファイルを何回も保存することができます。各アーカイブメンバーは、`TarInfo` オブジェクトで確認できます。詳細については `TarInfo` オブジェクト を参照してください。

`TarFile` オブジェクトは `with` 文のコンテキストマネージャーとして利用できます。`with` ブロックが終了したときにオブジェクトはクローズされます。例外が発生した時、内部で利用されているファイルオブジェクトのみがクローズされ、書き込み用にオープンされたアーカイブのファイナライズは行われないことに注意してください。例節のユースケースを参照してください。

バージョン 2.7 で追加: コンテキスト管理のプロトコルがサポートされました。

```
class tarfile.TarFile(name=None, mode='r', fileobj=None, format=DEFAULT_FORMAT,
                      tarinfo=TarInfo, dereference=False, ignore_zeros=False, encoding=ENCODING, errors=None, pax_headers=None, debug=0, errorlevel=0)
```

以下のすべての引数はオプションで、インスタンス属性としてもアクセスできます。

name はアーカイブのパス名です。 *fileobj* が渡された場合は省略可能です。その場合、ファイルオブジェクトに *name* 属性があれば、それを利用します。

mode は、既存のアーカイブから読み込むための 'r'、既存のアーカイブに追記するための 'a'、あるいは既存のファイルがあれば上書きして新しいファイルを作成する 'w' のいずれかです。

fileobj が与えられていれば、それを使ってデータを読み書きします。もしそれが決定できれば、*mode* は *fileobj* のモードで上書きされます。 *fileobj* は位置 0 から利用されます。

注釈: *TarFile* をクローズした時、 *fileobj* はクローズされません。

format はアーカイブのフォーマットを制御します。モジュールレベルで定義されている、 *USTAR_FORMAT*、 *GNU_FORMAT*、あるいは *PAX_FORMAT* のいずれかである必要があります。

バージョン 2.6 で追加.

tarinfo 引数を利用して、デフォルトの *TarInfo* クラスを別のクラスで置き換えることができます。

バージョン 2.6 で追加.

dereference が *False* だった場合、シンボリックリンクやハードリンクがアーカイブに追加されます。 *True* だった場合、リンクのターゲットとなるファイルの内容がアーカイブに追加されます。シンボリックリンクをサポートしていないシステムでは効果がありません。

ignore_zeros が *False* だった場合、空ブロックをアーカイブの終端として扱います。 *True* だった場合、空の (無効な) ブロックをスキップして、可能な限り多くのメンバーを取得しようとします。このオプションは、連結されたり、壊れたアーカイブファイルを扱うときにのみ、意味があります。

debug は 0 (デバッグメッセージ無し) から 3 (全デバッグメッセージ) まで設定できます。このメッセージは `sys.stderr` に書き込まれます。

errorlevel が 0 の場合、 *TarFile.extract()* 使用時にすべてのエラーが無視されます。エラーが無視された場合でも、 *debug* が有効であれば、エラーメッセージは出力されます。 *errorlevel* が '1' の場合、すべての致命的な (*fatal*) エラーは *OSError* または *IOError* を送出します。2 の場合、すべての致命的でない (*non-fatal*) エラーも *TarError* 例外として送出されます。

encoding と *errors* 引数は、文字列と unicode オブジェクトとの間の相互変換方法を指定します。デフォルトの設定で、ほとんどのユーザーでうまく動作するでしょう。詳しい情報は、 [Unicode に関する問題節](#) を参照してください。

バージョン 2.6 で追加.

`pax_headers` 引数は、オプションの、unicode 文字列の辞書で、`format` が `PAX_FORMAT` だった場合に `pax` グローバルヘッダに追加されます。

バージョン 2.6 で追加.

classmethod `TarFile.open(...)`

代替コンストラクターです。モジュールレベルでの `tarfile.open()` 関数は、実際はこのクラスメソッドへのショートカットです。

`TarFile.getmember(name)`

メンバー `name` に対する `TarInfo` オブジェクトを返します。`name` がアーカイブに見つからなければ、`KeyError` が送出されます。

注釈: アーカイブ内にメンバーが複数ある場合は、最後に出現するものが最新のバージョンとみなされます。

`TarFile.getmembers()`

`TarInfo` アーカイブのメンバーをオブジェクトのリストとして返します。このリストはアーカイブ内のメンバーと同じ順番です。

`TarFile.getnames()`

メンバーをその名前前のリストを返します。これは `getmembers()` で返されるリストと同じ順番です。

`TarFile.list(verbose=True)`

内容の一覧を `sys.stdout` に出力します。`verbose` が `False` であれば、メンバー名のみ表示します。`True` であれば、`ls -l` に似た出力を生成します。

`TarFile.next()`

`TarFile` が読み込み用にオープンされている時、アーカイブの次のメンバーを `TarInfo` オブジェクトとして返します。もしそれ以上利用可能なものがなければ、`None` を返します。

`TarFile.extractall(path=".", members=None)`

すべてのメンバーをアーカイブから現在の作業ディレクトリまたは `path` に抽出します。オプションの `members` が与えられるときには、`getmembers()` で返されるリストの一部でなければなりません。所有者、変更時刻、アクセス権限のようなディレクトリ情報はすべてのメンバーが抽出された後にセットされます。これは二つの問題を回避するためです。一つはディレクトリの変更時刻はその中にファイルが作成されるたびにリセットされるということ、もう一つはディレクトリに書き込み許可がなければその中のファイル抽出は失敗してしまうということです。

警告: 内容を信頼できない tar アーカイブを、事前の内部チェック前に展開してはいけません。ファイルが `path` の外側に作られる可能性があります。例えば、`"/` で始まる絶対パスのファイル名や、2重ドット `"..` で始まるパスのファイル名です。

バージョン 2.5 で追加.

`TarFile.extract(member, path='')`

アーカイブからメンバーの完全な名前を使って、現在のディレクトリに展開します。ファイル情報はできる限り正確に展開されます。 `member` はファイル名もしくは `TarInfo` オブジェクトです。 `path` を使って別のディレクトリを指定することもできます。

注釈: `extract()` メソッドはいくつかの展開に関する問題を扱いません。ほとんどの場合、`extractall()` メソッドの利用を考慮すべきです。

警告: `extractall()` の警告を参してください。

`TarFile.extractfile(member)`

アーカイブからメンバーをファイルオブジェクトとして抽出します。 `member` は、ファイル名あるいは `TarInfo` オブジェクトです。もし `member` が普通のファイルであれば、ファイル風のオブジェクトを返します。もし `member` がリンクであれば、ファイル風のオブジェクトをリンクのターゲットから構成します。もし `member` が上のどれでもなければ、 `None` が返されます。

注釈: ファイル風のオブジェクトは読み出し専用です。このオブジェクトは `read()`, `readline()`, `readlines()`, `seek()`, `tell()`, `close()` の各メソッドを提供し、行に対するイテレーションをサポートします。

`TarFile.add(name, arcname=None, recursive=True, exclude=None, filter=None)`

ファイル `name` をアーカイブに追加します。 `name` は、任意のファイルタイプ (ディレクトリ、fifo、シンボリックリンク等) です。もし `arcname` が与えられていれば、それはアーカイブ内のファイルの代替名を指定します。デフォルトではディレクトリは再帰的に追加されます。これは、 `recursive` を `False` に設定することで避けることができます。 `exclude` を指定する場合、それは1つのファイル名を引数にとってブール値を返す関数である必要があります。この関数の戻り値が `True` の場合、そのファイルが除外されます。 `False` の場合、そのファイルは追加されます。 `filter` を指定する場合、それは `TarInfo` オブジェクトを引数として受け取り、操作した `TarInfo` オブジェクトを返す関数でなければなりません。代わりに `None` を返した場合、 `TarInfo` オブジェクトはアーカイブから除外されません。例にある例を参照してください。

バージョン 2.6 で変更: `exclude` パラメータが追加されました。

バージョン 2.7 で変更: `filter` パラメータが追加されました。

バージョン 2.7 で非推奨: `exclude` 引数は廃止予定です。代わりに `filter` 引数を利用してください。将来 `exclude` 引数が削除されたときに互換性を保つため、 `filter` は位置引数ではなくてキーワード引数として利用すべきです。

`TarFile.addfile(tarinfo, fileobj=None)`

`TarInfo` オブジェクト `tarinfo` をアーカイブに追加します。 `fileobj` が与えられている場合は、 `tarinfo.size` バイトがそれから読まれ、アーカイブに追加されます。 `TarInfo` オブジェクト

は直接もしくは `gettaringfo()` を使って作成することができます。

注釈: Windows プラットフォームでは、`fileobj` は、ファイルサイズに関する問題を避けるために、常にモード `'rb'` でオープンするべきです。

`TarFile.gettarinfo(name=None, arcname=None, fileobj=None)`

`os.stat()` の結果が、既存のファイルに相当するものから、`TarInfo` オブジェクトを作成します。このファイルは、`name` で名付けられるか、ファイル記述子を持つ *file object* `fileobj` として指定されます。`arcname` が与えられた場合、アーカイブ内のファイルに対して代替名を指定します。与えられない場合、名前は `fileobj` の `name` 属性、もしくは `name` 引数から取られます。

`TarInfo` の属性の一部は、`addfile()` を使用して追加する前に修正できます。ファイルオブジェクトが、ファイルの先頭にある通常のファイルオブジェクトでない場合、`size` などの属性は修正が必要かもしれません。これは、`GzipFile` などの属性に当てはまります。`name` も修正できるかもしれず、この場合、`arcname` はダミーの文字列にすることができます。

`TarFile.close()`

`TarFile` をクローズします。書き込みモードでは、完了ゼロブロックが 2 個アーカイブに追加されます。

`TarFile.posix`

この値を `True` にすることは、`format` を `USTAR_FORMAT` にすることと同じです。この値を `False` にすることは、`format` を `GNU_FORMAT` にすることと同じです。

バージョン 2.4 で変更: `posix` のデフォルト値が `False` になりました。

バージョン 2.6 で非推奨: 代わりに `format` 属性を利用してください。

`TarFile.pax_headers`

pax グローバルヘッダーに含まれる key-value ペアの辞書です。

バージョン 2.6 で追加.

12.5.2 TarInfo オブジェクト

`TarInfo` オブジェクトは `TarFile` の一つのメンバーを表します。ファイルに必要なすべての属性 (ファイルタイプ、ファイルサイズ、時刻、アクセス権限、所有者等のような) を保存する他に、そのタイプを決定するのに役に立ついくつかのメソッドを提供します。これにはファイルのデータそのものは含まれません。

`TarInfo` オブジェクトは `TarFile` のメソッド `getmember()`、`getmembers()` および `gettaringfo()` によって返されます。

`class tarfile.TarInfo(name="")`

`TarInfo` オブジェクトを作成します。

`TarInfo.frombuf(buf)`

`TarInfo` オブジェクトを文字列バッファ `buf` から作成して返します。

バージョン 2.6 で追加: バッファが不正な場合 `HeaderError` を送出します。

`TarInfo.fromtarfile(tarfile)`

`TarFile` オブジェクトの `tarfile` から、次のメンバーを読み込んで、それを `TarInfo` オブジェクトとして返します。

バージョン 2.6 で追加.

`TarInfo.tobuf(format=DEFAULT_FORMAT, encoding=ENCODING, errors='strict')`

`TarInfo` オブジェクトから文字列バッファを作成します。引数についての情報は、`TarFile` クラスのコンストラクターを参照してください。

バージョン 2.6 で変更: 引数が追加されました。

`TarInfo` オブジェクトには以下のデータ属性があります:

`TarInfo.name`

アーカイブメンバーの名前。

`TarInfo.size`

バイト単位でのサイズ。

`TarInfo.mtime`

最後に変更された時刻。

`TarInfo.mode`

許可ビット。

`TarInfo.type`

ファイルタイプ。 `type` は通常、定数 `REGTYPE`、`AREGTYPE`、`LNKTYPE`、`SYMTYPE`、`DIRTYPE`、`FIFOTYPE`、`CONTTYPE`、`CHRTYPE`、`BLKTYPE`、あるいは `GNUTYPE_SPARSE` のいずれかです。
`TarInfo` オブジェクトのタイプをもっと簡単に解決するには、下記の `is*()` メソッドを使って下さい。

`TarInfo.linkname`

リンク先ファイルの名前。これはタイプ `LNKTYPE` と `SYMTYPE` の `TarInfo` オブジェクトにだけ存在します。

`TarInfo.uid`

ファイルメンバーを保存した元のユーザーのユーザー ID。

`TarInfo.gid`

ファイルメンバーを保存した元のユーザーのグループ ID。

`TarInfo.uname`

ファイルメンバーを保存した元のユーザーのユーザー名。

`TarInfo.gname`

ファイルメンバーを保存した元のユーザーのグループ名。

`TarInfo.pax.headers`

pax 拡張ヘッダーに関連付けられた、key-value ペアの辞書。

バージョン 2.6 で追加.

`TarInfo` オブジェクトは便利な照会用のメソッドもいくつか提供しています:

`TarInfo.isfile()`

`TarInfo` オブジェクトが一般ファイルの場合に、`True` を返します。

`TarInfo.isreg()`

`isfile()` と同じです。

`TarInfo.isdir()`

ディレクトリの場合に `True` を返します。

`TarInfo.issym()`

シンボリックリンクの場合に `True` を返します。

`TarInfo.islnk()`

ハードリンクの場合に `True` を返します。

`TarInfo.ischr()`

キャラクターデバイスの場合に `True` を返します。

`TarInfo.isblk()`

ブロックデバイスの場合に `True` を返します。

`TarInfo.isfifo()`

FIFO の場合に `True` を返します。

`TarInfo.isdev()`

キャラクターデバイス、ブロックデバイスあるいは FIFO のいずれかの場合に `True` を返します。

12.5.3 例

tar アーカイブから現在のディレクトリにすべて抽出する方法:

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall()
tar.close()
```

tar アーカイブの一部を、リストの代わりにジェネレーター関数を利用して `TarFile.extractall()` で展開する方法:

```
import os
import tarfile

def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py":
            yield tarinfo
```

(次のページに続く)

(前のページからの続き)

```
tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

非圧縮 tar アーカイブをファイル名のリストから作成する方法:

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

with 文を利用した同じ例:

```
import tarfile
with tarfile.open("sample.tar", "w") as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

gzip 圧縮 tar アーカイブを作成してメンバー情報のいくつかを表示する方法:

```
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print tarinfo.name, "is", tarinfo.size, "bytes in size and is",
    if tarinfo.isreg():
        print "a regular file."
    elif tarinfo.isdir():
        print "a directory."
    else:
        print "something else."
tar.close()
```

`TarFile.add()` 関数の `filter` 引数を利用してユーザー情報をリセットしながらアーカイブを作成する方法:

```
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
    tarinfo.uname = tarinfo.gname = "root"
    return tarinfo
tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```

12.5.4 サポートしている tar フォーマット

`tarfile` モジュールは 3 種類の tar フォーマットを作成することができます:

- POSIX.1-1988 ustar format (*USTAR_FORMAT*). ファイル名の長さは 256 文字までで、リンク名の長さは 100 文字までです。最大のファイルサイズは 8GiB です。このフォーマットは古くて制限が多いですが、広くサポートされています。
- GNU tar format (*GNU_FORMAT*). 長いファイル名とリンク名、8GiB を超えるファイルやスパーズ (sparse) ファイルをサポートしています。これは GNU/Linux システムにおいてデファクト・スタンダードになっています。 *tarfile* モジュールは長いファイル名を完全にサポートしています。スパーズファイルは読み込みのみサポートしています。
- POSIX.1-2001 pax フォーマット (*PAX_FORMAT*). 最も柔軟性があり、ほぼ制限が無いフォーマットです。長いファイル名やリンク名、大きいファイルをサポートし、パス名をポータブルな方法で保存します。しかし、現在のところ、すべての tar の実装が pax フォーマットを正しく扱えるわけではありません。

pax フォーマットは既存の ustar フォーマットの拡張です。ustar では保存できない情報を追加のヘッダーを利用して保存します。pax には 2 種類のヘッダーがあります。1 つ目は拡張ヘッダーで、その次のファイルヘッダーに影響します。2 つ目はグローバルヘッダーで、アーカイブ全体に対して有効で、それ以降のすべてのファイルに影響します。すべての pax ヘッダーの内容は、ポータブル性のために UTF-8 で保存されます。

他にも、読み込みのみサポートしている tar フォーマットがいくつかあります:

- ancient V7 フォーマット。これは Unix 7th Edition から存在する、最初の tar フォーマットです。通常のファイルとディレクトリのみ保存します。名前は 100 文字を超えてはならず、ユーザー/グループ名に関する情報は保存されません。いくつかのアーカイブは、フィールドが ASCII でない文字を含む場合に、ヘッダーのチェックサムを計算を誤ります。
- SunOS tar 拡張フォーマット。POSIX.1-2001 pax フォーマットの亜流ですが、互換性がありません。

12.5.5 Unicode に関する問題

tar フォーマットはもともと、テープドライブにファイルシステムのバックアップを取る目的で設計されました。現在、tar アーカイブはファイルを配布する場合に一般的に用いられ、ネットワーク越しに送受信されます。オリジナルのフォーマットの抱える 1 つの問題 (ほか多くのフォーマットも同じですが) は、文字エンコーディングが異なる環境を考慮していないことです。例えば、通常の UTF-8 の環境で作成されたアーカイブは、非 ASCII 文字を含んでいた場合 *Latin-1* のシステムでは正しく読み込むことができません。非 ASCII 文字を含む名前 (ファイル名、リンク名、ユーザー/グループ名) が破壊されます。不幸なことに、アーカイブのエンコーディングを自動検出する方法はありません。

pax フォーマットはこの問題を解決するように設計されました。このフォーマットは、非 ASCII 文字の名前を UTF-8 で保存します。pax アーカイブを読み込むときに、この UTF-8 の名前がローカルのファイルシステムのエンコーディングに変換されます。

unicode 変換の動作は、*TarFile* クラスの *encoding* と *errors* キーワード引数によって制御されます。

encoding のデフォルト値はローカルの文字エンコーディングです。これは *sys.getfilesystemencoding()* と *sys.getdefaultencoding()* から取得されます。読み込みモ-

ドでは、*encoding* は pax フォーマット内の unicode の名前をローカルの文字エンコーディングに変換するために利用されます。書き込みモードでは、*encoding* の扱いは選択されたアーカイブフォーマットに依存します。[PAX_FORMAT](#) の場合、入力された非 ASCII 文字を含む文字は UTF-8 文字列として保存する前に一旦デコードする必要があるので、そこで *encoding* が利用されます。それ以外のフォーマットでは、*encoding* は、入力された名前に unicode が含まれない限りは利用されません。unicode が含まれている場合、アーカイブに保存する前に *encoding* でエンコードされます。

errors 引数は、*encoding* を利用して変換できない文字の扱いを指定します。利用可能な値は、[Codec 基底クラス](#) 節でリストアップされています。読み込みモードでは、追加の値として 'utf-8' を選択することができます。エラーが発生したときは UTF-8 を利用することができます。(これがデフォルトです) 書き込みモードでは、*errors* のデフォルト値は 'strict' になっていて、名前が気づかなくうちに変化することが無いようにしています。

第 13 章

ファイルフォーマット

この章で説明されるモジュールはマークアップや e-mail でない様々なファイルフォーマットを構文解析します。

13.1 csv — CSV ファイルの読み書き

バージョン 2.3 で追加.

CSV (Comma Separated Values、カンマ区切り値列) と呼ばれる形式は、スプレッドシートやデータベース間でのデータのインポートやエクスポートにおける最も一般的な形式です。“CSV 標準”は存在しないため、CSV 形式はデータを読み書きする多くのアプリケーション上の操作に応じて定義されているにすぎません。標準がないということは、異なるアプリケーションによって生成されたり取り込まれたりするデータ間では、しばしば微妙な違いが発生するということを意味します。こうした違いのために、複数のデータ源から得られた CSV ファイルを処理する作業が鬱陶しいものになることがあります。とはいえ、デリミタ (delimiter) やクオート文字の相違はあっても、全体的な形式は十分似通っているため、こうしたデータを効率的に操作し、データの読み書きにおける細々としたことをプログラマから隠蔽するような単一のモジュールを書くことは可能です。

`csv` モジュールでは、CSV 形式で書かれたテーブル状のデータを読み書きするためのクラスを実装しています。このモジュールを使うことで、プログラマは Excel で使われている CSV 形式に関して詳しい知識をなくても、“このデータを Excel で推奨されている形式で書いてください”とか、“データを Excel で作成されたこのファイルから読み出してください”と言うことができます。プログラマはまた、他のアプリケーションが解釈できる CSV 形式を記述したり、独自の特殊な目的をもった CSV 形式を定義することもできます。

`csv` モジュールの `reader` および `writer` オブジェクトはシーケンス型を読み書きします。プログラマは `DictReader` や `DictWriter` クラスを使うことで、データを辞書形式で読み書きすることもできます。

注釈: このバージョンの `csv` モジュールは Unicode 入力をサポートしていません。また、現在のところ、ASCII NUL 文字に関連したいくつかの問題があります。従って、安全を期すには、全ての入力を UTF-8 または印字可能な ASCII にしなければなりません。これについては [使用例](#) 節の例を参照してください。

参考:

PEP 305 - CSV File API Python へのこのモジュールの追加を提案している Python 改良案 (PEP: Python Enhancement Proposal)。

13.1.1 モジュールコンテンツ

`csv` モジュールでは以下の関数を定義しています:

`csv.reader(csvfile, dialect='excel', **fmtparams)`

与えられた `csvfile` 内の行を反復処理するような reader オブジェクトを返します。 `csvfile` はイテレータ (*iterator*) プロトコルをサポートし、 `next()` メソッドが呼ばれた際に常に文字列を返すような任意のオブジェクトにすることができます — ファイルオブジェクトでも リストでも構いません。 `csvfile` がファイルオブジェクトの場合、ファイルオブジェクトの形式に違いがあるようなプラットフォームでは 'b' フラグを付けて開かなければなりません。 オプションとして `dialect` パラメタを与えることができ、特定の CSV 表現形式 (`dialect`) 特有のパラメタの集合を定義するために使われます。 `dialect` パラメタは `Dialect` クラスのサブクラス のインスタンスか、 `list_dialects()` 関数が返す文字列の一つにすることができます。 別のオプションである `fmtparams` キーワード引数は、現在の表現形式における個々の書式パラメタを上書きするために与えることができます。 表現形式および書式化パラメタの詳細については、 [Dialect クラスと書式化パラメタ](#) 節を参照してください。

csv ファイルから読み込まれた各行は、文字列のリストとして返されます。 データ型の変換が自動的に行われることはありません。

短い利用例:

```
>>> import csv
>>> with open('eggs.csv', 'rb') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
...     for row in spamreader:
...         print ', '.join(row)
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

バージョン 2.5 で変更: パーサが複数行に亘るクオートされたフィールドに関して厳格になりました。以前は、クオートされたフィールドの中で終端の改行文字無しに行が終わった場合、返されるフィールドには改行が挿入されていましたが、この振る舞いはフィールドの中に復帰文字を含むようなファイルを読むときに問題を起こしていました。そこでフィールドに改行文字を挿入せずに返すように改められました。この結果、フィールドに埋め込まれた改行文字が重要ならば、入力に改行文字を保存するような仕方で複数行に分割されるはずですが。

`csv.writer(csvfile, dialect='excel', **fmtparams)`

ユーザが与えたデータをデリミタで区切られた文字列に変換し、与えられたファイルオブジェクトに書き込むための writer オブジェクトを返します。 `csvfile` は `write()` メソッドを持つ任意のオブジェクトです。 `csvfile` がファイルオブジェクトの場合、 'b' フラグが意味を持つプラットフォームでは 'b' フラグを付けて開かなければなりません。 オプションとして `dialect` 引数を与えることができ、利用する CSV 表現形式 (`dialect`) を指定することができます。 `dialect` パラメタは `Dialect` クラスのサブクラス

のインスタンスか、`list_dialects()` 関数が返す文字列の 1 つにすることができます。別のオプション引数である `fmtparams` キーワード引数は、現在の表現形式における個々の書式パラメタを上書きするために与えることができます。`dialect` と書式パラメタについての詳細は、[Dialect クラスと書式化パラメタ](#) 節を参照してください。DB API を実装するモジュールとのインタフェースを可能な限り容易にするために、`None` は空文字列として書き込まれます。この処理は可逆な変換ではありませんが、SQL で NULL データ値を CSV にダンプする処理を、`cursor.fetch*` 呼び出しによって返されたデータを前処理することなく簡単に行うことができます。浮動小数点数は、書き出される前に `repr()` を使って文字列に変換されます。他の非文字列データは、書き出される前に `str()` を使って文字列に変換されます。

短い利用例:

```
import csv
with open('eggs.csv', 'wb') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                            quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

`csv.register_dialect(name[, dialect], **fmtparams)`

`dialect` を `name` と関連付けます。`name` は文字列か Unicode オブジェクトでなければなりません。表現形式 (dialect) は `Dialect` のサブクラスを渡すか、またはキーワード引数 `fmtparams`、もしくは両方で指定できますが、キーワード引数の方が優先されます。表現形式と書式化パラメタについての詳細は、[Dialect クラスと書式化パラメタ](#) 節を参照してください。

`csv.unregister_dialect(name)`

`name` に関連づけられた表現形式を表現形式レジストリから削除します。`name` が表現形式名でない場合には `Error` を送出します。

`csv.get_dialect(name)`

`name` に関連づけられた表現形式を返します。`name` が表現形式名でない場合には `Error` を送出します。

バージョン 2.5 で変更: この関数は `immutable` な `Dialect` クラスを返します。以前は、要求された `dialect` のインスタンスが返されていました。ユーザーはクラスを操作することで、アクティブな reader や writer の動作を変更することができます。

`csv.list_dialects()`

登録されている全ての表現形式を返します。

`csv.field_size_limit([new_limit])`

パーサが許容する現在の最大フィールドサイズを返します。`new_limit` が渡されたときは、その値が新しい上限になります。

バージョン 2.5 で追加.

`csv` モジュールでは以下のクラスを定義しています:

```
class csv.DictReader(f, fieldnames=None, restkey=None, restval=None, dialect='excel', *args,
                    **kws)
```

Create an object which operates like a regular reader but maps the information read into a dict whose keys are given by the optional *fieldnames* parameter. The *fieldnames* parameter is a *sequence* whose elements are associated with the fields of the input data in order. These elements become the keys of the resulting dictionary. If the *fieldnames* parameter is omitted, the values in the first row of the file *f* will be used as the fieldnames. If the row read has more fields than the fieldnames sequence, the remaining data is added as a sequence keyed by the value of *restkey*. If the row read has fewer fields than the fieldnames sequence, the remaining keys take the value of the optional *restval* parameter. Any other optional or keyword arguments are passed to the underlying *reader* instance.

短い利用例:

```
>>> import csv
>>> with open('names.csv') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Baked Beans
Lovely Spam
Wonderful Spam
```

class `csv.DictWriter` (*f*, *fieldnames*, *restval*='', *extrasaction*='raise', *dialect*='excel', **args*, ***kws*)

Create an object which operates like a regular writer but maps dictionaries onto output rows. The *fieldnames* parameter is a *sequence* of keys that identify the order in which values in the dictionary passed to the *writerow()* method are written to the file *f*. The optional *restval* parameter specifies the value to be written if the dictionary is missing a key in *fieldnames*. If the dictionary passed to the *writerow()* method contains a key not found in *fieldnames*, the optional *extrasaction* parameter indicates what action to take. If it is set to 'raise' a *ValueError* is raised. If it is set to 'ignore', extra values in the dictionary are ignored. Any other optional or keyword arguments are passed to the underlying *writer* instance.

Note that unlike the *DictReader* class, the *fieldnames* parameter of the *DictWriter* is not optional. Since Python's *dict* objects are not ordered, there is not enough information available to deduce the order in which the row should be written to the file *f*.

短い利用例:

```
import csv

with open('names.csv', 'w') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

class `csv.Dialect`

Dialect クラスはコンテナクラスで、基本的な用途としては、その属性を特定の *reader* や *writer* インスタンスのパラメタを定義するために用います。

class `csv.excel`

excel クラスは Excel で生成される CSV ファイルの通常のプロパティを定義します。これは 'excel' という名前の dialect として登録されています。

class `csv.excel_tab`

excel_tab クラスは Excel で生成されるタブ分割ファイルの通常のプロパティを定義します。これは 'excel-tab' という名前の dialect として登録されています。

class `csv.Sniffer`

Sniffer クラスは CSV ファイルの書式を推理するために用いられるクラスです。

Sniffer クラスではメソッドを二つ提供しています:

sniff (*sample*, *delimiters*=None)

与えられた *sample* を解析し、発見されたパラメタを反映した *Dialect* サブクラスを返します。オプションの *delimiters* パラメタを与えた場合、有効なデリミタ文字を含んでいるはずの文字列として解釈されます。

has_header (*sample*)

(CSV 形式と仮定される) サンプルテキストを解析して、最初の行がカラムヘッダの羅列のように推察される場合 *True* を返します。

Sniffer の利用例:

```
with open('example.csv', 'rb') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... process CSV file contents here ...
```

`csv` モジュールでは以下の定数を定義しています:

csv.QUOTE_ALL

writer オブジェクトに対し、全てのフィールドをクオートするように指示します。

csv.QUOTE_MINIMAL

writer オブジェクトに対し、*delimiter*、*quotechar* または *lineterminator* に含まれる任意の文字のような特別な文字を含むフィールドだけをクオートするように指示します。

csv.QUOTE_NONNUMERIC

writer オブジェクトに対し、全ての非数値フィールドをクオートするように指示します。

reader に対しては、クオートされていない全てのフィールドを *float* 型に変換するよう指示します。

csv.QUOTE_NONE

writer オブジェクトに対し、フィールドを決してクオートしないように指示します。現在の *delimiter* が出力データ中に現れた場合、現在設定されている *escapechar* 文字が前に付けられます。 *escapechar* がセットされていない場合、エスケープが必要な文字に遭遇した *writer* は *Error* を送出します。

`reader` に対しては、クオート文字の特別扱いをしないように指示します。

`csv` モジュールでは以下の例外を定義しています:

exception `csv.Error`

全ての関数において、エラーが検出された際に送出される例外です。

13.1.2 Dialect クラスと書式化パラメタ

レコードに対する入出力形式の指定をより簡単にするために、特定の書式化パラメタは表現形式 (dialect) にまとめてグループ化されます。表現形式は `Dialect` クラスのサブクラスで、様々なクラス特有のメソッドと、`validate()` メソッドを一つ持っています。 `reader` または `writer` オブジェクトを生成するとき、プログラマは文字列または `Dialect` クラスのサブクラスを表現形式パラメタとして渡さなければなりません。さらに、 `dialect` パラメタの代りに、プログラマは上で定義されている属性と同じ名前を持つ個々の書式化パラメタを `Dialect` クラスに指定することができます。

Dialect は以下の属性をサポートしています:

Dialect.delimiter

フィールド間を分割するのに用いられる 1 文字からなる文字列です。デフォルトでは `,` です。

Dialect.doublequote

フィールド内に現れた `quotechar` のインスタンスで、クオートではないその文字自身でなければならない文字をどのようにクオートするかを制御します。 `True` の場合、この文字は二重化されます。 `False` の場合、 `escapechar` は `quotechar` の前に置かれます。デフォルトでは `True` です。

出力においては、 `doublequote` が `False` で `escapechar` がセットされていない場合、フィールド内に `quotechar` が現れると `Error` が送出されます。

Dialect.escapechar

`writer` が、 `quoting` が `QUOTE_NONE` に設定されている場合に `delimiter` をエスケープするため、および、 `doublequote` が `False` の場合に `quotechar` をエスケープするために用いられる、1 文字からなる文字列です。読み込み時には `escapechar` はそれに引き続く文字の特別な意味を取り除きます。デフォルトでは `None` で、エスケープを行いません。

Dialect.lineterminator

`writer` が作り出す各行を終端する際に用いられる文字列です。デフォルトでは `'\r\n'` です。

注釈: `reader` は `'\r'` または `'\n'` のどちらかを行末と認識するようにハードコードされており、 `lineterminator` を無視します。この振る舞いは将来変更されるかもしれません。

Dialect.quotechar

`delimiter` や `quotechar` といった特殊文字を含むか、改行文字を含むフィールドをクオートする際に用いられる 1 文字からなる文字列です。デフォルトでは `'` です。

Dialect.quoting

クオートがいつ writer によって生成されるか、また reader によって認識されるかを制御します。`QUOTE_*` 定数のいずれか (モジュールコンテンツ 節参照) をとることができ、デフォルトでは `QUOTE_MINIMAL` です。

`Dialect.skipinitialspace`

`True` の場合、`delimiter` の直後に続く空白は無視されます。デフォルトでは `False` です。

`Dialect.strict`

`True` の場合、不正な CSV 入力に対して `Error` を送出します。デフォルトでは `False` です。

13.1.3 reader オブジェクト

reader オブジェクト (`DictReader` インスタンス、および `reader()` 関数によって返されたオブジェクト) は、以下の public なメソッドを持っています:

`csvreader.next()`

reader の反復可能なオブジェクトから、現在の表現形式に基づいて次の行を解析して返します。

reader オブジェクトには以下の公開属性があります:

`csvreader.dialect`

パーサで使われる表現形式の読み取り専用の記述です。

`csvreader.line_num`

ソースイテレータから読んだ行数です。この数は返されるレコードの数とは、レコードが複数行に亘ることがあるので、一致しません。

バージョン 2.5 で追加。

`DictReader` オブジェクトは、以下の public な属性を持っています:

`csvreader.fieldnames`

オブジェクトを生成するときに渡されなかった場合、この属性は最初のアクセス時か、ファイルから最初のレコードを読み出したときに初期化されます。

バージョン 2.6 で変更。

13.1.4 writer オブジェクト

Writer オブジェクト (`DictWriter` インスタンス、および `writer()` 関数によって返されたオブジェクト) は、以下の public なメソッドを持っています: `row` には、Writer オブジェクトの場合には文字列か数値のシーケンスを指定し、`DictWriter` オブジェクトの場合はフィールド名をキーとして対応する文字列か数値を格納した辞書オブジェクトを指定します (数値は `str()` で変換されます)。複素数を出力する場合、値をかっこで囲んで出力します。このため、CSV ファイルを読み込むアプリケーションで (そのアプリケーションが複素数をサポートしていたとしても) 問題が発生する場合があります。

`csvwriter.writerow(row)`

`row` パラメタを現在の表現形式に基づいて書式化し、writer のファイルオブジェクトに書き込みます。

`csvwriter.writerows(rows)`

`rows` 引数 (上で解説した `row` オブジェクトのイテラブル) の全ての要素を現在の表現形式に基づいて書式化し、`writer` のファイルオブジェクトに書き込みます。

`writer` オブジェクトには以下の公開属性があります:

`csvwriter.dialect`

`writer` で使われる表現形式の読み取り専用の記述です。

`DictWriter` のオブジェクトは以下の `public` メソッドを持っています:

`DictWriter.writeheader()`

(コンストラクタで指定された) フィールド名の行を出力します。

バージョン 2.7 で追加.

13.1.5 使用例

最も簡単な CSV ファイル読み込みの例です:

```
import csv
with open('some.csv', 'rb') as f:
    reader = csv.reader(f)
    for row in reader:
        print row
```

別の書式での読み込み:

```
import csv
with open('passwd', 'rb') as f:
    reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE_NONE)
    for row in reader:
        print row
```

上に対して、単純な書き込みのプログラム例は以下のようになります。

```
import csv
with open('some.csv', 'wb') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

新しい表現形式の登録:

```
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
with open('passwd', 'rb') as f:
    reader = csv.reader(f, 'unixpwd')
```

もう少し手の込んだ `reader` の使い方 — エラーを捉えてレポートします。

```
import csv, sys
filename = 'some.csv'
with open(filename, 'rb') as f:
    reader = csv.reader(f)
    try:
        for row in reader:
            print row
    except csv.Error as e:
        sys.exit('file %s, line %d: %s' % (filename, reader.line_num, e))
```

このモジュールは文字列の解析は直接サポートしませんが、簡単にできます。

```
import csv
for row in csv.reader(['one,two,three']):
    print row
```

`csv` モジュールは直接は Unicode の読み書きをサポートしませんが、ASCII NUL 文字に関わる問題のために 8 ビットクリーンに書き込みます。ですから、NUL を使う UTF-16 のようなエンコーディングを避ける限り エンコード・デコードを行なう関数やクラスを書くことができます。UTF-8 がお勧めです。

以下の `unicode_csv_reader()` は Unicode の CSV データ (Unicode 文字列のリスト) を扱うための `csv.reader` をラップするジェネレータ (*generator*) です。 `utf_8_encoder()` は一度に 1 文字列 (または行) ずつ Unicode 文字列を UTF-8 としてエンコードするジェネレータです。エンコードされた文字列は CSV reader により分解され、 `unicode_csv_reader()` が UTF-8 エンコードの分解された文字列をデコードして Unicode に戻します。

```
import csv

def unicode_csv_reader(unicode_csv_data, dialect=csv.excel, **kwargs):
    # csv.py doesn't do Unicode; encode temporarily as UTF-8:
    csv_reader = csv.reader(utf_8_encoder(unicode_csv_data),
                             dialect=dialect, **kwargs)
    for row in csv_reader:
        # decode UTF-8 back to Unicode, cell by cell:
        yield [unicode(cell, 'utf-8') for cell in row]

def utf_8_encoder(unicode_csv_data):
    for line in unicode_csv_data:
        yield line.encode('utf-8')
```

その他のエンコーディングには以下の `UnicodeReader` クラスと `UnicodeWriter` クラスが使えます。二つのクラスは `encoding` パラメータをコンストラクタで取り、本物の reader や writer に渡されるデータが UTF-8 でエンコードされていることを保証します。

```
import csv, codecs, cStringIO

class UTF8Recoder:
    """
    Iterator that reads an encoded stream and reencodes the input to UTF-8
```

(次のページに続く)

(前のページからの続き)

```

"""
def __init__(self, f, encoding):
    self.reader = codecs.getreader(encoding)(f)

def __iter__(self):
    return self

def next(self):
    return self.reader.next().encode("utf-8")

class UnicodeReader:
    """
    A CSV reader which will iterate over lines in the CSV file "f",
    which is encoded in the given encoding.
    """

    def __init__(self, f, dialect=csv.excel, encoding="utf-8", **kwds):
        f = UTF8Recoder(f, encoding)
        self.reader = csv.reader(f, dialect=dialect, **kwds)

    def next(self):
        row = self.reader.next()
        return [unicode(s, "utf-8") for s in row]

    def __iter__(self):
        return self

class UnicodeWriter:
    """
    A CSV writer which will write rows to CSV file "f",
    which is encoded in the given encoding.
    """

    def __init__(self, f, dialect=csv.excel, encoding="utf-8", **kwds):
        # Redirect output to a queue
        self.queue = cStringIO.StringIO()
        self.writer = csv.writer(self.queue, dialect=dialect, **kwds)
        self.stream = f
        self.encoder = codecs.getincrementalencoder(encoding)()

    def writerow(self, row):
        self.writer.writerow([s.encode("utf-8") for s in row])
        # Fetch UTF-8 output from the queue ...
        data = self.queue.getvalue()
        data = data.decode("utf-8")
        # ... and reencode it into the target encoding
        data = self.encoder.encode(data)
        # write to the target stream
        self.stream.write(data)
        # empty queue
        self.queue.truncate(0)

```

(次のページに続く)

(前のページからの続き)

```
def writerows(self, rows):
    for row in rows:
        self.writerow(row)
```

13.2 ConfigParser — 設定ファイルの構文解析器

注釈: `ConfigParser` モジュールは Python 3 で `configparser` に改名されました。 `2to3` ツールが自動的にソース内の `import` を修正します。

This module defines the class `ConfigParser`. The `ConfigParser` class implements a basic configuration file parser language which provides a structure similar to what you would find on Microsoft Windows INI files. You can use this to write Python programs which can be customized by end users easily.

注釈: このライブラリでは、Windows のレジストリ用に拡張された INI 文法はサポートしていません。

参考:

`shlex` モジュール アプリケーション設定ファイルのフォーマットとして使える、Unix シェルに似たミニ言語の作成を支援します。

`json` モジュール `json` モジュールは、同じ目的に利用できる JavaScript の文法のサブセットを実装しています。

設定ファイルは 1 つ以上のセクションからなり、セクションは `[section]` ヘッダとそれに続く **RFC 822** 形式の `name: value` エントリからなっています。(section 3.1.1 "LONG HEADER FIELDS" を参照) `name=value` という形式も使えます。値の先頭にある空白文字は削除されるので注意してください。オプションの値には、同じセクションか `DEFAULT` セクションにある値を参照するような書式化文字列を含めることができます。初期化時や検索時に別のデフォルト値を与えることもできます。 `'#'` か `;` で始まる行は無視され、コメントを書くために利用できます。

設定ファイルは、特定の文字 (`#` と `;`) で始まるコメントを含んでいることがあります。コメントはある行の中に単独で書かれる場合もありますし、値やセクション名のある行に書かれる場合もあります。後者がコメントとして認識されるためには、その前に空白文字が入っている必要があります。(後方互換性のために、`#` ではなく `;` で一行コメントを書いてください。)

最も高機能なクラス `SafeConfigParser` は置換機能をサポートします。これは同じセクション内の値や `DEFAULT` という特別なセクションの値を参照するフォーマット文字列を使うことができるという意味です。さらに初期化のときにデフォルトの値を追加することもできます。

例えば:

```
[My Section]
foodir: %(dir)s/whatever
dir=frob
long: this value continues
    in the next line
```

この場合 `%(dir)s` は変数 `dir` (この場合は `frob`) に展開されます。参照の展開は必要に応じて実行されます。

Default values can be specified by passing them into the `ConfigParser` constructor as a dictionary. Additional defaults may be passed into the `get()` method which will override all others.

Sections are normally stored in a built-in dictionary. An alternative dictionary type can be passed to the `ConfigParser` constructor. For example, if a dictionary type is passed that sorts its keys, the sections will be sorted on write-back, as will be the keys within each section.

class `ConfigParser.RawConfigParser` (`[defaults[, dict_type[, allow_no_value]]]`)

基本的な設定オブジェクトです。 `defaults` が与えられた場合、オブジェクトに固有のデフォルト値がその値で初期化されます。 `dict_type` が与えられた場合、それが、セクションのリストの格納、セクション内のオプションの格納、デフォルト値のために利用されます。 `allow_no_value` (デフォルト: `False`) が真の時、値のないオプションが許可されます。この場合の値は `None` になります。

このクラスは値の置換をサポートしません。

全てのオプション名が `optionxform()` メソッドに渡されます。このメソッドのデフォルトの実装では、オプション名を小文字に変換します。

バージョン 2.3 で追加。

バージョン 2.6 で変更: `dict_type` が追加されました。

バージョン 2.7 で変更: `dict_type` のデフォルトが `collections.OrderedDict` になりました。
`allow_no_value` が追加されました。

class `ConfigParser.ConfigParser` (`[defaults[, dict_type[, allow_no_value]]]`)

`RawConfigParser` の派生クラスで値の置換を実装しており、`get()` メソッドと `items()` メソッドに省略可能な引数を追加しています。 `defaults` に含まれる値は `%()s` による値の置換に適切なものである必要があります。 `__name__` は組み込みのデフォルト値で、セクション名が含まれるので `defaults` で設定してもオーバーライドされます。

置換で使われるすべてのオプション名は、ほかのオプション名への参照と同様に `optionxform()` メソッドを介して渡されます。 `optionxform()` のデフォルト実装を使うと、値 `foo %(bar)s` および `foo %(BAR)s` は同一になります。

バージョン 2.3 で追加。

バージョン 2.6 で変更: `dict_type` が追加されました。

バージョン 2.7 で変更: `dict_type` のデフォルトが `collections.OrderedDict` になりました。
`allow_no_value` が追加されました。

class ConfigParser.SafeConfigParser ([defaults[, dict_type[, allow_no_value]]])

Derived class of *ConfigParser* that implements a more-sane variant of the magical interpolation feature. This implementation is more predictable as well. New applications should prefer this version if they don't need to be compatible with older versions of Python.

バージョン 2.3 で追加.

バージョン 2.6 で変更: *dict_type* が追加されました。

バージョン 2.7 で変更: *dict_type* のデフォルトが *collections.OrderedDict* になりました。
allow_no_value が追加されました。

exception ConfigParser.Error

他の全ての configparser の例外の基底クラスです。

exception ConfigParser.NoSectionError

指定したセクションが見つからなかった時に起きる例外です。

exception ConfigParser.DuplicateSectionError

すでに存在するセクション名に対して *add_section()* が呼び出された際に起きる例外です。

exception ConfigParser.NoOptionError

指定したオプションが指定したセクションに存在しなかった時に起きる例外です。

exception ConfigParser.InterpolationError

文字列の置換中に問題が起きた時に発生する例外の基底クラスです。

exception ConfigParser.InterpolationDepthError

InterpolationError の派生クラスで、文字列の置換回数が *MAX_INTERPOLATION_DEPTH* を越えたために完了しなかった場合に発生する例外です。

exception ConfigParser.InterpolationMissingOptionError

InterpolationError の派生クラスで、値が参照しているオプションが見つからない場合に発生する例外です。

バージョン 2.3 で追加.

exception ConfigParser.InterpolationSyntaxError

InterpolationError の派生クラスで、指定された構文で値を置換することができなかった場合に発生する例外です。

バージョン 2.3 で追加.

exception ConfigParser.MissingSectionHeaderError

セクションヘッダを持たないファイルを構文解析しようとした時に起きる例外です。

exception ConfigParser.ParsingError

ファイルの構文解析中にエラーが起きた場合に発生する例外です。

ConfigParser.MAX_INTERPOLATION_DEPTH

The maximum depth for recursive interpolation for *get()* when the *raw* parameter is false. This is relevant only for the *ConfigParser* class.

参考:

`shlex` モジュール アプリケーション用の設定ファイルフォーマットとして使える、Unix シェルライクなミニ言語の作成を支援します。

13.2.1 RawConfigParser オブジェクト

`RawConfigParser` クラスのインスタンスは以下のメソッドを持ちます:

`RawConfigParser.defaults()`

インスタンス全体で使われるデフォルト値の辞書を返します。

`RawConfigParser.sections()`

利用可能なセクションのリストを返します。DEFAULT はこのリストに含まれません。

`RawConfigParser.add_section(section)`

`section` という名前のセクションをインスタンスに追加します。同名のセクションが存在した場合、`DuplicateSectionError` が発生します。DEFAULT (もしくは大文字小文字が違うもの) が渡された場合、`ValueError` が発生します。

`RawConfigParser.has_section(section)`

指定したセクションがコンフィグレーションファイルに存在するかを返します。DEFAULT セクションは存在するとみなされません。

`RawConfigParser.options(section)`

`section` で指定したセクションで利用できるオプションのリストを返します。

`RawConfigParser.has_option(section, option)`

与えられたセクションが存在してかつオプションが与えられていれば `True` を返し、そうでなければ `False` を返します。

バージョン 1.6 で追加。

`RawConfigParser.read(filenames)`

Attempt to read and parse a list of filenames, returning a list of filenames which were successfully parsed. If *filenames* is a string or Unicode string, it is treated as a single filename. If a file named in *filenames* cannot be opened, that file will be ignored. This is designed so that you can specify a list of potential configuration file locations (for example, the current directory, the user's home directory, and some system-wide directory), and all existing configuration files in the list will be read. If none of the named files exist, the `ConfigParser` instance will contain an empty dataset. An application which requires initial values to be loaded from a file should load the required file or files using `readfp()` before calling `read()` for any optional files:

```
import ConfigParser, os

config = ConfigParser.ConfigParser()
config.readfp(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')])
```

バージョン 2.4 で変更: 解析に成功したファイル名のリストを返します。

`RawConfigParser.readfp(fp[, filename])`

`fp` で与えられるファイルがファイルのようなオブジェクトを読み込んで構文解析します (`readline()` メソッドだけを使います)。もし `filename` が省略されて `fp` が `name` 属性を持っていれば `filename` の代わりに使われます。ファイル名の初期値は `<??>` です。

`RawConfigParser.get(section, option)`

`section` の `option` 変数を取得します。

`RawConfigParser.getint(section, option)`

`section` の `option` を整数として評価する関数です。

`RawConfigParser.getfloat(section, option)`

`section` の `option` を浮動小数点数として評価する関数です。

`RawConfigParser.getboolean(section, option)`

指定した `section` の `option` 値をブール値に型強制する便宜メソッドです。 `option` として受理できる値は、真 (True) としては "1"、"yes"、"true"、"on"、偽 (False) としては "0"、"no"、"false"、"off" です。これらの文字列値に対しては大文字小文字の区別をしません。その他の値の場合には `ValueError` を送出します。

`RawConfigParser.items(section)`

与えられた `section` のそれぞれのオプションについて (name, value) ペアのリストを返します。

`RawConfigParser.set(section, option, value)`

If the given section exists, set the given option to the specified value; otherwise raise `NoSectionError`. While it is possible to use `RawConfigParser` (or `ConfigParser` with `raw` parameters set to true) for internal storage of non-string values, full functionality (including interpolation and output to files) can only be achieved using string values.

バージョン 1.6 で追加.

`RawConfigParser.write(fileobject)`

設定を文字列表現に変換してファイルオブジェクトに書き出します。この文字列表現は `read()` で読み込むことができます。

バージョン 1.6 で追加.

`RawConfigParser.remove_option(section, option)`

指定された `section` から指定された `option` を削除します。セクションが存在しなければ、`NoSectionError` を起こします。存在するオプションを削除した時は `True` を、そうでない時は `False` を返します。

バージョン 1.6 で追加.

`RawConfigParser.remove_section(section)`

指定された `section` を設定から削除します。もし指定されたセクションが存在すれば `True`、そうでなければ `False` を返します。

`RawConfigParser.optionxform(option)`

入力ファイル中に見つかったオプション名か、クライアントコードから渡されたオプション名 *option* を、内部で利用する形式に変換します。デフォルトでは *option* を全て小文字に変換した名前が返されます。サブクラスではこの関数をオーバーライドすることでこの振舞いを替えることができます。

振舞いを替えるために `ConfigParser` を継承して新たにクラスを作る必要はありません、あるインスタンスのメソッドを文字列を引数に取る関数で置き換えることもできます。たとえば、このメソッドを `str()` に設定することで大小文字の差を区別するように変更することができます:

```
cfgparser = ConfigParser()
...
cfgparser.optionxform = str
```

設定ファイルを読み込むときには、`optionxform()` が呼ばれる前にオプション名の前後の空白文字が取り除かれることに注意してください。

13.2.2 ConfigParser オブジェクト

The `ConfigParser` class extends some methods of the `RawConfigParser` interface, adding some optional arguments.

`ConfigParser.get(section, option[, raw[, vars]])`

section の *option* 変数を取得します。このメソッドに渡される *vars* は辞書でなくてはいけません。(もし渡されているならば) *vars*、*section*、*defaults* の順に *option* が探されます。

raw が真でない時には、全ての '%' 置換は展開されてから返されます。置換後の値はオプションと同じ順序で探されます。

`ConfigParser.items(section[, raw[, vars]])`

指定した *section* 内の各オプションに対して、(name, value) のペアからなるリストを返します。省略可能な引数は `get()` メソッドと同じ意味を持ちます。

バージョン 2.3 で追加。

13.2.3 SafeConfigParser オブジェクト

The `SafeConfigParser` class implements the same extended interface as `ConfigParser`, with the following addition:

`SafeConfigParser.set(section, option, value)`

もし与えられたセクションが存在している場合は、指定された値を与えられたオプションに設定します。そうでない場合は `NoSectionError` を発生させます。 *value* は文字列 (`str` または `unicode`) でなければならず、そうでない場合には `TypeError` が発生します。

バージョン 2.4 で追加。

13.2.4 使用例

設定ファイルを書き出す例:

```
import ConfigParser

config = ConfigParser.RawConfigParser()

# When adding sections or items, add them in the reverse order of
# how you want them to be displayed in the actual file.
# In addition, please note that using RawConfigParser's and the raw
# mode of ConfigParser's respective set functions, you can assign
# non-string values to keys internally, but will receive an error
# when attempting to write to a file or when you get it in non-raw
# mode. SafeConfigParser does not allow such assignments to take place.
config.add_section('Section1')
config.set('Section1', 'an_int', '15')
config.set('Section1', 'a_bool', 'true')
config.set('Section1', 'a_float', '3.1415')
config.set('Section1', 'baz', 'fun')
config.set('Section1', 'bar', 'Python')
config.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# Writing our configuration file to 'example.cfg'
with open('example.cfg', 'wb') as configfile:
    config.write(configfile)
```

設定ファイルを読み込む例:

```
import ConfigParser

config = ConfigParser.RawConfigParser()
config.read('example.cfg')

# getfloat() raises an exception if the value is not a float
# getint() and getboolean() also do this for their respective types
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print a_float + an_int

# Notice that the next output does not interpolate '%(bar)s' or '%(baz)s'.
# This is because we are using a RawConfigParser().
if config.getboolean('Section1', 'a_bool'):
    print config.get('Section1', 'foo')
```

To get interpolation, you will need to use a *ConfigParser* or *SafeConfigParser*:

```
import ConfigParser

config = ConfigParser.ConfigParser()
config.read('example.cfg')
```

(次のページに続く)

(前のページからの続き)

```
# Set the third, optional argument of get to 1 if you wish to use raw mode.
print config.get('Section1', 'foo', 0) # -> "Python is fun!"
print config.get('Section1', 'foo', 1) # -> "%(bar)s is %(baz)s!"

# The optional fourth argument is a dict with members that will take
# precedence in interpolation.
print config.get('Section1', 'foo', 0, {'bar': 'Documentation',
                                         'baz': 'evil'})
```

3 種類全ての ConfigParser クラスで、デフォルト値を利用できます。別にオプションが指定されていなかった場合、このデフォルト値は置換機能でも利用されます:

```
import ConfigParser

# New instance with 'bar' and 'baz' defaulting to 'Life' and 'hard' each
config = ConfigParser.SafeConfigParser({'bar': 'Life', 'baz': 'hard'})
config.read('example.cfg')

print config.get('Section1', 'foo') # -> "Python is fun!"
config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print config.get('Section1', 'foo') # -> "Life is hard!"
```

opt_move 関数は、オプションをセクション間で移動することができます:

```
def opt_move(config, section1, section2, option):
    try:
        config.set(section2, option, config.get(section1, option, 1))
    except ConfigParser.NoSectionError:
        # Create non-existent section
        config.add_section(section2)
        opt_move(config, section1, section2, option)
    else:
        config.remove_option(section1, option)
```

いくつかの設定ファイルでは、値のない設定項目がある以外は *ConfigParser* の文法と同じ文法になっています。コンストラクタの *allow_no_value* 引数で、そのような値を許可することができます。

```
>>> import ConfigParser
>>> import io

>>> sample_config = """
... [mysqld]
... user = mysql
... pid-file = /var/run/mysqld/mysqld.pid
... skip-external-locking
... old_passwords = 1
... skip-bdb
... skip-innodb
... """
```

(次のページに続く)

(前のページからの続き)

```

>>> config = ConfigParser.RawConfigParser(allow_no_value=True)
>>> config.readfp(io.BytesIO(sample_config))

>>> # Settings with values are treated as before:
>>> config.get("mysqld", "user")
'mysql'

>>> # Settings without values provide None:
>>> config.get("mysqld", "skip-bdb")

>>> # Settings which aren't specified still raise an error:
>>> config.get("mysqld", "does-not-exist")
Traceback (most recent call last):
...
ConfigParser.NoOptionError: No option 'does-not-exist' in section: 'mysqld'

```

13.3 robotparser — robots.txt のためのパーザ

注釈: `robotparser` モジュールは、Python 3 では `urllib.robotparser` にリネームされました。 [2to3](#) ツールが自動的にソースコードの `import` を修正します。

このモジュールでは単一のクラス、`RobotFileParser` を提供します。このクラスは、特定のユーザーエージェントが `robots.txt` ファイルを公開している Web サイトのある URL を取得可能かどうかの質問に答えます。`robots.txt` ファイルの構造に関する詳細は <http://www.robotstxt.org/orig.html> を参照してください。

class `robotparser.RobotFileParser` (`url=""`)

`url` の `robots.txt` に対し読み込み、パース、応答するメソッドを提供します。

set_url (`url`)

`robots.txt` ファイルを参照するための URL を設定します。

read ()

`robots.txt` URL を読み出し、パーザに入力します。

parse (`lines`)

引数 `lines` の内容を解釈します。

can_fetch (`useragent, url`)

解釈された `robots.txt` ファイル中に記載された規則に従ったとき、`useragent` が `url` を取得してもよい場合には `True` を返します。

mtime ()

`robots.txt` ファイルを最後に取得した時刻を返します。この値は、定期的に新たな `robots.txt` をチェックする必要がある、長時間動作する Web スパイダープログラムを実装する際に便利

です。

modified()

robots.txt ファイルを最後に取得した時刻を現在の時刻に設定します。

以下に RobotFileParser クラスの利用例を示します。

```
>>> import robotparser
>>> rp = robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```

13.4 netrc — netrc ファイルの処理

バージョン 1.5.2 で追加。

ソースコード: [Lib/netrc.py](#)

netrc クラスは、Unix **ftp** プログラムや他の FTP クライアントで用いられる netrc ファイル形式を解析し、カプセル化 (encapsulate) します。

class netrc.netrc (*[file]*)

netrc のインスタンスやサブクラスのインスタンスは netrc ファイルのデータをカプセル化します。初期化の際の引数が存在する場合、解析対象となるファイルの指定になります。引数がない場合、ユーザのホームディレクトリ下にある *.netrc* が読み出されます。解析エラーが発生した場合、ファイル名、行番号、解析を中断したトークンに関する情報の入った *NetrcParseError* を送出します。POSIX システムにおいて引数を指定しない場合、ファイルのオーナーシップやパーミッションが安全でない (プロセスを実行しているユーザ以外が所有者であるか、誰にでも読み書き出来てしまう) のに *.netrc* ファイル内にパスワードが含まれていると、*NetrcParseError* を送出します。このセキュリティ的な振る舞いは、ftp などの *.netrc* を使うプログラムと同じものです。

バージョン 2.7.6 で変更: POSIX パーミッションのチェックが追加されました。

exception netrc.NetrcParseError

ソースファイルのテキスト中で文法エラーに遭遇した場合に *netrc* クラスによって送出される例外です。この例外のインスタンスは 3 つのインスタンス変数を持っています: *msg* はテキストによるエラーの説明、*filename* はソースファイルの名前、そして *lineno* はエラーが発見された行番号です。

13.4.1 netrc オブジェクト

netrc インスタンスは以下のメソッドを持っています:

`netrc.authenticators(host)`

`host` の認証情報として、三要素のタプル (`login`, `account`, `password`) を返します。与えられた `host` に対するエントリが `netrc` ファイルにない場合、`'default'` エントリに関連付けられたタプルが返されます。`host` に対応するエントリがなく、`default` エントリもない場合、`None` を返します。

`netrc.__repr__()`

クラスの持っているデータを `netrc` ファイルの書式に従った文字列で出力します。(コメントは無視され、エントリが並べ替えられる可能性があります。)

`netrc` のインスタンスは以下の public なインスタンス変数を持っています:

`netrc.hosts`

ホスト名を (`login`, `account`, `password`) からなるタプルに対応づけている辞書です。`'default'` エントリがある場合、その名前の擬似ホスト名として表現されます。

`netrc.macros`

マクロ名を文字列のリストに対応付けている辞書です。

注釈: 利用可能なパスワードの文字セットは、ASCII のサブセットのみです。2.3 より前のバージョンでは厳しく制限されていましたが、2.3 以降では全ての ASCII の記号を使用することができます。しかし、空白文字と印刷不可文字を使用することはできません。この制限は `.netrc` ファイルの解析方法によるものであり、将来解除されます。

13.5 xdrlib — XDR データのエンコードおよびデコード

ソースコード: [Lib/xdrlib.py](#)

`xdrlib` モジュールは外部データ表現標準 (External Data Representation Standard) のサポートを実現します。この標準は 1987 年に Sun Microsystems, Inc. によって書かれ、[RFC 1014](#) で定義されています。このモジュールでは RFC で記述されているほとんどのデータ型をサポートしています。

`xdrlib` モジュールでは 2 つのクラスが定義されています。一つは変数を XDR 表現にパックするためのクラスで、もう一方は XDR 表現からアンパックするためのものです。2 つの例外クラスが同様に定義されています。

class `xdrlib.Packer`

`Packer` はデータを XDR 表現にパックするためのクラスです。`Packer` クラスのインスタンス生成は引数なしで行われます。

class `xdrlib.Unpacker(data)`

`Unpacker` は `Packer` と対をなして、文字列バッファから XDR をアンパックするためのクラスです。入力バッファ `data` を引数に与えてインスタンスを生成します。

参考:

RFC 1014 - XDR: External Data Representation Standard この RFC が、かつてこのモジュールが最初に書かれた当時に XDR 標準であったデータのエンコード方法を定義していました。現在は **RFC 1832** に更新されているようです。

RFC 1832 - XDR: External Data Representation Standard こちらが新しい方の RFC で、XDR の改訂版が定義されています。

13.5.1 Packer オブジェクト

Packer インスタンスには以下のメソッドがあります:

`Packer.get_buffer()`

現在のパック処理用バッファを文字列で返します。

`Packer.reset()`

パック処理用バッファをリセットして、空文字にします。

一般的には、適切な `pack_type()` メソッドを使えば、一般に用いられているほとんどの XDR データをパックすることができます。各々のメソッドは一つの引数を取り、パックしたい値を与えます。単純なデータ型をパックするメソッドとして、以下のメソッド: `pack_uint()`、`pack_int()`、`pack_enum()`、`pack_bool()`、`pack_uhyper()` そして `pack_hyper()` がサポートされています。

`Packer.pack_float(value)`

単精度 (single-precision) の浮動小数点数 *value* をパックします。

`Packer.pack_double(value)`

倍精度 (double-precision) の浮動小数点数 *value* をパックします。

以下のメソッドは文字列、バイト列、不透明データ (opaque data) のパック処理をサポートします:

`Packer.pack_fstring(n, s)`

固定長の文字列、*s* をパックします。*n* は文字列の長さですが、この値自体はデータバッファにはパックされません。4 バイトのアラインメントを保証するために、文字列は必要に応じて null バイト列でパディングされます。

`Packer.pack_fopaque(n, data)`

`pack_fstring()` と同じく、固定長の不透明データストリームをパックします。

`Packer.pack_string(s)`

可変長の文字列 *s* をパックします。文字列の長さが最初に符号なし整数でパックされ、続いて `pack_fstring()` を使って文字列データがパックされます。

`Packer.pack_opaque(data)`

`pack_string()` と同じく、可変長の不透明データ文字列をパックします。

`Packer.pack_bytes(bytes)`

`pack_string()` と同じく、可変長のバイトストリームをパックします。

以下のメソッドはアレイやリストのパック処理をサポートします:

`Packer.pack_list(list, pack_item)`

一様な項目からなる *list* をパックします。このメソッドはサイズ不定、すなわち、全てのリスト内容を網羅するまでサイズが分からないリストに対して有用です。リストのすべての項目に対し、最初に符号無し整数 1 がパックされ、続いてリスト中のデータがパックされます。*pack_item* は個々の項目をパックするために呼び出される関数です。リストの末端に到達すると、符号無し整数 0 がパックされます。

例えば、整数のリストをパックするには、コードは以下になるはずです：

```
import xdrlib
p = xdrlib.Packer()
p.pack_list([1, 2, 3], p.pack_int)
```

`Packer.pack_farray(n, array, pack_item)`

一様な項目からなる固定長のリスト (*array*) をパックします。*n* はリストの長さです。この値はデータバッファにパック されませんが、`len(array)` が *n* と等しくない場合、例外 `ValueError` が送出されます。上と同様に、*pack_item* は個々の要素をパック処理するための関数です。

`Packer.pack_array(list, pack_item)`

一様な項目からなる可変長の *list* をパックします。まず、リストの長さが符号無し整数でパックされ、つづいて各要素が上の `pack_farray()` と同じやり方でパックされます。

13.5.2 Unpacker オブジェクト

`Unpacker` クラスは以下のメソッドを提供します：

`Unpacker.reset(data)`

文字列バッファを *data* でリセットします。

`Unpacker.get_position()`

データバッファ中の現在のアンパック処理位置を返します。

`Unpacker.set_position(position)`

データバッファ中のアンパック処理位置を *position* に設定します。`get_position()` および `set_position()` は注意して使わなければなりません。

`Unpacker.get_buffer()`

現在のアンパック処理用データバッファを文字列で返します。

`Unpacker.done()`

アンパック処理を終了させます。全てのデータがまだアンパックされていなければ、例外 `Error` が送出されます。

上のメソッドに加えて、`Packer` でパック処理できるデータ型はいずれも `Unpacker` でアンパック処理できます。アンパック処理メソッドは `unpack_type()` の形式をとり、引数をとりません。これらのメソッドはアンパックされたデータオブジェクトを返します。

`Unpacker.unpack_float()`

単精度の浮動小数点数をアンパックします。

`Unpacker.unpack_double()`

`unpack_float()` と同様に、倍精度の浮動小数点数をアンパックします。

上のメソッドに加えて、文字列、バイト列、不透明データをアンパックする以下のメソッドが提供されています:

`Unpacker.unpack_fstring(n)`

固定長の文字列をアンパックして返します。*n* は予想される文字列の長さです。4 バイトのアラインメントを保証するために null バイトによるパディングが行われているものと仮定して処理を行います。

`Unpacker.unpack_fopaque(n)`

`unpack_fstring()` と同様に、固定長の不透明データストリームをアンパックして返します。

`Unpacker.unpack_string()`

可変長の文字列をアンパックして返します。最初に文字列の長さが符号無し整数としてアンパックされ、次に `unpack_fstring()` を使って文字列データがアンパックされます。

`Unpacker.unpack_opaque()`

`unpack_string()` と同様に、可変長の不透明データ文字列をアンパックして返します。

`Unpacker.unpack_bytes()`

`unpack_string()` と同様に、可変長のバイトストリームをアンパックして返します。

以下メソッドはアレイおよびリストのアンパック処理をサポートします:

`Unpacker.unpack_list(unpack_item)`

一様な項目からなるリストをアンパック処理してかえします。リストは一度に 1 要素ずつアンパック処理されます、まず符号無し整数によるフラグがアンパックされます。もしフラグが 1 なら、要素はアンパックされ、返り値のリストに追加されます。フラグが 0 であれば、リストの終端を示します。`unpack_item` は個々の項目をアンパック処理するために呼び出される関数です。

`Unpacker.unpack_farray(n, unpack_item)`

一様な項目からなる固定長のアレイをアンパックして (リストとして) 返します。*n* はバッファ内に存在すると期待されるリストの要素数です。上と同様に、`unpack_item` は各要素をアンパックするために使われる関数です。

`Unpacker.unpack_array(unpack_item)`

一様な項目からなる可変長の *list* をアンパックして返します。まず、リストの長さが符号無し整数としてアンパックされ、続いて各要素が上の `unpack_farray()` のようにしてアンパック処理されます。

13.5.3 例外

このモジュールでの例外はクラスインスタンスとしてコードされています:

exception `xdrlib.Error`

ベースとなる例外クラスです。 `Error` public な属性として `msg` を持ち、エラーの詳細が収められています。

exception `xdrlib.ConversionError`

Error から派生したクラスです。インスタンス変数は塚されていません。

これらの例外を補足する方法を以下の例に示します:

```
import xdrlib
p = xdrlib.Packer()
try:
    p.pack_double(8.01)
except xdrlib.ConversionError as instance:
    print 'packing the double failed:', instance.msg
```

13.6 plistlib — Mac OS X .plist ファイルの生成と解析

バージョン 2.6 で変更: このモジュールは以前は Mac 専用ライブラリだけにありましたが、全てのプラットフォームで使えるようにしました。

ソースコード: [Lib/plistlib.py](#)

このモジュールは主に Mac OS X で使われる「プロパティリスト」XML ファイルを読み書きするインターフェイスを提供します。

プロパティリスト (`.plist`) ファイル形式は基本的型のオブジェクト、たとえば辞書やリスト、数、文字列など、に対する単純な XML による保存形式です。たいてい、トップレベルのオブジェクトは辞書です。

値は文字列、整数、浮動小数点数、ブール型、タプル、リスト、辞書 (ただし文字列だけがキーになれます)、`Data` または `datetime.datetime` のオブジェクトです。文字列の値は (辞書のキーも含めて) Unicode 文字列であって構いません – それらは UTF-8 で書き出されます。

<data> plist 型は `Data` クラスを通じてサポートされます。これは Python の文字列に対する薄いラップです。文字列に制御文字を含めなければならない場合は `Data` を使って下さい。

参考:

[PList マニュアルページ](#) このファイル形式の Apple の文書。

このモジュールは以下の関数を定義しています:

`plistlib.readPlist (pathOrFile)`

plist ファイルを読み込みます。 `pathOrFile` はファイル名でも (読み込み可能な) ファイルオブジェクトでも構いません。展開されたルートオブジェクト (たいていは辞書です) を返します。

XML データは `xml.parsers.expat` にある Expat パーサを使って解析されます – 間違いのある XML に対して送られる可能性のある例外についてはそちらの文書を参照して下さい。未知の要素は plist 解析器から単純に無視されます。

`plistlib.writePlist (rootObject, pathOrFile)`

`rootObject` を plist ファイルに書き込みます。 `pathOrFile` はファイル名でも (書き込み可能な) ファイルオブジェクトでも構いません。

`TypeError` が、オブジェクトがサポート外の型のものであったりサポート外の型のオブジェクトを含むコンテナだった場合に、送出されます。

`plistlib.readPlistFromString (data)`

文字列から plist を読み取ります。ルートオブジェクトを返します。

`plistlib.writePlistToString (rootObject)`

`rootObject` を plist 形式の文字列として返します。

`plistlib.readPlistFromResource (path, restype='plst', resid=0)`

`path` のリソースフォークの中の `restype` タイプのリソースから plist を読み込みます。使用可能: Mac OS X。

注釈: Python 3.x でこの関数は削除されました。

`plistlib.writePlistToResource (rootObject, path, restype='plst', resid=0)`

`rootObject` を `path` のリソースフォークの中に `restype` タイプのリソースとして書き込みます。使用可能: Mac OS X。

注釈: Python 3.x でこの関数は削除されました。

以下のクラスが使用可能です:

class `plistlib.Data (data)`

文字列 `data` を包むラップオブジェクトを返します。plist 中に入れられる `<data>` 型を表すものとして plist への変換、plist からの変換関数で使われます。

これには `data` という一つの属性があり、そこに収められた Python 文字列を取り出すのに使えます。

13.6.1 例

plist を作ります:

```
p1 = dict(
    aString="Doodah",
    aList=["A", "B", 12, 32.1, [1, 2, 3]],
    aFloat = 0.1,
    anInt = 728,
    aDict=dict(
        anotherString="<hello & hi there!>",
        aUnicodeValue=u'M\xe4ssig, Ma\xdf',
        aTrueValue=True,
        aFalseValue=False,
    ),
    someData = Data("<binary gunk>"),
```

(次のページに続く)

(前のページからの続き)

```
someMoreData = Data("<lots of binary gunk>" * 10),
aDate = datetime.datetime.fromtimestamp(time.mktime(time.gmtime())),
)
# unicode keys are possible, but a little awkward to use:
pl[u'\xc5benraa'] = "That was a unicode key."
writePlist(pl, fileName)
```

plist を解析します:

```
pl = readPlist(pathOrFile)
print pl["aKey"]
```


第 14 章

暗号関連のサービス

この章で記述されているモジュールでは、暗号の本質に関わる様々なアルゴリズムを実装しています。これらは必要に応じてインストールすることで使えます。概要を以下に示します:

14.1 `hashlib` — セキュアハッシュおよびメッセージダイジェスト

バージョン 2.5 で追加.

ソースコード: [Lib/hashlib.py](#)

このモジュールは、セキュアハッシュやメッセージダイジェスト用のさまざまなアルゴリズムを実装したものです。FIPS のセキュアなハッシュアルゴリズムである SHA1、SHA224、SHA256、SHA384 および SHA512 (FIPS 180-2 で定義されているもの) だけでなく RSA の MD5 アルゴリズム (Internet [RFC 1321](#) で定義されています) も実装しています。「セキュアなハッシュ」と「メッセージダイジェスト」はどちらも同じ意味です。古くからあるアルゴリズムは「メッセージダイジェスト」と呼ばれていますが、最近は「セキュアハッシュ」という用語が用いられています。

注釈: `adler32` や `crc32` ハッシュ関数は `zlib` モジュールで提供されています。

警告: 幾つかのアルゴリズムはハッシュの衝突に弱いことが知られています。最後の “参考” セクションを見てください。

各 `hash` の名前が付いたコンストラクタがあります。いずれも同一で簡単なインターフェイスのあるハッシュオブジェクトを返します。例えば、SHA1 ハッシュオブジェクトを作るには `sha1()` を使います。このオブジェクトには `update()` を用いて任意の文字列を渡すことが出来ます。`digest()` や `hexdigest()` メソッドを用いて、それまでに渡したデータを連結したものの `digest` をいつでも要求することが出来ます。

このモジュールで常に使用できるハッシュアルゴリズムのコンストラクタは `md5()`、`sha1()`、`sha224()`、`sha256()`、`sha384()` および `sha512()` です。それ以外のアルゴリズムが使用できるかどうかは、Python が使用している OpenSSL ライブラリに依存します。

たとえば、'Nobody inspects the spammish repetition' というバイト文字列のダイジェストを取得するには次のようにします:

```
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
>>> m.digest_size
16
>>> m.block_size
64
```

もっと簡潔に書くと、このようになります:

```
>>> hashlib.sha224("Nobody inspects the spammish repetition").hexdigest()
'a4337bc45a8fc544c03f52dc550cd6e1e87021bc896588bd79e901e2'
```

一般的な `new()` コンストラクタで、第一引数にアルゴリズム名を文字列で受け取ります。他にも、上記ハッシュだけでなく OpenSSL ライブラリーが提供するような他のアルゴリズムにアクセスすることが出来ます。名前のあるコンストラクタの方が `new()` よりもずっと速いので望ましいです。

`new()` に OpenSSL のアルゴリズムを指定する例です:

```
>>> h = hashlib.new('ripemd160')
>>> h.update("Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

このモジュールは以下の定数属性を提供しています:

`hashlib.algorithms`

このモジュールによってサポートされていることが保証されるハッシュアルゴリズムの名前が入ったタプル。

バージョン 2.7 で追加.

`hashlib.algorithms_guaranteed`

このモジュールによってすべてのプラットフォームでサポートされていることが保証されるハッシュアルゴリズムの名前を含む集合です。

バージョン 2.7.9 で追加.

`hashlib.algorithms_available`

実行中の Python インタープリタで利用可能なハッシュアルゴリズム名の set です。これらの名前は `new()` に渡すことが出来ます。`algorithms_guaranteed` は常にサブセットです。この set の中に同じアルゴリズムが違う名前が複数回現れることがあります (OpenSSL 由来)。

バージョン 2.7.9 で追加.

コンストラクタが返すハッシュオブジェクトには、次のような定数属性が用意されています:

`hash.digest_size`

生成されたハッシュのバイト数。

`hash.block_size`

内部で使われるハッシュアルゴリズムのブロックのバイト数。

ハッシュオブジェクトには次のようなメソッドがあります:

`hash.update (arg)`

文字列 *arg* でハッシュオブジェクトを更新します。繰り返し呼び出すことは引数全ての連結で一回呼び出すことと等価です。例えば `m.update (a) ; m.update (b)` は `m.update (a+b)` と等価です。

バージョン 2.7 で変更: ハッシュアルゴリズムが OpenSSL によって提供されていて、データが 2048 バイトを超えている場合には、ハッシュの更新が実行中でも他のスレッドが実行出来るように、Python *GIL* が解放されます。

`hash.digest ()`

これまでに `update ()` メソッドに渡した文字列のダイジェストを返します。これは `digest_size` バイトの文字列であり、非 ASCII 文字や null バイトを含むこともあります。

`hash.hexdigest ()`

`digest ()` と似ていますが、倍の長さの、16 進形式文字列を返します。これは、電子メールなどの非バイナリ環境で値を交換する場合に便利です。

`hash.copy ()`

ハッシュオブジェクトのコピー (“クローン”) を返します。これは、共通部分を持つ複数の文字列のダイジェストを効率的に計算するために使用します。

14.1.1 鍵導出

鍵の導出【derivation】と引き伸ばし【stretching】のアルゴリズムはセキュアなパスワードのハッシュ化のために設計されました。 `sha1 (password)` のようなうぶなアルゴリズムは、ブルートフォース攻撃への抵抗者にはなりません。良いパスワードハッシュ化は調節可能で、遅くて、 `salt` を含まなければなりません。

`hashlib.pbkdf2_hmac (name, password, salt, rounds, dklen=None)`

この関数は PKCS#5 のパスワードに基づいた鍵導出関数 2 を提供しています。疑似乱数関数として HMAC を使用しています。

文字列 *name* は、HMAC のハッシュダイジェストにしたいアルゴリズムの名前で、例えば `'sha1'` や `'sha256'` です。 *password* と *salt* はバイト列のバッファとして解釈されます。アプリケーションとライブラリは *password* に相応しい値 (例えば 1024) に制限すべきです。 *salt* は `os.urandom ()` のような適切な所源から得た、およそ 16 かそれよりも多いバイト列でなければなりません。

rounds 数はハッシュアルゴリズムと計算機の能力に基いて決められるべきです。2013 年現在の場合、SHA-256 に対して最低でも 100,000 ラウンドをお奨めします。

dklen が None の場合、ハッシュアルゴリズム *name* のダイジェストサイズが使われます。例えば SHA-512 では 64 です。

```
>>> import hashlib, binascii
>>> dk = hashlib.pbkdf2_hmac('sha256', b'password', b'salt', 100000)
>>> binascii.hexlify(dk)
b'0394a2ede332c9a13eb82e9b24631604c31df978b4e2f0fbd2c549944f9d79a5'
```

バージョン 2.7.8 で追加.

注釈: `pbkdf2_hmac` の高速な実装は OpenSSL 使用版で利用可能です。Python 実装は `hmac` のインラインバージョンを使います。それはおよそ 3 倍遅く、GIL を解放しません。

参考:

`hmac` モジュール `ハッシュを用いてメッセージ認証コードを生成するモジュールです。`

`base64` モジュール バイナリハッシュを非バイナリ環境用にエンコードするもうひとつの方法です。

<http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf> FIPS 180-2 のセキュアハッシュアルゴリズムについての説明。

<https://ja.wikipedia.org/wiki/%E6%9A%97%E5%8F%B7%E5%AD%A6%E7%9A%84%E3%83%8F%E3%83%83%E3%>

どのアルゴリズムにどんな既知の問題があって、それが実際に利用する際にどう影響するかについての Wikipedia の記事。

14.2 hmac — メッセージ認証のための鍵付きハッシュ化

バージョン 2.2 で追加.

ソースコード: [Lib/hmac.py](#)

このモジュールでは **RFC 2104** で記述されている HMAC アルゴリズムを実装しています。

hmac.new (key[, msg[, digestmod]])

新しい hmac オブジェクトを返します。 `msg` が与えられると、`update(msg)` が呼び出されます。
`digestmod` は利用するダイジェストコンストラクターまたは HMAC オブジェクトのモジュールです。
デフォルトは `hashlib.md5` のコンストラクターです。

HMAC オブジェクトは以下のメソッドを持っています:

HMAC.update(*msg*)

hmac オブジェクトを *msg* で更新します。このメソッドの呼出の繰り返しは、それらの引数を全て結合した引数で単一の呼び出しをした際と同じになります。すなわち `m.update(a); m.update(b)` は `m.update(a + b)` と等価です。

HMAC.digest()

これまで `update()` メソッドに渡されたバイト列のダイジェスト値を返します。これはコンストラクタに与えられた `digest_size` と同じ長さのバイト列で、NUL バイトを含む非 ASCII 文字が含まれることがあります。

警告: `digest()` の出力結果と外部から供給されたダイジェストを検証ルーチン内で比較しようとするのであれば、タイミング攻撃への脆弱性を減らすために、`==` 演算子ではなく `compare_digest()` を使うことをお勧めします。

HMAC.hexdigest()

`digest()` と似ていますが、返される文字列は倍の長さとなり、16 進形式となります。これは、電子メールなどの非バイナリ環境で値を交換する場合に便利です。

警告: `hexdigest()` の出力結果と外部から供給されたダイジェストを検証ルーチン内で比較しようとするのであれば、タイミング攻撃への脆弱性を減らすために、`==` 演算子ではなく `compare_digest()` を使うことをお勧めします。

HMAC.copy()

hmac オブジェクトのコピー (“クローン”) を返します。このコピーは最初の部分文字列が共通になっている文字列のダイジェスト値を効率よく計算するために使うことができます。

このモジュールは以下のヘルパ関数も提供しています:

hmac.compare_digest(a, b)

`a == b` を返します。この関数は、内容ベースの短絡的な振る舞いを避けることによってタイミング分析を防ぐよう設計されたアプローチを用い、暗号化に用いるのに相応しいものとしています。`a` と `b` は両方が同じ型でなければなりません: `unicode` または `bytes-like object` のどちらか一方。

注釈: `a` と `b` が異なる長さであったりエラーが発生すれば場合には、タイミング攻撃で理論上 `a` と `b` の型と長さについての情報が暴露されますが、その値は明らかになりません。

バージョン 2.7.7 で追加.

参考:

`hashlib` モジュール セキュアハッシュ関数を提供する Python モジュールです。

14.3 md5 — MD5 メッセージダイジェストアルゴリズム

バージョン 2.5 で非推奨: 代わりにモジュール `hashlib` を使ってください。

このモジュールは RSA 社の MD5 メッセージダイジェストアルゴリズムへのインタフェースを実装してい

ます。(Internet [RFC 1321](#) も参照してください)。利用方法は極めて単純です。まず md5 オブジェクトを `new()` を使って生成します。後は `update()` メソッドを使って、生成されたオブジェクトに任意の文字列データを入力します。オブジェクトに入力された文字列データ全体の *digest* ("fingerprint" として知られる強力な 128-bit チェックサム) は `digest()` を使っていつでも調べることができます。

たとえば、'Nobody inspects the spammish repetition' というバイト文字列のダイジェストを取得するには次のようにします:

```
>>> import md5
>>> m = md5.new()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
```

もっと簡潔に書くと、このようになります:

```
>>> md5.new("Nobody inspects the spammish repetition").digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
```

以下の値はモジュールの中で定数として与えられており、`new()` で返される md5 オブジェクトの属性としても与えられます:

`md5.digest_size`

返されるダイジェスト値のバイト数で表した長さ。常に 16 です。

md5 クラスオブジェクトは以下のメソッドをサポートします:

`md5.new([arg])`

新たな md5 オブジェクトを返します。もし *arg* が存在するなら、`update(arg)` を呼び出します。

`md5.md5([arg])`

下位互換性のために、`new()` の別名として提供されています。

md5 オブジェクトは以下のメソッドをサポートします:

`md5.update(arg)`

文字列 *arg* を入力として md5 オブジェクトを更新します。このメソッドを繰り返して呼び出す操作は、それぞれの呼び出し時の引数 *arg* を結合したデータを引数として一回の呼び出す操作と同等になります: つまり、`m.update(a); m.update(b)` は `m.update(a+b)` と同等です。

`md5.digest()`

これまで `update()` で与えてきた文字列入力のダイジェストを返します。返り値は 16 バイトの文字列で、null バイトを含む非 ASCII 文字が入っているかもしれません。

`md5.hexdigest()`

`digest()` に似ていますが、ダイジェストは長さ 32 の文字列になり、16 進表記文字しか含みません。この文字列は電子メールやその他のバイナリを受け付けない環境でダイジェストを安全にやりとりするために使うことができます。

`md5.copy()`

`md5` オブジェクトのコピー (‘クローン’) を返します。冒頭の部分文字列が共通な複数の文字列のダイジェストを効率よく計算する際に使うことができます。

参考:

Module `sha` Secure Hash Algorithm (SHA) を実装した類似のモジュール。SHA アルゴリズムはより安全なハッシュアルゴリズムだと考えられています。

14.4 sha — SHA-1 メッセージダイジェストアルゴリズム

バージョン 2.5 で非推奨: 代わりにモジュール `hashlib` を使ってください。

このモジュールは、SHA-1 として知られている、NIST のセキュアハッシュアルゴリズムへのインターフェースを実装しています。SHA-1 はオリジナルの SHA ハッシュアルゴリズムを改善したバージョンです。 `md5` モジュールと同じように使用します。 `sha` オブジェクトを生成するために `new()` を使い、 `update()` メソッドを使って、このオブジェクトに任意の文字列を入力し、それまでに入力した文字列全体の *digest* をいつでも調べるすることができます。SHA-1 のダイジェストは MD5 の 128 bit とは異なり、160 bit です。

`sha.new([string])`

新たな `sha` オブジェクトを返します。もし *string* が存在するなら、 `update(string)` を呼び出します。

以下の値はモジュールの中で定数として与えられており、 `new()` で返される `sha` オブジェクトの属性としても与えられます:

`sha.blocksize`

ハッシュ関数に入力されるブロックのサイズ。このサイズは常に 1 です。このサイズは、任意の文字列をハッシュできるようにするために使われます。

`sha.digest_size`

返されるダイジェスト値をバイト数で表した長さ。常に 20 です。

`sha` オブジェクトには `md5` オブジェクトと同じメソッドがあります。

`sha.update(arg)`

文字列 *arg* を入力として `sha` オブジェクトを更新します。このメソッドを繰り返し呼び出す (操作は、それぞれの呼び出し時の引数を結合したデータを引数として一回の呼び出す操作と同等になります。つまり、 `m.update(a); m.update(b)` は `m.update(a+b)` と同等です。

`sha.digest()`

これまで `update()` メソッドで与えてきた文字列のダイジェストを返します。戻り値は 20 バイトの文字列で、 `null` バイトを含む非 ASCII 文字が入っているかもしれません。

`sha.hexdigest()`

`digits()` と似ていますが、ダイジェストは長さ 40 の文字列になり、16 進表記数字しか含みません。電子メールやその他のバイナリを受け付けない環境で安全に値をやりとりするために使うことができます。

`sha.copy()`

`sha` オブジェクトのコピー (” クローン ”) を返します。冒頭の部分文字列が共通な複数の文字列のダイジェストを効率よく計算する際に使うことができます。

参考:

‘セキュアハッシュスタンダード <<http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf>>’
セキュアハッシュアルゴリズムは NIST のドキュメント FIPS PUB 180-2 で定義されています。‘セキュアハッシュスタンダード <<http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf>>’, 2002 年 8 月出版。

‘暗号ツールキット (セキュアハッシュ) <<http://csrc.nist.gov/CryptoToolkit/tkhash.html>>’
NIST から提供されているセキュアハッシュに関するさまざまな情報へのリンク

第 15 章

汎用オペレーティングシステムサービス

本章に記述されたモジュールは、ファイルの取り扱いや時間計測のような (ほぼ) すべてのオペレーティングシステムで利用可能な機能にインタフェースを提供します。これらのインタフェースは、Unix もしくは C のインタフェースを基に作られますが、ほとんどの他のシステムで同様に利用可能です。概要を以下に記述します:

15.1 `os` — 雑多なオペレーティングシステムインタフェース

このモジュールは、OS 依存の機能を利用するポータブルな方法を提供します。単純なファイルの読み書きについては、`open()` を参照してください。パス操作については、`os.path` モジュールを参照してください。コマンドラインに与えられたすべてのファイルから行を読み込んでいくには、`fileinput` モジュールを参照してください。一時ファイルや一時ディレクトリの作成については、`tempfile` モジュールを参照してください。高レベルなファイルとディレクトリの操作については、`shutil` モジュールを参照してください。

利用可能性に関する注意:

- Python の、すべての OS 依存モジュールの設計方針は、可能な限り同一のインタフェースで同一の機能を利用できるようにする、というものです。例えば、`os.stat(path)` は `path` に関する `stat` 情報を、(POSIX を元にした) 同じフォーマットで返します。
- 特定のオペレーティングシステム固有の拡張も `os` を介して利用することができますが、これらの利用はもちろん、可搬性を脅かします。
- 「利用できる環境: Unix」の意味はこの関数が Unix システムにあることが多いということです。このことは特定の OS における存在を主張するものではありません。
- 特に記述がない場合、「利用できる環境: Unix」と書かれている関数は、Unix をコアにしている Mac OS X でも利用することができます。

注釈: このモジュール内のすべての関数は、間違った、あるいはアクセス出来ないファイル名やファイルパス、その他型が合っていない OS が受理しない引数に対して、`OSError` を送出します。

exception `os.error`

組み込みの `OSError` 例外に対するエイリアスです。

os.name

import されているオペレーティング・システム依存モジュールの名前です。現在次の名前が登録されています: 'posix', 'nt', 'os2', 'ce', 'java', 'riscos'.

参考:

`sys.platform` はより細かな粒度を持っています。 `os.uname()` はシステム依存のバージョン情報を提供します。

`platform` モジュールはシステムの詳細な識別情報をチェックする機能を提供しています。

15.1.1 プロセスのパラメーター

これらの関数とデータアイテムは、現在のプロセスおよびユーザーに対する情報提供および操作のための機能を提供しています。

os.environ

マップ型 オブジェクトは文字の環境を表します。例えば、 `environ['HOME']` はホームディレクトリのパス名であり、C における `getenv("HOME")` と等価です。

このマップ型の内容は、 `os` モジュールの最初の import の時点、通常は Python の起動時に `site.py` が処理される中で取り込まれます。それ以後に変更された環境変数は `os.environ` を直接変更しない限り `os.environ` には反映されません。

プラットフォーム上で `putenv()` がサポートされている場合、このマップ型オブジェクトは環境変数に対する変更にも使えます。 `putenv()` はマップ型オブジェクトが修正される時に、自動的に呼ばれることになります。

注釈: `putenv()` を直接呼び出しても `os.environ` の内容は変わらないので、 `os.environ` を直接変更する方がベターです。

注釈: FreeBSD と Mac OS X を含む一部のプラットフォームでは、 `environ` の値を変更するとメモリリークの原因になる場合があります。システムの `putenv()` に関するドキュメントを参照してください。

`putenv()` が提供されていない場合、このマップ型オブジェクトに変更を加えたコピーを適切なプロセス生成機能に渡すことで、生成された子プロセスが変更された環境変数を利用するようにできます。

プラットフォームが `unsetenv()` 関数をサポートしている場合、このマップ型オブジェクトからアイテムを削除することで環境変数を消すことができます。 `unsetenv()` は `os.environ` からアイテムが取り除かれた時に自動的に呼ばれます。 `pop()` または `clear()` が呼ばれた時も同様です。

バージョン 2.6 で変更: `os.environ.clear()` か `os.environ.pop()` を呼び出した時も、(delete した時と同様に) 環境変数を削除するようになりました。

`os.chdir(path)`

`os.fchdir(fd)`

`os.getcwd()`

これらの関数は、[ファイルとディレクトリ](#) 節で説明されています。

`os.ctermid()`

プロセスの制御端末に対応するファイル名を返します。

利用できる環境：Unix。

`os.getegid()`

現在のプロセスの実効グループ id を返します。この id は現在のプロセスで実行されているファイルの "set id" ビットに対応します。

利用できる環境：Unix。

`os.geteuid()`

現在のプロセスの実効ユーザー id を返します。

利用できる環境：Unix。

`os.getgid()`

現在のプロセスの実グループ id を返します。

利用できる環境：Unix。

`os.getgroups()`

現在のプロセスに関連づけられた従属グループ id のリストを返します。

利用できる環境：Unix。

注釈：Mac OS X では `getgroups()` の挙動は他の Unix プラットフォームとはいくぶん異なります。Python の Deployment Target が 10.5 以前でビルドされている場合、`getgroups()` は現在のユーザープロセスに関連付けられている実効グループ id を返します；このリストはシステムで定義されたエントリ数（通常は 16）に制限され、適切な特権があれば `setgroups()` の呼び出しによって変更される場合があります。Deployment Target が 10.5 より新しい場合、`getgroups()` はプロセスの実効ユーザー id に関連付けられたユーザーの現在のグループアクセスリストを返します；グループアクセスリストはプロセスのライフタイムで変更される可能性があり、`setgroups()` の呼び出しの影響を受けず、長さ 16 の制限を受けません。Deployment Target の値 `MACOSX_DEPLOYMENT_TARGET` は、`sysconfig.get_config_var()` で取得することができます。

`os.initgroups(username, gid)`

システムの `initgroups()` を呼び出し、指定された `username` がメンバーである全グループと `gid` で指定されたグループでグループアクセスリストを初期化します。

利用できる環境：Unix。

バージョン 2.7 で追加。

`os.getlogin()`

現在のプロセスの制御端末にログインしているユーザ名を返します。ほとんどの場合、ユーザが誰かを知りたいときには環境変数 `LOGNAME` を、プロセスの実ユーザ ID のログイン名を知りたいときには `pwd.getpwuid(os.getuid())[0]` を使うほうが便利です。

利用できる環境: Unix。

`os.getpgid(pid)`

プロセス id `pid` のプロセスのプロセスグループ id を返します。もし `pid` が 0 ならば、現在のプロセスのプロセスグループ id を返します。

利用できる環境: Unix。

バージョン 2.3 で追加。

`os.getpgrp()`

現在のプロセスグループの id を返します。

利用できる環境: Unix。

`os.getpid()`

現在のプロセス id を返します。

利用できる環境: Unix、Windows。

`os.getppid()`

親プロセスの id を返します。

利用できる環境: Unix。

`os.getresuid()`

現在のプロセスの実ユーザー id、実効ユーザー id、および保存ユーザー id を示す、`(ruid, euid, suid)` のタプルを返します。

利用できる環境: Unix。

バージョン 2.7 で追加。

`os.getresgid()`

現在のプロセスの実グループ id、実効グループ id、および保存グループ id を示す、`(rgid, egid, sgid)` のタプルを返します。

利用できる環境: Unix。

バージョン 2.7 で追加。

`os.getuid()`

現在のプロセスの実ユーザー id を返します。

利用できる環境: Unix。

`os.getenv(varname[, value])`

環境変数 `varname` が存在する場合にはその値を返し、存在しない場合には `value` を返します。 `value` の

デフォルト値は `None` です。

利用できる環境：主な Unix 互換環境、Windows。

`os.putenv (varname, value)`

`varname` と名づけられた環境変数の値を文字列 `value` に設定します。このような環境変数への変更は、`os.system()`、`popen()`、`fork()` および `execv()` により起動された子プロセスに影響します。

利用できる環境：主な Unix 互換環境、Windows。

注釈：FreeBSD と Mac OS X を含む一部のプラットフォームでは、`environ` の値を変更するとメモリリークの原因になる場合があります。システムの `putenv` に関するドキュメントを参照してください。

`putenv()` がサポートされている場合、`os.environ` のアイテムに対する代入を行うと自動的に `putenv()` が呼び出されます；直接 `putenv()` を呼び出した場合 `os.environ` は更新されないため、実際には `os.environ` のアイテムに代入する方が望ましい操作です。

`os.setegid (egid)`

現在のプロセスに実効グループ id をセットします。

利用できる環境：Unix。

`os.seteuid (euid)`

現在のプロセスに実効ユーザー id をセットします。

利用できる環境：Unix。

`os.setgid (gid)`

現在のプロセスにグループ id をセットします。

利用できる環境：Unix。

`os.setgroups (groups)`

現在のグループに関連付けられた従属グループ id のリストを `groups` に設定します。`groups` はシーケンス型でなくてはならず、各要素はグループを特定する整数でなくてはなりません。通常、この操作はスーパーユーザーしか利用できません。

利用できる環境：Unix。

バージョン 2.2 で追加。

注釈：Mac OS X では、`groups` の長さはシステムで定義された実効グループ id の最大数（通常は 16）を超えない場合があります。`setgroups()` 呼び出しで設定されたものと同じグループリストが返されないケースについては、`getgroups()` のドキュメントを参照してください。

`os.setpgrp ()`

システムコール `setpgrp()` か `setpgrp(0, 0)` のどちらか（実装されているもの）を呼び出します。機能については UNIX マニュアルを参照して下さい。

利用できる環境 : Unix。

os.setpgid(*pid*, *pgrp*)

システムコール `setpgid()` を呼び出してプロセス *pid* のプロセスのプロセスグループ *id* を *pgrp* に設定します。この動作に関しては Unix のマニュアルを参照してください。

利用できる環境 : Unix。

os.setregid(*rgid*, *egid*)

現在のプロセスの実グループ *id* および実効グループ *id* を設定します。

利用できる環境 : Unix。

os.setresgid(*rgid*, *egid*, *sgid*)

現在のプロセスの、実グループ *id*、実効グループ *id*、および保存グループ *id* を設定します。

利用できる環境 : Unix。

バージョン 2.7 で追加。

os.setresuid(*ruid*, *euid*, *suid*)

現在のプロセスの実ユーザー *id*、実効ユーザー *id*、および保存ユーザー *id* を設定します。

利用できる環境 : Unix。

バージョン 2.7 で追加。

os.setreuid(*ruid*, *euid*)

現在のプロセスの実ユーザー *id* および実効ユーザー *id* を設定します。

利用できる環境 : Unix。

os.getsid(*pid*)

`getsid()` システムコールを呼び出します。機能については Unix のマニュアルを参照してください。

利用できる環境 : Unix。

バージョン 2.4 で追加。

os.setsid()

`setsid()` システムコールを呼び出します。機能については Unix のマニュアルを参照してください。

利用できる環境 : Unix。

os.setuid(*uid*)

現在のプロセスのユーザー *id* を設定します。

利用できる環境 : Unix。

os.strerror(*code*)

エラーコード *code* に対応するエラーメッセージを返します。未知のエラーコードの対して `strerror()` が `NULL` を返すプラットフォームでは、`ValueError` が送出されます。

利用できる環境 : Unix、Windows。

`os.umask(mask)`

現在の数値 `umask` を設定し、以前の `umask` 値を返します。

利用できる環境: Unix、Windows。

`os.uname()`

現在のオペレーティングシステムを特定する情報の入った 5 要素のタプルを返します。このタプルには 5 つの文字列: (`sysname`, `nodename`, `release`, `version`, `machine`) が入っています。システムによっては、ノード名を 8 文字、または先頭の要素だけに切り詰めます; ホスト名を取得する方法としては、`socket.gethostname()` を使う方がよいでしょう、あるいは `socket.gethostbyaddr(socket.gethostname())` でもかまいません。

利用できる環境: Unix 互換環境。

`os.unsetenv(varname)`

`varname` という名前の環境変数を取り消します。このような環境の変化は `os.system()`, `popen()` または `fork()` と `execv()` で開始されるサブプロセスに影響を与えます。

`unsetenv()` がサポートされている場合、`os.environ` のアイテムの削除を行うと自動的に `unsetenv()` が呼び出されます。直接 `unsetenv()` を呼び出した場合 `os.environ` は更新されないため、実際には `os.environ` のアイテムを削除の方が望ましい操作です。

利用できる環境: 主な Unix 互換環境、Windows。

15.1.2 ファイルオブジェクトの生成

以下の関数は新しいファイルオブジェクトを作成します。(`open()` も参照してください)

`os.fdopen(fd[, mode[, bufsize]])`

ファイル記述子 `fd` に接続している、開かれたファイルオブジェクトを返します。引数 `mode` および `bufsize` は、組み込み関数 `open()` における対応する引数と同じ意味を持ちます。`fdopen()` が例外を起こした場合は、`fd` はそのままにされます (クローズされません)。

利用できる環境: Unix、Windows。

バージョン 2.3 で変更: 引数 `mode` は、指定されるならば、`'r'`, `'w'`, `'a'` のいずれかの文字で始まらなければなりません。そうでなければ `ValueError` が送出されます。

バージョン 2.5 で変更: Unix では、引数 `mode` が `'a'` で始まる時には `O_APPEND` フラグがファイル記述子に設定されます。(ほとんどのプラットフォームで `fdopen()` 実装が既に行なっていることです)。

`os.popen(command[, mode[, bufsize]])`

`command` への、または `command` からのパイプ入出力を開きます。戻り値はパイプに接続されている開かれたファイルオブジェクトで、`mode` が `'r'` (標準の設定です) または `'w'` によって読み出しまたは書き込みを行うことができます。引数 `bufsize` は、組み込み関数 `open()` における対応する引数と同じ意味を持ちます。`command` の終了ステータス (`wait()` で指定された書式でコード化されてい

ます) は、`close()` メソッドの戻り値として取得することができます。例外は終了ステータスがゼロ (すなわちエラーなしで終了) の場合で、このときには `None` を返します。

利用できる環境: Unix、Windows。

バージョン 2.6 で非推奨: この関数は廃止予定です。 `subprocess` モジュールを使用してください。特に 古い関数を `subprocess` モジュールで置き換える セクションをチェックしてください。

バージョン 2.0 で変更: この関数は、Python の初期のバージョンでは、Windows 環境下で信頼できない動作をしていました。これは Windows に付属して提供されるライブラリの `_popen()` 関数を利用したことによるものです。新しいバージョンの Python では、Windows 付属のライブラリにある壊れた実装を利用しません。

`os.tmpfile()`

更新モード (`w+b`) で開かれた新しいファイルオブジェクトを返します。このファイルはディレクトリエントリ登録に関連付けられておらず、このファイルに対するファイル記述子がなくなると自動的に削除されます。

利用できる環境: Unix、Windows。

幾つかの少し異なった方法で子プロセスを作成するために、幾つかの `popen*` () 関数が提供されています。

バージョン 2.6 で非推奨: 全ての `popen*` () 関数は撤廃されました。代わりに `subprocess` モジュールを利用してください。

`popen*` () の変種はどれも、`bufsize` が指定されている場合には I/O パイプのバッファサイズを表します。`mode` を指定する場合には、文字列 `'b'` または `'t'` でなければなりません; これは、Windows でファイルをバイナリモードで開くかテキストモードで開くかを定めるために必要です。`mode` の標準の設定値は `'t'` です。

また Unix ではこれらの変種はいずれも `cmd` をシーケンスにできます。その場合、引数はシェルの介在なしに直接 (`os.spawnv()` のように) 渡されます。`cmd` が文字列の場合、引数は (`os.system()` のように) シェルに渡されます。

以下のメソッドは子プロセスから終了ステータスを取得できるようにはしていません。入出力ストリームを制御し、かつ終了コードの取得も行える唯一の方法は、`subprocess` モジュールを利用する事です。以下のメソッドは Unix でのみ利用可能です。

これらの関数の利用に関係して起きうるデッドロック状態についての議論は、[フロー制御の問題](#) 節を参照してください。

`os.popen2(cmd[, mode[, bufsize]])`

`cmd` を子プロセスとして実行します。ファイル・オブジェクト (`child.stdin`, `child.stdout`) を返します。

バージョン 2.6 で非推奨: この関数は廃止予定です。 `subprocess` モジュールを使用してください。特に 古い関数を `subprocess` モジュールで置き換える セクションをチェックしてください。

利用できる環境: Unix、Windows。

バージョン 2.0 で追加.

`os.popen3(cmd[, mode[, bufsize]])`

`cmd` を子プロセスとして実行します。ファイルオブジェクト (`child_stdin`, `child_stdout`, `child_stderr`) を返します。

バージョン 2.6 で非推奨: この関数は廃止予定です。 `subprocess` モジュールを使用してください。特に 古い関数を `subprocess` モジュールで置き換える セクションをチェックしてください。

利用できる環境: Unix、Windows。

バージョン 2.0 で追加。

`os.popen4(cmd[, mode[, bufsize]])`

`cmd` を子プロセスとして実行します。ファイルオブジェクト (`child_stdin`, `child_stdout_and_stderr`) を返します。

バージョン 2.6 で非推奨: この関数は廃止予定です。 `subprocess` モジュールを使用してください。特に 古い関数を `subprocess` モジュールで置き換える セクションをチェックしてください。

利用できる環境: Unix、Windows。

バージョン 2.0 で追加。

(`child_stdin`, `child_stdout`, および `child_stderr` は子プロセスの視点で名付けられているので注意してください。すなわち、`child_stdin` とは子プロセスの標準入力を意味します。)

この機能は `popen2` モジュール内の同じ名前の関数を使っても実現できますが、これらの関数の戻り値は異なる順序を持っています。

15.1.3 ファイル記述子の操作

これらの関数は、ファイル記述子を使って参照されている I/O ストリームを操作します。

ファイル記述子とは現在のプロセスで開かれたファイルに対応する小さな整数です。例えば、標準入力 of ファイル記述子は常に 0 で、標準出力は 1、標準エラーは 2 です。プロセスから開かれたその他のファイルには 3、4、5、などが割り振られます。「ファイル記述子」という名称は少し誤解を与えるものかもしれませんが、Unix プラットフォームにおいて、ソケットやパイプもファイル記述子によって参照されます。

ファイルオブジェクトに紐付けられたファイル記述子は `fileno()` メソッドによって取得可能です。ただし、ファイル記述子を直接使うとファイルオブジェクトのメソッドは経由しませんので、内部でバッファするかどうかといったファイルオブジェクトの都合は無視されます。

`os.close(fd)`

ファイル記述子 `fd` をクローズします。

利用できる環境: Unix、Windows。

注釈: この関数は低水準の I/O 向けのもので、`os.open()` や `pipe()` が返すファイル記述子に対して使用しなければなりません。組み込み関数 `open()` や `popen()`、`fdopen()` が返す”ファイルオ

プロジェクト”を閉じるには、オブジェクトの `close()` メソッドを使用してください。

os.closerange(*fd_low*, *fd_high*)

fd_low (を含む) から *fd_high* (含まない) までの全てのファイル記述子を、エラーを無視しながら閉じます。次のコードと等価です

```
for fd in xrange(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

利用できる環境: Unix、Windows。

バージョン 2.6 で追加.

os.dup(*fd*)

ファイル記述子 *fd* の複製を返します。

利用できる環境: Unix、Windows。

os.dup2(*fd*, *fd2*)

ファイル記述子を *fd* から *fd2* に複製し、必要なら後者の記述子を前もって閉じておきます。

利用できる環境: Unix、Windows。

os.fchmod(*fd*, *mode*)

fd で指定されたファイルのモードを *mode* に変更します。 *mode* に指定できる値については、`chmod()` のドキュメントを参照してください。

利用できる環境: Unix。

バージョン 2.6 で追加.

os.fchown(*fd*, *uid*, *gid*)

fd で指定されたファイルの owner id と group id を、 *uid* と *gid* に変更します。どちらかの id を変更しない場合は、 -1 を渡してください。

利用できる環境: Unix。

バージョン 2.6 で追加.

os.fdatasync(*fd*)

ファイル記述子 *fd* を持つファイルのディスクへの書き込みを強制します。メタデータの更新は強制しません。

利用できる環境: Unix。

注釈: この関数は MacOS では利用できません。

os.fpathconf (*fd*, *name*)

オープンしているファイルに関連するシステム設定情報を返します。*name* には取得したい設定名を指定します；これは定義済みのシステム値名の文字列で、多くの標準 (POSIX.1、Unix 95、Unix 98 その他) で定義されています。プラットフォームによっては別の名前も定義されています。ホストオペレーティングシステムの関知する名前は `pathconf.names` 辞書で与えられています。このマップ型オブジェクトに入っていない設定変数については、*name* に整数を渡してもかまいません。

もし *name* が文字列でかつ不明である場合、`ValueError` を送出します。*name* の指定値がホストシステムでサポートされておらず、`pathconf.names` にも入っていない場合、`errno.EINVAL` をエラー番号として `OSError` を送出します。

利用できる環境：Unix。

os.fstat (*fd*)

`stat()` のようにファイル記述子 *fd* の状態を返します。

利用できる環境：Unix、Windows。

os.fstatvfs (*fd*)

`statvfs()` のように、ファイル記述子 *fd* に関連づけられたファイルが入っているファイルシステムに関する情報を返します。

利用できる環境：Unix。

os.fsync (*fd*)

ファイル記述子 *fd* を持つファイルのディスクへの書き込みを強制します。Unix では、ネイティブの `fsync()` 関数を、Windows では `_commit()` 関数を呼び出します。

Python のファイルオブジェクト *f* を使う場合、*f* の内部バッファを確実にディスクに書き込むために、まず `f.flush()` を実行し、それから `os.fsync(f.fileno())` してください。

利用できる環境：Unix, Windows (2.2.3 以降)

os.ftruncate (*fd*, *length*)

ファイル記述子 *fd* に対応するファイルを、サイズが最大で *length* バイトになるように切り詰めます。

利用できる環境：Unix。

os.isatty (*fd*)

ファイル記述子 *fd* がオープンされていて、`tty()` (のような) デバイスに接続されている場合、`True` を返します。そうでない場合は `False` を返します。

os.lseek (*fd*, *pos*, *how*)

ファイル記述子 *fd* の現在の位置を *pos* に設定します。*pos* の意味は *how* で修飾されます：ファイルの先頭からの相対には `SEEK_SET` か 0 を設定します；現在の位置からの相対には `SEEK_CUR` か 1 を設定します；ファイルの末尾からの相対には `SEEK_END` か 2 を設定します。戻り値は開始位置からの、バイトで数えた新しいカーソル位置です。

利用できる環境：Unix、Windows。

os.SEEK_SET

os.**SEEK_CUR**

os.**SEEK_END**

`lseek()` 関数に渡すパラメーター。値は順に 0, 1, 2 です。

利用出来る環境 : Windows, Unix

バージョン 2.5 で追加.

os.**open**(*file*, *flags*[, *mode*])

ファイル *file* を開き、*flag* に従って様々なフラグを設定し、可能なら *mode* に従ってファイルモードを設定します。*mode* の標準の設定値は 0777 (8 進表現) で、先に現在の umask を使ってマスクを掛けます。新たに開かれたファイルのファイル記述子を返します。

フラグとファイルモードの値についての詳細は C ランタイムのドキュメントを参照してください ; (`O_RDONLY` や `O_WRONLY` のような) フラグ定数はこのモジュールでも定義されています (`open()` フラグ定数を参照してください)。特に、Windows ではバイナリモードでファイルを開く時に `O_BINARY` を加える必要があります。

利用できる環境 : Unix、Windows。

注釈: この関数は低レベルの I/O のためのものです。通常の利用では、`read()` や `write()` (やその他多くの) メソッドを持つ「ファイルオブジェクト」を返す、組み込み関数 `open()` を使ってください。ファイル記述子を「ファイルオブジェクト」でラップするには `fdopen()` を使ってください。

os.**openpty**()

新しい擬似端末のペアを開きます。ファイル記述子のペア (*master*, *slave*) を返し、それぞれ `pty` および `tty` を表します。(少しだけ) より可搬性のあるアプローチとしては、`pty` モジュールを使用してください。

利用できる環境 : 一部の Unix 互換環境。

os.**pipe**()

パイプを作成します。ファイル記述子のペア (*r*, *w*) を返し、それぞれ読み込み、書き出し用に使うことができます。

利用できる環境 : Unix、Windows。

os.**read**(*fd*, *n*)

ファイル記述子 *fd* から最大で *n* バイト読み出します。読み出されたバイト列の入った文字列を返します。*fd* が参照しているファイルの終端に達した場合、空の文字列が返されます。

利用できる環境 : Unix、Windows。

注釈: この関数は低水準の I/O 向けのもので、`os.open()` や `pipe()` が返すファイル記述子に対して使用されなければなりません。組み込み関数 `open()` や `popen()`、`fdopen()`、あるいは `sys.stdin` が返す「ファイルオブジェクト」を読み込むには、オブジェクトの `read()` か `readline()`

メソッドを使用してください。

`os.tcgetpgrp (fd)`

`fd` (`os.open()` が返すオープンしたファイル記述子) で与えられる端末に関連付けられたプロセスグループを返します。

利用できる環境: Unix。

`os.tcsetpgrp (fd, pg)`

`fd` (`os.open()` が返すオープンしたファイル記述子) で与えられる端末に関連付けられたプロセスグループを `pg` に設定します。

利用できる環境: Unix。

`os.ttyname (fd)`

ファイル記述子 `fd` に関連付けられている端末デバイスを特定する文字列を返します。`fd` が端末に関連付けられていない場合、例外が送出されます。

利用できる環境: Unix。

`os.write (fd, str)`

ファイル記述子 `fd` に文字列 `str` を書き込みます。実際に書き込まれたバイト数を返します。

利用できる環境: Unix、Windows。

注釈: この関数は低水準の I/O 向けのもので、`os.open()` や `pipe()` が返すファイル記述子に対して使用しなければなりません。組み込み関数 `open()` や `popen()`、`fdopen()`、あるいは `sys.stdout` や `sys.stderr` が返す”ファイルオブジェクト”に書き込むには、オブジェクトの `write()` メソッドを使用してください。

`open()` フラグ定数

以下の定数は `open()` 関数の `flags` 引数に利用します。これらの定数は、ビット単位に OR 演算子 `|` で組み合わせることができます。一部、すべてのプラットフォームでは使用できない定数があります。利用可能かどうかや使い方については、Unix では `open(2)`、Windows では [MSDN](#) を参照してください。

`os.O_RDONLY`

`os.O_WRONLY`

`os.O_RDWR`

`os.O_APPEND`

`os.O_CREAT`

`os.O_EXCL`

`os.O_TRUNC`

上記の定数は Unix および Windows で利用可能です。

`os.O_DSYNC`

`os.O_RSYNC`
`os.O_SYNC`
`os.O_NDELAY`
`os.O_NONBLOCK`
`os.O_NOCTTY`

上記の定数は Unix でのみ利用可能です。

`os.O_BINARY`
`os.O_NOINHERIT`
`os.O_SHORT_LIVED`
`os.O_TEMPORARY`
`os.O_RANDOM`
`os.O_SEQUENTIAL`
`os.O_TEXT`

上記の定数は Windows でのみ利用可能です。

`os.O_ASYNC`
`os.O_DIRECT`
`os.O_DIRECTORY`
`os.O_NOFOLLOW`
`os.O_NOATIME`
`os.O_SHLOCK`
`os.O_EXLOCK`

上記の定数は拡張仕様であり、C ライブラリで定義されていない場合は利用できません。

15.1.4 ファイルとディレクトリ

`os.access(path, mode)`

実 uid/gid を使って *path* に対するアクセスが可能か調べます。ほとんどのオペレーティングシステムは実効 uid/gid を使うため、このルーチンは suid/sgid 環境において、プログラムを起動したユーザーが *path* に対するアクセス権をもっているかを調べるために使われます。*path* が存在するかどうかを調べるには *mode* を `F_OK` にします。ファイルアクセス権限 (パーミッション) を調べるには、`R_OK`, `W_OK`, `X_OK` から一つまたはそれ以上のフラグを論理和指定でとることもできます。アクセスが許可されている場合 `True` を、そうでない場合 `False` を返します。詳細は `access(2)` の Unix マニュアルページを参照してください。

利用できる環境: Unix、Windows。

注釈: ユーザーが、例えばファイルを開く権限を持っているかどうかを調べるために実際に `open()` を行う前に `access()` を使用することはセキュリティホールの原因になります。なぜなら、調べた時点とオープンした時点との時間差を利用してそのユーザーがファイルを不当に操作してしまうかもしれないからです。その場合は `EAFP` テクニックを利用するのが望ましいやり方です。例えば

```
if os.access("myfile", os.R_OK):
    with open("myfile") as fp:
        return fp.read()
return "some default data"
```

このコードは次のように書いたほうが良いです

```
try:
    fp = open("myfile")
except IOError as e:
    if e.errno == errno.EACCES:
        return "some default data"
    # Not a permission error.
    raise
else:
    with fp:
        return fp.read()
```

注釈: I/O 操作は `access()` が成功を示した時でも失敗することがあります。特にネットワークファイルシステムが通常の POSIX のパーミッションビットモデルをはみ出すアクセス権限操作を備える場合にはそのようなことが起こりえます。

`os.F_OK`

`access()` の `mode` に渡すための値で、`path` が存在するかどうかを調べます。

`os.R_OK`

`access()` の `mode` に渡すための値で、`path` が読み出し可能かどうかを調べます。

`os.W_OK`

`access()` の `mode` に渡すための値で、`path` が書き込み可能かどうかを調べます。

`os.X_OK`

`access()` の `mode` に渡すための値で、`path` が実行可能かどうかを調べます。

`os.chdir(path)`

現在の作業ディレクトリを `path` に設定します。

利用できる環境: Unix、Windows。

`os.fchdir(fd)`

現在の作業ディレクトリをファイル記述子 `fd` が表すディレクトリに変更します。記述子はオープンしているファイルではなく、オープンしたディレクトリを参照していなければなりません。

利用できる環境: Unix。

バージョン 2.3 で追加。

`os.getcwd()`

現在の作業ディレクトリを表す文字列を返します。

利用できる環境: Unix、Windows。

`os.getcwdu()`

現在の作業ディレクトリを表現するユニコードオブジェクトを返します。

利用できる環境: Unix、Windows。

バージョン 2.3 で追加.

`os.chflags(path, flags)`

path のフラグを *flags* に変更します。 *flags* は、以下の値 (*stat* モジュールで定義されているもの) をビット単位の論理和で組み合わせることができます:

- *stat.UF_NODUMP*
- *stat.UF_IMMUTABLE*
- *stat.UF_APPEND*
- *stat.UF_OPAQUE*
- *stat.UF_NOUNLINK*
- *stat.UF_COMPRESSED*
- *stat.UF_HIDDEN*
- *stat.SF_ARCHIVED*
- *stat.SF_IMMUTABLE*
- *stat.SF_APPEND*
- *stat.SF_NOUNLINK*
- *stat.SF_SNAPSHOT*

利用できる環境: Unix。

バージョン 2.6 で追加.

`os.chroot(path)`

現在のプロセスに対してルートディレクトリを *path* に変更します。 利用出来る環境: Unix。

バージョン 2.2 で追加.

`os.chmod(path, mode)`

path のモードを数値 *mode* に変更します。 *mode* は、 (*stat* モジュールで定義されている) 以下の値のいずれかまたはビット単位の論理和で組み合わせた値を取り得ます:

- *stat.S_ISUID*
- *stat.S_ISGID*

- `stat.S_ENFMT`
- `stat.S_ISVTX`
- `stat.S_IREAD`
- `stat.S_IWRITE`
- `stat.S_IXEXEC`
- `stat.S_IRWXU`
- `stat.S_IRUSR`
- `stat.S_IWUSR`
- `stat.S_IXUSR`
- `stat.S_IRWXG`
- `stat.S_IRGRP`
- `stat.S_IWGRP`
- `stat.S_IXGRP`
- `stat.S_IRWXO`
- `stat.S_IROTH`
- `stat.S_IWOTH`
- `stat.S_IXOTH`

利用できる環境 : Unix、Windows。

注釈: Windows でも `chmod()` はサポートされていますが、ファイルの読み込み専用フラグを (定数 `S_IWRITE` と `S_IREAD`, または対応する整数値を通して) 設定できるだけです。他のビットは全て無視されます。

os.**chown** (*path*, *uid*, *gid*)

path の所有者 (owner) *id* とグループ *id* を、数値 *uid* および *gid* に変更します。いずれかの *id* を変更せずにおくには、その値として -1 をセットします。

利用できる環境 : Unix。

os.**lchflags** (*path*, *flags*)

path のフラグを数値 *flags* に設定します。 `chflags()` に似ていますが、シンボリックリンクをたどりません。

利用できる環境 : Unix。

バージョン 2.6 で追加.

os.lchmod(*path*, *mode*)

path のモードを数値 *mode* に変更します。パスがシンボリックリンクの場合はそのリンク先ではなくシンボリックリンクそのものに対して作用します。*mode* に指定できる値については `chmod()` のドキュメントを参照してください。

利用できる環境: Unix。

バージョン 2.6 で追加。

os.lchown(*path*, *uid*, *gid*)

path の所有者 (owner) *uid* とグループ *gid* を、数値 *uid* および *gid* に変更します。この関数はシンボリックリンクをたどりません。

利用できる環境: Unix。

バージョン 2.3 で追加。

os.link(*source*, *link_name*)

source を指しているハードリンク *link_name* を作成します。

利用できる環境: Unix。

os.listdir(*path*)

path で指定されたディレクトリ内のエントリ名が入ったリストを返します。リスト内の順番は不定です。特殊エントリ `'.'` および `'..'` は、それらがディレクトリに入っているにもかかわらずリストには含まれません。

利用できる環境: Unix、Windows。

バージョン 2.3 で変更: Windows NT/2k/XP と Unix では、*path* が Unicode オブジェクトの場合、Unicode オブジェクトのリストが返されます。デコード不可能なファイル名は依然として string オブジェクトになります。

os.lstat(*path*)

与えられたパスに対して `lstat()` システムコールと同じ処理を行います。`stat()` と似ていますが、シンボリックリンクをたどりません。シンボリックリンクをサポートしていないプラットフォームでは `stat()` の別名です。

os.mkfifo(*path* [, *mode*])

数値で指定されたモード *mode* を持つ FIFO (名前付きパイプ) を *path* に作成します。*mode* の標準の値は `0666` (8 進) です。現在の `umask` 値が前もって *mode* からマスクされます。

利用できる環境: Unix。

FIFO は通常のファイルのようにアクセスできるパイプです。FIFO は (例えば `os.unlink()` を使って) 削除されるまで存在しつづけます。一般的に、FIFO は "クライアント" と "サーバー" 形式のプロセス間でランデブーを行うために使われます: この時、サーバーは FIFO を読み込み用に、クライアントは書き出し用にオープンします。`mkfifo()` は FIFO をオープンしない — 単にランデブーポイントを作成するだけ — なので注意してください。

`os.mknod(filename[, mode=0600[, device=0]])`

filename という名前で、ファイルシステム・ノード (ファイル、デバイス特殊ファイル、または、名前つきパイプ) を作ります。 *mode* は、作ろうとするノードの使用権限とタイプを、 `stat.S_IFREG`, `stat.S_IFCHR`, `stat.S_IFBLK`, `stat.S_IFIFO` (これらの定数は *stat* で使用可能) のいずれかと (ビット OR で) 組み合わせて指定します。 `S_IFCHR` と `S_IFBLK` を指定すると、 *device* は新しく作られたデバイス特殊ファイルを (おそらく *os.makedev()* を使って) 定義し、指定しなかった場合には無視します。

バージョン 2.3 で追加。

`os.major(device)`

RAW デバイス番号から、デバイスのメジャー番号を取り出します (通常 `stat` の `st_dev` か `st_rdev` フィールドです)。

バージョン 2.3 で追加。

`os.minor(device)`

RAW デバイス番号から、デバイスのマイナー番号を取り出します (通常 `stat` の `st_dev` か `st_rdev` フィールドです)。

バージョン 2.3 で追加。

`os.makedev(major, minor)`

メジャーおよびマイナーデバイス番号から、新しく RAW デバイス番号を作成します。

バージョン 2.3 で追加。

`os.mkdir(path[, mode])`

数値で指定されたモード *mode* をもつディレクトリ *path* を作成します。 *mode* の標準の値は 0777 (8 進) です。指定されたディレクトリがすでに存在する場合は *OSError* 例外が送出されます。

いくつかのシステムにおいては *mode* は無視されます。それが使われる時には、最初に現在の `umask` 値でマスクされます。もし最後の 9 ビット (つまり *mode* の 8 進法表記の最後の 3 桁) を除いたビットが設定されていたら、それらの意味はプラットフォームに依存します。いくつかのプラットフォームではそれらは無視され、それらを設定するためには明示的に *chmod()* を呼ぶ必要があるでしょう。

一時ディレクトリを作成することもできます: *tempfile* モジュールの *tempfile.mkdtemp()* 関数を参照してください。

利用できる環境: Unix、Windows。

`os.makedirs(path[, mode])`

再帰的なディレクトリ作成関数です。 *mkdir()* に似ていますが、末端 (leaf) となるディレクトリを作成するために必要な中間の全てのディレクトリを作成します。末端ディレクトリがすでに存在する場合や、作成ができなかった場合には *error* 例外を送出します。 *mode* の標準の値は 0777 (8 進) です。

mode パラメータは *mkdir()* に渡されます; それがどのように解釈されるかは *mkdir()* の説明を見てください。

注釈: `makedirs()` は作り出すパス要素が `os.pardir` を含むと混乱することになります。

バージョン 1.5.2 で追加.

バージョン 2.3 で変更: この関数は UNC パスを正しく扱えるようになりました .

`os.pathconf(path, name)`

名前付きファイルに関連するシステム設定情報を返します。 `name` には取得したい設定名を指定します ; これは定義済みのシステム値名の文字列で、多くの標準 (POSIX.1、 Unix 95、 Unix 98 その他) で定義されています。 プラットフォームによっては別の名前も定義しています。 ホストオペレーティングシステムの関知する名前は `pathconf_names` 辞書で与えられています。 このマップ型オブジェクトに入っていない設定変数については、 `name` に整数を渡してもかまいません。

もし `name` が文字列でかつ不明である場合、 `ValueError` を送出します。 `name` の指定値がホストシステムでサポートされておらず、 `pathconf_names` にも入っていない場合、 `errno.EINVAL` をエラー番号として `OSError` を送出します。

利用できる環境 : Unix 。

`os.pathconf_names`

`pathconf()` および `fpathconf()` が受理するシステム設定名を、ホストオペレーティングシステムで定義されている整数値に対応付けている辞書です。 この辞書はシステムでどの設定名が定義されているかを決定するために利用できます。 利用出来る環境 : Unix 。

`os.readlink(path)`

シンボリックリンクが指しているパスを表す文字列を返します。 返される値は絶対パスにも、 相対パスにもなり得ます ; 相対パスの場合、 `os.path.join(os.path.dirname(path), result)` を使って絶対パスに変換することができます。

バージョン 2.6 で変更: `path` が unicode オブジェクトだった場合、 戻り値も unicode オブジェクトになります。

利用できる環境 : Unix 。

`os.remove(path)`

ファイル `path` を削除 (消去) します。 `path` がディレクトリの場合、 `OSError` が送出されます ; ディレクトリの削除については `rmdir()` を参照してください。 この関数は下で述べられている `unlink()` 関数と同一です。 Windows では、使用中のファイルを削除しようと試みると例外を送出します ; Unix では、ディレクトリエントリは削除されますが、記憶装置上にアロケーションされたファイル領域は元のファイルが使われなくなるまで残されます。

利用できる環境 : Unix、 Windows 。

`os.removedirs(path)`

再帰的なディレクトリ削除関数です。 `rmdir()` と同じように動作しますが、末端ディレクトリがうまく削除できるかぎり、 `removedirs()` は `path` に現れる親ディレクトリをエラーが送出されるまで (このエラーは通常、指定したディレクトリの親ディレクトリが空でないことを意味するだけなので無視されます) 順に削除することを試みます。 例えば、 `os.removedirs('foo/bar/baz')` では最初

にディレクトリ 'foo/bar/baz' を削除し、次に 'foo/bar' さらに 'foo' をそれらが空ならば削除します。末端のディレクトリが削除できなかった場合には `OSError` が送出されます。

バージョン 1.5.2 で追加。

`os.rename(src, dst)`

ファイルまたはディレクトリ *src* を *dst* に名前変更します。 *dst* がディレクトリの場合、 `OSError` が送出されます。 Unix では、 *dst* が存在し、かつファイルの場合、ユーザの権限があるかぎり暗黙のうちに元のファイルが置き換えられます。この操作はいくつかの Unix 系システムにおいて、 *src* と *dst* が異なるファイルシステム上にあると失敗することがあります。ファイル名の変更が成功する場合、この操作は原子的 (atomic) 操作となります (これは POSIX 要求仕様です)。Windows では、 *dst* が既に存在する場合には、たとえファイルの場合でも `OSError` が送出されます ; これは *dst* が既に存在するファイル名の場合、名前変更の原子的操作を実装する手段がないからです。

利用できる環境 : Unix、Windows。

`os.rename(old, new)`

再帰的にディレクトリやファイル名を変更する関数です。 `rename()` のように動作しますが、新たなパス名を持つファイルを配置するために必要な途中のディレクトリ構造をまず作成しようと試みます。名前変更の後、元のファイル名のパス要素は `removedirs()` を使って右側から順に削除されます。

バージョン 1.5.2 で追加。

注釈: この関数はコピー元の末端のディレクトリまたはファイルを削除する権限がない場合には失敗します。

`os.rmdir(path)`

ディレクトリ *path* を削除します。ディレクトリが空の場合にだけ正常に動作します。そうでなければ `OSError` が送出されます。ディレクトリツリー全体を削除するには `shutil.rmtree()` を使います。

利用できる環境 : Unix、Windows。

`os.stat(path)`

与えられた *path* に対して `stat()` システムコール相当の処理を実行します。(この関数はシンボリックリンクをたどります。シンボリックリンクに対して `stat` したい場合は `lstat()` を利用してください)

返り値はオブジェクトになり、その属性はおおむね `stat` 構造体のメンバーに対応します:

- `st_mode` - 保護ビット
- `st_ino` - inode 番号
- `st_dev` - デバイス
- `st_nlink` - ハードリンクの数
- `st_uid` - 所有者のユーザー id

- `st_gid` - 所有者のグループ id
- `st_size` - ファイルのサイズ (単位: バイト)
- `st_atime` - 最近にアクセスされた時間,
- `st_mtime` - 最近に内容を変更した時間,
- `st_ctime` - プラットフォーム依存; Unix では最近のメタデータ変更時間、Windows ではファイルが生成された時間

バージョン 2.3 で変更: もし `stat_float_times()` が `True` を返す場合、時間値は浮動小数点で秒を計ります。ファイルシステムがサポートしていれば、秒の小数点以下の桁も含めて返されます。詳細な説明は `stat_float_times()` を参照してください。

(Linux のような) 一部の Unix システムでは、以下の属性が利用できる場合があります:

- `st_blocks` - ファイルの大きさを 512 バイトのブロックサイズ単位で示す
- `st_blksize` - 効率的にファイルシステム I/O ができる好ましいファイルシステムのブロックサイズ
- `st_rdev` - i ノードデバイスの場合、デバイスのタイプ
- `st_flags` - ファイルに対するユーザー定義のフラグ

他の (FreeBSD のような) Unix システムでは、以下の属性が利用できる場合があります (ただし root ユーザ以外が使うと値が入っていない場合があります):

- `st_gen` - ファイル生成番号
- `st_birthtime` - ファイル作成時刻

RISCOS システムでは、以下の属性も利用できます:

- `st_ftime` (file type),
- `st_attrs` (attributes),
- `st_obtype` (object type)

注釈: `st_atime`、`st_mtime`、および `st_ctime` 属性の厳密な意味や精度はオペレーティングシステムやファイルシステムによって変わります。例えば、FAT や FAT32 ファイルシステムを使用している Windows システムでは、`st_mtime` の精度は 2 秒であり、`st_atime` の精度は 1 日に過ぎません。詳しくはお使いのオペレーティングシステムのドキュメントを参照してください。

後方互換性のため、`stat()` の戻り値は `stat` 構造体の最も重要な (そして移植性の高い) メンバーを表すタプルとしてもアクセス可能です。これは少なくとも 10 個の整数からなり、`st_mode`、`st_ino`、`st_dev`、`st_nlink`、`st_uid`、`st_gid`、`st_size`、`st_atime`、`st_mtime`、`st_ctime` の順になります。実装によってはそれ以上のアイテムが末尾に追加されます。

標準モジュール `stat` は `stat` 構造体からの情報の取り出しに役立つ関数と定数を定義しています。(Windows では、一部のアイテムにダミー値が入ります)

例

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
(33188, 422511, 769, 1, 1032, 100, 926, 1105022698, 1105022732, 1105022732)
>>> statinfo.st_size
926
```

利用できる環境: Unix、Windows。

バージョン 2.2 で変更: 値へのアクセスを戻り値の属性として追加しました。

バージョン 2.5 で変更: `st_gen` および `st_birthtime` 属性が追加されました。

`os.stat_float_times([newvalue])`

`stat_result` がタイムスタンプに浮動小数点オブジェクトを使うかどうかを決定します。 *newvalue* が `True` の場合、以後の `stat()` 呼び出しは浮動小数点を返し、`False` の場合には以後整数を返します。 *newvalue* が省略された場合、現在の設定どおりの戻り値になります。

古いバージョンの Python と互換性を保つため、`stat_result` にタプルとしてアクセスすると、常に整数が返されます。

バージョン 2.5 で変更: Python は現在デフォルトで浮動小数点値を返します。タイムスタンプが浮動小数点では正常に動作しないアプリケーションは、この関数で古い挙動を利用できます。

タイムスタンプの精度 (すなわち最小の小数部分) はシステム依存です。システムによっては秒単位の精度しかサポートしません。そういったシステムでは小数部分は常に 0 です。

この設定の変更は、プログラムの起動時に、`__main__` モジュールの中でのみ行うことを推奨します。ライブラリは決してこの設定を変更すべきではありません。浮動小数点型のタイムスタンプを処理すると不正確な動作をするようなライブラリを使う場合、ライブラリが修正されるまで、その機能を停止させておくべきです。

`os.statvfs(path)`

与えられたパスに対して `statvfs()` システムコールを実行します。戻り値はオブジェクトで、その属性は与えられたパスが格納されているファイルシステムについて記述したものです。各属性は `statvfs` 構造体のメンバーに対応します: `f_bsize`, `f_frsize`, `f_blocks`, `f_bfree`, `f_bavail`, `f_files`, `f_ffree`, `f_favail`, `f_flag`, `f_namemax`。

後方互換性のために、戻り値は上の順にそれぞれ対応する属性値が並んだタプルとしてアクセスすることもできます。標準モジュール `statvfs` では、シーケンスとしてアクセスする場合に、`statvfs` 構造体から情報を引き出す上便利な関数や定数を定義しています; これは属性として各フィールドにアクセスできないバージョンの Python で動作する必要のあるコードを書く際に便利です。

利用できる環境: Unix。

バージョン 2.2 で変更: 値へのアクセスを戻り値の属性として追加しました。

`os.symlink(source, link_name)`

`source` を指すシンボリックリンク `link_name` を作成します。

利用できる環境 : Unix。

`os.tempnam([dir[, prefix]])`

一時ファイル (temporary file) を生成する上でファイル名として相応しい一意なパス名を返します。この値は一時的なディレクトリエントリを表す絶対パスで、`dir` ディレクトリの下か、`dir` が省略されたり `None` の場合には一時ファイルを置くための共通のディレクトリの下になります。`prefix` が与えられており、かつ `None` でない場合、ファイル名の先頭につけられる短い接頭辞になります。アプリケーションは `tempnam()` が返したパス名を使って正しくファイルを生成し、生成したファイルを管理する責任があります; 一時ファイルの自動消去機能は提供されていません。Unix では、環境変数 `TMPDIR` が `dir` を上書きし、Windows では環境変数 `TMP` が使われます。これら関数の固有の振る舞いについては C ライブラリの実装に依存しています; いくつかの側面についてはシステムのドキュメントで曖昧です。

警告: `tempnam()` を使うと、symlink 攻撃に対して脆弱になります; 代りに `tmpfile()` (ファイルオブジェクトの生成) を使うよう検討してください。

利用できる環境 : Unix、Windows。

`os.tmpnam()`

一時ファイル (temporary file) を生成する上でファイル名として相応しい一意なパス名を返します。この値は一時ファイルを置くための共通のディレクトリ下の一時的なディレクトリエントリを表す絶対パスです。アプリケーションは `tmpnam()` が返したパス名を使って正しくファイルを生成し、生成したファイルを管理する責任があります; 一時ファイルの自動消去機能は提供されていません。

警告: `tmpnam()` を使うと、symlink 攻撃に対して脆弱になります; 代りに `tmpfile()` (ファイルオブジェクトの生成) を使うよう検討してください。

利用できる環境 : Unix, Windows。この関数はおそらく Windows では使うべきではないでしょう; Microsoft の `tmpnam()` 実装では、常に現在のドライブのルートディレクトリ下のファイル名を生成しますが、これは一般的にはテンポラリファイルを置く場所としてはひどい場所です (アクセス権限によっては、この名前をつかってファイルを開くことすらできないかもしれません)。

`os.TMP_MAX`

`tmpnam()` がテンポラリ名を再利用し始めるまでに生成できる一意な名前の最大数です。

`os.unlink(path)`

ファイル `path` を削除します。 `remove()` と同じです; `unlink()` の名前は伝統的な Unix の関数名です。

利用できる環境 : Unix、Windows。

`os.utime(path, times)`

`path` で指定されたファイルに最終アクセス時刻および最終修正時刻を設定します。 `times` が `None` の

場合、ファイルの最終アクセス時刻および最終更新時刻は現在の時刻になります。(この動作は、その *path* に対して Unix の `touch` プログラムを実行するのに似ています)。そうでない場合、*times* は 2 要素のタプルで、(`atime`, `mtime`) の形式をとらなくてはなりません。これらはそれぞれアクセス時刻および修正時刻を設定するために使われます。*path* にディレクトリを指定できるかどうかは、オペレーティングシステムがディレクトリをファイルの一種として実装しているかどうかに依存します(例えば、Windows はそうではありません)。ここで設定した時刻の値は、オペレーティングシステムがアクセス時刻や更新時刻を記録する際の精度によっては、後で `stat()` 呼び出したときの値と同じにならないかも知れないので注意してください。`stat()` も参照してください。

バージョン 2.0 で変更: *times* として `None` をサポートするようにしました。

利用できる環境: Unix、Windows。

`os.walk(top, topdown=True, onerror=None, followlinks=False)`

ディレクトリツリー以下のファイル名を、ツリーをトップダウンもしくはボトムアップに走査することで作成します。ディレクトリ *top* を根に持つディレクトリツリーに含まれる、各ディレクトリ (*top* 自身を含む) ごとに、タプル (`dirpath`, `dirnames`, `filenames`) を `yield` します。

dirpath は文字列で、ディレクトリへのパスです。*dirnames* は *dirpath* 内のサブディレクトリ名のリスト ('.' と '..' は除く) です。*filenames* は *dirpath* 内の非ディレクトリ・ファイル名のリストです。このリスト内の名前にはファイル名までのパスが含まれません。*dirpath* 内のファイルやディレクトリへの (*top* からたどった) フルパスを得るには、`os.path.join(dirpath, name)` を使用してください。

オプション引数 *topdown* が `True` であるか、指定されなかった場合、各ディレクトリからタプルを生成した後で、サブディレクトリからタプルを生成します。(ディレクトリはトップダウンで生成)。*topdown* が `False` の場合、ディレクトリに対応するタプルは、そのディレクトリ以下の全てのサブディレクトリに対応するタブルの後で (ボトムアップで) 生成されます。*topdown* の値によらず、サブディレクトリのリストは、ディレクトリとそのサブディレクトリのタプルを生成する前に取り出されます。

topdown が `True` のとき、呼び出し側は *dirnames* リストを、インプレースで (たとえば、`del` やスライスを使った代入で) 変更でき、`walk()` は *dirnames* に残っているサブディレクトリ内のみを再帰します。これにより、検索を省略したり、特定の訪問順序を強制したり、呼び出し側が `walk()` を再開する前に、呼び出し側が作った、または名前を変更したディレクトリを、`walk()` に知らせたりすることができます。*topdown* が `False` のときに *dirnames* を変更しても効果はありません。ボトムアップモードでは *dirpath* 自身が生成される前に *dirnames* 内のディレクトリの情報が生成されるからです。

デフォルトでは、`listdir()` 呼び出しからのエラーは無視されます。オプション引数の *onerror* を指定する場合は関数でなければなりません; この関数は単一の引数として `OSError` インスタンスを伴って呼び出されます。この関数でエラーを報告して走査を継続したり、例外を送出して走査を中止したりできます。ファイル名は例外オブジェクトの `filename` 属性として利用できます。

デフォルトでは、`walk()` はディレクトリへのシンボリックリンクをたどりません。*followlinks* に `True` を指定すると、ディレクトリへのシンボリックリンクをサポートしているシステムでは、シンボリックリンクの指しているディレクトリを走査します。

バージョン 2.6 で追加: *followlinks* 引数

注釈: `followlinks` に `True` を指定すると、シンボリックリンクが親ディレクトリを指していた場合に、無限ループになることに注意してください。 `walk()` はすでにたどったディレクトリを管理したりはしません。

注釈: 相対パスを渡した場合、 `walk()` が再開されるまでの間に現在の作業ディレクトリを変更しないでください。 `walk()` はカレントディレクトリを変更しませんし、呼び出し側もカレントディレクトリを変更しないと仮定しています。

以下の例では、最初のディレクトリ以下にある各ディレクトリに含まれる、非ディレクトリファイルのバイト数を表示します。ただし、CVS サブディレクトリ以下は見に行きません

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print root, "consumes",
    print sum(getsize(join(root, name)) for name in files),
    print "bytes in", len(files), "non-directory files"
    if 'CVS' in dirs:
        dirs.remove('CVS') # don't visit CVS directories
```

次の例では、ツリーをボトムアップで走査することが不可欠になります ; `rmdir()` はディレクトリが空になるまで削除を許さないからです

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION: This is dangerous! For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
```

バージョン 2.3 で追加.

15.1.5 プロセス管理

以下の関数はプロセスの生成や管理に利用できます。

さまざまな `exec*` 関数は、プロセス内にロードされる新しいプログラムに与えるための、引数のリストを取ります。どの関数の場合でも、新しいプログラムに渡されるリストの最初の引数は、ユーザがコマンドラインで入力する引数ではなく、そのプログラム自体の名前です。C プログラマならば、プログラムの `main()` に渡される `argv[0]` だと考えれば良いでしょう。たとえば、`os.execv('/bin/echo', ['foo', 'bar'])`

が標準出力に出力するのは `bar` だけで、`foo` は無視されたかのように見えることになります。

`os.abort()`

`SIGABRT` シグナルを現在のプロセスに対して生成します。Unix では、デフォルトの動作はコアダンプの生成です；Windows では、プロセスは即座に終了コード 3 を返します。この関数の呼び出しは `signal.signal()` を使って `SIGABRT` に対し登録された Python シグナルハンドラーを呼び出さないことに注意してください。

利用できる環境：Unix、Windows。

`os.exec1(path, arg0, arg1, ...)`

`os.execle(path, arg0, arg1, ..., env)`

`os.execlp(file, arg0, arg1, ...)`

`os.execlpe(file, arg0, arg1, ..., env)`

`os.execv(path, args)`

`os.execve(path, args, env)`

`os.execvp(file, args)`

`os.execvpe(file, args, env)`

これらの関数はすべて、現在のプロセスを置き換える形で新たなプログラムを実行します；現在のプロセスは返り値を返しません。Unix では、新たに実行される実行コードは現在のプロセス内に読み込まれ、呼び出し側と同じプロセス ID を持つことになります。エラーは `OSError` 例外として報告されます。

現在のプロセスは瞬時に置き換えられます。開かれているファイルオブジェクトやファイル記述子はフラッシュされません。そのため、バッファ内にデータが残っているかもしれない場合、`exec*` 関数を実行する前に `sys.stdout.flush()` か `os.fsync()` を利用してバッファをフラッシュしておく必要があります。

”l” および ”v” のついた `exec*` 関数は、コマンドライン引数をどのように渡すかが異なります。”l” 型は、コードを書くときにパラメタ数が決まっている場合に、おそらくもっとも簡単に利用できます。個々のパラメタは単に `exec1*()` 関数の追加パラメタとなります。”v” 型は、パラメタの数が可変の時に便利で、リストかタプルの引数が `args` パラメタとして渡されます。どちらの場合も、子プロセスに渡す引数は動作させようとしているコマンドの名前から始まるべきですが、これは強制されません。

末尾近くに ”p” をもつ型 (`execlp()`, `execlpe()`, `execvp()`, および `execvpe()`) は、プログラム `file` を探すために環境変数 `PATH` を利用します。環境変数が (次の段で述べる `exec*e` 型関数で) 置き換えられる場合、環境変数は `PATH` を決定する上の情報源として使われます。その他の型、`exec1()`, `execle()`, `execv()`, および `execve()` では、実行コードを探すために `PATH` を使いません。 `path` には適切に設定された絶対パスまたは相対パスが入っていないてはなりません。

`execle()`、`execlpe()`、`execve()`、および `execvpe()` (すべて末尾に ”e” がついていますが) では、`env` パラメーターは新たなプロセスで利用される環境変数を定義するためのマップ型でなくてはなりません (現在のプロセスの環境変数の代わりに利用されます)；`exec1()`、`execlp()`、`execv()`、および `execvp()` では、すべて新たなプロセスは現在のプロセスの環境を引き継ぎます。

利用できる環境：Unix、Windows。

`os._exit(n)`

終了ステータス n でプロセスを終了します。この時クリーンアップハンドラーの呼び出しや、標準入力バッファのフラッシュなどはいりません。

利用できる環境: Unix、Windows。

注釈: 終了する標準的な方法は `sys.exit(n)` です。 `_exit()` は通常、 `fork()` された後の子プロセスでのみ使われます。

以下の終了コードは必須ではありませんが `_exit()` で使うことができます。一般に、メールサーバーの外部コマンド配送プログラムのような、Python で書かれたシステムプログラムに使います。

注釈: いくつかのバリエーションがあって、これらのすべてがすべての Unix プラットフォームで使えるわけではありません。以下の定数は下層のプラットフォームで定義されていれば定義されます。

OS.**EX_OK**

エラーが起きなかったことを表す終了コード。

利用できる環境: Unix。

バージョン 2.3 で追加。

OS.**EX_USAGE**

誤った個数の引数が渡された時など、コマンドが間違っ使われたことを表す終了コード。

利用できる環境: Unix。

バージョン 2.3 で追加。

OS.**EX_DATAERR**

入力データが誤っていたことを表す終了コード。

利用できる環境: Unix。

バージョン 2.3 で追加。

OS.**EX_NOINPUT**

入力ファイルが存在しなかった、または、読み込み不可だったことを表す終了コード。

利用できる環境: Unix。

バージョン 2.3 で追加。

OS.**EX_NOUSER**

指定されたユーザーが存在しなかったことを表す終了コード。

利用できる環境: Unix。

バージョン 2.3 で追加。

os.EX_NOHOST

指定されたホストが存在しなかったことを表す終了コード。

利用できる環境 : Unix 。

バージョン 2.3 で追加.

os.EX_UNAVAILABLE

要求されたサービスが利用できないことを表す終了コード。

利用できる環境 : Unix 。

バージョン 2.3 で追加.

os.EX_SOFTWARE

内部ソフトウェアエラーが検出されたことを表す終了コード。

利用できる環境 : Unix 。

バージョン 2.3 で追加.

os.EX_OSERR

fork できない、pipe の作成ができないなど、オペレーティングシステムのエラーが検出されたことを表す終了コード。

利用できる環境 : Unix 。

バージョン 2.3 で追加.

os.EX_OSFILE

システムファイルが存在しなかった、開けなかった、あるいはその他のエラーが起きたことを表す終了コード。

利用できる環境 : Unix 。

バージョン 2.3 で追加.

os.EX_CANTCREAT

ユーザーには作成できない出力ファイルを指定したことを表す終了コード。

利用できる環境 : Unix 。

バージョン 2.3 で追加.

os.EX_IOERR

ファイルの I/O を行っている途中にエラーが発生した時の終了コード。

利用できる環境 : Unix 。

バージョン 2.3 で追加.

os.EX_TEMPFAIL

一時的な失敗が発生したことを表す終了コード。これは、再試行可能な操作の途中に、ネットワークに接続できないというような、実際にはエラーではないかも知れないことを意味します。

利用できる環境 : Unix 。

バージョン 2.3 で追加.

os.EX_PROTOCOL

プロトコル交換が不正、不適切、または理解不能なことを表す終了コード。

利用できる環境 : Unix 。

バージョン 2.3 で追加.

os.EX_NOPERM

操作を行うために十分な許可がなかった (ファイルシステムの問題を除く) ことを表す終了コード。

利用できる環境 : Unix 。

バージョン 2.3 で追加.

os.EX_CONFIG

設定エラーが起こったことを表す終了コード。

利用できる環境 : Unix 。

バージョン 2.3 で追加.

os.EX_NOTFOUND

”an entry was not found” のようなことを表す終了コード。

利用できる環境 : Unix 。

バージョン 2.3 で追加.

os.fork()

子プロセスを fork します。子プロセスでは 0 が返り、親プロセスでは子プロセスの id が返ります。エラーが発生した場合は、*OSError* を送出します。

FreeBSD 6.3 以下、Cygwin、OS/2 EMX を含む一部のプラットフォームにおいて、fork() をスレッド内から利用した場合に既知の問題があることに注意してください。

警告: SSL モジュールを fork() とともに使うアプリケーションについて、*ssl* を参照して下さい。

利用できる環境 : Unix 。

os.forkpty()

子プロセスを fork します。この時新しい擬似端末を子プロセスの制御端末として使います。親プロセスでは (pid, fd) からなるペアが返り、fd は擬似端末のマスター側のファイル記述子となります。可搬性のあるアプローチを取るには、*pty* モジュールを利用してください。エラーが発生した場合は、*OSError* を送出します。

利用できる環境 : 一部の Unix 互換環境。

`os.kill(pid, sig)`

プロセス *pid* にシグナル *sig* を送ります。ホストプラットフォームで利用可能なシグナルを特定する定数は *signal* モジュールで定義されています。

Windows: *signal.CTRL_C_EVENT* と *signal.CTRL_BREAK_EVENT* は、同じコンソールウィンドウを共有しているコンソールプロセス (例: 子プロセス) にだけ送ることができる特別なシグナルです。その他の値を *sig* に与えると、そのプロセスが無条件に `TerminateProcess` API によって kill され、終了コードが *sig* に設定されます。Windows の *kill()* は kill するプロセスのハンドルも受け取ります。

バージョン 2.7 で追加: Windows サポート

`os.killpg(pgid, sig)`

プロセスグループ *pgid* にシグナル *sig* を送ります。

利用できる環境: Unix。

バージョン 2.3 で追加.

`os.nice(increment)`

プロセスの "nice 値" に *increment* を加えます。新たな nice 値を返します。

利用できる環境: Unix。

`os.plock(op)`

プログラムのセグメントをメモリ内にロックします。 *op* (`<sys/lock.h>` で定義されています) にはどのセグメントをロックするかを指定します。

利用できる環境: Unix。

`os.popen(...)`

`os.popen2(...)`

`os.popen3(...)`

`os.popen4(...)`

子プロセスを起動し、子プロセスとの通信のために開かれたパイプを返します。これらの関数は [ファイルオブジェクトの生成](#) 節で説明されています。

`os.spawnl(mode, path, ...)`

`os.spawnle(mode, path, ..., env)`

`os.spawnlp(mode, file, ...)`

`os.spawnlpe(mode, file, ..., env)`

`os.spawnv(mode, path, args)`

`os.spawnve(mode, path, args, env)`

`os.spawnvp(mode, file, args)`

`os.spawnvpe(mode, file, args, env)`

新たなプロセス内でプログラム *path* を実行します。

(*subprocess* モジュールが、新しいプロセスを実行して結果を取得するための、より強力な機能を提供しています。この関数の代わりに *subprocess* モジュールを利用することが推奨されています。[subprocess](#) モジュールのドキュメントの、 [古い関数を subprocess モジュールで置き換える](#) セクションを参照してください)

`mode` が `P_NOWAIT` の場合、この関数は新たなプロセスのプロセス ID を返します；`mode` が `P_WAIT` の場合、子プロセスが正常に終了するとその終了コードが返ります。そうでない場合にはプロセスを `kill` したシグナル `signal` に対して `-signal` が返ります。Windows では、プロセス ID は実際にはプロセスハンドル値になるので、`waitpid()` 関数で使えます。

”l” および ”v” のついた `spawn*` 関数は、コマンドライン引数をどのように渡すかが異なります。”l” 型は、コードを書くときにパラメタ数が決まっている場合に、おそらくもっとも簡単に利用できます。個々のパラメタは単に `spawnl*()` 関数の追加パラメタとなります。 ”v” 型は、パラメタの数が可変の時に便利で、リストかタプルの引数が `args` パラメタとして渡されます。どちらの場合も、子プロセスに渡す引数は動作させようとしているコマンドの名前から始まらなければなりません。

末尾近くに ”p” をもつ型 (`spawnlp()`, `spawnlpe()`, `spawnvp()`, `spawnvpe()`) は、プログラム `file` を探すために環境変数 `PATH` を利用します。環境変数が (次の段で述べる `spawn*e` 型関数で) 置き換えられる場合、環境変数は `PATH` を決定する上の情報源として使われます。その他の型、`spawnl()`, `spawnle()`, `spawnv()`, および `spawnve()` では、実行コードを探すために `PATH` を使いません。 `path` には適切に設定された絶対パスまたは相対パスが入っていないとなりません。

`spawnle()`, `spawnlpe()`, `spawnve()`, および `spawnvpe()` (すべて末尾に ”e” がついています) では、`env` パラメタは新たなプロセスで利用される環境変数を定義するためのマップ型でなくてはなりません；`spawnl()`、`spawnlp()`、`spawnv()`、および `spawnvp()` では、すべて新たなプロセスは現在のプロセスの環境を引き継ぎます。 `env` 辞書のキーと値はすべて文字列である必要があります。不正なキーや値を与えると関数が失敗し、127 を返します。

例えば、以下の `spawnlp()` および `spawnvpe()` 呼び出しは等価です

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

利用できる環境：Unix、Windows `spawnlp()`、`spawnlpe()`、`spawnvp()`、および `spawnvpe()` は Windows では利用できません。 `spawnle()` および `spawnve()` は Windows においてスレッドセーフではありません；代わりに `subprocess` モジュールの利用を推奨します。

バージョン 1.6 で追加。

os.P_NOWAIT

os.P_NOWAITO

`spawn*` 関数ファミリに対する `mode` パラメタとして取れる値です。この値のいずれかを `mode` として与えた場合、`spawn*()` 関数は新たなプロセスが生成されるとすぐに、プロセスの ID を戻り値として返ります。

利用できる環境：Unix、Windows。

バージョン 1.6 で追加。

os.P_WAIT

`spawn*` 関数ファミリに対する `mode` パラメタとして取れる値です。この値を `mode` として与えた場

合、`spawn*()` 関数は新たなプロセスを起動して完了するまで返らず、プロセスがうまく終了した場合には終了コードを、シグナルによってプロセスが `kill` された場合には `-signal` を返します。

利用できる環境: Unix、Windows。

バージョン 1.6 で追加。

`os.P_DETACH`

`os.P_OVERLAY`

`spawn*` 関数ファミリに対する `mode` パラメタとして取れる値です。これらの値は上の値よりもやや可搬性において劣っています。`P_DETACH` は `P_NOWAIT` に似ていますが、新たなプロセスは呼び出しプロセスのコンソールから切り離され (detach) ます。`P_OVERLAY` が使われた場合、現在のプロセスは置き換えられます; 従って `spawn*()` は返りません。

利用出来る環境: Windows.

バージョン 1.6 で追加。

`os.startfile(path[, operation])`

ファイルを関連付けられたアプリケーションを使ってスタートします。

`operation` が指定されないか、または `'open'` である時、この動作は、Windows の Explorer 上でのファイルをダブルクリックした、あるいはコマンドプロンプト上でファイル名を `start` コマンドの引数としての実行した場合と等価です: ファイルは拡張子が関連付けされているアプリケーション (が存在する場合) を使って開かれます。

他の `operation` が与えられる場合、それはファイルに対して何がなされるべきかを表す "command verb" (コマンドを表す動詞) でなければなりません。Microsoft が文書化している動詞は、`'print'` と `'edit'` (ファイルに対して) および `'explore'` と `'find'` (ディレクトリに対して) です。

`startfile()` は関連付けされたアプリケーションが起動すると同時に返ります。アプリケーションが閉じるまで待機させるためのオプションはなく、アプリケーションの終了状態を取得する方法もありません。引数 `path` はカレントディレクトリからの相対パスです。絶対パスで指定したい場合は、最初の文字はスラッシュ (`'/'`) ではないので注意してください; もし最初の文字がスラッシュなら、下層の `Win32 ShellExecute()` 関数は動作しません。`os.path.normpath()` 関数を使って、Win32 用に正しくコード化されたパスになるようにしてください。

利用出来る環境: Windows.

バージョン 2.0 で追加。

バージョン 2.5 で追加: `operation` パラメータ。

`os.system(command)`

サブシェル内でコマンド (文字列) を実行します。この関数は標準 C 関数 `system()` を使って実装されており、`system()` と同じ制限があります。`sys.stdin` などに対する変更を行っても、実行されるコマンドの環境には反映されません。

Unix では、返り値はプロセスの終了ステータスで、`wait()` で定義されている書式にコード化されています。POSIX は `system()` 関数の返り値の意味について定義していないので、Python の `system()` における返り値はシステム依存となることに注意してください。

Windows では、戻り値は `command` を実行した後にシステムシェルから返される値で、Windows の環境変数 `COMSPEC` となります：`command.com` ベースのシステム (Windows 95, 98 および ME) では、この値は常に 0 です；`cmd.exe` ベースのシステム (Windows NT, 2000 および XP) では、この値は実行したコマンドの終了ステータスです；ネイティブでないシェルを使っているシステムについては、使っているシェルのドキュメントを参照してください。

`subprocess` モジュールは、新しいプロセスを実行して結果を取得するためのより強力な機能を提供しています。この関数の代わりに `subprocess` モジュールを利用することが推奨されています。`subprocess` モジュールのドキュメントの [古い関数を subprocess モジュールで置き換える](#) 節のレシピを参考にして下さい。

利用できる環境：Unix、Windows。

`os.times()`

(プロセッサまたはその他の) 積算時間を秒で表す浮動小数点数からなる、5 要素のタプルを返します。タプルの要素は、ユーザ時間 (user time)、システム時間 (system time)、子プロセスのユーザ時間、子プロセスのシステム時間、そして過去のある固定時点からの経過時間で、この順に並んでいます。Unix マニュアルページ `times(2)` または対応する Windows プラットフォーム API ドキュメントを参照してください。Windows では、最初の 2 つの要素だけが埋められ、残りは 0 になります。

利用できる環境：Unix, Windows

`os.wait()`

子プロセスの実行完了を待機し、子プロセスの `pid` と終了コードインジケータ — 16 ビットの数値で、下位バイトがプロセスを `kill` したシグナル番号、上位バイトが終了ステータス (シグナル番号がゼロの場合) — の入ったタプルを返します；コアダンプファイルが生成された場合、下位バイトの最上桁ビットが立てられます。

利用できる環境：Unix。

`os.waitpid(pid, options)`

この関数の詳細は Unix と Windows で異なります。

Unix の場合：プロセス id `pid` で与えられた子プロセスの完了を待機し、子プロセスのプロセス id と (`wait()` と同様にコード化された) 終了ステータスインジケータからなるタプルを返します。この関数の動作は `options` によって変わります。通常の操作では 0 にします。

`pid` が 0 よりも大きい場合、`waitpid()` は特定のプロセスのステータス情報を要求します。`pid` が 0 の場合、現在のプロセスグループ内の任意の子プロセスの状態に対する要求です。`pid` が -1 の場合、現在のプロセスの任意の子プロセスに対する要求です。`pid` が -1 よりも小さい場合、プロセスグループ `-pid` (すなわち `pid` の絶対値) 内の任意のプロセスに対する要求です。

システムコールが -1 を返した時、`OSError` を `errno` と共に送出します。

Windows では、プロセスハンドル `pid` を指定してプロセスの終了を待って、`pid` と、終了ステータスを 8bit 左シフトした値のタプルを返します。(シフトは、この関数をクロスプラットフォームで利用しやすくするために行われます) 0 以下の `pid` は Windows では特別な意味を持っておらず、例外を発生させます。`options` の値は効果がありません。`pid` は、子プロセスでなくても、プロセス ID を知ってい

どんなプロセスでも参照することが可能です。 `spawn*` 関数を `P_NOWAIT` と共に呼び出した場合、適切なプロセスハンドルが返されます。

os.wait3 (options)

`waitpid()` に似ていますが、プロセス id を引数に取らず、子プロセス id、終了ステータスインジケータ、リソース使用情報の 3 要素からなるタプルを返します。リソース使用情報の詳しい情報は `resource.getrusage()` を参照してください。オプション引数は `waitpid()` および `wait4()` と同じです。

利用できる環境 : Unix。

バージョン 2.5 で追加.

os.wait4 (pid, options)

`waitpid()` に似ていますが、子プロセス id、終了ステータスインジケータ、リソース使用情報の 3 要素からなるタプルを返します。リソース使用情報の詳しい情報は `resource.getrusage()` を参照してください。 `wait4()` の引数は `waitpid()` に与えられるものと同じです。

利用できる環境 : Unix。

バージョン 2.5 で追加.

os.WNOHANG

子プロセス状態がすぐに取得できなかった場合に直ちに終了するようにするための `waitpid()` のオプションです。この場合、関数は (0, 0) を返します。

利用できる環境 : Unix。

os.WCONTINUED

このオプションによって子プロセスは前回状態が報告された後にジョブ制御による停止状態から実行を再開された場合に報告されるようになります。

利用できる環境 : ある種の Unix システム。

バージョン 2.3 で追加.

os.WUNTRACED

このオプションによって子プロセスは停止されていながら停止されてから状態が報告されていない場合に報告されるようになります。

利用できる環境 : Unix。

バージョン 2.3 で追加.

以下の関数は `system()`、`wait()`、あるいは `waitpid()` が返すプロセス状態コードを引数にとります。これらの関数はプロセスの配置を決めるために利用できます。

os.WCOREDUMP (status)

プロセスに対してコアダンプが生成されていた場合には `True` を、それ以外の場合は `False` を返します。

利用できる環境 : Unix。

バージョン 2.3 で追加。

os.WIFCONTINUED (*status*)

プロセスがジョブ制御による停止状態から実行を再開された (continue) 場合に True を、それ以外の場合は False を返します。

利用できる環境 : Unix 。

バージョン 2.3 で追加。

os.WIFSTOPPED (*status*)

プロセスが停止された (stop) 場合に True を、それ以外の場合は False を返します。

利用できる環境 : Unix 。

os.WIFSIGNALED (*status*)

プロセスがシグナルによって終了した (exit) 場合に True を、それ以外の場合は False を返します。

利用できる環境 : Unix 。

os.WIFEXITED (*status*)

プロセスが `exit(2)` システムコールで終了した場合に True を、それ以外の場合は False を返します。

利用できる環境 : Unix 。

os.WEXITSTATUS (*status*)

WIFEXITED(*status*) が真の場合、`exit(2)` システムコールに渡された整数パラメーターを返します。そうでない場合、返される値には意味がありません。

利用できる環境 : Unix 。

os.WSTOPSIG (*status*)

プロセスを停止させたシグナル番号を返します。

利用できる環境 : Unix 。

os.WTERMSIG (*status*)

プロセスを終了させたシグナル番号を返します。

利用できる環境 : Unix 。

15.1.6 雑多なシステム情報

os.confstr (*name*)

システム設定値を文字列で返します。 *name* には取得したい設定名を指定します ; この値は定義済みのシステム値名を表す文字列にすることができます ; 名前は多くの標準 (POSIX.1、 Unix 95、 Unix 98 その他) で定義されています。ホストオペレーティングシステムの関知する名前は `confstr_names` 辞書のキーとして与えられています。このマップ型オブジェクトに入っていない設定変数については、 *name* に整数を渡してもかまいません。

`name` に指定された設定値が定義されていない場合、`None` を返します。

`name` が文字列で、かつ不明の場合、`ValueError` を送出します。`name` の指定値がホストシステムでサポートされておらず、`confstr_names` にも入っていない場合、`errno.EINVAL` をエラー番号として `OSError` を送出します。

利用できる環境 : Unix

`os.confstr_names`

`confstr()` が受理する名前を、ホストオペレーティングシステムで定義されている整数値に対応付けている辞書です。この辞書はシステムでどの設定名が定義されているかを決定するために利用できます。

利用できる環境 : Unix。

`os.getloadavg()`

過去 1 分、5 分、および 15 分間の、システムの実行キューの平均プロセス数を返します。平均負荷が得られない場合には `OSError` を送出します。

利用できる環境 : Unix。

バージョン 2.3 で追加。

`os.sysconf(name)`

整数値のシステム設定値を返します。`name` で指定された設定値が定義されていない場合、`-1` が返されます。`name` に関するコメントとしては、`confstr()` で述べた内容が同様に当てはまります；既知の設定名についての情報を与える辞書は `sysconf_names` で与えられています。

利用できる環境 : Unix。

`os.sysconf_names`

`sysconf()` が受理する名前を、ホストオペレーティングシステムで定義されている整数値に対応付けている辞書です。この辞書はシステムでどの設定名が定義されているかを決定するために利用できます。

利用できる環境 : Unix。

以下のデータ値はパス名編集操作をサポートするために利用されます。これらの値はすべてのプラットフォームで定義されています。

パス名に対する高レベルの操作は `os.path` モジュールで定義されています。

`os.curdir`

現在のディレクトリ参照するためにオペレーティングシステムで使われる文字列定数です。POSIX と Windows では `'.'` になります。`os.path` から利用できます。

`os.pardir`

親ディレクトリを参照するためにオペレーティングシステムで使われる文字列定数です。POSIX と Windows では `'..'` になります。`os.path` から利用できます。

`os.sep`

パス名を要素に分割するためにオペレーティングシステムで利用されている文字です。例えば POSIX

では '/' で、Windows では '\\\\' です。しかし、このことを知っているだけではパス名を解析したり、パス名同士を結合したりするには不十分です — こうした操作には `os.path.split()` や `os.path.join()` を使用してください — が、たまに便利なこともあります。 `os.path` から利用できます。

`os.altsep`

文字パス名を要素に分割する際にオペレーティングシステムで利用されるもう一つの文字で、分割文字が一つしかない場合には `None` になります。この値は `sep` がバックスラッシュとなっている DOS や Windows システムでは '/' に設定されています。 `os.path` から利用できます。

`os.extsep`

ベースのファイル名と拡張子を分ける文字です。例えば、 `os.py` であれば '.' です。 `os.path` から利用できます。

バージョン 2.2 で追加。

`os.pathsep`

(PATH のような) サーチパス内の要素を分割するためにオペレーティングシステムが慣習的に用いる文字で、POSIX における ':' や DOS および Windows における ';' に相当します。 `os.path` から利用できます。

`os.defpath`

`exec*p*` や `spawn*p*` において、環境変数辞書内に 'PATH' キーがない場合に使われる標準設定のサーチパスです。 `os.path` から利用できます。

`os.linesep`

現在のプラットフォーム上で行を分割 (あるいは終端) するために用いられている文字列です。この値は例えば POSIX での '\n' や Mac OS での '\r' のように、単一の文字にもなりますし、例えば Windows での '\r\n' のように複数の文字列にもなります。テキストモードで開いたファイルに書き込む時には、 `os.linesep` を利用しないでください。すべてのプラットフォームで、単一の '\n' を使用してください。

`os.devnull`

ヌルデバイスのファイルパスです。例えば POSIX では '/dev/null' で、Windows では 'nul' です。この値は `os.path` から利用できます。

バージョン 2.4 で追加。

15.1.7 雑多な関数

`os.urandom(n)`

暗号に関する用途に適した n バイトからなるランダムな文字列を返します。

この関数は OS 固有の乱数発生源からランダムなバイト列を生成して返します。この関数の返すデータは暗号を用いたアプリケーションで十分利用できる程度に予測不能ですが、実際のクオリティは OS の実装によって異なります。Unix 系のシステムでは `/dev/urandom` への問い合わせを行い、Windows では `CryptGenRandom()` を使います。乱数発生源が見つからない場合、 `NotImplementedError`

を送出します。

プラットフォームが提供している乱数発生器へのインターフェイスについては、`random.SystemRandom` を参照してください。

バージョン 2.4 で追加。

15.2 io — ストリームを扱うコアツール

バージョン 2.6 で追加。

`io` モジュールはストリーム処理を行う Python インタフェースを提供します。Python 2.x では、これは組み込みの `file` オブジェクトの代わりとして提案されていますが、Python 3.x では、これがファイルやストリームのデフォルトインタフェースです。

注釈: このモジュールはもともと、Python 3.x のために設計されたものなので、このドキュメントの中で使われるすべての "bytes" は (bytes がエイリアスとなる) `str` 型のことで、すべての "text" は `unicode` 型のことです。さらに、`io` API では、この 2 つの型は入れ替えられません。

I/O 階層の最上位には抽象基底クラスの `IOBase` があります。`IOBase` ではストリームに対して基本的なインタフェースを定義しています。しかしながら、ストリームに対する読み込みと書き込みが分離されていないことに注意してください。実装においては与えられた操作をサポートしない場合は `IOError` を送出することが許されています。

`IOBase` の拡張は、単純なストリームに対する生のバイト列の読み書きを扱う `RawIOBase` です。`FileIO` は、`RawIOBase` を継承してマシンのファイルシステム中のファイルへのインタフェースを提供します。

`BufferedIOBase` では生のバイトストリーム (`RawIOBase`) 上にバッファ処理を追加します。そのサブクラスの `BufferedWriter`, `BufferedReader`, `BufferedRWPair` では、それぞれ読み込み専用、書き込み専用、読み書き可能なストリームをバッファします。`BufferedRandom` ではランダムアクセスストリームに対してバッファされたインタフェースを提供します。`BytesIO` はインメモリバイトへのシンプルなストリームです。

`IOBase` のもう一つのサブクラスである `TextIOBase` は、テキストを表すバイトストリームを扱い、`unicode` エンコードやデコードといった処理を行います。`TextIOWrapper` はその拡張で、バッファ付き生ストリーム (`BufferedIOBase`) へのバッファされたテキストインタフェースです。最後に `StringIO` は Unicode テキストに対するインメモリストリームです。

引数名は規約に含まれていません。そして `open()` の引数だけがキーワード引数として用いられることが意図されています。

15.2.1 モジュールインタフェース

`io.DEFAULT_BUFFER_SIZE`

モジュールのバッファ I/O クラスで使用するデフォルトのバッファサイズを指定する整数値です。

`open()` は可能であればファイル全体のサイズ (`os.stat()` で取得されます) を使用します。

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

`file` を開き、対応するストリームを返します。ファイルが開けなかった場合、`IOError` が発生します。

`file` は開きたいファイルの (絶対、またはカレントワーキングディレクトリに対する相対) パス名を示す文字列か、開きたいファイルがラップされているファイル記述子です。 (ファイル記述子が与えられた場合、`closefd` が `False` に設定されていない限り、返された I/O オブジェクトが閉じられたときにそのファイル識別子も閉じられます)

`mode` はオプションの文字列です。これによってファイルをどのようなモードで開くか明示することができます。デフォルトは `'r'` でテキストモードで読み取り専用で開くことを指します。他にも `'w'` は書き込み専用 (もしファイルが存在していた場合は上書きになります) となり、`'a'` では追記モードとなります。 (`'a'` はいくつかの Unix システムではすべての書き込みがシーク位置に関係なくファイルの末尾に追記されることを意味します) テキストモードでは、もし `encoding` が指定されていなかった場合、エンコーディングはプラットフォーム依存となります。 (生のバイトデータの読み込みと書き込みはバイナリモードを用いて、`encoding` は未指定のままとします) 指定可能なモードは次の表の通りです。

文字	意味
<code>'r'</code>	読み込み用を開く (デフォルト)
<code>'w'</code>	書き込み用を開き、ファイルはまず切り詰められる
<code>'a'</code>	書き込み用を開き、ファイルが存在する場合は末尾に追記する
<code>'b'</code>	バイナリモード
<code>'t'</code>	テキストモード (デフォルト)
<code>'+'</code>	ディスクファイルを更新用を開く (読み込み / 書き込み)
<code>'U'</code>	ユニバーサル改行モード (後方互換性のためのものです; 新しいコードでは使うべきではありません)

デフォルトモードは `'rt'` です (テキストを読み込み専用で開きます)。バイナリのランダムアクセスでは `'w+b'` はファイルを開き、ファイルを 0 バイトに切り詰めます。一方で `'r+b'` でファイルを開くとサイズの切り詰めは行われません。

Python ではバイナリモードで開かれたファイルとテキストモードで開かれたファイルは区別されます。オペレーティングシステムが区別しない場合でもこの区別は適用されます。バイナリモードで開かれたファイル (つまり `mode` 引数に `'b'` が含まれるとき) では、中身を `bytes` オブジェクトとして返し、一切のデコードを行いません。テキストモード (デフォルトか `mode` 引数に `'t'` が含まれている場合) では、ファイルの内容は `unicode` 文字列として返され、バイト列はプラットフォーム依存のエンコーディングか、`encoding` が指定された場合は指定されたエンコーディングを使ってデコードされます。

オプションの `buffering` はバッファ用の設定を行う整数値です。0 を設定することでバッファがオフになります (バイナリモードでのみ有効です)。1 の場合は 1 行ごとのバッファリングを行い (テキストモードでのみ利用可能です)、1 より大きい場合は固定サイズチャンクバッファのサイズを表します。`buffering` 引数が与えられなければ、デフォルトのバッファリングポリシーは以下のように働きます:

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

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

`errors` はエンコードやデコードの際のエラーをどのように扱うかを指定する文字列で、バイナリモードでは使えません。'strict' を指定すると、エンコードエラーがあった場合 `ValueError` 例外が発生します (デフォルトである `None` は同様の処理を行います)。'ignore' を指定した場合はエラーを無視します。'replace' を指定した場合は正常に変換されなかった文字の代わりにマーカ (例えば '?' のような文字) を挿入します。書き込みの際には 'xmlcharrefreplace' (適切な XML 文字参照に置き換える) か 'backslashreplace' (バックスラッシュによるエスケープシーケンスに置き換える) のどちらかが使用できます。 `codecs.register_error()` に登録されている他のエラー処理名も指定できます。

`newline` は `universal newlines` モードがどのように働くかを制御します (テキストモードでのみはたります)。これは `None`, `''`, `'\n'`, `'\r'`, `'\r\n'` のいずれかです。これは以下のようにはたります:

- ストリームからの入力の読み込み時、`newline` が `None` であれば、ユニバーサル改行モードが有効になります。入力中の行は `'\n'`, `'\r'`, または `'\r\n'` で終端され、呼び出し元に返される前に `'\n'` に切り詰められます。' ' なら、ユニバーサル改行モードは有効になりますが、行末は翻訳されずに呼び出し元に返されます。その他の正当な値なら、入力行は与えられた文字列でのみ終端され、行末は翻訳されずに呼び出し元に返されます。
- 出力時、`newline` が `None` の場合は、すべての `'\n'` 文字はシステムのデフォルト行区切り文字 `os.linesep` に変換されます。もし `newline` が `''` の場合、変換は起こりません。もし `newline` に他の適切な値が指定された場合は、`'\n'` 文字は与えられた文字に変換されます。

もし `closefd` が `False` で、ファイル名ではなくてファイル記述子が与えられていた場合、処理中のファイル記述子はファイルが閉じられた後も開いたままとなります。もしファイル名が与えられていた場合は、`closefd` は関係ありません。しかし `True` でなければなりません (デフォルト値)。

`open()` によって返されるファイルオブジェクトの型はモードに依存します。 `open()` がテキストモードでファイルを開くために使われた場合 (`'w'`, `'r'`, `'wt'`, `'rt'` など) `TextIOBase` のサブクラス (具体的には `TextIOWrapper`) が返されます。バッファリングをしてバイナリモードでファイルを開く場合、 `BufferedIOBase` のサブクラスが返されます。具体的なクラスは多様です。もし読み取り専用のバイナリモードだった場合は `BufferedReader` が返されます。書き込み専用のバイナリモードだった場合は `BufferedWriter` が返されます。読み書き可能なバイナリモードの場合は

`BufferedRandom` が返されます。バッファリングが無効な場合、raw ストリーム、`RawIOBase` のサブクラス、`FileIO` が返されます。

`unicode` 文字列や `bytes` 文字列をファイルとして読み書きすることも可能です。`unicode` 文字列では `StringIO` を使えばテキストモードで開いたファイルのように扱えます。`bytes` では `BytesIO` を使えばバイナリモードで開いたファイルのように扱えます。

exception `io.BlockingIOError`

非ブロッキングストリームでブロック処理が起きた場合に発生するエラーです。`IOError` を継承しています。

`IOError` で持っている属性以外に `BlockingIOError` では次の属性を持っています。

characters_written

ブロック前にストリームに書き込まれる文字数を保持する整数値です。

exception `io.UnsupportedOperation`

`IOError` と `ValueError` を継承した例外で、ストリームに未サポートの操作が行われた場合に発生します。

15.2.2 I/O 基底クラス

class `io.IOBase`

すべての I/O クラスの抽象基底クラスです。バイトストリームへの操作を行います。パブリックなコンストラクタはありません。

継承先のクラスが選択的にオーバーライドできるように、このクラスは多くのメソッドに空の抽象実装をしています。デフォルトの実装では、読み込み、書き込み、シークができないファイルを表現します。

`IOBase` では `read()`, `readinto()`, `write()` が宣言されていませんが、これはシグナチャが変化するため、実装やクライアントはこれらのメソッドをインタフェースの一部として考えるべきです。また、実装はサポートしていない操作を呼び出されたときは `IOError` を発生させるかもしれません。

ファイルへのバイナリデータの読み書きに用いられる基本型は (`str` としても知られる) `bytes` です。メソッドの引数によっては `bytearray` や `bytes` 配列の `memoryview` のこともあります。`readinto()` などのケースでは、`bytearray` のような書き込み可能オブジェクトが必要です。テキスト I/O クラスは `unicode` データを扱います。

バージョン 2.7 で変更: 実装は `memoryview` 引数をサポートする必要があります。

閉じられたストリームに対するメソッド呼び出しは (問い合わせであっても) 未定義です。この場合、実装は `IOError` を発生させることがあります。

`IOBase` (とそのサブクラス) はイテレータプロトコルをサポートします。`IOBase` オブジェクトをイテレートすると、ストリーム内の行が `yield` されます。行は、ストリームが (`bytes` を与える) バイナリストリームか (`unicode` 文字列を与える) テキストストリームかによって、少し違う定義がされています。下の `readline()` を参照してください。

`IOBase` はコンテキストマネージャでもあります。そのため `with` 構文をサポートします。次の例では、`with` 構文が終わった後で—たとえ例外が発生した場合でも、`file` は閉じられます。

```
with io.open('spam.txt', 'w') as file:
    file.write(u'Spam and eggs!')
```

`IOBase` は以下のデータ属性とメソッドを提供します:

`close()`

このストリームをフラッシュして閉じます。このメソッドはファイルが既に閉じられていた場合は特に何の効果もありません。いったんファイルが閉じられると、すべてのファイルに対する操作 (例えば読み込みや書き込み) で `ValueError` が発生します。

利便性のためにこのメソッドを複数回呼ぶことは許されています。しかし、効果があるのは最初の 1 回だけです。

`closed`

ストリームが閉じられていた場合 `True` になります。

`fileno()`

ストリームが保持しているファイル記述子 (整数値) が存在する場合はそれを返します。もし `IO` オブジェクトがファイル記述子を使っていない場合は `IOError` が発生します。

`flush()`

適用可能であればストリームの書き込みバッファをフラッシュします。読み込み専用や非ブロッキングストリームでは何もしません。

`isatty()`

ストリームが対話的であれば (つまりターミナルや `tty` デバイスにつながっている場合) `True` を返します。

`readable()`

ストリームが読み込める場合 `True` を返します。 `False` の場合は `read()` は `IOError` を発生させます。

`readline(limit=-1)`

ストリームから 1 行読み込んで返します。もし `limit` が指定された場合、最大で `limit` バイトが読み込まれます。

バイナリファイルでは行末文字は常に `b'\n'` となります。テキストファイルでは、認識される行末文字を選択するために `open()` に対する `newline` 引数が使われます。

`readlines(hint=-1)`

ストリームから行のリストを読み込んで返します。 `hint` を指定することで、読み込む行数を制御できます。もし読み込んだすべての行のサイズ (バイト数、もしくは文字数) が `hint` の値を超えた場合、読み込みをそこで終了します。

`file.readlines()` を呼びださなくても `for line in file: ...` のように、`file` オブジェクトを直接イテレートすることができることに注意してください。

seek (*offset*, *whence*=*SEEK_SET*)

ストリーム位置を指定された *offset* バイトに変更します。 *offset* は *whence* で指定された位置からの相対位置として解釈されます。 *whence* のデフォルト値は *SEEK_SET* です。 *whence* に指定できる値は:

- *SEEK_SET* または 0 – ストリームの先頭 (デフォルト)。 *offset* は 0 もしくは正の値でなければなりません。
- *SEEK_CUR* または 1 – 現在のストリーム位置。 *offset* は負の値も可能です。
- *SEEK_END* または 2 – ストリームの末尾。 *offset* は通常負の値です。

新しい絶対位置を返します。

バージョン 2.7 で追加: *SEEK_** 定数

seekable ()

もしストリームがランダムアクセスをサポートしていた場合 *True* を返します。 *False* の場合は *seek()*、*tell()*、*truncate()* は *IOError* を発生させます。

tell ()

現在のストリーム位置を返します。

truncate (*size*=*None*)

指定された *size* バイト (または *size* が指定されなければ現在の位置) にストリームをリサイズします。現在のストリーム位置は変更されません。このリサイズは、現在のファイルサイズを拡大または縮小させることができます。拡大の場合には、新しいファイル領域の内容はプラットフォームに依存します (ほとんどのシステムでは、追加のバイトが 0 で埋められます。Windows では不定です)。新しいファイルサイズが返されます。

writable ()

ストリームが書き込みをサポートしている場合 *True* を返します。 *False* の場合は *write()*、*truncate()* は *IOError* を返します。

writelines (*lines*)

ストリームに複数行書き込みます。行区切り文字は付与されないので、通常書き込む各行の行末には行区切り文字があります。

__del__ ()

オブジェクトの破壊の用意をします。このメソッドはインスタンスの *close()* メソッドを呼びます。 *IOBase* はこのメソッドのデフォルトの実装を提供します

class *io.RawIOBase*

生のバイナリ I/O への基底クラスです。 *IOBase* を継承しています。パブリックコンストラクタはありません。

生のバイナリ I/O は典型的に、下にある OS デバイスや API への、低レベルなアクセスを提供し、高レベルな基本要素へとカプセル化しようとはしません (これはこのページで後述する *Buffered I/O* や *Text I/O* に任せます)。

IOBase の属性やメソッドに加えて、*RawIOBase* は次のメソッドを提供します:

read(*n*=-1)

オブジェクトを *n* バイトまで読み込み、それを返します。簡単のため、*n* が指定されていないか -1 なら、`readall()` が呼び出されます。そうでなければ、システムコール呼び出しが一度だけ行われます。既に EOF に達していたら空のバイトオブジェクトが返されます。オペレーティングシステムコールが返したものが *n* バイトより少なければ、*n* バイトより少なく返されることがあります。

0 バイトが返って、*n* が 0 でなければ、それはファイルの終端を表します。オブジェクトがノンブロッキングモードで、1 バイトも読み込めなければ、`None` が返されます。

readall()

EOF までストリームからすべてのバイトを読み込みます。必要な場合はストリームに対して複数の呼び出しをします。

readinto(*b*)

b に最大 `len(b)` バイト分読み込み、読み込んだバイト数を返します。オブジェクト *b* はメモリ確保済みの書き込み可能なバイト配列、`bytearray` もしくは `memoryview` である必要があります。オブジェクトがノンブロッキングモードで、1 バイトも読み込めなければ、`None` が返されます。

write(*b*)

b を生ストリームに書き込み、書き込んだバイト数を返します。このオブジェクト *b* はバイト配列、`bytes`、`bytearray`、`memoryview` のいずれかである必要があります。返り値は、根底の生ストリームの性質によって、特にノンブロッキングモードである場合に、`len(b)` より小さくなることがあります。生ストリームがブロックされないように設定されていて、かつ 1 バイトも即座に書き込むことができない場合は、`None` が返されます。このメソッドから返った後で呼び出し元は *b* を解放したり変更したりするかもしれないので、実装はメソッド呼び出しの間だけ *b* にアクセスすべきです。

class io.BufferedIOBase

何らかのバッファリングをサポートするバイナリストリームの基底クラスです。`IOBase` を継承します。パブリックなコンストラクタはありません。

`RawIOBase` との主な違いは、メソッド `read()`、`readinto()` および `write()` は、ことによると複数回のシステムコールを行って、(それぞれ) 要求されただけの入力を読み込もうとしたり与えられた出力の全てを消費しようとする点です。

加えて、元になる生ストリームが非ブロッキングモードでかつ準備ができていない場合に、これらのメソッドは、`BlockingIOError` を送出するかもしれません。対応する `RawIOBase` バージョンと違って、`None` を返すことはありません。

さらに、`read()` メソッドは、`readinto()` に従うデフォルト実装を持ちません。

通常の `BufferedIOBase` 実装は `RawIOBase` 実装を継承せずに、`BufferedWriter` と `BufferedReader` がするようにこれをラップすべきです。

`BufferedIOBase` は `IOBase` からのメソッドと属性に加えて、以下のメソッドを提供もしくはオーバーライドします:

raw

`BufferedIOBase` が扱う根底の生ストリーム (`RawIOBase` インスタンス) を返します。これは `BufferedIOBase` API には含まれず、よって実装に含まれないことがあります。

`detach()`

根底の生ストリームをバッファから分離して返します。

生ストリームが取り外された後、バッファは使用不能状態になります。

バッファには、`BytesIO` など、このメソッドで返される単体のストリームという概念を持たないものがあります。これらは `UnsupportedOperation` を送出します。

バージョン 2.7 で追加.

`read(n=-1)`

最大で n バイト読み込み、返します。引数が省略されるか、`None` か、または負の値であった場合、データは EOF に到達するまで読み込まれます。ストリームが既に EOF に到達していた場合は空の bytes オブジェクトが返されます。

引数が正で、元になる生ストリームが対話的でなければ、必要なバイト数を満たすように複数回の生 `read` が発行されるかもしれません (先に EOF に到達しない限りは)。対話的な場合は、最大で一回の raw `read` しか発行されず、短い結果でも EOF に達したことを意味しません。

元になる生ストリームがノンブロッキングモードで、呼び出された時点でデータを持っていないければ、`BlockingIOError` が送出されます。

`read1(n=-1)`

根底の生ストリームの `read()` メソッドを高々 1 回呼び出し、最大で n バイト読み込み、返します。これは、`BufferedIOBase` オブジェクトの上に独自のバッファリングを実装するとき便利です。

`readinto(b)`

b に最大 `len(b)` バイト分読み込み、読み込んだバイト数を返します。オブジェクト b はメモリ確保済みの書き込み可能なバイト配列、`bytearray` もしくは `memoryview` である必要があります。

`read()` と同様、元になる生ストリームが '対話的' でない限り、複数回の `read` が発行されるかもしれません。

元になる生ストリームがノンブロッキングモードで、呼び出された時点でデータを持っていないければ、`BlockingIOError` が送出されます。

`write(b)`

b を書き込み、書き込んだバイト数を返します (これは常に `len(b)` と等しいです。なぜなら、もし書き込みに失敗した場合は `IOError` が発生するからです)。このオブジェクト b はバイト配列、`bytes`、`bytearray`、`memoryview` のいずれかである必要があります。実際の実装に依って、これらのバイトは根底のストリームに読めるように書きこまれたり、パフォーマンスとレイテンシの理由でバッファに保持されたりします。

ノンブロッキングモードであるとき、バッファが満杯で根底の生ストリームが書き込み時点でさらなるデータを受け付けられない場合 `BlockingIOError` が送出されます。

このメソッドが戻った後で、呼び出し元は *b* を解放、または変更するかもしれないので、実装はメソッド呼び出しの間だけ *b* にアクセスすべきです。

15.2.3 生ファイル I/O

class `io.FileIO` (*name*, *mode*='r', *closefd*=True)

`FileIO` はバイトデータを含む OS レベルのファイルを表します。 `RawIOBase` インタフェースを (したがって `IOBase` インタフェースも) 実装しています。

name はこの 2 つのいずれかに出来ます:

- 開くファイルのパスを表す文字列
- 結果の `FileIO` オブジェクトがアクセスを与える、既存の OS レベルファイルディスクリプタの数を表す整数

mode はそれぞれ読み込み (デフォルト)、書き込み、追記を表す 'r'、'w'、'a' にすることができます。ファイルは書き込みまたは追記モードで開かれたときに存在しなければ作成されます。書き込みモードでは存在したファイル内容は消されます。読み込みと書き込みを同時に行いたければ '+' をモードに加えて下さい。

このクラスの `read()` (正の引数で呼び出されたとき), `readinto()` および `write()` メソッドは、単にシステムコールを一度呼び出します。

`IOBase` および `RawIOBase` から継承した属性とメソッドに加えて、`FileIO` は以下のデータ属性とメソッドを提供しています:

mode

コンストラクタに渡されたモードです。

name

ファイル名。コンストラクタに名前が渡されなかったときはファイル記述子になります。

15.2.4 バッファ付きストリーム

バッファ付き I/O ストリームは、I/O デバイスに生 I/O より高レベルなインタフェースを提供します。

class `io.BytesIO` ([*initial_bytes*])

インメモリの bytes バッファを利用したストリームの実装。 `BufferedIOBase` を継承します。

省略可能な引数 *initial_bytes* は、初期データを含んだ bytes オブジェクトです。

`BytesIO` は `BufferedIOBase` または `IOBase` からのメソッドに加えて、以下のメソッドを提供もしくはオーバーライドします:

getvalue()

バッファの全内容を保持した bytes を返します。

`read1()`

`BytesIO` においては、このメソッドは `read()` と同じです。

class `io.BufferedReader` (`raw`, `buffer_size=DEFAULT_BUFFER_SIZE`)

読み込み可能でシーケンシャルな `RawIOBase` オブジェクトへの、高レベルなアクセスを提供するバッファです。 `BufferedIOBase` を継承します。このオブジェクトからデータを読み込むとき、根底の生ストリームからより大きい量のデータが要求されることがあり、内部バッファに保存されます。バッファされたデータは、続く読み込み時に直接返されます。

このコンストラクタは与えられた `raw` ストリームと `buffer_size` に対し `BufferedReader` を生成します。 `buffer_size` が省略された場合、代わりに `DEFAULT_BUFFER_SIZE` が使われます。

`BufferedReader` は `BufferedIOBase` または `IOBase` からのメソッドに加えて、以下のメソッドを提供もしくはオーバーライドします:

`peek([n])`

位置を進めずにストリームからバイト列を返します。これを果たすために生ストリームに対して行われる `read` は高々一度だけです。返されるバイト数は、要求より少ないかもしれませんが、多いかもしれません。

`read([n])`

`n` バイトを読み込んで返します。 `n` が与えられないかまたは負の値ならば、EOF まで、または非ブロッキングモード中で `read` 呼び出しがブロックされるまでを返します。

`read1(n)`

生ストリームに対したただ一度の呼び出しで最大 `n` バイトを読み込んで返します。少なくとも 1 バイトがバッファされていれば、バッファされているバイト列だけが返されます。それ以外の場合にはちょうど一回生ストリームに `read` 呼び出しが行われます。

class `io.BufferedWriter` (`raw`, `buffer_size=DEFAULT_BUFFER_SIZE`)

書き込み可能でシーケンシャルな `RawIOBase` オブジェクトへの、高レベルなアクセスを提供するバッファです。 `BufferedIOBase` を継承します。このオブジェクトに書き込むとき、データは通常内部バッファに保持されます。このバッファは、以下のような種々の状況で根底の `RawIOBase` オブジェクトに書きこまれます:

- 保留中の全データに対してバッファが足りなくなったとき;
- `flush()` が呼び出されたとき;
- `seek()` が (`BufferedRandom` オブジェクトに対して) 呼び出されたとき;
- `BufferedWriter` オブジェクトが閉じられたり破棄されたりしたとき。

このコンストラクタは与えられた書き込み可能な `raw` ストリームに対し `BufferedWriter` を生成します。 `buffer_size` が省略された場合、 `DEFAULT_BUFFER_SIZE` がデフォルトになります。

第三引数 `max_buffer_size` が提供されていますが、使われず、非推奨です。

`BufferedWriter` は `BufferedIOBase` または `IOBase` からのメソッドに加えて、以下のメソッドを提供もしくはオーバーライドします:

flush()

バッファに保持されたバイト列を生ストリームに強制的に流し込みます。生ストリームがブロックした場合 *BlockingIOError* が送出されます。

write(b)

b を書き込み、書き込んだバイト数を返します。このオブジェクト *b* はバイト配列、*bytes*、*bytearray*、*memoryview* のいずれかである必要があります。ノンブロッキングモードのときは、バッファが書き込まれる必要があるところで生ストリームがブロックした場合 *BlockingIOError* が送出されます。

class io.BufferedRandom(raw, buffer_size=DEFAULT_BUFFER_SIZE)

ランダムアクセスストリームへのバッファ付きインタフェース。 *BufferedReader* および *BufferedWriter* を継承し、さらに *seek()* および *tell()* をサポートしています。

このコンストラクタは第一引数として与えられるシーク可能な生ストリームに対し、リーダーおよびライターを作成します。 *buffer_size* が省略された場合、 *DEFAULT_BUFFER_SIZE* がデフォルトになります。

第三引数 *max_buffer_size* が提供されていますが、使われず、非推奨です。

BufferedRandom は *BufferedReader* や *BufferedWriter* にできることは何でもできます。

class io.BufferedRWPair(reader, writer, buffer_size=DEFAULT_BUFFER_SIZE)

2つの単方向 *RawIOBase* オブジェクト – 一つは読み込み可能、他方が書き込み可能 – を組み合わせてバッファ付きの双方向 I/O オブジェクトにしたものです。 *BufferedIOBase* を継承しています。

reader と *writer* はそれぞれ読み込み可能、書き込み可能な *RawIOBase* オブジェクトです。 *buffer_size* が省略された場合 *DEFAULT_BUFFER_SIZE* がデフォルトになります。

第四引数 *max_buffer_size* が提供されていますが、使われず、非推奨です。

BufferedRWPair は、 *UnsupportedOperation* を送出する *detach()* を除く、 *BufferedIOBase* の全てのメソッドを実装します。

警告: *BufferedRWPair* は下層の生ストリームのアクセスを同期しようとはしません。同じオブジェクトをリーダーとライターとして渡してはいけません。その場合は代わりに *BufferedRandom* を使用してください。

15.2.5 テキスト I/O

class io.TextIOBase

テキストストリームの基底クラスです。このクラスはストリーム I/O への Unicode 文字と行に基づいたインタフェースを提供します。Python の *unicode* 文字列は変更不可能なので、*readinto()* メソッドは存在しません。 *IOBase* を継承します。パブリックなコンストラクタはありません。

IOBase から継承した属性とメソッドに加えて、 *TextIOBase* は以下のデータ属性とメソッドを提供しています:

encoding

エンコーディング名で、ストリームのバイト列を文字列にデコードするとき、また文字列をバイト列にエンコードするときに使われます。

errors

このエンコーダやデコーダのエラー設定です。

newlines

文字列、文字列のタプル、または `None` で、改行がどのように読み換えられるかを指定します。実装や内部コンストラクタのフラグに依って、これは利用できないことがあります。

buffer

`TextIOBase` が扱う根底のバイナリバッファ (`BufferedIOBase` インスタンス) です。これは `TextIOBase` API には含まれず、よって実装に含まれないことがあります。

detach()

根底のバイナリバッファを `TextIOBase` から分離して返します。

根底のバッファが取り外された後、`TextIOBase` は使用不能状態になります。

`TextIOBase` 実装には、`StringIO` など、根底のバッファという概念を持たないものがあります。これらを読み出すと `UnsupportedOperation` を送出します。

バージョン 2.7 で追加.

read(*n=-1*)

最大 *n* 文字をストリームから読み込み、一つの `unicode` にして返します。 *n* が負の値または `None` ならば、EOF まで読みます。

readline(*limit=-1*)

改行または EOF まで読み込み、一つの `unicode` を返します。ストリームが既に EOF に到達している場合、空文字列が返されます。

もし *limit* が指定された場合、最大で *limit* バイトが読み込まれます。

seek(*offset, whence=SEEK_SET*)

指定された *offset* にストリーム位置を変更します。挙動は *whence* 引数によります。 *whence* のデフォルト値は `SEEK_SET` です。:

- `SEEK_SET` または 0: ストリームの先頭からシークします (デフォルト)。 *offset* は `TextIOBase.tell()` が返す数か 0 のどちらかでなければなりません。それ以外の *offset* 値は未定義の挙動を起こします。
- `SEEK_CUR` または 1: 現在の位置に "シークします"。 *offset* は 0 でなければなりません。つまり何もしません (他の値はサポートされていません)。
- `SEEK_END` または 2: ストリーム終端へシークします。 *offset* は 0 でなければなりません (他の値はサポートされていません)。

新しい絶対位置を、不透明な数値で返します。

バージョン 2.7 で追加: `SEEK_*` 定数.

tell()

ストリームの現在位置を不透明な数値で返します。この値は根底のバイナリストレージ内でのバイト数を表すとは限りません。

write(s)

unicode 文字列 *s* をストリームに書き込み、書き込まれた文字数を返します。

class io.TextIOWrapper (*buffer*, *encoding=None*, *errors=None*, *newline=None*,
line_buffering=False)
BufferedIOBase バイナリストリーム上のバッファ付きテキストストリーム。 *TextIOBase* を継承します。

encoding にはストリームをデコードしたりそれを使ってエンコードしたりするエンコーディング名を渡します。デフォルトは *locale.getpreferredencoding()* です。

errors はオプションの文字列で、エンコードやデコードの際のエラーをどのように扱うかを指定します。エンコードエラーがあったら *ValueError* 例外を送出させるには 'strict' を渡します (デフォルトの *None* でも同じです)。エラーを無視させるには 'ignore' です。 (注意しなければならないのは、エンコーディングエラーを無視するとデータ喪失につながる可能性があるということです。) 'replace' は正常に変換されなかった文字の代わりにマーカ (たとえば '?') を挿入させます。書き込み時には 'xmlcharrefreplace' (適切な XML 文字参照に置き換え) や 'backslashreplace' (バックスラッシュによるエスケープシーケンスに置き換え) も使えます。他にも *codecs.register_error()* で登録されたエラー処理名が有効です。

newline は行末をどのように処理するかを制御します。これは *None*, ' ', '\n', '\r', '\r\n' のいずれかです。これは以下のように働きます:

- ストリームからの入力を読み込んでいる時、*newline* が *None* の場合、*universal newlines* モードが有効になります。入力中の行は '\n'、'\r'、または '\r\n' で終わり、呼び出し元に返される前に '\n' に変換されます。' ' の場合、ユニバーサル改行モードは有効になりますが、行末は変換されずに呼び出し元に返されます。その他の合法的な値の場合、入力行は与えられた文字列でのみ終わり、行末は変換されずに呼び出し元に返されます。
- 出力時、*newline* が *None* の場合は、すべての '\n' 文字はシステムのデフォルト行区切り文字 *os.linesep* に変換されます。もし *newline* が ' ' の場合、変換は起こりません。もし *newline* に他の適切な値が指定された場合は、'\n' 文字は与えられた文字に変換されます。

line_buffering が *True* の場合、*write* への呼び出しが改行文字もしくはキャリッジリターンを含んでいれば、暗黙的に *flush()* が呼び出されます。

TextIOBase およびその親クラスの属性に加えて、*TextIOWrapper* は以下の属性を提供しています:

line_buffering

行バッファリングが有効かどうか。

class io.StringIO (*initial_value=u'', newline=u'\n'*)
Unicode テキストのためのインメモリストリーム。 *TextIOWrapper* を継承します。

バッファの初期値を *initial_value* で与えることが出来ます。改行変換を有効にすると、改行コードは *write()* によってエンコードされます。ストリームはバッファの開始位置に配置されます。

newline 引数は *TextIOWrapper* のものと同じように働きます。デフォルトでは `\n` 文字だけを行末とみなし、また、改行の変換は行いません。 *newline* に `None` をセットすると改行コードを全てのプラットフォームで `\n` で書き込みますが、読み込み時にはそれでもユニバーサル改行としてのデコードは実行されます。

StringIO およびその親クラスから継承したメソッドに加えて *StringIO* は以下のメソッドを提供しています:

getvalue()

StringIO オブジェクトの *close()* メソッドが呼び出される前の、任意の時点でのバッファの全内容を含む `unicode` を返します。改行コードのデコードは *read()* によって行われますが、これによるストリーム位置の変更は起こりません。

使用例:

```
import io

output = io.StringIO()
output.write(u'First line.\n')
output.write(u'Second line.\n')

# Retrieve file contents -- this will be
# u'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

class io.IncrementalNewlineDecoder

改行を *universal newlines* モードにデコードするヘルパーコーデックです。 *codecs.IncrementalDecoder* を継承しています。

15.2.6 進んだ話題

ここで、上述の I/O 実装に関係するいくつかの進んだ話題について議論します。

パフォーマンス

バイナリ I/O

バッファ付き I/O は、ユーザが 1 バイトだけ要求したときでさえ、データを大きな塊でのみ読み書きします。これにより、オペレーティングシステムのバッファ無し I/O ルーチン呼び出して実行する非効率性をすべて隠しています。その成果は、OS と処理される I/O の種類に本当にとても大きく依存します (例えば、Linux のような現行の OS では、バッファ無しディスク I/O がバッファ付き I/O と同じくらい早いことがあります)。し

かし、最低でも、バッファ付き I/O は予測できるパフォーマンスを提供します。ですから、ほとんどいつも、バッファ無し I/O より、バッファ付きの I/O を使うほうが望ましいです。

テキスト I/O

(ファイルなどの) バイナリストレージ上のテキスト I/O は、同じストレージ上のバイナリ I/O より非常に遅いです。なぜならこれは、文字コーデックを使った Unicode からバイナリデータへの変換を暗示しているからです。これは大量のテキストデータ (例えば非常に大きなログファイル) を扱うときに顕著に成り得ます。同様に、`TextIOWrapper.tell()` や `TextIOWrapper.seek()` はどちらも、使われている復元アルゴリズムのために遅くなります。

しかし *StringIO* は、ネイティブなインメモリ Unicode コンテナで、*BytesIO* と同程度の速度を示します。

マルチスレッディング

(Unix における `read(2)` のような) オペレーティングシステムコールの、それがラッピングしているものがスレッドセーフであるような範囲内では、*FileIO* オブジェクトもまた、スレッドセーフです。

バイナリバッファ付きオブジェクト (*BufferedReader*, *BufferedWriter*, *BufferedRandom* および *BufferedRWPair* のインスタンス) は、その内部構造をロックを使って保護します。このため、これらを複数のスレッドから同時に呼び出しても安全です。

TextIOWrapper オブジェクトはスレッドセーフではありません。

リエントラント性

バイナリバッファ付きオブジェクト (*BufferedReader*, *BufferedWriter*, *BufferedRandom* および *BufferedRWPair* のインスタンス) は、リエントラントではありません。リエントラントな呼び出しは普通の状況では起こりませんが、I/O を *signal* ハンドラで行なっているときに起こりえます。バッファ化されたオブジェクトに、すでに同じスレッドからアクセスされているのにもかかわらず、再び入ろうとすると *RuntimeError* が送出されます。

`open()` 関数は *TextIOWrapper* 内部のバッファ付きオブジェクトをラップするため、テキストファイルにも暗黙に拡張されます。これは、標準ストリームを含むので、組み込み関数 `print()` にも同様に影響します。

15.3 time — 時刻データへのアクセスと変換

このモジュールでは、時刻に関するさまざまな関数を提供します。関連した機能について、*datetime*, *calendar* モジュールも参照してください。

このモジュールは常に利用可能ですが、すべての関数がすべてのプラットフォームで利用可能なわけではありません。このモジュールで定義されているほとんどの関数は、プラットフォーム上の同名の C ライブラリ関数を呼び出します。これらの関数に対する意味付けはプラットフォーム間で異なるため、プラットフォーム提供のドキュメントを読んでおくとう便利でしょう。

まずいくつかの用語の説明と慣習について整理します。

- エポック (*epoch*) は、時刻の計測がはじまった時点のことです。その年の 1 月 1 日の午前 0 時に ”エポックからの経過時間” が 0 になるように設定されます。Unix ではエポックは 1970 年です。エポックがどうなっているかを知るには、`gmtime(0)` の値を見るとよいでしょう。
- このモジュールの中の関数は、エポック以前あるいは遠い未来の日付や時刻を扱うことができません。将来カットオフ (関数が正しく日付や時刻を扱えなくなる) が起きる時点は、C ライブラリによって決まります。32-bit システムではカットオフは通常 2038 年です。
- **2000 年問題 (Y2K):** Python はプラットフォームの C ライブラリに依存しています。C ライブラリは日付および時刻をエポックからの経過秒で表現するので、一般的に 2000 年問題を持ちません。時刻を表現する `struct_time` (下記を参照してください) を入力として受け取る関数は一般的に 4 桁表記の西暦年を要求します。以前のバージョンとの互換性のために、モジュール変数 `accept2dyear` がゼロでない整数の場合、2 桁の西暦年をサポートします。この変数の初期値は環境変数 `PYTHONY2K` が空文字列のとき 1 に設定されます。空文字列でない文字列が設定されている場合、0 に設定されます。こうして、`PYTHONY2K` を空文字列でない文字列に設定することで、西暦年の入力がすべて 4 桁の西暦年でない限りならなくすることができます。2 桁の西暦年が入力された場合には、POSIX または X/Open 標準に従って変換されます: 69-99 の西暦年は 1969-1999 となり、0-68 の西暦年は 2000-2068 になります。100-1899 は常に不正な値になります。
- UTC は協定世界時 (Coordinated Universal Time) のことです (以前はグリニッジ標準時または GMT として知られていました)。UTC の頭文字の並びは誤りではなく、英仏の妥協によるものです。
- DST は夏時間 (Daylight Saving Time) のことで、一年のうちの一定期間に 1 時間タイムゾーンを修正することです。DST のルールは不可思議で (地域ごとに法律で定められています)、年ごとに変わることもあります。C ライブラリはローカルルールを記したテーブルを持っており (柔軟に対応するため、たいていはシステムファイルから読み込まれます)、この点に関しては唯一の真実の知識の源です。
- 多くの現時刻を返す関数 (real-time functions) の精度は、値や引数を表現するために使う単位から想像されるよりも低いかも知れません。例えば、ほとんどの Unix システムにおいて、クロックの 1 ティックの精度は 50 から 100 分の 1 秒に過ぎません。
- 一方、`time()` および `sleep()` は Unix の同等の関数よりましな精度を持っています: 時刻は浮動小数点で表され、`time()` は可能なかぎり最も正確な時刻を (Unix の `gettimeofday()` があればそれを使って) 返します。また `sleep()` にはゼロでない端数を与えることができます (Unix の `select()` があれば、それを使って実装しています)。
- `gmtime()`, `localtime()`, `strptime()` が返す時刻値、および `asctime()`, `mktime()`, `strftime()` に与える時刻値はどちらも 9 つの整数からなるシーケンスです。 `gmtime()`, `localtime()`, `strptime()` の戻り値では個々のフィールドに属性名でアクセスすることもできます。

これらのオブジェクトについての解説は `struct_time` を参照してください。

バージョン 2.2 で変更: 時刻値の配列はタプルから `struct_time` に変更され、それぞれのフィールドに属性名がつけられました。

- 時間の表現を変換するには、以下の関数を利用してください:

対象	変換先	関数
エポックからの秒数	UTC の <code>struct_time</code>	<code>gmtime()</code>
エポックからの秒数	ローカル時間の <code>struct_time</code>	<code>localtime()</code>
UTC の <code>struct_time</code>	エポックからの秒数	<code>calendar.timegm()</code>
ローカル時間の <code>struct_time</code>	エポックからの秒数	<code>mktime()</code>

このモジュールでは以下の関数とデータ型を定義します:

`time.accept2dayear`

2 桁の西暦年を使えるかを指定するブール型の値です。標準では真ですが、環境変数 `PYTHONY2K` が空文字列でない値に設定されている場合には偽になります。実行時に変更することもできます。

`time.altzone`

ローカルの夏時間タイムゾーンにおける UTC からの時刻オフセットで、西に行くほど増加する、秒で表した値です (ほとんどの西ヨーロッパでは負になり、アメリカでは正、イギリスではゼロになります)。daylight がゼロでないときのみ使用してください。

`time.asctime([t])`

`gmtime()` や `localtime()` が返す時刻を表現するタプル又は `struct_time` を、'Sun Jun 20 23:21:05 1993' といった書式の 24 文字の文字列に変換します。t が与えられていない場合には、`localtime()` が返す現在の時刻が使われます。`asctime()` はロケール情報を使いません。

注釈: 同名の C の関数と違って、末尾に改行文字を加えません。

バージョン 2.1 で変更: t が省略可能になりました。

`time.clock()`

Unix では、現在のプロセッサ時間秒を浮動小数点数で返します。時刻の精度および "プロセッサ時間 (processor time)" の定義そのものは同じ名前の C 関数に依存します。いずれにせよ、この関数は Python のベンチマークや計時アルゴリズムに使われています。

Windows では、最初にこの関数が呼び出されてからの経過時間を wall-clock 秒で返します。この関数は Win32 関数 `QueryPerformanceCounter()` に基づいていて、その精度は通常 1 マイクロ秒以下です。

`time.ctime([secs])`

エポックからの経過秒数で表現された時刻を、ローカルの時刻を表現する文字列に変換します。`secs` を指定しないか `None` を指定した場合、`time()` が返す値を現在の時刻として使用します。`ctime(secs)` は `asctime(localtime(secs))` と等価です。ローカル情報は `ctime()` には使用されません。

バージョン 2.1 で変更: `secs` が省略可能になりました。

バージョン 2.4 で変更: `secs` が `None` の場合に現在時刻を使うようになりました。

`time.daylight`

DST タイムゾーンが定義されている場合ゼロでない値になります。

`time.gmtime([secs])`

エポックからの経過時間で表現された時刻を、UTC で `struct_time` に変換します。このとき `dst` フラグは常にゼロとして扱われます。`secs` を指定しないか `None` を指定した場合、`time()` が返す値を現在の時刻として使用します。秒の端数は無視されます。`struct_time` オブジェクトについては前述の説明を参照してください。`calendar.timegm()` はこの関数と逆の変換を行います。

バージョン 2.1 で変更: `secs` が省略可能になりました。

バージョン 2.4 で変更: `secs` が `None` の場合に現在時刻を使うようになりました。

`time.localtime([secs])`

`gmtime()` に似ていますが、ローカル時間に変換します。`secs` を指定しないか `None` を指定した場合、`time()` が返す値を現在の時刻として使用します。DST が適用されている場合は `dst` フラグには 1 が設定されます。

バージョン 2.1 で変更: `secs` が省略可能になりました。

バージョン 2.4 で変更: `secs` が `None` の場合に現在時刻を使うようになりました。

`time.mktime(t)`

`localtime()` の逆を行います。引数は `struct_time` が 9 個の要素すべての値を持つ完全なタプル (`dst` フラグも必要です; 現在の時刻に DST が適用されるか不明の場合は -1 を使用してください) で、UTC ではなく ローカル 時間を指定します。戻り値は `time()` との互換性のために浮動小数点になります。入力した値が正しい時刻を表現できない場合、例外 `OverflowError` または `ValueError` が送出されます (どちらが送出されるかは、無効な値を受け取ったのが Python と下層の C ライブラリのどちらなのかによって決まります)。この関数で生成できる最も過去の時刻値はプラットフォームに依存します。

`time.sleep(secs)`

与えられた秒数の間、呼び出したスレッドの実行を停止します。より精度の高い実行停止時間を指定するために、引数は浮動小数点にしてもかまいません。何らかのシステムシグナルがキャッチされた場合、それに続いてシグナル処理ルーチンが実行され、`sleep()` を停止します。従って実際の実行停止時間は要求した時間よりも短くなるかもしれません。また、システムが他の処理をスケジュールするために、実行停止時間が要求した時間よりも多少長い時間になることもあります。

`time.strftime(format[, t])`

Convert a tuple or `struct_time` representing a time as returned by `gmtime()` or `localtime()` to a string as specified by the `format` argument. If `t` is not provided, the current time as returned by `localtime()` is used. `format` must be a string. `ValueError` is raised if any field in `t` is outside of the allowed range. `strftime()` returns a locale dependent byte string; the result may be converted to unicode by doing `strftime(<myformat>).decode(locale.getlocale()[1])`.

バージョン 2.1 で変更: `t` が省略可能になりました。

バージョン 2.4 で変更: `t` のフィールド値が許容範囲外の値の場合に `ValueError` を送出するようになりました。

バージョン 2.5 で変更: 0 は時刻値タブルのどこでも使用可能になりました。もし不正な値の場合には正常な値に修正されます。

format 文字列には以下のディレクティブ (指示語) を埋め込むことができます。これらはフィールド長や精度のオプションを付けずに表され、*strftime()* の結果の対応する文字列に置き換えられます:

ディレクティブ	意味	注釈
%a	ロケールの短縮された曜日名になります。	
%A	ロケールの曜日名になります。	
%b	ロケールの短縮された月名になります。	
%B	ロケールの月名になります。	
%c	ロケールの日時を適切な形式で表します。	
%d	月中の日にちの 10 進表記になります [01,31]。	
%H	時 (24 時間表記) の 10 進表記になります [00,23]。	
%I	時 (12 時間表記) の 10 進表記になります [01,12]。	
%j	年中の日にちの 10 進表記になります [001,366]。	
%m	月の 10 進表記になります [01,12]。	
%M	分の 10 進表記になります [00,59]。	
%p	ロケールの AM もしくは PM と等価な文字列になります。	(1)
%S	秒の 10 進表記になります [00,61]。	(2)
%U	年の初めから何週目か (日曜を週の始まりとします) を表す 10 進数になります [00,53]。年が明けてから最初の日曜日までのすべての曜日は 0 週目に属すると見なされます。	(3)
%w	曜日の 10 進表記になります [0 (日曜日),6]。	
%W	年の初めから何週目か (月曜を週の始まりとします) を表す 10 進数になります [00,53]。年が明けてから最初の月曜日までの全ての曜日は 0 週目に属すると見なされます。	(3)
%x	ロケールの日付を適切な形式で表します。	
%X	ロケールの時間を適切な形式で表します。	
%y	西暦の下 2 桁の 10 進表記になります [00,99]。	
%Y	西暦 (4 桁) の 10 進表記を表します。	
%Z	タイムゾーンの名前を表します (タイムゾーンがない場合には空文字列)。	
%%	文字 '%' を表します。	

注釈:

- (1) *strptime()* 関数で使う場合、%p ディレクティブが出力結果の時刻フィールドに影響を及ぼすのは、時刻を解釈するために %I を使ったときのみです。
- (2) 値の幅は実際に 0 から 61 です; これは閏秒を考慮したもので、(極めて稀ですが) 2 秒の閏秒も加味しています。
- (3) *strptime()* 関数で使う場合、%U および %W を計算に使うのは曜日と年を指定したときだけ

です。

以下に **RFC 2822** インターネット電子メール標準で定義されている日付表現と互換の書式の例を示します。^{*1}

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

一部のプラットフォームではさらにいくつかのディレクティブがサポートされていますが、標準 ANSI C で意味のある値はここで列挙したものだけです。あなたのプラットフォームでサポートされている書式コードの全一覧については、`strftime(3)` のドキュメントを参照してください。

一部のプラットフォームでは、フィールドの幅や精度を指定するオプションがディレクティブの先頭の文字 '%' の直後に付けられるようになっていました; この機能も移植性はありません。フィールドの幅は通常 2 ですが、%j は例外で 3 です。

`time.strptime(string[, format])`

時刻を表現する文字列を書式に従って解釈します。返される値は `gmtime()` や `localtime()` が返すような `struct_time` です。

`format` パラメータは `strftime()` で使うものと同じディレクティブを使います; このパラメータの値はデフォルトでは "%a %b %d %H:%M:%S %Y" で、`ctime()` が返すフォーマットに一致します。`string` が `format` に従って解釈できなかった場合や解析しようとする `string` が解析後に余分なデータを持っていた場合、`ValueError` が送出されます。欠落したデータについて、適切な値を推測できない場合はデフォルトの値で埋められ、その値は (1900, 1, 1, 0, 0, 0, 0, 1, -1) です。

例えば:

```
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")
time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0,
                  tm_sec=0, tm_wday=3, tm_yday=335, tm_isdst=-1)
```

%Z ディレクティブへのサポートは `tzname` に収められている値と `daylight` が真かどうかで決められます。このため、常に既知の (かつ夏時間でないと考えられている) UTC や GMT を認識する時以外はプラットフォーム固有の動作になります。

ドキュメント内で説明されているディレクティブだけがサポートされています。`strftime()` はプラットフォームごとに実装されているので、説明されていないディレクティブも利用できるかもしれません。しかし、`strptime()` はプラットフォーム非依存なので、ドキュメント内でサポートされているとされているディレクティブ以外は利用できません。

class `time.struct_time`

`gmtime()`, `localtime()` および `strptime()` が返す時刻値シーケンスの型です。これは名前付

^{*1} %Z の使用は現在非推奨です。ただし、ここで実現したい時間および分オフセットへの展開を行ってくれる %z エスケープはすべての ANSI C ライブラリでサポートされているわけではありません。また、1982 年に提出されたオリジナルの **RFC 822** 標準では西暦の表現を 2 桁とするよう要求している (%Y でなく %y) もの、実際には 2000 年になるだいが以前から 4 桁の西暦表現に移行しています。その後 **RFC 822** は撤廃され、4 桁の西暦表現は **RFC 1123** で初めて勧告され、**RFC 2822** において義務付けられました。

きタプル (*named tuple*) のインタフェースをもったオブジェクトです。値はインデックスでも属性名でもアクセス可能です。以下の値があります:

インデックス	属性	値
0	tm_year	(例えば 1993)
1	tm_mon	[1,12] の間の数
2	tm_mday	[1,31] の間の数
3	tm_hour	[0,23] の間の数
4	tm_min	[0,59] の間の数
5	tm_sec	[0,61] の間の数 <i>strptime()</i> の説明にある (2) を読んで下さい
6	tm_wday	[0,6] の間の数、月曜が 0 になります
7	tm_yday	[1,366] の間の数
8	tm_isdst	0, 1 または -1; 以下を参照してください

バージョン 2.2 で追加.

C の構造体と違って、月の値が 0 から 11 ではなく 1 から 12 であることに注意してください。西暦年の値は上の *2000 年問題 (Y2K)* で述べたように扱われます。

mktime() の呼び出し時に、tm_isdst は夏時間が有効な場合は 1、そうでない場合は 0 に設定されることがあります。値が -1 の場合は夏時間について不明なことを表していて、普通 tm_isdst は正しい状態に設定されます。

struct_time を引数とする関数に正しくない長さの *struct_time* や要素の型が正しくない *struct_time* を与えた場合には、*TypeError* が送出されます。

`time.time()`

エポックからの秒数を浮動小数点数で返します。時刻は常に浮動小数点で返されますが、すべてのシステムが 1 秒より高い精度で時刻を提供するとは限らないので注意してください。この関数が返す値は通常減少していくことはありませんが、この関数を 2 回呼び出し、呼び出しの間にシステムクロックの時刻を巻き戻して設定した場合には、以前の呼び出しよりも低い値が返ることがあります。

`time.timezone`

(DST でない) ローカルタイムゾーンの UTC からの時刻オフセットで、西に行くほど増加する秒で表した値です (ほとんどの西ヨーロッパでは負になり、アメリカでは正、イギリスではゼロになります)。

`time.tzname`

二つの文字列からなるタプルです。最初の要素は DST でないローカルのタイムゾーン名です。ふたつめの要素は DST のタイムゾーンです。DST のタイムゾーンが定義されていない場合、二つ目の文字列を使うべきではありません。

`time.tzset()`

Reset the time conversion rules used by the library routines. The environment variable TZ specifies how this is done. It will also set the variables tzname (from the TZ environment variable), timezone (non-DST seconds West of UTC), altzone (DST seconds west of UTC) and daylight (to 0 if this timezone does not have any daylight saving time rules, or to nonzero if there is a time, past, present or future when daylight saving time applies).

バージョン 2.3 で追加.

利用できる環境: Unix。

注釈: 多くの場合、環境変数 `TZ` を変更すると、`tzset()` を呼ばない限り `localtime()` のような関数の出力に影響を及ぼすため、値が信頼できなくなってしまう。

`TZ` 環境変数には空白文字を含めてはなりません。

環境変数 `TZ` の標準的な書式は以下の通りです (分かりやすいように空白を入れています):

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

各値は以下のようになっています:

`std` と `dst` 三文字またはそれ以上の英数字で、タイムゾーンの略称を与えます。この値は `time.tzname` になります。

`offset` オフセットは形式: `± hh[:mm[:ss]]` をとります。この表現は、UTC 時刻にするためにローカルな時間に加算する必要がある時間値を示します。'-' が先頭につく場合、そのタイムゾーンは本子午線 (Prime Meridian) より東側にあります; それ以外の場合は本子午線の西側です。オフセットが `dst` の後ろに続かない場合、夏時間は標準時より一時間先行しているものと仮定します。

`start[/time], end[/time]` いつ DST に移動し、DST から戻ってくるかを示します。開始および終了日時の形式は以下のいずれかです:

`Jn` ユリウス日 (Julian day) n ($1 \leq n \leq 365$) を表します。うるう日は計算に含められないため、2月28日は常に59で、3月1日は60になります。

`n` ゼロから始まるユリウス日 ($0 \leq n \leq 365$) です。うるう日は計算に含められるため、2月29日を参照することができます。

`Mm.n.d` m 月の第 n 週における d 番目の日 ($0 \leq d \leq 6, 1 \leq n \leq 5, 1 \leq m \leq 12$) を表します。週5は月 m における最終週の d 番目の日を表し、第4週か第5週のどちらかになります。週1は日 d が最初に現れる日を指します。日0は日曜日です。

`time` は `offset` とほぼ同じで、先頭に符号 ('-' や '+') を付けてはいけないところだけが違います。時刻が指定されていなければ、デフォルトの値 02:00:00 になります。

```
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'
```

多くの Unix システム (*BSD, Linux, Solaris, および Darwin を含む) では、システムの `zoneinfo`

(`tzfile(5)`) データベースを使ったほうが、タイムゾーンごとの規則を指定する上で便利です。これを行うには、必要なタイムゾーンデータファイルへのパスをシステムの `'zoneinfo'` タイムゾーンデータベースからの相対で表した値を環境変数 `TZ` に設定します。システムの `'zoneinfo'` は通常 `/usr/share/zoneinfo` にあります。例えば、`'US/Eastern'`、`'Australia/Melbourne'`、`'Egypt'` ないし `'Europe/Amsterdam'` と指定します。

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

参考:

`datetime` モジュール 日付と時刻に対する、よりオブジェクト指向のインタフェースです。

`locale` モジュール 国際化サービスです。ロケールの設定は `strftime()` および `strptime()` の多くの書式指定子の解釈に影響を及ぼします。

`calendar` モジュール 一般的なカレンダーに関する関数群です。`timegm()` はこのモジュールの `gmtime()` の逆を行う関数です。

15.4 argparse — コマンドラインオプション、引数、サブコマンドのパースー

バージョン 2.7 で追加.

ソースコード: [Lib/argparse.py](#)

チュートリアル

このページでは API のリファレンス情報が記載されています。argparse チュートリアル では、コマンドラインの解析についてより優しく説明しています。

`argparse` モジュールはユーザーフレンドリーなコマンドラインインタフェースの作成を簡単にします。プログラムがどんな引数が必要としているのかを定義すると、`argparse` が `sys.argv` からそのオプションを解析する部分の面倒を見ます。`argparse` モジュールは自動的にヘルプと使用方法メッセージを生成し、ユーザーが不正な引数をプログラムに指定したときにエラーを発生させます。

15.4.1 例

次のコードは、整数のリストを受け取って合計か最大値を返す Python プログラムです:

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print args.accumulate(args.integers)
```

上の Python コードが `prog.py` という名前のファイルに保存されたと仮定します。コマンドラインから便利なヘルプメッセージを表示できます:

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
  N                an integer for the accumulator

optional arguments:
  -h, --help      show this help message and exit
  --sum           sum the integers (default: find the max)
```

適切な引数を与えて実行した場合、このプログラムはコマンドライン引数の整数列の合計が最大値を表示します:

```
$ python prog.py 1 2 3 4
4

$ python prog.py 1 2 3 4 --sum
10
```

不正な引数を与えられた場合、エラーを発生させます:

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

以降の節では、この例をひと通り説明して行きます。

パーサーを作る

`argparse` を使う時の最初のステップは、`ArgumentParser` オブジェクトを生成することです:

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

`ArgumentParser` オブジェクトはコマンドラインを解析して Python データ型にするために必要なすべての情報を保持します。

引数を追加する

`ArgumentParser` にプログラム引数の情報を与えるために、`add_argument()` メソッドを呼び出します。一般的に、このメソッドの呼び出しは `ArgumentParser` に、コマンドラインの文字列を受け取ってそれをオブジェクトにする方法を教えます。この情報は保存され、`parse_args()` が呼び出されたときに利用されます。例えば:

```
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
...                       help='an integer for the accumulator')
>>> parser.add_argument('--sum', dest='accumulate', action='store_const',
...                       const=sum, default=max,
...                       help='sum the integers (default: find the max)')
```

あとで `parse_args()` を呼び出すと、`integers` と `accumulate` という 2 つの属性を持ったオブジェクトを返します。`integers` 属性は 1 つ以上の整数のリストで、`accumulate` 属性はコマンドラインから `--sum` が指定された場合は `sum()` 関数に、それ以外の場合は `max()` 関数になります。

引数を解析する

`ArgumentParser` は引数を `parse_args()` メソッドで解析します。このメソッドはコマンドラインを調べ、各引数を正しい型に変換して、適切なアクションを実行します。ほとんどの場合、これはコマンドラインの解析結果から、シンプルな `Namespace` オブジェクトを構築することを意味します:

```
>>> parser.parse_args(['--sum', '7', '-1', '42'])
Namespace(accumulate=<built-in function sum>, integers=[7, -1, 42])
```

スクリプト内では、`parse_args()` は通常引数なしで呼び出され、`ArgumentParser` は自動的に `sys.argv` からコマンドライン引数を取得します。

15.4.2 ArgumentParser オブジェクト

```
class argparse.ArgumentParser (prog=None, usage=None, description=None, epilog=None,
                                parents=[],          formatter_class=argparse.HelpFormatter,
                                prefix_chars='-',      fromfile_prefix_chars=None,      ar-
                                gument_default=None,   conflict_handler='error',
                                add_help=True)
```

新しい `ArgumentParser` オブジェクトを生成します。すべての引数はキーワード引数として渡すべきです。各引数についてはあとで詳しく説明しますが、簡単に言うと:

- `prog` - プログラム名 (デフォルト: `sys.argv[0]`)
- `usage` - プログラムの利用方法を記述する文字列 (デフォルト: パーサーに追加された引数から生成される)

- *description* - 引数のヘルプの前に表示されるテキスト (デフォルト: none)
- *epilog* - 引数のヘルプの後で表示されるテキスト (デフォルト: none)
- *parents* - *ArgumentParser* オブジェクトのリストで、このオブジェクトの引数が追加される
- *formatter_class* - ヘルプ出力をカスタマイズするためのクラス
- *prefix_chars* - オプションの引数の prefix になる文字集合 (デフォルト: '-')
- *fromfile_prefix_chars* - 追加の引数を読み込むファイルの prefix になる文字集合 (デフォルト: None)
- *argument_default* - 引数のグローバルなデフォルト値 (デフォルト: None)
- *conflict_handler* - 衝突するオプションを解決する方法 (通常は不要)
- *add_help* - `-h/--help` オプションをパーサーに追加する (デフォルト: True)

以下の節では各オプションの利用方法を説明します。

prog

デフォルトでは、*ArgumentParser* オブジェクトはヘルプメッセージ中に表示するプログラム名を `sys.argv[0]` から取得します。このデフォルトの動作は、プログラムがコマンドライン上の起動方法に合わせてヘルプメッセージを作成するため、ほとんどの場合望ましい挙動になります。例えば、`myprogram.py` という名前のファイルに次のコードがあるとしたら:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

このプログラムのヘルプは、プログラム名として (プログラムがどこから起動されたのかに関わらず) `myprogram.py` を表示します:

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
  -h, --help  show this help message and exit
  --foo FOO   foo help
$ cd ..
$ python subdir/myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
  -h, --help  show this help message and exit
  --foo FOO   foo help
```

このデフォルトの動作を変更するには、*ArgumentParser* の `prog=` 引数に他の値を指定します:


```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

optional arguments:
  -h, --help  show this help message and exit
```

プログラム名は、`sys.argv[0]` から取られた場合でも `prog=` 引数で与えられた場合でも、ヘルプメッセージ中では `%(prog)s` フォーマット指定子で利用できます。

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

optional arguments:
  -h, --help  show this help message and exit
  --foo FOO   foo of the myprogram program
```

usage

デフォルトでは、`ArgumentParser` は使用法メッセージを、保持している引数から生成します:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [-h] [--foo [FOO]] bar [bar ...]

positional arguments:
  bar                bar help

optional arguments:
  -h, --help  show this help message and exit
  --foo [FOO] foo help
```

デフォルトのメッセージは `usage=` キーワード引数で変更できます:

```
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%(prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [options]

positional arguments:
  bar                bar help

optional arguments:
  -h, --help  show this help message and exit
  --foo [FOO] foo help
```

`%(prog)s` フォーマット指定子を、使用法メッセージ内でプログラム名として利用できます。

description

多くの場合、`ArgumentParser` のコンストラクターを呼び出すときに `description=` キーワード引数が使用されます。この引数はプログラムが何をしてどう動くのかについての短い説明になります。ヘルプメッセージで、この説明がコマンドラインの利用法と引数のヘルプメッセージの間に表示されます:

```
>>> parser = argparse.ArgumentParser(description='A foo that bars')
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
  -h, --help  show this help message and exit
```

デフォルトでは、説明は与えられたスペースに合わせて折り返されます。この挙動を変更するには、`formatter_class` 引数を参照してください。

epilog

いくつかのプログラムは、プログラムについての追加の説明を引数の説明の後に表示します。このテキストは `ArgumentParser` の `epilog=` 引数に指定できます:

```
>>> parser = argparse.ArgumentParser(
...     description='A foo that bars',
...     epilog="And that's how you'd foo a bar")
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
  -h, --help  show this help message and exit

And that's how you'd foo a bar
```

`description` 引数と同じく、`epilog=` テキストもデフォルトで折り返され、`ArgumentParser` の `formatter_class` 引数で動作を調整できます。

parents

ときどき、いくつかのパースーが共通の引数セットを共有することがあります。それらの引数を繰り返し定義する代わりに、すべての共通引数を持ったパースーを `ArgumentParser` の `parents=` 引数に渡すことができます。 `parents=` 引数は `ArgumentParser` オブジェクトのリストを受け取り、すべての位置アクションとオプションのアクションをそれらから集め、そのアクションを構築中の `ArgumentParser` オブジェクトに追加します:

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

一番親になるパーサーに `add_help=False` を指定していることに注目してください。こうしないと、`ArgumentParser` は2つの `-h/--help` オプションを与えられる(1つは親から、もうひとつは子から)ことになり、エラーが発生します。

注釈: `parents=` に渡す前にパーサーを完全に初期化する必要があります。子パーサーを作成してから親パーサーを変更した場合、その変更は子パーサーに反映されません。

formatter_class

`ArgumentParser` オブジェクトは代替のフォーマットクラスを指定することでヘルプのフォーマットをカスタマイズできます。現在、3つのフォーマットクラスがあります:

```
class argparse.RawDescriptionHelpFormatter
class argparse.RawTextHelpFormatter
class argparse.ArgumentDefaultsHelpFormatter
```

最初の2つは説明のテキストがどう表示されるかについてより制御できるようになっており、残りの1つは引数のデフォルト値についての情報を自動的に追加します。

デフォルトでは `ArgumentParser` オブジェクトはコマンドラインヘルプの中の *description* と *epilog* を折り返して表示します:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     description='''this description
...         was indented weird
...         but that is okay''',
...     epilog='''
...         likewise for this epilog whose whitespace will
...         be cleaned up and whose words will be wrapped
...         across a couple lines''')
>>> parser.print_help()
usage: PROG [-h]
```

(次のページに続く)

(前のページからの続き)

```
this description was indented weird but that is okay

optional arguments:
  -h, --help  show this help message and exit

likewise for this epilog whose whitespace will be cleaned up and whose words
will be wrapped across a couple lines
```

`formatter_class=` に *RawDescriptionHelpFormatter* を渡した場合、*description* と *epilog* は整形済みとされ改行されません:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.RawDescriptionHelpFormatter,
...     description=textwrap.dedent('''\
...         Please do not mess up this text!
...         -----
...             I have indented it
...             exactly the way
...             I want it
...         '''))
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!
-----

    I have indented it
    exactly the way
    I want it

optional arguments:
  -h, --help  show this help message and exit
```

RawTextHelpFormatter maintains whitespace for all sorts of help text, including argument descriptions. However, multiple new lines are replaced with one. If you wish to preserve multiple blank lines, add spaces between the newlines.

残りの利用できるフォーマットクラスである *ArgumentDefaultsHelpFormatter* は各引数のデフォルト値を自動的にヘルプに追加します:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
>>> parser.add_argument('--foo', type=int, default=42, help='FOO!')
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar [bar ...]]

positional arguments:
  bar                BAR! (default: [1, 2, 3])
```

(次のページに続く)

(前のページからの続き)

```
optional arguments:
  -h, --help  show this help message and exit
  --foo FOO   FOO! (default: 42)
```

prefix_chars

ほとんどのコマンドラインオプションは、`-f/--foo` のように接頭辞に `-` を使います。`+f` や `/foo` のような、他の、あるいは追加の接頭辞文字をサポートしなければならない場合、`ArgumentParser` のコンストラクターに `prefix_chars=` 引数を使って指定します:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+-')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')
```

`prefix_chars=` 引数のデフォルトは `'-'` です。`-` を含まない文字セットを指定すると、`-f/--foo` オプションが使用できなくなります。

fromfile_prefix_chars

ときどき、例えば非常に長い引数リストを扱う場合に、その引数リストを毎回コマンドラインにタイプする代わりにファイルに置いておきたい場合があります。`ArgumentParser` のコンストラクターに `fromfile_prefix_chars=` 引数が渡された場合、指定された文字のいずれかで始まる引数はファイルとして扱われ、そのファイルに含まれる引数リストに置換されます。例えば:

```
>>> with open('args.txt', 'w') as fp:
...     fp.write('-f\nbar')
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

ファイルから読み込まれる引数は、デフォルトでは 1 行に 1 つ (ただし、`convert_arg_line_to_args()` も参照してください) で、コマンドライン上でファイルを参照する引数があった場所にその引数があったものとして扱われます。なので、上の例では、`['-f', 'foo', '@args.txt']` は `['-f', 'foo', '-f', 'bar']` と等価になります。

`fromfile_prefix_chars=` 引数のデフォルト値は `None` で、引数がファイル参照として扱われることが無いことを意味しています。

argument.default

一般的には、引数のデフォルト値は `add_argument()` メソッドにデフォルト値を渡すか、`set_defaults()` メソッドに名前と値のペアを渡すことで指定します。しかしまれに、1 つのパースャ全体に適用されるデフォ

ルト引数が便利ことがあります。これを行うには、`ArgumentParser` に `argument_default=` キーワード引数を渡します。例えば、全体で `parse_args()` メソッド呼び出しの属性の生成を抑制するには、`argument_default=SUPPRESS` を指定します:

```
>>> parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar', nargs='?')
>>> parser.parse_args(['--foo', '1', 'BAR'])
Namespace(bar='BAR', foo='1')
>>> parser.parse_args([])
Namespace()
```

conflict_handler

`ArgumentParser` オブジェクトは同じオプション文字列に対して複数のアクションを許可していません。デフォルトでは、`ArgumentParser` オブジェクトは、すでに利用されているオプション文字列を使って新しい引数をつくろうとしたときに例外を送出します:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
..
ArgumentError: argument --foo: conflicting option string(s): --foo
```

ときどき (例えば `parents` を利用する場合など)、古い引数を同じオプション文字列で上書きするほうが便利な場合があります。この動作をするには、`ArgumentParser` の `conflict_handler=` 引数に `'resolve'` を渡します:

```
>>> parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]

optional arguments:
  -h, --help  show this help message and exit
  -f FOO      old foo help
  --foo FOO   new foo help
```

`ArgumentParser` オブジェクトは、すべてのオプション文字列が上書きされた場合にだけアクションを削除することに注目してください。上の例では、`--foo` オプション文字列だけが上書きされているので、古い `-f/--foo` アクションは `-f` アクションとして残っています。

add_help

デフォルトでは、`ArgumentParser` オブジェクトはシンプルにパーサーのヘルプメッセージを表示するオプションを自動的に追加します。例えば、以下のコードを含む `myprogram.py` ファイルについて考えてください:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

コマンドラインに `-h` か `--help` が指定された場合、`ArgumentParser` の `help` が表示されます:

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
  -h, --help  show this help message and exit
  --foo FOO   foo help
```

必要に応じて、この `help` オプションを無効にする場合があります。これは `ArgumentParser` の `add_help=` 引数に `False` を渡すことで可能です:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [--foo FOO]

optional arguments:
  --foo FOO   foo help
```

ヘルプオプションは通常 `-h/--help` です。例外は `prefix_chars=` が指定されてその中に `-` が無かった場合で、その場合は `-h` と `--help` は有効なオプションではありません。この場合、`prefix_chars` の最初の文字がヘルプオプションの接頭辞として利用されます:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]

optional arguments:
  +h, ++help  show this help message and exit
```

15.4.3 add_argument() メソッド

`ArgumentParser.add_argument` (*name or flags...* [, *action*] [, *nargs*] [, *const*] [, *default*] [, *type*] [, *choices*] [, *required*] [, *help*] [, *metavar*] [, *dest*])

1 つのコマンドライン引数がどう解析されるかを定義します。各引数についての詳細は後述しますが、簡単に言うと:

- *name or flags* - 名前か、あるいはオプション文字列のリスト (例: `foo` や `-f`, `--foo`)。
- *action* - コマンドラインにこの引数があった時のアクション。
- *nargs* - 消費すべきコマンドライン引数の数。
- *const* - 一部の *action* と *nargs* の組み合わせで利用される定数。

- *default* - コマンドラインに引数がなかった場合に生成される値。
- *type* - コマンドライン引数が変換されるべき型。
- *choices* - 引数として許される値のコンテナー。
- *required* - コマンドラインオプションが省略可能かどうか (オプション引数のみ)。
- *help* - 引数が何なのかを示す簡潔な説明。
- *metavar* - 使用法メッセージの中で使われる引数の名前。
- *dest* - `parse_args()` が返すオブジェクトに追加される属性名。

以下の節では各オプションの利用方法を説明します。

name or flags

`add_argument()` メソッドは、指定されている引数が `-f` や `--foo` のようなオプション引数なのか、ファイル名リストなどの位置引数なのかを知る必要があります。そのため、`add_argument()` の第 1 引数は、フラグのリストか、シンプルな引数名のどちらかになります。例えば、オプション引数は次のようにして作成します:

```
>>> parser.add_argument('-f', '--foo')
```

一方、位置引数は次のように作成します:

```
>>> parser.add_argument('bar')
```

`parse_args()` が呼ばれた時、オプション引数は接頭辞 `-` により識別され、それ以外の引数は位置引数として扱われます:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
Namespace(bar='BAR', foo='FOO')
>>> parser.parse_args(['--foo', 'FOO'])
usage: PROG [-h] [-f FOO] bar
PROG: error: too few arguments
```

action

`ArgumentParser` オブジェクトはコマンドライン引数にアクションを割り当てます。このアクションは、割り当てられたコマンドライン引数に関してどんな処理でもできますが、ほとんどのアクションは単に `parse_args()` が返すオブジェクトに属性を追加するだけです。action キーワード引数は、コマンドライン引数がどう処理されるかを指定します。提供されているアクションは:

- 'store' - これは単に引数の値を格納します。これはデフォルトのアクションです。例えば:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args('--foo 1'.split())
Namespace(foo='1')
```

- 'store_const' - このアクションは *const* キーワード引数で指定された値を格納します。
'store_const' アクションは、何かの種類のフラグを指定するオプション引数によく使われます。例えば:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_const', const=42)
>>> parser.parse_args(['--foo'])
Namespace(foo=42)
```

- 'store_true', 'store_false' - これらは 'store_const' の特別版で、それぞれ True と False を格納するのに使います。さらに、これらはデフォルト値をそれぞれ False と True にします。例えば:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('--bar', action='store_false')
>>> parser.add_argument('--baz', action='store_false')
>>> parser.parse_args('--foo --bar'.split())
Namespace(bar=False, baz=True, foo=True)
```

- 'append' - このアクションはリストを格納して、各引数の値をそのリストに追加します。このアクションは複数回指定を許可したいオプションに便利です。利用例:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append')
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['1', '2'])
```

- 'append_const' - このアクションはリストを格納して、*const* キーワード引数に与えられた値をそのリストに追加します (*const* キーワード引数のデフォルト値はあまり役に立たない None であることに注意)。
'append_const' アクションは、定数を同じリストに複数回格納する場合に便利です。例えば:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const', ↵
↵const=str)
>>> parser.add_argument('--int', dest='types', action='append_const', ↵
↵const=int)
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<type 'str'>, <type 'int'>])
```

- 'count' - このアクションはキーワード引数の数を数えます。例えば、verbose レベルを上げるのに役立ちます:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count')
>>> parser.parse_args(['-vvv'])
Namespace(verbose=3)
```

- 'help' - このアクションは現在のパーサー中のすべてのオプションのヘルプメッセージを表示し、終了します。出力の生成方法の詳細については [ArgumentParser](#) を参照してください。
- 'version' - このアクションは [add_argument\(\)](#) の呼び出しに `version=` キーワード引数を期待します。指定されたときはバージョン情報を表示して終了します:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--version', action='version', version='% (prog)s 2.0')
>>> parser.parse_args(['--version'])
PROG 2.0
```

Action のサブクラスまたは同じインターフェイスを実装したほかのオブジェクト渡すことで、任意のアクションを指定することもできます。これをするお奨めの方法は、[argparse.Action](#) を継承して、`__call__` と、必要であれば `__init__` をオーバーライドすることです。

カスタムアクションの例です:

```
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None, **kwargs):
...         if nargs is not None:
...             raise ValueError("nargs not allowed")
...         super(FooAction, self).__init__(option_strings, dest, **kwargs)
...     def __call__(self, parser, namespace, values, option_string=None):
...         print '%r %r %r' % (namespace, values, option_string)
...         setattr(namespace, self.dest, values)
...
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=FooAction)
>>> parser.add_argument('bar', action=FooAction)
>>> args = parser.parse_args('1 --foo 2'.split())
Namespace(bar=None, foo=None) '1' None
Namespace(bar='1', foo=None) '2' '--foo'
>>> args
Namespace(bar='1', foo='2')
```

詳細は [Action](#) を参照してください。

nargs

ArgumentParser オブジェクトは通常 1 つのコマンドライン引数を 1 つのアクションに渡します。nargs キーワード引数は 1 つのアクションにそれ以外の数のコマンドライン引数を割り当てます。指定できる値は:

- N (整数) – N 個の引数がコマンドラインから集められ、リストに格納されます。例えば:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs=2)
>>> parser.add_argument('bar', nargs=1)
>>> parser.parse_args('c --foo a b'.split())
Namespace(bar=['c'], foo=['a', 'b'])
```

`nargs=1` は 1 要素のリストを作ることに注意してください。これはデフォルトの、要素がそのまま属性になる動作とは異なります。

- `'?'` – 可能なら 1 つの引数がコマンドラインから取られ、1 つのアイテムを作ります。コマンドライン引数が存在しない場合、*default* の値が生成されます。オプション引数の場合、さらにオプション引数が指定され、その後にコマンドライン引数が無いというケースもあります。この場合は *const* の値が生成されます。この動作の例です:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='?', const='c', default='d')
>>> parser.add_argument('bar', nargs='?', default='d')
>>> parser.parse_args(['XX', '--foo', 'YY'])
Namespace(bar='XX', foo='YY')
>>> parser.parse_args(['XX', '--foo'])
Namespace(bar='XX', foo='c')
>>> parser.parse_args([])
Namespace(bar='d', foo='d')
```

`nargs='?'` のよくある利用例の 1 つは、入出力ファイルの指定オプションです:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', nargs='?', type=argparse.FileType('r'),
...                     default=sys.stdin)
>>> parser.add_argument('outfile', nargs='?', type=argparse.FileType('w'),
...                     default=sys.stdout)
>>> parser.parse_args(['input.txt', 'output.txt'])
Namespace(infile=<open file 'input.txt', mode 'r' at 0x...>,
           outfile=<open file 'output.txt', mode 'w' at 0x...>)
>>> parser.parse_args([])
Namespace(infile=<open file '<stdin>', mode 'r' at 0x...>,
           outfile=<open file '<stdout>', mode 'w' at 0x...>)
```

- `'*'` – すべてのコマンドライン引数がリストに集められます。複数の位置引数が `nargs='*'` を持つことにあまり意味はありませんが、複数のオプション引数が `nargs='*'` を持つことはありえます。例えば:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='*')
>>> parser.add_argument('--bar', nargs='*')
>>> parser.add_argument('baz', nargs='*')
>>> parser.parse_args('a b --foo x y --bar 1 2'.split())
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])
```

- `'+'` – `'*'` と同じように、すべてのコマンドライン引数をリストに集めます。加えて、最低でも 1 つ

のコマンドライン引数が存在しない場合にエラーメッセージを生成します。例えば:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.parse_args(['a', 'b'])
Namespace(foo=['a', 'b'])
>>> parser.parse_args([])
usage: PROG [-h] foo [foo ...]
PROG: error: too few arguments
```

- `argparse.REMAINDER` – コマンドライン引数の残りすべてをリストとして集めます。これは他のコマンドラインツールに対して処理を渡すようなツールによく使われます。例えば:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo')
>>> parser.add_argument('command')
>>> parser.add_argument('args', nargs=argparse.REMAINDER)
>>> print parser.parse_args('--foo B cmd --arg1 XX ZZ'.split())
Namespace(args=['--arg1', 'XX', 'ZZ'], command='cmd', foo='B')
```

`nargs` キーワード引数が指定されない場合、消費される引数の数は *action* によって決定されます。通常これは、1つのコマンドライン引数は1つのアイテムになる(リストにはならない)ことを意味します。

const

`add_argument()` の `const` 引数は、コマンドライン引数から読み込まれないけれども `ArgumentParser` のいくつかのアクションで必要とされる値のために使われます。この引数のよくある2つの使用法は:

- `add_argument()` が `action='store_const'` か `action='append_const'` で呼び出された時、これらのアクションは `const` の値を `parse_args()` が返すオブジェクトの属性に追加します。サンプルは *action* の説明を参照してください。
- `add_argument()` がオプション文字列 (`-f` や `--foo`) と `nargs='?'` で呼び出された場合。この場合0個か1つのコマンドライン引数を取るオプション引数が作られます。オプション引数にコマンドライン引数が続かなかった場合、`const` の値が代わりに利用されます。サンプルは *nargs* の説明を参照してください。

'`store_const`' と '`append_const`' アクションでは、`const` キーワード引数を与える必要があります。他のアクションでは、デフォルトは `None` になります。

default

すべてのオプション引数といくつかの位置引数はコマンドライン上で省略されることがあります。`add_argument()` の `default` キーワード引数 (デフォルト: `None`) は、コマンドライン引数が存在しなかった場合に利用する値を指定します。オプション引数では、オプション文字列がコマンドライン上に存在しなかったときに `default` の値が利用されます:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
>>> parser.parse_args([])
Namespace(foo=42)
```

default の値が文字列の場合、パーサーは値をコマンドライン引数のように解析します。具体的には、パーサーは返り値 *Namespace* の属性を設定する前に、*type* 変換引数が与えられていればそれらを適用します。そうでない場合、パーサーは値をそのまま使用します:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

nargs が ? か * である位置引数では、コマンドライン引数が指定されなかった場合 default の値が使われま
す。例えば:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

default=argparse.SUPPRESS を渡すと、コマンドライン引数が存在しないときに属性の追加をしなくなります。:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

type

デフォルトでは、*ArgumentParser* オブジェクトはコマンドライン引数を単なる文字列として読み込みます。しかし、コマンドラインの文字列は *float*, *int* など別の型として扱うべき事がよくあります。*add_argument()* の *type* キーワード引数により型チェックと型変換を行うことができます。一般的なビルトインデータ型や関数を *type* 引数の値として直接指定できます:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.add_argument('bar', type=file)
>>> parser.parse_args('2 temp.txt'.split())
```

(次のページに続く)

(前のページからの続き)

```
Namespace(bar=<open file 'temp.txt', mode 'r' at 0x...>, foo=2)
```

`type` 引数がデフォルト引数に適用されている場合の情報は、[default](#) キーワード引数の節を参照してください。

いろいろな種類のファイルを簡単に扱うために、`argparse` モジュールは `file` オブジェクトの `mode=`, `bufsize=` 引数を取る `FileType` ファクトリを提供しています。例えば、書き込み可能なファイルを作るために `FileType('w')` を利用できます:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar', type=argparse.FileType('w'))
>>> parser.parse_args(['out.txt'])
Namespace(bar=<open file 'out.txt', mode 'w' at 0x...>)
```

`type=` には 1 つの文字列を引数に受け取って変換結果を返すような任意の呼び出し可能オブジェクトを渡すことができます:

```
>>> def perfect_square(string):
...     value = int(string)
...     sqrt = math.sqrt(value)
...     if sqrt != int(sqrt):
...         msg = "%r is not a perfect square" % string
...         raise argparse.ArgumentTypeError(msg)
...     return value
...
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=perfect_square)
>>> parser.parse_args(['9'])
Namespace(foo=9)
>>> parser.parse_args(['7'])
usage: PROG [-h] foo
PROG: error: argument foo: '7' is not a perfect square
```

さらに、[choices](#) キーワード引数を使って、値の範囲をチェックすることもできます:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=int, choices=xrange(5, 10))
>>> parser.parse_args(['7'])
Namespace(foo=7)
>>> parser.parse_args(['11'])
usage: PROG [-h] {5,6,7,8,9}
PROG: error: argument foo: invalid choice: 11 (choose from 5, 6, 7, 8, 9)
```

詳細は [choices](#) 節を参照してください。

choices

コマンドライン引数をいくつかの選択肢の中から選ばせたい場合があります。これは `add_argument()` に `choices` キーワード引数を渡すことで可能です。コマンドラインを解析する時、引数の値がチェックされ、そ

の値が選択肢の中に含まれていない場合はエラーメッセージを表示します:

```
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock',
'paper', 'scissors')
```

choices コンテナに含まれているかどうかのチェックは、*type* による型変換が実行された後であることに注意してください。なので、*choices* に格納する オブジェクトの型は指定された *type* にマッチしている必要があります:

```
>>> parser = argparse.ArgumentParser(prog='doors.py')
>>> parser.add_argument('door', type=int, choices=range(1, 4))
>>> print(parser.parse_args(['3']))
Namespace(door=3)
>>> parser.parse_args(['4'])
usage: doors.py [-h] {1,2,3}
doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)
```

`in` 演算をサポートしている任意のオブジェクトを *choices* に渡すことができます。なので、*dict*, *set*, その他カスタムコンテナなどは全てサポートしています。

required

通常 *argparse* モジュールは、`-f` や `--bar` といったフラグは 任意 の引数 (オプション引数) だと仮定し、コマンドライン上になくても良いものとして扱います。フラグの指定を 必須 にするには、*add_argument()* の *required=* キーワード引数に `True` を指定します:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
usage: argparse.py [-h] [--foo FOO]
argparse.py: error: option --foo is required
```

上の例のように、引数が *required* と指定されると、*parse_args()* はそのフラグがコマンドラインに存在しないときにエラーを表示します。

注釈: ユーザーは、通常 フラグ の指定は 任意 であると認識しているため、必須にするのは一般的には悪いやり方で、できる限り避けるべきです。

help

`help` の値はその引数の簡潔な説明を含む文字列です。ユーザーが (コマンドライン上で `-h` か `--help` を指定するなどして) ヘルプを要求したとき、この `help` の説明が各引数に表示されます:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', action='store_true',
...                       help='foo the bars before frobbling')
>>> parser.add_argument('bar', nargs='+',
...                       help='one of the bars to be frobbled')
>>> parser.parse_args(['-h'])
usage: frobble [-h] [--foo] bar [bar ...]

positional arguments:
  bar      one of the bars to be frobbled

optional arguments:
  -h, --help  show this help message and exit
  --foo      foo the bars before frobbling
```

`help` 文字列には、プログラム名や引数の *default* などを繰り返し記述するのを避けるためのフォーマット指定子を含めることができます。利用できる指定子には、プログラム名 `%(prog)s` と、`%(default)s` や `%(type)s` など `add_argument()` のキーワード引数の多くが含まれます:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
...                       help='the bar to %(prog)s (default: %(default)s)')
>>> parser.print_help()
usage: frobble [-h] [bar]

positional arguments:
  bar      the bar to frobble (default: 42)

optional arguments:
  -h, --help  show this help message and exit
```

`argparse` は `help` に `argparse.SUPPRESS` を設定することで、特定のオプションをヘルプに表示させないことができます:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]

optional arguments:
  -h, --help  show this help message and exit
```

metavar

`ArgumentParser` がヘルプメッセージを出力する時、各引数に対してなんらかの参照方法が必要です。デ

フォルトでは、ArgumentParser オブジェクトは各オブジェクトの”名前”として *dest* を利用します。デフォルトでは、位置引数には *dest* の値をそのまま 利用し、オプション引数については *dest* の値を大文字に変換して 利用します。なので、1 つの *dest*='bar' である位置引数は bar として参照されます。1 つのオプション引数 --foo が 1 つのコマンドライン引数を要求するときは、その引数は FOO として参照されます。例です:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage: [-h] [--foo FOO] bar

positional arguments:
  bar

optional arguments:
  -h, --help  show this help message and exit
  --foo FOO
```

代わりの名前を、metavar として指定できます:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage: [-h] [--foo YYY] XXX

positional arguments:
  XXX

optional arguments:
  -h, --help  show this help message and exit
  --foo YYY
```

metavar は 表示される 名前だけを変更することに注意してください。 *parse_args()* の返すオブジェクトの属性名は *dest* の値のままです。

nargs を指定した場合、metavar が複数回利用されるかもしれません。metavar にタプルを渡すと、各引数に対して異なる名前を指定できます:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs=2)
>>> parser.add_argument('--foo', nargs=2, metavar=('bar', 'baz'))
>>> parser.print_help()
usage: PROG [-h] [-x X X] [--foo bar baz]

optional arguments:
  -h, --help  show this help message and exit
```

(次のページに続く)

(前のページからの続き)

```
-x X X
--foo bar baz
```

dest

ほとんどの `ArgumentParser` のアクションは `parse_args()` が返すオブジェクトに対する属性として値を追加します。この属性の名前は `add_argument()` の `dest` キーワード引数によって決定されます。位置引数のアクションについては、`dest` は通常 `add_argument()` の第一引数として渡します:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

オプション引数のアクションについては、`dest` の値は通常オプション文字列から生成されます。`ArgumentParser` は最初の長いオプション文字列を選択し、先頭の `--` を除去することで `dest` の値を生成します。長いオプション文字列が指定されていない場合、最初の短いオプション文字列から先頭の `-` 文字を除去することで `dest` を生成します。先頭以外のすべての `-` 文字は、妥当な属性名になるように `_` 文字へ変換されます。次の例はこの動作を示しています:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args('-f 1 -x 2'.split())
Namespace(foo_bar='1', x='2')
>>> parser.parse_args('--foo 1 -y 2'.split())
Namespace(foo_bar='1', x='2')
```

`dest` にカスタムの属性名を与えることも可能です:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

Action クラス

Action クラスは Action API、すなわちコマンドラインからの引数进行处理する呼び出し可能オブジェクトを返す呼び出し可能オブジェクトを実装します。この API に従うあらゆるオブジェクトは `action` 引数として `add_argument()` に渡すことが出来ます。

```
class argparse.Action(option_strings, dest, nargs=None, const=None, default=None,
                      type=None, choices=None, required=False, help=None,
                      metavar=None)
```

Action オブジェクトは、コマンドラインからの一つ以上の文字列から単一の引数を解析するのに必要とされる情報を表現するために、ArgumentParser によって使われます。Action クラス 2 つの位置引数と、`action` そ

れ自身を除く `ArgumentParser.add_argument()` に渡される全てのキーワード引数を受け付けなければなりません。

Action のインスタンス (あるいは `action` パラメータも渡す任意の callable の返却値) は、属性 `"dest"`, `"option_strings"`, `"default"`, `"type"`, `"required"`, `"help"`, などを定義しなければなりません。これらの属性を定義するのを確実にするために最も簡単な方法は、`Action.__init__` を呼び出すことです。

Action インスタンスは callable でなければならず、従って、サブクラスは 4 つの引数を受け取る `__call__` メソッドをオーバーライドしなければなりません:

- `parser` - このアクションを持っている `ArgumentParser` オブジェクト。
- `namespace` - `parse_args()` が返す `Namespace` オブジェクト。ほとんどのアクションはこのオブジェクトに属性を `setattr()` を使って追加します。
- `values` - 型変換が適用された後の、関連付けられたコマンドライン引数。型変換は `add_argument()` メソッドの `type` キーワード引数で指定されます。
- `option_string` - このアクションを実行したオプション文字列。 `option_string` 引数はオプションで、アクションが位置引数に関連付けられた場合は渡されません。

`__call__` メソッドでは任意のアクションを行えますが、典型的にはそれは `dest`, `values` に基づく `namespace` に属性をセットすることでしょう。

15.4.4 parse_args() メソッド

`ArgumentParser.parse_args(args=None, namespace=None)`

引数の文字列をオブジェクトに変換し、`namespace` オブジェクトの属性に代入します。結果の `namespace` オブジェクトを返します。

事前の `add_argument()` メソッドの呼び出しが、どのオブジェクトが生成されてどう代入されるかを決定します。詳細は `add_argument()` のドキュメントを参照してください。

- `args` - List of strings to parse. The default is taken from `sys.argv`.
- `namespace` - An object to take the attributes. The default is a new empty `Namespace` object.

オプション値の文法

`parse_args()` メソッドはオプションの値があればそれを指定する複数の方法をサポートしています。一番シンプルな方法は、オプションとその値は 2 つの別々の引数として渡されます:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args(['-x', 'X'])
Namespace(foo=None, x='X')
>>> parser.parse_args(['--foo', 'FOO'])
Namespace(foo='FOO', x=None)
```

長いオプション (1 文字よりも長い名前を持ったオプション) では、オプションとその値は `=` で区切られた 1 つのコマンドライン引数として渡すこともできます:

```
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

短いオプション (1 文字のオプション) では、オプションとその値は連結して渡すことができます:

```
>>> parser.parse_args(['-xX'])
Namespace(foo=None, x='X')
```

複数の短いオプションは、最後の 1 つ (か、0 個) のオプションだけが値を要求する場合には、1 つの接頭辞だけで連結できます:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', action='store_true')
>>> parser.add_argument('-y', action='store_true')
>>> parser.add_argument('-z')
>>> parser.parse_args(['-xyzZ'])
Namespace(x=True, y=True, z='Z')
```

不正な引数

`parse_args()` は、コマンドラインの解析中に、曖昧なオプション、不正な型、不正なオプション、位置引数の数の不一致などのエラーを検証します。それらのエラーが発生した場合、エラーメッセージと使用方法メッセージを表示して終了します:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')

>>> # invalid type
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'

>>> # invalid option
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

>>> # wrong number of arguments
>>> parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
```

- を含む引数

`parse_args()` メソッドは、ユーザーが明らかなミスをした場合はエラーを表示しますが、いくつか本質的

に曖昧な場面があります。例えば、コマンドライン引数 `-1` は、オプションの指定かもしれませんが位置引数かもしれません。`parse_args()` メソッドはこれを次のように扱います: 負の数として解釈でき、パーサーに負の数のように解釈できるオプションが存在しない場合にのみ、`-` で始まる位置引数になりえます:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional argument
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional arguments
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')

>>> # negative number options present, so -1 is an option
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')

>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

>>> # negative number options present, so both -1s are options
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

`-` で始まる位置引数があって、それが負の数として解釈できない場合、ダミーの引数 `---` を挿入して、`parse_args()` にそれ以降のすべてが位置引数だと教えることができます:

```
>>> parser.parse_args(['---', '-f'])
Namespace(foo='-f', one=None)
```

引数の短縮形 (先頭文字でのマッチング)

`parse_args()` メソッドでは長いオプションを曖昧さが無い範囲 (先頭の文字が一意であること) で短縮できます:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')
>>> parser.parse_args(['-bac MMM'].split())
Namespace(bacon='MMM', badger=None)
```

(次のページに続く)

(前のページからの続き)

```
>>> parser.parse_args('-bad WOOD'.split())
Namespace(bacon=None, badger='WOOD')
>>> parser.parse_args('-ba BA'.split())
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

引数が複数のオプションになり得る場合はエラーになります。

sys.argv 以外

ArgumentParser が `sys.argv` 以外の引数を解析できると役に立つ場合があります。その場合は文字列のリストを `parse_args()` に渡します。これはインタラクティブプロンプトからテストするときに便利です:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=xrange(10),
...     nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(
...     '--sum', dest='accumulate', action='store_const', const=sum,
...     default=max, help='sum the integers (default: find the max)')
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3, 4])
>>> parser.parse_args(['1', '2', '3', '4', '--sum'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```

Namespace オブジェクト

class argparse.Namespace

`parse_args()` が属性を格納して返すためのオブジェクトにデフォルトで使用するシンプルなクラスです。

デフォルトでは、`parse_args()` は *Namespace* の新しいオブジェクトに必要な属性を設定して返します。このクラスはシンプルに設計されており、単に読みやすい文字列表現を持った *object* のサブクラスです。もし属性を辞書のように扱える方が良ければ、標準的な Python のイディオム `vars()` を利用できます:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

ArgumentParser が、新しい *Namespace* オブジェクトではなく、既存のオブジェクトに属性を設定する方が良い場合があります。これは `namespace=` キーワード引数を指定することで可能です:

```
>>> class C(object):
...     pass
... 
```

(次のページに続く)

(前のページからの続き)

```
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

15.4.5 その他のユーティリティ

サブコマンド

`ArgumentParser.add_subparsers([title][, description][, prog][, parser_class][, action][, option_string][, dest][, help][, metavar])`

多くのプログラムは、その機能をサブコマンドへと分割します。例えば `svn` プログラムは `svn checkout`, `svn update`, `svn commit` などのサブコマンドを利用できます。機能をサブコマンドに分割するのは、プログラムがいくつかの異なる機能を持っていて、それぞれが異なるコマンドライン引数を必要とする場合には良いアイデアです。 `ArgumentParser` は `add_subparsers()` メソッドによりサブコマンドをサポートしています。 `add_subparsers()` メソッドは通常引数なしに呼び出され、特殊なアクションオブジェクトを返します。このオブジェクトには1つのメソッド `add_parser()` があり、コマンド名と `ArgumentParser` コンストラクターの任意の引数を受け取り、通常の方法で操作できる `ArgumentParser` オブジェクトを返します。

パラメーターの説明:

- `title` - ヘルプ出力でのサブパーサーグループのタイトル; デフォルトは、`description` が指定されている場合は "subcommands" に、指定されていない場合は位置引数のタイトルになります
- `description` - ヘルプ出力に表示されるサブパーサーグループの説明です。デフォルトは `None` になります
- `prog` - サブコマンドのヘルプに表示される使用方法の説明です。デフォルトではプログラム名と位置引数の後ろに、サブパーサーの引数が続きます
- `parser_class` - サブパーサーのインスタンスを作成する時に使用されるクラスです。デフォルトでは現在のパーサーのクラス (例: `ArgumentParser`) になります
- `action` - コマンドラインにこの引数があった時の基本のアクション。
- `dest` - サブコマンド名を格納する属性の名前です。デフォルトは `None` で値は格納されません
- `help` - ヘルプ出力に表示されるサブパーサーグループのヘルプです。デフォルトは `None` です
- `metavar` - 利用可能なサブコマンドをヘルプ内で表示するための文字列です。デフォルトは `None` で、サブコマンドを `{cmd1, cmd2, ...}` のような形式で表します

いくつかの使用例:

```

>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true', help='foo help')
>>> subparsers = parser.add_subparsers(help='sub-command help')
>>>
>>> # create the parser for the "a" command
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')
>>>
>>> # create the parser for the "b" command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices='XYZ', help='baz help')
>>>
>>> # parse some argument lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)

```

`parse_args()` が返すオブジェクトにはメインパーサーとコマンドラインで選択されたサブパーサーによる属性だけが設定されており、選択されなかったサブコマンドのパーサーの属性が設定されていないことに注意してください。なので、上の例では、a コマンドが指定されたときは `foo`, `bar` 属性だけが存在し、b コマンドが指定されたときは `foo`, `baz` 属性だけが存在しています。

同じように、サブパーサーにヘルプメッセージが要求された場合は、そのパーサーに対するヘルプだけが表示されます。ヘルプメッセージには親パーサーや兄弟パーサーのヘルプメッセージを表示しません。(ただし、各サブパーサーコマンドのヘルプメッセージは、上の例にもあるように `add_parser()` の `help=` 引数によって指定できます)

```

>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}    sub-command help
  a        a help
  b        b help

optional arguments:
  -h, --help  show this help message and exit
  --foo      foo help

>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar    bar help

optional arguments:
  -h, --help  show this help message and exit

>>> parser.parse_args(['b', '--help'])

```

(次のページに続く)

(前のページからの続き)

```
usage: PROG b [-h] [--baz {X,Y,Z}]

optional arguments:
  -h, --help            show this help message and exit
  --baz {X,Y,Z}        baz help
```

`add_subparsers()` メソッドは `title` と `description` キーワード引数もサポートしています。どちらかが存在する場合、サブパーサーのコマンドはヘルプ出力でそれぞれのグループの中に表示されます。例えば:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
...                                   description='valid subcommands',
...                                   help='additional help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage:  [-h] {foo,bar} ...

optional arguments:
  -h, --help  show this help message and exit

subcommands:
  valid subcommands

  {foo,bar}  additional help
```

サブコマンドを扱う 1 つの便利な方法は `add_subparsers()` メソッドと `set_defaults()` を組み合わせて、各サブパーサーにどの Python 関数を実行するかを教えることです。例えば:

```
>>> # sub-command functions
>>> def foo(args):
...     print args.x * args.y
...
>>> def bar(args):
...     print '(%s)' % args.z
...
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>>
>>> # create the parser for the "foo" command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)
>>>
>>> # create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
```

(次のページに続く)

(前のページからの続き)

```
>>> parser_bar.set_defaults(func=bar)
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
2.0
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
((XYZYX))
```

こうすると、`parse_args()` が引数の解析が終わってから適切な関数を呼び出すようになります。このように関数をアクションに関連付けるのは大抵サブパーサーごとに異なるアクションを扱う最も簡単な方法です。ただし、実行されたサブパーサーの名前を確認する必要がある場合は、`add_subparsers()` を呼び出すときに `dest` キーワードを指定できます:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

FileType オブジェクト

class `argparse.FileType` (*mode='r', bufsize=None*)

`FileType` ファクトリは `ArgumentParser.add_argument()` の `type` 引数に渡すことができるオブジェクトを生成します。 `type` が `FileType` オブジェクトである引数はコマンドライン引数を、指定されたモード、バッファサイズでファイルとして開きます:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--output', type=argparse.FileType('wb', 0))
>>> parser.parse_args(['--output', 'out'])
Namespace(output=<open file 'out', mode 'wb' at 0x...>)
```

`FileType` オブジェクトは擬似引数 `-` を識別し、読み込み用の `FileType` であれば `sys.stdin` を、書き込み用の `FileType` であれば `sys.stdout` に変換します:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<open file '<stdin>', mode 'r' at 0x...>)
```

引数グループ

`ArgumentParser.add_argument_group(title=None, description=None)`

デフォルトでは、`ArgumentParser` はヘルプメッセージを表示するときに、コマンドライン引数を”位置引数”と”オプション引数”にグループ化します。このデフォルトの動作よりも良い引数のグループ化方法がある場合、`add_argument_group()` メソッドで適切なグループを作成できます:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
>>> parser.print_help()
usage: PROG [--foo FOO] bar

group:
  bar      bar help
  --foo FOO  foo help
```

`add_argument_group()` メソッドは、通常の `ArgumentParser` と同じような `add_argument()` メソッドを持つ引数グループオブジェクトを返します。引数がグループに追加された時、パーサーはその引数を通常の引数のように扱いますが、ヘルプメッセージではその引数を分離されたグループの中に表示します。`add_argument_group()` メソッドには、この表示をカスタマイズするための `title` と `description` 引数があります:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo

group1:
  group1 description

  foo      foo help

group2:
  group2 description

  --bar BAR  bar help
```

ユーザー定義グループにないすべての引数は通常の”位置引数”と”オプション引数”セクションに表示されます。

相互排他

`ArgumentParser.add_mutually_exclusive_group(required=False)`

相互排他グループを作ります。`argparse` は相互排他グループの中でただ 1 つの引数のみが存在することを確認します:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo
```

`add_mutually_exclusive_group()` メソッドの引数 *required* に True 値を指定すると、その相互排他引数のどれか 1 つを選ぶことが要求されます:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required
```

現在のところ、相互排他引数グループは `add_argument_group()` の *title* と *description* 引数をサポートしていません。

パーサーのデフォルト値

`ArgumentParser.set_defaults(**kwargs)`

ほとんどの場合、`parse_args()` が返すオブジェクトの属性はコマンドライン引数の内容と引数のアクションによってのみ決定されます。`set_defaults()` を使うと与えられたコマンドライン引数の内容によらず追加の属性を決定することが可能です:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)
```

パーサーレベルのデフォルト値は常に引数レベルのデフォルト値を上書きします:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')
```

パーサーレベルの `default` は、複数のパーサーを扱うときに特に便利です。このタイプの例については `add_subparsers()` メソッドを参照してください。

`ArgumentParser.get_default(dest)`

`add_argument()` が `set_defaults()` によって指定された、`namespace` の属性のデフォルト値を取得します:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='badger')
>>> parser.get_default('foo')
'badger'
```

ヘルプの表示

ほとんどの典型的なアプリケーションでは、`parse_args()` が使用法やエラーメッセージのフォーマットと表示について面倒を見ます。しかし、いくつかのフォーマットメソッドが利用できます:

`ArgumentParser.print_usage(file=None)`

`ArgumentParser` がコマンドラインからどう実行されるべきかの短い説明を表示します。 `file` が `None` の時は、 `sys.stdout` に出力されます。

`ArgumentParser.print_help(file=None)`

プログラムの使用法と `ArgumentParser` に登録された引数についての情報を含むヘルプメッセージを表示します。 `file` が `None` の時は、 `sys.stdout` に出力されます。

これらのメソッドの、表示する代わりにシンプルに文字列を返すバージョンもあります:

`ArgumentParser.format_usage()`

`ArgumentParser` がコマンドラインからどう実行されるべきかの短い説明を格納した文字列を返します。

`ArgumentParser.format_help()`

プログラムの使用法と `ArgumentParser` に登録された引数についての情報を含むヘルプメッセージを格納した文字列を返します。

部分解析

`ArgumentParser.parse_known_args(args=None, namespace=None)`

ときどき、スクリプトがコマンドライン引数のいくつかだけを解析し、残りの引数は別のスクリプトやプログラムに渡すことがあります。こういった場合、 `parse_known_args()` メソッドが便利です。これは `parse_args()` と同じように動作しますが、余分な引数が存在してもエラーを生成しません。代わりに、評価された `namespace` オブジェクトと、残りの引数文字列のリストからなる 2 要素タプルを返します。

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

警告: 先頭文字でのマッチングルールは `parse_known_args()` にも適用されます。パーサーはそれが既知のオプションの先頭文字の一つであった場合、それを引数リストに残さず 1 個のオプションの消費とすることがあります。

ファイル解析のカスタマイズ

`ArgumentParser.convert_arg_line_to_args(arg_line)`

ファイルから引数を読み込む場合 (`ArgumentParser` コンストラクターの `fromfile_prefix_chars` キーワード引数を参照)、1 行につき 1 つの引数を読み込みます。 `convert_arg_line_to_args()` を変更することでこの動作をカスタマイズできます。

このメソッドは、引数ファイルから読まれた文字列である 1 つの引数 `arg_line` を受け取ります。そしてその文字列を解析した結果の引数のリストを返します。このメソッドはファイルから 1 行読みこむごとに、順番に呼ばれます。

このメソッドを上書きする便利な例として、スペースで区切られた単語を 1 つの引数として扱います:

```
def convert_arg_line_to_args(self, arg_line):  
    return arg_line.split()
```

終了メソッド

`ArgumentParser.exit(status=0, message=None)`

このメソッドは、`message` が指定されていればそれを表示した後、指定された終了ステータス `status` でプログラムを終了します。

`ArgumentParser.error(message)`

このメソッドは `message` を含む使用法メッセージを標準エラーに表示して、終了ステータス 2 でプログラムを終了します。

15.4.6 optparse からのアップグレード

もともと、`argparse` モジュールは `optparse` モジュールとの互換性を保って開発しようと試みられました。しかし、特に新しい `nargs=` 指定子とより良い使用法メッセージのために必要な変更のために、`optparse` を透過的に拡張することは難しかったのです。`optparse` のほとんどすべてがコピーアンドペーストされたリモンキーパッチを当てられたりしたとき、もはや後方互換性を保とうとすることは現実的ではありませんでした。

`argparse` モジュールは標準ライブラリ `optparse` モジュールを、以下を含むたくさんの方法で改善しています:

- 位置指定引数を扱う
- サブコマンドのサポート

- +, / のような代替オプションプレフィクスを許容する
- zero-or-more スタイル、one-or-more スタイルの引数を扱う
- より有益な使用方法メッセージの生成
- カスタム type, カスタム action のために遥かに簡単なインターフェイスを提供する

`optparse` から `argparse` への現実的なアップグレードパス:

- すべての `optparse.OptionParser.add_option()` の呼び出しを、`ArgumentParser.add_argument()` の呼び出しに置き換える。
- Replace `(options, args) = parser.parse_args()` with `args = parser.parse_args()` and add additional `ArgumentParser.add_argument()` calls for the positional arguments. Keep in mind that what was previously called options, now in the `argparse` context is called args.
- Replace `optparse.OptionParser.disable_interspersed_args()` by setting nargs of a positional argument to `argparse.REMAINDER`, or use `parse_known_args()` to collect unparsed argument strings in a separate list.
- コールバック・アクションと `callback_*` キーワード引数を `type` や `action` 引数に置き換える。
- `type` キーワード引数に渡していた文字列の名前を、それに応じたオブジェクト (例: int, float, complex, ...) に置き換える。
- `optparse.Values` を `Namespace` に置き換え、`optparse.OptionError` と `optparse.OptionValueError` を `ArgumentError` に置き換える。
- `%default` や `%prog` などの暗黙の引数を含む文字列を、`%(default)s` や `%(prog)s` などの、通常の Python で辞書を使う場合のフォーマット文字列に置き換える。
- `OptionParser` のコンストラクターの `version` 引数を、`parser.add_argument('--version', action='version', version='<the version>')` に置き換える

15.5 optparse — コマンドラインオプション解析器

バージョン 2.3 で追加.

バージョン 2.7 で非推奨: `optparse` モジュールは廃止予定であり、これ以上の開発は行われません。
`argparse` モジュールを使用してください。

ソースコード: [Lib/optparse.py](#)

`optparse` モジュールは、昔からある `getopt` よりも簡便で、柔軟性に富み、かつ強力なコマンドライン解析ライブラリです。`optparse` では、より宣言的なスタイルのコマンドライン解析手法、すなわち `OptionParser` のインスタンスを作成してオプションを追加してゆき、そのインスタンスでコマンドライン

を解析するという手法をとっています。 `optparse` を使うと、GNU/POSIX 構文でオプションを指定できるだけでなく、使用法やヘルプメッセージの生成も行えます。

`optparse` を使った簡単なスクリプトの例を以下に示します:

```
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```

このようにわずかな行数のコードによって、スクリプトのユーザはコマンドライン上で例えば以下のような「よくある使い方」を実行できるようになります:

```
<yourscript> --file=outfile -q
```

コマンドライン解析の中で、`optparse` はユーザの指定したコマンドライン引数値に応じて `parse_args()` の返す `options` の属性値を設定してゆきます。 `parse_args()` がコマンドライン解析から処理を戻したとき、`options.filename` は "outfile" に、`options.verbose` は `False` になっているはずです。`optparse` は長い形式と短い形式の両方のオプション表記をサポートしており、短い形式は結合して指定できます。また、様々な形でオプションに引数値を関連付けられます。従って、以下のコマンドラインは全て上の例と同じ意味になります:

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

さらに、ユーザが以下のいずれかを実行すると

```
<yourscript> -h
<yourscript> --help
```

`optparse` はスクリプトのオプションについて簡単にまとめた内容を出力します:

```
Usage: <yourscript> [options]

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  write report to FILE
  -q, --quiet           don't print status messages to stdout
```

`yourscript` の中身は実行時に決まります (通常は `sys.argv[0]` になります)。

15.5.1 背景

`optparse` は、素直で慣習に則ったコマンドラインインタフェースを備えたプログラムの作成を援助する目的で設計されました。その結果、Unix で慣習的に使われているコマンドラインの構文や機能だけをサポートするに留まっています。こうした慣習に詳しくなければ、よく知っておくためにもこの節を読んでおきましょう。

用語集

引数 (argument) コマンドラインでユーザが入力するテキストの塊で、シェルが `execl` や `execv` に引き渡すものです。Python では、引数は `sys.argv[1:]` の要素となります。(`sys.argv[0]` は実行しようとしているプログラムの名前です。引数解析に関しては、この要素はあまり重要ではありません。) Unix シェルでは、「語 (word)」という用語も使います。

場合によっては `sys.argv[1:]` 以外の引数リストを代入する方が望ましいことがあるので、「引数」は「`sys.argv[1:]` または `sys.argv[1:]` の代替として提供される別のリストの要素」と読むべきでしょう。

オプション (option) 追加的な情報を与えるための引数で、プログラムの実行に対する教示やカスタマイズを行います。オプションには多様な文法が存在します。伝統的な Unix における書法はハイフン (‘-’) の後ろに一文字が続くもので、例えば `-x` や `-F` です。また、伝統的な Unix における書法では、複数のオプションを一つの引数にまとめられます。例えば `-x -F` は `-xF` と等価です。GNU プロジェクトでは `--` の後ろにハイフンで区切りの語を指定する方法、例えば `--file` や `--dry-run` も提供しています。`optparse` は、これら二種類のオプション書法だけをサポートしています。

他に見られる他のオプション書法には以下のようなものがあります:

- ハイフンの後ろに数個の文字が続くもので、例えば `-pf` (このオプションは複数のオプションを一つにまとめたものとは違います)
- ハイフンの後ろに語が続くもので、例えば `-file` (これは技術的には上の書式と同じですが、通常同じプログラム上で一緒に使うことはありません)
- プラス記号の後ろに一文字、数個の文字、または語を続けたもので、例えば `+f`, `+rgb`
- スラッシュ記号の後ろに一文字、数個の文字、または語を続けたもので、例えば `/f`, `/file`

上記のオプション書法は `optparse` ではサポートしておらず、今後もサポートする予定はありません。これは故意によるものです: 最初の三つはどの環境の標準でもなく、最後の一つは VMS や MS-DOS、そして Windows を対象にしているときにしか意味をなさないからです。

オプション引数 (option argument) あるオプションの後ろに続く引数で、そのオプションに密接な関連を持ち、オプションと同時に引数リストから取り出されます。`optparse` では、オプション引数は以下のように別々の引数にできます:

```
-f foo
--file foo
```

また、一つの引数中にも入れられます:

```
-ffoo
--file=foo
```

通常、オプションは引数をとることとらないこともあります。あるオプションは引数をとることがなく、またあるオプションは常に引数をとります。多くの人々が「オプションのオプション引数」機能を欲しています。これは、あるオプションが引数が指定されている場合には引数を取り、そうでない場合には引数をもたないようにするという機能です。この機能は引数解析をあいまいにするため、議論的となっています: 例えば、もし `-a` がオプション引数を取り、`-b` がまったく別のオプションだとしたら、`-ab` をどうやって解析すればいいのでしょうか? こうした曖昧さが存在するため、`optparse` は今のところこの機能をサポートしていません。

位置引数 (positional argument、固定引数) 他のオプションが解析される、すなわち他のオプションとその引数が解析されて引数リストから除去された後に引数リストに置かれているものです。

必須のオプション (required option) コマンドラインで与えなければならないオプションです; 「必須のオプション (required option)」という語は、英語では矛盾した言葉です。 `optparse` では必須オプションの実装を妨げてはいませんが、とりたてて実装上役立つこともしていません。

例えば、下記のような架空のコマンドラインを考えてみましょう:

```
prog -v --report report.txt foo bar
```

`-v` と `--report` はどちらもオプションです。 `--report` オプションが引数をとるとすれば、`report.txt` はオプションの引数です。 `foo` と `bar` は固定引数になります。

オプションとは何か

オプションはプログラムの実行を調整したり、カスタマイズしたりするための補助的な情報を与えるために使います。もっとはっきりいうと、オプションはあくまでもオプション (省略可能) であるということです。本来、プログラムはともかくもオプションなしでうまく実行できてしかるべきです。(Unix や GNU ツールセットのプログラムをランダムにピックアップしてみてください。オプションを全く指定しなくてもちゃんと動くでしょう? 例外は `find`, `tar`, `dd` くらいです—これらの例外は、オプション文法が標準的でなく、インタフェースが混乱を招くと酷評されてきた変種のはみ出しものなのです)

多くの人が自分のプログラムに「必須のオプション」を持たせたいと考えます。しかしよく考えてください。必須なら、それは オプション (省略可能) ではないのです! プログラムを正しく動作させるのに絶対に必要な情報があるとすれば、そこには固定引数を割り当てるべきなのです。

良くできたコマンドラインインタフェース設計として、ファイルのコピーに使われる `cp` ユーティリティのことを考えてみましょう。ファイルのコピーでは、コピー先を指定せずにファイルをコピーするのは無意味な操作ですし、少なくとも一つのコピー元が必要です。従って、`cp` は引数無しで実行すると失敗します。とはいえ、`cp` はオプションを全く必要としない柔軟で便利なコマンドライン文法を備えています:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

まだあります。ほとんどの `cp` の実装では、ファイルモードや変更時刻を変えずにコピーする、シンボリック

リンクの追跡を行わない、すでにあるファイルを上書きする前にユーザに尋ねる、など、ファイルをコピーする方法をいじるための一連のオプションを実装しています。しかし、こうしたオプションは、一つのファイルを別の場所にコピーする、または複数のファイルを別のディレクトリにコピーするという、`cp` の中心的な処理を乱すことはないのです。

固定引数とは何か

固定引数とは、プログラムを動作させる上で絶対的に必要な情報となる引数です。

よいユーザインタフェースとは、可能な限り少ない固定引数をもつものです。プログラムを正しく動作させるために 17 個もの別個の情報が必要だとしたら、その方法 はさして問題にはなりません — ユーザはプログラムを正しく動作させられないうちに諦め、立ち去ってしまうからです。ユーザインタフェースがコマンドラインでも、設定ファイルでも、GUI やその他の何であっても同じです: 多くの要求をユーザに押し付ければ、ほとんどのユーザはただ音をあげてしまうだけなのです。

要するに、ユーザが絶対に提供しなければならない情報だけに制限する — そして可能な限りよく練られたデフォルト設定を使うよう試みてください。もちろん、プログラムには適度な柔軟性を持たせたいとも望むはありますが、それこそがオプションの果たす役割です。繰り返しますが、設定ファイルのエントリであろうが、GUI でできた「環境設定」ダイアログ上のウィジェットであろうが、コマンドラインオプションであろうが関係ありません — より多くのオプションを実装すればプログラムはより柔軟性を持ちますが、実装はより難解になるのです。高すぎる柔軟性はユーザを閉口させ、コードの維持をより難しくするのです。

15.5.2 チュートリアル

`optparse` はとても柔軟で強力でありながら、ほとんどの場合には簡単に利用できます。この節では、`optparse` ベースのプログラムで広く使われているコードパターンについて述べます。

まず、`OptionParser` クラスを `import` しておかなければなりません。次に、プログラムの冒頭で `OptionParser` インスタンスを生成しておきます:

```
from optparse import OptionParser
...
parser = OptionParser()
```

これでオプションを定義できるようになりました。基本的な構文は以下の通りです:

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

各オプションには、`-f` や `--file` のような一つまたは複数のオプション文字列と、パーザがコマンドライン上のオプションを見つけた際に、何を準備し、何を行うべきかを `optparse` に教えるためのオプション属性 (option attribute) がいくつか入ります。

通常、各オプションには短いオプション文字列と長いオプション文字列があります。例えば:

```
parser.add_option("-f", "--file", ...)
```


オプション文字列は、(ゼロ文字の場合も含め)いくらでも短く、またいくらでも長くできます。ただしオプション文字列は少なくとも一つなければなりません。

`OptionParser.add_option()` に渡されたオプション文字列は、実際にはこの関数で定義したオプションに対するラベルになります。簡単のため、以後ではコマンドライン上で オプションを見つける という表現をしばしば使いますが、これは実際には `optparse` がコマンドライン上の オプション文字列 を見つけ、対応づけられているオプションを探し出す、という処理に相当します。

オプションを全て定義したら、`optparse` にコマンドラインを解析するように指示します:

```
(options, args) = parser.parse_args()
```

(お望みなら、`parse_args()` に自作の引数リストを渡してもかまいません。とはいえ、実際にはそうした必要はほとんどないでしょう: `optionparser` はデフォルトで `sys.argv[1:]` を使うからです。)

`parse_args()` は二つの値を返します:

- 全てのオプションに対する値の入ったオブジェクト `options` — 例えば、`--file` が単一の文字列引数をとる場合、`options.file` はユーザが指定したファイル名になります。オプションを指定しなかった場合には `None` になります
- オプションの解析後に残った固定引数からなるリスト `args`

このチュートリアル節では、最も重要な四つのオプション属性: `action`, `type`, `dest` (destination), `help` についてしか触れません。このうち最も重要なのは `action` です。

オプション・アクションを理解する

アクション (action) は `optparse` がコマンドライン上にあるオプションを見つけたときに何をすべきかを指示します。`optparse` には押し着せのアクションのセットがハードコードされています。新たなアクションの追加は上級者向けの話題であり、`optparse` の拡張で触れます。ほとんどのアクションは、値を何らかの変数に記憶するよう `optparse` に指示します — 例えば、文字列をコマンドラインから取り出して、`options` の属性の中に入れる、といった具合にです。

オプション・アクションを指定しない場合、`optparse` のデフォルトの動作は `store` になります。

store アクション

もっとも良く使われるアクションは `store` です。このアクションは次の引数 (あるいは現在の引数の残りの部分) を取り出し、正しい型の値か確かめ、指定した保存先に保存するよう `optparse` に指示します。

例えば:

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

例えば、以下のように指定しておき、偽のコマンドラインを作成して `optparse` に解析させてみましょう:

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

オプション文字列 `-f` を見つけると、`optparse` は次の引数である `foo.txt` を消費し、その値を `options.filename` に保存します。従って、この `parse_args()` 呼び出し後には `options.filename` は `"foo.txt"` になっています。

オプションの型として、`optparse` は他にも `int` や `float` をサポートしています:

```
parser.add_option("-n", type="int", dest="num")
```

このオプションには長い形式のオプション文字列がないため、設定に問題がないということに注意してください。また、デフォルトのアクションは `store` なので、ここでは `action` を明示的に指定していません。

架空のコマンドラインをもう一つ解析してみましょう。今度は、オプション引数をオプションの右側にぴったりくっつけて一緒にくたにします: `-n42` (一つの引数のみ) は `-n 42` (二つの引数からなる) と等価になるので

```
(options, args) = parser.parse_args(["-n42"])
print options.num
```

は `42` を出力します。

型を指定しない場合、`optparse` は引数を `string` であると仮定します。デフォルトのアクションが `store` であることも併せて考えると、最初の例はもっと短くなります:

```
parser.add_option("-f", "--file", dest="filename")
```

保存先 (destination) を指定しない場合、`optparse` はデフォルト値としてオプション文字列から気のきいた名前を設定します: 最初に指定した長い形式のオプション文字列が `--foo-bar` であれば、デフォルトの保存先は `foo_bar` になります。長い形式のオプション文字列がなければ、`optparse` は最初に指定した短い形式のオプション文字列を探します: 例えば、`-f` に対する保存先は `f` になります。

`optparse` にはビルトインの `long` と `complex` 型も含まれています。型の追加については `optparse` の拡張で触れています。

ブール値 (フラグ) オプションの処理

フラグオプション—特定のオプションに対して真または偽の値の値を設定するオプション—はよく使われます。`optparse` では、二つのアクション、`store_true` および `store_false` をサポートしています。例えば、`verbose` というフラグを `-v` で有効にして、`-q` で無効にしたいとします:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

ここでは二つのオプションに同じ保存先を指定していますが、全く問題ありません (下記のように、デフォルト値の設定を少し注意深く行わなければならないだけです)

`-v` をコマンドライン上に見つけると、`optparse` は `options.verbose` を `True` に設定します。`-q` を見つければ、`options.verbose` は `False` にセットされます。

その他のアクション

この他にも、`optparse` は以下のようなアクションをサポートしています：

"store_const" 定数値を保存します

"append" オプションの引数を指定のリストに追加します

"count" 指定のカウンタを 1 増やします

"callback" 指定の関数を呼び出します

これらのアクションについては、[リファレンスガイド](#) 節の「リファレンスガイド」および [オプション処理コールバック](#) 節で触れます。

デフォルト値

上記の例は全て、何らかのコマンドラインオプションが見つかった時に何らかの変数 (保存先: destination) に値を設定していました。では、該当するオプションが見つからなかった場合には何が起きるのでしょうか？デフォルトは全く与えていないため、これらの値は全て `None` になります。たいていはこれで十分ですが、もっときちんと制御したい場合もあります。`optparse` では各保存先に対してデフォルト値を指定し、コマンドラインの解析前にデフォルト値が設定されるようにできます。

まず、`verbose/quiet` の例について考えてみましょう。`optparse` に対して、`-q` が無い限り `verbose` を `True` に設定させたいなら、以下のようにします：

```
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

デフォルトの値は特定のオプションではなく 保存先 に対して適用されます。また、これら二つのオプションはたまたま同じ保存先を持っているにすぎないため、上のコードは下のコードと全く等価になります：

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

下のような場合を考えてみましょう：

```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

やはり `verbose` のデフォルト値は `True` になります；特定の目的変数に対するデフォルト値として有効なのは、最後に指定した値だからです。

デフォルト値をすっきりと指定するには、`OptionParser` の `set_defaults()` メソッドを使います。このメソッドは `parse_args()` を呼び出す前ならいつでも使えます：

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

前の例と同様、あるオプションの値の保存先に対するデフォルトの値は最後に指定した値になります。コードを読みやすくするため、デフォルト値を設定するときには両方のやり方を混ぜるのではなく、片方だけを使うようにしましょう。

ヘルプの生成

optparse にはヘルプと使い方の説明 (usage text) を生成する機能があり、ユーザに優しいコマンドラインインタフェースを作成する上で役立ちます。やらなければならないのは、各オプションに対する *help* の値と、必要ならプログラム全体の使用法を説明する短いメッセージを与えることです。ユーザフレンドリな (ドキュメント付きの) オプションを追加した *OptionParser* を以下に示します:

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose", default=True,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output to FILE")
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, intermediate, "
                        "or expert [default: %default]")
```

optparse がコマンドライン上で *-h* や *--help* を見つけた場合や、*parser.print_help()* を呼び出した場合、この *OptionParser* は以下のようなメッセージを標準出力に出力します:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose          make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]
```

(*help* オプションでヘルプを出力した場合、*optparse* は出力後にプログラムを終了します。)

optparse ができるだけうまくメッセージを生成するよう手助けするには、他にもまだまだやるべきことがあります:

- スクリプト自体の利用法を表すメッセージを定義します:

```
usage = "usage: %prog [options] arg1 arg2"
```

`optparse` は `%prog` を現在のプログラム名、すなわち `os.path.basename(sys.argv[0])` と置き換えます。この文字列は詳細なオプションヘルプの前に展開され出力されます。

`usage` の文字列を指定しない場合、`optparse` は型どおりとはいえ気の利いたデフォルト値、`"Usage: %prog [options]"` を使います。固定引数をとらないスクリプトの場合はこれで十分でしょう。

- 全てのオプションにヘルプ文字列を定義します。行の折り返しは気にしなくてかまいません — `optparse` は行の折り返しに気を配り、見栄えのよいヘルプ出力を生成します。
- オプションが値をとるということは自動的に生成されるヘルプメッセージの中で分かります。例えば、`"mode" option` の場合にはこのようになります:

```
-m MODE, --mode=MODE
```

ここで `"MODE"` はメタ変数 (meta-variable) と呼ばれます: メタ変数は、ユーザが `-m/--mode` に対して指定するはずの引数を表します。デフォルトでは、`optparse` は保存先の変数名を大文字だけにしたものをメタ変数に使います。これは時として期待通りの結果になりません — 例えば、上の例の `--filename` オプションでは明示的に `metavar="FILE"` を設定しており、その結果自動生成されたオプション説明テキストは:

```
-f FILE, --filename=FILE
```

この機能の重要さは、単に表示スペースを節約するといった理由にとどまりません: 上の例では、手作業で書いたヘルプテキストの中でメタ変数として `FILE` を使っています。その結果、ユーザに対してやや堅苦しい表現の書法 `-f FILE` と、より平易に意味付けを説明した `"write output to FILE"` との間に対応があるというヒントを与えています。これは、エンドユーザにとってより明解で便利なヘルプテキストを作成する単純でありながら効果的な手法なのです。

バージョン 2.4 で追加: デフォルト値を持つオプションはヘルプ文字列に `%default` を含むことができます — `optparse` はそれをオプションのデフォルト値に `str()` を適用したもので置き換えます。オプションがデフォルト値を持たない (もしくはデフォルト値が `None` である) 場合、`%default` は `none` に展開されます。

オプションをグループ化する

たくさんのオプションを扱う場合、オプションをグループ分けするとヘルプ出力が見やすくなります。`OptionParser` は、複数のオプションをまとめたオプショングループを複数持つことができます。

オプションのグループは、`OptionGroup` を使って作成します:

```
class optparse.OptionGroup (parser, title, description=None)
```

ここでは:

- `parser` は、このグループが属する `OptionParser` のインスタンスです
- `title` はグループのタイトルです

- *description* はオプションで、グループの長い説明です

`OptionGroup` は (`OptionParser` のように) `OptionContainer` を継承していて、オプションをグループに追加するために `add_option()` メソッドを利用できます。

全てのオプションを定義したら、`OptionParser` の `add_option_group()` メソッドを使ってグループを定義済みのパーサーに追加します。

前のセクションで定義したパーサーに、続けて `OptionGroup` を追加します:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)
```

この結果のヘルプ出力は次のようになります:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose          make lots of noise [default]
  -q, --quiet            be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]

Dangerous Options:
  Caution: use these options at your own risk.  It is believed that some
  of them bite.

  -g                    Group option.
```

さらにサンプルを拡張して、複数のグループを使うようにしてみます:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true",
                help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
                help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every action done")
parser.add_option_group(group)
```

出力結果は次のようになります:

```
Usage: <yourscript> [options] arg1 arg2
```

Options:

```
-h, --help            show this help message and exit
-v, --verbose          make lots of noise [default]
-q, --quiet            be vewwy quiet (I'm hunting wabbits)
-f FILE, --filename=FILE
                        write output to FILE
-m MODE, --mode=MODE  interaction mode: novice, intermediate, or expert
                        [default: intermediate]
```

Dangerous Options:

Caution: use these options at your own risk. It is believed that some of them bite.

```
-g                    Group option.
```

Debug Options:

```
-d, --debug          Print debug information
-s, --sql            Print all SQL statements executed
-e                  Print every action done
```

もう1つの、特にオプショングループをプログラムから操作するときに利用できるメソッドがあります:

`OptionParser.get_option_group(opt_str)`

短いオプション文字列もしくは長いオプション文字列 *opt_str* (例. `'-o'`、`'--option'`) が属する *OptionGroup* を返します。そのような *OptionGroup* が無い場合は、`None` を返します。

バージョン番号の出力

optparse では、使用法メッセージと同様にプログラムのバージョン文字列を出力できます。*OptionParser* の `version` 引数に文字列を渡します:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

`%prog` は `usage` と同じような展開を受けます。その他にも `version` には何でも好きな内容を入れられます。`version` を指定した場合、*optparse* は自動的に `--version` オプションをパーザに渡します。コマンドライン中に `--version` が見つかると、*optparse* は `version` 文字列を展開して (`%prog` を置き換えて) 標準出力に出力し、プログラムを終了します。

例えば、`/usr/bin/foo` という名前のスクリプトなら:

```
$ /usr/bin/foo --version
foo 1.0
```

以下の2つのメソッドを、`version` 文字列を表示するために利用できます:

`OptionParser.print_version(file=None)`

現在のプログラムのバージョン (`self.version`) を *file* (デフォルト: `stdout`) へ表示します。

`print_usage()` と同じく、`self.version` の中の全ての `%prog` が現在のプログラム名に置き換えられます。`self.version` が空文字列だったり未定義だったときは何もしません。

`OptionParser.get_version()`

`print_version()` と同じですが、バージョン文字列を表示する代わりに返します。

`optparse` のエラー処理法

`optparse` を使う場合に気を付けなければならないエラーには、大きく分けてプログラマ側のエラーとユーザ側のエラーという二つの種類があります。プログラマ側のエラーの多くは、例えば不正なオプション文字列や定義されていないオプション属性の指定、あるいはオプション属性を指定し忘れるといった、誤った `OptionParser.add_option()`、呼び出しによるものです。こうした誤りは通常通りに処理されます。すなわち、例外 (`optparse.OptionError` や `TypeError`) を送出して、プログラムをクラッシュさせます。

もっと重要なのはユーザ側のエラーの処理です。というのも、ユーザの操作エラーというものはコードの安定性に関係なく起こるからです。`optparse` は、誤ったオプション引数の指定 (整数を引数にとるオプション `-n` に対して `-n 4x` と指定してしまうなど) や、引数を指定し忘れた場合 (`-n` が何らかの引数をとるオプションであるのに、`-n` が引数の末尾に来ている場合) といった、ユーザによるエラーを自動的に検出します。また、アプリケーション側で定義されたエラー条件が起きた場合、`OptionParser.error()` を呼び出してエラーを通知できます:

```
(options, args) = parser.parse_args()
...
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

いずれの場合にも `optparse` はエラーを同じやり方で処理します。すなわち、プログラムの使用法メッセージとエラーメッセージを標準エラー出力に出力して、終了ステータス 2 でプログラムを終了させます。

上に挙げた最初の例、すなわち整数を引数にとるオプションにユーザが `4x` を指定した場合を考えてみましょう:

```
$ /usr/bin/foo -n 4x
Usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

値を全く指定しない場合には、以下ようになります:

```
$ /usr/bin/foo -n
Usage: foo [options]

foo: error: -n option requires an argument
```

`optparse` は、常にエラーを引き起こしたオプションについて説明の入ったエラーメッセージを生成するよう気を配ります; 従って、`OptionParser.error()` をアプリケーションコードから呼び出す場合にも、同じようなメッセージになるようにしてください。

`optparse` のデフォルトのエラー処理動作が気に入らないのなら、`OptionParser` をサブクラス化して、`exit()` かつ/または `error()` をオーバーライドする必要があります。

全てをつなぎ合わせる

`optparse` を使ったスクリプトは、通常以下ようになります:

```
from optparse import OptionParser
...
def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                    help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                    action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                    action="store_false", dest="verbose")
    ...
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print "reading %s..." % options.filename
    ...

if __name__ == "__main__":
    main()
```

15.5.3 リファレンスガイド

parser を作る

`optparse` を使う最初の一步は `OptionParser` インスタンスを作ることです。

class `optparse.OptionParser(...)`

`OptionParser` のコンストラクタの引数はどれも必須ではありませんが、いくつかのキーワード引数がオプションとして使えます。これらはキーワード引数として渡さなければなりません。すなわち、引数が宣言されている順番に頼ってはいけません。

`usage` (デフォルト: `"%prog [options]"`) プログラムが間違った方法で実行されるかまたはヘルプオプションを付けて実行された場合に表示される使用方法です。 `optparse` は使用方法の文字列を表示する際に `%prog` を `os.path.basename(sys.argv[0])` (または `prog` キーワード引数が指定されていればその値) に展開します。使用方法メッセージを抑制するためには特別な `optparse.SUPPRESS_USAGE` という値を指定します。

`option_list` (デフォルト: `[]`) パーザに追加する `Option` オブジェクトのリストです。 `option_list` 中のオプションは `standard_option_list` (`OptionParser` のサブクラスでセットされる可能性のあるクラス属性) の後に追加されますが、バージョンやヘルプのオブ

ションよりは前になります。このオプションの使用は推奨されません。パーザを作成した後で、`add_option()` を使って追加してください。

`option_class` (デフォルト: `optparse.Option`) `add_option()` でパーザにオプションを追加するときに使用されるクラス。

`version` (デフォルト: `None`) ユーザがバージョンオプションを与えたときに表示されるバージョン文字列です。 `version` に真の値を与えると、`optparse` は自動的に単独のオプション文字列 `--version` とともにバージョンオプションを追加します。部分文字列 `%prog` は `usage` と同様に展開されます。

`conflict_handler` (デフォルト: `"error"`) オプション文字列が衝突するようなオプションがパーザに追加されたときにどうするかを指定します。 [オプション間の衝突](#) 節を参照して下さい。

`description` (デフォルト: `None`) プログラムの概要を表す一段落のテキストです。 `optparse` はユーザがヘルプを要求したときにこの概要を現在のターミナルの幅に合わせて整形し直して表示します (`usage` の後、オプションリストの前に表示されます)。

`formatter` (デフォルト: 新しい `IndentedHelpFormatter`) ヘルプテキストを表示する際に使われる `optparse.HelpFormatter` のインスタンスです。 `optparse` はこの目的のためにすぐ使えるクラスを二つ提供しています。 `IndentedHelpFormatter` と `TitledHelpFormatter` がそれです。

`add_help_option` (デフォルト: `True`) もし真ならば、`optparse` はパーザにヘルプオプションを (オプション文字列 `-h` と `--help` とともに) 追加します。

`prog` `usage` や `version` の中の `%prog` を展開するときに `os.path.basename(sys.argv[0])` の代わりに使われる文字列です。

`epilog` (デフォルト: `None`) オプションのヘルプの後に表示されるヘルプテキスト。

パーザへのオプション追加

パーザにオプションを加えていくにはいくつか方法があります。推奨するのは [チュートリアル](#) 節で示したような `OptionParser.add_option()` を使う方法です。 `add_option()` は以下の二つのうちいずれかの方法で呼び出せます:

- `(make_option())` などが返す `Option` インスタンスを渡します
- `make_option()` に (すなわち `Option` のコンストラクタに) 固定引数とキーワード引数の組み合わせを渡して、`Option` インスタンスを生成させます

もう一つの方法は、あらかじめ作成しておいた `Option` インスタンスからなるリストを、以下のようにして `OptionParser` のコンストラクタに渡すというものです:

```
option_list = [
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose"),
```

(次のページに続く)

(前のページからの続き)

```
]
parser = OptionParser(option_list=option_list)
```

(`make_option()` は `Option` インスタンスを生成するファクトリ関数です; 現在のところ、この関数は `Option` のコンストラクタの別名にすぎません。 `optparse` の将来のバージョンでは、`Option` を複数のクラスに分割し、`make_option()` は適切なクラスを選んでインスタンスを生成するようになる予定です。従って、`Option` を直接インスタンス化しないでください。)

オプションの定義

各々の `Option` インスタンス、は `-f` や `--file` といった同義のコマンドラインオプションからなる集合を表現しています。一つの `Option` には任意の数のオプションを短い形式でも長い形式でも指定できます。ただし、少なくとも一つは指定しなければなりません。

正しい方法で `Option` インスタンスを生成するには、`OptionParser` の `add_option()` を使います。

`OptionParser.add_option(option)`

`OptionParser.add_option(*opt_str, attr=value, ...)`

短い形式のオプション文字列を一つだけ持つようなオプションを生成するには次のようにします:

```
parser.add_option("-f", attr=value, ...)
```

また、長い形式のオプション文字列を一つだけ持つようなオプションの定義は次のようになります:

```
parser.add_option("--foo", attr=value, ...)
```

キーワード引数は新しい `Option` オブジェクトの属性を定義します。オプションの属性のうちでもっとも重要なのは `action` です。この属性は、他のどの属性と関連があるか、そしてどの属性が必要かに大きく作用します。関係のないオプション属性を指定したり、必要な属性を指定し忘れたりすると、`optparse` は誤りを解説した `OptionError` 例外を送出します。

コマンドライン上にあるオプションが見つかったときの `optparse` の振舞いを決定しているのはアクション (`action`) です。 `optparse` でハードコードされている標準的なアクションには以下のようなものがあります:

"store" オプションの引数を保存します (デフォルトの動作です)

"store.const" 定数値を保存します

"store.true" 真 (`True`) を保存します

"store.false" 偽 (`False`) を保存します

"append" オプションの引数を指定のリストに追加します

"append.const" オプションの引数をリストに追加します

"count" 指定のカウンタを 1 増やします

"callback" 指定の関数を呼び出します

"help" 全てのオプションとそのドキュメントの入った使用方法メッセージを出力します

(アクションを指定しない場合、デフォルトは "store" になります。このアクションでは、`type` および `dest` オプション属性を指定できます。標準的なオプション・アクションを参照してください。)

すでにお分かりのように、ほとんどのアクションはどこかに値を保存したり、値を更新したりします。この目的のために、`optparse` は常に特別なオブジェクトを作り出し、それは通常 `options` と呼ばれます (`optparse.Values` のインスタンスになっています)。オプションの引数 (や、その他の様々な値) は、`dest` (保存先: destination) オプション属性に従って、`options` の属性として保存されます。

例えばこれを呼び出した場合

```
parser.parse_args()
```

`optparse` はまず `options` オブジェクトを生成します:

```
options = Values()
```

パーザ中で以下のようなオプションが定義されていて

```
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
```

パースしたコマンドラインに以下のいずれかが入っていた場合:

```
-ffoo
-f foo
--file=foo
--file foo
```

`optparse` はこのオプションを見つけて、以下と同等の処理を行います

```
options.filename = "foo"
```

`type` および `dest` オプション属性は `action` と同じくらい重要ですが、全てのオプションで意味をなすのは `action` だけなのです。

オプション属性

以下のオプション属性は `OptionParser.add_option()` へのキーワード引数として渡すことができます。特定のオプションに無関係なオプション属性を渡した場合、または必須のオプションを渡しそこなった場合、`optparse` は `OptionError` を送出します。

`Option.action`

(デフォルト: "store")

このオプションがコマンドラインにあった場合に `optparse` に何をさせるかを決めます。取りうるオプションについては [こちら](#) を参照してください。

`Option.type`

(デフォルト: "string")

このオプションに与えられる引数の型 (たとえば "string" や "int") です。取りうるオプションについては [こちら](#) を参照してください。

`Option.dest`

(デフォルト: オプション文字列を使う)

このオプションのアクションがある値をどこかに書いたり書き換えたりを意味する場合、これは `optparse` にその書く場所を教えます。詳しく言えば `dest` には `optparse` がコマンドラインを解析しながら組み立てる `options` オブジェクトの属性の名前を指定します。

`Option.default`

コマンドラインに指定がなかったときにこのオプションの対象に使われる値です。 `OptionParser.set_defaults()` も参照してください。

`Option.nargs`

(デフォルト: 1)

このオプションがあったときに幾つの `type` 型の引数が消費されるべきかを指定します。1 より大きい場合、`optparse` は `dest` に値のタプルを格納します。

`Option.const`

定数を格納する動作のための、その定数です。

`Option.choices`

"choice" 型オプションに対してユーザが選べる選択肢となる文字列のリストです。

`Option.callback`

アクションが "callback" であるオプションに対し、このオプションがあったときに呼ばれる呼び出し可能オブジェクトです。呼び出し時に渡される引数の詳細については、[オプション処理コールバック](#) を参照してください。

`Option.callback_args`

`Option.callback_kwargs`

`callback` に渡される標準的な 4 つのコールバック引数の後ろに追加する、位置指定引数とキーワード引数。

`Option.help`

ユーザが `help` オプション (`--help` のような) を指定したときに表示される、使用可能な全オプションのリストの中のこのオプションに関する説明文です。説明文を提供しておかなければ、オプションは説明文なしで表示されます。オプションを隠すには特殊な値 `optparse.SUPPRESS_HELP` を使います。

`Option.metavar`

(デフォルト: オプション文字列を使う)

説明文を表示する際にオプションの引数の身代わりになるものです。例は [チュートリアル](#) 節を参照してください。

標準的なオプション・アクション

様々なオプション・アクションにはどれも互いに少しづつ異なった条件と作用があります。ほとんどのアクションに関連するオプション属性がいくつかあり、値を指定して *optparse* の挙動を操作できます。いくつかのアクションには必須の属性があり、必ず値を指定しなければなりません。

- "store" [関連: *type*, *dest*, *nargs*, *choices*]

オプションの後には必ず引数が続きます。引数は *type* に従って値に変換されて *dest* に保存されます。 *nargs* > 1 の場合、複数の引数をコマンドラインから取り出します。引数は全て *type* に従って変換され、 *dest* にタプルとして保存されます。 [標準のオプション型](#) 節を参照してください。

choices を (文字列のリストかタプルで) 指定した場合、型のデフォルト値は "choice" になります。

type を指定しない場合、デフォルトの値は "string" です。

dest を指定しない場合、 *optparse* は保存先を最初の長い形式のオプション文字列から導出します (例えば、 `--foo-bar` は `foo_bar` になります)。長い形式のオプション文字列がない場合、 *optparse* は最初の短い形式のオプションから保存先の変数名を導出します (`-f` は `f` になります)。

例:

```
parser.add_option("-f")
parser.add_option("-p", type="float", nargs=3, dest="point")
```

とすると、以下のようなコマンドライン

```
-f foo.txt -p 1 -3.5 4 -fbar.txt
```

を解析した場合、 *optparse* は以下のように設定を行います

```
options.f = "foo.txt"
options.point = (1.0, -3.5, 4.0)
options.f = "bar.txt"
```

- "store_const" [関連: *const*; 関連: *dest*]

値 *const* を *dest* に保存します。

例:

```
parser.add_option("-q", "--quiet",
                  action="store_const", const=0, dest="verbose")
parser.add_option("-v", "--verbose",
                  action="store_const", const=1, dest="verbose")
parser.add_option("--noisy",
                  action="store_const", const=2, dest="verbose")
```

とします。 `--noisy` が見つかると、 *optparse* は


```
options.verbose = 2
```

- "store_true" [関連: *dest*]

"store_const" の特殊なケースで、真 (True) を *dest* に保存します。

- "store_false" [関連: *dest*]

"store_true" と似ていて、偽 (False) を保存します。

例:

```
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

- "append" [関連: *type*, *dest*, *nargs*, *choices*]

このオプションの後ろには必ず引数が続きます。引数は *dest* のリストに追加されます。 *dest* のデフォルト値を指定しなかった場合、*optparse* がこのオプションを最初にみつけた時点で空のリストを自動的に生成します。 *nargs* > 1 の場合、複数の引数をコマンドラインから取り出し、長さ *nargs* のタプルを生成して *dest* に追加します。

type および *dest* のデフォルト値は "store" アクションと同じです。

例:

```
parser.add_option("-t", "--tracks", action="append", type="int")
```

-t3 がコマンドライン上で見つかると、*optparse* は:

```
options.tracks = []
options.tracks.append(int("3"))
```

その後、--tracks=4 が見つかると以下を実行します:

```
options.tracks.append(int("4"))
```

append アクションは、オプションの現在の値の append メソッドを呼び出します。これは、どのデフォルト値も append メソッドを持っていないことを意味します。また、デフォルト値が空でない場合、オプションの解析結果は、そのデフォルトの要素の後ろにコマンドラインからの値が追加されたものになる、ということも意味します:

```
>>> parser.add_option("--files", action="append", default=['~/mypkg/defaults
↳'])
>>> opts, args = parser.parse_args(['--files', 'overrides.mypkg'])
>>> opts.files
['~/mypkg/defaults', 'overrides.mypkg']
```

- "append_const" [関連: *const*; 関連: *dest*]

"store_const" と同様ですが、`const` の値は `dest` に追加 (append) されます。"append" の場合と同じように `dest` のデフォルトは `None` ですがこのオプションを最初にみつけた時点で空のリストを自動的に生成します。

- "count" [関連: `dest`]

`dest` に保存されている整数値をインクリメントします。`dest` は (デフォルトの値を指定しない限り) 最初にインクリメントを行う前にゼロに設定されます。

例:

```
parser.add_option("-v", action="count", dest="verbosity")
```

コマンドライン上で最初に `-v` が見つかると、`optparse` は:

```
options.verbosity = 0
options.verbosity += 1
```

以後、`-v` が見つかるたびに

```
options.verbosity += 1
```

- "callback" [必須: `callback`; 関連: `type`, `nargs`, `callback_args`, `callback_kwargs`]

`callback` に指定された関数を次のように呼び出します

```
func(option, opt_str, value, parser, *args, **kwargs)
```

詳細は、[オプション処理コールバック](#) 節を参照してください。

- "help"

現在のオプションパーザ内の全てのオプションに対する完全なヘルプメッセージを出力します。ヘルプメッセージは `OptionParser` のコンストラクタに渡した `usage` 文字列と、各オプションに渡した `help` 文字列から生成します。

オプションに `help` 文字列が指定されていなくても、オプションはヘルプメッセージ中に列挙されます。オプションを完全に表示させないようにするには、特殊な値 `optparse.SUPPRESS_HELP` を使ってください。

`optparse` は全ての `OptionParser` に自動的に `help` オプションを追加するので、通常自分で生成する必要はありません。

例:

```
from optparse import OptionParser, SUPPRESS_HELP

# usually, a help option is added automatically, but that can
# be suppressed using the add_help_option argument
parser = OptionParser(add_help_option=False)
```

(次のページに続く)

(前のページからの続き)

```

parser.add_option("-h", "--help", action="help")
parser.add_option("-v", action="store_true", dest="verbose",
                  help="Be moderately verbose")
parser.add_option("--file", dest="filename",
                  help="Input file to read data from")
parser.add_option("--secret", help=SUPPRESS_HELP)

```

`optparse` がコマンドライン上に `-h` または `--help` を見つけると、以下のようなヘルプメッセージを標準出力に出力します (`sys.argv[0]` は `"foo.py"` だとします):

```

Usage: foo.py [options]

Options:
  -h, --help            Show this help message and exit
  -v                    Be moderately verbose
  --file=FILENAME       Input file to read data from

```

ヘルプメッセージの出力後、`optparse` は `sys.exit(0)` でプロセスを終了します。

- "version"

`OptionParser` に指定されているバージョン番号を標準出力に出力して終了します。バージョン番号は、実際には `OptionParser` の `print_version()` メソッドで書式化されてから出力されます。通常、`OptionParser` のコンストラクタに `version` 引数が指定されたときのみ関係のあるアクションです。`help` オプションと同様、`optparse` はこのオプションを必要に応じて自動的に追加するので、`version` オプションを作成することはほとんどないでしょう。

標準のオプション型

`optparse` には、`"string"`、`"int"`、`"long"`、`"choice"`、`"float"`、`"complex"` の 6 種類のビルトインのオプション型があります。新たなオプションの型を追加したければ、`optparse` の拡張節を参照してください。

文字列オプションの引数はチェックや変換を一切受けません: コマンドライン上のテキストは保存先にそのまま保存されます (またはコールバックに渡されます)。

整数引数 (`"int"` または `"long"` 型) は次のように解析されます:

- 数が `0x` から始まるならば、16 進数として読み取られます
- 数が `0` から始まるならば、8 進数として読み取られます
- 数が `0b` から始まるならば、2 進数として読み取られます
- それ以外の場合、数は 10 進数として読み取られます

変換は適切な底 (2, 8, 10, 16 のどれか) とともに `int()` または `long()` を呼び出すことで行なわれます。この変換が失敗した場合 `optparse` の処理も失敗に終わりますが、より役に立つエラーメッセージを出力します。

"float" および "complex" のオプション引数は直接 `float()` や `complex()` で変換されます。エラーは同様の扱いです。

"choice" オプションは "string" オプションのサブタイプです。 `choices` オプションの属性 (文字列からなるシーケンス) には、利用できるオプション引数のセットを指定します。 `optparse.check_choice()` はユーザの指定したオプション引数とマスタリストを比較して、無効な文字列が指定された場合には `OptionValueError` を送出します。

引数を解析する

`OptionParser` を作成してオプションを追加していく上で大事なポイントは、 `parse_args()` メソッドの呼び出しです:

```
(options, args) = parser.parse_args(args=None, values=None)
```

ここで入力パラメータは

`args` 処理する引数のリスト (デフォルト: `sys.argv[1:]`)

`values` オプション引数を格納する `optparse.Values` のオブジェクト (デフォルト: 新しい `Values` のインスタンス) – 既存のオブジェクトを指定した場合、オプションのデフォルトは初期化されません

であり、戻り値は

`options` `values` に渡されたものと同じオブジェクト、または `optparse` によって生成された `optparse.Values` インスタンス

`args` 全てのオプションの処理が終わった後で残った固定引数

一番普通の使い方は一切キーワード引数を使わないというものです。 `values` を指定した場合、それは繰り返される `setattr()` の呼び出し (大雑把に言うと保存される各オプション引数につき一回ずつ) で更新されていき、 `parse_args()` で返されます。

`parse_args()` が引数リストでエラーに遭遇した場合、 `OptionParser` の `error()` メソッドを適切なエンドユーザ向けのエラーメッセージとともに呼び出します。この呼び出しにより、最終的に終了ステータス 2 (伝統的な Unix におけるコマンドラインエラーの終了ステータス) でプロセスを終了させることになります。

オプション解析器への問い合わせと操作

オプションパーザのデフォルトの振る舞いは、ある程度カスタマイズすることができます。また、オプションパーザの中を調べることもできます。 `OptionParser` は幾つかのヘルパーメソッドを提供しています:

`OptionParser.disable_interspersed_args()`

オプションで無い最初の引数を見つけた時点でパースを止めるように設定します。例えば、 `-a` と `-b` が両方とも引数を取らないシンプルなオプションだったとすると、 `optparse` は通常次の構文を受け付け:

```
prog -a arg1 -b arg2
```

それを次と同じように扱います

```
prog -a -b arg1 arg2
```

この機能が無効にしたいときは、`disable_interspersed_args()` メソッドを呼び出してください。古典的な Unix システムのように、最初のオプションでない引数を見つけたときにオプションの解析を止めるようになります。

別のコマンドを実行するコマンドをプロセッサを作成する際、別のコマンドのオプションと自身のオプションが混ざるのを防ぐために利用することができます。例えば、各コマンドがそれぞれ異なるオプションのセットを持つ場合などに有効です。

`OptionParser.enable_interspersed_args()`

オプションで無い最初の引数を見つけてもパースを止めないように設定します。オプションとコマンド引数の順序が混ざっても良いようになります。これはデフォルトの動作です。

`OptionParser.get_option(opt_str)`

オプション文字列 `opt_str` に対する `Option` インスタンスを返します。該当するオプションがなければ `None` を返します。

`OptionParser.has_option(opt_str)`

`OptionParser` に `(-q` や `--verbose` のような) オプション `opt_str` がある場合、真を返します。

`OptionParser.remove_option(opt_str)`

`OptionParser` に `opt_str` に対応するオプションがある場合、そのオプションを削除します。該当するオプションに他のオプション文字列が指定されていた場合、それらのオプション文字列は全て無効になります。`opt_str` がこの `OptionParser` オブジェクトのどのオプションにも属さない場合、`ValueError` を送出します。

オプション間の衝突

注意が足りないと、衝突するオプションを定義してしまうことがあります:

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

(とりわけ、`OptionParser` から標準的なオプションを備えた自前のサブクラスを定義してしまった場合にはよく起きます。)

ユーザがオプションを追加するたびに、`optparse` は既存のオプションとの衝突がないかチェックします。何らかの衝突が見付かると、現在設定されている衝突処理メカニズムを呼び出します。衝突処理メカニズムはコンストラクタ中で呼び出せます:

```
parser = OptionParser(..., conflict_handler=handler)
```

個別にも呼び出せます:

```
parser.set_conflict_handler(handler)
```

衝突時の処理をおこなうハンドラ (handler) には、以下のものが利用できます:

"error" (デフォルト) オプション間の衝突をプログラム上のエラーとみなし、`OptionConflictError` を送出します

"resolve" オプション間の衝突をインテリジェントに解決します (下記参照)

一例として、衝突をインテリジェントに解決する `OptionParser` を定義し、衝突を起こすようなオプションを追加してみましょう:

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

この時点で、`optparse` はすでに追加済のオプションがオプション文字列 `-n` を使っていることを検出します。`conflict_handler` が "resolve" なので、`optparse` は既に追加済のオプションリストの方から `-n` を除去して問題を解決します。従って、`-n` の除去されたオプションは `--dry-run` だけでしか有効にできなくなります。ユーザがヘルプ文字列を要求した場合、問題解決の結果を反映したメッセージが出力されます:

```
Options:
  --dry-run      do no harm
  ...
  -n, --noisy    be noisy
```

これまでに追加したオプション文字列を跡形もなく削り去り、ユーザがそのオプションをコマンドラインから起動する手段をなくせます。この場合、`optparse` はオプションを完全に除去してしまうので、こうしたオプションはヘルプテキストやその他のどこにも表示されなくなります。例えば、現在の `OptionParser` の場合、以下の操作:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

を行った時点で、最初の `-n/--dry-run` オプションはもはやアクセスできなくなります。このため、`optparse` はオプションを消去してしまい、ヘルプテキストだけが残ります:

```
Options:
  ...
  -n, --noisy    be noisy
  --dry-run      new dry-run option
```

クリーンアップ

`OptionParser` インスタンスはいくつかの循環参照を抱えています。このことは Python のガーベジコレクタにとって問題になるわけではありませんが、使い終わった `OptionParser` に対して `destroy()` を呼び出すことでこの循環参照を意図的に断ち切るという方法を選ぶこともできます。この方法は特に長時間実行するアプリケーションで `OptionParser` から大きなオブジェクトグラフが到達可能になっているような場合に有用です。

その他のメソッド

OptionParser にはその他にも幾つかの公開されたメソッドがあります:

OptionParser.set_usage(usage)

上で説明したコンストラクタの usage キーワード引数での規則に従った使用法の文字列をセットします。None を渡すとデフォルトの使用法文字列が使われるようになり、optparse.SUPPRESS_USAGE によって使用法メッセージを抑制できます。

OptionParser.print_usage(file=None)

現在のプログラムの使用法メッセージ (self.usage) を file (デフォルト: stdout) に表示します。self.usage 内にある全ての %prog という文字列は現在のプログラム名に置換されます。self.usage が空もしくは未定義の時は何もしません。

OptionParser.get_usage()

print_usage() と同じですが、使用法メッセージを表示する代わりに文字列として返します。

OptionParser.set_defaults(dest=value, ...)

幾つかの保存先に対してデフォルト値をまとめてセットします。set_defaults() を使うのは複数のオプションにデフォルト値をセットする好ましいやり方です。複数のオプションが同じ保存先を共有することがあり得るからです。たとえば幾つかの "mode" オプションが全て同じ保存先をセットするものだったとすると、どのオプションもデフォルトをセットすることができ、しかし最後に指定したものだけが有効になります:

```
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced",
                  default="novice")      # overridden below
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice",
                  default="advanced")    # overrides above setting
```

こうした混乱を避けるために set_defaults() を使います:

```
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice")
```

15.5.4 オプション処理コールバック

optparse の組み込みのアクションや型が望みにかなったものでない場合、二つの選択肢があります: 一つは optparse の拡張、もう一つは callback オプションの定義です。optparse の拡張は汎用性に富んでいますが、単純なケースに対していささか大げさでもあります。大体は簡単なコールバックで事足りるでしょう。

callback オプションの定義は二つのステップからなります:

- "callback" アクションを使ってオプション自体を定義する

- コールバックを書く。コールバックは少なくとも後で説明する 4 つの引数をとる関数 (またはメソッド) でなければなりません

callback オプションの定義

callback オプションを最も簡単に定義するには、`OptionParser.add_option()` メソッドを使います。`action` の他に指定しなければならない属性は `callback` すなわちコールバックする関数自体です:

```
parser.add_option("-c", action="callback", callback=my_callback)
```

`callback` は関数 (または呼び出し可能オブジェクト) なので、callback オプションを定義する時にはあらかじめ `my_callback()` を定義しておかなければなりません。この単純なケースでは、`optparse` は `-c` が何らかの引数をとるかどうかが判別できず、通常は `-c` が引数を伴わないことを意味します — 知りたいことはただ単に `-c` がコマンドライン上に現れたかどうかだけです。とはいえ、場合によっては、自分のコールバック関数に任意の個数のコマンドライン引数を消費させたいこともあるでしょう。これがコールバック関数をトリッキーなものにしています; これについてはこの節の後の方で説明します。

`optparse` は常に四つの引数をコールバックに渡し、その他には `callback_args` および `callback_kwargs` で指定した追加引数しか渡しません。従って、最小のコールバック関数シグネチャは:

```
def my_callback(option, opt, value, parser):
```

コールバックの四つの引数については後で説明します。

callback オプションを定義する場合には、他にもいくつかオプション属性を指定できます:

type 他で使われているのと同じ意味です: "store" や "append" アクションの時と同じく、この属性は `optparse` に引数の一つ消費して `type` で指定した型に変換させます。 `optparse` は変換後の値をどこかに保存する代わりにコールバック関数に渡します。

nargs これも他で使われているのと同じ意味です: このオプションが指定されていて、かつ `nargs > 1` である場合、 `optparse` は `nargs` 個の引数を消費します。このとき各引数は `type` 型に変換できなければなりません。変換後の値はタプルとしてコールバックに渡されます。

callback_args その他の固定引数からなるタプルで、コールバックに渡されます

callback_kwargs その他のキーワード引数からなる辞書で、コールバックに渡されます

コールバック関数はどのように呼び出されるか

コールバックは全て以下の形式で呼び出されます:

```
func(option, opt_str, value, parser, *args, **kwargs)
```

ここでは:

`option` コールバックを呼び出している `Option` のインスタンスです

`opt_str` は、コールバック呼び出しのきっかけとなったコマンドライン上のオプション文字列です。(長い形式のオプションに対する省略形が使われている場合、`opt_str` は完全な、正式な形のオプション文字列となります — 例えば、ユーザが `--foobar` の短縮形として `--foo` をコマンドラインに入力した時には、`opt_str` は `"--foobar"` となります。)

`value` オプションの引数で、コマンドライン上に見つかったものです。`optparse` は、`type` が設定されている場合、単一の引数しかとりません。`value` の型はオプションの型として指定された型になります。このオプションに対する `type` が `None` である (引数なしの) 場合、`value` は `None` になります。`nargs > 1` であれば、`value` は適切な型をもつ値のタプルになります。

`parser` 現在のオプション解析の全てを駆動している `OptionParser` インスタンスです。この変数が有用なのは、この値を介してインスタンス属性としていくつかの興味深いデータにアクセスできるからです:

`parser.largs` 現在放置されている引数、すなわち、すでに消費されたものの、オプションでもオプション引数でもない引数からなるリストです。`parser.largs` は自由に変更でき、たとえば引数を追加したりできます (このリストは `args`、すなわち `parse_args()` の二つ目の戻り値になります)

`parser.rargs` 現在残っている引数、すなわち、`opt_str` および `value` があれば除き、それ以外の引数が残っているリストです。`parser.rargs` は自由に変更でき、例えばさらに引数を消費したりできます。

`parser.values` オプションの値がデフォルトで保存されるオブジェクト (`optparse.OptionValues` のインスタンス) です。この値を使うと、コールバック関数がオプションの値を記憶するために、他の `optparse` と同じ機構を使えるようにするため、グローバル変数や閉包 (closure) を台無しにしないので便利です。コマンドライン上にすでに現れているオプションの値にもアクセスできます。

`args` `callback_args` オプション属性で与えられた任意の固定引数からなるタプルです。

`kwargs` `callback_kwargs` オプション属性で与えられた任意のキーワード引数からなるタプルです。

コールバック中で例外を送出する

オプション自体か、あるいはその引数に問題がある場合、コールバック関数は `OptionValueError` を送出しなければなりません。`optparse` はこの例外をとらえてプログラムを終了させ、ユーザが指定しておいたエラーメッセージを標準エラー出力に出力します。エラーメッセージは明確、簡潔かつ正確で、どのオプションに誤りがあるかを示さなければなりません。さもなければ、ユーザは自分の操作のどこに問題があるかを解決するのに苦労することになります。

コールバックの例 1: ありふれたコールバック

引数をとらず、発見したオプションを単に記録するだけのコールバックオプションの例を以下に示します:

```
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True
```

(次のページに続く)

(前のページからの続き)

```
parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

もちろん、"store_true" アクションを使っても実現できます。

コールバックの例 2: オプションの順番をチェックする

もう少し面白みのある例を示します: この例では、`-b` を発見して、その後で `-a` がコマンドライン中に現れた場合にはエラーになります。

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1
...
parser.add_option("-a", action="callback", callback=check_order)
parser.add_option("-b", action="store_true", dest="b")
```

コールバックの例 3: オプションの順番をチェックする (汎用的)

このコールバック (フラグを立てるが、`-b` が既に指定されていればエラーになる) を同様の複数のオプションに対して再利用したければ、もう少し作業する必要があります: エラーメッセージとセットされるフラグを一般化しなければなりません。

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("-a", action="callback", callback=check_order, dest='a')
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback", callback=check_order, dest='c')
```

コールバックの例 4: 任意の条件をチェックする

もちろん、単に定義済みのオプションの値を調べるだけにとどまらず、コールバックには任意の条件を入れられます。例えば、満月でなければ呼び出してはならないオプションがあるとしましょう。やらなければならないことはこれだけです:

```
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is full"
                                % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("--foo",
                  action="callback", callback=check_moon, dest="foo")
```

(`is_moon_full()` の定義は読者への課題としましょう。)

コールバックの例 5: 固定引数

決まった数の引数をとるようなコールバックオプションを定義するなら、問題はやや興味深くなってきます。引数をとるようコールバックに指定するのは、`"store"` や `"append"` オプションの定義に似ています。`type` を定義していれば、そのオプションは引数を受け取ったときに該当する型に変換できなければなりません。さらに `nargs` を指定すれば、オプションは `nargs` 個の引数を受け取ります。

標準の `"store"` アクションをエミュレートする例を以下に示します:

```
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
    ...
parser.add_option("--foo",
                  action="callback", callback=store_value,
                  type="int", nargs=3, dest="foo")
```

`optparse` は 3 個の引数を受け取り、それらを整数に変換するところまで面倒をみてくれます。ユーザは単にそれを保存するだけです。(他の処理もできます; いうまでもなく、この例にはコールバックは必要ありません)

コールバックの例 6: 可変個の引数

あるオプションに可変個の引数を持たせたいと考えているなら、問題はいささか手強くなってきます。この場合、`optparse` では該当する組み込みのオプション解析機能を提供していないので、自分でコールバックを書かなければなりません。さらに、`optparse` が普段処理している、伝統的な Unix コマンドライン解析における難題を自分で解決しなければなりません。とりわけ、コールバック関数では引数が裸の `--` や `-` の場合における慣習的な処理規則:

- either `--` or `-` can be option arguments
- 裸の `--` (何らかのオプションの引数でない場合): コマンドライン処理を停止し、`--` を無視します
- 裸の `-` (何らかのオプションの引数でない場合): コマンドライン処理を停止しますが、`-` は残します (`parser.largs` に追加します)

オプションが可変個の引数をとるようにさせたいなら、いくつかの巧妙で厄介な問題に配慮しなければなりません。どういう実装をとるかは、アプリケーションでどのようなトレードオフを考慮するかによります (このため、`optparse` では可変個の引数に関する問題を直接的に取り扱わないのです)。

とはいえ、可変個の引数をもつオプションに対するスタブ (stub、仲介インタフェース) を以下に示しておきます:

```
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []
```

(次のページに続く)

(前のページからの続き)

```

def floatable(str):
    try:
        float(str)
        return True
    except ValueError:
        return False

for arg in parser.rargs:
    # stop on --foo like options
    if arg[:2] == "--" and len(arg) > 2:
        break
    # stop on -a, but not on -3 or -3.0
    if arg[:1] == "-" and len(arg) > 1 and not floatable(arg):
        break
    value.append(arg)

del parser.rargs[:len(value)]
setattr(parser.values, option.dest, value)

...
parser.add_option("-c", "--callback", dest="vararg_attr",
                  action="callback", callback=vararg_callback)

```

15.5.5 optparse の拡張

`optparse` がコマンドラインオプションをどのように解釈するかを決める二つの重要な要素はそれぞれのオプションのアクションと型なので、拡張の方向は新しいアクションと型を追加することになると思います。

新しい型の追加

新しい型を追加するためには、`optparse` の `Option` クラスのサブクラスを自身で定義する必要があります。このクラスには `optparse` における型を定義する一対の属性があります。それは `TYPES` と `TYPE_CHECKER` です。

`Option.TYPES`

`TYPES` は型名のタプルです。新しく作るサブクラスでは、タプル `TYPES` を単純に標準のものを利用して新しく定義すると良いでしょう。

`Option.TYPE_CHECKER`

`TYPE_CHECKER` は辞書で型名を型チェック関数に対応付けるものです。型チェック関数は以下のようなシグネチャを持ちます:

```
def check_mytype(option, opt, value)
```

ここで `option` は `Option` のインスタンスであり、`opt` はオプション文字列 (たとえば `-f`) で、`value` は望みの型としてチェックされ変換されるべくコマンドラインで与えられる文字列です。`check_mytype()` は想定されている型 `mytype` のオブジェクトを返さなければなりません。型

チェック関数から返される値は `OptionParser.parse_args()` で返される `OptionValues` インスタンスに収められるか、またはコールバックに `value` パラメータとして渡されます。

型チェック関数は何か問題に遭遇したら `OptionValueError` を送出しなければなりません。`OptionValueError` は文字列一つを引数に取り、それはそのまま `OptionParser` の `error()` メソッドに渡され、そこでプログラム名と文字列 `"error: "` が前置されてプロセスが終了する前に `stderr` に出力されます。

馬鹿馬鹿しい例ですが、Python スタイルの複素数を解析する `"complex"` オプション型を作ってみせることにします。(`optparse` 1.3 が複素数のサポートを組み込んでしまったため以前にも増して馬鹿らしくなりましたが、気にしないでください。)

最初に必要な `import` 文を書きます:

```
from copy import copy
from optparse import Option, OptionValueError
```

まずは型チェック関数を定義しなければなりません。これは後で (これから定義する `Option` のサブクラスの `TYPE_CHECKER` クラス属性の中で) 参照されることになります:

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

最後に `Option` のサブクラスです:

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(もしここで `Option.TYPE_CHECKER` に `copy()` を適用しなければ、`optparse` の `Option` クラスの `TYPE_CHECKER` 属性をいじってしまうことになります。Python の常として、良いマナーと常識以外にそうすることを止めるものはありません。)

これだけです! もう新しいオプション型を使うスクリプトを他の `optparse` に基づいたスクリプトとまるで同じように書くことができます。ただし、`OptionParser` に `Option` でなく `MyOption` を使うように指示しなければなりません:

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

別のやり方として、オプションリストを構築して `OptionParser` に渡すという方法もあります。`add_option()` を上でやったように使わないならば、`OptionParser` にどのクラスを使うのか教える必要はありません:

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

新しいアクションの追加

新しいアクションの追加はもう少しトリッキーです。というのも *optparse* が使っている二つのアクションの分類を理解する必要があるからです:

”store” アクション *optparse* が値を現在の OptionValues の属性に格納することになるアクションです。この種類のオプションは Option のコンストラクタに *dest* 属性を与えることが要求されます。

”typed” アクション コマンドラインから引数を受け取り、それがある型であることが期待されているアクションです。もう少しはっきり言えば、その型に変換される文字列を受け取るものです。この種類のオプションは Option のコンストラクタに *type* 属性を与えることが要求されます。

この分類には重複する部分があります。デフォルトの ”store” アクションには "store", "store_const", "append", "count" などがありますが、デフォルトの ”typed” オプションは "store", "append", "callback" の三つです。

アクションを追加する際に、以下の Option のクラス属性 (全て文字列のリストです) の中の少なくとも一つに付け加えることでそのアクションを分類する必要があります:

Option.ACTIONS

全てのアクションは ACTIONS にリストされていなければなりません。

Option.STORE_ACTIONS

”store” アクションはここにもリストされます。

Option.TYPED_ACTIONS

”typed” アクションはここにもリストされます。

Option.ALWAYS_TYPED_ACTIONS

型を取るアクション (つまりそのオプションが値を取る) はここにもリストされます。このことの唯一の効果は *optparse* が、型の指定が無くアクションが *ALWAYS_TYPED_ACTIONS* のリストにあるオプションに、デフォルト型 "string" を割り当てるということです。

実際に新しいアクションを実装するには、Option の *take_action()* メソッドをオーバーライドしてそのアクションを認識する場合分けを追加しなければなりません。

例えば、"extend" アクションというのを追加してみましょう。このアクションは標準的な "append" アクションと似ていますが、コマンドラインから一つだけ値を読み取って既存のリストに追加するのではなく、複数の値をコンマ区切りの文字列として読み取ってそれらで既存のリストを拡張します。すなわち、もし *--names* が "string" 型の "extend" オプションだとすると、次のコマンドライン

```
--names=foo,bar --names blah --names ding,dong
```

の結果は次のリストになります


```
["foo", "bar", "blah", "ding", "dong"]
```

再び Option のサブクラスを定義します:

```
class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)

    def take_action(self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(
                self, action, dest, opt, value, values, parser)
```

注意すべきは次のようなところです:

- "extend" はコマンドラインの値を予期していると同時にその値をどこかに格納しますので、*STORE_ACTIONS* と *TYPED_ACTIONS* の両方に入ります。
- *optparse* が "extend" アクションに "string" 型を割り当てるように "extend" アクションは *ALWAYS.TYPED_ACTIONS* にも入れてあります。
- *MyOption.take_action()* にはこの新しいアクション一つの扱いだけを実装しており、他の標準的な *optparse* のアクションについては *Option.take_action()* に制御を戻すようにしてあります。
- *values* は *optparse.parser.Values* クラスのインスタンスであり、非常に有用な *ensure_value()* メソッドを提供しています。 *ensure_value()* は本質的に安全弁付きの *getattr()* です。次のように呼び出します

```
values.ensure_value(attr, value)
```

values に *attr* 属性が無いが *None* だった場合に、*ensure_value()* は最初に *value* をセットし、それから *value* を返します。この振る舞いは "extend", "append", "count" のように、データを変数に集積し、またその変数がある型 (最初の二つはリスト、最後のは整数) であると期待されるアクションを作るのにとても使い易いものです。 *ensure_value()* を使えば、作ったアクションを使うスクリプトはオプションに保存先にデフォルト値をセットすることに煩わされずに済みます。デフォルトを *None* にしておけば *ensure_value()* がそれが必要になったときに適当な値を返してくれます。

15.6 getopt — C 言語スタイルのコマンドラインオプションパーサ

ソースコード: [Lib/getopt.py](#)

注釈: `getopt` モジュールは、C 言語の `getopt()` 関数に慣れ親しんだ人ためにデザインされた API を持つコマンドラインオプションのパーサです。 `getopt()` 関数に慣れ親しんでない人や、コードを少なくしてよりよいヘルプメッセージを表示させたい場合は、 `argparse` モジュールの使用を検討してください。

このモジュールは `sys.argv` に入っているコマンドラインオプションの構文解析を支援します。 `'-'` や `'--'` の特別扱いも含めて、Unix の `getopt()` と同じ記法をサポートしています。3 番目の引数 (省略可能) を設定することで、GNU のソフトウェアでサポートされているような長形式のオプションも利用できます。

このモジュールは 2 つの関数と 1 つの例外を提供しています:

`getopt.getopt(args, options[, long_options])`

コマンドラインオプションとパラメータのリストを構文解析します。 `args` は構文解析の対象になる引数のリストです。これは先頭のプログラム名を除いたもので、通常 `sys.argv[1:]` で与えられます。 `options` はスクリプトで認識させたいオプション文字と、引数が必要な場合にはコロン (':') をつけます。つまり Unix の `getopt()` と同じフォーマットになります。

注釈: GNU の `getopt()` とは違って、オプションでない引数の後は全てオプションではないと判断されます。これは GNU でない、Unix システムの挙動に近いものです。

`long_options` は長形式のオプションの名前を示す文字列のリストです。名前には、先頭の `'--'` は含めません。引数が必要な場合には名前の最後に等号 ('=') を入れます。オプション引数はサポートしていません。長形式のオプションだけを受けつけるためには、 `options` は空文字列である必要があります。長形式のオプションは、該当するオプションを一意に決定できる長さまで入力されていれば認識されます。たとえば、 `long_options` が `['foo', 'frob']` の場合、 `--fo` は `--foo` にマッチしますが、 `--f` では一意に決定できないので、 `GetoptError` が送出されます。

返り値は 2 つの要素から成っています: 最初は `(option, value)` のタプルのリスト、2 つ目はオプションリストを取り除いたあとに残ったプログラムの引数リストです (`args` の末尾部分のスライスになります)。それぞれの引数と値のタプルの最初の要素は、短形式の時はハイフン 1 つで始まる文字列 (例: `'-x'`)、長形式の時はハイフン 2 つで始まる文字列 (例: `'--long-option'`) となり、引数が 2 番目の要素になります。引数をとらない場合には空文字列が入ります。オプションは見つかった順に並んでいて、複数回同じオプションを指定できます。長形式と短形式のオプションは混在できます。

`getopt.gnu_getopt(args, options[, long_options])`

この関数はデフォルトで GNU スタイルのスキャンモードを使う以外は `getopt()` と同じように動作します。つまり、オプションとオプションでない引数とを混在させることができます。 `getopt()` 関数はオプションでない引数を見つけると解析を停止します。

オプション文字列の最初の文字を '+' にするか、環境変数 `POSIIXLY_CORRECT` を設定することで、オプションでない引数を見つけると解析を停止するように振舞いを変えることができます。

バージョン 2.3 で追加。

exception `getopt.GetoptError`

引数リストの中に認識できないオプションがあった場合か、引数が必要なオプションに引数が与えられ

なかった場合に発生します。例外の引数は原因を示す文字列です。長形式のオプションについては、不要な引数が与えられた場合にもこの例外が発生します。msg 属性と opt 属性で、エラーメッセージと関連するオプションを取得できます。特に関係するオプションが無い場合には opt は空文字列となります。

バージョン 1.6 で変更: `error` への別名として `GetoptError` が導入されました。

exception `getopt.error`

`GetoptError` へのエイリアスです。後方互換性のために残されています。

Unix スタイルのオプションを使った例です:

```
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

長形式のオプションを使っても同様です:

```
>>> s = '--condition=foo --testing --output-file abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', [
...     'condition=', 'output-file=', 'testing'])
>>> optlist
[('--condition', 'foo'), ('--testing', ''), ('--output-file', 'abc.def'), ('-x', 'a1')]
>>> args
['a2']
```

スクリプト中での典型的な使い方は以下ようになります:

```
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
    except getopt.GetoptError as err:
        # print help information and exit:
        print str(err) # will print something like "option -a not recognized"
        usage()
        sys.exit(2)
    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
```

(次のページに続く)

(前のページからの続き)

```

        verbose = True
    elif o in ("-h", "--help"):
        usage()
        sys.exit()
    elif o in ("-o", "--output"):
        output = a
    else:
        assert False, "unhandled option"

# ...

if __name__ == "__main__":
    main()

```

`argparse` モジュールを使えば、より良いヘルプメッセージとエラーメッセージを持った同じコマンドラインインタフェースをより少ないコードで実現できます:

```

import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    args = parser.parse_args()
    # ... do something with args.output ...
    # ... do something with args.verbose ..

```

参考:

`argparse` モジュール 別のコマンドラインオプションと引数の解析ライブラリ。

15.7 logging — Python 用ロギング機能

Important

このページには、リファレンス情報だけが含まれています。チュートリアルは、以下のページを参照してください

- 基本チュートリアル
- 上級チュートリアル
- ロギングクックブック

Source code: `Lib/logging/__init__.py`

バージョン 2.3 で追加.

このモジュールは、アプリケーションやライブラリのための柔軟なエラーログ記録 (logging) システムを実装するための関数やクラスを定義しています。

標準ライブラリモジュールとしてログ記録 API が提供される利点は、すべての Python モジュールがログ記録に参加できることであり、これによってあなたが書くアプリケーションのログにサードパーティーのモジュールが出力するメッセージを含ませることができます。

このモジュールは、多くの機能性と柔軟性を提供します。ロギングに慣れていないなら、つかむのに一番いいのはチュートリアルを読むことです (右のリンクを参照してください)。

モジュールで定義されている基本的なクラスと関数を、以下に列挙します。

- ログガーは、アプリケーションコードが直接使うインタフェースを公開します。
- ハンドラは、(ログガーによって生成された) ログ記録を適切な送信先に送ります。
- フィルタは、どのログ記録を出力するかを決定する、きめ細かい機能を提供します。
- フォーマッタは、ログ記録が最終的に出力されるレイアウトを指定します。

15.7.1 ログガーオブジェクト

ログガーには以下のような属性とメソッドがあります。ログガーを直接インスタンス化することはできず、常にモジュール関数 `logging.getLogger(name)` を介してインスタンス化することに注意してください。同じ `name` で `getLogger()` を複数回呼び出すと、常に同じログガー・オブジェクトへの参照が返されます。

`name` は `foo.bar.baz` のようにピリオドで分割された (ただし単なるブレーンな `foo` もありえます) 潜在的に階層的な値です。階層リスト中でより下位のログガーは、上位のログガーの子です。例えば、`foo` という名前を持つログガーがあるとき、`foo.bar`, `foo.bar.baz`, `foo.bam` という名前を持つログガーはすべて `foo` の子孫です。ログガー名の階層は Python パッケージ階層と類似していて、推奨される構築方法 `logging.getLogger(__name__)` を使用してログガーをモジュール単位で構成すれば、Python パッケージ階層と同一になります。これは、モジュールの中では `__name__` が Python パッケージ名前空間におけるモジュール名だからです。

```
class logging.Logger
```

`Logger.propagate`

これが真と評価されると、記録されたイベントがこのログガーに向けられるとそれはこのログガーに所属する全てのハンドラに渡されるとともに、上位 (祖先) ログガーのハンドラにも渡されます。メッセージは、祖先ログガーのハンドラに直接渡されます - 今問題にしている祖先ログガーのレベルもフィルタも、どちらも考慮されません。

この値の評価結果が偽になる場合、ロギングメッセージは祖先ログガーのハンドラに渡されません。

コンストラクタはこの属性を `True` に設定します。

注釈: 一つのログガー と 一つ以上のその祖先にハンドラを接続すると、同一レコードが何度も発行されます。一般的に言って、ハンドラを複数のログガーに接続しようとするべきではありません - ログガーの

階層においての最上位のロガーがそれを接続するのに相応しく、そうすれば `propagate` 設定を `True` にすることで、全ての子孫ロガーが記録する全てのイベントをそのハンドラが参照することが出来ます。一般的なシナリオは、ハンドラはルートロガーに対してのみ接続し、残りのものについての `propagate` 設定を注意深く行います。

`Logger.setLevel(level)`

Sets the threshold for this logger to *level*. Logging messages which are less severe than *level* will be ignored. When a logger is created, the level is set to `NOTSET` (which causes all messages to be processed when the logger is the root logger, or delegation to the parent when the logger is a non-root logger). Note that the root logger is created with level `WARNING`.

「親ロガーに委譲」という用語の意味は、もしロガーのレベルが `NOTSET` ならば、祖先ロガーの系列の中を `NOTSET` 以外のレベルの祖先を見つけるかルートに到達するまで辿っていく、ということです。

もし `NOTSET` 以外のレベルの祖先が見つかったなら、その祖先のレベルが探索を開始したロガーの実効レベルとして扱われ、ログイベントがどのように処理されるかを決めるのに使われます。

ルートに到達した場合、ルートのレベルが `NOTSET` ならばすべてのメッセージは処理されます。そうでなければルートのレベルが実効レベルとして使われます。

レベルの一覧については [ロギングレベル](#) を参照してください。

`Logger.isEnabledFor(lvl)`

深刻度が *lvl* のメッセージが、このロガーで処理されることになっているかどうかを示します。このメソッドはまず、`logging.disable(lvl)` で設定されるモジュールレベルの深刻度レベルを調べ、次にロガーの実効レベルを `getEffectiveLevel()` で調べます。

`Logger.getEffectiveLevel()`

このロガーの実効レベルを示します。 `NOTSET` 以外の値が `setLevel()` で設定されていた場合、その値が返されます。そうでない場合、 `NOTSET` 以外の値が見つかるまでロガーの階層をルートロガーの方向に追跡します。見つかった場合、その値が返されます。返される値は整数で、典型的には `logging.DEBUG`, `logging.INFO` 等のうち一つです。

`Logger.getChild(suffix)`

このロガーの子であるロガーを、接頭辞によって決定し、返します。従って、`logging.getLogger('abc').getChild('def.ghi')` は、`logging.getLogger('abc.def.ghi')` によって返されるのと同じロガーを返すことになります。これは簡便なメソッドで、親ロガーがリテラルでなく `__name__` などを使って名付けられているときに便利です。

バージョン 2.7 で追加。

`Logger.debug(msg, *args, **kwargs)`

レベル `DEBUG` のメッセージをこのロガーで記録します。 *msg* はメッセージの書式化文字列で、 *args* は *msg* に文字列書式化演算子を使って取り込むための引数です。(これは、書式化文字列の中でキーワードを使い、引数として単一の辞書を渡すことができる、ということを意味します。)

キーワード引数 *kwargs* からは 2 つのキーワードが調べられます。一つ目は `exc_info` で、この値の評価値が `false` でない場合、例外情報をログメッセージに追加します。(`sys.exc_info()` の返す形式

の) 例外情報を表すタプルが与えられていれば、それをメッセージに使います。それ以外の場合には、`sys.exc_info()` を呼び出して例外情報を取得します。

2 番目のキーワード引数は `extra` で、当該ログイベント用に作られる `LogRecord` の `__dict__` にユーザー定義属性を加えるのに使われる辞書を渡すために用いられます。これらの属性は好きなように使えます。たとえば、ログメッセージの一部にすることもできます。以下の例を見てください:

```
FORMAT = '%(asctime)-15s %(clientip)s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logger = logging.getLogger('tcpserver')
logger.warning('Protocol problem: %s', 'connection reset', extra=d)
```

出力はこのようになります

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs Protocol problem: connection_
↪reset
```

`extra` で渡される辞書のキーはロギングシステムで使われているものと衝突しないようにしなければなりません。(どのキーがロギングシステムで使われているかについての詳細は *Formatter* のドキュメントを参照してください。)

これらの属性をログメッセージに使うことにしたなら、少し注意が必要です。上の例では、`'clientip'` と `'user'` が `LogRecord` の属性辞書に含まれていることを期待した書式化文字列で *Formatter* がセトアップされています。もしこれらが欠けていると、書式化例外が発生してしまうためメッセージはログに残りません。したがってこの場合、常にこれらのキーを含む `extra` 辞書を渡す必要があります。

このようなことは煩わしいかもしれませんが、この機能は限定された場面で使われるように意図しているものなのです。たとえば同じコードがいくつかのコンテキストで実行されるマルチスレッドのサーバで、興味のある条件が現れるのがそのコンテキストに依存している(上の例で言えば、リモートのクライアント IP アドレスや認証されたユーザ名など)、というような場合です。そういった場面では、それ用の *Formatter* が特定の *Handler* と共に使われるというのはよくあることです。

`Logger.info(msg, *args, **kwargs)`

レベル INFO のメッセージをこのロガーで記録します。引数は *debug()* と同じように解釈されます。

`Logger.warning(msg, *args, **kwargs)`

レベル WARNING のメッセージをこのロガーで記録します。引数は *debug()* と同じように解釈されます。

`Logger.error(msg, *args, **kwargs)`

レベル ERROR のメッセージをこのロガーで記録します。引数は *debug()* と同じように解釈されます。

`Logger.critical(msg, *args, **kwargs)`

レベル CRITICAL のメッセージをこのロガーで記録します。引数は *debug()* と同じように解釈されます。

`Logger.log(lvl, msg, *args, **kwargs)`

整数で表したレベル *lvl* のメッセージをこのロガーで記録します。その他の引数は *debug()* と同じように解釈されます。

`Logger.exception(msg, *args, **kwargs)`

レベル `ERROR` のメッセージをこのロガーで記録します。引数は、渡された *exc_info* の詳細が展開されないことを除いて、*debug()* と同じように解釈されます。例外情報が常にログメッセージに追加されます。このメソッドは例外ハンドラからのみ呼び出されるべきです。

`Logger.addFilter(filter)`

指定されたフィルタ *filter* をこのロガーに追加します。

`Logger.removeFilter(filter)`

指定されたフィルタ *filter* をこのロガーから取り除きます。

`Logger.filter(record)`

レコードに対してこのロガーのフィルタを適用し、レコードが処理されるべき場合に真を返します。フィルタのいずれかの値が偽を返すまで、それらは順番に試されていきます。いずれも偽を返さなければ、レコードは処理される (ハンドラに渡される) ことになります。ひとつでも偽を返せば、発生したレコードはもはや処理されることはありません。

`Logger.addHandler(hdlr)`

指定されたハンドラ *hdlr* をこのロガーに追加します。

`Logger.removeHandler(hdlr)`

指定されたハンドラ *hdlr* をこのロガーから取り除きます。

`Logger.findCaller()`

呼び出し元のソースファイル名と行番号を調べます。ファイル名と行番号、関数名を 3 要素のタプルで返します。

バージョン 2.4 で変更: 関数名が追加されました。以前のバージョンでは、ファイル名と行番号の、2 要素のタプルを返していました。

`Logger.handle(record)`

レコードを、このロガーおよびその上位ロガー (ただし *propagate* の値が `false` になったところまで) に関連付けられているすべてのハンドラに渡して処理します。このメソッドは、ローカルで生成されたレコードだけでなく、ソケットから受信した unpickle されたレコードに対しても同様に用いられます。*filter()* によって、ロガーレベルでのフィルタが適用されます。

`Logger.makeRecord(name, lvl, fn, lno, msg, args, exc_info, func=None, extra=None)`

このメソッドは、特殊な *LogRecord* インスタンスを生成するためにサブクラスでオーバーライドできるファクトリメソッドです。

バージョン 2.5 で変更: *func* と *extra* が追加されました。

15.7.2 ログingleレベル

ログレベルの数値は以下の表のように与えられています。これらは基本的に自分でレベルを定義したい人のためのもので、定義するレベルを既存のレベルの間に位置づけるためには具体的な値が必要になります。もし数値が他のレベルと同じだったら、既存の値は上書きされその名前は失われます。

レベル	Numeric value
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

15.7.3 ハンドラオブジェクト

ハンドラ (Handler) は以下の属性とメソッドを持ちます。Handler は直接インスタンス化されることはありません; このクラスはより便利なサブクラスの基底クラスとして働きます。しかしながら、サブクラスにおける `__init__()` メソッドでは、`Handler.__init__()` を呼び出す必要があります。

`Handler.__init__(level=NOTSET)`

レベルを設定して、Handler インスタンスを初期化します。空のリストを使ってフィルタを設定し、I/O 機構へのアクセスを直列化するために (`createLock()` を使って) ロックを生成します。

`Handler.createLock()`

スレッドセーフでない背後の I/O 機能に対するアクセスを直列化するために用いられるスレッドロック (thread lock) を初期化します。

`Handler.acquire()`

`createLock()` で生成されたスレッドロックを獲得します。

`Handler.release()`

`acquire()` で獲得したスレッドロックを解放します。

`Handler.setLevel(level)`

このハンドラに対する閾値を `level` に設定します。`level` よりも深刻でないログメッセージは無視されます。ハンドラが生成された際、レベルは NOTSET (すべてのメッセージが処理される) に設定されます。

レベルの一覧については [ログingleレベル](#) を参照してください。

`Handler.setFormatter(fmt)`

このハンドラのフォーマッタを `fmt` に設定します。

`Handler.addFilter(filter)`

指定されたフィルタ `filter` をこのハンドラに追加します。

`Handler.removeFilter(filter)`

指定されたフィルタ *filter* をこのハンドラから除去します。

`Handler.filter(record)`

レコードに対してこのハンドラのフィルタを適用し、レコードが処理されるべき場合に真を返します。フィルタのいずれかの値が偽を返すまで、それらは順番に試されていきます。いずれも偽を返さなければ、レコードは発行されることになります。ひとつでも偽を返せば、ハンドラはレコードを発行しません。

`Handler.flush()`

すべてのログ出力がフラッシュされるようにします。このクラスのバージョンではなにも行わず、サブクラスで実装するためのものです。

`Handler.close()`

ハンドラで使われているすべてのリソースの後始末を行います。このバージョンでは何も出力せず、`shutdown()` が呼ばれたときに閉じられたハンドラを内部リストから削除します。サブクラスではオーバーライドされた `close()` メソッドからこのメソッドが必ず呼ばれるようにしてください。

`Handler.handle(record)`

ハンドラに追加されたフィルタの条件に応じて、指定されたログレコードを出力します。このメソッドは I/O スレッドロックの獲得/解放を伴う実際のログ出力をラップします。

`Handler.handleError(record)`

このメソッドは `emit()` の呼び出し中に例外に遭遇した際にハンドラから呼び出されます。モジュールレベル属性 `raiseExceptions` が `False` の場合、例外は暗黙のまま無視されます。ほとんどの場合、これがロギングシステムの望ましい動作です - というのは、ほとんどのユーザはロギングシステム自体のエラーは気にせず、むしろアプリケーションのエラーに興味があるからです。しかしながら、望むならこのメソッドを自作のハンドラと置き換えることもできます。*record* には、例外発生時に処理されていたレコードが入ります。(`raiseExceptions` のデフォルト値は `True` です。これは開発中はその方が便利だからです)。

`Handler.format(record)`

レコードに対する書式化を行います - フォーマットが設定されていれば、それを使います。そうでない場合、モジュールにデフォルト指定されたフォーマットを使います。

`Handler.emit(record)`

指定されたログ記録レコードを実際にログ記録する際のすべての処理を行います。このメソッドはサブクラスで実装されることを意図しており、そのためこのクラスのバージョンは `NotImplementedError` を送出します。

標準として含まれているハンドラについては、`logging.handlers` を参照してください。

15.7.4 フォーマッタオブジェクト

フォーマッタ (*Formatter*) は以下の属性とメソッドを持っています。 *Formatter* は *LogRecord* を (通常は) 人間か外部のシステムで解釈できる文字列に変換する役割を担っています。基底クラスの *Formatter* では書式化文字列を指定することができます。何も指定されなかった場合、ロギングコール中のメッセージ以

外の情報だけを持つ '`% (message) s`' の値が使われます。フォーマットされた出力に情報の要素 (タイムスタンプなど) を追加したいなら、このまま読み進めてください。

`Formatter` は `LogRecord` 属性の知識を利用できるような書式化文字列を用いて初期化することができます。例えば、上で言及したデフォルト値では、ユーザによるメッセージと引数はあらかじめ書式化されて、`LogRecord` の `message` 属性に入っていることを利用しています。この書式化文字列は、Python 標準の `%` を使った変換文字列で構成されます。文字列整形に関する詳細は [文字列フォーマット操作](#) を参照してください。

`LogRecord` の便利なマッピングキーは、`LogRecord` 属性 の節で与えられます。

class `logging.Formatter` (*fmt=None, datefmt=None*)

`Formatter` クラスの新たなインスタンスを返します。インスタンスは全体としてのメッセージに対する書式化文字列と、メッセージの日付/時刻部分のための書式化文字列を伴って初期化されます。*fmt* が指定されない場合、'`% (message) s`' が使われます。*datefmt* が指定されない場合、ISO8601 日付書式が使われます。

format (*record*)

レコードの属性辞書が、文字列を書式化する演算で被演算子として使われます。書式化された結果の文字列を返します。辞書を書式化する前に、二つの準備段階を経ます。レコードの `message` 属性が `msg % args` を使って処理されます。書式化された文字列が '`(asctime)`' を含むなら、`formatTime()` が呼び出され、イベントの発生時刻を書式化します。例外情報が存在する場合、`formatException()` を使って書式化され、メッセージに追加されます。ここで注意していただきたいのは、書式化された例外情報は `exc_text` にキャッシュされるという点です。これが有用なのは例外情報がピクル化されて回線を送ることができるからですが、しかし二つ以上の `Formatter` サブクラスで例外情報の書式化をカスタマイズしている場合には注意が必要になります。この場合、フォーマッタが書式化を終えるごとにキャッシュをクリアして、次のフォーマッタがキャッシュされた値を使わずに新鮮な状態で再計算するようにしなければならないことになります。

formatTime (*record, datefmt=None*)

このメソッドは、フォーマッタが書式化された時間を利用したい際に、`format()` から呼び出されます。このメソッドは特定の要求を提供するためにフォーマッタで上書きすることができますが、基本的な振る舞いは以下ようになります: *datefmt* (文字列) が指定された場合、レコードが生成された時刻を書式化するために `time.strftime()` で使われます。そうでない場合、ISO8601 書式が使われます。結果の文字列が返されます。

この関数は、ユーザが設定できる関数を使って、生成時刻をタプルに変換します。デフォルトでは、`time.localtime()` が使われます。特定のフォーマッタインスタンスに対してこれを変更するには、`converter` 属性を `time.localtime()` や `time.gmtime()` と同じ署名をもつ関数に設定してください。すべてのフォーマッタインスタンスに対してこれを変更するには、例えば全てのロギング時刻を GMT で表示するには、`Formatter` クラスの `converter` 属性を設定してください。

formatException (*exc_info*)

指定された例外情報 (`sys.exc_info()` が返すような標準例外のタプル) を文字列として書式化します。デフォルトの実装は単に `traceback.print_exception()` を使います。結果の文字列が返されます。

15.7.5 フィルタオブジェクト

フィルタ (`Filter`) は、ハンドラ や ロガー によって使われ、レベルによって提供されるのよりも洗練されたフィルタリングを実現します。基底のフィルタクラスは、ロガー階層構造内の特定地点の配下にあるイベントだけを許可します。例えば、`'A.B'` で初期化されたフィルタは、ロガー `'A.B'`, `'A.B.C'`, `'A.B.C.D'`, `'A.B.D'` 等によって記録されたイベントは許可しますが、`'A.BB'`, `'B.A.B'` などは許可しません。空の文字列で初期化された場合、すべてのイベントを通過させます。

```
class logging.Filter(name="")
```

`Filter` クラスのインスタンスを返します。 `name` が指定されていれば、 `name` はロガーの名前を表します。指定されたロガーとその子ロガーのイベントがフィルタを通過できるようになります。 `name` が指定されなければ、すべてのイベントを通過させます。

```
filter(record)
```

指定されたレコードがログされるべきか？ no ならばゼロを、yes ならばゼロでない値を返します。適切と判断されれば、このメソッドによってレコードはその場で修正されることがあります。

ハンドラに対するフィルタはハンドラがイベントを発行する前に試され、一方ではロガーに対するフィルタは、イベントが (`debug()`, `info()` などによって) ログされる際には、ハンドラにイベントが送信される前にはいつでも試されることに注意してください。そのフィルタがそれら子孫ロガーにも適用されていない限り、子孫ロガーによって生成されたイベントはロガーのフィルタ設定によってフィルタされることはありません。

実際には、`Filter` をサブクラス化する必要はありません。同じ意味の `filter` メソッドを持つ、すべてのインスタンスを通せます。

フィルタは本来、レコードをレベルよりも洗練された基準に基づいてフィルタするために使われますが、それが取り付けられたハンドラやロガーによって処理されるレコードをすべて監視します。これは、特定のロガーやハンドラに処理されたレコードの数を数えたり、処理されている `LogRecord` の属性を追加、変更、削除したりするときに便利です。もちろん、`LogRecord` を変更するには注意が必要ですが、これにより、ログにコンテキスト情報を注入できます (`filters-contextual` を参照してください)。

15.7.6 LogRecord オブジェクト

`LogRecord` インスタンスは、何かをログ記録するたびに `Logger` によって生成されます。また、`makeLogRecord()` を通して (例えば、ワイヤを通して受け取られた pickle 化されたイベントから) 手動で生成することも出来ます。

```
class logging.LogRecord(name, level, pathname, lineno, msg, args, exc_info, func=None)
```

ログされているイベントに適切なすべての情報を含みます。

基本的な情報は `msg` と `args` に渡され、レコードの `message` フィールドは `msg % args` による結合で生成されます。

パラメータ

- `name` – この `LogRecord` で表されるイベントをログするのに使われるロガーの名前で

す。ここで与える名前が、たとえ他の (祖先の) ロガーに結び付けられたハンドラによって発せられるとしても、与えたこの値のままであることを注意してください。

- **level** – このロギングイベントの数値のレベル (DEBUG, INFO などのいずれか) です。なお、これは `LogRecord` の 2 つの 属性に変換されます。数値 `levelno` と、対応するレベル名 `levelname` です。
- **pathname** – ロギングの呼び出しが発せられたファイルの完全なパス名。
- **lineno** – ロギングの呼び出しが発せられたソース行番号。
- **msg** – イベント記述メッセージで、これは変数データのプレースホルダを持つフォーマット文字列になり得ます。
- **args** – `msg` 引数と組み合わせてイベント記述を得るための変数データです。
- **exc_info** – 現在の例外情報を含む例外タプルか、利用できる例外情報がない場合は `None` です。
- **func** – ロギングの呼び出しを行った関数またはメソッドの名前です。

バージョン 2.5 で変更: `func` が追加されました。

`getMessage()`

ユーザが提供した引数をメッセージに交ぜた後、この `LogRecord` インスタンスへのメッセージを返します。ユーザがロギングの呼び出しに与えた引数が文字列でなければ、その引数に `str()` が呼ばれ、文字列に変換されます。これにより、`__str__` メソッドが実際のフォーマット文字列を返せるようなユーザ定義のクラスをメッセージとして使えます。

15.7.7 `LogRecord` 属性

`LogRecord` には幾つかの属性があり、そのほとんどはコンストラクタのパラメタから得られます。(なお、`LogRecord` コンストラクタのパラメタと `LogRecord` 属性が常に厳密に対応するわけではありません。) これらの属性は、レコードからのデータをフォーマット文字列に統合するのに使えます。以下のテーブルに、属性名、意味、そして `%` 形式フォーマット文字列における対応するプレースホルダを (アルファベット順に) 列挙します。

属 性 名	フォーマット	説明
args	このフォーマットを自分で使う必要はないでしょう。	msg に組み合わせて message を生成するための引数のタプル、または、マージに用いられる辞書 (引数が一つしかなく、かつそれが辞書の場合)。
asc-time	%(asctime)s	<i>LogRecord</i> が生成された時刻を人間が読める書式で表したもの。デフォルトでは "2003-07-08 16:49:45,896" 形式 (コンマ以降の数字は時刻のミリ秒部分) です。
created	%(created)f	<i>LogRecord</i> が生成された時刻 (<i>time.time()</i> によって返される形式で)。
exc_info	このフォーマットを自分で使う必要はないでしょう。	(<i>sys.exc_info</i> 風の) 例外タプルか、例外が起こっていない場合は None。
filename	%(filename)s	pathname のファイル名部分。
func-Name	%(funcName)s	ロギングの呼び出しを含む関数の名前。
level-name	%(levelname)s	メッセージのための文字のロギングレベル ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL')。
levelno	%(levelno)s	メッセージのための数値のロギングレベル (DEBUG, INFO, WARNING, ERROR, CRITICAL)。
lineno	%(lineno)d	ロギングの呼び出しが発せられたソース行番号 (利用できる場合のみ)。
モジュール	%(module)s	モジュール (filename の名前部分)。
msecs	%(msecs)d	<i>LogRecord</i> が生成された時刻のミリ秒部分。
message	%(message)s	msg % args として求められた、ログメッセージ。 <i>Formatter.format()</i> が呼び出されたときに設定されます。
msg	このフォーマットを自分で使う必要はないでしょう。	元のロギングの呼び出しで渡されたフォーマット文字列。 args と合わせて、message、または任意のオブジェクトを生成します (arbitrary-object-messages 参照)。
name	%(name)s	ロギングに使われたロガーの名前。
pathname	%(pathname)s	ロギングの呼び出しが発せられたファイルの完全なパス名 (利用できる場合のみ)。
プロセス	%(process)d	プロセス ID (利用可能な場合のみ)。
process-Name	%(processName)s	プロセス名 (利用可能な場合のみ)。
relative-Created	%(relativeCreated)	logging モジュールが読み込まれた時刻に対する、LogRecord が生成された時刻を、ミリ秒で表したもの。
thread	%(thread)d	スレッド ID (利用可能な場合のみ)。
thread-Name	%(threadName)s	スレッド名 (利用可能な場合のみ)。

バージョン 2.5 で変更: *funcName* が追加されました。

バージョン 2.6 で変更: *processName* が追加されました。

15.7.8 LoggerAdapter オブジェクト

LoggerAdapter インスタンスは文脈情報をログ記録呼び出しに渡すのを簡単にするために使われます。使い方の例は コンテキスト情報をログ記録出力に付加する を参照してください。

バージョン 2.6 で追加.

class logging.LoggerAdapter(*logger*, *extra*)

内部で使う *Logger* インスタンスと辞書風 (dict-like) オブジェクトで初期化した *LoggerAdapter* のインスタンスを返します。

process (*msg*, *kwargs*)

文脈情報を挿入するために、ログ記録呼び出しに渡されたメッセージおよび/またはキーワード引数に変更を加えます。ここでの実装は *extra* としてコンストラクタに渡されたオブジェクトを取り、'extra' キーを使って *kwargs* に加えます。返り値は (*msg*, *kwargs*) というタプルで、(変更されているはずの) 渡された引数を含みます。

LoggerAdapter は上記に加え *Logger* のメソッド *debug()*, *info()*, *warning()*, *error()*, *exception()*, *critical()*, *log()*, *isEnabledFor()* をサポートします。これらは *Logger* の対応するメソッドと同じシグニチャを持つため、2 つのインスタンスは区別せずに利用出来ます。

バージョン 2.7 で変更: *isEnabledFor()* が *LoggerAdapter* に追加されました。このメソッドは元のロガーに処理を委譲します。

15.7.9 スレッドセーフ性

logging モジュールは、クライアントで特殊な作業を必要としない限りスレッドセーフになっています。このスレッドセーフ性はスレッドロックによって達成されています; モジュールの共有データへのアクセスを直列化するためのロックが一つ存在し、各ハンドラでも背後にある I/O へのアクセスを直列化するためにロックを生成します。

signal モジュールを使用して非同期シグナルハンドラを実装している場合、そのようなハンドラからはログ記録を使用できないかもしれません。これは、*threading* モジュールにおけるロック実装が常にリエントラントではなく、そのようなシグナルハンドラから呼び出すことができないからです。

15.7.10 モジュールレベルの関数

上で述べたクラスに加えて、いくつかのモジュールレベルの関数が存在します。

logging.getLogger([*name*])

指定された名前のロガーを返します。名前が指定されていなければ、ロガー階層のルート (root) にある

ロガーを返します。 *name* を指定する場合には、通常は *"a"*, *"a.b"*, *"a.b.c.d"* といったドット区切りの階層的な名前にします。名前の付け方はログ機能を使う開発者次第です。

与えられた名前に対して、この関数はどの呼び出しでも同じロガーインスタンスを返します。したがって、ロガーインスタンスをアプリケーションの各部でやりとりする必要はありません。

`logging.getLoggerClass()`

標準の *Logger* クラスが、最後に *setLoggerClass()* に渡したクラスを返します。この関数は、新たなクラス定義の中で呼び出して、カスタマイズした *Logger* クラスのインストールが既に他のコードで適用したカスタマイズを取り消さないことを保証するために使われることがあります。例えば以下のようにします:

```
class MyLogger(logging.getLoggerClass()):
    # ... override behaviour here
```

`logging.debug(msg[, *args[, **kwargs]])`

レベル *DEBUG* のメッセージをルートロガーで記録します。 *msg* はメッセージの書式化文字列で、 *args* は *msg* に文字列書式化演算子を使って取り込むための引数です。(これは、書式化文字列の中でキーワードを使い、引数として単一の辞書を渡すことができる、ということを意味します。)

キーワード引数 *kwargs* からは 2 つのキーワードが調べられます。一つ目は *exc_info* で、この値の評価値が *false* でない場合、例外情報をログメッセージに追加します。(*sys.exc_info()* の返す形式の) 例外情報を表すタプルが与えられていれば、それをメッセージに使います。それ以外の場合には、 *sys.exc_info()* を呼び出して例外情報を取得します。

ほかのキーワード引数は *extra* で、当該ロギイベント用に作られる *LogRecord* の *__dict__* にユーザー定義属性を加えるのに使われる辞書を渡すために用いられます。これらの属性は好きなように使えます。たとえば、ログメッセージの一部にすることもできます。以下の例を見てください:

```
FORMAT = "%(asctime)-15s %(clientip)s %(user)-8s %(message)s"
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logging.warning("Protocol problem: %s", "connection reset", extra=d)
```

これは以下のような出力を行います:

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs Protocol problem: connection_
↵reset
```

extra で渡される辞書のキーはロギングシステムで使われているものと衝突しないようにしなければなりません。(どのキーがロギングシステムで使われているかについての詳細は *Formatter* のドキュメントを参照してください。)

これらの属性をログメッセージに使うことにしたなら、少し注意が必要です。上の例では、 *'clientip'* と *'user'* が *LogRecord* の属性辞書に含まれていることを期待した書式化文字列で *Formatter* がセットアップされています。もしこれらが欠けていると、書式化例外が発生してしまうためメッセージはログに残りません。したがってこの場合、常にこれらのキーを含む *extra* 辞書を渡す必要があります。

このようなことは煩わしいかもしれませんが、この機能は限定された場面で使われるように意図してい

るものなのです。たとえば同じコードがいくつかのコンテキストで実行されるマルチスレッドのサーバで、興味のある条件が現れるのがそのコンテキストに依存している（上の例で言えば、リモートのクライアント IP アドレスや認証されたユーザ名など）、というような場合です。そういった場面では、それ用の *Formatter* が特定の *Handler* と共に使われるというのはよくあることです。

バージョン 2.5 で変更: *extra* が追加されました。

```
logging.info(msg[, *args[, **kwargs]])
```

レベル INFO のメッセージをルートロガーで記録します。引数は *debug()* と同じように解釈されます。

```
logging.warning(msg[, *args[, **kwargs]])
```

レベル WARNING のメッセージをルートロガーで記録します。引数は *debug()* と同じように解釈されます。

```
logging.error(msg[, *args[, **kwargs]])
```

レベル ERROR のメッセージをルートロガーで記録します。引数は *debug()* と同じように解釈されます。

```
logging.critical(msg[, *args[, **kwargs]])
```

レベル CRITICAL のメッセージをルートロガーで記録します。引数は *debug()* と同じように解釈されます。

```
logging.exception(msg[, *args[, **kwargs]])
```

レベル ERROR のメッセージをルートロガーで記録します。引数は、渡された *exc_info* の詳細が展開されないことを除いて、*debug()* と同じように解釈されます。例外情報が常にログメッセージに追加されます。この関数は例外ハンドラからのみ呼び出されるべきです。

```
logging.log(level, msg[, *args[, **kwargs]])
```

レベル *level* のメッセージをルートロガーで記録します。その他の引数は *debug()* と同じように解釈されます。

注釈: 上述の便利なルートロガーに処理を委譲するモジュールレベル関数は *basicConfig()* を呼び出して、少なくとも 1 つのハンドラが利用できることを保証します。これにより Python の 2.7.1 以前や 3.2 以前のバージョンでは、スレッドが開始される 前に 少なくともひとつのハンドラがルートロガーに加えられるのでない限り、スレッド内で使うべき ではありません。以前のバージョンの Python では、*basicConfig()* のスレッドセーフ性の欠陥により、(珍しい状況下とはいえ) ハンドラがルートロガーに複数回加えられることがあり、ログ内のメッセージが重複するという予期しない結果をもたらすことがあります。

```
logging.disable(lvl)
```

全てのロガーのレベル *lvl* を上書きし、これはロガー自身の出力レベルよりも優先されます。アプリケーション全体を横断するログ出力を一時的に調整する必要が生じたら、この関数は便利でしょう。この効果は重大度 *lvl* 以下の全てのロギング呼び出しを無効にすることですので、INFO で呼び出しをすれば、INFO と DEBUG イベントが捨てられる一方で、重大度 WARNING 以上のものは、ロガーの有効レベルに基いて処理されます。 `logging.disable(logging.NOTSET)` が呼び出されると、こ

の上書きレベルは削除され、ログ出力は再び個々のロガーの有効レベルに依存するようになります。

`logging.addLevelName(lvl, levelName)`

内部的な辞書の中でレベル `lvl` をテキスト `levelName` に関連付けます。これは例えば `Formatter` でメッセージを書式化する際のように、数字のレベルをテキスト表現に対応付ける際に用いられます。この関数は自作のレベルを定義するために使うこともできます。使われるレベルに対する唯一の制限は、レベルは正の整数でなくてはならず、メッセージの深刻度が上がるに従ってレベルの数も上がらなくてはならないということです。

注釈: もしもあなたが独自のレベルを定義したいと考えるならば、どうか `custom-levels` のセクションをご参照下さいませようお願いします。

`logging.getLevelName(lvl)`

ログ記録レベル `lvl` のテキスト表現を返します。レベルが定義済みのレベル `CRITICAL`, `ERROR`, `WARNING`, `INFO`, `DEBUG` のいずれかである場合、対応する文字列が返されます。 `addLevelName()` を使ってレベルに名前に関連付けていた場合、 `lvl` に関連付けられた名前が返されます。定義済みのレベルに対応する数値を指定した場合、レベルに対応した文字列表現を返します。そうでない場合、文字列 `"Level %s" % lvl` を返します。

注釈: たとえば `Logger` のインスタンスとハンドラへの設定をする場合には、数値のレベルを使用すべきです。この関数は、数値のレベルを、書式記述子 `%(levelname)s` (`LogRecord` 属性 参照) によって書式化されるログ出力の表示用レベル名に変換するのに使用されます。

`logging.makeLogRecord(attrdict)`

属性が `attrdict` で定義された、新しい `LogRecord` インスタンスを生成して返します。この関数は、pickle された `LogRecord` 属性の辞書をソケットを介して送信し、受信端で `LogRecord` インスタンスとして再構成する場合に便利です。

`logging.basicConfig(**kwargs)`

デフォルトの `Formatter` を持つ `StreamHandler` を生成してルートロガーに追加し、ロギングシステムの基本的な環境設定を行います。関数 `debug()`, `info()`, `warning()`, `error()`, `critical()` は、ルートロガーにハンドラが定義されていない場合に自動的に `basicConfig()` を呼び出します。

この関数はルートロガーに設定されたハンドラがあれば何もしません。

バージョン 2.4 で変更: 以前は `basicConfig()` はキーワード引数を取りませんでした。

注釈: この関数は、他のスレッドが開始される前にメインスレッドから呼び出されるべきです。Python の 2.7.1 や 3.2 以前のバージョンでは、この関数が複数のスレッドから呼ばれると(珍しい状況下とはいえ) ハンドラがルートロガーに複数回加えられることがあり、ログ内のメッセージが重複するという予期しない結果をもたらすことがあります。

以下のキーワード引数がサポートされます。

フォーマット	説明
<i>filename</i>	StreamHandler ではなく指定された名前で FileHandler が作られます。
<i>filemode</i>	If <i>filename</i> is specified, open the file in this mode. Defaults to 'a'.
<i>format</i>	指定された書式化文字列をハンドラで使います。
<i>datefmt</i>	指定された日時の書式で <code>time.strftime()</code> が受け付けるものを使います。
<i>level</i>	ルートロガーのレベルを指定された レベル に設定します。
<i>stream</i>	Use the specified stream to initialize the StreamHandler. Note that this argument is incompatible with <i>filename</i> - if both are present, <i>stream</i> is ignored.

`logging.shutdown()`

ロギングシステムに対して、バッファのフラッシュを行い、すべてのハンドラを閉じることで順次シャットダウンを行うように告知します。この関数はアプリケーションの終了時に呼ばれるべきであり、また呼び出し以降はそれ以上ロギングシステムを使ってはなりません。

`logging.setLoggerClass(klass)`

ロギングシステムに対して、ロガーをインスタンス化する際にクラス *klass* を使うように指示します。指定するクラスは引数として名前だけをとるようなメソッド `__init__()` を定義していなければならず、`__init__()` では `Logger.__init__()` を呼び出さなければなりません。典型的な利用法として、この関数は自作のロガーを必要とするようなアプリケーションにおいて、他のロガーがインスタンス化される前にインスタンス化されます。

15.7.11 warnings モジュールとの統合

`captureWarnings()` 関数を使って、`logging` を `warnings` モジュールと統合できます。

`logging.captureWarnings(capture)`

この関数は、`logging` による警告の補足を、有効にまたは無効にします。

`capture` が `True` なら、`warnings` モジュールに発せられた警告は、ロギングシステムにリダイレクトされるようになります。具体的には、警告が `warnings.formatwarning()` でフォーマット化され、結果の文字列が 'py.warnings' という名のロガーに、WARNING の重大度でロギングされるようになります。

`capture` が `False` なら、警告のロギングシステムに対するリダイレクトは止められ、警告は元の (すなわち、`captureWarnings(True)` が呼び出される前に有効だった) 送信先にリダイレクトされるようになります。

参考:

`logging.config` モジュール `logging` モジュールの環境設定 API です。

`logging.handlers` モジュール `logging` モジュールに含まれる、便利なハンドラです。

PEP 282 - ログシステム この機能を Python 標準ライブラリに含めることを述べた提案です。

Original Python logging package これは、`logging` パッケージのオリジナルのソースです。このサイトから利用できるバージョンのパッケージは、`logging` パッケージを標準ライブラリに含まない、Python 1.5.2, 2.1.x および 2.2.x で使うのに適しています。

15.8 logging.config — ロギングの環境設定

Important

このページには、リファレンス情報だけが含まれています。チュートリアルは、以下のページを参照してください

- 基本チュートリアル
- 上級チュートリアル
- ロギングクックブック

ソースコード: [Lib/logging/config.py](#)

この節は、`logging` モジュールを設定するための API を解説します。

15.8.1 環境設定のための関数

以下の関数は `logging` モジュールの環境設定をします。これらの関数は、`logging.config` にあります。これらの関数の使用はオプションです — `logging` モジュールはこれらの関数を使うか、(`logging` 自体で定義されている) 主要な API を呼び出し、`logging` か `logging.handlers` で宣言されているハンドラを定義することで設定できます。

`logging.config.dictConfig(config)`

辞書からロギング環境設定を取得します。この辞書の内容は、以下の [環境設定辞書スキーマ](#) で記述されています。

環境設定中にエラーに遭遇すると、この関数は適宜メッセージを記述しつつ `ValueError`, `TypeError`, `AttributeError` または `ImportError` を送出します。例外を送出する条件を(不完全かもしれませんが) 以下に列挙します:

- 文字列でなかったり、実際のロギングレベルと関係ない文字列であったりする `level`。
- ブール値でない `propagate` の値。
- 対応する行き先を持たない `id`。
- インクリメンタルな呼び出しの中で見つかった存在しないハンドラ `id`。
- 無効なロガー名。

- 内部や外部のオブジェクトに関わる不可能性。

解析は DictConfigurator クラスによって行われます。このクラスのコンストラクタは環境設定に使われる辞書に渡され、このクラスは `configure()` メソッドを持ちます。 `logging.config` モジュールは、呼び出し可能属性 `dictConfigClass` を持ち、これはまず DictConfigurator に設定されます。 `dictConfigClass` の値は適切な独自の実装で置き換えられます。

`dictConfig()` は `dictConfigClass` を、指定された辞書を渡して呼び出し、それから返されたオブジェクトの `configure()` メソッドを呼び出して、環境設定を作用させます:

```
def dictConfig(config):
    dictConfigClass(config).configure()
```

例えば、DictConfigurator のサブクラスは、自身の `__init__()` で DictConfigurator. `__init__()` を呼び出し、それから続く `configure()` の呼び出しに使えるカスタムの接頭辞を設定できます。 `dictConfigClass` は、この新しいサブクラスに束縛され、そして `dictConfig()` はちょうどデフォルトの、カスタマイズされていない状態のように呼び出せます。

バージョン 2.7 で追加.

`logging.config.fileConfig(fname, defaults=None, disable_existing_loggers=True)`

ログ記録の環境設定を `configparser` 形式のファイル `fname` から読み出します。そのファイルの形式は [環境設定ファイルの書式](#) で記述されているとおりにならなければなりません。この関数はアプリケーションから何度も呼び出すことができ、これによって、(設定を選択し、選択された設定を読み出す機構をデベロッパが提供していれば) 複数の準備済みの設定からエンドユーザが選択するようにできます。

パラメータ

- **defaults** – ConfigParser に渡されるデフォルト値をこの引数で指定することができます。
- **disable_existing_loggers** – `False` が指定された場合は、この呼び出しが行われたときに存在するロガーは有効のまま残されます。後方互換性のあるやり方で古い振る舞いを保つので、デフォルト値は `True` になっています。そのような振る舞いでは、既存のロガーまたはそれらのロガーの先祖がロギング設定の中で明示的に名付けられていない限り、既存のロガーを無効にします。

バージョン 2.6 で変更: `disable_existing_loggers` キーワード引数が追加されました。以前は、既存のロガーは必ず 無効にされていました。

`logging.config.listen(port=DEFAULT_LOGGING_CONFIG_PORT)`

指定されたポートでソケットサーバを起動し、新しい設定を待ち受けます。ポートが指定されなかった場合は、モジュールのデフォルトの `DEFAULT_LOGGING_CONFIG_PORT` が使用されます。ロギング設定は `fileConfig()` で処理できるファイルとして送信されます。 `Thread` インスタンスを返し、このインスタンスの `start()` を呼び出してサーバを起動し、適切なところで `join()` を呼び出すことができます。サーバを停止するには、 `stopListening()` を呼び出します。

ソケットに設定を送るには、まず設定ファイルを読み、それを `struct.pack('>L', n)` を使って長さ 4 バイトのバイナリにパックしたものを前に付けたバイト列としてソケットに送ります。

注釈: 設定の部分が `eval()` を通して渡されるので、この関数の使用はユーザに対してセキュリティリスクを公開してしまうかもしれません。この関数は単に `localhost` 上のソケットに接続してリモートマシンからは接続を受け付けませんが、`listen()` を呼んだプロセスのアカウントで信頼されていないコードが実行されるシナリオが存在します。特に、`listen()` を呼んだプロセスがユーザがお互いを信頼することができないマルチユーザのマシン上で実行される場合、悪意のあるユーザは、犠牲者のユーザのプロセスで本質的に任意のコードを実行するように細工することができます。単に犠牲者の `listen()` ソケットに接続して、犠牲者のプロセスで実行したいコードを実行するような設定を送るだけです。これは、特にデフォルトポートが使用されている場合に行うのがより簡単ですが、異なるポートが使用されていたとしてもそれほど難しくありません。

`logging.config.stopListening()`

`listen()` を呼び出して作成された、待ち受け中のサーバを停止します。通常 `listen()` の戻り値に対して `join()` が呼ばれる前に呼び出します。

15.8.2 環境設定辞書スキーマ

ロギング設定を記述するには、生成するさまざまなオブジェクトと、それらのつながりを列挙しなければなりません。例えば、`'console'` という名前のハンドラを生成し、`'startup'` という名前のロガーがメッセージを `'console'` ハンドラに送るというようなことを記述します。これらのオブジェクトは、`logging` モジュールによって提供されるものに限らず、独自のフォーマッタやハンドラクラスを書くことも出来ます。このクラスへのパラメータは、`sys.stderr` のような外部オブジェクトを必要とすることもあります。これらのオブジェクトとつながりを記述する構文は、以下の [オブジェクトの接続](#) で定義されています。

辞書スキーマの詳細

`dictConfig()` に渡される辞書は、以下のキーを含んでいなければなりません:

- `version` - スキーマのバージョンを表す整数値に設定されます。現在有効な値は 1 ですが、このキーがあることで、このスキーマは後方互換性を保ちながら発展できます。

その他すべてのキーは省略可能ですが、与えられたなら以下に記述するように解釈されます。以下のすべての場合において、`'環境設定辞書'` と記載されている所では、その辞書に特殊な `'()'` キーがあるかを調べることで、カスタムのインスタント化が必要であるか判断されます。その場合は、以下の [ユーザ定義オブジェクト](#) で記述されている機構がインスタンス生成に使われます。そうでなければ、インスタンス化するべきものを決定するのにコンテキストが使われます。

- `formatters` - 対応する値は辞書で、そのそれぞれのキーがフォーマッタ id になり、それぞれの値が対応する `Formatter` インスタンスをどのように環境設定するかを記述する辞書になります。

環境設定辞書から、(デフォルトが `None` の) キー `format` と `datefmt` を検索し、それらが `Formatter` インスタンスを構成するのに使われます。

- *filters* - 対応する値は辞書で、そのそれぞれのキーがフィルタ id になり、それぞれの値が対応する Filter インスタンスをどのように環境設定するかを記述する辞書になります。

環境設定辞書は、(デフォルトが空文字列の) キー `name` を検索され、それらが `logging.Filter` インスタンスを構成するのに使われます。

- *handlers* - 対応する値は辞書で、そのそれぞれのキーがハンドラ id になり、それぞれの値が対応する Handler インスタンスをどのように環境設定するかを記述する辞書になります。

環境設定辞書は、以下のキーを検索されます:

- `class` (必須)。これはハンドラクラスの完全に修飾された名前です。
- `level` (任意)。ハンドラのレベルです。
- `formatter` (任意)。このハンドラへのフォーマッタの id です。
- `filters` (任意)。このハンドラへのフィルタの id のリストです。

その他の すべての キーは、ハンドラのコンストラクタにキーワード引数として渡されます。例えば、以下のコード片が与えられたとすると:

```
handlers:
  console:
    class : logging.StreamHandler
    formatter: brief
    level   : INFO
    filters: [allow_foo]
    stream  : ext://sys.stdout
  file:
    class : logging.handlers.RotatingFileHandler
    formatter: precise
    filename: logconfig.log
    maxBytes: 1024
    backupCount: 3
```

id が `console` であるハンドラが、`sys.stdout` を根底のストリームにして、`logging.StreamHandler` としてインスタンス化されます。id が `file` であるハンドラが、`filename='logconfig.log'`, `maxBytes=1024`, `backupCount=3` をキーワード引数にして、`logging.handlers.RotatingFileHandler` としてインスタンス化されます。

- *loggers* - 対応する値は辞書で、そのそれぞれのキーがロガー名になり、それぞれの値が対応する Logger インスタンスをどのように環境設定するかを記述する辞書になります。

環境設定辞書は、以下のキーを検索されます:

- `level` (任意)。ロガーのレベルです。
- `propagate` (任意)。ロガーの伝播の設定です。
- `filters` (任意)。このロガーへのフィルタの id のリストです。
- `handlers` (任意)。このロガーへのハンドラの id のリストです。

指定されたロガーは、指定されたレベル、伝播、ハンドラに従って環境設定されます。

- *root* - これは、ルートロガーへの設定になります。この環境設定の進行は、`propagate` 設定が適用されないことを除き、他のロガーと同じです。
- *incremental* - この環境設定が既存の環境設定に対する増分として解釈されるかどうかです。この値のデフォルトは `False` で、指定された環境設定は、既存の `fileConfig()` API によって使われているのと同じ意味上で、既存の環境設定を置き換えます。

指定された値が `True` なら、環境設定は [増分設定](#) の節で記述されているように進行します。

- *disable_existing_loggers* - 既存のロガーをすべて無効にするべきかどうかです。この設定は、`fileConfig()` における同じ名前のパラメータと同じです。設定されていなければ、このパラメータのデフォルトは `True` です。この値は、*incremental* が `True` なら無視されます。

増分設定

増分設定に完全な柔軟性を提供するのには難しいです。例えば、フィルタやフォーマッタのようなオブジェクトは匿名なので、一旦環境設定がなされると、設定を拡張するときにそのような匿名オブジェクトを参照することができません。

さらに、一旦環境設定がなされた後、実行時にロガー、ハンドラ、フィルタ、フォーマッタのオブジェクトグラフを任意に変えなければならない例もありません。ロガーとハンドラの冗長性は、レベル (または、ロガーの場合には、伝播フラグ) を設定することによってのみ制御できます。安全な方法でオブジェクトグラフを任意に変えることは、マルチスレッド環境で問題となります。不可能ではないですが、その効用は実装に加えられる複雑さに見合いません。

従って、環境設定辞書の *incremental* キーが与えられ、これが `True` であるとき、システムは `formatters` と `filters` の項目を完全に無視し、`handlers` の項目の `level` 設定と、`loggers` と `root` の項目の `level` と `propagate` 設定のみを処理します。

環境設定辞書の値を使うことで、設定は pickle 化された辞書としてネットワークを通してソケットリスナに送ることができます。これにより、長時間起動するアプリケーションのロギングの冗長性を、アプリケーションを止めて再起動する必要なしに、いつでも変更することができます。

オブジェクトの接続

このスキーマは、ロギングオブジェクトの一揃い - ロガー、ハンドラ、フォーマッタ、フィルタ - について記述します。これらは、オブジェクトグラフ上でお互い接続されます。従って、このスキーマは、オブジェクト間の接続を表現しなければなりません。例えば、環境設定で、特定のロガーが特定のハンドラに取り付けられたとします。この議論では、ロガーとハンドラが、これら 2 つの接続のそれぞれ送信元と送信先であるといえます。もちろん、この設定オブジェクト中では、これはハンドラへの参照を保持しているロガーで表されます。設定辞書中で、これは次のようになされます。まず、送信先オブジェクトを曖昧さなく指定する `id` を与えます。そして、その `id` を送信元オブジェクトの環境設定で使い、送信元とその `id` をもつ送信先が接続されていることを示します。

ですから、例えば、以下の YAML のコード片を例にとると:

```

formatters:
  brief:
    # configuration for formatter with id 'brief' goes here
  precise:
    # configuration for formatter with id 'precise' goes here
handlers:
  h1: #This is an id
    # configuration of handler with id 'h1' goes here
    formatter: brief
  h2: #This is another id
    # configuration of handler with id 'h2' goes here
    formatter: precise
loggers:
  foo.bar.baz:
    # other configuration for logger 'foo.bar.baz'
    handlers: [h1, h2]

```

(注釈: YAML がここで使われているのは、辞書の等価な Python 形式よりもこちらのほうが少し読みやすいからです。)

ロガーの id は、プログラム上でロガーへの参照を得るために使われるロガー名で、たとえば `foo.bar.baz` です。フォーマッタとフィルタの id は、(上の `brief`, `precise` のような) 任意の文字列値にできます。これらは一時的なもので、環境設定辞書の処理にのみ意味があり、オブジェクト間の接続を決定するのに使われま
す。また、これらは設定の呼び出しが完了したとき、どこにも残りません。

上記のコード片は、`foo.bar.baz` というの名ロガーに、ハンドラ `h1` と `h2` で表される 2 つのハンドラを接続することを示します。`h1` のフォーマッタは `id brief` で記述されるもので、`h2` のフォーマッタは `id precise` で記述されるものです。

ユーザ定義オブジェクト

このスキーマは、ハンドラ、フィルタ、フォーマッタのための、ユーザ定義オブジェクトをサポートします。(ロガーは、異なるインスタンスに対して異なる型を持つ必要はないので、この環境設定スキーマは、ユーザ定義ロガークラスをサポートしていません。)

設定されるオブジェクトは、それらの設定を詳述する辞書によって記述されます。場所によっては、あるオブジェクトがどのようにインスタンス化されるかというコンテキストを、ロギングシステムが推測できます。しかし、ユーザ定義オブジェクトがインスタンス化されるとき、システムはどのようにこれを行うかを知りません。ユーザ定義オブジェクトのインスタンス化を完全に柔軟なものにするため、ユーザは 'ファクトリ' - 設定辞書を引数として呼ばれ、インスタンス化されたオブジェクトを返す呼び出し可能オブジェクト - を提供する必要があります。これは特殊キー '()' で利用できる、ファクトリへの絶対インポートパスによって合図されます。ここに具体的な例を挙げます:

```

formatters:
  brief:
    format: '%(message)s'
  default:
    format: '%(asctime)s %(levelname)-8s %(name)-15s %(message)s'

```

(次のページに続く)

(前のページからの続き)

```

datefmt: '%Y-%m-%d %H:%M:%S'
custom:
    (): my.package.customFormatterFactory
    bar: baz
    spam: 99.9
    answer: 42

```

上記の YAML コード片は 3 つのフォーマッタを定義します。1 つ目は、id が `brief` で、指定されたフォーマット文字列をもつ、標準 `logging.Formatter` インスタンスです。2 つ目は、id が `default` で、長いフォーマットを持ち、時間フォーマットも定義していて、結果はその 2 つのフォーマット文字列で初期化された `logging.Formatter` になります。Python ソース形式で見ると、`brief` と `default` フォーマッタは、それぞれ設定の部分辞書:

```

{
    'format' : '%(message)s'
}

```

および:

```

{
    'format' : '%(asctime)s %(levelname)-8s %(name)-15s %(message)s',
    'datefmt' : '%Y-%m-%d %H:%M:%S'
}

```

を持ち、これらの辞書が特殊キー `'()'` を持たないので、インスタンス化はコンテキストから推測され、結果として標準の `logging.Formatter` インスタンスが生成されます。id が `custom` である、3 つ目のフォーマッタの設定をする部分辞書は:

```

{
    '()' : 'my.package.customFormatterFactory',
    'bar' : 'baz',
    'spam' : 99.9,
    'answer' : 42
}

```

で、ユーザ定義のインスタンス化が望まれることを示す特殊キー `'()'` を含みます。この場合、指定された呼び出し可能ファクトリオブジェクトが使われます。これが実際の呼び出し可能オブジェクトであれば、それが直接使われます - そうではなく、(この例のように) 文字列を指定したなら、実際の呼び出し可能オブジェクトは、通常のインポート機構を使って検索されます。その呼び出し可能オブジェクトは、環境設定の部分辞書の、残りの要素をキーワード引数として呼ばれます。上記の例では、id が `custom` のフォーマッタは、以下の呼び出しによって返されるものとみなされます:

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

キー `'()'` が特殊キーとして使われるのは、キーワードパラメータ名として不正で、呼び出しに使われるキーワード引数と衝突し得ないからです。 `'()'` はまた、対応する値が呼び出し可能オブジェクトであると覚えやすくします。

外部オブジェクトへのアクセス

環境設定が、例えば `sys.stderr` のような、設定の外部のオブジェクトへの参照を必要とすることがあります。設定辞書が Python コードで構成されていれば話は簡単ですが、これがテキストファイル (JSON, YAML 等) を通して提供されていると問題となります。テキストファイルでは、`sys.stderr` をリテラル文字列 `'sys.stderr'` と区別する標準の方法がありません。この区別を容易にするため、環境設定システムは、文字列中の特定の特殊接頭辞を見つけ、それらを特殊に扱います。例えば、リテラル文字列 `'ext://sys.stderr'` が設定中の値として与えられたら、この `ext://` は剥ぎ取られ、この値の残りが普通のインポート機構で処理されます。

このような接頭辞の処理は、プロトコルの処理と同じようになされます。どちらの機構も、正規表現 `^(?P<prefix>[a-z]+):/(?P<suffix>.*)$` にマッチする接頭辞を検索し、それによって `prefix` が認識されたなら、接頭辞に応じたやり方で `suffix` が処理され、その処理の結果によって文字列値が置き換えられます。接頭辞が認識されなければ、その文字列値はそのまま残されます。

内部オブジェクトへのアクセス

外部オブジェクトと同様、環境設定内部のオブジェクトへのアクセスを必要とすることもあります。これは、その各オブジェクトを司る環境設定システムによって暗黙に行われます。例えば、ロガーやハンドラの `level` に対する文字列値 `'DEBUG'` は、自動的に値 `logging.DEBUG` に変換されますし、`handlers`, `filters` および `formatter` の項目は、オブジェクト `id` を取って、適切な送信先オブジェクトを決定します。

しかし、ユーザ定義モジュールには、`logging` モジュールには分からないような、より一般的な機構が必要です。例えば、`logging.handlers.MemoryHandler` があって、委譲する先の別のハンドラである `target` 引数を取るとします。システムはこのクラスをすでに知っているから、設定中で、与えられた `target` は関連するターゲットハンドラのオブジェクト `id` でさえあればよく、システムはその `id` からハンドラを決定します。しかし、ユーザが `my.package.MyHandler` を定義して、それが `alternate` ハンドラを持つなら、設定システムは `alternate` がハンドラを参照していることを知りません。これを知らせるのに、一般的な解析システムで、ユーザはこのように指定できます:

```
handlers:
  file:
    # configuration of file handler goes here

  custom:
    (): my.package.MyHandler
    alternate: cfg://handlers.file
```

リテラル文字列 `'cfg://handlers.file'` は、`ext://` 接頭辞が付いた文字列と同じように分析されますが、インポート名前空間ではなく、環境設定自体が検索されます。この機構は `str.format` でできるのと同じようにドットやインデックスのアクセスができます。従って、環境設定において以下のコード片が与えられれば:

```
handlers:
  email:
    class: logging.handlers.SMTPHandler
    mailhost: localhost
```

(次のページに続く)

(前のページからの続き)

```

fromaddr: my_app@domain.tld
toaddrs:
  - support_team@domain.tld
  - dev_team@domain.tld
subject: Houston, we have a problem.

```

文字列 `'cfg://handlers'` は、キー `handlers` をもつ辞書であると分析され、文字列 `'cfg://handlers.email'` は、`handlers` 辞書内の、`email` キーをもつ辞書であると分析されます。文字列 `'cfg://handlers.email.toaddrs[1]'` は、`'dev_team@domain.tld'` と分析され、`'cfg://handlers.email.toaddrs[0]'` は値 `'support_team@domain.tld'` と分析されます。`subject` の値には、`'cfg://handlers.email.subject'` または等価な `'cfg://handlers.email[subject]'` でアクセスできます。後者が必要なのは、キーがスペースや非アルファベット文字を含むときのみです。インデックス値が十進数字のみで構成されているなら、まず対応する整数値を使ってアクセスが試みられ、必要なら文字列値で代替します。

文字列 `cfg://handlers.myhandler.mykey.123` が与えられると、これは `config_dict['handlers']['myhandler']['mykey']['123']` と分析されます。文字列が `cfg://handlers.myhandler.mykey[123]` と指定されたら、システムは `config_dict['handlers']['myhandler']['mykey'][123]` から値を引き出そうとし、失敗したら `config_dict['handlers']['myhandler']['mykey']['123']` で代替します。

インポート解決とカスタムインポーター

インポート解決は、デフォルトではインポートを行うために `__import__()` 組み込み関数を使用します。これを独自のインポートメカニズムに置き換えたいと思うかもしれませんが、もしそうなら、`DictConfigurator` あるいはその上位クラスである `BaseConfigurator` クラスの `importer` 属性を置換することができます。ただし、この関数はクラスからディスクリプタ経由でアクセスされる点に注意する必要があります。インポートを行うために Python callable を使用していて、それをインスタンスレベルではなくクラスレベルで定義したければ、`staticmethod()` でそれをラップする必要があります。例えば:

```

from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)

```

`configurator` インスタンス に対してインポート callable をセットする場合は、`staticmethod()` でラップする必要はありません。

15.8.3 環境設定ファイルの書式

`fileConfig()` が解釈できる環境設定ファイルの形式は、`configparser` の機能に基づいています。ファイルには、`[loggers]`, `[handlers]`, `[formatters]` といったセクションが入っていなければならず、各セクションではファイル中で定義されている各タイプのエンティティを名前指定しています。こうしたエンティティの各々について、そのエンティティをどう設定するかを示した個別のセクションがあります。すな

わち、log01 という名前の [loggers] セクションにあるロガーに対しては、対応する詳細設定がセクション [logger_log01] に収められています。同様に、hand01 という名前の [handlers] セクションにあるハンドラは [handler_hand01] と呼ばれるセクションに設定をもつことになり、[formatters] セクションにある form01 は [formatter_form01] というセクションで設定が指定されています。ルートロガーの設定は [logger_root] と呼ばれるセクションで指定されていなければなりません。

注釈: `fileConfig()` API は `dictConfig()` API よりも古く、ロギングのある種の側面についてカバーする機能に欠けています。たとえば `fileConfig()` では数値レベルを超えたメッセージを単に拾うフィルタリングを行う `Filter` オブジェクトを構成出来ません。`Filter` のインスタンスをロギングの設定において持つ必要があるならば、`dictConfig()` を使う必要があるでしょう。設定の機能における将来の拡張は `dictConfig()` に対して行われることに注意してください。ですから、そうするのが便利であるときに新しい API に乗り換えるのは良い考えです。

ファイルにおけるこれらのセクションの例を以下に示します。

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

ルートロガーでは、レベルとハンドラのリストを指定しなければなりません。ルートロガーのセクションの例を以下に示します。

```
[logger_root]
level=NOTSET
handlers=hand01
```

level エントリは DEBUG, INFO, WARNING, ERROR, CRITICAL のうちのの一つか、NOTSET になります。ルートロガーの場合にのみ、NOTSET はすべてのメッセージがログ記録されることを意味します。レベル値は logging パッケージの名前空間のコンテキストにおいて `eval()` されます。

handlers エントリはコンマで区切られたハンドラ名からなるリストで、[handlers] セクションになくてもなりません。また、これらの各ハンドラの名前に対応するセクションが設定ファイルに存在しなければなりません。

ルートロガー以外のロガーでは、いくつか追加の情報が必要になります。これは以下の例のように表されます。

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

`level` および `handlers` エントリはルートロガーのエントリと同様に解釈されますが、非ルートロガーのレベルが `NOTSET` に指定された場合、ロギングシステムはロガー階層のより上位のロガーにロガーの実効レベルを問い合わせるところが違います。 `propagate` エントリは、メッセージをロガー階層におけるこのロガーの上位のハンドラに伝播させることを示す 1 に設定されるか、メッセージを階層の上位に伝播しないことを示す 0 に設定されます。 `qualname` エントリはロガーのチャンネル名を階層的に表したもの、すなわちアプリケーションがこのロガーを取得する際に使う名前になります。

ハンドラの環境設定を指定しているセクションは以下の例のようになります。

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

`class` エントリはハンドラのクラス (`logging` パッケージの名前空間において `eval()` で決定されます) を示します。 `level` はロガーの場合と同じように解釈され、 `NOTSET` は ”すべてを記録する (log everything)” と解釈されます。

バージョン 2.6 で変更: ハンドラクラスのドット区切りモジュールおよびクラス名としての解決のサポートが追加されました。

`formatter` エントリはこのハンドラのフォーマッタに対するキー名を表します。空文字列の場合、デフォルトのフォーマッタ (`logging.defaultFormatter`) が使われます。名前が指定されている場合、その名前は `[formatters]` セクションになくてもならず、対応するセクションが設定ファイル中になければなりません。

`args` エントリは、 `logging` パッケージの名前空間のコンテキストで `eval()` される際、ハンドラクラスのコンストラクタに対する引数からなるリストになります。典型的なエントリがどうやって作成されるかについては、対応するハンドラのコンストラクタが、以下の例を参照してください。

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
```

(次のページに続く)

(前のページからの続き)

```
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args= (('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
```

フォーマッタの環境設定を指定しているセクションは以下のような形式です。

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s
datefmt=
class=logging.Formatter
```

`format` エントリは全体を書式化する文字列で、`datefmt` エントリは `strftime()` 互換の日付/時刻書式化文字列です。空文字列の場合、パッケージによって ISO8601 形式の日付/時刻に置き換えられ、日付書式化文字列 `\'%Y-%m-%d %H:%M:%S\'` を指定した場合とほとんど同じになります。ISO8601 形式ではミリ秒も指定しており、上の書式化文字列の結果にカンマで区切って追加されます。ISO8601 形式の時刻の例は 2003-01-23 00:29:50,411 です。

`class` エントリはオプションです。これはフォーマッタクラスの名前を (モジュール名とクラス名をドットでつないだもので) 指し示すものです。このオプションは `Formatter` の subclasses をインスタンス化するのに便利です。 `Formatter` の subclasses が、展開もしくは要約された形式の例外トレースバックを表示することができます。

注釈: `eval()` を使用していることで、上述のようにソケット経由で設定を送受信するために `listen()` を

使用していることに起因する潜在的なセキュリティリスクがあります。そのリスクは、相互に信頼できない多数のユーザが同じマシン上でコードを実行する場合に制限されています; 詳細は `listen()` ドキュメンテーションを参照してください。

参考:

`logging` モジュール `logging` モジュールの API リファレンス。

`logging.handlers` モジュール `logging` モジュールに含まれる、便利なハンドラです。

15.9 logging.handlers — ロギングハンドラ

Important

このページには、リファレンス情報だけが含まれています。チュートリアルは、以下のページを参照してください

- 基本チュートリアル
- 上級チュートリアル
- ロギングクックブック

Source code: [Lib/logging/handlers.py](#)

このパッケージでは、以下の便利なハンドラが提供されています。なお、これらのハンドラのうち、3 つ (`StreamHandler`, `FileHandler` および `NullHandler`) は、実際には `logging` モジュール自身で定義されていますが、他のハンドラと一緒にここでドキュメント化します。

15.9.1 StreamHandler

`logging` コアパッケージに含まれる `StreamHandler` クラスは、ログ出力を `sys.stdout`, `sys.stderr` あるいは何らかのファイル風 (file-like) オブジェクト (あるいは、より正確に言えば `write()` および `flush()` メソッドをサポートする何らかのオブジェクト) といったストリームに送信します。

class `logging.StreamHandler` (`stream=None`)

`StreamHandler` クラスの新たなインスタンスを返します。 `stream` が指定された場合、インスタンスはログ出力先として指定されたストリームを使います; そうでない場合、 `sys.stderr` が使われます。

emit (`record`)

フォーマッタが指定されていれば、フォーマッタを使ってレコードを書式化します。次に、レコードが改行とともにストリームに書き込まれます。例外情報が存在する場合、 `traceback.print_exception()` を使って書式化され、ストリームの末尾につけられます。

flush()

ストリームの *flush()* メソッドを呼び出してバッファをフラッシュします。 *close()* メソッドは *Handler* から継承しているため何も出力を行わないので、 *flush()* 呼び出しを明示的に行う必要があるかもしれません。

15.9.2 FileHandler

logging コアパッケージに含まれる *FileHandler* クラスは、ログ出力をディスク上のファイルに送信します。このクラスは出力機能を *StreamHandler* から継承しています。

class logging.FileHandler (*filename, mode='a', encoding=None, delay=False*)

FileHandler クラスの新たなインスタンスを返します。指定されたファイルが開かれ、ログ記録のためのストリームとして使われます。 *mode* が指定されなかった場合、 *'a'* が使われます。 *encoding* が *None* でない場合、その値はファイルを開くときのエンコーディングとして使われます。 *delay* が真ならば、ファイルを開くのは最初の *emit()* 呼び出しまで遅らせられます。デフォルトでは、ファイルは無制限に大きくなりつづけます。

バージョン 2.6 で変更: *delay* が追加されました。

close()

ファイルを閉じます。

emit (*record*)

record をファイルに出力します。

15.9.3 NullHandler

バージョン 2.7 で追加.

logging コアパッケージに含まれる *NullHandler* クラスは、いかなる書式化も出力も行いません。これは本質的には、ライブラリ開発者に使われる 'no-op' ハンドラです。

class logging.NullHandler

NullHandler クラスの新しいインスタンスを返します。

emit (*record*)

このメソッドは何もしません。

handle (*record*)

このメソッドは何もしません。

createLock ()

アクセスが特殊化される必要がある I/O が下がないので、このメソッドはロックに対して *None* を返します。

NullHandler の使い方の詳しい情報は、 *library-config* を参照してください。

15.9.4 WatchedFileHandler

バージョン 2.6 で追加.

`logging.handlers` モジュールに含まれる `WatchedFileHandler` クラスは、ログ記録先のファイルを監視する `FileHandler` の一種です。ファイルが変更された場合、ファイルを閉じてからファイル名を使って開き直します。

ファイルはログファイルをローテーションさせる `newsyslog` や `logrotate` のようなプログラムを使うことで変更されることがあります。このハンドラは、Unix/Linux で使われることを意図していますが、ファイルが最後にログを出力してから変わったかどうかを監視します。(ファイルはデバイスや inode が変わることによって変わったと判断します。) ファイルが変わったら古いファイルのストリームは閉じて、現在のファイルを新しいストリームを取得するために開きます。

このハンドラを Windows で使うことは適切ではありません。というのも Windows では開いているログファイルを移動したり削除したりできないからです - `logging` はファイルを排他的ロックを掛けて開きます - そのためこうしたハンドラは必要ないのです。さらに、Windows では `ST_INO` がサポートされていません; `stat()` はこの値として常に 0 を返します。

class `logging.handlers.WatchedFileHandler` (`filename`[, `mode`[, `encoding`[, `delay`]]])

`WatchedFileHandler` クラスの新たなインスタンスを返します。指定されたファイルが開かれ、ログ記録のためのストリームとして使われます。`mode` が指定されなかった場合、`'a'` が使われます。`encoding` が `None` でない場合、その値はファイルを開くときのエンコーディングとして使われます。`delay` が真ならば、ファイルを開くのは最初の `emit()` 呼び出しまで遅らせられます。デフォルトでは、ファイルは無制限に大きくなりつづけます。

emit (`record`)

レコードをファイルに出力しますが、その前にファイルが変更されていないかチェックします。もし変更されていれば、レコードをファイルに出力する前に、既存のストリームはフラッシュして閉じられ、ファイルが再度開かれます。

15.9.5 RotatingFileHandler

`logging.handlers` モジュールに含まれる `RotatingFileHandler` クラスは、ディスク上のログファイルに対するローテーション処理をサポートします。

class `logging.handlers.RotatingFileHandler` (`filename`, `mode`='a', `maxBytes`=0, `backupCount`=0, `encoding`=None, `delay`=0)

`RotatingFileHandler` クラスの新たなインスタンスを返します。指定されたファイルが開かれ、ログ記録のためのストリームとして使われます。`mode` が指定されなかった場合、`'a'` が使われます。`encoding` が `None` でない場合、その値はファイルを開くときのエンコーディングとして使われます。`delay` が真ならば、ファイルを開くのは最初の `emit()` 呼び出しまで遅らせられます。デフォルトでは、ファイルは無制限に大きくなりつづけます。

`maxBytes` および `backupCount` 値を指定することで、あらかじめ決められたサイズでファイルをロールオーバー (`rollover`) させることができます。指定サイズを超えそうになると、ファイルは閉じられ、暗黙のうちに新たなファイルが開かれます。ロールオーバーは現在のログファイルの長さが `maxBytes`

に近くなると常に起きます。 *maxBytes* または *backupCount* がゼロなら、ロールオーバーは起こりません。 *backupCount* が非ゼロの場合、システムは古いログファイルをファイル名に ".1", ".2" といった拡張子を追加して保存します。例えば、 *backupCount* が 5 で、基本のファイル名が `app.log` なら、 `app.log`, `app.log.1`, `app.log.2` ... と続き、 `app.log.5` までを得ることになります。ログの書き込み対象になるファイルは常に `app.log` です。このファイルが満杯になると、ファイルは閉じられ、 `app.log.1` に名前が変更されます。 `app.log.1`, `app.log.2` などが存在する場合、それらのファイルはそれぞれ `app.log.2`, `app.log.3` といった具合に名前が変更されます。

バージョン 2.6 で変更: *delay* が追加されました。

doRollover()

上述のような方法でロールオーバーを行います。

emit(record)

上述のようなロールオーバーを行いながら、レコードをファイルに出力します。

15.9.6 TimedRotatingFileHandler

`logging.handlers` モジュールに含まれる `TimedRotatingFileHandler` クラスは、特定の時間間隔でのログローテーションをサポートしています。

class `logging.handlers.TimedRotatingFileHandler` (*filename*, *when*='h', *interval*=1, *backupCount*=0, *encoding*=None, *delay*=False, *utc*=False)

`TimedRotatingFileHandler` クラスの新たなインスタンスを返します。 *filename* に指定したファイルを開き、ログ出力先のストリームとして使います。ログファイルのローテーション時には、ファイル名に拡張子 (suffix) をつけます。ログファイルのローテーションは *when* および *interval* の積に基づいて行います。

when は *interval* の単位を指定するために使います。使える値は下表の通りです。大小文字の区別は行いません。

値	<i>interval</i> の単位
'S'	秒
'M'	分
'H'	時間
'D'	日
'W0'-'W6'	曜日 (0=月曜)
'midnight'	深夜

曜日ベースのローテーションを使う場合は、月曜として 'W0' を、火曜として 'W1' を、...、日曜として 'W6' を指定します。このケースの場合は、 *interval* は使われません。

古いログファイルを保存する際にロギングシステムは拡張子を付けます。拡張子は日付と時間に基づいて、 `strftime` の `%Y-%m-%d_%H-%M-%S` 形式かその前方の一部を、ロールオーバー間隔に依存した形で使います。

最初に次のロールオーバー時間を計算するとき (ハンドラが生成されるとき)、次のローテーションがいつ起こるかを計算するために、既存のログファイルの最終変更時刻または現在の時間が使用されます。

`utc` 引数が `true` の場合時刻は UTC になり、それ以外では現地時間が使われます。

`backupCount` がゼロでない場合、保存されるファイル数は高々 `backupCount` 個で、それ以上のファイルがロールオーバーされる時に作られるならば、一番古いものが削除されます。削除のロジックは `interval` で決まるファイルを削除するので、`interval` を変えると古いファイルが残ったままになることもあります。

`delay` が `true` なら、ファイルを開くのは `emit()` の最初の呼び出しまで延期されます。

バージョン 2.6 で変更: `delay`, `utc` が追加されました。

doRollover()

上述のような方法でロールオーバーを行います。

emit(record)

上で説明した方法でロールオーバーを行いながら、レコードをファイルに出力します。

15.9.7 SocketHandler

`logging.handlers` モジュールに含まれる `SocketHandler` クラスは、ログ出力をネットワークソケットに送信します。基底クラスでは TCP ソケットを用います。

class logging.handlers.SocketHandler(host, port)

アドレスが `host` および `port` で与えられた遠隔のマシンと通信するようにした `SocketHandler` クラスのインスタンスを生成して返します。

close()

ソケットを閉じます。

emit()

レコードの属性辞書を pickle して、バイナリ形式でソケットに書き込みます。ソケット操作でエラーが生じた場合、暗黙のうちにパケットは捨てられます。事前に接続が失われていた場合、接続を再度確立します。受信端でレコードを unpickle して `LogRecord` にするには、`makeLogRecord()` 関数を使ってください。

handleError()

`emit()` の処理中に発生したエラーを処理します。よくある原因は接続の消失です。次のイベント発生時に再試行できるようにソケットを閉じます。

makeSocket()

サブクラスで必要なソケット形式を詳細に定義できるようにするためのファクトリメソッドです。デフォルトの実装では、TCP ソケット (`socket.SOCK_STREAM`) を生成します。

makePickle(record)

レコードの属性辞書を pickle してから先頭に長さ情報を付けてバイナリ形式にして、ソケットを介して送信できるようにして返します。

`pickle` が完全に安全というわけではないことに注意してください。セキュリティに関して心配なら、より安全なメカニズムを実装するためにこのメソッドをオーバーライドすると良いでしょう。例えば、HMAC を使って `pickle` に署名して、受け取る側ではそれを検証することができます。あるいはまた、受け取る側でグローバルなオブジェクトの `unpickle` を無効にすることができます。

send (packet)

`pickle` された文字列 *packet* をソケットに送信します。この関数はネットワークがビジーの時に発生する部分的送信に対応しています。

createSocket ()

ソケットの生成を試みます。失敗時には、指数的な減速アルゴリズムを使います。最初の失敗時には、ハンドラは送ろうとしていたメッセージを落とします。続くメッセージが同じインスタンスで扱われたとき、幾らかの時間が経過するまで接続を試みません。デフォルトのパラメタは、最初の遅延時間が 1 秒で、その遅延時間の後でそれでも接続が確保できないなら、遅延時間は 2 倍づつになり、最大で 30 秒になります。

この働きは、以下のハンドラ属性で制御されます：

- `retryStart` (最初の遅延時間、デフォルトは 1.0 秒)。
- `retryFactor` (乗数、デフォルトは 2.0)。
- `retryMax` (最大遅延時間、デフォルトは 30.0 秒)。

これは、リモートリスナがハンドラが使われた後に起動すると、(ハンドラは遅延が経過するまで接続を試みようとはせず、遅延時間中に黙ってメッセージを落とすだけなので) メッセージが失われてしまうこともあるということです。

15.9.8 DatagramHandler

`logging.handlers` モジュールに含まれる `DatagramHandler` クラスは、`SocketHandler` を継承しており、UDP ソケットを介したログ記録メッセージの送信をサポートしています。

class `logging.handlers.DatagramHandler (host, port)`

アドレスが *host* および *port* で与えられた遠隔のマシンと通信するようにした `DatagramHandler` クラスのインスタンスを生成して返します。

emit ()

レコードの属性辞書を `pickle` して、バイナリ形式でソケットに書き込みます。ソケット操作でエラーが生じた場合、暗黙のうちにパケットは捨てられます。事前に接続が失われていた場合、接続を再度確立します。受信端でレコードを `unpickle` して `LogRecord` にするには、`makeLogRecord()` 関数を使ってください。

makeSocket ()

ここで `SocketHandler` のファクトリメソッドをオーバーライドして、UDP ソケット (`socket.SOCK_DGRAM`) を生成しています。

send(*s*)

pickle された文字列をソケットに送信します。

15.9.9 SysLogHandler

`logging.handlers` モジュールに含まれる `SysLogHandler` クラスは、ログ記録メッセージを遠隔またはローカルの Unix syslog に送信する機能をサポートしています。

```
class logging.handlers.SysLogHandler (address=('localhost',
                                             LOG_UDP_PORT), facility=LOG_USER, sock-
                                             type=socket.SOCK_DGRAM)
```

遠隔の Unix マシンと通信するための、`SysLogHandler` クラスの新たなインスタンスを返します。マシンのアドレスは (*host*, *port*) のタプル形式をとる *address* で与えられます。 *address* が指定されない場合、 ('localhost', 514) が使われます。アドレスは UDP ソケットを使って開かれます。 (*host*, *port*) のタプル形式の代わりに文字列で "/dev/log" のように与えることもできます。この場合、Unix ドメインソケットが syslog にメッセージを送るのに使われます。 *facility* が指定されない場合、 LOG_USER が使われます。開かれるソケットの型は、 *socktype* 引数に依り、デフォルトは `socket.SOCK_DGRAM` で、UDP ソケットを開きます。(rsyslog のような新しい syslog デーモンと使うために) TCP ソケットを開くには、 `socket.SOCK_STREAM` の値を指定してください。

なお、あなたのサーバが UDP ポート 514 を聴取していないなら、 `SysLogHandler` が働かないことがあります。その場合は、あなたがドメインソケットに使うべきアドレスが何か調べてください - これはシステム依存です。例えば、Linux システムでは通常 '/dev/log' ですが、OS/X では '/var/run/syslog' です。あなたのプラットフォームを調べ、適切なアドレスを使うことが必要になります (あなたのアプリケーションがいくつかのプラットフォームで動く必要があるなら、これを実行時に確かめなければならないかもしれません)。Windows では、UDP オプションを使う必要性がかなり高いです。

バージョン 2.7 で変更: *socktype* が追加されました。

close()

遠隔ホストへのソケットを閉じます。

emit(*record*)

レコードは書式化された後、syslog サーバに送信されます。例外情報が存在しても、サーバには送信されません。

encodePriority(*facility*, *priority*)

ファシリティおよび優先度を整数に符号化します。値は文字列でも整数でも渡すことができます。文字列が渡された場合、内部の対応付け辞書が使われ、整数に変換されます。

シンボリックな LOG_ 値は `SysLogHandler` で定義されています。これは `sys/syslog.h` ヘッダーファイルで定義された値を反映しています。

優先度

名前 (文字列)	シンボル値
alert	LOG_ALERT
crit or critical	LOG_CRIT
debug	LOG_DEBUG
emerg or panic	LOG_EMERG
err or error	LOG_ERR
info	LOG_INFO
notice	LOG_NOTICE
warn or warning	LOG_WARNING

ファシリティ

名前 (文字列)	シンボル値
auth	LOG_AUTH
authpriv	LOG_AUTHPRIV
cron	LOG_CRON
daemon	LOG_DAEMON
ftp	LOG_FTP
kern	LOG_KERN
lpr	LOG_LPR
mail	LOG_MAIL
news	LOG_NEWS
syslog	LOG_SYSLOG
user	LOG_USER
uucp	LOG_UUCP
local0	LOG_LOCAL0
local1	LOG_LOCAL1
local2	LOG_LOCAL2
local3	LOG_LOCAL3
local4	LOG_LOCAL4
local5	LOG_LOCAL5
local6	LOG_LOCAL6
local7	LOG_LOCAL7

mapPriority (*levelname*)

ログレベル名を syslog 優先度名に対応付けます。カスタムレベルを使用している場合や、デフォルトアルゴリズムがニーズに適していない場合には、このメソッドをオーバーライドする必要があります。デフォルトアルゴリズムは、DEBUG, INFO, WARNING, ERROR, CRITICAL を等価な syslog 名に、他のすべてのレベル名を "warning" に対応付けます。

15.9.10 NTEventLogHandler

`logging.handlers` モジュールに含まれる `NTEventLogHandler` クラスは、ログ記録メッセージをローカルな Windows NT, Windows 2000, または Windows XP のイベントログに送信する機能をサポートします。この機能を使えるようにするには、Mark Hammond による Python 用 Win32 拡張パッケージをインストールする必要があります。

class `logging.handlers.NTEventLogHandler` (*appname*, *dllname=None*, *log-*
type='Application')

`NTEventLogHandler` クラスの新たなインスタンスを返します。 *appname* はイベントログに表示する際のアプリケーション名を定義するために使われます。この名前を使って適切なレジストリエントリが生成されます。 *dllname* はログに保存するメッセージ定義の入った .dll または .exe ファイルへの完全修飾パス名を与えなければなりません (指定されない場合、'win32service.pyd' が使われます - このライブラリは Win32 拡張とともにインストールされ、いくつかのプレースホルダとなるメッセージ定義を含んでいます)。これらのプレースホルダを利用すると、メッセージの発信源全体がログに記録されるため、イベントログは巨大になるので注意してください。 *logtype* は 'Application', 'System', 'Security' のいずれかで、デフォルトは 'Application' です。

close()

現時点では、イベントログエントリの発信源としてのアプリケーション名をレジストリから除去することはできません。しかしこれを行うと、イベントログビューアで意図した通りにログが見えなくなるでしょう - これはイベントログが .dll 名を取得するためにレジストリにアクセスできなければならないからです。現在のバージョンではこの操作を行いません。

emit (*record*)

メッセージ ID、イベントカテゴリ、イベント型を決定し、メッセージを NT イベントログに記録します。

getEventCategory (*record*)

レコードに対するイベントカテゴリを返します。自作のカテゴリを指定したい場合、このメソッドをオーバーライドしてください。このクラスのバージョンのメソッドは 0 を返します。

getEventType (*record*)

レコードのイベント型を返します。自作の型を指定したい場合、このメソッドをオーバーライドしてください。このクラスのバージョンのメソッドは、ハンドラの `typemap` 属性を使って対応付けを行います。この属性は `__init__()` で初期化され、DEBUG, INFO, WARNING, ERROR, CRITICAL が入っています。自作のレベルを使っているのなら、このメソッドをオーバーライドするか、ハンドラの `typemap` 属性に適切な辞書を配置する必要があるでしょう。

getMessageID (*record*)

レコードのメッセージ ID を返します。自作のメッセージを使っているのなら、ロガーに渡される `msg` を書式化文字列ではなく ID にします。その上で、辞書参照を行ってメッセージ ID を得ます。このクラスのバージョンでは 1 を返します。この値は `win32service.pyd` における基本メッセージ ID です。

15.9.11 SMTPHandler

`logging.handlers` モジュールに含まれる `SMTPHandler` クラスは、SMTP を介したログ記録メッセージの送信機能をサポートします。

class `logging.handlers.SMTPHandler` (*mailhost, fromaddr, toaddrs, subject, credentials=None, secure=None*)

新たな `SMTPHandler` クラスのインスタンスを返します。インスタンスは email の from および to アドレス行、および subject 行とともに初期化されます。 `toaddrs` は文字列からなるリストでなければなりません。非標準の SMTP ポートを指定するには、 `mailhost` 引数に (host, port) のタプル形式を指定します。文字列を使った場合、標準の SMTP ポートが使われます。もし SMTP サーバが認証を必要とするならば、 (username, password) のタプル形式を `credentials` 引数に指定することができます。

セキュアプロトコル (TLS) の使用を指定するには `secure` 引数にタプルを渡してください。これは認証情報が渡された場合のみ使用されます。タプルは、空のタプルか、キーファイルの名前を持つ 1 要素のタプルか、またはキーファイルと証明書ファイルの名前を持つ 2 要素のタプルのいずれかでなければなりません。(このタプルは `smtplib.SMTP.starttls()` メソッドに渡されます。)

バージョン 2.6 で変更: `credentials` が追加されました。

バージョン 2.7 で変更: `secure` が追加されました。

emit (*record*)

レコードを書式化し、指定されたアドレスに送信します。

getSubject (*record*)

レコードに応じたサブジェクト行を指定したいなら、このメソッドをオーバーライドしてください。

15.9.12 MemoryHandler

`logging.handlers` モジュールに含まれる `MemoryHandler` は、ログ記録するレコードをメモリ上にバッファリングし、定期的にその内容をターゲット (*target*) となるハンドラにフラッシュする機能をサポートしています。フラッシュ処理はバッファが一杯になるか、ある深刻度かそれ以上のレベルを持つイベントが観測された際に行われます。

`MemoryHandler` はより一般的な抽象クラス、 `BufferingHandler` のサブクラスです。この抽象クラスでは、ログ記録するレコードをメモリ上にバッファリングします。各レコードがバッファに追加される毎に、 `shouldFlush()` を呼び出してバッファをフラッシュすべきかどうか調べます。フラッシュする必要がある場合、 `flush()` がフラッシュ処理を行うものと想定されます。

class `logging.handlers.BufferingHandler` (*capacity*)

指定した許容量のバッファでハンドラを初期化します。

emit (*record*)

レコードをバッファに追加します。 `shouldFlush()` が true を返す場合、バッファを処理するために `flush()` を呼び出します。

flush ()

このメソッドをオーバーライドして、自作のフラッシュ動作を実装することができます。このクラスのバージョンのメソッドでは、単にバッファの内容を削除して空にします。

shouldFlush (*record*)

バッファが許容量に達している場合に `true` を返します。このメソッドは自作のフラッシュ処理方針を実装するためにオーバーライドすることができます。

class `logging.handlers.MemoryHandler` (*capacity*, *flushLevel=ERROR*, *target=None*)

`MemoryHandler` クラスの新たなインスタンスを返します。インスタンスはサイズ *capacity* のバッファとともに初期化されます。 *flushLevel* が指定されていない場合、 `ERROR` が使われます。 *target* が指定されていない場合、ハンドラが何らかの意味のある処理を行う前に `setTarget()` でターゲットを指定する必要があります。

close ()

`flush()` を呼び出し、ターゲットを `None` に設定してバッファを消去します。

flush ()

`MemoryHandler` の場合、フラッシュ処理は単に、バッファされたレコードをターゲットがあれば送信することを意味します。これと異なる動作を行いたい場合、オーバーライドしてください。

setTarget (*target*)

ターゲットハンドラをこのハンドラに設定します。

shouldFlush (*record*)

バッファが一杯になっているか、 *flushLevel* またはそれ以上のレコードでないかを調べます。

15.9.13 HTTPHandler

`logging.handlers` モジュールに含まれる `HTTPHandler` クラスは、ログ記録メッセージを GET または POST セマンティクスを使って Web サーバに送信する機能をサポートしています。

class `logging.handlers.HTTPHandler` (*host*, *url*, *method='GET'*)

`HTTPHandler` クラスの新たなインスタンスを返します。 *host* は特別なポートを使うことが必要な場合には、 `host:port` の形式で使うこともできます。

mapLogRecord (*record*)

URL エンコードされて Web サーバに送信することになる *record* に基く辞書を供給します。デフォルトの実装は単に `record.__dict__` を返却するだけです。例えば Web サーバに送信したいのは `LogRecord` のサブセットのみである、であるとか、あるいはもっと何か特別なカスタマイズが必要である、といった場合には、このメソッドをオーバーライド出来ます。

emit (*record*)

レコードを URL エンコードされた辞書形式で Web サーバに送信します。レコードを送信のために辞書に変換するために `mapLogRecord()` が呼び出されます。

注釈: Web server に送信するためのレコードを準備することは一般的な書式化操作とは同じではありませんので、 `setFormatter()` を使って `Formatter` を指定することは、 `HTTPHandler`

には効果はありません。 `format()` を呼び出す代わりに、このハンドラは `mapLogRecord()` を呼び出し、その後その返却辞書を Web server に送信するのに適した様式にエンコードするために `urllib.urlencode()` を呼び出します。

参考:

`logging` モジュール `logging` モジュールの API リファレンス。

`logging.config` モジュール `logging` モジュールの環境設定 API です。

15.10 `getpass` — 可搬性のあるパスワード入力機構

`getpass` モジュールは二つの関数を提供します:

`getpass.getpass([prompt[, stream]])`

エコーなしでユーザーにパスワードを入力させるプロンプト。ユーザーは `prompt` の文字列をプロンプトに使い、デフォルトは 'Password:' です。Unix ではプロンプトはファイルに似たオブジェクト `stream` へ出力されます。 `stream` のデフォルトは、制御端末 (`/dev/tty`) か、それが利用できない場合は `sys.stderr` です。(この引数は Windows では無視されます。)

もしエコー無しの入力を利用できない場合は、`getpass()` は `stream` に警告メッセージを出力し、`sys.stdin` から読み込み、`GetPassWarning` 警告を発生させます。

バージョン 2.5 で変更: パラメータ `stream` の追加。

バージョン 2.6 で変更: Unix ではデフォルトで、`sys.stdin` と `sys.stderr` へ fallback するまゝに `/dev/tty` を利用します。

注釈: IDLE から `getpass` を呼び出した場合、入力は IDLE のウィンドウではなく、IDLE を起動したターミナルから行われます。

exception `getpass.GetPassWarning`

`UserWarning` のサブクラスで、入力がエコーされてしまった場合に発生します。

`getpass.getuser()`

ユーザーの "ログイン名" を返します。

この関数は環境変数 `LOGNAME` `USER` `LNAME` `USERNAME` の順序でチェックして、最初の空ではない文字列が設定された値を返します。もし、なにも設定されていない場合は `pwd` モジュールが提供するシステム上のパスワードデータベースから返します。それ以外は、例外が上がります。

15.11 `curses` — 文字セル表示を扱うための端末操作

バージョン 1.6 で変更: `ncurses` ライブラリのサポートが追加され、パッケージになりました。

`curses` モジュールは、可搬性のある高度な端末操作のデファクトスタンダードである、`curses` ライブラリへのインタフェースを提供します。

`curses` が最も広く用いられているのは Unix 環境ですが、DOS、OS/2、そしておそらく他のシステムで利用できるバージョンもあります。この拡張モジュールは Linux および BSD 系の Unix で動作するオープンソースの `curses` ライブラリである `ncurses` の API に合致するように設計されています。

注釈: version 5.4 から、`ncurses` ライブラリは `nl_langinfo` 関数を利用して非 ASCII データをどう解釈するかを決定するようになりました。これは、アプリケーションは `locale.setlocale()` 関数を呼び出して、Unicode 文字列をシステムの利用可能なエンコーディングのどれかでエンコードする必要があることを意味します。この例では、システムのデフォルトエンコーディングを利用しています:

```
import locale
locale.setlocale(locale.LC_ALL, '')
code = locale.getpreferredencoding()
```

この後、`str.encode()` を呼び出すときに `code` を利用します。

参考:

`curses.ascii` モジュール ロケール設定に関わらず ASCII 文字を扱うためのユーティリティ。

`curses.panel` モジュール `curses` ウィンドウにデプス機能を追加するパネルスタック拡張。

`curses.textpad` モジュール **Emacs** ライクなキーバインディングをサポートする編集可能な `curses` 用テキストウィジェット。

`curses-howto` Andrew Kuchling および Eric Raymond によって書かれた、`curses` を Python で使うためのチュートリアルです。

Python のソースコードの中の `Demo/curses/` ディレクトリには、このモジュールが提供している `curses` バインディングを使った幾つかのサンプルプログラムがあります。

15.11.1 関数

`curses` モジュールでは以下の例外を定義しています:

exception `curses.error`

`curses` ライブラリ関数がエラーを返した際に送出される例外です。

注釈: 関数やメソッドにおけるオプションの引数 `x` および `y` がある場合、デフォルト値は常に現在のカーソルになります。オプションの `attr` がある場合、デフォルト値は `A_NORMAL` です。

`curses` では以下の関数を定義しています:

curses.baudrate()

端末の出力速度をビット/秒で返します。ソフトウェア端末エミュレータの場合、これは固定の高い値を持つことになります。この関数は歴史的な理由で入れられています; かつては、この関数は時間遅延を生成するための出力ループを書くために用いられたり、行速度に応じてインタフェースを切り替えたりするために用いられたりしていました。

curses.beep()

注意を促す短い音を鳴らします。

curses.can_change_color()

端末に表示される色をプログラマが変更できるか否かによって、True または False を返します。

curses.cbreak()

cbreak モードに入ります。cbreak モード ("rare" モードと呼ばれることもあります) では、通常の tty 行バッファリングはオフにされ、文字を一文字一文字読むことができます。ただし、raw モードとは異なり、特殊文字 (割り込み:interrupt、終了:quit、一時停止:suspend、およびフロー制御) については、tty ドライバおよび呼び出し側のプログラムに対する通常の効果をもっています。まず `raw()` を呼び出し、次いで `cbreak()` を呼び出すと、端末を cbreak モードにします。

curses.color_content(*color_number*)

色 *color_number* の赤、緑、および青 (RGB) 要素の強度を返します。*color_number* は 0 から COLORS の間でなければなりません。与えられた色の R、G、B、の値からなる三要素のタプルが返されます。この値は 0 (その成分はない) から 1000 (その成分の最大強度) の範囲をとります。

curses.color_pair(*color_number*)

指定された色の表示テキストにおける属性値を返します。属性値は A_STANDOUT、A_REVERSE、およびその他の A_* 属性と組み合わせられています。`pair_number()` はこの関数の逆です。

curses.curs_set(*visibility*)

カーソルの状態を設定します。*visibility* は 0、1、または 2 に設定され、それぞれ不可視、通常、または非常に可視、を意味します。要求された可視属性を端末がサポートしている場合、以前のカーソル状態が返されます; そうでなければ例外が送出されます。多くの端末では、"可視 (通常)" モードは下線カーソルで、"非常に可視" モードはブロックカーソルです。

curses.def_prog_mode()

現在の端末属性を、稼働中のプログラムが curses を使う際のモードである "プログラム" モードとして保存します。(このモードの反対は、プログラムが curses を使わない "シェル" モードです。) その後 `reset_prog_mode()` を呼ぶとこのモードを復旧します。

curses.def_shell_mode()

現在の端末属性を、稼働中のプログラムが curses を使っていないときのモードである "シェル" モードとして保存します。(このモードの反対は、プログラムが curses 機能を利用している "プログラム" モードです。) その後 `reset_shell_mode()` を呼ぶとこのモードを復旧します。

curses.delay_output(*ms*)

出力に *ms* ミリ秒の一時停止を入れます。

curses.doupdate()

物理スクリーンを更新します。curses ライブラリは、現在の物理スクリーンの内容と、次の状態として

要求されている仮想スクリーンをそれぞれ表す、2つのデータ構造を保持しています。`douupdate()` は更新を適用し、物理スクリーンを仮想スクリーンに一致させます。

仮想スクリーンは `addstr()` のような書き込み操作をウィンドウに行った後に `noutrefresh()` を呼び出して更新することができます。通常の `refresh()` 呼び出しは、単に `noutrefresh()` を呼んだ後に `douupdate()` を呼ぶだけです; 複数のウィンドウを更新しなければならない場合、すべてのウィンドウに対して `noutrefresh()` を呼び出した後、一度だけ `douupdate()` を呼ぶことで、パフォーマンスを向上させることができ、おそらくスクリーンのちらつきも押さえることができます。

`curses.echo()`

echo モードに入ります。echo モードでは、各文字入力はスクリーン上に入力された通りにエコーバックされます。

`curses.endwin()`

ライブラリの非初期化を行い、端末を通常の状態に戻します。

`curses.erasechar()`

ユーザの現在の消去文字 (erase character) を返します。Unix オペレーティングシステムでは、この値は `curses` プログラムが制御している端末の属性であり、`curses` ライブラリ自体では設定されません。

`curses.filter()`

`filter()` ルーチンを使う場合、`initscr()` を呼ぶ前に呼び出さなくてはなりません。この手順のもとらす効果は以下の通りです: まず二つの関数の呼び出しの間は、`LINES` は 1 に設定されます; `clear`、`cup`、`cud`、`cudl`、`cuul`、`cuu`、`vpa` は無効化されます; `home` 文字列は `cr` の値に設定されます。これにより、カーソルは現在の行に制限されるので、スクリーンの更新も同様に制限されます。この関数は、スクリーンの他の部分に影響を及ぼさずに文字単位の行編集を行う場合に利用できます。

`curses.flash()`

スクリーンを点滅します。すなわち、画面を色反転して、短時間でもとにもどします。人によっては、`beep()` で生成される注意音よりも、このような ”目に見えるベル” を好みます。

`curses.flushinp()`

すべての入力バッファをフラッシュします。この関数は、ユーザによってすでに入力されているが、まだプログラムによって処理されていないすべての先行入力文字を破棄します。

`curses.getmouse()`

`getch()` が `KEY_MOUSE` を返してマウスイベントを通知した後、この関数を呼んで待ち行列上に置かれているマウスイベントを取得しなければなりません。イベントは (`id`, `x`, `y`, `z`, `bstate`) の 5 要素のタプルで表現されています。`id` は複数のデバイスを区別するための ID 値で、`x`, `y`, `z` はイベントの座標値です。(現在 `z` は使われていません) `bstate` は整数値で、その各ビットはイベントのタイプを示す値に設定されています。この値は以下に示す定数のうち一つまたはそれ以上のビット単位 OR になっています。以下の定数の `n` は 1 から 4 のボタン番号を示します: `BUTTONn_PRESSED`, `BUTTONn_RELEASED`, `BUTTONn_CLICKED`, `BUTTONn_DOUBLE_CLICKED`, `BUTTONn_TRIPLE_CLICKED`, `BUTTON_SHIFT`, `BUTTON_CTRL`, `BUTTON_ALT`。

`curses.getsyx()`

仮想スクリーンにおける現在のカーソル位置を `y` および `x` の順で返します。`leaveok` が真に設定されていれば、`-1`, `-1` が返されます。

`curses.getwin(file)`

以前の `putwin()` 呼び出しでファイルに保存されている、ウィンドウ関連データを読み出します。次に、このルーチンはそのデータを使って新たなウィンドウを生成し初期化して、その新規ウィンドウオブジェクトを返します。

`curses.has_colors()`

端末が色表示を行える場合には `True` を返します。そうでない場合には `False` を返します。

`curses.has_ic()`

端末が文字の挿入/削除機能を持つ場合に `True` を返します。最近の端末エミュレータはどれもこの機能を持っており、この関数は歴史的な理由のためだけに存在しています。

`curses.has_il()`

端末が行の挿入/削除機能を持つ場合に `True` を返します。最近の端末エミュレータはどれもこの機能を持っていて、この関数は歴史的な理由のためだけに存在しています。

`curses.has_key(ch)`

キー値 `ch` をとり、現在の端末タイプがその値のキーを認識できる場合に `True` を返します。

`curses.halfdelay(tenths)`

半遅延モード、すなわち `cbreak` モードに似た、ユーザが打鍵した文字がすぐにプログラムで利用できるようになるモードで使われます。しかしながら、何も入力されなかった場合、十分の *tenths* 秒後に例外が送出されます。*tenths* の値は 1 から 255 の間でなければなりません。半遅延モードから抜けるには `nocbreak()` を使います。

`curses.init_color(color_number, r, g, b)`

色の定義を変更します。変更したい色番号と、その後に 3 つ組みの RGB 値 (赤、緑、青の成分の大きさ) をとります。*color_number* の値は 0 から `COLORS` の間でなければなりません。*r, g, b* の値は 0 から 1000 の間でなければなりません。`init_color()` を使うと、スクリーン上でカラーが使用されている部分はすべて新しい設定に即時変更されます。この関数はほとんどの端末で何も行いません;
`can_change_color()` が 1 を返す場合にのみ動作します。

`curses.init_pair(pair_number, fg, bg)`

色ペアの定義を変更します。3 つの引数: 変更したい色ペア、前景色の色番号、背景色の色番号、をとります。*pair_number* は 1 から `COLOR_PAIRS - 1` の間でなければなりません (0 色ペアは黒色背景に白色前景となるように設定されており、変更することができません)。*fg* および *bg* 引数は 0 と `COLORS` の間でなければなりません。色ペアが以前に初期化されていれば、スクリーンを更新して、指定された色ペアの部分新たな設定に変更します。

`curses.initscr()`

ライブラリを初期化します。スクリーン全体をあらわす `WindowObject` を返します。

注釈: 端末のオープン時にエラーが発生した場合、`curses` ライブラリによってインタープリタが終了される場合があります。

`curses.is_term_resized(nlines, ncols)`

`resize_term()` によってウィンドウ構造が変更されている場合に `True` を、そうでない場合は

`False` を返します。

`curses.isendwin()`

`endwin()` がすでに呼び出されている (すなわち、`curses` ライブラリが非初期化されている) 場合に `True` を返します。

`curses.keyname(k)`

`k` に番号付けされているキーの名前を返します。印字可能な ASCII 文字を生成するキーの名前はそのキーの文字自体になります。コントロールキーと組み合わせたキーの名前は、キャレットの後に対応する ASCII 文字が続く 2 文字の文字列になります。Alt キーと組み合わせたキー (128–255) の名前は、先頭に 'M-' が付き、その後に対応する ASCII 文字が続く文字列になります。

`curses.killchar()`

ユーザの現在の行削除文字を返します。Unix オペレーティングシステムでは、この値は `curses` プログラムが制御している端末の属性であり、`curses` ライブラリ自体では設定されません。

`curses.longname()`

現在の端末について記述している `terminfo` の長形式 `name` フィールドが入った文字列を返します。`verbose` 形式記述の最大長は 128 文字です。この値は `initscr()` 呼び出しの後でのみ定義されています。

`curses.meta(yes)`

`yes` が 1 の場合、8 ビット文字の入力を許可します。`yes` が 0 の場合、7 ビット文字だけを許可します。

`curses.mouseinterval(interval)`

ボタンが押されてから離されるまでの時間をマウスクリック一回として認識する最大の時間間隔をミリ秒で設定します。返り値は以前の内部設定値になります。デフォルトは 200 ミリ秒 (5 分の 1 秒) です。

`curses.mousemask(mousemask)`

報告すべきマウスイベントを設定し、(`availmask`, `oldmask`) の組からなるタプルを返します。`availmask` はどの指定されたマウスイベントのどれが報告されるかを示します; どのイベント指定も完全に失敗した場合には 0 が返ります。`oldmask` は与えられたウィンドウの以前のマウスイベントマスクです。この関数が呼ばれない限り、マウスイベントは何も報告されません。

`curses.napms(ms)`

`ms` ミリ秒間スリープします。

`curses.newpad(nlines, ncols)`

与えられた行とカラム数を持つパッド (`pad`) データ構造を生成し、そのポインタを返します。パッドはウィンドウオブジェクトとして返されます。

パッドはウィンドウと同じようなものですが、スクリーンのサイズによる制限を受けず、スクリーンの特定の部分に関連付けられていなくてもかまいません。大きなウィンドウが必要であり、スクリーンにはそのウィンドウの一部しか一度に表示しない場合に使えます。(スクロールや入力エコーなどによる) パッドに対する再描画は起こりません。パッドに対する `refresh()` および `noutrefresh()` メソッドは、パッド中の表示する部分と表示するために利用するスクリーン上の位置を指定する 6 つの引数が必要です。これらの引数は `pminrow`, `pmincol`, `sminrow`, `smincol`, `smaxrow`, `smaxcol` です; `p` で始まる引数はパッド中の表示領域の左上位置で、`s` で始まる引数はパッド領域を表示するスクリーン上のクリップ矩形を指定します。

`curses.newwin(nlines, ncols)`

`curses.newwin(nlines, ncols, begin_y, begin_x)`

左上の角が `(begin_y, begin_x)` で、高さ/幅が `nlines / ncols` の新規ウィンドウを返します。

デフォルトでは、ウィンドウは指定された位置からスクリーンの右下まで広がります。

`curses.nl()`

newline モードに入ります。このモードはリターンキーを入力中の改行として変換し、出力時に改行文字を復帰 (return) と改行 (line-feed) に変換します。newline モードは初期化時にはオンになっています。

`curses.nocbreak()`

cbreak モードを終了します。行バッファリングを行う通常の "cooked" モードに戻ります。

`curses.noecho()`

echo モードを終了します。入力のエコーバックはオフにされます。

`curses.nonl()`

newline モードを終了します。入力時のリターンキーから改行への変換、および出力時の改行から復帰/改行への低レベル変換を無効化します (ただし、`addch('\n')` の振る舞いは変更せず、仮想スクリーン上では常に復帰と改行に等しくなります)。変換をオフにすることで、`curses` は水平方向の動きを少しだけ高速化できることがあります; また、入力中のリターンキーの検出ができるようになります。

`curses.noqiflush()`

`noqiflush()` ルーチンを使うと、通常行われている INTR、QUIT、および SUSP 文字による入力および出力キューのフラッシュが行われなくなります。シグナルハンドラが終了した際、割り込みが発生しなかったかのように出力を続たい場合、ハンドラ中で `noqiflush()` を呼び出すことができます。

`curses.noraw()`

raw モードから離れます。行バッファリングを行う通常の "cooked" モードに戻ります。

`curses.pair_content(pair_number)`

要求された色ペアの色を含むタプル (fg, bg) を返します。`pair_number` は 1 から `COLOR_PAIRS - 1` の間でなければなりません。

`curses.pair_number(attr)`

`attr` に対する色ペアセットの番号を返します。`color_pair()` はこの関数の逆に相当します。

`curses.putp(string)`

`tputs(str, 1, putchar)` と等価です; 現在の端末における、指定された terminfo 機能の値を出力します。`putp()` の出力は常に標準出力に送られるので注意して下さい。

`curses.qiflush([flag])`

`flag` が False なら、`noqiflush()` を呼ぶのと同じ効果です。`flag` が True か、引数が与えられていない場合、制御文字が読み出された最にキューはフラッシュされます。

`curses.raw()`

raw モードに入ります。raw モードでは、通常の行バッファリングと割り込み (interrupt)、終了 (quit)、一時停止 (suspend)、およびフロー制御キーはオフになります; 文字は `curses` 入力関数に一字ずつ渡されます。

`curses.reset_prog_mode()`

端末を "program" モードに復旧し、あらかじめ `def_prog_mode()` で保存した内容に戻します。

`curses.reset_shell_mode()`

端末を "shell" モードに復旧し、あらかじめ `def_shell_mode()` で保存した内容に戻します。

`curses.resetty()`

端末モードの状態を最後に `savetty()` を呼び出した時の状態に戻します。

`curses.resize_term(nlines, ncols)`

`resizeterm()` で使用されるバックエンド関数で、ウィンドウサイズを変更する時、`resize_term()` は拡張された領域を埋めます。呼び出したアプリケーションはそれらの領域を適切なデータで埋めなくてはなりません。`resize_term()` 関数はすべてのウィンドウのサイズ変更を試みます。ただし、パッド呼び出しの慣例により、アプリケーションとの追加のやり取りを行わないサイズ変更は行えません。

`curses.resizeterm(nlines, ncols)`

現在の標準ウィンドウのサイズを指定された寸法に変更し、`curses` ライブラリが使用する、その他のウィンドウサイズを記憶しているデータ (特に SIGWINCH ハンドラ) を調整します。

`curses.savetty()`

`resetty()` で使用される、バッファ内の端末モードの現在の状態を保存します。

`curses.setsyx(y, x)`

仮想スクリーンのカーソルを y, x に設定します。 y および x がどちらも -1 の場合、`leaveok` が設定されます。

`curses.setupterm([termstr, fd])`

端末を初期化します。`termstr` は端末名となる文字列です。省略した場合、環境変数 `TERM` の値が使用されます。`fd` は送信される初期化シーケンスへのファイル記述子です。指定されない場合、`sys.stdout` のファイル記述子が使用されます。

`curses.start_color()`

プログラマがカラーを利用したい場合で、かつ他の何らかのカラー操作ルーチンを呼び出す前に呼び出さなくてはなりません。この関数は `initscr()` を呼んだ直後に呼ぶようにしておくといでしょう。

`start_color()` は 8 つの基本色 (黒、赤、緑、黄、青、マゼンタ、シアン、および白) と、色数の最大値と端末がサポートする色ペアの最大数が入っている、`curses` モジュールにおける二つのグローバル変数、`COLORS` および `COLOR_PAIRS` を初期化します。この関数はまた、色設定を端末のスイッチが入れられたときの状態に戻します。

`curses.termattrs()`

端末がサポートするすべてのビデオ属性を論理和した値を返します。この情報は、`curses` プログラムがスクリーンの見え方を完全に制御する必要がある場合に便利です。

`curses.termname()`

14 文字以下になるように切り詰められた環境変数 `TERM` の値を返します。

`curses.tigetflag(capname)`

terminfo 機能名 `capname` に対応する機能値を真偽値で返します。`capname` が真偽値で表せる機能値で

ない場合 `-1` を返し、機能がキャンセルされているか、端末記述上に見つからない場合 `0` を返します。

`curses.tigetnum(capname)`

terminfo 機能名 *capname* に対応する機能値を数値で返します。*capname* が数値で表せる機能値でない場合 `-2` を返し、機能がキャンセルされているか、端末記述上に見つからない場合 `-1` を返します。

`curses.tigetstr(capname)`

terminfo 機能名 *capname* に対応する機能値を文字列で返します。*capname* が文字列で表せる機能値でない場合、あるいは機能がキャンセルされているか、端末記述上に見つからない場合 `None` を返します。

`curses.tparm(str[, ...])`

str を与えられたパラメタを使って文字列にインスタンス化します。*str* は terminfo データベースから得られたパラメタを持つ文字列でなければなりません。例えば、`tparm(tigetstr("cup"), 5, 3)` は `'\033[6;4H'` のようになります。厳密には端末の形式によって異なる結果となります。

`curses.typeahead(fd)`

先読みチェックに使うためのファイル記述子 *fd* を指定します。*fd* が `-1` の場合、先読みチェックは行われません。

`curses` ライブラリはスクリーンを更新する間、先読み文字列を定期的に検索することで”行はみ出し最適化 (line-breakout optimization)”を行います。入力 that 得られ、かつ入力は端末からのものである場合、現在行おうとしている更新は `refresh` や `doupdate` を再度呼び出すまで先送りにします。この関数は異なるファイル記述子で先読みチェックを行うように指定することができます。

`curses.unctrl(ch)`

ch の印字可能な表現を文字列で返します。制御文字は例えば `^C` のようにキャレットに続く文字として表示されます。印字可能文字はそのままです。

`curses.ungetch(ch)`

ch をプッシュし、次に `getch()` を呼び出した時にその値が返るようにします。

注釈: `getch()` を呼び出すまでは *ch* は一つしかプッシュできません。

`curses.ungetmouse(id, x, y, z, bstate)`

与えられた状態データが関連付けられた `KEY_MOUSE` イベントを入力キューにプッシュします。

`curses.use_env(flag)`

この関数を使う場合、`initscr()` または `newterm` を呼ぶ前に呼び出さなくてはなりません。*flag* が `False` の場合、環境変数 `LINES` および `COLUMNS` の値 (デフォルトで使用されます) の値が設定されていたり、`curses` がウィンドウ内で動作して (この場合 `LINES` や `COLUMNS` が設定されていないとウィンドウのサイズを使います) いても、terminfo データベースに指定された `lines` および `columns` の値を使います。

`curses.use_default_colors()`

この機能をサポートしている端末上で、色の値としてデフォルト値を使う設定をします。あなたのアプリケーションで透過性とサポートするためにこの関数を使ってください。デフォルトの色は色番号-1

に割り当てられます。この関数を呼んだ後、たとえば `init_pair(x, curses.COLOR_RED, -1)` は色ペア x を赤い前景色とデフォルトの背景色に初期化します。

`curses.wrapper(func, ...)`

`curses` を初期化し、呼び出し可能なオブジェクト `func` (その他の `curses` を使用するアプリケーション) を呼び出します。アプリケーションが例外を送出した場合、この関数は端末を例外を再送出する前に正常な状態に戻し、トレースバックを作成します。その後、呼び出し可能なオブジェクト `func` には、第一引数としてメインウィンドウ `'stdscr'` が、続いて `wrapper()` に渡されたその他の引数が渡されます。`func` を呼び出す前、`wrapper()` は `cbreak` モードをオンに、エコーをオフに、端末キーボードを有効にし、端末が色表示をサポートしている場合は色を初期化します。終了時 (通常終了、例外による終了のどちらでも)、`cooked` モードに戻し、エコーをオンにし、端末キーボードを無効にします。

15.11.2 Window オブジェクト

上記の `initscr()` や `newwin()` 関数が返す window オブジェクトは以下のメソッドを持っています:

`window.addch(ch[, attr])`

`window.addch(y, x, ch[, attr])`

注釈: ここでの文字は Python における文字 (長さ 1 の文字列) ではなく C における文字 (ASCII コード) を意味します (この注釈は文字について触れているドキュメントではどこでも当てはまります)。組み込みの `ord()` は文字列をコードの集まりにする際に便利です。

(y , x) にある文字 `ch` を属性 `attr` で描画します。このときその場所に以前描画された文字は上書きされます。デフォルトでは、文字の位置および属性はウィンドウオブジェクトにおける現在の設定になります。

注釈: Writing outside the window, subwindow, or pad raises a `curses.error`. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the character is printed.

`window.addnstr(str, n[, attr])`

`window.addnstr(y, x, str, n[, attr])`

文字列 `str` から最大で n 文字を (y , x) に属性 `attr` で描画します。以前ディスプレイにあった内容はすべて上書きされます。

`window.addstr(str[, attr])`

`window.addstr(y, x, str[, attr])`

(y , x) に文字列 `str` を属性 `attr` で描画します。以前ディスプレイにあった内容はすべて上書きされます。

注釈: Writing outside the window, subwindow, or pad raises `curses.error`. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the string is printed.

`window.attroff(attr)`

現在のウィンドウに書き込まれたすべての内容に対し "バックグラウンド" に設定された属性 `attr` を除去します。

`window.attron(attr)`

現在のウィンドウに書き込まれたすべての内容に対し "バックグラウンド" に属性 `attr` を追加します。

`window.attrset(attr)`

"バックグラウンド" の属性セットを `attr` に設定します。初期値は 0 (属性なし) です。

`window.bkgd(ch[, attr])`

ウィンドウ上の背景プロパティを、`attr` を属性とする文字 `ch` に設定します。変更はそのウィンドウ中のすべての文字に以下のようにして適用されます:

- ウィンドウ中のすべての文字の属性が新たな背景属性に変更されます。
- 以前の背景文字が出現すると、常に新たな背景文字に変更されます。

`window.bkgdset(ch[, attr])`

ウィンドウの背景を設定します。ウィンドウの背景は、文字と何らかの属性の組み合わせから成り立ちます。背景情報の属性の部分は、ウィンドウ上に描画されている空白でないすべての文字と組み合わせられ (OR され) ます。空白文字には文字部分と属性部分の両方が組み合わせられます。背景は文字のプロパティとなり、スクロールや行/文字の挿入/削除操作の際には文字と一緒に移動します。

`window.border([ls[, rs[, ts[, bs[, tl[, tr[, bl[, br]]]]]]]))`

ウィンドウの縁に境界線を描画します。各引数には境界の特定部分を表現するために使われる文字を指定します; 詳細は以下のテーブルを参照してください。文字は整数または 1 文字からなる文字列で指定されます。

注釈: どの引数も、0 を指定した場合デフォルトの文字が使われるようになります。キーワード引数は使うことができません。デフォルトはテーブル内で示しています:

引数	説明	デフォルト値
<i>ls</i>	左側	ACS_VLINE
<i>rs</i>	右側	ACS_VLINE
<i>ts</i>	上側	ACS_HLINE
<i>bs</i>	下側	ACS_HLINE
<i>tl</i>	左上の角	ACS_ULCORNER
<i>tr</i>	右上の角	ACS_URCORNER
<i>bl</i>	左下の角	ACS_LLCORNER
<i>br</i>	右下の角	ACS_LRCORNER

`window.box([vertch, horch])`

`border()` と同様ですが、*ls* および *rs* は共に *vertch* で、*ts* および *bs* は共に *horch* です。この関数では、角に使われるデフォルト文字が常に使用されます。

`window.chgat(attr)`

`window.chgat(num, attr)`

`window.chgat(y, x, attr)`

`window.chgat(y, x, num, attr)`

Set the attributes of *num* characters at the current cursor position, or at position (*y*, *x*) if supplied. If *num* is not given or is `-1`, the attribute will be set on all the characters to the end of the line. This function moves cursor to position (*y*, *x*) if supplied. The changed line will be touched using the `touchline()` method so that the contents will be redisplayed by the next window refresh.

`window.clear()`

`erase()` に似ていますが、次に `refresh()` が呼び出された際にすべてのウィンドウを再描画するようにします。

`window.clearok(yes)`

yes が 1 ならば、次の `refresh()` はウィンドウを完全に消去します。

`window.clrtoobot()`

カーソルの位置からウィンドウの端までを消去します: カーソル以降のすべての行が削除されるため、`clrtoeol()` と等価です。

`window.clrtoeol()`

カーソル位置から行末までを消去します。

`window.cursyncup()`

ウィンドウのすべての親ウィンドウについて、現在のカーソル位置を反映するように更新します。

`window.delch([y, x])`

(*y*, *x*) にある文字を削除します。

`window.deleteln()`

カーソルの下にある行を削除します。後続の行はすべて 1 行上に移動します。

`window.derwin(begin_y, begin_x)`

`window.derwin (nlines, ncols, begin_y, begin_x)`

”derive window (ウィンドウを派生する)” の短縮形です。 `derwin()` は `subwin()` と同じですが、 `begin_y` および `begin_x` はスクリーン全体の原点ではなく、ウィンドウの原点からの相対位置です。派生したウィンドウオブジェクトが返されます。

`window.echochar (ch[, attr])`

文字 `ch` に属性 `attr` を付与し、即座に `refresh()` をウィンドウに対して呼び出します。

`window.enclose (y, x)`

与えられた文字セル座標をスクリーン原点から相対的なものとし、ウィンドウの中に含まれるかを調べて、True または False を返します。スクリーン上のウィンドウの一部がマウスイベントの発生場所を含むかどうかを調べる上で便利です。

`window.erase()`

ウィンドウをクリアします。

`window.getbegyx()`

左上の角の座標をあらわすタプル (`y`, `x`) を返します。

`window.getbkgd()`

与えられたウィンドウの現在の背景文字と属性のペアを返します。

`window.getch ([y, x])`

文字を 1 個取得します。返される整数は ASCII の範囲の値となる わけではないので注意してください。ファンクションキー、キーパッドのキー等は 256 よりも大きな数字を返します。無遅延 (no-delay) モードでは、入力がない場合 -1 が返されます。それ以外の場合は、 `getch()` はキー入力を待ちます。

`window.getkey ([y, x])`

文字を 1 個取得し、 `getch()` が返すような整数ではなく文字列を返します。ファンクションキー、キーパッドのキー、およびその他特殊キーはキー名を含む複数文字を返します。無遅延モードでは、入力がない場合例外を送出します。

`window.getmaxyx()`

ウィンドウの高さおよび幅を表すタプル (`y`, `x`) を返します。

`window.getparyx()`

親ウィンドウ中におけるウィンドウの開始位置を `x` と `y` の二つの整数で返します。ウィンドウに親ウィンドウがない場合 -1, -1 を返します。

`window.getstr ([y, x])`

原始的な文字編集機能つきで、ユーザの入力文字列を読み取ります。

`window.getyx()`

ウィンドウの左上角からの相対で表した現在のカーソル位置をタプル (`y`, `x`) で返します。

`window.hline (ch, n)`

`window.hline (y, x, ch, n)`

(`y`, `x`) から始まり、`n` の長さを持つ、文字 `ch` で作られる水平線を表示します。

`window.idcok(flag)`

`flag` が `False` の場合、`curses` は端末のハードウェアによる文字挿入/削除機能を使おうとしなくなります; `flag` が `True` ならば、文字挿入/削除は有効にされます。 `curses` が最初に初期化された際には文字挿入/削除はデフォルトで有効になっています。

`window.idlok(yes)`

`yes` が 1 であれば、`curses` はハードウェアの行編集機能の利用を試みます。行挿入/削除は無効化されます。

`window.immedok(flag)`

`flag` が `True` ならば、ウィンドウイメージ内における何らかの変更があるとウィンドウを更新するようになります; すなわち、`refresh()` を自分で呼ばなくても良くなります。とはいえ、`wrefresh` を繰り返し呼び出すことになるため、この操作はかなりパフォーマンスを低下させます。デフォルトでは無効になっています。

`window.inch([y,x])`

ウィンドウの指定の位置の文字を返します。下位 8 ビットが本来の文字で、それより上のビットは属性です。

`window.insch(ch[,attr])`

`window.insch(y,x,ch[,attr])`

(`y`, `x`) に文字 `ch` を属性 `attr` で描画し、行の `x` からの内容を 1 文字分右にずらします。

`window.insdelln(nlines)`

`nlines` 行を指定されたウィンドウの現在の行の上に挿入します。その下にある `nlines` 行は失われます。負の `nlines` を指定すると、カーソルのある行以降の `nlines` を削除し、削除された行の後ろに続く内容が上に来ます。その下にある `nlines` は消去されます。現在のカーソル位置はそのままです。

`window.insertln()`

カーソルの下に空行を 1 行入れます。それ以降の行は 1 行づつ下に移動します。

`window.insnstr(str,n[,attr])`

`window.insnstr(y,x,str,n[,attr])`

文字列をカーソルの下にある文字の前に (一行に収まるだけ) 最大 `n` 文字挿入します。 `n` がゼロまたは負の値の場合、文字列全体が挿入されます。カーソルの右にあるすべての文字は右に移動し、行の左端にある文字は失われます。カーソル位置は (`y`, `x` が指定されていた場合はそこに移動しますが、その後は) 変化しません。

`window.insstr(str[,attr])`

`window.insstr(y,x,str[,attr])`

キャラクタ文字列を (行に収まるだけ) カーソルより前に挿入します。カーソルの右側にある文字はすべて右にシフトし、行の右端の文字は失われます。カーソル位置は (`y`, `x` が指定されていた場合はそこに移動しますが、その後は) 変化しません。

`window.instr([n])`

`window.instr(y,x[,n])`

現在のカーソル位置、または `y`, `x` が指定されている場合にはその場所から始まるキャラクタ文字列をウィンドウから抽出して返します。属性は文字から剥ぎ取られます。 `n` が指定された場合、`instr()`

は (末尾の NUL 文字を除いて) 最大で n 文字までの長さからなる文字列を返します。

`window.is_linetouched(line)`

指定した行が、最後に `refresh()` を呼んだ時から変更されている場合に `True` を返します; そうでない場合には `False` を返します。 `line` が現在のウィンドウ上の有効な行でない場合、`curses.error` 例外を送出します。

`window.is_wintouched()`

指定したウィンドウが、最後に `refresh()` を呼んだ時から変更されている場合に `True` を返します; そうでない場合には `False` を返します。

`window.keypad(yes)`

`yes` が 1 の場合、ある種のキー (キーパッドやファンクションキー) によって生成されたエスケープシーケンスは `curses` で解釈されます。 `yes` が 0 の場合、エスケープシーケンスは入力ストリームにそのままの状態に残されます。

`window.leaveok(yes)`

`yes` が 1 の場合、カーソルは "カーソル位置" に移動せず現在の場所にとどめます。これにより、カーソルの移動を減らせる可能性があります。この場合、カーソルは不可視にされます。

`yes` が 0 の場合、カーソルは更新の際に常に "カーソル位置" に移動します。

`window.move(new_y, new_x)`

カーソルを (`new_y`, `new_x`) に移動します。

`window.mvderwin(y, x)`

ウィンドウを親ウィンドウの中で移動します。ウィンドウのスクリーン相対となるパラメタ群は変化しません。このルーチンは親ウィンドウの一部をスクリーン上の同じ物理位置に表示する際に用いられます。

`window.mvwin(new_y, new_x)`

ウィンドウの左上角が (`new_y`, `new_x`) になるように移動します。

`window.nodelay(yes)`

`yes` が 1 の場合、`getch()` は非ブロックで動作します。

`window.notimeout(yes)`

`yes` が 1 の場合、エスケープシーケンスはタイムアウトしなくなります。

`yes` が 0 の場合、数ミリ秒の間エスケープシーケンスは解釈されず、入力ストリーム中にそのままの状態に残されます。

`window.noutrefresh()`

更新をマークはしますが待機します。この関数はウィンドウのデータ構造を表現したい内容を反映するように更新しますが、物理スクリーン上に反映させるための強制更新を行いません。更新を行うためには `doupdate()` を呼び出します。

`window.overlay(destwin [, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol])`

ウィンドウを `destwin` の上に重ね書き (overlay) します。ウィンドウは同じサイズである必要はなく、重

なっている領域だけが複写されます。この複写は非破壊的です。これは現在の背景文字が *destwin* の内容を上書きしないことを意味します。

複写領域をきめ細かく制御するために、*overlay()* の第二形式を使うことができます。 *sminrow* および *smincol* は元のウィンドウの左上の座標で、他の変数は *destwin* 内の矩形を表します。

`window.overwrite(destwin[, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol])`

destwin の上にウィンドウの内容を上書き (overwrite) します。ウィンドウは同じサイズである必要はなく、重なっている領域だけが複写されます。この複写は破壊的です。これは現在の背景文字が *destwin* の内容を上書きすることを意味します。

複写領域をきめ細かく制御するために、*overwrite()* の第二形式を使うことができます。 *sminrow* および *smincol* は元のウィンドウの左上の座標で、他の変数は *destwin* 内の矩形を表します。

`window.putwin(file)`

ウィンドウに関連付けられているすべてのデータを与えられたファイルオブジェクトに書き込みます。この情報は後に *getwin()* 関数を使って取得することができます。

`window.redrawln(beg, num)`

beg 行から始まる *num* スクリーン行の表示内容が壊れており、次の *refresh()* 呼び出しで完全に再描画されなければならないことを通知します。

`window.redrawwin()`

ウィンドウ全体を更新 (touch) し、次の *refresh()* 呼び出しで完全に再描画されるようにします。

`window.refresh([pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol])`

ディスプレイを即時更新し (現実のウィンドウとこれまでの描画/削除メソッドの内容との同期をとり) ます。

6 つのオプション引数はウィンドウが *newpad()* で生成された場合にのみ指定することができます。追加の引数はパッドやスクリーンのどの部分が含まれるのかを示すために必要です。 *pminrow* および *pmincol* にはパッドが表示されている矩形の左上角を指定します。 *sminrow*, *smincol*, *smaxrow*, および *smaxcol* には、スクリーン上に表示される矩形の縁を指定します。パッド内に表示される矩形の右下角はスクリーン座標から計算されるので、矩形は同じサイズでなければなりません。矩形は両方とも、それぞれのウィンドウ構造内に完全に含まれていなければなりません。 *pminrow*, *pmincol*, *sminrow*, または *smincol* に負の値を指定すると、ゼロを指定したものとして扱われます。

`window.resize(nlines, ncols)`

curses ウィンドウの記憶域を、指定値のサイズに調整するため再割当てします。サイズが現在の値より大きい場合、ウィンドウのデータは現在の背景設定 (*bkgdset()* で設定) で埋められマージされます。

`window.scroll([lines=1])`

スクリーンまたはスクロール領域を上 *lines* 行スクロールします。

`window.scrollok(flag)`

ウィンドウのカーソルが、最下行で改行を行ったり最後の文字を入力したりした結果、ウィンドウやスクロール領域の縁からはみ出して移動した際の動作を制御します。 *flag* が偽の場合、カーソルは最下行にそのままにしておかれます。 *flag* が真の場合、ウィンドウは 1 行上にスクロールします。端末の物理スクロール効果を得るためには *idlok()* も呼び出す必要があるので注意してください。

`window.setscrreg(top, bottom)`

スクロール領域を *top* から *bottom* に設定します。スクロール動作はすべてこの領域で行われます。

`window.standend()`

A_STANDOUT 属性をオフにします。端末によっては、この操作ですべての属性をオフにする副作用が発生します。

`window.standout()`

A_STANDOUT 属性をオンにします。

`window.subpad(begin_y, begin_x)`

`window.subpad(nlines, ncols, begin_y, begin_x)`

左上の角が (*begin_y*, *begin_x*) にあり、幅/高さがそれぞれ *ncols* / *nlines* であるようなサブウィンドウを返します。

`window.subwin(begin_y, begin_x)`

`window.subwin(nlines, ncols, begin_y, begin_x)`

左上の角が (*begin_y*, *begin_x*) にあり、幅/高さがそれぞれ *ncols* / *nlines* であるようなサブウィンドウを返します。

デフォルトでは、サブウィンドウは指定された場所からウィンドウの右下角まで広がります。

`window.syncdown()`

このウィンドウの上位のウィンドウのいずれかで更新 (*touch*) された各場所をこのウィンドウ内でも更新します。このルーチンは *refresh()* から呼び出されるので、手動で呼び出す必要はほとんどないはずです。

`window.syncok(flag)`

flag を *True* にして呼び出すと、ウィンドウが変更された際は常に *syncup()* を自動的に呼ぶようになります。

`window.syncup()`

ウィンドウ内で更新 (*touch*) した場所を、上位のすべてのウィンドウ内でも更新します。

`window.timeout(delay)`

ウィンドウのブロックまたは非ブロック読み込み動作を設定します。*delay* が負の場合、ブロック読み出しが使われ、入力を無期限で待ち受けます。*delay* がゼロの場合、非ブロック読み出しが使われ、入力待ちの文字がない場合 *getch()* は -1 を返します。*delay* が正の値であれば、*getch()* は *delay* ミリ秒間ブロックし、ブロック後の時点で入力がない場合には -1 を返します。

`window.touchline(start, count[, changed])`

start から始まる *count* 行が変更されたかのように振舞わせます。もし *changed* が与えられた場合、その引数は指定された行が変更された (*changed*=1) か、変更されていないか (*changed*=0) を指定します。

`window.touchwin()`

描画を最適化するために、すべてのウィンドウが変更されたかのように振舞わせます。

`window.untouchwin()`

ウィンドウ内のすべての行を、最後に *refresh()* を呼んだ際から変更されていないものとしてマークします。

`window.vline(ch, n)`

`window.vline(y, x, ch, n)`

`(y, x)` から始まり、`n` の長さを持つ、文字 `ch` で作られる垂直線を表示します。

15.11.3 定数

`curses` モジュールでは以下のデータメンバを定義しています:

`curses.ERR`

`getch()` のような整数を返す `curses` ルーチンのいくつかは、失敗した際に `ERR` を返します。

`curses.OK`

`napms()` のような整数を返す `curses` ルーチンのいくつかは、成功した際に `OK` を返します。

`curses.version`

モジュールの現在のバージョンを表現する文字列です。 `__version__` でも取得できます。

Some constants are available to specify character cell attributes. The exact constants available are system dependent.

属性	意味
<code>A.ALTCCHARSET</code>	Alternate character set mode
<code>A.BLINK</code>	Blink mode
<code>A.BOLD</code>	Bold mode
<code>A.DIM</code>	Dim mode
<code>A.INVIS</code>	Invisible or blank mode
<code>A.NORMAL</code>	Normal attribute
<code>A.PROTECT</code>	Protected mode
<code>A.REVERSE</code>	Reverse background and foreground colors
<code>A.STANDOUT</code>	Standout mode
<code>A.UNDERLINE</code>	Underline mode
<code>A.HORIZONTAL</code>	Horizontal highlight
<code>A.LEFT</code>	Left highlight
<code>A.LOW</code>	Low highlight
<code>A.RIGHT</code>	Right highlight
<code>A.TOP</code>	Top highlight
<code>A.VERTICAL</code>	Vertical highlight
<code>A.CHARTEXT</code>	Bit-mask to extract a character

Several constants are available to extract corresponding attributes returned by some methods.

Bit-mask	意味
A_ATTRIBUTES	Bit-mask to extract attributes
A_CHARTEXT	Bit-mask to extract a character
A_COLOR	Bit-mask to extract color-pair field information

キーは `KEY_` で始まる名前をもつ整数定数です。利用可能なキーキャップはシステムに依存します。

キー定数	キー
<code>KEY_MIN</code>	最小のキー値
<code>KEY_BREAK</code>	ブレイクキー (Break, 信頼できません)
<code>KEY_DOWN</code>	下矢印
<code>KEY_UP</code>	上矢印
<code>KEY_LEFT</code>	左矢印
<code>KEY_RIGHT</code>	右矢印
<code>KEY_HOME</code>	ホームキー (Home, または上左矢印)
<code>KEY_BACKSPACE</code>	バックスペース (Backspace, 信頼できません)
<code>KEY_F0</code>	ファンクションキー。64 個までサポートされています。
<code>KEY_Fn</code>	ファンクションキー <i>n</i> の値
<code>KEY_DL</code>	行削除 (Delete line)
<code>KEY_IL</code>	行挿入 (Insert line)
<code>KEY_DC</code>	文字削除 (Delete char)
<code>KEY_IC</code>	文字挿入、または文字挿入モードへ入る
<code>KEY_EIC</code>	文字挿入モードから抜ける
<code>KEY_CLEAR</code>	画面消去
<code>KEY_EOS</code>	画面の末端まで消去
<code>KEY_EOL</code>	行末端まで消去
<code>KEY_SF</code>	前に 1 行スクロール
<code>KEY_SR</code>	後ろ (逆方向) に 1 行スクロール
<code>KEY_NPAGE</code>	次のページ (Page Next)
<code>KEY_PPAGE</code>	前のページ (Page Prev)
<code>KEY_STAB</code>	タブ設定
<code>KEY_CTAB</code>	タブリセット
<code>KEY_CATAB</code>	すべてのタブをリセット
<code>KEY_ENTER</code>	入力または送信 (信頼できません)
<code>KEY_SRESET</code>	ソフトウェア (部分的) リセット (信頼できません)
<code>KEY_RESET</code>	リセットまたはハードリセット (信頼できません)
<code>KEY_PRINT</code>	印刷 (Print)
<code>KEY_LL</code>	下ホーム (Home down) または最下行 (左下)
<code>KEY_A1</code>	キーパッドの左上キー
<code>KEY_A3</code>	キーパッドの右上キー
<code>KEY_B2</code>	キーパッドの中央キー

次のページに続く

表 1 – 前のページからの続き

キー定数	キー
KEY_C1	キーパッドの左下キー
KEY_C3	キーパッドの右下キー
KEY_BTAB	Back tab
KEY_BEG	開始 (Beg)
KEY_CANCEL	キャンセル (Cancel)
KEY_CLOSE	閉じる (Close)
KEY_COMMAND	コマンド (Cmd)
KEY_COPY	コピー (Copy)
KEY_CREATE	生成 (Create)
KEY_END	終了 (End)
KEY_EXIT	終了 (Exit)
KEY_FIND	検索 (Find)
KEY_HELP	ヘルプ (Help)
KEY_MARK	マーク (Mark)
KEY_MESSAGE	メッセージ (Message)
KEY_MOVE	移動 (Move)
KEY_NEXT	次へ (Next)
KEY_OPEN	開く (Open)
KEY_OPTIONS	オプション (Options)
KEY_PREVIOUS	前へ (Prev)
KEY_REDO	やり直し (Redo)
KEY_REFERENCE	参照 (Ref)
KEY_REFRESH	更新 (Refresh)
KEY_REPLACE	置換 (Replace)
KEY_RESTART	再起動 (Restart)
KEY_RESUME	再開 (Resume)
KEY_SAVE	保存 (Save)
KEY_SBEG	シフト付き Beg
KEY_SCANCEL	シフト付き Cancel
KEY_SCOMMAND	シフト付き Command
KEY_SCOPY	シフト付き Copy
KEY_SCREATE	シフト付き Create
KEY_SDC	シフト付き Delete char
KEY_SDL	シフト付き Delete line
KEY_SELECT	選択 (Select)
KEY_SEND	シフト付き End
KEY_SEOL	シフト付き Clear line
KEY_SEXIT	シフト付き Exit
KEY_SFIND	シフト付き Find
KEY_SHELP	シフト付き Help

次のページに続く

表 1 – 前のページからの続き

キー定数	キー
KEY_SHOME	シフト付き Home
KEY_SIC	シフト付き Input
KEY_SLEFT	シフト付き Left arrow
KEY_SMESSAGE	シフト付き Message
KEY_SMOVE	シフト付き Move
KEY_SNEXT	シフト付き Next
KEY_SOPTIONS	シフト付き Options
KEY_SPREVIOUS	シフト付き Prev
KEY_SPRINT	シフト付き Print
KEY_SREDO	シフト付き Redo
KEY_SREPLACE	シフト付き Replace
KEY_SRIGHT	シフト付き Right arrow
KEY_SRSUME	シフト付き Resume
KEY_SSAVE	シフト付き Save
KEY_SSUSPEND	シフト付き Suspend
KEY_SUNDO	シフト付き Undo
KEY_SUSPEND	一時停止 (Suspend)
KEY_UNDO	元に戻す (Undo)
KEY_MOUSE	マウスイベント通知
KEY_RESIZE	端末リサイズイベント
KEY_MAX	最大キー値

VT100 や、X 端末エミュレータのようなソフトウェアエミュレーションでは、通常少なくとも 4 つのファンクションキー (KEY_F1, KEY_F2, KEY_F3, KEY_F4) が利用可能で、矢印キーは KEY_UP, KEY_DOWN, KEY_LEFT および KEY_RIGHT が対応付けられています。計算機に PC キーボードが付属している場合、矢印キーと 12 個のファンクションキー (古い PC キーボードには 10 個しかファンクションキーがないかもしれません) が利用できると考えてよいでしょう; また、以下のキーパッド対応付けは標準的なものです:

キーキャップ	定数
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END
Page Up	KEY_PPAGE
Page Down	KEY_NPAGE

代替文字セットを以下の表に列挙します。これらは VT100 端末から継承したものであり、X 端末のようなソフトウェアエミュレーション上で一般に利用可能なものです。グラフィックが利用できない場合、curses は印字可能 ASCII 文字による粗雑な近似出力を行います。

注釈: これらは `initscr()` が呼び出された後でしか利用できません。

ACS コード	意味
ACS_BBSS	右上角の別名
ACS_BLOCK	黒四角ブロック
ACS_BOARD	白四角ブロック
ACS_BSBS	水平線の別名
ACS_BSSB	左上角の別名
ACS_BSSS	上向き T 字罫線の別名
ACS_BTEE	下向き T 字罫線
ACS_BULLET	黒丸 (bullet)
ACS_CKBOARD	チェッカーボードボタン (点描)
ACS_DARROW	下向き矢印
ACS_DEGREE	度記号
ACS_DIAMOND	ダイヤモンド
ACS_GEQUAL	大なりイコール
ACS_HLINE	水平線
ACS_LANTERN	ランタン (lantern) シンボル
ACS_LARROW	左向き矢印
ACS_LEQUAL	小なりイコール
ACS_LLCORNER	左下角
ACS_LRCORNER	右下角
ACS_LTEE	左向き T 字罫線
ACS_NEQUAL	不等号
ACS_PI	パイ記号
ACS_PLMINUS	プラスマイナス記号
ACS_PLUS	大プラス記号
ACS_RARROW	右向き矢印
ACS_RTEE	右向き T 字罫線
ACS_S1	スキャンライン 1
ACS_S3	スキャンライン 3
ACS_S7	スキャンライン 7
ACS_S9	スキャンライン 9
ACS_SBBS	右下角の別名
ACS_SBSB	垂直線の別名
ACS_SBSS	右向き T 字罫線の別名
ACS_SSBB	左下角の別名
ACS_SSBS	下向き T 字罫線の別名
ACS_SSSB	左向き T 字罫線の別名

次のページに続く

表 2 – 前のページからの続き

ACS コード	意味
ACS_SSSS	交差罫線または大プラス記号の別名
ACS_STERLING	ポンドスターリング記号
ACS_TTEE	上向き T 字罫線
ACS_UARROW	上向き矢印
ACS_ULCORNER	左上角
ACS_URCORNER	右上角
ACS_VLINE	垂直線

以下のテーブルは定義済みの色を列挙したものです:

定数	色
COLOR_BLACK	黒
COLOR_BLUE	青
COLOR_CYAN	シアン (薄く緑がかった青)
COLOR_GREEN	緑
COLOR_MAGENTA	マゼンタ (紫がかった赤)
COLOR_RED	赤
COLOR_WHITE	白
COLOR_YELLOW	黄色

15.12 `curses.textpad` — `curses` プログラムのためのテキスト入力ウィジェット

バージョン 1.6 で追加.

`curses.textpad` モジュールでは、`curses` ウィンドウ内での基本的なテキスト編集を処理し、Emacs に似た (すなわち Netscape Navigator, BBedit 6.x, FrameMaker, その他諸々のプログラムとも似た) キーバインドをサポートしている `Textbox` クラスを提供します。このモジュールではまた、テキストボックスを枠で囲むなどの目的のために有用な、矩形描画関数を提供しています。

`curses.textpad` モジュールでは以下の関数を定義しています:

`curses.textpad.rectangle` (*win, uly, ulx, lry, lrx*)

矩形を描画します。最初の引数はウィンドウオブジェクトでなければなりません; 残りの引数はそのウィンドウからの相対座標になります。2 番目および 3 番目の引数は描画すべき矩形の左上角の *y* および *x* 座標です; 4 番目および 5 番目の引数は右下角の *y* および *x* 座標です。矩形は、VT100/IBM PC におけるフォーム文字を利用できる端末 (`xterm` やその他のほとんどのソフトウェア端末エミュレータを含む) ではそれを使って描画されます。そうでなければ ASCII 文字のダッシュ、垂直バー、およびプラス記号で描画されます。

15.12.1 Textbox オブジェクト

以下のような `Textbox` オブジェクトをインスタンス生成することができます:

```
class curses.textpad.Textbox(win)
```

テキストボックスウィジェットオブジェクトを返します。`win` 引数は、テキストボックスを入れるための `WindowObject` でなければなりません。テキストボックスの編集カーソルは、最初はテキストボックスが入っているウィンドウの左上角に配置され、その座標は (0, 0) です。インスタンスの `stripspaces` フラグの初期値はオンに設定されます。

`Textbox` オブジェクトは以下のメソッドを持ちます:

```
edit ([validator])
```

普段使うことになるエントリポイントです。終了キーストロークの一つが入力されるまで編集キーストロークを受け付けます。`validator` を与える場合、関数でなければなりません。`validator` はキーストロークが入力されるたびにそのキーストロークが引数となって呼び出されます; 返された値に対して、コマンドキーストロークとして解釈が行われます。このメソッドはウィンドウの内容を文字列として返します; ウィンドウ内の空白が含まれるかどうかは `stripspaces` 属性で決められます。

```
do_command (ch)
```

単一のコマンドキーストロークを処理します。以下にサポートされている特殊キーストロークを示します:

キーストローク	動作
Control-A	ウィンドウの左端に移動します。
Control-B	カーソルを左へ移動し、必要なら前の行に折り返します。
Control-D	カーソル下の文字を削除します。
Control-E	右端 (<code>stripspaces</code> がオフのとき) または行末 (<code>stripspaces</code> がオンのとき) に移動します。
Control-F	カーソルを右に移動し、必要なら次の行に折り返します。
Control-G	ウィンドウを終了し、その内容を返します。
Control-H	逆方向に文字を削除します。
Control-J	ウィンドウが 1 行であれば終了し、そうでなければ新しい行を挿入します。
Control-K	行が空白行ならその行全体を削除し、そうでなければカーソル以降行末までを消去します。
Control-L	スクリーンを更新します。
Control-N	カーソルを下に移動します; 1 行下に移動します。
Control-O	カーソルの場所に空行を 1 行挿入します。
Control-P	カーソルを上移動します; 1 行上に移動します。

移動操作は、カーソルがウィンドウの縁にあって移動ができない場合には何も行いません。場合によっては、以下のような同義のキーストロークがサポートされています:

定数	キーストローク
KEY_LEFT	Control-B
KEY_RIGHT	Control-F
KEY_UP	Control-P
KEY_DOWN	Control-N
KEY_BACKSPACE	Control-h

他のキーストロークは、与えられた文字を挿入し、(行折り返し付きで) 右に移動するコマンドとして扱われます。

`gather()`

ウィンドウの内容を文字列として返します; ウィンドウ内の空白が含まれるかどうかは `stripspaces` メンバ変数で決められます。

`stripspaces`

この属性はウィンドウ内の空白領域の解釈方法を制御するためのフラグです。フラグがオンに設定されている場合、各行の末端にある空白領域は無視されます; すなわち、末端空白領域にカーソルが入ると、その場所の代わりに行の末尾にカーソルが移動します。また、末端の空白領域はウィンドウの内容を取得する際に剥ぎ取られます。

15.13 `curses.ascii` — ASCII 文字に関するユーティリティ

バージョン 1.6 で追加.

`curses.ascii` モジュールでは、ASCII 文字を指す名前定数と、様々な ASCII 文字区分についてある文字が帰属するかどうかを調べる関数を提供します。このモジュールで提供されている定数は以下の制御文字の名前です:

名前	意味
NUL	
SOH	ヘディング開始、コンソール割り込み
STX	テキスト開始
ETX	テキスト終了
EOT	テキスト伝送終了
ENQ	問い合わせ、ACK フロー制御時に使用
ACK	肯定応答
BEL	ベル
BS	一文字後退
TAB	タブ
HT	TAB の別名: "水平タブ"
LF	改行
NL	LF の別名: "改行"

次のページに続く

表 3 – 前のページからの続き

名前	意味
VT	垂直タブ
FF	改頁
CR	復帰
SO	シフトアウト、他の文字セットの開始
SI	シフトイン、標準の文字セットに復帰
DLE	データリンクでのエスケープ
DC1	装置制御 1、フロー制御のための XON
DC2	装置制御 2、ブロックモードフロー制御
DC3	装置制御 3、フロー制御のための XOFF
DC4	装置制御 4
NAK	否定応答
SYN	同期信号
ETB	ブロック転送終了
CAN	キャンセル (Cancel)
EM	媒体終端
SUB	代入文字
ESC	エスケープ文字
FS	ファイル区切り文字
GS	グループ区切り文字
RS	レコード区切り文字、ブロックモード終了子
US	単位区切り文字
SP	空白文字
DEL	削除

これらの大部分は、最近では実際に定数の意味通りに使われることがほとんどないので注意してください。これらのニーモニック符号はデジタル計算機より前のテレプリンタにおける慣習から付けられたものです。

このモジュールでは、標準 C ライブラリの関数を雛型とする以下の関数をサポートしています：

```
curses.ascii.isalnum(c)
```

ASCII 英数文字かどうかを調べます; `isalpha(c)` or `isdigit(c)` と等価です。

```
curses.ascii.isalpha(c)
```

ASCII アルファベット文字かどうかを調べます; `isupper(c)` or `islower(c)` と等価です。

```
curses.ascii.isascii(c)
```

文字が 7 ビット ASCII 文字に合致するかどうかを調べます。

```
curses.ascii.isblank(c)
```

ASCII 余白文字、すなわち空白または水平タブかどうかを調べます。

```
curses.ascii.iscntrl(c)
```

ASCII 制御文字 (0x00 から 0x1f の範囲または 0x7f) かどうかを調べます。

`curses.ascii.isdigit(c)`

ASCII 10 進数字、すなわち '0' から '9' までの文字かどうかを調べます。c in string.digits と等価です。

`curses.ascii.isgraph(c)`

空白以外の ASCII 印字可能文字かどうかを調べます。

`curses.ascii.islower(c)`

ASCII 小文字かどうかを調べます。

`curses.ascii.isprint(c)`

空白文字を含め、ASCII 印字可能文字かどうかを調べます。

`curses.ascii.ispunct(c)`

空白または英数字以外の ASCII 印字可能文字かどうかを調べます。

`curses.ascii.isspace(c)`

ASCII 余白文字、すなわち空白、改行、復帰、改頁、水平タブ、垂直タブかどうかを調べます。

`curses.ascii.isupper(c)`

ASCII 大文字かどうかを調べます。

`curses.ascii.isxdigit(c)`

ASCII 16 進数字かどうかを調べます。c in string.hexdigits と等価です。

`curses.ascii.isctrl(c)`

ASCII 制御文字 (0 から 31 までの値) かどうかを調べます。

`curses.ascii.ismeta(c)`

非 ASCII 文字 (0x80 またはそれ以上の値) かどうかを調べます。

これらの関数は数字も文字列も使えます; 引数を文字列にした場合、組み込み関数 `ord()` を使って変換されます。

これらの関数は全て、関数に渡した文字列の最初の文字から得られたビット値を調べるので注意してください; 関数はホスト計算機で使われている文字列エンコーディングについて何ら関知しません。文字列エンコーディングについて関知する (そして国際化に関するプロパティを正しく扱う) 関数については、モジュール `string` を参照してください。

以下の 2 つの関数は、引数として 1 文字の文字列または整数で表したバイト値のどちらでもとり得ます; これらの関数は引数と同じ型で値を返します。

`curses.ascii.ascii(c)`

ASCII 値を返します。c の下位 7 ビットに対応します。

`curses.ascii.ctrl(c)`

与えた文字に対応する制御文字を返します (0x1f とビット単位で論理積を取ります)。

`curses.ascii.alt(c)`

与えた文字に対応する 8 ビット文字を返します (0x80 とビット単位で論理和を取ります)。

以下の関数は 1 文字からなる文字列値または整数値を引数に取り、文字列を返します。

`curses.ascii.unctrl(c)`

ASCII 文字 *c* の文字列表現を返します。もし *c* が印字可能文字であれば、返される文字列は *c* そのものになります。もし *c* が制御文字 (0x00–0x1f) であれば、キャレット ('^') と、その後ろに続く *c* に対応した大文字からなる文字列になります。*c* が ASCII 削除文字 (0x7f) であれば、文字列は '^?' になります。*c* のメタビット (0x80) がセットされていれば、メタビットは取り去られ、前述のルールが適用され、'!' が前につけられます。

`curses.ascii.controlnames`

0 (NUL) から 0x1f (US) までの 32 の ASCII 制御文字と、空白文字 SP のニーモニック符号名からなる 33 要素の文字列によるシーケンスです。

15.14 `curses.panel` — `curses` のためのパネルスタック拡張

パネルは深さ (depth) の機能が追加されたウィンドウです。これにより、ウィンドウをお互いに重ね合わせることができ、各ウィンドウの可視部分だけが表示されます。パネルはスタック中に追加したり、スタック内で上下移動させたり、スタックから除去することができます。

15.14.1 関数

`curses.panel` では以下の関数を定義しています:

`curses.panel.bottom_panel()`

パネルスタックの最下層のパネルを返します。

`curses.panel.new_panel(win)`

与えられたウィンドウ *win* に関連付けられたパネルオブジェクトを返します。返されたパネルオブジェクトを参照しておく必要があることに注意してください。もし参照しなければ、パネルオブジェクトはガベージコレクションされてパネルスタックから削除されます。

`curses.panel.top_panel()`

パネルスタックの最上層のパネルを返します。

`curses.panel.update_panels()`

仮想スクリーンをパネルスタック変更後の状態に更新します。この関数では `curses.doupdate()` を呼ばないので、ユーザは自分で呼び出す必要があります。

15.14.2 Panel オブジェクト

上記の `new_panel()` が返す Panel オブジェクトはスタック順の概念を持つウィンドウです。ウィンドウはパネルに関連付けられており、表示する内容を決定している一方、パネルのメソッドはパネルスタック中のウィンドウの深さ管理を担います。

Panel オブジェクトは以下のメソッドを持っています:

`Panel.above()`

現在のパネルの上にあるパネルを返します。

`Panel.below()`

現在のパネルの下にあるパネルを返します。

`Panel.bottom()`

パネルをスタックの最下層にプッシュします。

`Panel.hidden()`

パネルが隠れている (不可視である) 場合に真を返し、そうでない場合偽を返します。

`Panel.hide()`

パネルを隠します。この操作ではオブジェクトは消去されず、スクリーン上のウィンドウを不可視にするだけです。

`Panel.move(y, x)`

パネルをスクリーン座標 (*y*, *x*) に移動します。

`Panel.replace(win)`

パネルに関連付けられたウィンドウを *win* に変更します。

`Panel.set_userptr(obj)`

パネルのユーザポインタを *obj* に設定します。このメソッドは任意のデータをパネルに関連付けるために使われ、任意の Python オブジェクトにすることができます。

`Panel.show()`

(隠れているはずの) パネルを表示します。

`Panel.top()`

パネルをスタックの最上層にプッシュします。

`Panel.userptr()`

パネルのユーザポインタを返します。任意の Python オブジェクトです。

`Panel.window()`

パネルに関連付けられているウィンドウオブジェクトを返します。

15.15 platform — 実行中プラットフォームの固有情報を参照する

バージョン 2.3 で追加.

ソースコード: [Lib/platform.py](#)

注釈: プラットフォーム毎にアルファベット順に並べています。Linux については Unix セクションを参照してください。

15.15.1 クロスプラットフォーム

`platform.architecture(executable=sys.executable, bits="", linkage="")`

`executable` で指定した実行可能ファイル (省略時は Python インタープリタのバイナリ) の各種アーキテクチャ情報を調べます。

戻り値はタプル (`bits`, `linkage`) で、アーキテクチャのビット数と実行可能ファイルのリンク形式を示します。どちらの値も文字列で返ります。

値を決定できない場合はパラメータプリセットから与えられる値を返します。`bits` に `''` を与えた場合、サポートされているポインタサイズを知るために `sizeof(pointer)` (Python バージョン < 1.5.2 では `sizeof(long)`) が使用されます。

この関数は、システムの `file` コマンドを使用します。`file` はほとんどの Unix プラットフォームと一部の非 Unix プラットフォームで利用可能ですが、`file` コマンドが利用できず、かつ `executable` が Python インタープリタでない場合には適切なデフォルト値が返ります。

注釈: Mac OS X (とひょっとすると他のプラットフォーム) では、実行可能ファイルは複数のアーキテクチャを含んだユニバーサル形式かもしれません。

現在のインタプリタが "64-bit" であるかどうかを調べるには、`sys.maxsize` の方が信頼できます:

```
is_64bits = sys.maxsize > 2**32
```

`platform.machine()`

`'i386'` のような、機種を返します。不明な場合は空文字列を返します。

`platform.node()`

コンピュータのネットワーク名を返します。ネットワーク名は完全修飾名とは限りません。不明な場合は空文字列を返します。

`platform.platform(aliased=0, terse=0)`

実行中プラットフォームを識別する文字列を返します。この文字列には、有益な情報をできるだけ多く付加しています。

戻り値は機械で処理しやすい形式ではなく、人間にとって読みやすい形式となっています。異なったプラットフォームでは異なった戻り値となるようになっています。

`aliased` が真なら、システムの名称として一般的な名称ではなく、別名を使用して結果を返します。たとえば、SunOS は Solaris となります。この機能は `system.alias()` で実装されています。

`terse` が真なら、プラットフォームを特定するために最低限必要な情報だけを返します。

`platform.processor()`

`'amd64'` のような、(現実の) プロセッサ名を返します。

不明な場合は空文字列を返します。NetBSD のようにこの情報を提供しない、または `machine()` と同じ値しか返さないプラットフォームも多く存在しますので、注意してください。

`platform.python_build()`

Python のビルド番号と日付を、(buildno, builddate) のタプルで返します。

`platform.python_compiler()`

Python をコンパイルする際に使用したコンパイラを示す文字列を返します。

`platform.python_branch()`

Python 実装のバージョン管理システム上のブランチを特定する文字列を返します。

バージョン 2.6 で追加。

`platform.python_implementation()`

Python 実装を指定する文字列を返します。戻り値は: 'CPython', 'IronPython', 'Jython', 'PyPy' のいずれかです。

バージョン 2.6 で追加。

`platform.python_revision()`

Python 実装のバージョン管理システム上のリビジョンを特定する文字列を返します。

バージョン 2.6 で追加。

`platform.python_version()`

Python のバージョンを、'major.minor.patchlevel' 形式の文字列で返します。

`sys.version` と異なり、patchlevel (デフォルトでは 0) も必ず含まれています。

`platform.python_version_tuple()`

Python のバージョンを、文字列のタプル (major, minor, patchlevel) で返します。

`sys.version` と異なり、patchlevel (デフォルトでは '0') も必ず含まれています。

`platform.release()`

'2.2.0' や 'NT' のような、システムのリリース情報を返します。不明な場合は空文字列を返します。

`platform.system()`

'Linux', 'Windows', 'Java' のような、システム/OS 名を返します。不明な場合は空文字列を返します。

`platform.system_alias(system, release, version)`

マーケティング目的で使われる一般的な別名に変換して (system, release, version) を返します。混乱を避けるために、情報を並べなおす場合があります。

`platform.version()`

'#3 on degas' のような、システムのリリース情報を返します。不明な場合は空文字列を返します。

`platform.uname()`

非常に可搬性の高い uname インターフェイスで、実行中プラットフォームを示す情報を、文字列のタプル (system, node, release, version, machine, processor) で返します。

`os.uname()` と異なり、複数のプロセッサ名が候補としてタプルに追加される場合があります。

不明な項目は '' となります。

15.15.2 Java プラットフォーム

`platform.java_ver(release="", vendor="", vminfo=("", "", ""), osinfo=("", "", ""))`

Jython 用のバージョンインターフェースです。

タプル (release, vendor, vminfo, osinfo) を返します。vminfo はタプル (vm_name, vm_release, vm_vendor)、osinfo はタプル (os_name, os_version, os_arch) です。不明な項目は引数で指定した値 (デフォルトは '') となります。

15.15.3 Windows プラットフォーム

`platform.win32_ver(release="", version="", csd="", ptype="")`

Windows レジストリから追加のバージョン情報を取得して、タプル (release, version, csd, ptype) を返します。それぞれ、OS リリース、バージョン番号、CSD レベル (サービスパック)、OS タイプ (マルチ/シングルプロセッサ) を指しています。

参考: ptype はシングルプロセッサの NT 上では 'Uniprocessor Free'、マルチプロセッサでは 'Multiprocessor Free' となります。'Free' がついていない場合はデバッグ用のコードが含まれていないことを示し、'Checked' がついていれば引数や範囲のチェックなどのデバッグ用コードが含まれていることを示します。

注釈: この関数は、Mark Hammond の win32all がインストールされた環境で良く動作しますが、Python 2.3 以上なら一応動作します。(Python 2.6 からサポートされました) もちろん、この関数が使えるのは Win32 互換プラットフォームのみです。

Win95/98 固有

`platform.popen(cmd, mode='r', bufsize=None)`

可搬性の高い `popen()` インターフェースで、可能なら `win32pipe.popen()` を使用します。`win32pipe.popen()` は Windows NT では利用可能ですが、Windows 9x ではハングしてしまいます。

15.15.4 Mac OS プラットフォーム

`platform.mac_ver(release="", versioninfo=("", "", ""), machine="")`

Mac OS のバージョン情報を、タプル (release, versioninfo, machine) で返します。versioninfo は、タプル (version, dev_stage, non_release_version) です。

不明な項目は '' となります。タプルの要素は全て文字列です。

15.15.5 Unix プラットフォーム

```
platform.dist (distname="", version="", id="", supported_dists=('SuSE', 'debian', 'redhat', 'mandrake', ...))
```

この関数は、現在 `linux_distribution()` が提供している機能の古いバージョンです。新しいコードを書くときは、`linux_distribution()` を利用してください。

`linux_distribution()` との違いは、`dist()` は常に `supported_dists` パラメータから取ったディストリビューションの短い名前を返す事です。

バージョン 2.6 で非推奨.

```
platform.linux_distribution (distname="", version="", id="", supported_dists=('SuSE', 'debian', 'redhat', 'mandrake', ...), full_distribution_name=1)
```

OS ディストリビューション名の取得を試みます。

`supported_dists` は、検索する Linux ディストリビューションを定義するために利用します。デフォルトでは、リリースファイル名で定義されている、現在サポートされている Linux ディストリビューションのリストです。

`full_distribution_name` が `True` (デフォルト) の場合、OS から読み込まれた完全なディストリビューション名が返されます。それ以外の場合、`supported_dists` で利用された短い名前が返されます。

戻り値はタプル (`distname`, `version`, `id`) で、不明な項目は引数で指定した値となります。`id` はバージョン番号に続く括弧内の要素です。これは通常バージョンコードネームです。

注釈: This function is deprecated since Python 3.5 and removed in Python 3.8. See alternative like the `distro` package.

バージョン 2.6 で追加.

```
platform.libc_ver (executable=sys.executable, lib="", version="", chunksize=2048)
```

`executable` で指定したファイル (省略時は Python インタープリタ) がリンクしている `libc` バージョンの取得を試みます。戻り値は文字列のタプル (`lib`, `version`) で、不明な項目は引数で指定した値となります。

この関数は、実行形式に追加されるシンボルの細かな違いによって、`libc` のバージョンを特定します。この違いは `gcc` でコンパイルされた実行可能ファイルでのみ有効だと思われます。

`chunksize` にはファイルから情報を取得するために読み込むバイト数を指定します。

15.16 errno — 標準の errno システムシンボル

このモジュールから標準の `errno` システムシンボルを取得することができます。個々のシンボルの値は `errno` に対応する整数値です。これらのシンボルの名前は、`linux/include/errno.h` から借用されており、かなり網羅的ではありません。

errno.errorcode

errno 値を背後のシステムにおける文字列表現に対応付ける辞書です。例えば、`errno.errorcode[errno.EPERM]` は 'EPERM' に対応付けられます。

数値のエラーコードをエラーメッセージに変換するには、`os.strerror()` を使ってください。

以下のリストの内、現在のプラットフォームで使われていないシンボルはモジュール上で定義されていません。定義されているシンボルだけを挙げたリストは `errno.errorcode.keys()` として取得することができます。取得できるシンボルには以下のようなものがあります：

errno.EPERM

許可されていない操作です (Operation not permitted)

errno.ENOENT

ファイルまたはディレクトリがありません (No such file or directory)

errno.ESRCH

指定したプロセスが存在しません (No such process)

errno.EINTR

割り込みシステムコールです (Interrupted system call)

errno.EIO

I/O エラーです (I/O error)

errno.ENXIO

そのようなデバイスまたはアドレスはありません (No such device or address)

errno.E2BIG

引数リストが長すぎます (Arg list too long)

errno.ENOEXEC

実行形式にエラーがあります (Exec format error)

errno.EBADF

ファイル番号が間違っています (Bad file number)

errno.ECHILD

子プロセスがありません (No child processes)

errno.EAGAIN

再試行してください (Try again)

errno.ENOMEM

空きメモリがありません (Out of memory)

errno.EACCES

許可がありません (Permission denied)

errno.EFAULT

不正なアドレスです (Bad address)

`errno.ENOTBLK`

ブロックデバイスが必要です (Block device required)

`errno.EBUSY`

そのデバイスまたはリソースは使用中です (Device or resource busy)

`errno.EEXIST`

ファイルがすでに存在します (File exists)

`errno.EXDEV`

デバイスをまたいだリンクです (Cross-device link)

`errno.ENODEV`

そのようなデバイスはありません (No such device)

`errno.ENOTDIR`

ディレクトリではありません (Not a directory)

`errno.EISDIR`

ディレクトリです (Is a directory)

`errno.EINVAL`

無効な引数です (Invalid argument)

`errno.ENFILE`

ファイルテーブルがオーバーフローしています (File table overflow)

`errno.EMFILE`

開かれたファイルが多すぎます (Too many open files)

`errno.ENOTTY`

タイプライタではありません (Not a typewriter)

`errno.ETXTBSY`

テキストファイルが使用中です (Text file busy)

`errno.EFBIG`

ファイルが大きすぎます (File too large)

`errno.ENOSPC`

デバイス上に空きがありません (No space left on device)

`errno.ESPIPE`

不正なシークです (Illegal seek)

`errno.EROFS`

リードオンリーのファイルシステムです (Read-only file system)

`errno.EMLINK`

リンクが多すぎます (Too many links)

`errno.EPIPE`

壊れたパイプです (Broken pipe)

`errno.EDOM`

数学引数が関数の定義域を越えています (Math argument out of domain of func)

`errno.ERANGE`

表現できない数学演算結果になりました (Math result not representable)

`errno.EDEADLK`

リソースのデッドロックが起きます (Resource deadlock would occur)

`errno.ENAMETOOLONG`

ファイル名が長すぎます (File name too long)

`errno.ENOLCK`

レコードロックが利用できません (No record locks available)

`errno.ENOSYS`

実装されていない機能です (Function not implemented)

`errno.ENOTEMPTY`

ディレクトリが空ではありません (Directory not empty)

`errno.ELOOP`

これ以上シンボリックリンクを追跡できません (Too many symbolic links encountered)

`errno.EWOULDBLOCK`

操作がブロックします (Operation would block)

`errno.ENOMSG`

指定された型のメッセージはありません (No message of desired type)

`errno.EIDRM`

識別子が除去されました (Identifier removed)

`errno.ECHRNG`

チャネル番号が範囲を超えました (Channel number out of range)

`errno.EL2NSYNC`

レベル 2 で同期がとれていません (Level 2 not synchronized)

`errno.EL3HLT`

レベル 3 で終了しました (Level 3 halted)

`errno.EL3RST`

レベル 3 でリセットしました (Level 3 reset)

`errno.ELNRNG`

リンク番号が範囲を超えています (Link number out of range)

`errno.EUNATCH`

プロトコルドライバが接続されていません (Protocol driver not attached)

`errno.ENOCSI`

CSI 構造体がありません (No CSI structure available)

`errno.EL2HLT`

レベル 2 で終了しました (Level 2 halted)

`errno.EBADE`

無効な変換です (Invalid exchange)

`errno.EBADR`

無効な要求記述子です (Invalid request descriptor)

`errno.EXFULL`

変換テーブルが一杯です (Exchange full)

`errno.ENOANO`

陰極がありません (No anode)

`errno.EBADRQC`

無効なリクエストコードです (Invalid request code)

`errno.EBADSLT`

無効なスロットです (Invalid slot)

`errno.EDEADLOCK`

ファイルロックにおけるデッドロックエラーです (File locking deadlock error)

`errno.EBFONT`

フォントファイル形式が間違っています (Bad font file format)

`errno.ENOSTR`

ストリーム型でないデバイスです (Device not a stream)

`errno.ENODATA`

利用可能なデータがありません (No data available)

`errno.ETIME`

時間切れです (Timer expired)

`errno.ENOSR`

ストリームリソースを使い切りました (Out of streams resources)

`errno.ENONET`

計算機はネットワーク上にありません (Machine is not on the network)

`errno.ENOPKG`

パッケージがインストールされていません (Package not installed)

`errno.EREMOTE`

対象物は遠隔にあります (Object is remote)

`errno.ENOLINK`

リンクが切られました (Link has been severed)

`errno.EADV`

Advertise エラーです (Advertise error)

`errno.ESRMNT`

Srmount エラーです (Srmount error)

`errno.ECOMM`

送信時の通信エラーです (Communication error on send)

`errno.EPROTO`

プロトコルエラーです (Protocol error)

`errno.EMULTIHOP`

多重ホップを試みました (Multihop attempted)

`errno.EDOTDOT`

RFS 特有のエラーです (RFS specific error)

`errno.EBADMSG`

データメッセージではありません (Not a data message)

`errno.EOVERFLOW`

定義されたデータ型にとって大きすぎる値です (Value too large for defined data type)

`errno.ENOTUNIQ`

名前がネットワーク上で一意ではありません (Name not unique on network)

`errno.EBADFD`

ファイル記述子の状態が不正です (File descriptor in bad state)

`errno.EREMCHG`

遠隔のアドレスが変更されました (Remote address changed)

`errno.ELIBACC`

必要な共有ライブラリにアクセスできません (Can not access a needed shared library)

`errno.ELIBBAD`

壊れた共有ライブラリにアクセスしています (Accessing a corrupted shared library)

`errno.ELIBSCN`

a.out の .lib セクションが壊れています (.lib section in a.out corrupted)

`errno.ELIBMAX`

リンクを試みる共有ライブラリが多すぎます (Attempting to link in too many shared libraries)

`errno.ELIBEXEC`

共有ライブラリを直接実行することができません (Cannot exec a shared library directly)

`errno.EILSEQ`

不正なバイト列です (Illegal byte sequence)

`errno.ERESTART`

割り込みシステムコールを復帰しなければなりません (Interrupted system call should be restarted)

`errno.ESTRPIPE`

ストリームパイプのエラーです (Streams pipe error)

`errno.EUSERS`

ユーザが多すぎます (Too many users)

`errno.ENOTSOCK`

非ソケットに対するソケット操作です (Socket operation on non-socket)

`errno.EDESTADDRREQ`

目的アドレスが必要です (Destination address required)

`errno.EMSGSIZE`

メッセージが長すぎます (Message too long)

`errno.EPROTOTYPE`

ソケットに対して不正なプロトコル型です (Protocol wrong type for socket)

`errno.ENOPROTOOPT`

利用できないプロトコルです (Protocol not available)

`errno.EPROTONOSUPPORT`

サポートされていないプロトコルです (Protocol not supported)

`errno.ESOCKTNOSUPPORT`

サポートされていないソケット型です (Socket type not supported)

`errno.EOPNOTSUPP`

通信端点に対してサポートされていない操作です (Operation not supported on transport endpoint)

`errno.EPFNOSUPPORT`

サポートされていないプロトコルファミリです (Protocol family not supported)

`errno.EAFNOSUPPORT`

プロトコルでサポートされていないアドレスファミリです (Address family not supported by protocol)

`errno.EADDRINUSE`

アドレスは使用中です (Address already in use)

`errno.EADDRNOTAVAIL`

要求されたアドレスを割り当てできません (Cannot assign requested address)

`errno.ENETDOWN`

ネットワークがダウンしています (Network is down)

`errno.ENETUNREACH`

ネットワークに到達できません (Network is unreachable)

`errno.ENETRESET`

リセットによってネットワーク接続が切られました (Network dropped connection because of reset)

`errno.ECONNABORTED`

ソフトウェアによって接続が終了されました (Software caused connection abort)

`errno.ECONNRESET`

接続がピアによってリセットされました (Connection reset by peer)

`errno.ENOBUFS`

バッファに空きがありません (No buffer space available)

`errno.EISCONN`

通信端点がすでに接続されています (Transport endpoint is already connected)

`errno.ENOTCONN`

通信端点が接続されていません (Transport endpoint is not connected)

`errno.ESHUTDOWN`

通信端点のシャットダウン後は送信できません (Cannot send after transport endpoint shutdown)

`errno.ETOOMANYREFS`

参照が多すぎます: 接続できません (Too many references: cannot splice)

`errno.ETIMEDOUT`

接続がタイムアウトしました (Connection timed out)

`errno.ECONNREFUSED`

接続を拒否されました (Connection refused)

`errno.EHOSTDOWN`

ホストはシステムダウンしています (Host is down)

`errno.EHOSTUNREACH`

ホストへの経路がありません (No route to host)

`errno.EALREADY`

すでに処理中です (Operation already in progress)

`errno.EINPROGRESS`

現在処理中です (Operation now in progress)

`errno.ESTALE`

無効な NFS ファイルハンドルです (Stale NFS file handle)

`errno.EUCLEAN`

構造のクリーニングが必要です (Structure needs cleaning)

`errno.ENOTNAM`

XENIX 名前付きファイルではありません (Not a XENIX named type file)

`errno.ENAVAIL`

XENIX セマフォは利用できません (No XENIX semaphores available)

`errno.EISNAM`

名前付きファイルです (Is a named type file)

`errno.EREMOTEIO`

遠隔側の I/O エラーです (Remote I/O error)

`errno.EDQUOT`

ディスククォータを超えました (Quota exceeded)

15.17 ctypes — Python のための外部関数ライブラリ

バージョン 2.5 で追加.

`ctypes` は Python のための外部関数ライブラリです。このライブラリは C と互換性のあるデータ型を提供し、動的リンク/共有ライブラリ内の関数呼び出しを可能にします。動的リンク/共有ライブラリを純粋な Python でラップするために使うことができます。

15.17.1 ctypes チュートリアル

注意: このチュートリアルのコードサンプルは動作確認のために `doctest` を使います。コードサンプルの中には Linux、Windows、あるいは Mac OS X 上で異なる動作をするものがあるため、サンプルのコメントに `doctest` 命令を入れてあります。

注意: いくつかのコードサンプルで `ctypes` の `c_int` 型を参照しています。32 ビットシステムにおいてこの型は `c_long` 型のエイリアスです。そのため、`c_int` 型を想定しているときに `c_long` が表示されたとしても、混乱しないようにしてください — 実際には同じ型なのです。

動的リンクライブラリをロードする

動的リンクライブラリをロードするために、`ctypes` は `cdll` をエクスポートします。Windows では `windll` と `oledll` オブジェクトをエクスポートします。

これらのオブジェクトの属性としてライブラリにアクセスすることでライブラリをロードします。`cdll` は標準 `cdecl` 呼び出し規約を用いて関数をエクスポートしているライブラリをロードします。それに対して、`windll` ライブラリは `stdcall` 呼び出し規約を用いる関数を呼び出します。`oledll` も `stdcall` 呼び出し規約を使いますが、関数が Windows HRESULT エラーコードを返すことを想定しています。このエラーコードは関数呼び出しが失敗したとき、`WindowsError` 例外を自動的に送出させるために使われます。

Windows 用の例ですが、`msvcrt` はほとんどの標準 C 関数が含まれている MS 標準 C ライブラリであり、`cdecl` 呼び出し規約を使うことに注意してください:

```
>>> from ctypes import *
>>> print windll.kernel32
<WinDLL 'kernel32', handle ... at ...>
>>> print cdll.msvcrt
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt
>>>
```

Windows では通常の `.dll` ファイル拡張子を自動的に追加します。

Linux ではライブラリをロードするために拡張子を含むファイル名を指定する必要があるので、ロードしたライブラリに対する属性アクセスはできません。dll ローダーの `LoadLibrary()` メソッドを使うか、コンストラクタを呼び出して CDLL のインスタンスをすることでライブラリをロードするかのどちらかを行わなければならない:

```
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

ロードした dll から関数にアクセスする

dll オブジェクトの属性として関数にアクセスします:

```
>>> from ctypes import *
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print windll.kernel32.GetModuleHandleA
<_FuncPtr object at 0x...>
>>> print windll.kernel32.MyOwnFunction
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
>>>
```

`kernel32` や `user32` のような win32 システム dll は、多くの場合関数の UNICODE バージョンに加えて ANSI バージョンもエクスポートすることに注意してください。UNICODE バージョンは後ろに `W` が付いた名前でエクスポートされ、ANSI バージョンは `A` が付いた名前でエクスポートされます。与えられたモジュールのモジュールハンドルを返す win32 `GetModuleHandle` 関数は次のような C プロトタイプを持ちます。UNICODE バージョンが定義されているかどうかにより `GetModuleHandle` としてどちらか一つを公開するためにマクロが使われます:


```
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

`windll` は魔法を使ってどちらか一つを選ぶようなことはしません。 `GetModuleHandleA` もしくは `GetModuleHandleW` を明示的に指定して必要とするバージョンにアクセスし、文字列か Unicode 文字列を使ってそれぞれ呼び出さなければなりません。

時には、dll が関数を `"??2@YAPAXI@Z"` のような Python 識別子として有効でない名前でエクスポートすることがあります。このような場合に関数を取り出すには、`getattr()` を使わなければなりません。:

```
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z")
<_FuncPtr object at 0x...>
>>>
```

Windows では、名前ではなく序数によって関数をエクスポートする dll もあります。こうした関数には序数を使って dll オブジェクトにインデックス指定することでアクセスします:

```
>>> cdll.kernel32[1]
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
>>>
```

関数を呼び出す

これらの関数は他の Python 呼び出し可能オブジェクトと同じように呼び出すことができます。この例では `time()` 関数 (Unix エポックからのシステム時間を秒単位で返す) と、`GetModuleHandleA()` 関数 (`win32` モジュールハンドルを返す) を使います。

この例は両方の関数を `NULL` ポインタとともに呼び出します (`None` を `NULL` ポインタとして使う必要があります):

```
>>> print libc.time(None)
1150640792
>>> print hex(windll.kernel32.GetModuleHandleA(None))
0x1d000000
>>>
```

`ctypes` は引数の数を間違えたり、あるいは呼び出し規約を間違えた関数呼び出しからあなたを守ろうとします。残念ながら、これは Windows でしか機能しません。関数が返った後にスタックを調べることでこれを行います。したがって、エラーは発生しますが、その関数は呼び出された 後です:

```
>>> windll.kernel32.GetModuleHandleA()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>> windll.kernel32.GetModuleHandleA(0, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

同じ例外が `cdecl` 呼び出し規約を使って `stdcall` 関数を呼び出したときに送出されますし、逆の場合も同様です。:

```
>>> cdll.kernel32.GetModuleHandleA(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>>

>>> windll.msvcrt.printf("spam")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

正しい呼び出し規約を知るためには、呼び出したい関数についての C ヘッダファイルもしくはドキュメントを見なければなりません。

Windows では、関数が無効な引数とともに呼び出された場合の一般保護例外によるクラッシュを防ぐために、`ctypes` は win32 構造化例外処理を使います:

```
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
WindowsError: exception: access violation reading 0x00000020
>>>
```

しかし、`ctypes` を使って Python をクラッシュさせる方法は十分なほどあるので、よく注意すべきです。

`None`、整数、長整数、バイト文字列および Unicode 文字列だけが、こうした関数呼び出しにおいてパラメータとして直接使えるネイティブの Python オブジェクトです。 `None` は C の `NULL` ポインタとして渡され、バイト文字列と Unicode 文字列はそのデータを含むメモリブロックへのポインタ (`char *` または `wchar_t *`) として渡されます。 Python 整数と Python 長整数はプラットフォームのデフォルトの C `int` 型として渡され、その値は C `int` 型に合うようにマスクされます。

他のパラメータ型をもつ関数呼び出しに移る前に、`ctypes` データ型についてさらに学ぶ必要があります。

基本データ型

`ctypes` ではいくつかの C 互換のプリミティブなデータ型を定義しています:

ctypes の型	C の型	Python の型
<code>c_bool</code>	<code>_Bool</code>	<code>bool</code> (1)
<code>c_char</code>	<code>char</code>	1 文字の文字列
<code>c_wchar</code>	<code>wchar_t</code>	1 文字の Unicode 文字列
<code>c_byte</code>	<code>char</code>	整数/長整数
<code>c_ubyte</code>	<code>unsigned char</code>	整数/長整数
<code>c_short</code>	<code>short</code>	整数/長整数
<code>c_ushort</code>	<code>unsigned short</code>	整数/長整数
<code>c_int</code>	<code>int</code>	整数/長整数
<code>c_uint</code>	<code>unsigned int</code>	整数/長整数
<code>c_long</code>	<code>long</code>	整数/長整数
<code>c_ulong</code>	<code>unsigned long</code>	整数/長整数
<code>c_longlong</code>	<code>__int64</code> または <code>long long</code>	整数/長整数
<code>c_ulonglong</code>	<code>unsigned __int64</code> または <code>unsigned long long</code>	整数/長整数
<code>c_float</code>	<code>float</code>	浮動小数点数
<code>c_double</code>	<code>double</code>	浮動小数点数
<code>c_longdouble</code>	<code>long double</code>	浮動小数点数
<code>c_char_p</code>	<code>char *</code> (NUL 終端)	文字列または <code>None</code>
<code>c_wchar_p</code>	<code>wchar_t *</code> (NUL 終端)	Unicode または <code>None</code>
<code>c_void_p</code>	<code>void *</code>	整数/長整数または <code>None</code>

(1) コンストラクタは任意のオブジェクトをその真偽値として受け取ります。

これら全ての型はその型を呼び出すことによって作成でき、オプションとして型と値が合っている初期化子を指定することができます:

```
>>> c_int()
c_long(0)
>>> c_char_p("Hello, World")
c_char_p('Hello, World')
>>> c_ushort(-3)
c_ushort(65533)
>>>
```

これらの型は変更可能であり、値を後で変更することもできます:

```
>>> i = c_int(42)
>>> print i
c_long(42)
>>> print i.value
42
>>> i.value = -99
>>> print i.value
-99
>>>
```

新しい値をポインタ型 `c_char_p`, `c_wchar_p` および `c_void_p` のインスタンスへ代入すると、変わるのは

指しているメモリ位置であって、メモリブロックの内容ではありません (これは当然で、なぜなら、Python 文字列は変更不可能だからです):

```
>>> s = "Hello, World"
>>> c_s = c_char_p(s)
>>> print c_s
c_char_p('Hello, World')
>>> c_s.value = "Hi, there"
>>> print c_s
c_char_p('Hi, there')
>>> print s                                # first string is unchanged
Hello, World
>>>
```

しかし、変更可能なメモリを指すポインタであることを想定している関数へそれらを渡さないように注意すべきです。もし変更可能なメモリブロックが必要なら、ctypes には `create_string_buffer()` 関数があり、いろいろな方法で作成することができます。現在のメモリブロックの内容は `raw` プロパティを使ってアクセス (あるいは変更) することができます。もし現在のメモリブロックに NUL 終端文字列としてアクセスしたいなら、`value` プロパティを使ってください:

```
>>> from ctypes import *
>>> p = create_string_buffer(3)          # create a 3 byte buffer, initialized to NUL
↳ bytes
>>> print sizeof(p), repr(p.raw)
3 '\x00\x00\x00'
>>> p = create_string_buffer("Hello")    # create a buffer containing a NUL
↳ terminated string
>>> print sizeof(p), repr(p.raw)
6 'Hello\x00'
>>> print repr(p.value)
'Hello'
>>> p = create_string_buffer("Hello", 10) # create a 10 byte buffer
>>> print sizeof(p), repr(p.raw)
10 'Hello\x00\x00\x00\x00\x00'
>>> p.value = "Hi"
>>> print sizeof(p), repr(p.raw)
10 'Hi\x00lo\x00\x00\x00\x00'
>>>
```

`create_string_buffer()` 関数は初期の ctypes リリースにあった `c_string()` 関数だけでなく、(エイリアスとしてはまだ利用できる) `c_buffer()` 関数をも置き換えるものです。C の型 `wchar_t` の Unicode 文字を含む変更可能なメモリブロックを作成するには、`create_unicode_buffer()` 関数を使ってください。

続・関数を呼び出す

`printf` は `sys.stdout` ではなく、本物の標準出力チャンネルへプリントすることに注意してください。したがって、これらの例はコンソールプロンプトでのみ動作し、`IDLE` や `PythonWin` では動作しません。:

```
>>> printf = libc.printf
>>> printf("Hello, %s\n", "World!")
Hello, World!
14
>>> printf("Hello, %S\n", u"World!")
Hello, World!
14
>>> printf("%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf("%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: Don't know how to convert_
↪parameter 2
>>>
```

前に述べたように、必要な C のデータ型へ変換できるようにするためには、整数、文字列および Unicode 文字列を除くすべての Python 型を対応する *ctypes* 型でラップしなければなりません。:

```
>>> printf("An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
31
>>>
```

自作のデータ型とともに関数を呼び出す

自作のクラスのインスタンスを関数引数として使えるように、*ctypes* 引数変換をカスタマイズすることもできます。*ctypes* は `_as_parameter_` 属性を探し出し、関数引数として使います。もちろん、整数、文字列もしくは Unicode の中の一つでなければなりません。:

```
>>> class Bottles(object):
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf("%d bottles of beer\n", bottles)
42 bottles of beer
19
>>>
```

インスタンスのデータを `_as_parameter_` インスタンス変数の中に入れたくない場合には、そのデータを利用できるようにする *property()* を定義することができます。

要求される引数の型を指定する (関数プロトタイプ)

`argtypes` 属性を設定することによって、DLL からエクスポートされている関数に要求される引数の型を指定することができます。

`argtypes` は C データ型のシーケンスでなければなりません (この場合 `printf` 関数はおそらく良い例ではありません。なぜなら、引数の数が可変であり、フォーマット文字列に依存した異なる型のパラメータを取るからです。一方では、この機能の実験にはとても便利です)。

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf("String '%s', Int %d, Double %f\n", "Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>>
```

(C の関数のプロトタイプのように) 書式を指定すると互換性のない引数型になるのを防ぎ、引数を有効な型へ変換しようとします。

```
>>> printf("%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: wrong type
>>> printf("%s %d %f\n", "X", 2, 3)
X 2 3.000000
13
>>>
```

関数呼び出しへ渡す自作のクラスを定義した場合には、`argtypes` シーケンスの中で使えるようにするために、そのクラスに `from_param()` クラスメソッドを実装しなければなりません。`from_param()` クラスメソッドは関数呼び出しへ渡された Python オブジェクトを受け取り、型チェックもしくはこのオブジェクトが受け入れ可能であると確かめるために必要なことはすべて行ってから、オブジェクト自身、`_as_parameter_` 属性、あるいは、この場合に C 関数引数として渡したい何かの値を返さなければなりません。繰り返しになりますが、その返される結果は整数、文字列、Unicode、`ctypes` インスタンス、あるいは `_as_parameter_` 属性をもつオブジェクトであるべきです。

戻り値の型

デフォルトでは、関数は `C int` を返すと仮定されます。他の戻り値の型を指定するには、関数オブジェクトの `restype` 属性に設定します。

さらに高度な例として、`strchr` 関数を使います。この関数は文字列ポインタと `char` を受け取り、文字列へのポインタを返します。

```
>>> strchr = libc.strchr
>>> strchr("abcdef", ord("d"))
8059983
>>> strchr.restype = c_char_p # c_char_p is a pointer to a string
>>> strchr("abcdef", ord("d"))
'def'
>>> print strchr("abcdef", ord("x"))
None
>>>
```

上の `ord("x")` 呼び出しを避けたいなら、`argtypes` 属性を設定することができます。二番目の引数が一

文字の Python 文字列から C の char へ変換されます。:

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr("abcdef", "d")
'def'
>>> strchr("abcdef", "def")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: one character string expected
>>> print strchr("abcdef", "x")
None
>>> strchr("abcdef", "d")
'def'
>>>
```

外部関数が整数を返す場合は、`restype` 属性として呼び出し可能な Python オブジェクト (例えば、関数またはクラス) を使うこともできます。呼び出し可能オブジェクトは C 関数が返す 整数 とともに呼び出され、この呼び出しの結果は関数呼び出しの結果として使われるでしょう。これはエラーの戻り値をチェックして自動的に例外を送出させるために役に立ちます。:

```
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
...
>>>
>>> GetModuleHandle.restype = ValidHandle
>>> GetModuleHandle(None)
486539264
>>> GetModuleHandle("something silly")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in ValidHandle
WindowsError: [Errno 126] The specified module could not be found.
>>>
```

`WinError` はエラーコードの文字列表現を得るために Windows の `FormatMessage()` api を呼び出し、例外を返す関数です。`WinError` はオプションでエラーコードパラメータを取ります。このパラメータが使われない場合は、エラーコードを取り出すために `GetLastError()` を呼び出します。

`errcheck` 属性によってもっと強力なエラーチェック機構を利用できることに注意してください。詳細はリファレンスマニュアルを参照してください。

ポインタを渡す (または、パラメータの参照渡し)

時には、C api 関数がパラメータのデータ型としてポインタを想定していることがあります。おそらくパラメータと同一の場所へ書き込むためか、もしくはそのデータが大きすぎて値渡しできない場合です。これはパラメータの参照渡しとしても知られています。

`ctypes` は `byref()` 関数をエクスポートしており、パラメータを参照渡しするために使用します。`pointer()` 関数を使っても同じ効果が得られます。しかし、`pointer()` は本当のポインタオブジェクトを構築するためより多くの処理を行うことから、Python 側でポインタオブジェクト自体を必要としないならば `byref()` を使う方がより高速です。:

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer('\000' * 32)
>>> print i.value, f.value, repr(s.value)
0 0.0 ''
>>> libc sscanf("1 3.14 Hello", "%d %f %s",
...             byref(i), byref(f), s)
3
>>> print i.value, f.value, repr(s.value)
1 3.1400001049 'Hello'
>>>
```

構造体と共用体

構造体と共用体は `ctypes` モジュールに定義されている `Structure` および `Union` ベースクラスからの派生クラスでなければなりません。それぞれのサブクラスは `_fields_` 属性を定義する必要があります。`_fields_` はフィールド名とフィールド型を持つ 2 要素タプルのリストでなければなりません。

フィールド型は `c_int` か他の `ctypes` 型 (構造体、共用体、配列、ポインタ) から派生した `ctypes` 型である必要があります。

`x` と `y` という名前の二つの整数からなる簡単な POINT 構造体の例です。コンストラクタで構造体の初期化する方法の説明にもなっています。:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
>>> point = POINT(10, 20)
>>> print point.x, point.y
10 20
>>> point = POINT(y=5)
>>> print point.x, point.y
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: too many initializers
>>>
```

しかし、もっと複雑な構造体を構築することもできます。ある構造体は、他の構造体をフィールド型として使うことで、他の構造体を含むことができます。

`upperleft` と `lowerright` という名前の二つの POINT を持つ RECT 構造体です。:

```
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                  ("lowerright", POINT)]
...
>>> rc = RECT(point)
>>> print rc.upperleft.x, rc.upperleft.y
0 5
>>> print rc.lowerright.x, rc.lowerright.y
0 0
>>>
```

入れ子になった構造体はいくつかの方法を用いてコンストラクタで初期化することができます。:

```
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

フィールド *descriptor* (記述子) は クラス から取り出せます。デバッグするときに役に立つ情報を得ることができます:

```
>>> print POINT.x
<Field type=c_long, ofs=0, size=4>
>>> print POINT.y
<Field type=c_long, ofs=4, size=4>
>>>
```

警告: *ctypes* では、ビットフィールドのある共用体や構造体の関数への値渡しはサポートしていません。これは 32-bit の x86 環境では動くかもしれませんが、このライブラリでは一般の場合に動作することは保証していません。

構造体/共用体アライメントとバイトオーダー

デフォルトでは、構造体 (Structure) と共用体 (Union) のフィールドは C コンパイラが行うのと同じ方法でアライメントされています。サブクラスを定義するときに `_pack_` クラス属性を指定することでこの動作を変えることは可能です。このクラス属性には正の整数を設定する必要があり、フィールドの最大アライメントを指定します。これは MSVC で `#pragma pack(n)` が行っていることと同じです。

ctypes は Structure と Union に対してネイティブのバイトオーダーを使います。ネイティブではないバイトオーダーの構造体を作成するには、*BigEndianStructure*、*LittleEndianStructure*、*BigEndianUnion* および *LittleEndianUnion* ベースクラスの中の一つを使います。これらのクラスにポインタフィールドを持たせることはできません。

構造体と共用体におけるビットフィールド

ビットフィールドを含む構造体と共用体を作ることができます。ビットフィールドは整数フィールドに対してのみ作ることができ、ビット幅は `_fields_` タブルの第三要素で指定します。:

```
>>> class Int(Structure):
...     _fields_ = [("first_16", c_int, 16),
...                 ("second_16", c_int, 16)]
...
>>> print Int.first_16
<Field type=c_long, ofs=0:0, bits=16>
>>> print Int.second_16
<Field type=c_long, ofs=0:16, bits=16>
>>>
```

配列

配列 (Array) はシーケンスであり、決まった数の同じ型のインスタンスを持ちます。

推奨されている配列の作成方法はデータ型に正の整数を掛けることです。:

```
TenPointsArrayType = POINT * 10
```

ややわざとらしいデータ型の例になりますが、他のものに混ざって 4 個の POINT がある構造体です:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class MyStruct(Structure):
...     _fields_ = [("a", c_int),
...                 ("b", c_float),
...                 ("point_array", POINT * 4)]
>>>
>>> print len(MyStruct().point_array)
4
>>>
```

インスタンスはクラスを呼び出す通常の方法で作成します。:

```
arr = TenPointsArrayType()
for pt in arr:
    print pt.x, pt.y
```

上記のコードは 0 0 という行が並んだものを表示します。配列の要素がゼロで初期化されているためです。

正しい型の初期化子を指定することもできます。:

```
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print ii
<c_long_Array_10 object at 0x...>
>>> for i in ii: print i,
...

```

(次のページに続く)

(前のページからの続き)

```
1 2 3 4 5 6 7 8 9 10
>>>
```

ポインタ

ポインタのインスタンスは `ctypes` 型に対して `pointer()` 関数を呼び出して作成します。:

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

次のように、ポインタインスタンスは、ポインタが指すオブジェクト (上の例では `i`) を返す `contents` 属性を持ちます:

```
>>> pi.contents
c_long(42)
>>>
```

`ctypes` は OOR (original object return、元のオブジェクトを返すこと) ではないことに注意してください。属性を取り出す度に、新しい同等のオブジェクトを作成しているのです。:

```
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
>>>
```

別の `c_int` インスタンスがポインタの `contents` 属性に代入されると、これが記憶されているメモリ位置を指すポインタに変化します。:

```
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
c_long(99)
>>>
```

ポインタインスタンスは整数でインデックス指定することもできます。:

```
>>> pi[0]
99
>>>
```

整数インデックスへ代入するとポインタが指す値が変更されます。:

```
>>> print i
c_long(99)
>>> pi[0] = 22
```

(次のページに続く)

(前のページからの続き)

```
>>> print i
c_long(22)
>>>
```

0 ではないインデックスを使うこともできますが、C の場合と同じように自分が何をしているかを理解している必要があります。任意のメモリ位置にアクセスもしくは変更できるのです。一般的にこの機能を使うのは、C 関数からポインタを受け取り、そのポインタが単一の要素ではなく実際に配列を指していると分かっている場合だけです。

舞台裏では、`pointer()` 関数は単にポインタインスタンスを作成するという以上のことを行っています。はじめにポインタ型を作成する必要があります。これは任意の `ctypes` 型を受け取る `POINTER()` 関数を使って行われ、新しい型を返します。:

```
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_int'>
>>> PI(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected c_int instead of int
>>> PI(c_int(42))
<ctypes.LP_c_int object at 0x...>
>>>
```

ポインタ型を引数なしで呼び出すと NULL ポインタを作成します。NULL ポインタは `False` ブール値を持っています。:

```
>>> null_ptr = POINTER(c_int)()
>>> print bool(null_ptr)
False
>>>
```

`ctypes` はポインタの指す値を取り出すときに NULL かどうかを調べます (しかし、NULL でない不正なポインタの指す値の取り出す行為は Python をクラッシュさせるでしょう)。:

```
>>> null_ptr[0]
Traceback (most recent call last):
  ....
ValueError: NULL pointer access
>>>

>>> null_ptr[0] = 1234
Traceback (most recent call last):
  ....
ValueError: NULL pointer access
>>>
```

型変換

たいいていの場合、ctypes は厳密な型チェックを行います。これが意味するのは、関数の `argtypes` リスト内に、もしくは、構造体定義におけるメンバーフィールドの型として `POINTER(c_int)` がある場合、厳密に同じ型のインスタンスだけを受け取るということです。このルールには ctypes が他のオブジェクトを受け取る場合に例外がいくつかあります。例えば、ポインタ型の代わりに互換性のある配列インスタンスを渡すことができます。このように、`POINTER(c_int)` に対して、ctypes は `c_int` の配列を受け取ります。:

```
>>> class Bar(Structure):
...     _fields_ = [("count", c_int), ("values", POINTER(c_int))]
...
>>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print bar.values[i]
...
1
2
3
>>>
```

それに加えて、`argtypes` で関数の引数が明示的に (`POINTER(c_int)` などの) ポインタ型であると宣言されていた場合、ポインタ型が指し示している型のオブジェクト (この場合では `c_int`) を関数に渡すことができます。この場合 ctypes は、必要となる `byref()` での変換を自動的に適用します。

`POINTER` 型フィールドを `NULL` に設定するために、`None` を代入してもかまいません。:

```
>>> bar.values = None
>>>
```

時には、非互換な型のインスタンスであることもあります。C では、ある型を他の型へキャストすることができます。ctypes は同じやり方で使える `cast()` 関数を提供しています。上で定義した `Bar` 構造体は `POINTER(c_int)` ポインタまたは `c_int` 配列を `values` フィールドに対して受け取り、他の型のインスタンスは受け取りません:

```
>>> bar.values = (c_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: incompatible types, c_byte_Array_4 instance instead of LP_c_long_
↳instance
>>>
```

このような場合には、`cast()` 関数が便利です。

`cast()` 関数は ctypes インスタンスを異なる ctypes データ型を指すポインタへキャストするために使えます。`cast()` は二つのパラメータ、ある種のポインタかそのポインタへ変換できる ctypes オブジェクトと、ctypes ポインタ型を取ります。そして、第二引数のインスタンスを返します。このインスタンスは第一引数と同じメモリブロックを参照しています:

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
>>>
```

したがって、`cast()` を `Bar` 構造体の `values` フィールドへ代入するために使うことができます:

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print bar.values[0]
0
>>>
```

不完全型

不完全型 はメンバーがまだ指定されていない構造体、共用体もしくは配列です。C では、前方宣言により指定され、後で定義されます。:

```
struct cell; /* forward declaration */

struct cell {
    char *name;
    struct cell *next;
};
```

`ctypes` コードへの直接的な変換ではこうなるでしょう。しかし、動作しません:

```
>>> class cell(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
>>>
```

なぜなら、新しい `class cell` はクラス文自体の中では利用できないからです。`ctypes` では、`cell` クラスを定義して、`_fields_` 属性をクラス文の後で設定することができます。:

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
...                  ("next", POINTER(cell))]
>>>
```

試してみましょう。`cell` のインスタンスを二つ作り、互いに参照し合うようにします。最後に、つながったポインタを何度かたどります。:


```
>>> c1 = cell()
>>> c1.name = "foo"
>>> c2 = cell()
>>> c2.name = "bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print p.name,
...     p = p.next[0]
...
foo bar foo bar foo bar foo bar
>>>
```

コールバック関数

ctypes は C の呼び出し可能な関数ポインタを Python 呼び出し可能オブジェクトから作成できるようにします。これらは コールバック関数 と呼ばれることがあります。

最初に、コールバック関数のためのクラスを作る必要があります。そのクラスには呼び出し規約、戻り値の型およびこの関数が受け取る引数の数と型についての情報があります。

CFUNCTYPE ファクトリ関数は通常の `cdecl` 呼び出し規約を用いてコールバック関数のための型を作成します。Windows では、WINFUNCTYPE ファクトリ関数が `stdcall` 呼び出し規約を用いてコールバック関数の型を作成します。

これらのファクトリ関数はともに最初の引数に戻り値の型、残りの引数としてコールバック関数が想定する引数の型を渡して呼び出されます。

標準 C ライブラリの `qsort()` 関数を使う例を示します。これはコールバック関数の助けをかりて要素をソートするために使われます。 `qsort()` は整数の配列をソートするために使われます。:

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
>>>
```

`qsort()` はソートするデータを指すポインタ、データ配列の要素の数、要素の一つの大きさ、およびコールバック関数である比較関数へのポインタを引数に渡して呼び出さなければなりません。そして、コールバック関数は要素を指す二つのポインタを渡されて呼び出され、一番目が二番目より小さいなら負の数を、等しいならゼロを、それ以外なら正の数を返さなければなりません。

コールバック関数は整数へのポインタを受け取り、整数を返す必要があります。まず、コールバック関数のための `type` を作成します。:

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>>
```

コールバック関数ののはじめての実装なので、受け取った引数を単純に表示して、0 を返します (漸進型開発 (incremental development) です ;-):

```
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a, b
...     return 0
...
>>>
```

C の呼び出し可能なコールバック関数を作成します。:

```
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

そうすると、準備完了です。:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00..
↪.>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00..
↪.>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00..
↪.>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00..
↪.>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00..
↪.>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00..
↪.>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00..
↪.>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00..
↪.>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00..
↪.>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00..
↪.>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00..
↪.>
>>>
```

ポインタの中身にアクセスする方法がわかっているので、コールバック関数を再定義しましょう。:

```
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a[0], b[0]
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

Windows での実行結果です。:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 7 1
py_cmp_func 33 1
py_cmp_func 99 1
py_cmp_func 5 1
py_cmp_func 7 5
py_cmp_func 33 5
py_cmp_func 99 5
py_cmp_func 7 99
py_cmp_func 33 99
py_cmp_func 7 33
>>>
```

linux ではソート関数のはるかに効率的に動作しており、実施する比較の数が少ないように見えるのが不思議です。:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
>>>
```

ええ、ほぼ完成です! 最終段階は、実際に二つの要素を比較して実用的な結果を返すことです。:

```
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a[0], b[0]
...     return a[0] - b[0]
...
>>>
```

Windows での最終的な実行結果です。:

```
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
py_cmp_func 33 7
py_cmp_func 99 33
py_cmp_func 5 99
py_cmp_func 1 99
py_cmp_func 33 7
py_cmp_func 1 33
py_cmp_func 5 33
py_cmp_func 5 7
py_cmp_func 1 7
py_cmp_func 5 1
>>>
```

Linux では:

```
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
py_cmp_func 5 1
```

(次のページに続く)

(前のページからの続き)

```
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

Windows の `qsort()` 関数は linux バージョンより多く比較する必要があることがわかり、非常におもしろいですね!

簡単に確認できるように、今では配列はソートされています。:

```
>>> for i in ia: print i,
...
1 5 7 33 99
>>>
```

注釈: C コードから使われる限り、`CFUNCTYPE()` オブジェクトへの参照を確実に保持してください。`ctypes` は保持しません。もしあなたがやらなければ、オブジェクトはゴミ集めされてしまい、コールバックしたときにあなたのプログラムをクラッシュさせるかもしれません。

同様に、コールバック関数が Python の管理外 (例えば、コールバックを呼び出す外部のコード) で作られたスレッドで呼び出された場合、`ctypes` は全ての呼び出しごとに新しいダミーの Python スレッドを作成することに注意してください。この動作はほとんどの目的に対して正しいものですが、同じ C スレッドからの呼び出しだったとしても、`threading.local` で格納された値は異なるコールバックをまたいで生存はしません。

dll からエクスポートされている値へアクセスする

共有ライブラリの一部は関数だけでなく変数もエクスポートしています。Python ライブラリにある例としては `Py_OptimizeFlag`、起動時の `-O` または `-OO` フラグに依存して、0, 1 または 2 が設定される整数があります。

`ctypes` は型の `in_dll()` クラスメソッドを使ってこのように値にアクセスできます。`pythonapi` は Python C api へアクセスできるようにするための予め定義されたシンボルです。:

```
>>> opt_flag = c_int.in_dll(pythonapi, "Py_OptimizeFlag")
>>> print opt_flag
c_long(0)
>>>
```

インタプリタが `-O` を指定されて動き始めた場合、サンプルは `c_long(1)` を表示するでしょうし、`-OO` が指定されたならば `c_long(2)` を表示するでしょう。

ポインタの使い方を説明する拡張例では、Python がエクスポートする `PyImport_FrozenModules` ポインタにアクセスします。

Python ドキュメントから引用すると: このポインタは `"struct frozen"` のレコードからなり、終端の要素のメン

バが `NULL` かゼロになっているような配列を指すよう初期化されます。フリーズされたモジュールを `import` するとき、このテーブルを検索します。サードパーティ製のコードからこのポインタに仕掛けを講じて、動的に生成されたフリーズ化モジュールの集合を提供するようにできます。

これで、このポインタを操作することが役に立つことを証明できるでしょう。例の大きさを制限するために、このテーブルを `ctypes` を使って読む方法だけを示します。:

```
>>> from ctypes import *
>>>
>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("code", POINTER(c_ubyte)),
...                 ("size", c_int)]
...
>>>
```

私たちは `struct _frozen` データ型を定義済みなので、このテーブルを指すポインタを得ることができます。:

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "PyImport_FrozenModules")
>>>
```

`table` が `struct_frozen` レコードの配列への `pointer` なので、その配列に対して反復処理を行えます。しかし、ループが確実に終了するようにする必要があります。なぜなら、ポインタに大きさの情報がないからです。遅かれ早かれ、アクセス違反が何かでクラッシュすることになるでしょう。 `NULL` エントリに達したときはループを抜ける方が良いです。:

```
>>> for item in table:
...     print item.name, item.size
...     if item.name is None:
...         break
...
__hello__ 104
__phello__ -104
__phello__.spam 104
None 0
>>>
```

標準 Python はフローズンモジュールとフローズンパッケージ (負のサイズのメンバーで表されています) を持っているという事実はあまり知られておらず、テストにだけ使われています。例えば、`import __hello__` を試してみてください。

びっくり仰天

There are some edge cases in `ctypes` where you might expect something other than what actually happens.

次に示す例について考えてみてください。:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print rc.a.x, rc.a.y, rc.b.x, rc.b.y
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print rc.a.x, rc.a.y, rc.b.x, rc.b.y
3 4 3 4
>>>
```

うーん、最後の文に 3 4 1 2 と表示されることを期待していたはずですが。何が起きたのでしょうか? 上の行の `rc.a, rc.b = rc.b, rc.a` の各段階はこうになります。:

```
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
>>>
```

`temp0` と `temp1` は前記の `rc` オブジェクトの内部バッファでまだ使われているオブジェクトです。したがって、`rc.a = temp0` を実行すると `temp0` のバッファ内容が `rc` のバッファへコピーされます。さらに、これは `temp1` の内容を変更します。そのため、最後の代入 `rc.b = temp1` は、期待する結果にはならないのです。

`Structure`、`Union` および `Array` のサブオブジェクトを取り出しても、そのサブオブジェクトがコピーされるわけではなく、ルートオブジェクトの内部バッファにアクセスするラッパーオブジェクトを取り出すことを覚えておいてください。

期待とは違う振る舞いをする別の例はこれです。:

```
>>> s = c_char_p()
>>> s.value = "abc def ghi"
>>> s.value
'abc def ghi'
>>> s.value is s.value
False
>>>
```

なぜ `False` と表示されるのでしょうか? `ctypes` インスタンスはメモリと、メモリの内容にアクセスするいくつかの *descriptor* (記述子) を含むオブジェクトです。メモリブロックに Python オブジェクトを保存してもオブジェクト自身が保存される訳ではなく、オブジェクトの `contents` が保存されます。その `contents` に再アクセスすると新しい Python オブジェクトがその度に作られます。

可変サイズのデータ型

`ctypes` は可変サイズの配列と構造体をサポートしています。

`resize()` 関数は既存の `ctypes` オブジェクトのメモリバッファのサイズを変更したい場合に使えます。この関数は第一引数にオブジェクト、第二引数に要求されたサイズをバイト単位で指定します。メモリブロックはオブジェクト型で指定される通常のメモリブロックより小さくすることはできません。これをやろうとすると、`ValueError` が送出されます。:

```
>>> short_array = (c_short * 4)()
>>> print sizeof(short_array)
8
>>> resize(short_array, 4)
Traceback (most recent call last):
...
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>>
```

これはこれで上手くいっていますが、この配列の追加した要素へどうやってアクセスするのでしょうか? この型は要素の数が4個であるとまだ認識しているので、他の要素にアクセスするとエラーになります。:

```
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
...
IndexError: invalid index
>>>
```

`ctypes` で可変サイズのデータ型を使うもう一つの方法は、必要なサイズが分かった後に Python の動的性質を使って一つ一つデータ型を (再) 定義することです。

15.17.2 ctypes リファレンス

共有ライブラリを見つける

コンパイルされる言語でプログラミングしている場合、共有ライブラリはプログラムをコンパイル/リンクしているときと、そのプログラムが動作しているときにアクセスされます。

`ctypes` ライブラリローダーはプログラムが動作しているときのように振る舞い、ランタイムローダーを直接呼び出すのに対し、`find_library()` 関数の目的はコンパイラが行うのと似た方法でライブラリを探し出すことです。(複数のバージョンの共有ライブラリがあるプラットフォームでは、一番最近に見つかったものがロードされます)。

`ctypes.util` モジュールはロードするライブラリを決めるのに役立つ関数を提供します。

`ctypes.util.find_library(name)`

ライブラリを見つけてパス名を返そうと試みます。 *name* は `lib` のような接頭辞、 `.so`、`.dylib` のような接尾辞、あるいは、バージョン番号が何も付いていないライブラリの名前です (これは `posix` リンカ のオプション `-l` に使われている形式です)。もしライブラリが見つからなければ、 `None` を返します。

厳密な機能はシステムに依存します。

Linux では、 `find_library()` はライブラリファイルを見つけるために外部プログラム (`/sbin/ldconfig`, `gcc` および `objdump`) を実行しようとします。ライブラリファイルのファイル名を返します。いくつか例があります。:

```
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
'libbz2.so.1.0'
>>>
```

OS X では、 `find_library()` はライブラリの位置を探すために、予め定義された複数の命名方法とパスを試し、成功すればフルパスを返します。:

```
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
>>>
```

Windows では、 `find_library()` はシステムの探索パスに沿って探し、フルパスを返します。しかし、予め定義された命名方法がないため、 `find_library("c")` のような呼び出しは失敗し、 `None` を返します。

`ctypes` で共有ライブラリをラップする場合、 `find_library()` を使って実行時にライブラリの場所を特定するのではなく、共有ライブラリの名前を開発時に決めておいて、ラッパーモジュールにハードコードする方が良いでしょう。

共有ライブラリをロードする

共有ライブラリを Python プロセスへロードする方法はいくつかあります。一つの方法は下記のクラスの一つをインスタンス化することです:

```
class ctypes.CDLL(name, mode=DEFAULT_MODE, handle=None, use_errno=False,
                  use_last_error=False)
```

このクラスのインスタンスはロードされた共有ライブラリをあらわします。これらのライブラリの関数は標準 C 呼び出し規約を使用し、 `int` を返すと仮定されます。

```
class ctypes.OleDLL(name, mode=DEFAULT_MODE, handle=None, use_errno=False,
                    use_last_error=False)
```

Windows 用: このクラスのインスタンスはロードされた共有ライブラリをあらわします。これらのライブラリの関数は `stdcall` 呼び出し規約を使用し、windows 固有の `HRESULT` コードを返すと仮定されます。 `HRESULT` 値には関数呼び出しが失敗したのか成功したのかを特定する情報とともに、補足のエラーコードが含まれます。戻り値が失敗を知らせたならば、`WindowsError` が自動的に送出されます。

```
class ctypes.WinDLL(name, mode=DEFAULT_MODE, handle=None, use_errno=False,
                    use_last_error=False)
```

Windows 用: このクラスのインスタンスはロードされた共有ライブラリをあらわします。これらのライブラリの関数は `stdcall` 呼び出し規約を使用し、デフォルトでは `int` を返すと仮定されます。

Windows CE では標準呼び出し規約だけが使われます。便宜上、このプラットフォームでは、`WinDLL` と `OleDLL` が標準呼び出し規約を使用します。

これらのライブラリがエクスポートするどの関数でも呼び出す前に Python GIL (*global interpreter lock*) は解放され、後でまた獲得されます。

```
class ctypes.PyDLL(name, mode=DEFAULT_MODE, handle=None)
```

Python GIL が関数呼び出しの間解放 されず、関数実行の後に Python エラーフラグがチェックされるということを除けば、このクラスのインスタンスは `CDLL` インスタンスのように振る舞います。エラーフラグがセットされた場合、Python 例外が送出されます。

要するに、これは Python C api 関数を直接呼び出すのに便利だというだけです。

これらすべてのクラスは少なくとも一つの引数、すなわちロードする共有ライブラリのパスを渡して呼び出すことでインスタンス化されます。すでにロード済みの共有ライブラリへのハンドルがあるなら、`handle` 名前付き引数として渡すことができます。土台となっているプラットフォームの `dlopen` または `LoadLibrary` 関数がプロセスヘライブラリをロードするために使われ、そのライブラリに対するハンドルを得ます。

`mode` パラメータを使うと、ライブラリがどうやってロードされたかを特定できます。詳細は `dlopen(3)` マニュアルページを参考にしてください。Windows では `mode` は無視されます。POSIX システムでは `RTLD_NOW` が常に追加され、設定変更はできません。

`use_errno` 変数が真に設定されたとき、システムの `errno` エラーナンバーに安全にアクセスする `ctypes` の仕組みが有効化されます。 `ctypes` はシステムの `errno` 変数のスレッド限定のコピーを管理します。もし、`use_errno=True` の状態で作られた外部関数を呼び出したなら、関数呼び出し前の `errno` 変数は `ctypes` のプライベートコピーと置き換えられ、同じことが関数呼び出しの直後にも発生します。

`ctypes.get_errno()` 関数は `ctypes` のプライベートコピーの値を返します。そして、 `ctypes.set_errno()` 関数は `ctypes` のプライベートコピーを置き換え、以前の値を返します。

`use_last_error` パラメータは、真に設定されたとき、 `GetLastError()` と `SetLastError()` Windows API によって管理される Windows エラーコードに対するのと同じ仕組みが有効化されます。 `ctypes.get_last_error()` と `ctypes.set_last_error()` は Windows エラーコードの `ctypes` プライベートコピーを変更したり要求したりするのに使われます。

バージョン 2.6 で追加: `use_last_error` と `use_errno` オプション変数が追加されました。

ctypes.RTLD_GLOBAL

mode パラメータとして使うフラグ。このフラグが利用できないプラットフォームでは、整数のゼロと定義されています。

ctypes.RTLD_LOCAL

mode パラメータとして使うフラグ。これが利用できないプラットフォームでは、*RTLD_GLOBAL* と同様です。

ctypes.DEFAULT_MODE

共有ライブラリをロードするために使われるデフォルトモード。OSX 10.3 では *RTLD_GLOBAL* であり、そうでなければ *RTLD_LOCAL* と同じです。

これらのクラスのインスタンスには公開メソッドはありません。共有ライブラリからエクスポートされた関数は、属性として、もしくは添字でアクセスできます。属性を通した関数へのアクセスは結果がキャッシュされ、従って繰り返しアクセスされると毎回同じオブジェクトを返すことに注意してください。それとは反対に、添字を通したアクセスは毎回新しいオブジェクトを返します:

```
>>> libc.time == libc.time
True
>>> libc['time'] == libc['time']
False
```

次に述べる公開属性が利用できます。それらの名前はエクスポートされた関数名に衝突しないように下線で始まります。:

PyDLL.handle

ライブラリへのアクセスに用いられるシステムハンドル。

PyDLL.name

コンストラクタに渡されたライブラリの名前。

共有ライブラリは (*LibraryLoader* クラスのインスタンスである) 前もって作られたオブジェクトの一つを使うことによってロードすることもできます。それらの *LoadLibrary()* メソッドを呼び出すか、ローダーインスタンスの属性としてライブラリを取り出すかのどちらかによりロードします。

class ctypes.LibraryLoader (dlltype)

共有ライブラリをロードするクラス。 *dlltype* は *CDLL*、*PyDLL*、*WinDLL* もしくは *OleDLL* 型の一つであるべきです。

__getattr__() は次のような特別なはたらきをします。ライブラリローダーインスタンスの属性として共有ライブラリにアクセスするとそれがロードされるということを可能にします。結果はキャッシュされます。そのため、繰り返し属性アクセスを行うといつも同じライブラリが返されます。

LoadLibrary (name)

共有ライブラリをプロセスへロードし、それを返します。このメソッドはライブラリの新しいインスタンスを常に返します。

これらの前もって作られたライブラリローダーを利用することができます。:

`ctypes.cdll`

CDLL インスタンスを作ります。

`ctypes.windll`

Windows 用: *WinDLL* インスタンスを作ります。

`ctypes.ole32`

Windows 用: *OleDLL* インスタンスを作ります。

`ctypes.pydll`

PyDLL インスタンスを作ります。

C Python api に直接アクセスするために、すぐに使用できる Python 共有ライブラリオブジェクトが用意されています。:

`ctypes.pythonapi`

属性として Python C api 関数を公開する *PyDLL* のインスタンス。これらすべての関数は `C int` を返すと仮定されますが、もちろん常に正しいとは限りません。そのため、これらの関数を使うためには正しい `restype` 属性を代入しなければなりません。

外部関数

前節で説明した通り、外部関数はロードされた共有ライブラリの属性としてアクセスできます。デフォルトではこの方法で作成された関数オブジェクトはどんな数の引数でも受け取り、引数としてどんな `ctypes` データのインスタンスをも受け取り、そして、ライブラリローダーが指定したデフォルトの結果の値の型を返します。関数オブジェクトはプライベートクラスのインスタンスです。:

class `ctypes.FuncPtr`

C の呼び出し可能外部関数のためのベースクラス。

外部関数のインスタンスも C 互換データ型です。それらは C の関数ポインタを表しています。

この振る舞いは外部関数オブジェクトの特別な属性に代入することによって、カスタマイズすることができます。

restype

外部関数の結果の型を指定するために `ctypes` 型を代入する。何も返さない関数を表す `void` に対しては `None` を使います。

`ctypes` 型ではない呼び出し可能な Python オブジェクトを代入することは可能です。このような場合、関数が `C int` を返すと仮定され、呼び出し可能オブジェクトはこの整数を引数に呼び出されます。さらに処理を行ったり、エラーチェックをしたりできるようにするためです。これの使用は推奨されません。より柔軟な後処理やエラーチェックのためには `restype` として `ctypes` 型を使い、`errcheck` 属性へ呼び出し可能オブジェクトを代入してください。

argtypes

関数が受け取る引数の型を指定するために `ctypes` 型のタプルを代入します。`stdcall` 呼び出し規約をつかう関数はこのタプルの長さと同じ数の引数で呼び出されます。その上、C 呼び出し規約をつかう関数は追加の不特定の引数も取ります。

外部関数が呼ばれたとき、それぞれの実引数は `argtypes` タブルの要素の `from_param()` クラスメソッドへ渡されます。このメソッドは実引数を外部関数が受け取るオブジェクトに合わせて変えられるようにします。例えば、`argtypes` タブルの `c_char_p` 要素は、`ctypes` 変換規則にしたがって引数として渡された Unicode 文字列をバイト文字列へ変換します。

新: `ctypes` 型でない要素を `argtypes` に入れることができますが、個々の要素は引数として使える値 (整数、文字列、`ctypes` インスタンス) を返す `from_param()` メソッドを持っていなければなりません。これにより関数パラメータとしてカスタムオブジェクトを適合するように変更できるアダプタが定義可能となります。

errcheck

Python 関数または他の呼び出し可能オブジェクトをこの属性に代入します。呼び出し可能オブジェクトは三つ以上の引数とともに呼び出されます。

callable (*result, func, arguments*)

result は外部関数が返すもので、*restype* 属性で指定されます。

func は外部関数オブジェクト自身で、これにより複数の関数の処理結果をチェックまたは後処理するために、同じ呼び出し可能オブジェクトを再利用できるようになります。

arguments は関数呼び出しに最初に渡されたパラメータが入ったタプルです。これにより使われた引数に基づいた特別な振る舞いをさせることができますようになります。

この関数が返すオブジェクトは外部関数呼び出しから返された値でしょう。しかし、戻り値をチェックして、外部関数呼び出しが失敗しているなら例外を送出させることもできます。

exception `ctypes.ArgumentError`

この例外は外部関数呼び出しが渡された引数を変換できなかったときに送られます。

関数プロトタイプ

外部関数は関数プロトタイプをインスタンス化することによって作成されます。関数プロトタイプは C の関数プロトタイプと似ています。実装を定義せずに、関数 (戻り値、引数の型、呼び出し規約) を記述します。ファクトリ関数は関数に要求する戻り値の型と引数の型とともに呼び出されます。

`ctypes.CFUNCTYPE` (*restype, *argtypes, use_errno=False, use_last_error=False*)

返された関数プロトタイプは標準 C 呼び出し規約をつかう関数を作成します。関数は呼び出されている間 GIL を解放します。 *use_errno* が真に設定されれば、呼び出しの前後で System 変数 `errno` の `ctypes` プライベートコピーは本当の `errno` の値と交換されます。 *use_last_error* も Windows エラーコードに対するのと同様です。

バージョン 2.6 で変更: オプションの *use_errno* と *use_last_error* 変数が追加されました。

`ctypes.WINFUNCTYPE` (*restype, *argtypes, use_errno=False, use_last_error=False*)

Windows 用: 返された関数プロトタイプは `stdcall` 呼び出し規約をつかう関数を作成します。ただし、`WINFUNCTYPE()` が `CFUNCTYPE()` と同じである Windows CE を除きます。関数は呼び出されている間 GIL を解放します。 *use_errno* と *use_last_error* は前述と同じ意味を持ちます。

`ctypes.PYFUNCTYPE` (*restype*, **argtypes*)

返された関数プロトタイプは Python 呼び出し規約をつかう関数を作成します。関数は呼び出されている間 GIL を解放 しません。

ファクトリ関数によって作られた関数プロトタイプは呼び出しのパラメータの型と数に依存した別の方法でインスタンス化することができます。:

prototype (*address*)

指定されたアドレス (整数でなくてはなりません) の外部関数を返します。

prototype (*callable*)

Python の *callable* から C の呼び出し可能関数 (コールバック関数) を作成します。

prototype (*func_spec*[, *paramflags*])

共有ライブラリがエクスポートしている外部関数を返します。 *func_spec* は 2 要素タプル (*name_or_ordinal*, *library*) でなければなりません。第一要素はエクスポートされた関数の名前である文字列、またはエクスポートされた関数の序数である小さい整数です。第二要素は共有ライブラリインスタンスです。

prototype (*vtbl_index*, *name*[, *paramflags*[, *iid*]])

COM メソッドを呼び出す外部関数を返します。 *vtbl_index* は仮想関数テーブルのインデックスで、非負の小さい整数です。 *name* は COM メソッドの名前です。 *iid* はオプションのインターフェイス識別子へのポインタで、拡張されたエラー情報の提供のために使われます。

COM メソッドは特殊な呼び出し規約を用います。このメソッドは *argtypes* タプルに指定されたパラメータに加えて、第一引数として COM インターフェイスへのポインタを必要とします。

オプションの *paramflags* パラメータは上述した機能より多機能な外部関数ラッパーを作成します。

paramflags は *argtypes* と同じ長さのタプルでなければなりません。

このタプルの個々の要素はパラメータについてのより詳細な情報を持ち、1、2 もしくは 3 要素を含むタプルでなければなりません。

第一要素はパラメータについてのフラグの組み合わせを含んだ整数です。

- 1 入力パラメータを関数に指定します。
- 2 出力パラメータ。外部関数が値を書き込みます。
- 4 デフォルトで整数ゼロになる入力パラメータ。

オプションの第二要素はパラメータ名の文字列です。これが指定された場合は、外部関数を名前付きパラメータで呼び出すことができます。

オプションの第三要素はこのパラメータのデフォルト値です。

この例では、デフォルトパラメータと名前付き引数をサポートするために Windows MessageBoxA 関数をラップする方法を示します。 windows ヘッドファイルの C の宣言はこれです。:

```
WINUSERAPI int WINAPI
MessageBoxA(
    HWND hWnd,
    LPCSTR lpText,
    LPCSTR lpCaption,
    UINT uType);
```

`ctypes` を使ってラップします。:

```
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCSTR, LPCSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption", None), (1,
↳ "flags", 0)
>>> MessageBox = prototype(("MessageBoxA", windll.user32), paramflags)
>>>
```

今は `MessageBox` 外部関数をこのような方法で呼び出すことができます。:

```
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
>>>
```

二番目の例は出力パラメータについて説明します。win32 の `GetWindowRect` 関数は、指定されたウィンドウの大きさを呼び出し側が与える `RECT` 構造体へコピーすることで取り出します。C の宣言はこうです。:

```
WINUSERAPI BOOL WINAPI
GetWindowRect(
    HWND hWnd,
    LPRECT lpRect);
```

`ctypes` を使ってラップします。:

```
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(("GetWindowRect", windll.user32), paramflags)
>>>
```

もし単一の値もしくは一つより多い場合には出力パラメータ値が入ったタプルがあるならば、出力パラメータを持つ関数は自動的に出力パラメータ値を返すでしょう。そのため、今は `GetWindowRect` 関数は呼び出されたときに `RECT` インスタンスを返します。

さらに出力処理やエラーチェックを行うために、出力パラメータを `errcheck` プロトコルと組み合わせることができます。win32 `GetWindowRect` api 関数は成功したか失敗したかを知らせるために `BOOL` を返します。そのため、この関数はエラーチェックを行って、api 呼び出しが失敗した場合に例外を送出させることができます。:


```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

`errcheck` 関数が増えなしに受け取った引数タプルを返したならば、`ctypes` は出力パラメータに対して通常の処理を続けます。RECT インスタンスの代わりに window 座標のタプルを返してほしいなら、関数のフィールドを取り出し、代わりにそれらを返すことができます。通常処理はもはや行われません。

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

ユーティリティ関数

`ctypes.addressof(obj)`

メモリバッファのアドレスを示す整数を返します。 *obj* は `ctypes` 型のインスタンスでなければなりません。

`ctypes.alignment(obj_or_type)`

`ctypes` 型のアライメントの必要条件を返します。 *obj_or_type* は `ctypes` 型またはインスタンスでなければなりません。

`ctypes.byref(obj[, offset])`

obj (`ctypes` 型のインスタンスでなければならない) への軽量ポインタを返します。 *offset* はデフォルトでは 0 で、内部ポインタへ加算される整数です。

`byref(obj, offset)` は、C コードとしては、以下のようにみなされます。:

```
((char *)&obj) + offset)
```

返されるオブジェクトは外部関数呼び出しのパラメータとしてのみ使用できます。 `pointer(obj)` と似たふるまいをしますが、作成が非常に速く行えます。

バージョン 2.6 で追加: *offset* オプション引数が追加されました。

`ctypes.cast(obj, type)`

この関数はCのキャスト演算子に似ています。 *obj* と同じメモリブロックを指している *type* の新しいインスタンスを返します。 *type* はポインタ型でなければならず、 *obj* はポインタとして解釈できるオブジェクトでなければなりません。

`ctypes.create_string_buffer (init_or_size[, size])`

この関数は変更可能な文字バッファを作成します。返されるオブジェクトは `c_char` の `ctypes` 配列です。

`init_or_size` は配列のサイズを指定する整数もしくは配列要素を初期化するために使われる文字列である必要があります。

第一引数として文字列が指定された場合は、バッファが文字列の長さより一要素分大きく作られます。配列の最後の要素が NUL 終端文字であるためです。文字列の長さを使うべきでない場合は、配列のサイズを指定するために整数を第二引数として渡すことができます。

第一引数が Unicode 文字列ならば、`ctypes` 変換規則にしたがい 8 ビット文字列へ変換されます。

`ctypes.create_unicode_buffer (init_or_size[, size])`

この関数は変更可能な Unicode 文字バッファを作成します。返されるオブジェクトは `c_wchar` の `ctypes` 配列です。

`init_or_size` は配列のサイズを指定する整数もしくは配列要素を初期化するために使われる Unicode 文字列です。

第一引数として Unicode 文字列が指定された場合は、バッファが文字列の長さより一要素分大きく作られます。それは配列の最後の要素が NUL 終端文字であるためです。文字列の長さを使うべきでない場合は、配列のサイズを指定するために整数を第二引数として渡すことができます。

第一引数が 8 ビット文字列ならば、`ctypes` 変換規則にしたがい Unicode 文字列へ変換されます。

`ctypes.DllCanUnloadNow()`

Windows 用: この関数は `ctypes` をつかってインプロセス COM サーバーを実装できるようにするためのフックです。_ctypes 拡張 dll がエクスポートしている `DllCanUnloadNow` 関数から呼び出されます。

`ctypes.DllGetClassObject()`

Windows 用: この関数は `ctypes` をつかってインプロセス COM サーバーを実装できるようにするためのフックです。_ctypes 拡張 dll がエクスポートしている `DllGetClassObject` 関数から呼び出されます。

`ctypes.util.find_library (name)`

ライブラリを検索し、パス名を返します。 `name` は `lib` のような接頭辞、`.so` や `.dylib` のような接尾辞、そして、バージョンナンバーを除くライブラリ名です (これは `posix` のリンカーオプション `-l` で使われる書式です)。もしライブラリが見つからなければ、`None` を返します。

厳密な機能はシステムに依存します。

バージョン 2.6 で変更: Windows 用: `find_library("m")` もしくは `find_library("c")` は `find_msvcr()` の呼び出し結果を返します。

`ctypes.util.find_msvcr()`

Windows 用: Python と拡張モジュールで使われる VC ランタイムライブラリのファイル名を返します。もしライブラリ名が同定できなければ、`None` を返します。

もし、例えば拡張モジュールにより割り付けられたメモリを `free(void *)` で解放する必要があるなら、メモリ割り付けを行ったのと同じライブラリの関数を使うことが重要です。

バージョン 2.6 で追加.

`ctypes.FormatError([code])`

Windows 用: エラーコード *code* の説明文を返します。エラーコードが指定されない場合は、Windows api 関数 `GetLastError` を呼び出して、もっとも新しいエラーコードが使われます。

`ctypes.GetLastError()`

Windows 用: 呼び出し側のスレッド内で Windows によって設定された最新のエラーコードを返します。この関数は Windows の `GetLastError()` 関数を直接実行します。ctypes のプライベートなエラーコードのコピーを返したりはしません。

`ctypes.get_errno()`

システムの *errno* 変数の、スレッドローカルなプライベートコピーを返します。

バージョン 2.6 で追加.

`ctypes.get_last_error()`

Windows 用: システムの `LastError` 変数の、スレッドローカルなプライベートコピーを返します。

バージョン 2.6 で追加.

`ctypes.memmove(dst, src, count)`

標準 C の `memmove` ライブラリ関数と同じものです。: *count* バイトを *src* から *dst* へコピーします。*dst* と *src* はポインタへ変換可能な整数または ctypes インスタンスでなければなりません。

`ctypes.memset(dst, c, count)`

標準 C の `memset` ライブラリ関数と同じものです。: アドレス *dst* のメモリブロックを値 *c* を *count* バイト分書き込みます。*dst* はアドレスを指定する整数または ctypes インスタンスである必要があります。

`ctypes.POINTER(type)`

このファクトリ関数は新しい ctypes ポインタ型を作成して返します。ポインタ型はキャッシュされ、内部で再利用されます。したがって、この関数を繰り返し呼び出してもコストは小さいです。*type* は ctypes 型でなければなりません。

`ctypes.pointer(obj)`

この関数は *obj* を指す新しいポインタインスタンスを作成します。戻り値は `POINTER(type(obj))` 型のオブジェクトです。

注意: 外部関数呼び出しへオブジェクトへのポインタを渡したいだけなら、はるかに高速な `byref(obj)` を使うべきです。

`ctypes.resize(obj, size)`

この関数は *obj* の内部メモリバッファのサイズを変更します。*obj* は ctypes 型のインスタンスでなければなりません。バッファを `sizeof(type(obj))` で与えられるオブジェクト型の本来のサイズより小さくすることはできませんが、バッファを拡大することはできます。

`ctypes.set_conversion_mode(encoding, errors)`

この関数は 8 ビット文字列と Unicode 文字列の間で変換するときに使われる規則を設定します。*encoding* は 'utf-8' や 'mbcs' のようなエンコーディングを指定する文字列でなければなりません。

ん。 `errors` はエンコーディング/デコーディングエラーについてのエラー処理を指定する文字列でなければなりません。指定可能な値の例としては、`"strict"`, `"replace"`, `"ignore"` があります。

`set_conversion_mode()` は以前の変換規則を含む 2 要素タプルを返します。windows では初期の変換規則は `('mbcs', 'ignore')` であり、他のシステムでは `('ascii', 'strict')` です。

`ctypes.set_errno(value)`

システム変数 `errno` の、呼び出し元スレッドでの `ctypes` のプライベートコピーの現在値を `value` に設定し、前の値を返します。

バージョン 2.6 で追加。

`ctypes.set_last_error(value)`

Windows 用: システム変数 `LastError` の、呼び出し元スレッドでの `ctypes` のプライベートコピーの現在値を `value` に設定し、前の値を返します。

バージョン 2.6 で追加。

`ctypes.sizeof(obj_or_type)`

`ctypes` の型やインスタンスのメモリバッファのサイズをバイト数で返します。C の `sizeof` 演算子と同様の動きをします。

`ctypes.string_at(address[, size])`

この関数はメモリアドレス `address` から始まる文字列を返します。 `size` が指定された場合はサイズとして使われます。指定されなければ、文字列がゼロ終端されていると仮定します。

`ctypes.WinError(code=None, descr=None)`

Windows 用: この関数は `ctypes` の中でもおそらく最悪な名前がつけられたものです。 `WindowsError` のインスタンスを作成します。 `code` が指定されないならば、エラーコードを決めるために `GetLastError` が呼び出されます。 `descr` が指定されないならば、 `FormatError()` がエラーの説明文を得るために呼び出されます。

`ctypes.wstring_at(address[, size])`

この関数は Unicode 文字列としてメモリアドレス `address` から始まるワイドキャラクタ文字列を返します。 `size` が指定されたならば、文字列の文字数として使われます。指定されなければ、文字列がゼロ終端されていると仮定します。

データ型

`class ctypes._CData`

この非公開クラスはすべての `ctypes` データ型の共通のベースクラスです。他のことはさておき、すべての `ctypes` 型インスタンスは C 互換データを保持するメモリブロックを内部に持ちます。このメモリブロックのアドレスは `addressof()` ヘルパー関数が返します。別のインスタンス変数が `_objects` として公開されます。これはメモリブロックがポインタを含む場合に存続し続ける必要のある他の Python オブジェクトを含んでいます。

`ctypes` データ型の共通メソッド、すべてのクラスメソッドが存在します (正確には、[メタクラス](#) のメソッドです):

from.buffer (*source* [, *offset*])

このメソッドは *source* オブジェクトのバッファを共有する ctypes のインスタンスを返します。*source* オブジェクトは書き込み可能バッファインターフェースをサポートしている必要があります。オプションの *offset* 引数では *source* バッファのオフセットをバイト単位で指定します。デフォルトではゼロです。もし *source* バッファが十分に大きくなければ、`ValueError` が送出されます。

バージョン 2.6 で追加.

from.buffer_copy (*source* [, *offset*])

このメソッドは *source* オブジェクトの読み出し可能バッファをコピーすることで、ctypes のインスタンスを生成します。オプションの *offset* 引数では *source* バッファのオフセットをバイト単位で指定します。デフォルトではゼロです。もし *source* バッファが十分に大きくなければ、`ValueError` が送出されます。

バージョン 2.6 で追加.

from.address (*address*)

このメソッドは *address* で指定されたメモリを使って ctypes 型のインスタンスを返します。*address* は整数でなければなりません。

from.param (*obj*)

このメソッドは *obj* を ctypes 型に適合させます。外部関数の `argtypes` タプルに、その型があるとき、外部関数呼び出しで実際に使われるオブジェクトと共に呼び出されます。

すべての ctypes のデータ型は、それが型のインスタンスであれば、*obj* を返すこのクラスメソッドのデフォルトの実装を持ちます。いくつかの型は、別のオブジェクトも受け付けます。

in.dll (*library*, *name*)

このメソッドは、共有ライブラリによってエクスポートされた ctypes 型のインスタンスを返します。*name* はエクスポートされたデータの名前で、*library* はロードされた共有ライブラリです。

ctypes データ型共通のインスタンス変数:

`_b_base_`

ctypes 型データのインスタンスは、それ自身のメモリブロックを持たず、基底オブジェクトのメモリブロックの一部を共有することがあります。`_b_base_` 読み出し専用属性は、メモリブロックを保持する ctypes の基底オブジェクトです。

`_b_needsfree_`

この読み出し専用の変数は、ctypes データインスタンスが、それ自身に割り当てられたメモリブロックを持つとき `true` になります。それ以外の場合は `false` になります。

`_objects`

このメンバは `None`、または、メモリブロックの内容が正しく保つために、生存させておかなくてはならない Python オブジェクトを持つディクショナリです。このオブジェクトはデバッグでのみ使われます。決してディクショナリの内容を変更しないで下さい。

基本データ型

`class ctypes._SimpleCData`

この非公開クラスはすべての基本 ctypes データ型のベースクラスです。ここでこのクラスに触れたのは、基本 ctypes データ型の共通属性を含んでいるからです。 `_SimpleCData` は `_CData` のサブクラスですので、そのメソッドと属性を継承しています。

バージョン 2.6 で変更: ポインタと、ポインタを含まない ctypes データ型が pickle 化できるようになりました。

インスタンスは一つだけ属性を持ちます:

`value`

この属性は、インスタンスの実際の値を持ちます。整数型とポインタ型に対しては整数型、文字型に対しては一文字の文字列、文字へのポインタに対しては Python の文字列もしくは Unicode 文字列となります。

`value` 属性が ctypes インスタンスより参照されたとき、大抵の場合はそれぞれに対し新しいオブジェクトを返します。 `ctypes` はオリジナルのオブジェクトを返す実装にはなって おらず 新しいオブジェクトを構築します。同じことが他の ctypes オブジェクトインスタンスに対しても言えます。

基本データ型は、外部関数呼び出しの結果として返されたときや、例えば構造体のフィールドメンバーや配列要素を取り出すときに、ネイティブの Python 型へ透過的に変換されます。言い換えると、外部関数が `c_char_p` の `restype` を持つ場合は、 `c_char_p` インスタンスではなく 常に Python 文字列を受け取ることでしょう。

基本データ型のサブクラスはこの振る舞いを継承 しません。したがって、外部関数の `restype` が `c_void_p` のサブクラスならば、関数呼び出しからこのサブクラスのインスタンスを受け取ります。もちろん、 `value` 属性にアクセスしてポインタの値を得ることができます。

これらが基本 ctypes データ型です:

`class ctypes.c_byte`

C の signed char データ型を表し、小整数として値を解釈します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

`class ctypes.c_char`

C char データ型を表し、単一の文字として値を解釈します。コンストラクタはオプションの文字列初期化子を受け取り、その文字列の長さちょうど一文字である必要があります。

`class ctypes.c_char_p`

C char * データ型を表し、ゼロ終端文字列へのポインタでなければなりません。バイナリデータを指す可能性のある一般的なポインタに対しては `POINTER(c_char)` を使わなければなりません。コンストラクタは整数のアドレスもしくは文字列を受け取ります。

`class ctypes.c_double`

C double データ型を表します。コンストラクタはオプションの浮動小数点数初期化子を受け取ります。

class ctypes.c_longdouble

C long double データ型を表します。コンストラクタはオプションで浮動小数点数初期化子を受け取ります。sizeof(long double) == sizeof(double) であるプラットフォームでは *c_double* の別名です。

バージョン 2.6 で追加.

class ctypes.c_float

C float データ型を表します。コンストラクタはオプションの浮動小数点数初期化子を受け取ります。

class ctypes.c_int

C signed int データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。sizeof(int) == sizeof(long) であるプラットフォームでは、*c_long* の別名です。

class ctypes.c_int8

C 8-bit signed int データ型を表します。たいていは、*c_byte* の別名です。

class ctypes.c_int16

C 16-bit signed int データ型を表します。たいていは、*c_short* の別名です。

class ctypes.c_int32

C 32-bit signed int データ型を表します。たいていは、*c_int* の別名です。

class ctypes.c_int64

C 64-bit signed int データ型を表します。たいていは、*c_longlong* の別名です。

class ctypes.c_long

C signed long データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class ctypes.c_longlong

C signed long long データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class ctypes.c_short

C signed short データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class ctypes.c_size_t

C size_t データ型を表します。

class ctypes.c_ssize_t

C ssize_t データ型を表します。

バージョン 2.7 で追加.

class ctypes.c_ubyte

C の unsigned char データ型を表し、小さな整数として値を解釈します。コンストラクタはオプションの整数初期化子を受け取ります; オーバーフローのチェックは行われません。

class `ctypes.c_uint`

C の `unsigned int` データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります; オーバーフローのチェックは行われません。これは、`sizeof(int) == sizeof(long)` であるプラットフォームでは `c_ulong` の別名です。

class `ctypes.c_uint8`

C 8-bit `unsigned int` データ型を表します。たいていは、`c_ubyte` の別名です。

class `ctypes.c_uint16`

C 16-bit `unsigned int` データ型を表します。たいていは、`c_ushort` の別名です。

class `ctypes.c_uint32`

C 32-bit `unsigned int` データ型を表します。たいていは、`c_uint` の別名です。

class `ctypes.c_uint64`

C 64-bit `unsigned int` データ型を表します。たいていは、`c_ulonglong` の別名です。

class `ctypes.c_ulong`

C `unsigned long` データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class `ctypes.c_ulonglong`

C `unsigned long long` データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class `ctypes.c_ushort`

C `unsigned short` データ型を表します。コンストラクタはオプションの整数初期化子を受け取ります。オーバーフローのチェックは行われません。

class `ctypes.c_void_p`

C `void *` データ型を表します。値は整数として表されます。コンストラクタはオプションの整数初期化子を受け取ります。

class `ctypes.c_wchar`

C `wchar_t` データ型を表し、値は Unicode 文字列の単一の文字として解釈されます。コンストラクタはオプションの文字列初期化子を受け取り、その文字列の長さはちょうど一文字である必要があります。

class `ctypes.c_wchar_p`

C `wchar_t *` データ型を表し、ゼロ終端ワイド文字列へのポインタでなければなりません。コンストラクタは整数のアドレスもしくは文字列を受け取ります。

class `ctypes.c_bool`

C の `bool` データ型 (より正確には、C99 以降の `_Bool`) を表します。True または False の値を持ち、コンストラクタは真偽値と解釈できるオブジェクトを受け取ります。

バージョン 2.6 で追加。

class `ctypes.HRESULT`

Windows 用: HRESULT 値を表し、関数またはメソッド呼び出しに対する成功またはエラーの情報を含

んでいます。

class `ctypes.py_object`

`C PyObject *` データ型を表します。引数なしでこれ呼び出すと `NULL PyObject *` ポインタを作成します。

`ctypes.wintypes` モジュールは他の Windows 固有のデータ型を提供します。例えば、`HWND`, `WPARAM`, `DWORD` です。 `MSG` や `RECT` のような有用な構造体も定義されています。

構造化データ型

class `ctypes.Union` (**args*, ***kw*)

ネイティブのバイトオーダーでの共用体のための抽象ベースクラス。

class `ctypes.BigEndianStructure` (**args*, ***kw*)

ビッグエンディアン バイトオーダーでの構造体のための抽象ベースクラス。

class `ctypes.LittleEndianStructure` (**args*, ***kw*)

リトルエンディアン バイトオーダーでの構造体のための抽象ベースクラス。

ネイティブではないバイトオーダーを持つ構造体にポインタ型フィールドあるいはポインタ型フィールドを含む他のどんなデータ型をも入れることはできません。

class `ctypes.Structure` (**args*, ***kw*)

ネイティブのバイトオーダーでの構造体のための抽象ベースクラス。

具象構造体型と具象共用体型はこれらの型の一つをサブクラス化することで作らなければなりません。少なくとも、`_fields_` クラス変数を定義する必要があります。 `ctypes` は、属性に直接アクセスしてフィールドを読み書きできるようにする記述子 (*descriptor*) を作成するでしょう。これらは、

`_fields_`

構造体のフィールドを定義するシーケンス。要素は 2 要素タプルか 3 要素タプルでなければなりません。第一要素はフィールドの名前です。第二要素はフィールドの型を指定します。それはどんな `ctypes` データ型でも構いません。

`c_int` のような整数型のために、オプションの第三要素を与えることができます。フィールドのビット幅を定義する正の小整数である必要があります。

一つの構造体と共用体の中で、フィールド名はただ一つである必要があります。これはチェックされません。名前が繰り返してきたときにアクセスできるのは一つのフィールドだけです。

`Structure` サブクラスを定義するクラス文の 後で、 `_fields_` クラス変数を定義することができます。これにより、次のように自身を直接または間接的に参照するデータ型を作成できるようになります:

```
class List(Structure):
    pass
List._fields_ = [("pNext", POINTER(List)),
                 ...
                 ]
```

しかし、`_fields_` クラス変数はその型が最初に使われる（インスタンスが作成される、それに対して `sizeof()` が呼び出されるなど）より前に定義されていなければなりません。その後 `_fields_` クラス変数へ代入すると `AttributeError` が送出されます。

構造体のサブクラスを定義することができ、もしあるならサブクラス内で定義された `_fields_` に加えて、ベースクラスのフィールドも継承します。

`_pack_`

インスタンスの構造体フィールドのアライメントを上書きできるようにするオプションの小整数。`_pack_` は `_fields_` が代入されたときすでに定義されていなければなりません。そうでなければ、何の効果もありません。

`_anonymous_`

無名(匿名) フィールドの名前が並べあげられたオプションのシーケンス。`_fields_` が代入されたとき、`_anonymous_` がすでに定義されていなければなりません。そうでなければ、何ら影響はありません。

この変数に並べあげられたフィールドは構造体もしくは共用体フィールドでなければなりません。構造体フィールドまたは共用体フィールドを作る必要なく、入れ子になったフィールドに直接アクセスできるようにするために、`ctypes` は構造体の中に記述子を作成します。

型の例です (Windows):

```
class _U(Union):
    _fields_ = [("lptdesc", POINTER(TYPEDESC)),
                ("lpadesc", POINTER(ARRAYDESC)),
                ("hreftype", HREFTYPE)]

class TYPEDESC(Structure):
    _anonymous_ = ("u",)
    _fields_ = [("u", _U),
                ("vt", VARTYPE)]
```

TYPEDESC 構造体は COM データ型を表現しており、`vt` フィールドは共用体フィールドのどれが有効であるかを指定します。`u` フィールドは匿名フィールドとして定義されているため、TYPEDESC インスタンスから取り除かれてそのメンバーへ直接アクセスできます。`td.lptdesc` と `td.u.lptdesc` は同等ですが、前者がより高速です。なぜなら一時的な共用体インスタンスを作る必要がないためです。:

```
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

構造体のサブ-サブクラスを定義することができ、ベースクラスのフィールドを継承します。サブクラス定義に別の `_fields_` 変数がある場合は、この中で指定されたフィールドはベースクラスのフィールドへ追加されます。

構造体と共用体のコンストラクタは位置引数とキーワード引数の両方を受け取ります。位置引数は `_fields_` の中に現れたのと同じ順番でメンバーフィールドを初期化するために使われます。コンスト

ラクタのキーワード引数は属性代入として解釈され、そのため、同じ名前をもつ `_fields_` を初期化するか、`_fields_` に存在しない名前に対しては新しい属性を作ります。

配列とポインタ

class `ctypes.Array(*args)`

配列のための抽象基底クラスです。

具象配列型を作成するための推奨される方法は、任意の `ctypes` データ型に正の整数を乗算することです。代わりに、この型のサブクラスを作成し、`_length_` と `_type_` のクラス変数を定義することもできます。配列の要素は、標準の添え字とスライスによるアクセスを使用して読み書きを行うことができます。スライスの読み込みでは、結果のオブジェクト自体は `Array` ではありません。

`_length_`

配列の要素数を指定する正の整数。範囲外の添え字を指定すると、`IndexError` が送出されます。`len()` がこの整数を返します。

`_type_`

配列内の各要素の型を指定します。

配列のサブクラスのコンストラクタは、位置引数を受け付けて、配列を順番に初期化するために使用します。

class `ctypes._Pointer`

ポインタのためのプライベートな抽象基底クラスです。

具象ポインタ型は、ポイント先の型を持つ `POINTER()` を呼び出すことで、作成できます。これは、`pointer()` により自動的に行われます。

ポインタが配列を指す場合、その配列の要素は、標準の添え字とスライスによるアクセスを使用して読み書きが行えます。ポインタオブジェクトには、サイズがないため、`len()` 関数は `TypeError` を送出します。負の添え字は、(C と同様に) ポインタの 前 のメモリから読み込み、範囲外の添え字はおそらく (幸運な場合でも) アクセス違反によりクラッシュを起こします。

`_type_`

ポイント先の型を指定します。

contents

ポインタが指すオブジェクトを返します。この属性に割り当てると、ポインタが割り当てられたオブジェクトを指すようになります。

第 16 章

オプションのオペレーティングシステム サービス

この章で説明するモジュールでは、特定のオペレーティングシステムでだけ利用できるオペレーティングシステム機能へのインタフェースを提供します。このインタフェースは、おおむね Unix や C のインタフェースにならってモデル化してありますが、他のシステム上（Windows や NT など）でも利用できることがあります。次に概要を示します：

16.1 `select` — I/O 処理の完了を待機する

このモジュールでは、ほとんどのオペレーティングシステムで利用可能な `select()` および `poll()` 関数、Linux 2.5+ で利用可能な `epoll()`、多くの BSD で利用可能な `kqueue()` 関数に対するアクセスを提供しています。Windows 上ではソケットに対してしか動作しないので注意してください；その他のオペレーティングシステムでは、他のファイル形式でも（特に Unix ではパイプにも）動作します。通常のファイルに対して適用し、最後にファイルを読み出した時から内容が増えているかを決定するために使うことはできません。

以下はこのモジュールの定義：

exception `select.error`

エラーが発生したときに送出される例外です。エラーに付属する値は、`errno` からとったエラーコードを表す数値とそのエラーコードに対応する文字列（C 関数の `perror()` の出力相当のもの）からなるペアです。

`select.epoll([sizehint=-1])`

（Linux 2.5.44 以降でのみサポート）エッジポーリング（edge polling）オブジェクトを返します。このオブジェクトは、I/O イベントのエッジトリガもしくはレベルトリガインタフェースとして使うことができます。エッジポーリングオブジェクトが提供しているメソッドについては、[エッジおよびレベルトリガポーリング（*epoll*）オブジェクト](#) 節を参照してください。

バージョン 2.6 で追加。

`select.poll()`

（全てのオペレーティングシステムでサポートされているわけではありません）ポーリングオブジェクトを返します。このオブジェクトはファイル記述子を登録したり登録解除したりすることができ、ファイ

ル記述子に対する I/O イベント発生をポーリングすることができます; ポーリングオブジェクトが提供しているメソッドについては [ポーリングオブジェクト](#) 節を参照してください。

`select.kqueue()`

(BSD でのみサポート) カーネルキュー (kernel queue) オブジェクトを返します。カーネルキューオブジェクトが提供しているメソッドについては、[kqueue オブジェクト](#) 節を参照してください。

バージョン 2.6 で追加.

`select.kevent(ident, filter=KQ_FILTER_READ, flags=KQ_EV_ADD, fflags=0, data=0, udata=0)`

(BSD でのみサポート) カーネルイベント (kernel event) オブジェクトを返します。カーネルイベントオブジェクトが提供しているメソッドについては、[kevent オブジェクト](#) 節を参照してください。

バージョン 2.6 で追加.

`select.select(rlist, wlist, xlist[, timeout])`

Unix の `select()` システムコールに対する直接的なインタフェースです。最初の 3 つの引数は '待機可能オブジェクト' からなるシーケンスです: 待機可能オブジェクトとは、ファイル記述子を表す整数値か、そのような整数を返す引数なしメソッド `fileno()` を持つオブジェクトです。

- *rlist*: 読み込み可能になるまで待機
- *wlist*: 書き込み可能になるまで待機
- *xlist*: "例外状態 (exceptional condition)" になるまで待機 ("例外状態" については、システムのマニュアルページを参照してください)

引数に空のシーケンスを指定してもかまいませんが、3 つの引数全てを空のシーケンスにしてもよいかどうかはプラットフォームに依存します (Unix では動作し、Windows では動作しないことが知られています)。オプションの *timeout* 引数にはタイムアウトまでの秒数を浮動小数点数で指定します。 *timeout* 引数が省略された場合、関数は少なくとも一つのファイル記述子が何らかの準備完了状態になるまでブロックします。 *timeout* に 0 を指定した場合は、ポーリングを行いブロックしないことを示します。

戻り値は準備完了状態のオブジェクトからなる 3 つのリストです: したがってこのリストはそれぞれ関数の最初の 3 つの引数のサブセットになります。ファイル記述子のいずれも準備完了にならないままタイムアウトした場合、3 つの空のリストが返されます。

シーケンスの中に含めることのできるオブジェクトは Python ファイルオブジェクト (例えば `sys.stdin` や、`open()` または `os.popen()` が返すオブジェクト)、`socket.socket()` が返すソケットオブジェクトです。ラッパー (wrapper) クラスを自分で定義することもできます。この場合、適切な (まったくデタラメな数ではなく本物のファイル記述子を返す) `fileno()` メソッドを持つ必要があります。

注釈: `select()` は Windows のファイルオブジェクトを受理しませんが、ソケットは受理します。Windows では、背後の `select()` 関数は WinSock ライブラリで提供されており、WinSock によって生成されたものではないファイル記述子を扱うことができないのです。

`select.PIPE_BUF`

このモジュールの `select()` や `poll()` などのインタフェースにより書き込み可能になったと報告されたファイルは、`PIPE_BUF` バイトまではブロックしないで書き込みできることが保証されます。この値は POSIX では最低でも 512 であることが保証されています。 利用可能な環境: Unix

バージョン 2.7 で追加。

16.1.1 エッジおよびレベルトリガポーリング (epoll) オブジェクト

<http://linux.die.net/man/4/epoll>

eventmask

定数	意味
<code>EPOLLIN</code>	読み込み可能
<code>EPOLLOUT</code>	書き込み可能
<code>EPOLLPRI</code>	緊急の読み出しデータ
<code>EPOLLERR</code>	設定された fd にエラー状態が発生した
<code>EPOLLHUP</code>	設定された fd がハングアップした
<code>EPOLLET</code>	エッジトリガ動作に設定する。デフォルトではレベルトリガ動作
<code>EPOLLONESHOT</code>	1 ショット動作に設定する。1 回イベントが取り出されたら、その fd が内部で無効になる
<code>EPOLLRDNORM</code>	<code>EPOLLIN</code> と同じ
<code>EPOLLRDBAND</code>	priority data band を読み込める。
<code>EPOLLWRNORM</code>	<code>EPOLLOUT</code> と同じ
<code>EPOLLWRBAND</code>	priority data に書き込みできる。
<code>EPOLLMMSG</code>	無視される。

`epoll.close()`

epoll オブジェクトの制御用ファイル記述子を閉じます。

`epoll.fileno()`

制御用ファイル記述子の番号を返します。

`epoll.fromfd(fd)`

fd から epoll オブジェクトを作成します。

`epoll.register(fd[, eventmask])`

epoll オブジェクトにファイル記述子 fd を登録します。

注釈: **ポーリングオブジェクト** の `register` とは異なり、登録済みのファイル記述子を登録しようとする
と `IOError` が発生します。

`epoll.modify(fd, eventmask)`

ファイル記述子 fd の登録を変更する。

`epoll.unregister(fd)`

`epoll` オブジェクトから登録されたファイル記述子 *fd* を削除します。

`epoll.poll([timeout=-1 [, maxevents=-1]])`

イベントを待機します。*timeout* はタイムアウト時間で、単位は秒 (float 型) です。

16.1.2 ポーリングオブジェクト

`poll()` システムコールはほとんどの Unix システムでサポートされており、非常に多数のクライアントに同時にサービスを提供するようなネットワークサーバが高いスケーラビリティを持てるようにしています。`poll()` は対象のファイル記述子を列挙するだけでよいため、良くスケールします。一方、`select()` はビット対応表を構築し、対象ファイルの記述子に対応するビットを立て、その後全ての対応表の全てのビットを線形探索します。`select()` は $O(\text{最大のファイル記述子番号})$ なのに対し、`poll()` は $O(\text{対象とするファイル記述子の数})$ で済みます。

`poll.register(fd [, eventmask])`

ファイル記述子をポーリングオブジェクトに登録します。これ以降の `poll()` メソッド呼び出しでは、そのファイル記述子に処理待ち中の I/O イベントがあるかどうかを監視します。*fd* は整数か、整数値を返す `fileno()` メソッドを持つオブジェクトを取ります。ファイルオブジェクトも `fileno()` を実装しているので、引数として使うことができます。

eventmask はオプションのビットマスクで、どの種類の I/O イベントを監視したいかを記述します。この値は以下の表で述べる定数 `POLLIN`、`POLLPRI`、および `POLLOUT` の組み合わせにすることができます。ビットマスクを指定しない場合、標準の値が使われ、3 種類のイベント全てに対して監視が行われます。

定数	意味
<code>POLLIN</code>	読み出し可能なデータが存在する
<code>POLLPRI</code>	緊急の読み出し可能なデータが存在する
<code>POLLOUT</code>	書き出しの準備ができています: 書き出し処理がブロックしない
<code>POLLERR</code>	何らかのエラー状態
<code>POLLHUP</code>	ハングアップ
<code>POLLNVAL</code>	無効な要求: 記述子が開かれていない

登録済みのファイル記述子を登録してもエラーにはならず、一度だけ登録した場合と同じ効果になります。

`poll.modify(fd, eventmask)`

登録されているファイル記述子 *fd* を変更する。これは、`register(fd, eventmask)` と同じ効果を持ちます。登録されていないファイル記述子に対してこのメソッドを呼び出すと、`errno.ENOENT` で `IOError` 例外が発生します。

バージョン 2.6 で追加。

`poll.unregister(fd)`

ポーリングオブジェクトによって追跡中のファイル記述子を登録解除します。 `register()` メソッドと同様に、`fd` は整数か、整数値を返す `fileno()` メソッドを持つオブジェクトを取ります。

登録されていないファイル記述子を登録解除しようとすると `KeyError` 例外が送出されます。

`poll.poll([timeout])`

登録されたファイル記述子に対してポーリングを行い、報告すべき I/O イベントまたはエラーの発生したファイル記述子毎に 2 要素のタプル (`fd`, `event`) からなるリストを返します。リストは空になることもあります。`fd` はファイル記述子で、`event` は該当するファイル記述子について報告されたイベントを表すビットマスクです — 例えば `POLLIN` は入力待ちを示し、`POLLOUT` はファイル記述子に対する書き込みが可能を示す、などです。空のリストは呼び出しがタイムアウトしたか、報告すべきイベントがどのファイル記述子でも発生しなかったことを示します。`timeout` が与えられた場合、処理を戻すまで待機する時間の長さをミリ秒単位で指定します。`timeout` が省略されたり、負の値であったり、あるいは `None` の場合、そのポーリングオブジェクトが監視している何らかのイベントが発生するまでブロックします。

16.1.3 kqueue オブジェクト

`kqueue.close()`

`kqueue` オブジェクトの制御用ファイル記述子を閉じる。

`kqueue.fileno()`

制御用ファイル記述子の番号を返します。

`kqueue.fromfd(fd)`

与えられたファイル記述子から、`kqueue` オブジェクトを作成する。

`kqueue.control(changelist, max_events[, timeout])` → `eventlist`

`kevent` に対する低水準のインタフェース

- `changelist` must be an iterable of `kevent` objects or `None`
- `max_events` は 0 または正の整数
- `timeout` in seconds (floats possible); the default is `None`, to wait forever

16.1.4 kevent オブジェクト

<https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2>

`kevent.ident`

イベントを特定するための値。この値は、フィルタにもよりますが、大抵の場合はファイル記述子です。コンストラクタでは、`ident` として、整数値か `fileno()` メソッドを持ったオブジェクトを渡せます。`kevent` は内部で整数値を保存します。

`kevent.filter`

カーネルフィルタの名前。

定数	意味
KQ_FILTER_READ	記述子を受け取り、読み込めるデータが存在する時に戻る
KQ_FILTER_WRITE	記述子を受け取り、書き込み可能な時に戻る
KQ_FILTER_AIO	AIO リクエスト
KQ_FILTER_VNODE	<i>fflag</i> で監視されたイベントが 1 つ以上発生したときに戻る
KQ_FILTER_PROC	プロセス ID 上のイベントを監視する
KQ_FILTER_NETDEV	ネットワークデバイス上のイベントを監視する (Mac OS X では利用不可)
KQ_FILTER_SIGNAL	監視しているシグナルがプロセスに届いたときに戻る
KQ_FILTER_TIMER	任意のタイマを設定する

`kevent.flags`

フィルタアクション。

定数	意味
KQ_EV_ADD	イベントを追加または修正する
KQ_EV_DELETE	キューからイベントを取り除く
KQ_EV_ENABLE	<code>control()</code> がイベントを返すのを許可する
KQ_EV_DISABLE	イベントを無効にする
KQ_EV_ONESHOT	イベントを最初の発生後無効にする
KQ_EV_CLEAR	イベントを受け取った後で状態をリセットする
KQ_EV_SYSFLAGS	内部イベント
KQ_EV_FLAG1	内部イベント
KQ_EV_EOF	フィルタ依存の EOF 状態
KQ_EV_ERROR	戻り値を参照

`kevent.fflags`

フィルタ依存のフラグ。

KQ_FILTER_READ と KQ_FILTER_WRITE フィルタのフラグ:

定数	意味
KQ_NOTE_LOWAT	ソケットバッファの最低基準値

KQ_FILTER_VNODE フィルタのフラグ:

定数	意味
KQ_NOTE_DELETE	<i>unlink()</i> が呼ばれた
KQ_NOTE_WRITE	書き込みが発生した
KQ_NOTE_EXTEND	ファイルのサイズが拡張された
KQ_NOTE_ATTRIB	属性が変更された
KQ_NOTE_LINK	リンクカウントが変更された
KQ_NOTE_RENAME	ファイル名が変更された
KQ_NOTE_REVOKE	ファイルアクセスが破棄された

KQ_FILTER_PROC フィルタフラグ:

定数	意味
KQ_NOTE_EXIT	プロセスが終了した
KQ_NOTE_FORK	プロセスが <i>fork()</i> を呼び出した
KQ_NOTE_EXEC	プロセスが新しいプロセスを実行した
KQ_NOTE_PCTRLMASK	内部フィルタフラグ
KQ_NOTE_PDATAMASK	内部フィルタフラグ
KQ_NOTE_TRACK	<i>fork()</i> の呼び出しを超えてプロセスを監視する
KQ_NOTE_CHILD	<i>NOTE_TRACK</i> に対して子プロセスに渡される
KQ_NOTE_TRACKERR	子プロセスにアタッチできなかった

KQ_FILTER_NETDEV フィルタフラグ (Mac OS X では利用不可):

定数	意味
KQ_NOTE_LINKUP	リンクアップしている
KQ_NOTE_LINKDOWN	リンクダウンしている
KQ_NOTE_LINKINV	リンク状態が不正

`kevent.data`

フィルタ固有のデータ。

`kevent.udata`

ユーザー定義値。

16.2 threading — 高水準のスレッドインタフェース

ソースコード: [Lib/threading.py](#)

このモジュールでは、高水準のスレッドインタフェースをより低水準な *thread* モジュールの上に構築しています。 *mutex* と *Queue* モジュールのドキュメントも参照下さい。

また、`thread` がないために `threading` を使えないような状況向けに `dummy_threading` を提供しています。

注釈: Python 2.6 からこのモジュールは Java のスレッディング API の影響を受けた camelCase のプロパティを置き換える **PEP 8** に準拠したエイリアスを提供します。この更新された API は `multiprocessing` モジュールのものと互換です。しかしながら、camelCase の名称の廃止の予定は決まっておらず、Python 2.x と 3.x の両方でサポートされ続けます。

注釈: Python 2.5 から、幾つかの Thread のメソッドは間違った呼び出しに対して `AssertionError` の代わりに `RuntimeError` を返します。

CPython は *Global Interpreter Lock* のため、ある時点で Python コードを実行できるスレッドは 1 つに限られます (ただし、いくつかのパフォーマンスが強く求められるライブラリはこの制限を克服しています)。アプリケーションにマルチコアマシンの計算能力をより良く利用させたい場合は、`multiprocessing` モジュールの利用をお勧めします。ただし、I/O バウンドなタスクを並行して複数走らせたい場合においては、マルチスレッドは正しい選択肢です。

このモジュールでは以下のような関数とオブジェクトを定義しています:

`threading.active_count()`

`threading.activeCount()`

生存中の `Thread` オブジェクトの数を返します。この数は `enumerate()` の返すリストの長さと同じです。

バージョン 2.6 で変更: 新たに `active_count()` として使えるようになりました。

`threading.Condition()`

新しい条件変数 (condition variable) オブジェクトを返すファクトリ関数です。条件変数を使うと、ある複数のスレッドを別のスレッドの通知があるまで待機させられます。

Condition オブジェクトを参照してください。

`threading.current_thread()`

`threading.currentThread()`

関数を呼び出している処理のスレッドに対応する `Thread` オブジェクトを返します。関数を呼び出している処理のスレッドが `threading` モジュールで生成したものでない場合、限定的な機能しかもたないダミースレッドオブジェクトを返します。

バージョン 2.6 で変更: 新たに `current_thread()` として使えるようになりました。

`threading.enumerate()`

現在、生存中の `Thread` オブジェクト全てのリストを返します。リストには、デーモンスレッド (daemon thread)、`current_thread()` の生成するダミースレッドオブジェクト、そして主スレッドが入ります。終了したスレッドとまだ開始していないスレッドは入りません。

`threading.Event()`

新たなイベントオブジェクトを返すファクトリ関数です。イベントは `set()` メソッドを使うと `True` に、`clear()` メソッドを使うと `False` にセットされるようなフラグを管理します。`wait()` メソッドは、全てのフラグが真になるまでブロックするようになっています。

[Event オブジェクト](#) を参照してください。

class `threading.local`

スレッドローカルデータ (thread-local data) を表現するためのクラスです。スレッドローカルデータとは、値が各スレッド固有になるようなデータです。スレッドローカルデータを管理するには、`local` (または `local` のサブクラス) のインスタンスを作成して、その属性に値を代入します

```
mydata = threading.local()
mydata.x = 1
```

インスタンスの値はスレッドごとに違った値になります。

詳細と例題については、`_threading_local` モジュールのドキュメンテーション文字列を参照してください。

バージョン 2.4 で追加。

`threading.Lock()`

新しいプリミティブロック (primitive lock) オブジェクトを返すファクトリ関数です。スレッドが一度プリミティブロックを獲得すると、それ以後のロック獲得の試みはロックが解放されるまでブロックします。どのスレッドでもロックを解放できます。

[Lock オブジェクト](#) を参照してください。

`threading.RLock()`

新しい再入可能ロックオブジェクトを返すファクトリ関数です。再入可能ロックはそれを獲得したスレッドによって解放されなければなりません。いったんスレッドが再入可能ロックを獲得すると、同じスレッドはブロックされずにもう一度それを獲得できます；そのスレッドは獲得した回数だけ解放しなければいけません。

[RLock オブジェクト](#) を参照してください。

`threading.Semaphore([value])`

新しいセマフォ (semaphore) オブジェクトを返すファクトリ関数です。セマフォは、`release()` を呼び出した数から `acquire()` を呼び出した数を引き、初期値を足した値を表すカウンタを管理します。`acquire()` メソッドは、カウンタの値を負にせずに処理を戻せるまで必要ならば処理をブロックします。`value` を指定しない場合、デフォルトの値は 1 になります。

[Semaphore オブジェクト](#) を参照してください。

`threading.BoundedSemaphore([value])`

新しい有限セマフォ (bounded semaphore) オブジェクトを返すファクトリ関数です。有限セマフォは、現在の値が初期値を超過しないようチェックを行います。超過を起こした場合、`ValueError` を送出します。たいていの場合、セマフォは限られた容量のリソースを保護するために使われるものです。

従って、あまりにも頻繁なセマフォの解放はバグが生じているしるしです。 *value* を指定しない場合、デフォルトの値は 1 になります。

class `threading.Thread`

処理中のスレッドを表すクラスです。このクラスは制限のある範囲内で安全にサブクラス化できます。

Thread オブジェクト を参照してください。

class `threading.Timer`

指定時間経過後に関数を実行するスレッドです。

Timer オブジェクト を参照してください。

`threading.settrace(func)`

threading モジュールを使って開始した全てのスレッドにトレース関数を設定します。 *func* は各スレッドの *run()* を呼び出す前にスレッドの *sys.settrace()* に渡されます。

バージョン 2.3 で追加。

`threading.setprofile(func)`

threading モジュールを使って開始した全てのスレッドにプロファイル関数を設定します。 *func* は各スレッドの *run()* を呼び出す前にスレッドの *sys.setprofile()* に渡されます。

バージョン 2.3 で追加。

`threading.stack_size([size])`

新しいスレッドを作るときのスレッドスタックサイズを返します。オプションの *size* 引数は、これ以降に作成するスレッドのスタックサイズを指定するもので、0 (プラットフォームが設定されたデフォルト値) か、32,768 (32 KiB) 以上の正の整数である必要があります。 *size* を指定しない場合、0 が使われます。スレッドのスタックサイズの変更がサポートされていない場合、 *ThreadError* を発生させます。不正なスタックサイズが指定された場合、 *ValueError* を発生させて、スタックサイズを変更しません。32 KiB が現在のインタープリター自体のために十分であると保証された最小のスタックサイズです。いくつかのプラットフォームではスタックサイズに対して制限があることに注意してください。例えば最小のスタックサイズが 32 KiB より大きかったり、システムのメモリページサイズの整数倍の必要があるなどです。この制限についてはプラットフォームのドキュメントを参照する必要があります。(一般的なページサイズは 4 KiB なので、プラットフォームに関する情報がない場合は 4096 の整数倍のスタックサイズを選ぶといいかもしれません)。利用可能な環境: Windows, POSIX スレッドに対応したシステム。

バージョン 2.5 で追加。

exception `threading.ThreadError`

下記で記述するスレッド関連の様々なエラーで送出されます。多くのインターフェイスが *ThreadError* ではなく *RuntimeError* を使っていることに注意してください。

オブジェクトの詳細なインターフェイスを以下に説明します。

このモジュールのおおまかな設計は Java のスレッドモデルに基づいています。とはいえ、Java がロックと条件変数を全てのオブジェクトの基本的な挙動にしているのに対し、Python ではこれらを別個のオブジェクトに分けています。Python の *Thread* クラスがサポートしているのは Java の *Thread* クラスの挙動のサブ

セットにすぎません; 現状では、優先度 (priority) やスレッドグループがなく、スレッドの破壊 (destroy)、中断 (stop)、一時停止 (suspend)、復帰 (resume)、割り込み (interrupt) は行えません。Java の Thread クラスにおける静的メソッドに対応する機能が実装されている場合にはモジュールレベルの関数になっています。

以下に説明するメソッドは全て原子的 (atomic) に実行されます。

16.2.1 Thread オブジェクト

このクラスは個別のスレッド中で実行される活動 (activity) を表現します。活動を決める方法は 2 つあり、一つは呼出し可能オブジェクトをコンストラクタへ渡す方法、もう一つはサブクラスで `run()` メソッドをオーバーライドする方法です。(コンストラクタを除く) その他のメソッドは一切サブクラスでオーバーライドしてはなりません。言い換えるならば、このクラスの `__init__()` と `run()` メソッドだけをオーバーライドしてくださいということです。

ひとたびスレッドオブジェクトを生成すると、スレッドの `start()` メソッドを呼び出して活動を開始せねばなりません。 `start()` メソッドはそれぞれのスレッドの `run()` メソッドを起動します。

スレッドの活動が始まると、スレッドは '生存中 (alive)' とみなされます。スレッドは通常 `run()` メソッドが終了するまで生存中となります。もしくは、捕捉されない例外が送出されるまでです。 `is_alive()` メソッドはスレッドが生存中であるかどうか調べます。

他のスレッドはスレッドの `join()` メソッドを呼び出せます。このメソッドは、 `join()` を呼び出されたスレッドが終了するまで、メソッドの呼び出し手となるスレッドをブロックします。

スレッドには名前があります。名前はコンストラクタに渡したり、または、 `name` 属性を通して読み出したり、変更したりできます。

スレッドには "デーモンスレッド (daemon thread)" であるというフラグを立てられます。このフラグには、残っているスレッドがデーモンスレッドだけになった時に Python プログラム全体を終了させるという意味があります。フラグの初期値はスレッドを生成する側のスレッドから継承します。フラグの値は `daemon` 属性を通して設定できます。

注釈: デーモンスレッドは終了時にいきなり停止されます。デーモンスレッドで使われたリソース (開いているファイル、データベースのトランザクションなど) は適切に解放されないかもしれません。きちんと (gracefully) スレッドを停止したい場合は、スレッドを非デーモンスレッドにして、 `Event` のような適切なシグナル送信機構を使用してください。

スレッドには "主スレッド (main thread)" オブジェクトがあります。主スレッドは Python プログラムを最初に制御していたスレッドです。主スレッドはデーモンスレッドではありません。

"ダミースレッド (dummy thread)" オブジェクトを作成できる場合があります。ダミースレッドは、 "外来スレッド (alien thread)" に相当するスレッドオブジェクトです。ダミースレッドは、C コードから直接生成されたスレッドのような、 `threading` モジュールの外で開始された処理スレッドです。ダミースレッドオブジェクトには限られた機能しかなく、常に生存中、かつデーモンスレッドであるとみなされ、 `join()` できません。また、外来スレッドの終了を検出するのは不可能なので、ダミースレッドは削除できません。

class `threading.Thread` (*group=None, target=None, name=None, args=(), kwargs={}*)

コンストラクタは常にキーワード引数を使って呼び出さなければなりません。各引数は以下の通りです:

group は `None` でなければなりません。将来 `ThreadGroup` クラスが実装されたときの拡張用に予約されている引数です。

target は `run()` メソッドによって起動される呼び出し可能オブジェクトです。デフォルトでは何も呼び出さないことを示す `None` になっています。

name はスレッドの名前です。デフォルトでは、*N* を小さな 10 進数として、`"Thread- N"` という形式の一意的な名前を生成します。

args は *target* を呼び出すときの引数タプルです。デフォルトは `()` です。

kwargs は *target* を呼び出すときのキーワード引数の辞書です。デフォルトは `{}` です。

サブクラスでコンストラクタをオーバーライドした場合、必ずスレッドが何かを始める前に基底クラスのコンストラクタ (`Thread.__init__()`) を呼び出しておかなくてはなりません。

start()

スレッドの活動を開始します。

このメソッドは、スレッドオブジェクトあたり一度しか呼び出してはなりません。 `start()` は、オブジェクトの `run()` メソッドが個別の処理スレッド中で呼び出されるように調整します。

同じスレッドオブジェクトに対し、このメソッドを 2 回以上呼び出した場合、 `RuntimeError` を送出します。

run()

スレッドの活動をもたらすメソッドです。

このメソッドはサブクラスでオーバーライドできます。標準の `run()` メソッドでは、オブジェクトのコンストラクタの *target* 引数に呼び出し可能オブジェクトを指定した場合、 *args* および *kwargs* の引数列およびキーワード引数とともに呼び出します。

join([*timeout*])

スレッドが終了するまで待機します。このメソッドは、 `join()` を呼び出されたスレッドが、正常終了あるいは処理されない例外によって終了するか、オプションのタイムアウトが発生するまで、メソッドの呼び出し手となるスレッドをブロックします。

timeout 引数を指定して、 `None` 以外の値にする場合、タイムアウトを秒 (または端数秒) を表す浮動小数点数でなければなりません。 `join()` はいつでも `None` を返すので、 `isAlive()` を呼び出してタイムアウトしたかどうかを確認しなければなりません。もしスレッドがまだ生存中であれば、 `join()` はタイムアウトしています。

timeout が指定されないかまたは `None` であるときは、この操作はスレッドが終了するまでブロックします。

一つのスレッドに対して何度でも `join()` できます。

実行中のスレッドに対し、`join()` を呼び出そうとすると、デッドロックを引き起こすため、`RuntimeError` が送出されます。スレッドが開始される前に `join()` を呼び出そうとしても、同じ例外が送出されます。

name

識別のためにのみ用いられる文字列です。名前には機能上の意味づけ (semantics) はありません。複数のスレッドに同じ名前をつけてもかまいません。名前の初期値はコンストラクタで設定されます。

バージョン 2.6 で追加.

getName()**setName()**

`name` の、Python 2.6 以前の API です。

ident

‘スレッド識別子’、または、スレッドが開始されていなければ `None` です。非ゼロの整数です。`thread.get_ident()` 関数を参照下さい。スレッド識別子は、スレッドが終了した後、新たなスレッドが生成された場合、再利用され得ます。スレッド識別子は、スレッドが終了した後も利用できます。

バージョン 2.6 で追加.

is_alive()**isAlive()**

スレッドが生存中かどうかを返します。

このメソッドは `run()` メソッドが起動した直後からその `run()` メソッドが終了するまでの間 `True` を返します。モジュール関数、`enumerate()` は、全ての生存中のスレッドのリストを返します。

バージョン 2.6 で変更: 新たに `is_alive()` として使えるようになりました。

daemon

スレッドのデーモンフラグです。このフラグは `start()` の呼び出し前に設定されなければなりません。さもなくば、`RuntimeError` が送出されます。初期値は生成側のスレッドから継承されます。メインスレッドはデーモンスレッドではないので、メインスレッドから生成するスレッドはデフォルトで `daemon=False` です。

デーモンでない生存中のスレッドが全てなくなると、Python プログラム全体が終了します。

バージョン 2.6 で追加.

isDaemon()**setDaemon()**

`daemon` の、Python 2.6 以前の API です。

16.2.2 Lock オブジェクト

プリミティブロックとは、ロックが生じた際に特定のスレッドによって所有されない同期プリミティブです。Python では現在のところ拡張モジュール `thread` で直接実装されている最も低水準の同期プリミティブを使えます。

プリミティブロックは2つの状態、“ロック”または“アンロック”があります。このロックはアンロック状態で作成されます。ロックには基本となる二つのメソッド、`acquire()` と `release()` があります。ロックの状態がアンロックである場合、`acquire()` は状態をロックに変更して即座に処理を戻します。状態がロックの場合、`acquire()` は他のスレッドが `release()` を呼出してロックの状態をアンロックに変更するまでブロックします。その後、状態をロックに再度設定してから処理を戻します。`release()` メソッドを呼び出すのはロック状態のときでなければなりません; このメソッドはロックの状態をアンロックに変更し、即座に処理を戻します。アンロックの状態のロックを解放しようとする `ThreadError` が送出されます。

複数のスレッドにおいて `acquire()` がアンロック状態への遷移を待っているためにブロックが起きている時に `release()` を呼び出してロックの状態をアンロックにすると、一つのスレッドだけが処理を進行できます。どのスレッドが処理を進行できるのかは定義されておらず、実装によって異なるかもしれません。

全てのメソッドはアトミックに実行されます。

`Lock.acquire([blocking])`

ブロックあり、またはブロックなしでロックを獲得します。

引数 `blocking` を `True` (デフォルト) に設定して呼び出した場合、ロックがアンロック状態になるまでブロックします。そしてそれをロック状態にしてから `True` を返します。

引数 `blocking` の値を `False` にして呼び出すとブロックしません。`blocking` を `True` にして呼び出した場合にブロックするような状況では、直ちに `False` を返します。それ以外の場合には、ロックをロック状態にして `True` を返します。

`Lock.release()`

ロックを解放します。

ロックの状態がロックのとき、状態をアンロックにリセットして処理を戻します。他のスレッドがロックがアンロック状態になるのを待ってブロックしている場合、ただ一つのスレッドだけが処理を継続できるようにします。

アンロック状態のロックに対して起動された場合、`ThreadError` が送出されます。

戻り値はありません。

`locked()`

Return true if the lock is acquired.

16.2.3 RLock オブジェクト

再入可能ロック (reentrant lock) とは、同じスレッドが複数回獲得できるような同期プリミティブです。再入可能ロックの内部では、プリミティブロックの使うロック / アンロック状態に加え、“所有スレッド (owning

thread)”と”再帰レベル (recursion level)”という概念を用いています。ロック状態では何らかのスレッドがロックを所有しており、アンロック状態ではいかなるスレッドもロックを所有していません。

スレッドがこのロックの状態をロックにするには、ロックの `acquire()` メソッドを呼び出します。このメソッドは、スレッドがロックを所有すると処理を戻します。ロックの状態をアンロックにするには `release()` メソッドを呼び出します。 `acquire()` / `release()` からなるペアの呼び出しはネストできます; 最後に呼び出した `release()` (最も外側の呼び出しペア) だけが、ロックの状態をアンロックにリセットし、 `acquire()` でブロック中の別のスレッドの処理を進行させられます。

`RLock.acquire([blocking=1])`

ブロックあり、またはブロックなしでロックを獲得します。

引数なしで呼び出した場合: スレッドが既にロックを所有している場合、再帰レベルをインクリメントして即座に処理を戻します。それ以外の場合、他のスレッドがロックを所有していれば、そのロックの状態がアンロックになるまでブロックします。その後、ロックの状態がアンロックになる (いかなるスレッドもロックを所有しない状態になる) と、ロックの所有権を獲得し、再帰レベルを 1 にセットして処理を戻します。ロックの状態がアンロックになるのを待っているスレッドが複数ある場合、その中の一つだけがロックの所有権を獲得できます。この場合、戻り値はありません。

引数 `blocking` の値を `true` にして呼び出した場合、引数なしで呼び出したときと同じことを行ない、`true` を返します。

引数 `blocking` の値を `false` にして呼び出すとブロックしません。引数なしで呼び出した場合にブロックするような状況であった場合には直ちに `false` を返します。それ以外の場合には、引数なしで呼び出したときと同じ処理を行い `true` を返します。

`RLock.release()`

再帰レベルをデクリメントしてロックを解放します。デクリメント後に再帰レベルがゼロになった場合、ロックの状態をアンロック (いかなるスレッドにも所有されていない状態) にリセットし、ロックの状態がアンロックになるのを待ってブロックしているスレッドがある場合にはその中のただ一つだけが処理を進行できるようにします。デクリメント後も再帰レベルがゼロでない場合、ロックの状態はロックのままで、呼び出し側のスレッドに所有されたままになります。

呼び出し側のスレッドがロックを所有しているときにのみこのメソッドを呼び出してください。ロックの状態がアンロックの時にこのメソッドを呼び出すと、 `RuntimeError` が送出されます。

戻り値はありません。

16.2.4 Condition オブジェクト

条件変数 (condition variable) は常にある種のロックに関連付けられています; 条件変数に関連付けるロックは明示的に引き渡したり、デフォルトで生成させたりできます。(複数の条件変数で同じロックを共有するような場合には、引渡しによる関連付けが便利です。)

条件変数には、 `acquire()` メソッドおよび `release()` があり、関連付けされているロックの対応するメソッドを呼び出すようになっています。また、 `wait()`, `notify()`, `notifyAll()` といったメソッドがあります。これら三つのメソッドを呼び出せるのは、呼び出し手のスレッドがロックを獲得している時だけで

す。そうでない場合は `RuntimeError` が送出されます。

`wait()` メソッドは現在のスレッドのロックを解放し、他のスレッドが同じ条件変数に対して `notify()` または `notifyAll()` を呼び出して現在のスレッドを起こすまでブロックします。一度起こされると、再度ロックを獲得して処理を戻します。 `wait()` にはタイムアウトも設定できます。

`notify()` メソッドは条件変数待ちのスレッドを 1 つ起こします。 `notifyAll()` メソッドは条件変数待ちの全てのスレッドを起こします。

注意: `notify()` と `notifyAll()` はロックを解放しません; 従って、スレッドが起こされたとき、 `wait()` の呼び出しは即座に処理を戻すわけではなく、 `notify()` または `notifyAll()` を呼び出したスレッドが最終的にロックの所有権を放棄したときに初めて処理を返すのです。

豆知識: 条件変数を使う典型的なプログラミングスタイルでは、何らかの共有された状態変数へのアクセスを同期させるためにロックを使います; 状態変数が特定の状態に変化したことを知りたいスレッドは、自分の望む状態になるまで繰り返し `wait()` を呼び出します。その一方で、状態変更を行うスレッドは、前者のスレッドが待ち望んでいる状態であるかもしれないような状態へ変更を行ったときに `notify()` や `notifyAll()` を呼び出します。例えば、以下のコードは無制限のバッファ容量のときの一般的な生産者-消費者問題です:

```
# Consume one item
cv.acquire()
while not an_item_is_available():
    cv.wait()
get_an_available_item()
cv.release()

# Produce one item
cv.acquire()
make_an_item_available()
cv.notify()
cv.release()
```

`notify()` と `notifyAll()` のどちらを使うかは、その状態の変化に興味を持っている待ちスレッドが一つだけなのか、あるいは複数なのかで考えます。例えば、典型的な生産者-消費者問題では、バッファに 1 つの要素を加えた場合には消費者スレッドを 1 つしか起こさなくてかまいません。

class `threading.Condition` (`[lock]`)

`lock` を指定して、 `None` の値にする場合、 `Lock` または `RLock` オブジェクトでなければなりません。この場合、 `lock` は根底にあるロックオブジェクトとして使われます。それ以外の場合には新しい `RLock` オブジェクトを生成して使います。

acquire (`*args`)

根底にあるロックを獲得します。このメソッドは根底にあるロックの対応するメソッドを呼び出します。そのメソッドの戻り値を返します。

release ()

根底にあるロックを解放します。このメソッドは根底にあるロックの対応するメソッドを呼び出します。戻り値はありません。

wait (`[timeout]`)

通知 (`notify`) を受けるか、タイムアウトするまで待機します。呼び出し側のスレッドがロックを獲得していないときにこのメソッドを呼び出すと `RuntimeError` が送出されます。

このメソッドは根底にあるロックを解放し、他のスレッドが同じ条件変数に対して `notify()` または `notifyAll()` を呼び出して現在のスレッドを起こすか、オプションのタイムアウトが発生するまでブロックします。一度スレッドが起こされると、再度ロックを獲得して処理を戻します。

`timeout` 引数を指定して、`None` 以外の値にする場合、タイムアウトを秒 (または端数秒) を表す浮動小数点数でなければなりません。

根底にあるロックが `RLock` である場合、`release()` メソッドではロックは解放されません。というのも、ロックが再帰的に複数回獲得されている場合には、`release()` によって実際にアンロックが行われないかもしれないからです。その代わり、ロックが再帰的に複数回獲得されていても確実にアンロックを行える `RLock` クラスの内部インタフェースを使います。その後ロックを再獲得する時に、もう一つの内部インタフェースを使ってロックの再帰レベルを復帰します。

`notify(n=1)`

デフォルトで、この条件変数を待っている 1 つのスレッドを起こします。呼び出し側のスレッドがロックを獲得していないときにこのメソッドを呼び出すと `RuntimeError` が送出されます。

何らかの待機中スレッドがある場合、そのうち `n` スレッドを起こします。待機中のスレッドがなければ何もしません。

現在の実装では、少なくとも `n` スレッドが待機中であれば、ちょうど `n` スレッドを起こします。とはいえ、この挙動に依存するのは安全ではありません。将来、実装の最適化によって、複数のスレッドを起こすようになるかもしれないからです。

注意: 起こされたスレッドは実際にロックを再獲得できるまで `wait()` 呼び出しから戻りません。`notify()` はロックを解放しないので、`notify()` 呼び出し側は明示的にロックを解放しなければなりません。

`notify_all()`

`notifyAll()`

この条件を待っているすべてのスレッドを起こします。このメソッドは `notify()` のように動作しますが、1 つではなくすべての待ちスレッドを起こします。呼び出し側のスレッドがロックを獲得していない場合、`RuntimeError` が送出されます。

バージョン 2.6 で変更: 新たに `notify_all()` として使えるようになりました。

16.2.5 Semaphore オブジェクト

セマフォ (semaphore) は、計算機科学史上最も古い同期プリミティブの一つで、草創期のオランダ計算機科学者 Edsger W. Dijkstra によって発明されました (彼は `acquire()` と `release()` の代わりに `P()` と `V()` を使いました)。

セマフォは `acquire()` でデクリメントされ `release()` でインクリメントされるような内部カウンタを管理します。カウンタは決してゼロより小さくはありません; `acquire()` は、カウンタがゼロになっている場合、他のスレッドが `release()` を呼び出すまでブロックします。

class `threading.Semaphore` (`[value]`)

オプションの引数には、内部カウンタの初期値を指定します。デフォルトは 1 です。与えられた *value* が 0 より小さい場合、`ValueError` が送出されます。

acquire (`[blocking]`)

セマフォを獲得します。

引数なしで呼び出した場合: `acquire()` 処理に入ったときに内部カウンタがゼロより大きければ、カウンタを 1 デクリメントして即座に処理を戻します。 `acquire()` 処理に入ったときに内部カウンタがゼロの場合、他のスレッドが `release()` を呼び出してカウンタをゼロより大きくするまでブロックします。この処理は、適切なインターロック (interlock) を介して行い、複数の `acquire()` 呼び出しがブロックされた場合、 `release()` が正確に一つだけを起こせるようにします。この実装はランダムに一つ選択するだけでもよいので、ブロックされたスレッドがどの起こされる順番に依存してはなりません。この場合、戻り値はありません。

blocking 引数の値を真にした場合、引数なしで呼び出した場合と同じ処理を行って真を返します。

blocking を `false` にして呼び出すとブロックしません。引数なしで呼び出した場合にブロックするような状況であった場合には直ちに `false` を返します。それ以外の場合には、引数なしで呼び出したときと同じ処理を行い `true` を返します。

release ()

内部カウンタを 1 インクリメントして、セマフォを解放します。 `release()` 処理に入ったときにカウンタがゼロであり、カウンタの値がゼロより大きくなるのを待っている別のスレッドがあった場合、そのスレッドを起こします。

Semaphore の例

セマフォはしばしば、容量に限りのある資源、例えばデータベースサーバなどを保護するために使われます。リソースが固定の状況では、常に有限セマフォを使わなければなりません。主スレッドは、作業スレッドを立ち上げる前にセマフォを初期化します:

```
maxconnections = 5
...
pool_sema = BoundedSemaphore(value=maxconnections)
```

作業スレッドは、ひとたび立ち上がると、サーバへ接続する必要があるときにセマフォの `acquire()` および `release()` メソッドを呼び出します:

```
pool_sema.acquire()
conn = connectdb()
... use connection ...
conn.close()
pool_sema.release()
```

有限セマフォを使うと、セマフォを獲得回数以上に解放してしまうというプログラム上の間違いを見逃しにくくします。

16.2.6 Event オブジェクト

イベントは、あるスレッドがイベントを発信し、他のスレッドはそれを待つという、スレッド間で通信を行うための最も単純なメカニズムの一つです。

イベントオブジェクトは内部フラグを管理します。このフラグは `set()` メソッドで値を `true` に、`clear()` メソッドで値を `false` にリセットします。`wait()` メソッドはフラグが `true` になるまでブロックします。

class `threading.Event`

内部フラグの初期値は偽です。

is_set()

isSet()

内部フラグの値が `true` である場合にのみ `true` を返します。

バージョン 2.6 で変更: 新たに `is_set()` として使えるようになりました。

set()

内部フラグの値を `true` にセットします。フラグの値が `true` になるのを待っている全てのスレッドを起こします。一旦フラグが `true` になると、スレッドが `wait()` を呼び出しても全くブロックしなくなります。

clear()

内部フラグの値を `false` にリセットします。以降は、`set()` を呼び出して再び内部フラグの値を `true` にセットするまで、`wait()` を呼び出したスレッドはブロックするようになります。

wait([timeout])

内部フラグの値が `true` になるまでブロックします。`wait()` 処理に入った時点で内部フラグの値が `true` であれば、直ちに処理を戻します。そうでない場合、他のスレッドが `set()` を呼び出してフラグの値を `true` にセットするか、オプションのタイムアウトが発生するまでブロックします。

`timeout` 引数を指定して、`None` 以外の値にする場合、タイムアウトを秒 (または端数秒) を表す浮動小数点数でなければなりません。

このメソッドは終了時の内部フラグを返します。`timeout` が指定されて、操作がタイムアウトしたとき以外は、`True` を返すはずですが、

バージョン 2.7 で変更: 以前は、このメソッドは常に `None` を返していました。

16.2.7 Timer オブジェクト

このクラスは、一定時間経過後に実行される活動、すなわちタイマ活動を表現します。`Timer` は `Thread` のサブクラスであり、自作のスレッドを構築した一例でもあります。

タイマは `start()` メソッドを呼び出すとスレッドとして作動し始めします。(活動を開始する前に) `cancel()` メソッドを呼び出すと、タイマを停止できます。タイマが活動を実行するまでの待ち時間は、ユーザが指定した待ち時間と必ずしも厳密には一致しません。

例えば:

```
def hello():
    print "hello, world"

t = Timer(30.0, hello)
t.start()  # after 30 seconds, "hello, world" will be printed
```

class `threading.Timer` (*interval*, *function*, *args*=[], *kwargs*={})

interval 秒後に *function* を引数 *args*、キーワード引数 *kwargs* つきで実行するようなタイマを生成します。

cancel()

タイマをストップして、その動作の実行をキャンセルします。このメソッドはタイマがまだ活動待ち状態にある場合にのみ動作します。

16.2.8 with 文でのロック・条件変数・セマフォの使い方

このモジュールのオブジェクトで `acquire()` と `release()` 両メソッドを具えているものは全て `with` 文のコンテキストマネージャとして使うことができます。 `acquire()` メソッドが `with` 文のブロックに入るときに呼び出され、ブロック脱出時には `release()` メソッドが呼ばれます。

現在のところ、`Lock`、`RLock`、`Condition`、`Semaphore`、`BoundedSemaphore` を `with` 文のコンテキストマネージャとして使うことができます。以下の例を見てください。

```
import threading

some_rlock = threading.RLock()

with some_rlock:
    print "some_rlock is locked while this executes"
```

16.2.9 スレッド化されたコード中での Import

スレッドセーフな `import` のためには、継承の制限に起因する、ふたつの重要な制約があります。

- ひとつ目は、主とするモジュール以外では、`import` が新しいスレッドを生成しないようになっていなければならない。そして、そのスレッドを待たなければならない。この制約を守らない場合、生成されたスレッドが直接的、または、間接的にモジュールを `import` しようとした際に、デッドロックを引き起こす可能性があります。
- ふたつ目は、全ての `import` が、インタープリターが自身を終了させる前に完了しなければならない。これは、最も簡単な方法としては、`threading` モジュールを通して生成される非デーモンからのみ `import` を実行することで達成できます。デーモンスレッド、および、直接、`thread` モジュールから生成されたスレッドは、インタープリター終了後に `import` を実行しないようにする、別の同期の仕組みを必要とします。この制約を守らない場合、`intermittent` (間歇) 例外を引き起こし、インタープリターのシャットダウン中にクラッシュする可能性があります。(後から実行される `import` は、すでにアクセス可能でなくなった領域にアクセスしようとするためです)

16.3 thread — マルチスレッドのコントロール

注釈: Python 3 では `thread` モジュールは `_thread` に改名されました。2to3 ツールは、3 コードへの変換時に、自動的に `import` 宣言を適合させます。しかしながら、上位の `threading` モジュールを使うことを検討して下さい。

このモジュールはマルチスレッド (別名 軽量プロセス (*light-weight processes*) または タスク (*tasks*)) に用いられる低レベルプリミティブを提供します — グローバルデータ空間を共有するマルチスレッドを制御します。同期のための単純なロック (別名 *mutexes* または バイナリセマフォ (*binary semaphores*)) が提供されています。`threading` モジュールは、このモジュール上で、より使い易く高級なスレッディングの API を提供します。

このモジュールはオプションです。Windows, Linux, SGI IRIX, Solaris 2.x、そして同じような POSIX スレッド (別名 "pthread") 実装のシステム上でサポートされます。`thread` を使用することのできないシステムでは、`dummy_thread` が用意されています。`dummy_thread` はこのモジュールと同じインターフェースを持ち、置き換えて使用することができます。

定数と関数は以下のように定義されています:

exception `thread.error`

スレッド特有の例外です。

`thread.LockType`

これはロックオブジェクトのタイプです。

`thread.start_new_thread(function, args[, kwargs])`

新しいスレッドを開始して、その ID を返します。スレッドは引数リスト `args` (タプルでなければなりません) の関数 `function` を実行します。オプション引数 `kwargs` はキーワード引数の辞書を指定します。関数が戻るとき、スレッドは黙って終了します。関数が未定義の例外でターミネートしたとき、スタックトレースが表示され、そしてスレッドが終了します (しかし他のスレッドは走り続けます)。

`thread.interrupt_main()`

メインスレッドで `KeyboardInterrupt` を送出します。サブスレッドはこの関数を使ってメインスレッドに割り込みをかけることができます。

バージョン 2.3 で追加。

`thread.exit()`

`SystemExit` を送出します。それが捕えられないときは、黙ってスレッドを終了させます。

`thread.allocate_lock()`

新しいロックオブジェクトを返します。ロックのメソッドはこの後に記述されます。ロックは初期状態としてアンロック状態です。

`thread.get_ident()`

現在のスレッドの 'スレッド ID' を返します。非ゼロの整数です。この値は直接の意味を持っていません; 例えばスレッド特有のデータの辞書に索引をつけるためのような、マジッククッキーとして意図さ

れています。スレッドが終了し、他のスレッドが作られたとき、スレッド ID は再利用されるかもしれませんが。

`thread.stack_size([size])`

新しいスレッドが作られる際に使われるスレッドのスタックサイズを返します。オプションの *size* 引数は次に作られるスレッドに対するスタックサイズを指定するものですが、0 (プラットフォームまたは設定されたデフォルト) または少なくとも 32,768 (32kB) であるような正の整数でなければなりません。 *size* に指定がなければ 0 が使われます。もしスタックサイズの変更がサポートされていなければ `ThreadError` が送出されます。また指定されたスタックサイズが条件を満たしていなければ `ValueError` が送出されスタックサイズは変更されないままになります。32kB は今のところインタプリタ自体に十分なスタックスペースを保証するための値としてサポートされる最小のスタックサイズです。プラットフォームによってはスタックサイズの値に固有の制限が課されることもあります。たとえば 32kB より大きな最小スタックサイズを要求されたり、システムメモリサイズの倍数の割り当てを要求されるなどです - より詳しい情報はプラットフォームごとの文書で確認してください (4kB ページは一般的ですので、情報が見当たらないときには 4096 の倍数を指定しておくといいかもしれません)。利用可能: Windows, POSIX スレッドのあるシステム。

バージョン 2.5 で追加。

ロックオブジェクトは次のようなメソッドを持っています:

`lock.acquire([waitflag])`

オプションの引数なしで使用すると、このメソッドは他のスレッドがロックしているかどうかにかかわらずロックを獲得します。ただし、他のスレッドがすでにロックしている場合には解除されるまで待つからロックを獲得します (同時にロックを獲得できるスレッドはひとつだけであり、これこそがロックの存在理由です)。整数の引数 *waitflag* を指定すると、その値によって動作が変わります。引数が 0 のときは、待たずにすぐ獲得できる場合にだけロックを獲得します。0 以外の値を与えると、先の例と同様、ロックの状態にかかわらず獲得をおこないます。なお、ロックを獲得すると `True`、できなかったときには `False` を返します。

`lock.release()`

ロックを解放します。そのロックは既に獲得されたものでなければなりませんが、しかし同じスレッドによって獲得されたものである必要はありません。

`lock.locked()`

ロックの状態を返します: 同じスレッドによって獲得されたものなら `True`、違うのなら `False` を返します。

これらのメソッドに加えて、ロックオブジェクトは `with` 文を通じて以下の例のように使うこともできます。

```
import thread

a_lock = thread.allocate_lock()

with a_lock:
    print "a_lock is locked while this executes"
```

警告:

- スレッドは割り込みと奇妙な相互作用をします: `KeyboardInterrupt` 例外は任意のスレッドによって受け取られます。(`signal` モジュールが利用可能なとき、割り込みは常にメインスレッドへ行きます。)
- `sys.exit()` を呼び出す、あるいは `SystemExit` 例外を送出することは、 `thread.exit()` を呼び出すことと同じです。
- ロックの `acquire()` メソッドに割り込むことはできません — `KeyboardInterrupt` 例外は、ロックが獲得された後に発生します。
- メインスレッドが終了したとき、他のスレッドが生き残るかどうかは、システムに依存します。ネイティブスレッド実装を使う SGI, IRIX では生き残ります。その他の多くのシステムでは、 `try ... finally` 節や、オブジェクトデストラクタを実行せずに終了されます。
- メインスレッドが終了したとき、その通常のクリーンアップは行なわれず、(`try ... finally` 節が尊重されることは除きます)、標準 I/O ファイルはフラッシュされません。

16.4 dummy_threading — threading の代替モジュール

ソースコード: [Lib/dummy_threading.py](#)

このモジュールは `threading` モジュールのインターフェースをそっくりまねるものです。 `thread` モジュールがサポートされていないプラットフォームで `import` することを意図して作られたものです。

おすすめの使い道は:

```
try:
    import threading as _threading
except ImportError:
    import dummy_threading as _threading
```

生成するスレッドが他のブロックしたスレッドを待ち、デッドロック発生の可能性がある場合には、このモジュールを使わないようにしてください。ブロッキング I/O を使っている場合によく起きます。

16.5 dummy_thread — thread の代替モジュール

注釈: `dummy_thread` モジュールは、Python 3 では `_dummy_thread` に変更されました。 `2to3` ツールは自動的にソースコードの `import` を修正します。しかし、代わりに高レベルの `dummy_threading` モジュールの利用を検討するべきです。

ソースコード: [Lib/dummy_thread.py](#)

このモジュールは `thread` モジュールのインターフェースをそっくりまねるものです。 `thread` モジュールがサポートされていないプラットフォームで `import` することを意図して作られたものです。

おすすめの使い道は:

```
try:
    import thread as _thread
except ImportError:
    import dummy_thread as _thread
```

生成するスレッドが他のブロックしたスレッドを待ち、デッドロック発生の可能性がある場合には、このモジュールを使わないようにしてください。ブロッキング I/O を使っている場合によく起きます。

16.6 multiprocessing — プロセスベースの ”並列処理” インタフェース

バージョン 2.6 で追加.

16.6.1 はじめに

`multiprocessing` は、`threading` と似た API で複数のプロセスを生成をサポートするパッケージです。 `multiprocessing` パッケージは、ローカルとリモート両方の並行処理を提供します。また、このパッケージはスレッドの代わりにサブプロセスを使用することにより、[グローバルインタープリタロック](#) の問題を避ける工夫が行われています。このような特徴があるため `multiprocessing` モジュールを使うことで、マルチプロセッサマシンの性能を最大限に活用することができるようでしょう。なお、このモジュールは Unix と Windows で動作します。

`multiprocessing` モジュールでは `threading` モジュールには似たものがない API も導入しています。その最たるものが `Pool` オブジェクトです。これは複数の入力データに対して、サブプロセス群に入力データを分配して並列に関数実行する (データ並列) のに便利な手段を提供します。以下の例では、モジュール内で関数を定義するのに、子プロセスがそのモジュールをつつがなくインポート出来るようにする一般的な慣例を使っています。 `Pool` を用いたデータ並列の基礎的な実例はこのようなものです:

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    p = Pool(5)
    print(p.map(f, [1, 2, 3]))
```

標準出力に以下が出力されます:

```
[1, 4, 9]
```


Process クラス

`multiprocessing` モジュールでは、プロセスは以下の手順によって生成されます。はじめに `Process` のオブジェクトを作成し、続いて `start()` メソッドを呼び出します。この `Process` クラスは `threading.Thread` クラスと同様の API を持っています。まずは、簡単な例をもとにマルチプロセスを使用したプログラムについてみていきましょう

```
from multiprocessing import Process

def f(name):
    print 'hello', name

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

実行された個々のプロセス ID を表示するために拡張したサンプルコードを以下に例を示します:

```
from multiprocessing import Process
import os

def info(title):
    print title
    print 'module name:', __name__
    if hasattr(os, 'getppid'): # only available on Unix
        print 'parent process:', os.getppid()
    print 'process id:', os.getpid()

def f(name):
    info('function f')
    print 'hello', name

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

(Windows 環境で) `if __name__ == '__main__':` という記述が必要な理由については、[プログラミングガイドライン](#) を参照してください。

プロセス間でのオブジェクト交換

`multiprocessing` モジュールでは、プロセス間通信の手段が 2 つ用意されています。それぞれ以下に詳細を示します:

キュー (Queue)

`Queue` クラスは `Queue.Queue` クラスとほとんど同じように使うことができます。以下に例を示します。

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print q.get()      # prints "[42, None, 'hello']"
    p.join()
```

キューはスレッドセーフであり、プロセスセーフです。

パイプ (Pipe)

Pipe() 関数はパイプで繋がれたコネクションオブジェクトのペアを返します。デフォルトでは双方向性パイプを返します。以下に例を示します:

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print parent_conn.recv()  # prints "[42, None, 'hello']"
    p.join()
```

パイプのそれぞれの端を表す 2 つのコネクションオブジェクトが *Pipe()* 関数から返されます。各コネクションオブジェクトには、*send()*、*recv()*、その他のメソッドがあります。2 つのプロセス (またはスレッド) がパイプの 同じ 端で同時に読み込みや書き込みを行うと、パイプ内のデータが破損してしまうかもしれないことに注意してください。もちろん、各プロセスがパイプの別々の端を同時に使用するならば、データが破壊される危険性はありません。

プロセス間の同期

multiprocessing は *threading* モジュールと等価な同期プリミティブを備えています。以下の例では、ロックを使用して、一度に 1 つのプロセスしか標準出力に書き込まないようにしています:

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    print 'hello world', i
    l.release()
```

(次のページに続く)

(前のページからの続き)

```

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()

```

ロックを使用しないで標準出力に書き込んだ場合は、各プロセスからの出力がごちゃまぜになってしまいます。

プロセス間での状態の共有

これまでの話の流れで触れたとおり、並行プログラミングを行うときには、できるかぎり状態を共有しないのが定石です。複数のプロセスを使用するときは特にそうでしょう。

しかし、どうしてもプロセス間のデータ共有が必要な場合のために *multiprocessing* モジュールには 2 つの方法が用意されています。

共有メモリ (Shared memory)

データを共有メモリ上に保持するために *Value* クラス、もしくは *Array* クラスを使用することができます。以下のサンプルコードを使って、この機能についてみていきましょう

```

from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print num.value
    print arr[:]

```

このサンプルコードを実行すると以下のように表示されます

```

3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]

```

num と *arr* を生成するときに使用されている、引数 'd' と 'i' は *array* モジュールにより使用される種別の型コードです。ここで使用されている 'd' は倍精度浮動小数、'i' は符号付整数を表します。これらの共有オブジェクトは、プロセスセーフでありスレッドセーフです。

共有メモリを使用して、さらに柔軟なプログラミングを行うには `multiprocessing.sharedctypes` モジュールを使用します。このモジュールは共有メモリから割り当てられた任意の ctypes オブジェクトの生成をサポートします。

サーバープロセス (Server process)

`Manager()` 関数により生成されたマネージャーオブジェクトはサーバープロセスを管理します。マネージャーオブジェクトは Python のオブジェクトを保持して、他のプロセスがプロキシ経由でその Python オブジェクトを操作することができます。

`Manager()` 関数が返すマネージャは `list`、`dict`、`Namespace`、`Lock`、`RLock`、`Semaphore`、`BoundedSemaphore`、`Condition`、`Event`、`Queue`、`Value`、`Array` をサポートします。以下にサンプルコードを示します。

```
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    manager = Manager()

    d = manager.dict()
    l = manager.list(range(10))

    p = Process(target=f, args=(d, l))
    p.start()
    p.join()

    print d
    print l
```

このサンプルコードを実行すると以下のように表示されます

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

サーバープロセスのマネージャーオブジェクトは共有メモリのオブジェクトよりも柔軟であるといえます。それは、どのような型のオブジェクトでも使えるからです。また、1つのマネージャーオブジェクトはネットワーク経由で他のコンピューター上のプロセスによって共有することもできます。しかし、共有メモリより動作が遅いという欠点があります。

ワーカープロセスのプールを使用

`Pool` クラスは、ワーカープロセスをプールする機能を備えています。このクラスには、異なる方法でワーカープロセスへタスクを割り当てるいくつかのメソッドがあります。

例えば:

```
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x

if __name__ == '__main__':
    pool = Pool(processes=4)           # start 4 worker processes

    # print "[0, 1, 4, ..., 81]"
    print pool.map(f, range(10))

    # print same numbers in arbitrary order
    for i in pool.imap_unordered(f, range(10)):
        print i

    # evaluate "f(20)" asynchronously
    res = pool.apply_async(f, (20,))   # runs in *only* one process
    print res.get(timeout=1)           # prints "400"

    # evaluate "os.getpid()" asynchronously
    res = pool.apply_async(os.getpid, ()) # runs in *only* one process
    print res.get(timeout=1)           # prints the PID of that process

    # launching multiple evaluations asynchronously *may* use more processes
    multiple_results = [pool.apply_async(os.getpid, ()) for i in range(4)]
    print [res.get(timeout=1) for res in multiple_results]

    # make a single worker sleep for 10 secs
    res = pool.apply_async(time.sleep, (10,))
    try:
        print res.get(timeout=1)
    except TimeoutError:
        print "We lacked patience and got a multiprocessing.TimeoutError"
```

プールオブジェクトのメソッドは、そのプールを作成したプロセスのみが呼び出すべきです。

注釈: このパッケージに含まれる機能を使用するためには、子プロセスから `__main__` モジュールをインポートできる必要があります。このことについては [プログラミングガイドライン](#) で触れていますが、ここであらためて強調しておきます。なぜかという、いくつかのサンプルコード、例えば `multiprocessing.Pool.Pool` のサンプルはインタラクティブシェル上では動作しないからです。以下に例を示します:

```
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> p.map(f, [1,2,3])
```

(次のページに続く)

(前のページからの続き)

```
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
```

(このサンプルを試すと、3つのトレースバックすべてがほぼランダムに交互に重なって表示されます。そうになったら、なんとかしてマスタープロセスを止めましょう。)

16.6.2 リファレンス

multiprocessing パッケージは *threading* モジュールの API とほとんど同じです。

Process クラスと例外

```
class multiprocessing.Process (group=None, target=None, name=None, args=(),
                                kwargs={})
```

Process オブジェクトは各プロセスの処理を表します。Process クラスは *threading.Thread* クラスのすべてのメソッドと同じインタフェースを提供します。

コンストラクタは必ずキーワード引数で呼び出すべきです。引数 *group* には必ず `None` を渡してください。この引数は *threading.Thread* クラスとの互換性のためだけに残されています。引数 *target* には、呼び出し可能オブジェクト (Callable Object) を渡します。このオブジェクトは *run()* メソッドから呼び出されます。この引数はデフォルトで `None` となっており、何も呼び出されません。引数 *name* にはプロセス名を渡します。デフォルトでは、自動でユニークな名前が割り当てられます。命名規則は、`'Process-N1:N2:...:Nk'` となります。ここで N_1, N_2, \dots, N_k は整数の数列で、作成したプロセス数に対応します。引数 *args* は *target* で指定された呼び出し可能オブジェクトへの引数を渡します。同じく、引数 *kwargs* はキーワード引数を渡します。デフォルトでは、*target* には引数が渡されないようになっています。

サブクラスがコンストラクターをオーバーライドする場合は、そのプロセスに対する処理を行う前に基底クラスのコンストラクター (*Process.__init__()*) を実行しなければなりません。

run()

プロセスが実行する処理を表すメソッドです。

このメソッドはサブクラスでオーバーライドすることができます。標準の *run()* メソッドは、コンストラクターの *target* 引数として渡された呼び出し可能オブジェクトを呼び出します。もしコンストラクターに *args* もしくは *kwargs* 引数が渡されていれば、呼び出すオブジェクトにこれらの引数を渡します。

start()

プロセスの処理を開始するためのメソッドです。

各 Process オブジェクトに対し、このメソッドが 2 回以上呼び出されてはいけません。各プロセスでオブジェクトの `run()` メソッドを呼び出す準備を行います。

`join([timeout])`

`join()` されたプロセスが `terminate` を呼び出すまで、もしくはオプションで指定したタイムアウトが発生するまで呼び出し側のスレッドをブロックします。

`timeout` が `None` ならタイムアウトは設定されません。

1 つのプロセスは何回も `join` されることができます。

プロセスは自分自身を `join` することはできません。それはデッドロックを引き起こすことがあるからです。プロセスが `start` される前に `join` しようとするとエラーが発生します。

name

プロセス名です。

この名前は文字列で、プロセスの識別にのみ使用されます。特別な命名規則はありません。複数のプロセスが同じ名前を持つ場合もあります。また、この名前はコンストラクタにより初期化されます。

is_alive()

プロセスが実行中かを判別します。

おおまかに言って、プロセスオブジェクトは `start()` メソッドを呼び出してから子プロセス終了までの期間が実行中となります。

daemon

デーモンプロセスであるかのフラグであり、ブール値です。この属性は `start()` が呼び出される前に設定されている必要があります。

初期値は作成するプロセスから継承します。

あるプロセスが終了するとき、そのプロセスはその子プロセスであるデーモンプロセスすべてを終了させようとします。

デーモンプロセスは子プロセスを作成できないことに注意してください。もし作成できてしまうと、そのデーモンプロセスの親プロセスが終了したときにデーモンプロセスの子プロセスが孤児になってしまう場合があるからです。さらに言えば、デーモンプロセスは Unix デーモンやサービスではなく 通常のプロセスであり、非デーモンプロセスが終了すると終了されます (そして `join` されません)。

`threading.Thread` クラスの API に加えて `Process` クラスのオブジェクトには以下の属性およびメソッドがあります:

pid

プロセス ID を返します。プロセスの生成前は `None` が設定されています。

exitcode

子プロセスの終了コードです。子プロセスがまだ終了していない場合は `None` が返されます。負の値 `-N` は子プロセスがシグナル `N` で終了したことを表します。

authkey

プロセスの認証キーです (バイト文字列です)。

`multiprocessing` モジュールがメインプロセスにより初期化される場合には、`os.urandom()` 関数を使用してランダムな値が設定されます。

`Process` クラスのオブジェクトの作成時にその親プロセスから認証キーを継承します。もしくは `authkey` に別のバイト文字列を設定することもできます。

詳細は [認証キー](#) を参照してください。

terminate()

プロセスを終了します。Unix 環境では `SIGTERM` シグナルを、Windows 環境では `TerminateProcess()` を使用して終了させます。終了ハンドラーや `finally` 節などは、実行されないことに注意してください。

このメソッドにより終了するプロセスの子孫プロセスは、終了しません。そういった子孫プロセスは単純に孤児になります。

警告: このメソッドの使用時に、関連付けられたプロセスがパイプやキューを使用している場合には、使用中のパイプやキューが破損して他のプロセスから使用できなくなる可能性があります。同様に、プロセスがロックやセマフォなどを取得している場合には、このプロセスが終了してしまうと他のプロセスのデッドロックの原因になるでしょう。

プロセスオブジェクトが作成したプロセスのみが `start()`, `join()`, `is_alive()`, `terminate()` と `exitcode` のメソッドを呼び出すべきです。

以下の例では `Process` のメソッドの使い方を示しています:

```
>>> import multiprocessing, time, signal
>>> p = multiprocessing.Process(target=time.sleep, args=(1000,))
>>> print p, p.is_alive()
<Process(Process-1, initial)> False
>>> p.start()
>>> print p, p.is_alive()
<Process(Process-1, started)> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print p, p.is_alive()
<Process(Process-1, stopped[SIGTERM])> False
>>> p.exitcode == -signal.SIGTERM
True
```

exception multiprocessing.BufferTooShort

この例外は `Connection.recv_bytes_into()` によって発生し、バッファオブジェクトが小さすぎてメッセージが読み込めないことを示します。

`e` が `BufferTooShort` のインスタンスとすると、`e.args[0]` はバイト文字列でそのメッセージを取得できます。

パイプ (Pipe) とキュー (Queue)

複数のプロセスを使う場合、一般的にはメッセージパッシングをプロセス間通信に使用し、ロックのような同期プリミティブを使用しないようにします。

メッセージのやりとりのために `Pipe()` (2つのプロセス間の通信用)、もしくはキュー (複数のメッセージ生成プロセス (producer)、消費プロセス (consumer) の実現用) を使うことができます。

`Queue`, `multiprocessing.queues.SimpleQueue`, `JoinableQueue` は複数プロセスから生成/消費を行う FIFO キューです。これらのキューは標準ライブラリの `Queue.Queue` を模倣しています。`Queue` には Python 2.5 の `Queue.Queue` クラスで導入された `task_done()` と `join()` メソッドがないことが違う点です。

もし `JoinableQueue` を使用するなら、キューから削除される各タスクのために `JoinableQueue.task_done()` を呼び出さなければなりません。さもないと、いつか完了していないタスクを数えるためのセマフォがオーバーフローし、例外を発生させるでしょう。

管理オブジェクトを使用することで共有キューを作成することも覚えておいてください。詳細は [マネージャー](#) を参照してください。

注釈: `multiprocessing` は通常の `Queue.Empty` と、タイムアウトのシグナルを送るために `Queue.Full` 例外を使用します。それらは `Queue` からインポートする必要があるので `multiprocessing` の名前空間では利用できません。

注釈: オブジェクトがキューに追加される際、そのオブジェクトは pickle 化されています。そのため、バックグラウンドのスレッドが後になって下位層のパイプに pickle 化されたデータをフラッシュすることがあります。これにより、少し驚くような結果になりますが、実際に問題になることはないはずです。これが問題になるような状況では、かわりに `manager` を使ってキューを作成することができるからです。

- (1) 空のキューの中にオブジェクトを追加した後、キューの `empty()` メソッドが `False` を返すまでの間にごくわずかな遅延が起きることがあり、`get_nowait()` が `Queue.Empty` を発生させることなく制御が呼び出し元に返ってしまうことがあります。
 - (2) 複数のプロセスがオブジェクトをキューに詰めている場合、キューの反対側ではオブジェクトが詰められたのとは違う順序で取得される可能性があります。ただし、同一のプロセスから詰め込まれたオブジェクトは、それらのオブジェクト間では、必ず期待どおりの順序になります。
-

警告: `Queue` を利用しようとしている最中にプロセスを `Process.terminate()` や `os.kill()` で終了させる場合、キューにあるデータは破損し易くなります。終了した後で他のプロセスがキューを利用しようとすると、例外を発生させる可能性があります。

警告: 上述したように、もし子プロセスがキューへ要素を追加するなら (かつ `JoinableQueue.cancel_join_thread` を使用しないなら) そのプロセスはバッファされたすべての要素がパイプへフラッシュされるまで終了しません。

これは、そのプロセスを `join` しようとする場合、キューに追加されたすべての要素が消費されたことが確実でないかぎり、デッドロックを発生させる可能性があることを意味します。似たような現象で、子プロセスが非デーモンプロセスの場合、親プロセスは終了時に非デーモンのすべての子プロセスを `join` しようとしてハングアップする可能性があります。

マネージャーを使用して作成されたキューではこの問題はありません。詳細は [プログラミングガイドライン](#) を参照してください。

プロセス間通信におけるキューの使用例を知りたいなら [例](#) を参照してください。

`multiprocessing.Pipe([duplex])`

パイプの両端を表す `Connection` オブジェクトのペア (`conn1`, `conn2`) を返します。

`duplex` が `True` (デフォルト) ならパイプは双方向性です。 `duplex` が `False` ならパイプは一方方向性で、 `conn1` はメッセージの受信専用、 `conn2` はメッセージの送信専用になります。

`class multiprocessing.Queue([maxsize])`

パイプや 2~3 個のロック/セマフォを使用して実装されたプロセス共有キューを返します。あるプロセスが最初に要素をキューへ追加するとき、バッファからパイプの中へオブジェクトを転送する供給スレッドが開始されます。

標準ライブラリの `Queue` モジュールからの通常の `Queue.Empty` や `Queue.Full` 例外はタイムアウトのシグナルを送るために発生します。

`Queue` は `task_done()` や `join()` を除く `Queue.Queue` の全てのメソッドを実装します。

`qsize()`

おおよそのキューのサイズを返します。マルチスレッディング/マルチプロセスの特性上、この数値は信用できません。

これは `sem_getvalue()` が実装されていない Mac OS X のような Unix プラットホーム上で `NotImplementedError` を発生させる可能性があることを覚えておいてください。

`empty()`

キューが空っぽなら `True` を、そうでなければ `False` を返します。マルチスレッディング/マルチプロセッシングの特性上、これは信用できません。

`full()`

キューがいっぱいなら `True` を、そうでなければ `False` を返します。マルチスレッディング/マルチプロセッシングの特性上、これは信用できません。

`put(obj[, block[, timeout]])`

キューの中へ要素を追加します。オプションの引数 `block` が `True` (デフォルト) 且つ `timeout` が `None` (デフォルト) なら、空きスロットが利用可能になるまで必要であればブロックします。

`timeout` が正の数なら、最大 `timeout` 秒ブロックして、その時間内に空きスロットが利用できなかったら `Queue.Full` 例外を発生させます。それ以外 (`block` が `False`) で、空きスロットがすぐに利用可能な場合はキューに要素を追加します。そうでなければ `Queue.Full` 例外が発生します (その場合 `timeout` は無視されます)。

put_nowait (obj)

`put(obj, False)` と等価です。

get ([block[, timeout]])

キューから要素を取り出して削除します。オプションの引数 `block` が `True` (デフォルト) 且つ `timeout` が `None` (デフォルト) なら、要素が取り出せるまで必要であればブロックします。 `timeout` が正の数なら、最大 `timeout` 秒ブロックして、その時間内に要素が取り出せなかったら `Queue.Empty` 例外を発生させます。それ以外 (`block` が `False`) で、要素がすぐに取り出せる場合は要素を返します。そうでなければ `Queue.Empty` 例外が発生します (その場合 `timeout` は無視されます)。

get_nowait ()

`get(False)` と等価です。

`Queue` は `Queue.Queue` にはない追加メソッドがあります。これらのメソッドは通常、ほとんどのコードに必要ありません。

close ()

カレントプロセスからこのキューへそれ以上データが追加されないことを表します。バックグラウンドスレッドはパイプへバッファされたすべてのデータをフラッシュするとすぐに終了します。これはキューがガベージコレクトされるときに自動的に呼び出されます。

join_thread ()

バックグラウンドスレッドを `join` します。このメソッドは `close()` が呼び出された後でのみ使用されます。バッファされたすべてのデータがパイプへフラッシュされるのを保証するため、バックグラウンドスレッドが終了するまでブロックします。

デフォルトでは、あるプロセスがキューを作成していない場合、終了時にキューのバックグラウンドスレッドを `join` しようとします。そのプロセスは `join_thread()` が何もしないように `cancel_join_thread()` を呼び出すことができます。

cancel_join_thread ()

`join_thread()` がブロックするのを防ぎます。特にこれはバックグラウンドスレッドがそのプロセスの終了時に自動的に `join` されるのを防ぎます。詳細は `join_thread()` を参照してください。

このメソッドは `allow_exit_without_flush()` という名前のほうがよかったかもしれません。キューに追加されたデータが失われてしまいがちなため、このメソッドを使う必要はほぼ確実ないでしょう。本当にこれが必要になるのは、キューに追加されたデータを下位層のパイプにフラッシュすることなくカレントプロセスを直ちに終了する必要があり、かつ失われるデータに関心がない場合です。

注釈: このクラスに含まれる機能には、ホストとなるオペレーティングシステム上で動作している共有

セマフォ (shared semaphore) を使用しているものがあります。これが使用できない場合には、このクラスが無効になり、`Queue` をインスタンス化する時に `ImportError` が発生します。詳細は [bpo-3770](#) を参照してください。同様のことが、以下に列挙されている特殊なキューでも成り立ちます。

class multiprocessing.queues.SimpleQueue

単純化された `Queue` 型です。ロックされた `Pipe` に非常に似ています。

empty()

もしキューが空ならば `True` を、そうでなければ `False` を返します。

get()

キューからアイテムを取り除いて返します。

put(item)

`item` をキューに追加します。

class multiprocessing.JoinableQueue([maxsize])

`JoinableQueue` は `Queue` のサブクラスであり、`task_done()` や `join()` メソッドが追加されているキューです。

task_done()

以前にキューへ追加されたタスクが完了したことを表します。キュー消費スレッドによって使用されます。タスクをフェッチするために使用されるそれぞれの `get()` では、次に `task_done()` を呼び出してタスクの処理が完了したことをキューへ伝えます。

もし `join()` がブロッキング状態なら、全ての要素が処理されたときに復帰します (`task_done()` 呼び出しが全ての要素からキュー内へ `put()` されたと受け取ったことを意味します)。

キューにある要素より多く呼び出された場合 `ValueError` が発生します。

join()

キューにあるすべてのアイテムが取り出されて処理されるまでブロックします。

キューに要素が追加されると未終了タスク数が増えます。キューの要素が取り出されて全て処理が完了したことを表す `task_done()` を消費スレッドが呼び出すと数が減ります。未終了タスク数がゼロになると `join()` はブロッキングを解除します。

その他の関数

multiprocessing.active_children()

カレントプロセスのアクティブな子プロセスのすべてのリストを返します。

これを呼び出すと "join" してすでに終了しているプロセスには副作用があります。

multiprocessing.cpu_count()

システムの CPU 数を返します。 `NotImplementedError` が送出される場合があります。

`multiprocessing.current_process()`

カレントプロセスに対応する *Process* オブジェクトを返します。

threading.current_thread() とよく似た関数です。

`multiprocessing.freeze_support()`

multiprocessing を使用するプログラムが Windows の実行可能形式を生成しようとして固まったときのサポートを追加します。(py2exe, PyInstaller や cx.Freeze でテストされています。)

メインモジュールの `if __name__ == '__main__':` の直後にこの関数を呼び出す必要があります。以下に例を示します:

```
from multiprocessing import Process, freeze_support

def f():
    print 'hello world!'

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

もし `freeze_support()` の行がない場合、固まった実行可能形式を実行しようとするとき *RuntimeError* を発生させます。

`freeze_support()` の呼び出しは Windows 以外の OS では効果がありません。さらに、もしモジュールが Windows の通常の Python インタプリタによって実行されているならば (プログラムがフリーズされていなければ) `freeze_support()` は効果がありません。

`multiprocessing.set_executable()`

子プロセスを開始するときに、使用する Python インタプリタのパスを設定します。(デフォルトでは *sys.executable* が使用されます)。コードに組み込むときは、おそらく次のようにする必要があります

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

子プロセスを生成する前に行なってください。(Windows only)

注 釈: *multiprocessing* には *threading.active_count()*, *threading.enumerate()*, *threading.settrace()*, *threading.setprofile()*, *threading.Timer* や *threading.local* のような関数はありません。

Connection オブジェクト

Connection オブジェクトは pickle でシリアライズ可能なオブジェクトか文字列を送ったり、受け取ったりします。そういったオブジェクトはメッセージ指向の接続ソケットと考えられます。

Connection objects are usually created using *Pipe* – see also リスナーとクライアント。

class Connection**send(obj)**

コネクションの相手側へ `recv()` を使用して読み込むオブジェクトを送ります。

オブジェクトは pickle でシリアライズ可能でなければなりません。pickle が極端に大きすぎる (OS にも依りますが、およそ 32 MB+) と、`ValueError` 例外が送出されることがあります。

recv()

コネクションの相手側から `send()` を使用して送られたオブジェクトを返します。何か受け取るまでブロックします。何も受け取らずにコネクションの相手側でクローズされた場合 `EOFError` が発生します。

fileno()

コネクションが使用するハンドラーか、ファイル記述子を返します。

close()

コネクションをクローズします。

コネクションがガベージコレクトされるときに自動的に呼び出されます。

poll([timeout])

読み込み可能なデータがあるかどうかを返します。

`timeout` が指定されていない場合はすぐに返します。`timeout` に数値を指定すると、最大指定した秒数をブロッキングします。`timeout` に `None` を指定するとタイムアウトせずずっとブロッキングします。

send.bytes(buffer[, offset[, size]])

バッファインタフェースをサポートするオブジェクトから完全なメッセージとしてバイトデータを送ります。

`offset` が指定されると `buffer` のその位置からデータが読み込まれます。`size` が指定されるとバッファからその量のデータが読み込まれます。非常に大きなバッファ (OS に依存しますが、およそ 32MB+) を指定すると、`ValueError` 例外が発生するかもしれません。

recv.bytes([maxlength])

コネクションの相手側から送られたバイトデータの完全なメッセージを文字列として返します。何か受け取るまでブロックします。受け取るデータが何も残っておらず、相手側がコネクションを閉じていた場合、`EOFError` が送出されます。

`maxlength` を指定して、且つ `maxlength` よりメッセージが長い場合、`IOError` を発生させて、それ以上はコネクションから読み込めなくなります。

recv.bytes_into(buffer[, offset])

コネクションの相手側から送られたバイトデータを `buffer` に読み込み、メッセージのバイト数を返します。何か受け取るまでブロックします。何も受け取らずにコネクションの相手側でクローズされた場合 `EOFError` が発生します。

buffer は書き込み可能なバッファインタフェースを備えたオブジェクトでなければなりません。*offset* が与えられたら、その位置からバッファへメッセージが書き込まれます。オフセットは *buffer* バイトよりも小さい正の数でなければなりません。

バッファがあまりに小さいと `BufferTooShort` 例外が発生します。e が例外インスタンスとすると完全なメッセージは `e.args[0]` で確認できます。

例えば:

```
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes('thank you')
>>> a.recv_bytes()
'thank you'
>>> import array
>>> arr1 = array.array('i', range(5))
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

警告: `Connection.recv()` メソッドは受信したデータを自動的に unpickle 化します。それはメッセージを送ったプロセスが信頼できる場合を除いてセキュリティリスクになります。

そのため `Pipe()` を使用してコネクションオブジェクトを生成する場合を除いて、何らかの認証処理を実行した後で `recv()` や `send()` メソッドのみを使用すべきです。詳細は [認証キー](#) を参照してください。

警告: もしプロセスがパイプの読み込みまたは書き込み中に kill されると、メッセージの境界がどこなのか分からなくなってしまうので、そのパイプ内のデータは破損してしまいがちです。

同期プリミティブ

一般的にマルチプロセスプログラムは、マルチスレッドプログラムほどは同期プリミティブを必要としません。詳細は [threading](#) モジュールのドキュメントを参照してください。

マネージャーオブジェクトを使用して同期プリミティブを作成することも覚えておいてください。詳細は [マネージャー](#) を参照してください。

```
class multiprocessing.BoundedSemaphore([value])
```

有限セマフォオブジェクト: `threading.BoundedSemaphore` のクローンです。

インターフェイスがスレッドのものと一つだけ違います: `acquire` メソッドの最初の引数は `block`、オプション引数として `timeout` を取ります。これは `Lock.acquire()` ののと同じです。

注釈: Mac OS X では `sem_getvalue()` が実装されていないので `Semaphore` と区別が付きません。

class `multiprocessing.Condition` (`[lock]`)

状態変数: `threading.Condition` のクローンです。

`lock` を指定するなら `multiprocessing` の `Lock` か `RLock` オブジェクトにすべきです。

class `multiprocessing.Event`

`threading.Event` のクローンです。このメソッドは、終了時の内部セマフォの状態を返すので、タイムアウトが与えられ、実際にオペレーションがタイムアウトしたのでなければ、必ず `True` を返します。

バージョン 2.7 で変更: 以前はこのメソッドは常に `None` を返していました。

class `multiprocessing.Lock`

再帰しないロックオブジェクトで、`threading.Lock` 相当のものです。プロセスやスレッドがロックをいったん獲得 (`acquire`) すると、それに続くほかのプロセスやスレッドが獲得しようとする際、それが解放 (`release`) されるまではブロックされます。解放はどのプロセス、スレッドからも行えます。スレッドに対して適用される `threading.Lock` のコンセプトと振る舞いは、特筆すべきものがない限り、プロセスとスレッドに適用される `multiprocessing.Lock` に引き継がれています。

`Lock` は実際にはファクトリ関数で、デフォルトコンテキストで初期化された `multiprocessing.synchronize.Lock` のインスタンスを返すことに注意してください。

`Lock` は *context manager* プロトコルをサポートしています。つまり `with` 文で使うことができます。

acquire (`block=True`, `timeout=None`)

ブロックあり、またはブロックなしでロックを獲得します。

引数 `block` を `True` (デフォルト) に設定して呼び出した場合、ロックがアンロック状態になるまでブロックします。ブロックから抜けるとそれをロック状態にしてから `True` を返します。`threading.Lock.acquire()` の最初の引数とは名前が違っているので注意してください。

引数 `block` の値を `False` にして呼び出すとブロックしません。現在ロック状態であれば、直ちに `False` を返します。それ以外の場合には、ロックをロック状態にして `True` を返します。

`timeout` として正の浮動小数点数を与えて呼び出すと、ロックが獲得出来ないあいだ最大でこれで指定した秒数だけブロックします。`timeout` 値の負数はゼロと同じです。`timeout` 値の `None` (デフォルト) は無限にブロックすることを意味します。`block` が `False` の場合には `timeout` には実際的な意味はないので無視されます。ロックを獲得すると `True`、タイムアウトした場合は `False` で戻ります。`timeout` 引数は類似品の `threading.Lock.acquire()` にはないことに注意してください。

release ()

ロックを解放します。これはロックを獲得したプロセスやスレッドだけでなく、任意のプロセスや

スレッドから呼び出すことができます。

`threading.Lock.release()` と同じように振舞いますが、ロックされていない場合に呼び出すと `ValueError` となる点だけが違います。

class multiprocessing.RLock

再帰ロックオブジェクトで、`threading.RLock` 相当のものです。再帰ロックオブジェクトはそれを獲得 (acquire) したプロセスやスレッドが解放 (release) しなければなりません。プロセスやスレッドがロックをいったん獲得すると、同じプロセスやスレッドはブロックされずに再度獲得出来ます。そのプロセスやスレッドは獲得した回数ぶん解放しなければなりません。

`RLock` は実際にはファクトリ関数で、デフォルトコンテキストで初期化された `multiprocessing.synchronize.Lock` のインスタンスを返すことに注意してください。

`RLock` は *context manager* プロトコルをサポートしています。つまり `with` 文で使うことができます。

acquire (*block=True, timeout=None*)

ブロックあり、またはブロックなしでロックを獲得します。

block 引数を `True` にして呼び出した場合、ロックが既にカレントプロセスもしくはカレントスレッドが既に所有していない限りは、アンロック状態 (どのプロセス、スレッドも所有していない状態) になるまでブロックします。ブロックから抜けるとカレントプロセスもしくはカレントスレッドが (既に持っていないければ) 所有権を得て、再帰レベルをインクリメントし、`True` で戻ります。`threading.RLock.acquire()` の実装とはこの最初の引数の振る舞いが、その名前自身を始めとしていくつか違うので注意してください。

block 引数を `False` にして呼び出した場合、ブロックしません。ロックが他のプロセスもしくはスレッドにより獲得済み (つまり所有されている) であれば、カレントプロセスまたはカレントスレッドは所有権を得ず、再帰レベルも変更せずに、`False` で戻ります。ロックがアンロック状態の場合、カレントプロセスもしくはカレントスレッドは所有権を得て再帰レベルがインクリメントされ、`True` で戻ります。(—訳注: *block* の `True/False` 関係なくここでの説明では「所有権を持っている場合の2度目以降の *acquire*」の説明が欠けています。2度目以降の *acquire* では再帰レベルがインクリメントされて即座に返ります。全体読めばわかると思いますが一応。—)

timeout 引数の使い方と振る舞いは `Lock.acquire()` と同じです。*timeout* 引数は類似品の `threading.RLock.acquire()` にはないことに注意してください。

release()

再帰レベルをデクリメントしてロックを解放します。デクリメント後に再帰レベルがゼロになった場合、ロックの状態をアンロック (いかなるプロセス、いかなるスレッドにも所有されていない状態) にリセットし、ロックの状態がアンロックになるのを待ってブロックしているプロセスもしくはスレッドがある場合にはその中のただ一つだけが処理を進行できるようにします。デクリメント後も再帰レベルがゼロでない場合、ロックの状態はロックのままで、呼び出し側のプロセスもしくはスレッドに所有されたままになります。

このメソッドは呼び出しプロセスあるいはスレッドがロックを所有している場合に限り呼び出してください。所有者でないプロセスもしくはスレッドによって呼ばれるか、あるいはア

ンロック (未所有) 状態で呼ばれた場合、`AssertionError` が送出されます。同じ状況での `threading.RLock.release()` 実装とは例外の型が異なるので注意してください。

class multiprocessing.**Semaphore** (*[value]*)

セマフォオブジェクト: `threading.Semaphore` のクローンです。

インターフェイスがスレッドのものと一つだけ違います: `acquire` メソッドの最初の引数は `block`、オプション引数として `timeout` を取ります。これは `Lock.acquire()` でのもと同じです。

注釈: `BoundedSemaphore`, `Lock`, `RLock` と `Semaphore` の `acquire()` メソッドは `threading` ではサポートされていないタイムアウトパラメータを取ります。その引数はキーワード引数で受け取れる `acquire(block=True, timeout=None)` です。 `block` が `True` 且つ `timeout` が `None` ではないなら、タイムアウトが秒単位で設定されます。 `block` が `False` なら `timeout` は無視されます。

Mac OS X では `sem_timedwait` がサポートされていないので、`acquire()` にタイムアウトを与えて呼ぶと、ループ内でスリープすることでこの関数がエミュレートされます。

注釈: メインスレッドが `BoundedSemaphore.acquire()`, `Lock.acquire()`, `RLock.acquire()`, `Semaphore.acquire()`, `Condition.acquire()` 又は `Condition.wait()` を呼び出してブロッキング状態のときに `Ctrl-C` で生成される `SIGINT` シグナルを受け取ると、その呼び出しはすぐに中断されて `KeyboardInterrupt` が発生します。

これは同等のブロッキング呼び出しが実行中のときに `SIGINT` が無視される `threading` の振る舞いとは違っています。

注釈: このパッケージに含まれる機能には、ホストとなるオペレーティングシステム上で動作している共有セマフォを使用しているものがあります。これが使用できない場合には、`multiprocessing.synchronize` モジュールが無効になり、このモジュールのインポート時に `ImportError` が発生します。詳細は [bpo-3770](#) を参照してください。

共有 `ctypes` オブジェクト

子プロセスにより継承される共有メモリを使用する共有オブジェクトを作成することができます。

multiprocessing.Value (*typecode_or_type, *args[, lock]*)

共有メモリから割り当てられた `ctypes` オブジェクトを返します。デフォルトでは、返り値は実際のオブジェクトの同期ラッパーです。

`typecode_or_type` は返されるオブジェクトの型を決めます。それは `ctypes` の型が `array` モジュールで使われるような 1 文字の型コードかのどちらか一方です。 `*args` は型のコンストラクターへ渡されます。

`lock` が `True` (デフォルト) なら、値へ同期アクセスするために新たに再帰的なロックオブジェクトが作成されます。`lock` が `Lock` か `RLock` なら値への同期アクセスに使用されます。`lock` が `False` なら、返されたオブジェクトへのアクセスはロックにより自動的に保護されません。そのため、必ずしも ”プロセスセーフ” ではありません。

`+=` のような演算は、読み込みと書き込みを含むためアトミックではありません。このため、たとえば自動的に共有の値を増加させたい場合、以下のようにするのは不十分です

```
counter.value += 1
```

関連するロックが再帰的 (それがデフォルトです) なら、かわりに次のようにします

```
with counter.get_lock():
    counter.value += 1
```

`lock` はキーワード引数でのみ指定することに注意してください。

`multiprocessing.Array` (*typecode_or_type*, *size_or_initializer*, *, *lock=True*)

共有メモリから割り当てられた `ctypes` 配列を返します。デフォルトでは、返り値は実際の配列の同期ラッパーです。

typecode_or_type は返される配列の要素の型を決めます。それは `ctypes` の型か `array` モジュールで使われるような 1 文字の型コードかのどちらか一方です。*size_or_initializer* が整数なら、配列の長さを決定し、その配列はゼロで初期化されます。別の使用方法として *size_or_initializer* は配列の初期化に使用されるシーケンスになり、そのシーケンス長が配列の長さを決定します。

`lock` が `True` (デフォルト) なら、値へ同期アクセスするために新たなロックオブジェクトが作成されます。`lock` が `Lock` か `RLock` なら値への同期アクセスに使用されます。`lock` が `False` なら、返されたオブジェクトへのアクセスはロックにより自動的に保護されません。そのため、必ずしも ”プロセスセーフ” ではありません。

`lock` はキーワード引数としてのみ利用可能なことに注意してください。

`ctypes.c_char` の配列は文字列を格納して取り出せる `value` と `raw` 属性を持っていることを覚えておいてください。

`multiprocessing.sharedctypes` モジュール

`multiprocessing.sharedctypes` モジュールは子プロセスに継承される共有メモリの `ctypes` オブジェクトを割り当てる関数を提供します。

注釈: 共有メモリのポインターを格納することは可能ではありますが、特定プロセスのアドレス空間の位置を参照するということを覚えておいてください。しかし、そのポインターは別のプロセスのコンテキストにおいて無効になる確率が高いです。そして、別のプロセスからそのポインターを逆参照しようとするクラッシュを引き起こす可能性があります。

`multiprocessing.sharedctypes.RawArray` (*typecode_or_type*, *size_or_initializer*)

共有メモリから割り当てられた ctypes 配列を返します。

typecode_or_type は返される配列の要素の型を決めます。それは ctypes の型が `array` モジュールで使われるような 1 文字の型コードのどちらか一方です。 *size_or_initializer* が整数なら、それが配列の長さになり、その配列はゼロで初期化されます。別の使用方法として *size_or_initializer* には配列の初期化に使用されるシーケンスを設定することもでき、その場合はシーケンスの長さが配列の長さになります。

要素を取得したり設定したりすることは潜在的に非アトミックであることに注意してください。ロックを使用して自動的に同期化されたアクセスを保証するには `Array()` を使用してください。

`multiprocessing.sharedctypes.RawValue` (*typecode_or_type*, **args*)

共有メモリから割り当てられた ctypes オブジェクトを返します。

typecode_or_type は返されるオブジェクトの型を決めます。それは ctypes の型が `array` モジュールで使われるような 1 文字の型コードかのどちらか一方です。 **args* は型のコンストラクターへ渡されます。

値を取得したり設定したりすることは潜在的に非アトミックであることに注意してください。ロックを使用して自動的に同期化されたアクセスを保証するには `Value()` を使用してください。

`ctypes.c_char` の配列は文字列を格納して取り出せる `value` と `raw` 属性を持っていることを覚えておいてください。詳細は `ctypes` を参照してください。

`multiprocessing.sharedctypes.Array` (*typecode_or_type*, *size_or_initializer*, **args*[, *lock*])

`RawArray()` と同様ですが、*lock* の値によっては ctypes 配列をそのまま返す代わりに、プロセスセーフな同期ラッパーが返されます。

lock が `True` (デフォルト) なら、値へ同期アクセスするために新たな ロックオブジェクトが作成されます。 *lock* が `Lock` か `RLock` なら値への同期アクセスに使用されます。 *lock* が `False` なら、返された オブジェクトへのアクセスはロックにより自動的に保護されません。 そのため、必ずしも ”プロセスセーフ” ではありません。

lock はキーワード引数でのみ指定することに注意してください。

`multiprocessing.sharedctypes.Value` (*typecode_or_type*, **args*[, *lock*])

`RawValue()` と同様ですが、*lock* の値によっては ctypes オブジェクトをそのまま返す代わりに、プロセスセーフな同期ラッパーが返されます。

lock が `True` (デフォルト) なら、値へ同期アクセスするために新たな ロックオブジェクトが作成されます。 *lock* が `Lock` か `RLock` なら値への同期アクセスに使用されます。 *lock* が `False` なら、返された オブジェクトへのアクセスはロックにより自動的に保護されません。 そのため、必ずしも ”プロセスセーフ” ではありません。

lock はキーワード引数でのみ指定することに注意してください。

`multiprocessing.sharedctypes.copy` (*obj*)

共有メモリから割り当てられた ctypes オブジェクト *obj* をコピーしたオブジェクトを返します。

`multiprocessing.sharedctypes.synchronized(obj[, lock])`

同期アクセスに `lock` を使用する `ctypes` オブジェクトのためにプロセスセーフなラッパーオブジェクトを返します。`lock` が `None` (デフォルト) なら、`multiprocessing.RLock` オブジェクトが自動的に作成されます。

同期ラッパーがラップするオブジェクトに加えて 2 つのメソッドがあります。`get_obj()` はラップされたオブジェクトを返します。`get_lock()` は同期のために使用されるロックオブジェクトを返します。

ラッパー経由で `ctypes` オブジェクトにアクセスすることは raw `ctypes` オブジェクトへアクセスするよりずっと遅くなることに注意してください。

次の表は通常の `ctypes` 構文で共有メモリから共有 `ctypes` オブジェクトを作成するための構文を比較します。(MyStruct テーブル内には `ctypes.Structure` のサブクラスがあります。)

ctypes	type を使用する sharedctypes	typecode を使用する sharedctypes
<code>c_double(2.4)</code>	<code>RawValue(c_double, 2.4)</code>	<code>RawValue('d', 2.4)</code>
<code>MyStruct(4, 6)</code>	<code>RawValue(MyStruct, 4, 6)</code>	
<code>(c_short * 7)()</code>	<code>RawArray(c_short, 7)</code>	<code>RawArray('h', 7)</code>
<code>(c_int * 3)(9, 2, 8)</code>	<code>RawArray(c_int, (9, 2, 8))</code>	<code>RawArray('i', (9, 2, 8))</code>

以下に子プロセスが多くの `ctypes` オブジェクトを変更する例を紹介します:

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
        a.y **= 2

if __name__ == '__main__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', 'hello world', lock=lock)
    A = Array(Point, [(1.875, -6.25), (-5.75, 2.0), (2.375, 9.5)], lock=lock)

    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()
```

(次のページに続く)

(前のページからの続き)

```
print n.value
print x.value
print s.value
print [(a.x, a.y) for a in A]
```

結果は以下のように表示されます

```
49
0.1111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
```

マネージャー

`Manager` は別のプロセス間で共有されるデータの作成方法を提供します。マネージャオブジェクトは共有オブジェクトを管理するサーバプロセスを制御します。他のプロセスはプロキシ経由で共有オブジェクトへアクセスすることができます。

`multiprocessing.Manager()`

プロセス間でオブジェクトを共有するために使用される *SyncManager* オブジェクトを返します。返されたマネージャオブジェクトは生成される子プロセスに対応付けられ、共有オブジェクトを作成するメソッドや、共有オブジェクトに対応するプロキシを返すメソッドを持ちます。

マネージャプロセスは親プロセスが終了するか、ガベージコレクトされると停止します。マネージャークラスは *multiprocessing.managers* モジュールで定義されています:

class `multiprocessing.managers.BaseManager` (`[address[, authkey]]`)

`BaseManager` オブジェクトを作成します。

作成後、`start()` か `get_server().serve_forever()` を呼び出して、マネージャオブジェクトが確実に開始されたマネージャプロセスを参照するようにしてください。

`address` はマネージャプロセスが新たなコネクションを待ち受けるアドレスです。`address` が `None` の場合、任意のアドレスが設定されます。

`authkey` はサーバプロセスへ接続しようとするコネクションの正当性を検証するために使用される認証キーです。`authkey` が `None` の場合 `current_process().authkey` が使用されます。`authkey` を使用する場合は文字列でなければなりません。

start (`[initializer[, initargs]]`)

マネージャを開始するためにサブプロセスを開始します。`initializer` が `None` でなければ、サブプロセスは開始時に `initializer(*initargs)` を呼び出します。

get_server ()

マネージャの制御下にある実際のサーバーを表す `Server` オブジェクトを返します。`Server` オブジェクトは `serve_forever()` メソッドをサポートします:

```
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=('', 50000), authkey='abc')
>>> server = manager.get_server()
>>> server.serve_forever()
```

Server はさらに *address* 属性も持っています。

connect()

ローカルからリモートのマネージャーオブジェクトへ接続します:

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 5000), authkey='abc')
>>> m.connect()
```

shutdown()

マネージャーが使用するプロセスを停止します。これはサーバープロセスを開始するために *start()* が使用された場合のみ有効です。

これは複数回呼び出すことができます。

register(*typeid*[, *callable*[, *proxytype*[, *exposed*[, *method_to_typeid*[, *create_method*]]]])

マネージャークラスで呼び出し可能オブジェクト (*callable*) や型を登録するために使用されるクラスメソッドです。

typeid は特に共有オブジェクトの型を識別するために使用される ”型識別子” です。これは文字列でなければなりません。

callable はこの型識別子のオブジェクトを作成するために使用される呼び出し可能オブジェクトです。マネージャインスタンスが *from_address()* クラスメソッドを使用して作成されるか、*create_method* 引数が *False* の場合は *None* でも構いません。

proxytype はこの *typeid* で共有オブジェクトのプロキシを作成するために使用される *BaseProxy* のサブクラスです。 *None* の場合、プロキシクラスは自動的に作成されます。

exposed は *BaseProxy._callmethod()* を使用したアクセスが許されるべき *typeid* をプロキシするメソッド名のシーケンスを指定するために使用されます (*exposed* が *None* の場合 *proxytype._exposed_* が存在すればそれが代わりに使用されます)。 *exposed* リストが指定されない場合、共有オブジェクトのすべての ”パブリックメソッド” がアクセス可能になります。(ここでいう ”パブリックメソッド” とは *__call__()* メソッドを持つものと名前が *'_'* で始まらないあらゆる属性を意味します。)

method_to_typeid はプロキシが返す *exposed* メソッドの戻り値の型を指定するために使用されるマッピングで、メソッド名を *typeid* 文字列にマップします。(*method_to_typeid* が *None* の場合 *proxytype._method_to_typeid_* が存在すれば、それが代わりに使用されます。) メソッド名がこのマッピングのキーではないか、マッピングが *None* の場合、そのメソッドによって返されるオブジェクトが値として (by value) コピーされます。

create_method は、共有オブジェクトを作成し、それに対するプロキシを返すようサーバープロセスに伝える、名前 *typeid* のメソッドを作成するかを決定します。デフォルトでは *True* です。

BaseManager インスタンスも読み取り専用属性を 1 つ持っています:

address

マネージャーが使用するアドレスです。

class multiprocessing.managers.SyncManager

プロセス間の同期のために使用される *BaseManager* のサブクラスです。 `multiprocessing.Manager()` はこの型のオブジェクトを返します。

また共有のリストやディクショナリの作成もサポートします。

BoundedSemaphore (*[value]*)

共有 *threading.BoundedSemaphore* オブジェクトを作成して、そのプロキシを返します。

Condition (*[lock]*)

共有 *threading.Condition* オブジェクトを作成して、そのプロキシを返します。

lock が提供される場合 *threading.Lock* か *threading.RLock* オブジェクトのためのプロキシになります。

Event ()

共有 *threading.Event* オブジェクトを作成して、そのプロキシを返します。

Lock ()

共有 *threading.Lock* オブジェクトを作成して、そのプロキシを返します。

Namespace ()

共有 *Namespace* オブジェクトを作成して、そのプロキシを返します。

Queue (*[maxsize]*)

共有 *Queue.Queue* オブジェクトを作成して、そのプロキシを返します。

RLock ()

共有 *threading.RLock* オブジェクトを作成して、そのプロキシを返します。

Semaphore (*[value]*)

共有 *threading.Semaphore* オブジェクトを作成して、そのプロキシを返します。

Array (*typecode, sequence*)

配列を作成して、そのプロキシを返します。

Value (*typecode, value*)

書き込み可能な *value* 属性を作成して、そのプロキシを返します。

dict ()

dict (*mapping*)

dict (*sequence*)

共有 *dict* オブジェクトを作成して、そのプロキシを返します。

list ()

list (*sequence*)

共有 *list* オブジェクトを作成して、そのプロキシを返します。

注釈: プロキシには、自身の持つ辞書とリストのミュータブルな値や項目がいつ変更されたのかを知る方法がないため、これらの値や項目の変更はマネージャーを通して伝播しません。このような要素を変更するには、コンテナのプロキシに変更されたオブジェクトを再代入してください:

```
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
d = lproxy[0]
d['a'] = 1
d['b'] = 2
# at this point, the changes to d are not yet synced, but by
# reassigning the dictionary, the proxy is notified of the change
lproxy[0] = d
```

class multiprocessing.managers.Namespace

SyncManager に登録することのできる型です。

Namespace オブジェクトにはパブリックなメソッドはありませんが、書き込み可能な属性を持ちます。そのオブジェクト表現はその属性の値を表示します。

しかし、Namespace オブジェクトのためにプロキシを使用するとき '_' が先頭に付く属性はプロキシの属性になり、参照対象の属性にはなりません:

```
>>> manager = multiprocessing.Manager()
>>> Global = manager.Namespace()
>>> Global.x = 10
>>> Global.y = 'hello'
>>> Global._z = 12.3      # this is an attribute of the proxy
>>> print Global
Namespace(x=10, y='hello')
```

カスタマイズされたマネージャー

独自のマネージャーを作成するには、*BaseManager* のサブクラスを作成して、マネージャークラスで呼び出し可能なオブジェクトが新たな型を登録するために *register()* クラスメソッドを使用します。例えば:

```
from multiprocessing.managers import BaseManager

class MathsClass(object):
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
    pass
```

(次のページに続く)

(前のページからの続き)

```
MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    manager = MyManager()
    manager.start()
    maths = manager.Maths()
    print maths.add(4, 3)          # prints 7
    print maths.mul(7, 8)         # prints 56
```

リモートマネージャーを使用する

あるマシン上でマネージャーサーバーを実行して、他のマシンからそのサーバーを使用するクライアントを持つことができます (ファイアウォールを通過できることが前提)。

次のコマンドを実行することでリモートクライアントからアクセスを受け付ける 1 つの共有キューのためにサーバーを作成します:

```
>>> from multiprocessing.managers import BaseManager
>>> import Queue
>>> queue = Queue.Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda:queue)
>>> m = QueueManager(address=('', 50000), authkey='abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

あるクライアントからサーバーへのアクセスは次のようになります:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey='abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.put('hello')
```

別のクライアントもそれを使用することができます:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey='abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'
```

ローカルプロセスもそのキューへアクセスすることができます。クライアント上で上述のコードを使用してア

クセスします:

```
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super(Worker, self).__init__()
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
...
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('', 50000), authkey='abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

Proxy オブジェクト

プロキシは別のプロセスで (おそらく) 有効な共有オブジェクトを参照するオブジェクトです。共有オブジェクトはプロキシの参照対象になることができます。複数のプロキシオブジェクトが同じ参照対象を持つ可能性もあります。

プロキシオブジェクトはその参照対象が持つ対応メソッドを実行するメソッドを持ちます。(そうは言っても、参照対象のすべてのメソッドが必ずしもプロキシ経由で利用可能ではありません) プロキシは通常その参照対象ができることと同じ方法で使用されます:

```
>>> from multiprocessing import Manager
>>> manager = Manager()
>>> l = manager.list([i*i for i in range(10)])
>>> print l
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print repr(l)
<ListProxy object, typeid 'list' at 0x...>
>>> l[4]
16
>>> l[2:5]
[4, 9, 16]
```

プロキシに `str()` を適用すると参照対象のオブジェクト表現を返すのに対して、`repr()` を適用するとプロキシのオブジェクト表現を返すことに注意してください。

プロキシオブジェクトの重要な機能はプロセス間で受け渡し可能な pickle 化ができることです。しかし、プロキシが対応するマネージャープロセスに対して送信される場合、そのプロキシを unpickle するとその参照対象を生成することを覚えておいてください。例えば、これはある共有オブジェクトに別の共有オブジェクトが含まれることを意味します:

```
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)           # referent of a now contains referent of b
>>> print a, b
[[]] []
>>> b.append('hello')
>>> print a, b
[['hello']] ['hello']
```

注釈: `multiprocessing` のプロキシ型は値による比較に対して何もサポートしません。そのため、例えば以下のようになります:

```
>>> manager.list([1,2,3]) == [1,2,3]
False
```

比較を行いたいときは参照対象のコピーを使用してください。

class `multiprocessing.managers.BaseProxy`

プロキシオブジェクトは `BaseProxy` のサブクラスのインスタンスです。

`_callmethod(methodname[, args[, kwds]])`

プロキシの参照対象のメソッドの実行結果を返します。

`proxy` がプロキシで、プロキシ内の参照対象が `obj` ならこの式

```
proxy._callmethod(methodname, args, kwds)
```

はこの式を評価します

```
getattr(obj, methodname)(*args, **kwds)
```

(マネージャープロセス内の)。

返される値はその呼び出し結果のコピーか、新たな共有オブジェクトに対するプロキシになります。詳細は `BaseManager.register()` の `method_to_typeid` 引数のドキュメントを参照してください。

その呼び出しによって例外が発生した場合、`_callmethod()` によってその例外は再送出されません。他の例外がマネージャープロセスで発生したなら、`RemoteError` 例外に変換されたものが `_callmethod()` によって送出されます。

特に `methodname` が 公開 されていない場合は例外が発生することに注意してください。

`_callmethod()` の使用例になります:

```
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
```

(次のページに続く)

(前のページからの続き)

```
>>> l.__callmethod('__getslice__', (2, 7))    # equiv to `l[2:7]`
[2, 3, 4, 5, 6]
>>> l.__callmethod('__getitem__', (20,))      # equiv to `l[20]`
Traceback (most recent call last):
...
IndexError: list index out of range
```

__getvalue__()

参照対象のコピーを返します。

参照対象が unpickle 化できるなら例外を発生します。

__repr__()

プロキシオブジェクトのオブジェクト表現を返します。

__str__()

参照対象のオブジェクト表現を返します。

クリーンアップ

プロキシオブジェクトは弱参照 (weakref) コールバックを使用します。プロキシオブジェクトがガベージコレクションされるとときにその参照対象が所有するマネージャーからその登録を取り消せるようにするためです。

共有オブジェクトはプロキシが参照しなくなったときにマネージャープロセスから削除されます。

プロセスプール

Pool クラスでタスクを実行するプロセスのプールを作成することができます。

```
class multiprocessing.Pool ([processes[, initializer[, initargs[, maxtasksperchild]]])
```

プロセスプールオブジェクトは、ジョブを送り込めるワーカープロセスのプールを制御します。タイムアウトやコールバックのある非同期の実行をサポートし、並列 map 実装を持ちます。

processes は使用するワーカープロセスの数です。 *processes* が None の場合 `cpu_count()` が返す数を使用します。 *initializer* が None でない場合、各ワーカープロセスが開始時に `initializer(*initargs)` を呼び出します。

プールオブジェクトのメソッドは、そのプールを作成したプロセスのみが呼び出すべきです。

バージョン 2.7 で追加: *maxtasksperchild* は、ワーカープロセスが exit して新たなワーカープロセスと置き替えられるまでの間に、ワーカープロセスが完了することのできるタスクの数です。この設定により未利用のリソースが解放されるようになります。デフォルトの *maxtasksperchild* は None で、これはワーカープロセスがプールと同じ期間だけ生き続けるということを意味します。

注釈: Pool 中のワーカープロセスは、典型的にはプールのワークキューの存続期間とちょうど同じだけ生き続けます。ワーカーに確保されたリソースを解放するために (Apache, mod_wsgi, などのような) 他のシステムによく見られるパターンは、プール内のワーカーが設定された量だけの仕事を完了したら

`exit` とクリーンアップを行い、古いプロセスを置き換えるために新しいプロセスを生成するというものです。 `Pool` の `maxtasksperchild` 引数は、この能力をエンドユーザーに提供します。

`apply` (*func* [, *args* [, *kwds*]])

組み込み関数 `apply()` の同等品で、結果が取得出来るようになるまでブロックします。ですから作業を並列に実行するのにより相応しいのは `apply_async()` です。加えて *func* はプールのワーカーのうち一つの中で実行されるに過ぎません。

`apply_async` (*func* [, *args* [, *kwds* [, *callback*]]])

`apply()` メソッドの派生版で結果オブジェクトを返します。

callback を指定する場合は 1 つの引数を受け取る呼び出し可能オブジェクトでなければなりません。その結果を返せるようになったときに *callback* が結果オブジェクトに対して (その呼び出しが失敗しない限り) 適用されます。コールバックは直ちに完了すべきです。なぜなら、そうしなければ、結果を扱うスレッドがブロックするからです。

`map` (*func*, *iterable* [, *chunksize*])

`map()` 組み込み関数の並列版です (*iterable* な引数を 1 つだけサポートするという違いはありますが)。結果が出るまでブロックします。

このメソッドはイテラブルをいくつものチャンクに分割し、プロセスプールにそれぞれ独立したタスクとして送ります。(概算の) チャンクサイズは *chunksize* を正の整数に設定することで指定できます。

`map_async` (*func*, *iterable* [, *chunksize* [, *callback*]])

`map()` メソッドの派生版で結果オブジェクトを返します。

callback を指定する場合は 1 つの引数を受け取る呼び出し可能オブジェクトでなければなりません。その結果を返せるようになったときに *callback* が結果オブジェクトに対して (その呼び出しが失敗しない限り) 適用されます。コールバックは直ちに完了すべきです。なぜなら、そうしなければ、結果を扱うスレッドがブロックするからです。

`imap` (*func*, *iterable* [, *chunksize*])

`itertools.imap()` と同じです。

chunksize 引数は `map()` メソッドで使用されるものと同じです。引数 *iterable* がとても長いなら *chunksize* に大きな値を指定して使用する方がデフォルト値の 1 を使用するよりもジョブの完了がかなり速くなります。

また *chunksize* が 1 の場合 `imap()` メソッドが返すイテレーターの `next()` メソッドはオプションで *timeout* パラメーターを持ちます。 `next(timeout)` は、その結果が *timeout* 秒以内に返されないときに `multiprocessing.TimeoutError` を発生させます。

`imap_unordered` (*func*, *iterable* [, *chunksize*])

イテレーターが返す結果の順番が任意の順番で良いと見なされることを除けば `imap()` と同じです。(ワーカープロセスが 1 つしかない場合のみ ”正しい” 順番になることが保証されます。)

`close()`

これ以上プールでタスクが実行されないようにします。すべてのタスクが完了した後でワーカープ

プロセスが終了します。

terminate()

実行中の処理を完了させずにワーカプロセスをすぐに停止します。プールオブジェクトがガベージコレクトされるときに `terminate()` が呼び出されます。

join()

ワーカプロセスが終了するのを待ちます。 `join()` を使用する前に `close()` か `terminate()` を呼び出さなければなりません。

class multiprocessing.pool.AsyncResult

`Pool.apply_async()` や `Pool.map_async()` で返される結果のクラスです。

get([timeout])

結果を受け取ったときに返します。 `timeout` が `None` ではなくて、その結果が `timeout` 秒以内に受け取れない場合 `multiprocessing.TimeoutError` が発生します。リモートの呼び出しが例外を発生させる場合、その例外は `get()` が再発生させます。

wait([timeout])

その結果が有効になるか `timeout` 秒経つまで待ちます。

ready()

その呼び出しが完了しているかどうかを返します。

successful()

その呼び出しが例外を発生させることなく完了したかどうかを返します。その結果が返せる状態でない場合 `AssertionError` が発生します。

次の例はプールの使用例を紹介します:

```
from multiprocessing import Pool
import time

def f(x):
    return x*x

if __name__ == '__main__':
    pool = Pool(processes=4)           # start 4 worker processes

    result = pool.apply_async(f, (10,)) # evaluate "f(10)" asynchronously in a
↪single process
    print result.get(timeout=1)         # prints "100" unless your computer is
↪*very* slow

    print pool.map(f, range(10))        # prints "[0, 1, 4, ..., 81]"

    it = pool.imap(f, range(10))
    print it.next()                     # prints "0"
    print it.next()                     # prints "1"
    print it.next(timeout=1)            # prints "4" unless your computer is
↪*very* slow
```

(次のページに続く)

(前のページからの続き)

```
result = pool.apply_async(time.sleep, (10,))
print result.get(timeout=1)           # raises multiprocessing.TimeoutError
```

リスナーとクライアント

Usually message passing between processes is done using queues or by using *Connection* objects returned by *Pipe()*.

しかし *multiprocessing.connection* モジュールはさらに柔軟な仕組みがあります。基本的にはソケットもしくは Windows の名前付きパイプを扱う高レベルのメッセージ指向 API を提供して *hmac* モジュールを使用して ダイジェスト認証 もサポートします。

multiprocessing.connection.deliver_challenge (*connection*, *authkey*)

ランダム生成したメッセージをコネクションの相手側へ送信して応答を待ちます。

その応答がキーとして *authkey* を使用するメッセージのダイジェストと一致する場合、コネクションの相手側へ歓迎メッセージを送信します。そうでなければ *AuthenticationError* を発生させます。

multiprocessing.connection.answer_challenge (*connection*, *authkey*)

メッセージを受信して、そのキーとして *authkey* を使用するメッセージのダイジェストを計算し、ダイジェストを送り返します。

歓迎メッセージを受け取れない場合 *AuthenticationError* が発生します。

multiprocessing.connection.Client (*address*[, *family*[, *authenticate*[, *authkey*]]])

Attempt to set up a connection to the listener which is using address *address*, returning a *Connection*.

コネクション種別は *family* 引数で決定しますが、一般的には *address* のフォーマットから推測できるので、これは指定されません。(アドレスフォーマットを参照してください)

authenticate が *True* か *authkey* が文字列の場合、ダイジェスト認証が使用されます。認証に使用されるキーは *authkey*、又は *authkey* が *None* の場合は *current_process().authkey* のどちらかです。認証が失敗した場合 *AuthenticationError* が発生します。認証キーを参照してください。

class *multiprocessing.connection.Listener* ([*address*[, *family*[, *backlog*[, *authenticate*[, *authkey*]]]]])

コネクションを '待ち受ける' 束縛されたソケットか Windows の名前付きパイプのラッパーです。

address はリスナーオブジェクトの束縛されたソケットか名前付きパイプが使用するアドレスです。

注釈: '0.0.0.0' のアドレスを使用する場合、Windows 上の終点へ接続することができません。終点へ接続したい場合は '127.0.0.1' を使用すべきです。

family は使用するソケット (名前付きパイプ) の種別です。これは 'AF_INET' (TCP ソケット), 'AF_UNIX' (Unix ドメインソケット) または 'AF_PIPE' (Windows 名前付きパイプ) という文字列のどれか 1 つになります。これらのうち 'AF_INET' のみが利用可能であることが保証されてい

ます。 *family* が `None` の場合 *address* のフォーマットから推測されたものが使用されます。 *address* も `None` の場合はデフォルトが選択されます。詳細は [アドレスフォーマット](#) を参照してください。 *family* が `'AF_UNIX'` で *address* が `None` の場合 `tempfile.mkstemp()` を使用して作成されたプライベートな一時ディレクトリにソケットが作成されます。

リスナーオブジェクトがソケットを使用する場合、ソケットに束縛されるときに *backlog* (デフォルトでは 1 つ) がソケットの `listen()` メソッドに対して渡されます。

authenticate が `True` (デフォルトでは `False`) か *authkey* が `None` ではない場合、ダイジェスト認証が使用されます。

authkey が文字列の場合、認証キーとして使用されます。そうでない場合は `None` でなければいけません。

authkey が `None` 且つ *authenticate* が `True` の場合 `current_process().authkey` が認証キーとして使用されます。 *authkey* が `None` 且つ *authentication* が `False` の場合、認証は行われません。もし認証が失敗した場合 `AuthenticationError` が発生します。詳細 [認証キー](#) を参照してください。

accept()

Accept a connection on the bound socket or named pipe of the listener object and return a `Connection` object. If authentication is attempted and fails, then `AuthenticationError` is raised.

close()

リスナーオブジェクトの名前付きパイプが束縛されたソケットをクローズします。これはリスナーがガベージコレクトされるときに自動的に呼ばれます。そうは言っても、明示的に `close()` を呼び出す方が望ましいです。

リスナーオブジェクトは次の読み取り専用属性を持っています:

address

リスナーオブジェクトが使用中のアドレスです。

last_accepted

最後にコネクションを受け付けたアドレスです。有効なアドレスがない場合は `None` になります。

The module defines the following exceptions:

exception multiprocessing.connection.ProcessError

すべての `multiprocessing` 例外の基底クラスです。

exception multiprocessing.connection.BufferTooShort

この例外は `Connection.recv_bytes_into()` によって発生し、バッファオブジェクトが小さすぎてメッセージが読み込めないことを示します。

exception multiprocessing.connection.AuthenticationError

認証エラーがあった場合に送出されます。

exception multiprocessing.connection.TimeoutError

タイムアウトをサポートするメソッドでタイムアウトが過ぎたときに送出されます。

例

次のサーバーコードは認証キーとして 'secret password' を使用するリスナーを作成します。このサーバーはコネクションを待ってクライアントヘータを送信します:

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)      # family is deduced to be 'AF_INET'
listener = Listener(address, authkey='secret password')

conn = listener.accept()
print 'connection accepted from', listener.last_accepted

conn.send([2.25, None, 'junk', float])

conn.send_bytes('hello')

conn.send_bytes(array('i', [42, 1729]))

conn.close()
listener.close()
```

次のコードはサーバーへ接続して、サーバーからデータを受信します:

```
from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)
conn = Client(address, authkey='secret password')

print conn.recv()                # => [2.25, None, 'junk', float]

print conn.recv_bytes()          # => 'hello'

arr = array('i', [0, 0, 0, 0, 0])
print conn.recv_bytes_into(arr)  # => 8
print arr                        # => array('i', [42, 1729, 0, 0, 0])

conn.close()
```

アドレスフォーマット

- 'AF_INET' アドレスは (hostname, port) のタプルになります。 *hostname* は文字列で *port* は整数です。
- 'AF_UNIX' アドレスはファイルシステム上のファイル名の文字列です。
- 'AF_PIPE' アドレスは、次の形式を持つ文字列です `r'\.\pipe{PipeName}'`。 *ServerName* という名前のリモートコンピューター上の名前付きパイプに接続するために `Client()` を使用するには、代わりに `r'\ServerName\pipe{PipeName}'` 形式のアドレスを使用する必要があります

ます。

デフォルトでは、2つのバックスラッシュで始まる文字列は 'AF_UNIX' よりも 'AF_PIPE' として推測されることに注意してください。

認証キー

When one uses `Connection.recv()`, the data received is automatically unpickled. Unfortunately unpickling data from an untrusted source is a security risk. Therefore `Listener` and `Client()` use the `hmac` module to provide digest authentication.

認証キーはパスワードとしてみなされる文字列です。コネクションが確立すると、双方の終点で正しい接続先であることを証明するために知っているお互いの認証キーを要求します。(双方の終点が同じキーを使用して通信しようとしても、コネクション上でそのキーを送信することはできません。)

認証が要求されているにもかかわらず認証キーが指定されていない場合 `current_process().authkey` の返す値が使用されます。(詳細は `Process` を参照してください。) この値はカレントプロセスを作成する `Process` オブジェクトによって自動的に継承されます。これは(デフォルトでは)複数プロセスのプログラムの全プロセスが相互にコネクションを確立するときに使用される1つの認証キーを共有することを意味します。

適当な認証キーを `os.urandom()` を使用して生成することもできます。

ログ記録

ロギングのためにいくつかの機能が利用可能です。しかし `logging` パッケージは、(ハンドラー種別に依存して) 違うプロセスからのメッセージがごちゃ混ぜになるので、プロセスの共有ロックを使用しないことに注意してください。

`multiprocessing.get_logger()`

`multiprocessing` が使用するロガーを返します。必要に応じて新たなロガーを作成します。

最初に作成するとき、ロガーはレベルに `logging.NOTSET` が設定されていてデフォルトハンドラーがありません。このロガーへ送られるメッセージはデフォルトではルートロガーへ伝播されません。

Windows 上では子プロセスが親プロセスのロガーレベルを継承しないことに注意してください。さらにその他のロガーのカスタマイズ内容もすべて継承されません。

`multiprocessing.log_to_stderr()`

この関数は `get_logger()` に対する呼び出しを実行しますが、`get_logger` によって作成されるロガーを返すことに加えて、`'[% (levelname)s/% (processName)s] % (message)s'` のフォーマットを使用して `sys.stderr` へ出力を送るハンドラーを追加します。

以下にロギングを有効にした例を紹介します:

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
```

(次のページに続く)

(前のページからの続き)

```
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pymp-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

これらの2つのロギング関数があることに加えて、`multiprocessing` モジュールも2つの追加ロギングレベル属性を提供します。それは `SUBWARNING` と `SUBDEBUG` です。次の表は通常のレベル階層にうまく適合していることを表します。

レベル	Numeric value
<code>SUBWARNING</code>	25
<code>SUBDEBUG</code>	5

完全なロギングレベルの表については `logging` モジュールを参照してください。

こういった追加のロギングレベルは主に `multiprocessing` モジュールの信頼できるデバッグメッセージのために使用されます。以下に上述の例に `SUBDEBUG` を有効にしたものを紹介します：

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(multiprocessing.SUBDEBUG)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pymp-...
[INFO/SyncManager-...] manager serving at '/.../pymp-djGBXN/listener-...'
>>> del m
[SUBDEBUG/MainProcess] finalizer calling ...
[INFO/MainProcess] sending shutdown message to manager
[DEBUG/SyncManager-...] manager received shutdown message
[SUBDEBUG/SyncManager-...] calling <Finalize object, callback=unlink, ...
[SUBDEBUG/SyncManager-...] finalizer calling <built-in function unlink> ...
[SUBDEBUG/SyncManager-...] calling <Finalize object, dead>
[SUBDEBUG/SyncManager-...] finalizer calling <function rmtree at 0x5aa730> ...
[INFO/SyncManager-...] manager exiting with exitcode 0
```

`multiprocessing.dummy` モジュール

`multiprocessing.dummy` は `multiprocessing` の API を複製しますが `threading` モジュールのラッパーでしかありません。

16.6.3 プログラミングガイドライン

`multiprocessing` を使用するときを守るべき一定のガイドラインとイディオムを挙げます。

全てのプラットフォーム

共有状態を避ける

できるだけプロセス間で巨大なデータを移動することは避けるようにすべきです。

プロセス間の通信には、`threading` モジュールの低レベルな同期プリミティブを使うのではなく、キューやパイプを使うのが良いでしょう。

pickle 化の可能性

プロキシのメソッドへの引数は、pickle 化できるものにしてください。

プロキシのスレッドセーフ性

1 つのプロキシオブジェクトは、ロックで保護しないかぎり、2 つ以上のスレッドから使用してはいけません。

(異なるプロセスで 同じ プロキシを使用することは問題ではありません。)

ゾンビプロセスを join する

Unix 上ではプロセスが終了したときに joinしないと、そのプロセスはゾンビになります。新たなプロセスが開始する (または `active_children()` が呼ばれる) ときに、join されていないすべての完了プロセスが join されるので、あまり多くにはならないでしょう。また、終了したプロセスの `Process.is_alive` はそのプロセスを join します。そうは言っても、自分で開始したすべてのプロセスを明示的に join することはおそらく良いプラクティスです。

pickle/unpickle より継承する方が良い

Windows 上では `multiprocessing` の多くの型は子プロセスが使用できるようにするため pickle 化できなければなりません。しかし、パイプやキューを使用して他のプロセスへ共有オブジェクトを送ることは一般的に避けるべきです。その代わり、どこかで作成された共有リソースへのアクセスが必要なプロセスは、祖先のプロセスからそれを継承するようにプログラムを変更すべきです。

プロセスの強制終了を避ける

あるプロセスを停止するために `Process.terminate` メソッドを使用すると、そのプロセスが現在使用されている (ロック、セマフォ、パイプやキューのような) 共有リソースを破壊したり他のプロセスから利用できない状態を引き起こし易いです。

そのため、共有リソースを使用しないプロセスでのみ `Process.terminate` を使用することを考慮することがおそらく最善の方法です。

キューを使用するプロセスを join する

キューに要素を追加するプロセスは、全てのバッファされた要素が "feeder" スレッドによって下位層のパイプに対してフィードされるまで終了を待つということを覚えておいてください。(子プロセスはこの動作を避けるためにキューの `cancel_join_thread()` メソッドを呼ぶことができます。)

これはキューを使用するときに、キューに追加されたすべての要素が最終的にそのプロセスが `join` される前に削除されていることを確認する必要があることを意味します。そうしないと、そのキューに要素が追加したプロセスの終了を保証できません。デーモンではないプロセスは自動的に `join` されることも覚えておいてください。

次の例はデッドロックを引き起こします:

```
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()                # this deadlocks
    obj = queue.get()
```

修正するには最後の 2 行を入れ替えます (または単純に `p.join()` の行を削除します)。

明示的に子プロセスへリソースを渡す

Unix 上では子プロセスはグローバルなリソースを使用する親プロセスが作成した共有リソースを使用することができます。しかし、引数としてそのオブジェクトを子プロセスのコンストラクタへ渡す方が良いです。

これにより、コードが (潜在的に) Windows 互換になるだけでなく、子プロセスが生き続ける限り、そのオブジェクトは親プロセスでガベージコレクトされないことも保証されます。これは親プロセスでそのオブジェクトがガベージコレクトされるときにリソースが解放される場合に重要になるでしょう。

そのため、例えば

```
from multiprocessing import Process, Lock

def f():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

は、次のように書き直すべきです

```
from multiprocessing import Process, Lock
```

(次のページに続く)

(前のページからの続き)

```
def f(l):
    ... do something using "l" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

`sys.stdin` を file-like オブジェクトに置き換えることに注意する

`multiprocessing` は元々無条件に:

```
os.close(sys.stdin.fileno())
```

を `multiprocessing.Process.bootstrap()` メソッドの中で呼び出していました — これはプロセス内プロセス (processes-in-processes) で問題が起こしてしまいます。そこで、これは以下のように変更されました:

```
sys.stdin.close()
sys.stdin = open(os.devnull)
```

これによってプロセス同士が衝突して bad file descriptor エラーを起こすという根本的な問題は解決しましたが、アプリケーションの出力バッファを `sys.stdin()` から “file-like オブジェクト” に置き換えるという潜在的危険を持ち込んでしまいました。危険というのは、複数のプロセスが file-like オブジェクトの `close()` を呼び出すと、オブジェクトに同じデータが何度もフラッシュされ、破損してしまう可能性がある、というものです。

もし file-like オブジェクトを書いて独自のキャッシュを実装するなら、キャッシュするときに常に pid を記録しておき、pid が変わったらキャッシュを捨てることで、フォークセーフにできます。例:

```
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

より詳しい情報は [bpo-5155](#)、[bpo-5313](#)、[bpo-5331](#) を見てください

Windows

Windows には `os.fork()` がないのでいくつか追加制限があります:

さらなる pickle 化の可能性

`Process.__init__()` へ渡す全ての引数は、pickle 化できるものにしてください。これはとりわけ、Windows 上で、束縛メソッドや非束縛メソッドを直接 `target` 引数として使ってはならない、ということ意味着ます。メソッドではなく、関数を使うしかありません。

また `Process` をサブクラス化する場合、そのインスタンスが `Process.start` メソッドが呼ばれたときに pickle 化できるようにしてください。

グローバル変数

子プロセスで実行されるコードがグローバル変数にアクセスしようとする場合、子プロセスが見るその値は `Process.start` が呼ばれたときの親プロセスの値と同じではない可能性があります。

しかし、単にモジュールレベルの定数であるグローバル変数なら問題にはなりません。

メインモジュールの安全なインポート

新たな Python インタプリタによるメインモジュールのインポートが、意図しない副作用 (新たなプロセスを開始する等) を起こさずできるようにしてください。

例えば Windows で次のモジュールを実行しようとすると `RuntimeError` で失敗します:

```
from multiprocessing import Process

def foo():
    print 'hello'

p = Process(target=foo)
p.start()
```

代わりに、次のように `if __name__ == '__main__':` を使用してプログラムの ”エン트리ポイント” を保護すべきです:

```
from multiprocessing import Process, freeze_support

def foo():
    print 'hello'

if __name__ == '__main__':
    freeze_support()
    p = Process(target=foo)
    p.start()
```

(`freeze_support()` 行はプログラムが固まらずに通常どおり実行されるなら取り除けます。)

これは新たに生成された Python インタプリターがそのモジュールを安全にインポートして、モジュールの `foo()` 関数を実行します。

プールまたはマネージャーがメインモジュールで作成される場合に似たような制限が適用されます。

16.6.4 例

カスタマイズされたマネージャーやプロキシの作成方法と使用方法を紹介します:

```

#
# This module shows how to use arbitrary callables with a subclass of
# `BaseManager`.
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo(object):
    def f(self):
        print 'you called Foo.f()'
    def g(self):
        print 'you called Foo.g()'
    def _h(self):
        print 'you called Foo._h()'

# A simple generator function
def baz():
    for i in xrange(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ('next', '__next__')
    def __iter__(self):
        return self
    def next(self):
        return self._callmethod('next')
    def __next__(self):
        return self._callmethod('__next__')

# Function to return the operator module
def get_operator_module():
    return operator

##

class MyManager(BaseManager):
    pass

# register the Foo class; make `f()` and `g()` accessible via proxy
MyManager.register('Foo1', Foo)

# register the Foo class; make `g()` and `_h()` accessible via proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

```

(次のページに続く)

(前のページからの続き)

```

# register the generator function baz; use `GeneratorProxy` to make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions accessible via proxy
MyManager.register('operator', get_operator_module)

##

def test():
    manager = MyManager()
    manager.start()

    print '-' * 20

    f1 = manager.Foo1()
    f1.f()
    f1.g()
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print '-' * 20

    f2 = manager.Foo2()
    f2.g()
    f2._h()
    assert not hasattr(f2, 'f')
    assert sorted(f2._exposed_) == sorted(['g', '_h'])

    print '-' * 20

    it = manager.baz()
    for i in it:
        print '<%d>' % i,
    print

    print '-' * 20

    op = manager.operator()
    print 'op.add(23, 45) =', op.add(23, 45)
    print 'op.pow(2, 94) =', op.pow(2, 94)
    print 'op.getslice(range(10), 2, 6) =', op.getslice(range(10), 2, 6)
    print 'op.repeat(range(5), 3) =', op.repeat(range(5), 3)
    print 'op._exposed_ =', op._exposed_

##

if __name__ == '__main__':
    freeze_support()
    test()

```

Pool を使用する例です:


```

#
# A test of `multiprocessing.Pool` class
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

import multiprocessing
import time
import random
import sys

#
# Functions used by test code
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % (
        multiprocessing.current_process().name,
        func.__name__, args, result
    )

def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

def f(x):
    return 1.0 / (x-5.0)

def pow3(x):
    return x**3

def noop(x):
    pass

#
# Test code
#

def test():
    print 'cpu_count() = %d\n' % multiprocessing.cpu_count()

    #
    # Create pool

```

(次のページに続く)

(前のページからの続き)

```

#

PROCESSES = 4
print 'Creating pool with %d processes\n' % PROCESSES
pool = multiprocessing.Pool(PROCESSES)
print 'pool = %s' % pool
print

#
# Tests
#

TASKS = [(mul, (i, 7)) for i in range(10)] + \
        [(plus, (i, 8)) for i in range(10)]

results = [pool.apply_async(calculate, t) for t in TASKS]
imap_it = pool.imap(calculatestar, TASKS)
imap_unordered_it = pool.imap_unordered(calculatestar, TASKS)

print 'Ordered results using pool.apply_async():'
for r in results:
    print '\t', r.get()
print

print 'Ordered results using pool.imap():'
for x in imap_it:
    print '\t', x
print

print 'Unordered results using pool.imap_unordered():'
for x in imap_unordered_it:
    print '\t', x
print

print 'Ordered results using pool.map() --- will block till complete:'
for x in pool.map(calculatestar, TASKS):
    print '\t', x
print

#
# Simple benchmarks
#

N = 100000
print 'def pow3(x): return x**3'

t = time.time()
A = map(pow3, xrange(N))
print '\tmap(pow3, xrange(%d)):\n\t\t%s seconds' % \
      (N, time.time() - t)

```

(次のページに続く)

(前のページからの続き)

```

t = time.time()
B = pool.map(pow3, xrange(N))
print '\tpool.map(pow3, xrange(%d)):\n\t\t%s seconds' % \
      (N, time.time() - t)

t = time.time()
C = list(pool.imap(pow3, xrange(N), chunksize=N//8))
print '\tlist(pool.imap(pow3, xrange(%d), chunksize=%d)):\n\t\t%s' \
      ' seconds' % (N, N//8, time.time() - t)

assert A == B == C, (len(A), len(B), len(C))
print

L = [None] * 1000000
print 'def noop(x): pass'
print 'L = [None] * 1000000'

t = time.time()
A = map(noop, L)
print '\tmap(noop, L):\n\t\t%s seconds' % \
      (time.time() - t)

t = time.time()
B = pool.map(noop, L)
print '\tpool.map(noop, L):\n\t\t%s seconds' % \
      (time.time() - t)

t = time.time()
C = list(pool.imap(noop, L, chunksize=len(L)//8))
print '\tlist(pool.imap(noop, L, chunksize=%d)):\n\t\t%s seconds' % \
      (len(L)//8, time.time() - t)

assert A == B == C, (len(A), len(B), len(C))
print

del A, B, C, L

#
# Test error handling
#

print 'Testing error handling:'

try:
    print pool.apply(f, (5,))
except ZeroDivisionError:
    print '\tGot ZeroDivisionError as expected from pool.apply()'
else:
    raise AssertionError('expected ZeroDivisionError')

try:

```

(次のページに続く)

(前のページからの続き)

```

    print pool.map(f, range(10))
except ZeroDivisionError:
    print '\tGot ZeroDivisionError as expected from pool.map()'
else:
    raise AssertionError('expected ZeroDivisionError')

try:
    print list(pool.imap(f, range(10)))
except ZeroDivisionError:
    print '\tGot ZeroDivisionError as expected from list(pool.imap())'
else:
    raise AssertionError('expected ZeroDivisionError')

it = pool.imap(f, range(10))
for i in range(10):
    try:
        x = it.next()
    except ZeroDivisionError:
        if i == 5:
            pass
    except StopIteration:
        break
    else:
        if i == 5:
            raise AssertionError('expected ZeroDivisionError')

assert i == 9
print '\tGot ZeroDivisionError as expected from IMapIterator.next()'
print

#
# Testing timeouts
#

print 'Testing ApplyResult.get() with timeout:',
res = pool.apply_async(calculate, TASKS[0])
while 1:
    sys.stdout.flush()
    try:
        sys.stdout.write('\n\t%s' % res.get(0.02))
        break
    except multiprocessing.TimeoutError:
        sys.stdout.write('.')

print
print

print 'Testing IMapIterator.next() with timeout:',
it = pool.imap(calculatestar, TASKS)
while 1:
    sys.stdout.flush()
    try:

```

(次のページに続く)

(前のページからの続き)

```

        sys.stdout.write('\n\t%s' % it.next(0.02))
    except StopIteration:
        break
    except multiprocessing.TimeoutError:
        sys.stdout.write('.')

print
print

#
# Testing callback
#

print 'Testing callback:'

A = []
B = [56, 0, 1, 8, 27, 64, 125, 216, 343, 512, 729]

r = pool.apply_async(mul, (7, 8), callback=A.append)
r.wait()

r = pool.map_async(pow3, range(10), callback=A.extend)
r.wait()

if A == B:
    print '\tcallbacks succeeded\n'
else:
    print '\t*** callbacks failed\n\t\t%s != %s\n' % (A, B)

#
# Check there are no outstanding tasks
#

assert not pool._cache, 'cache = %r' % pool._cache

#
# Check close() methods
#

print 'Testing close():'

for worker in pool._pool:
    assert worker.is_alive()

result = pool.apply_async(time.sleep, [0.5])
pool.close()
pool.join()

assert result.get() is None

for worker in pool._pool:
    assert not worker.is_alive()

```

(次のページに続く)

(前のページからの続き)

```
print '\tclose() succeeded\n'

#
# Check terminate() method
#

print 'Testing terminate():'

pool = multiprocessing.Pool(2)
DELTA = 0.1
ignore = pool.apply(pow3, [2])
results = [pool.apply_async(time.sleep, [DELTA]) for i in range(100)]
pool.terminate()
pool.join()

for worker in pool._pool:
    assert not worker.is_alive()

print '\tterminate() succeeded\n'

#
# Check garbage collection
#

print 'Testing garbage collection:'

pool = multiprocessing.Pool(2)
DELTA = 0.1
processes = pool._pool
ignore = pool.apply(pow3, [2])
results = [pool.apply_async(time.sleep, [DELTA]) for i in range(100)]

results = pool = None

time.sleep(DELTA * 2)

for worker in processes:
    assert not worker.is_alive()

print '\tgarbage collection succeeded\n'

if __name__ == '__main__':
    multiprocessing.freeze_support()

    assert len(sys.argv) in (1, 2)

    if len(sys.argv) == 1 or sys.argv[1] == 'processes':
        print ' Using processes '.center(79, '-')
    elif sys.argv[1] == 'threads':
```

(次のページに続く)

(前のページからの続き)

```

    print ' Using threads '.center(79, '-')
    import multiprocessing.dummy as multiprocessing
else:
    print 'Usage:\n\t%s [processes | threads]' % sys.argv[0]
    raise SystemExit(2)

test()

```

ロック、コンディションやキューのような同期の例を紹介します:

```

#
# A test file for the `multiprocessing` package
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

import time, sys, random
from Queue import Empty

import multiprocessing                # may get overwritten

#### TEST_VALUE

def value_func(running, mutex):
    random.seed()
    time.sleep(random.random()*4)

    mutex.acquire()
    print '\n\t\t\t' + str(multiprocessing.current_process()) + ' has finished'
    running.value -= 1
    mutex.release()

def test_value():
    TASKS = 10
    running = multiprocessing.Value('i', TASKS)
    mutex = multiprocessing.Lock()

    for i in range(TASKS):
        p = multiprocessing.Process(target=value_func, args=(running, mutex))
        p.start()

    while running.value > 0:
        time.sleep(0.08)
        mutex.acquire()
        print running.value,
        sys.stdout.flush()
        mutex.release()

print

```

(次のページに続く)

(前のページからの続き)

```
print 'No more running processes'

#### TEST_QUEUE

def queue_func(queue):
    for i in range(30):
        time.sleep(0.5 * random.random())
        queue.put(i*i)
    queue.put('STOP')

def test_queue():
    q = multiprocessing.Queue()

    p = multiprocessing.Process(target=queue_func, args=(q,))
    p.start()

    o = None
    while o != 'STOP':
        try:
            o = q.get(timeout=0.3)
            print o,
            sys.stdout.flush()
        except Empty:
            print 'TIMEOUT'

    print

#### TEST_CONDITION

def condition_func(cond):
    cond.acquire()
    print '\t' + str(cond)
    time.sleep(2)
    print '\tchild is notifying'
    print '\t' + str(cond)
    cond.notify()
    cond.release()

def test_condition():
    cond = multiprocessing.Condition()

    p = multiprocessing.Process(target=condition_func, args=(cond,))
    print cond

    cond.acquire()
    print cond
    cond.acquire()
    print cond
```

(次のページに続く)

(前のページからの続き)

```

p.start()

print 'main is waiting'
cond.wait()
print 'main has woken up'

print cond
cond.release()
print cond
cond.release()

p.join()
print cond

#### TEST_SEMAPHORE

def semaphore_func(sema, mutex, running):
    sema.acquire()

    mutex.acquire()
    running.value += 1
    print running.value, 'tasks are running'
    mutex.release()

    random.seed()
    time.sleep(random.random()*2)

    mutex.acquire()
    running.value -= 1
    print '%s has finished' % multiprocessing.current_process()
    mutex.release()

    sema.release()

def test_semaphore():
    sema = multiprocessing.Semaphore(3)
    mutex = multiprocessing.RLock()
    running = multiprocessing.Value('i', 0)

    processes = [
        multiprocessing.Process(target=semaphore_func,
                                args=(sema, mutex, running))
        for i in range(10)
    ]

    for p in processes:
        p.start()

    for p in processes:
        p.join()

```

(次のページに続く)

(前のページからの続き)

```
#### TEST_JOIN_TIMEOUT

def join_timeout_func():
    print '\tchild sleeping'
    time.sleep(5.5)
    print '\n\tchild terminating'

def test_join_timeout():
    p = multiprocessing.Process(target=join_timeout_func)
    p.start()

    print 'waiting for process to finish'

    while 1:
        p.join(timeout=1)
        if not p.is_alive():
            break
        print '.',
        sys.stdout.flush()

#### TEST_EVENT

def event_func(event):
    print '\t%s is waiting' % multiprocessing.current_process()
    event.wait()
    print '\t%s has woken up' % multiprocessing.current_process()

def test_event():
    event = multiprocessing.Event()

    processes = [multiprocessing.Process(target=event_func, args=(event,))
                  for i in range(5)]

    for p in processes:
        p.start()

    print 'main is sleeping'
    time.sleep(2)

    print 'main is setting event'
    event.set()

    for p in processes:
        p.join()

#### TEST_SHAREDVALUES
```

(次のページに続く)

(前のページからの続き)

```

def sharedvalues_func(values, arrays, shared_values, shared_arrays):
    for i in range(len(values)):
        v = values[i][1]
        sv = shared_values[i].value
        assert v == sv

    for i in range(len(values)):
        a = arrays[i][1]
        sa = list(shared_arrays[i][:])
        assert a == sa

    print 'Tests passed'

def test_sharedvalues():
    values = [
        ('i', 10),
        ('h', -2),
        ('d', 1.25)
    ]
    arrays = [
        ('i', range(100)),
        ('d', [0.25 * i for i in range(100)]),
        ('H', range(1000))
    ]

    shared_values = [multiprocessing.Value(id, v) for id, v in values]
    shared_arrays = [multiprocessing.Array(id, a) for id, a in arrays]

    p = multiprocessing.Process(
        target=sharedvalues_func,
        args=(values, arrays, shared_values, shared_arrays)
    )
    p.start()
    p.join()

    assert p.exitcode == 0

####

def test(namespace=multiprocessing):
    global multiprocessing

    multiprocessing = namespace

    for func in [ test_value, test_queue, test_condition,
                  test_semaphore, test_join_timeout, test_event,
                  test_sharedvalues ]:

        print '\n\t##### %s\n' % func.__name__
        func()

```

(次のページに続く)

(前のページからの続き)

```

ignore = multiprocessing.active_children()          # cleanup any old processes
if hasattr(multiprocessing, '_debug_info'):
    info = multiprocessing._debug_info()
    if info:
        print info
        raise ValueError('there should be no positive refcounts left')

if __name__ == '__main__':
    multiprocessing.freeze_support()

    assert len(sys.argv) in (1, 2)

    if len(sys.argv) == 1 or sys.argv[1] == 'processes':
        print ' Using processes '.center(79, '-')
        namespace = multiprocessing
    elif sys.argv[1] == 'manager':
        print ' Using processes and a manager '.center(79, '-')
        namespace = multiprocessing.Manager()
        namespace.Process = multiprocessing.Process
        namespace.current_process = multiprocessing.current_process
        namespace.active_children = multiprocessing.active_children
    elif sys.argv[1] == 'threads':
        print ' Using threads '.center(79, '-')
        import multiprocessing.dummy as namespace
    else:
        print 'Usage:\n\t%s [processes | manager | threads]' % sys.argv[0]
        raise SystemExit(2)

    test(namespace)

```

ワーカープロセスのコレクションに対してタスクをフィードしてその結果をまとめるキューの使い方の例を紹介します:

```

#
# Simple example which uses a pool of workers to carry out some tasks.
#
# Notice that the results will probably not come out of the output
# queue in the same in the same order as the corresponding tasks were
# put on the input queue. If it is important to get the results back
# in the original order then consider using `Pool.map()` or
# `Pool.imap()` (which will save on the amount of code needed anyway).
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

import time
import random

```

(次のページに続く)

(前のページからの続き)

```

from multiprocessing import Process, Queue, current_process, freeze_support

#
# Function run by worker processes
#

def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

#
# Function used to calculate result
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % \
        (current_process().name, func.__name__, args, result)

#
# Functions referenced by tasks
#

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

#
#
#

def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # Create queues
    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)

    # Start worker processes
    for i in range(NUMBER_OF_PROCESSES):

```

(次のページに続く)

(前のページからの続き)

```

        Process(target=worker, args=(task_queue, done_queue)).start()

    # Get and print results
    print 'Unordered results:'
    for i in range(len(TASKS1)):
        print '\t', done_queue.get()

    # Add more tasks using `put()`
    for task in TASKS2:
        task_queue.put(task)

    # Get and print some more results
    for i in range(len(TASKS2)):
        print '\t', done_queue.get()

    # Tell child processes to stop
    for i in range(NUMBER_OF_PROCESSES):
        task_queue.put('STOP')

if __name__ == '__main__':
    freeze_support()
    test()

```

ワーカープロセスのプールが1つのソケットを共有してそれぞれの `SimpleHTTPServer.HTTPServer` インスタンスを実行する方法の例を紹介します。

```

#
# Example where a pool of http servers share a single listening socket
#
# On Windows this module depends on the ability to pickle a socket
# object so that the worker processes can inherit a copy of the server
# object. (We import `multiprocessing.reduction` to enable this pickling.)
#
# Not sure if we should synchronize access to `socket.accept()` method by
# using a process-shared lock -- does not seem to be necessary.
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

import os
import sys

from multiprocessing import Process, current_process, freeze_support
from BaseHTTPServer import HTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler

if sys.platform == 'win32':
    import multiprocessing.reduction    # make sockets pickable/inheritable

```

(次のページに続く)

(前のページからの続き)

```

def note(format, *args):
    sys.stderr.write('[%s]\t%s\n' % (current_process().name, format%args))

class RequestHandler(SimpleHTTPRequestHandler):
    # we override log_message() to show which process is handling the request
    def log_message(self, format, *args):
        note(format, *args)

def serve_forever(server):
    note('starting server')
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        pass

def runpool(address, number_of_processes):
    # create a single server object -- children will each inherit a copy
    server = HTTPServer(address, RequestHandler)

    # create child processes to act as workers
    for i in range(number_of_processes-1):
        Process(target=serve_forever, args=(server,)).start()

    # main process also acts as a worker
    serve_forever(server)

def test():
    DIR = os.path.join(os.path.dirname(__file__), '..')
    ADDRESS = ('localhost', 8000)
    NUMBER_OF_PROCESSES = 4

    print 'Serving at http://%s:%d using %d worker processes' % \
        (ADDRESS[0], ADDRESS[1], NUMBER_OF_PROCESSES)
    print 'To exit press Ctrl-C' + ['C', 'Break'][sys.platform=='win32']

    os.chdir(DIR)
    runpool(ADDRESS, NUMBER_OF_PROCESSES)

if __name__ == '__main__':
    freeze_support()
    test()

```

multiprocessing と *threading* を比較した簡単なベンチマークです:

```

#
# Simple benchmarks for the multiprocessing package

```

(次のページに続く)

(前のページからの続き)

```
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

import time, sys, multiprocessing, threading, Queue, gc

if sys.platform == 'win32':
    _timer = time.clock
else:
    _timer = time.time

delta = 1

#### TEST_QUEUESPEED

def queuespeed_func(q, c, iterations):
    a = '0' * 256
    c.acquire()
    c.notify()
    c.release()

    for i in xrange(iterations):
        q.put(a)

    q.put('STOP')

def test_queuespeed(Process, q, c):
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

        p = Process(target=queuespeed_func, args=(q, c, iterations))
        c.acquire()
        p.start()
        c.wait()
        c.release()

        result = None
        t = _timer()

        while result != 'STOP':
            result = q.get()

        elapsed = _timer() - t

        p.join()
```

(次のページに続く)

(前のページからの続き)

```
print iterations, 'objects passed through the queue in', elapsed, 'seconds'
print 'average number/sec:', iterations/elapsed

#### TEST_PIPESPEED

def pipe_func(c, cond, iterations):
    a = '0' * 256
    cond.acquire()
    cond.notify()
    cond.release()

    for i in xrange(iterations):
        c.send(a)

    c.send('STOP')

def test_pipespeed():
    c, d = multiprocessing.Pipe()
    cond = multiprocessing.Condition()
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

        p = multiprocessing.Process(target=pipe_func,
                                    args=(d, cond, iterations))

        cond.acquire()
        p.start()
        cond.wait()
        cond.release()

        result = None
        t = _timer()

        while result != 'STOP':
            result = c.recv()

        elapsed = _timer() - t
        p.join()

    print iterations, 'objects passed through connection in', elapsed, 'seconds'
    print 'average number/sec:', iterations/elapsed

#### TEST_SEQSPEED

def test_seqspeak(seq):
    elapsed = 0
    iterations = 1
```

(次のページに続く)

(前のページからの続き)

```
while elapsed < delta:
    iterations *= 2

    t = _timer()

    for i in xrange(iterations):
        a = seq[5]

    elapsed = _timer()-t

print iterations, 'iterations in', elapsed, 'seconds'
print 'average number/sec:', iterations/elapsed

#### TEST_LOCK

def test_lockspeed(l):
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

        t = _timer()

        for i in xrange(iterations):
            l.acquire()
            l.release()

        elapsed = _timer()-t

    print iterations, 'iterations in', elapsed, 'seconds'
    print 'average number/sec:', iterations/elapsed

#### TEST_CONDITION

def conditionspeed_func(c, N):
    c.acquire()
    c.notify()

    for i in xrange(N):
        c.wait()
        c.notify()

    c.release()

def test_conditionspeed(Process, c):
    elapsed = 0
    iterations = 1
```

(次のページに続く)

(前のページからの続き)

```

while elapsed < delta:
    iterations *= 2

    c.acquire()
    p = Process(target=conditionspeed_func, args=(c, iterations))
    p.start()

    c.wait()

    t = _timer()

    for i in xrange(iterations):
        c.notify()
        c.wait()

    elapsed = _timer()-t

    c.release()
    p.join()

print iterations * 2, 'waits in', elapsed, 'seconds'
print 'average number/sec:', iterations * 2 / elapsed

####

def test():
    manager = multiprocessing.Manager()

    gc.disable()

    print '\n\t##### testing Queue.Queue\n'
    test_queuespeed(threading.Thread, Queue.Queue(),
                    threading.Condition())
    print '\n\t##### testing multiprocessing.Queue\n'
    test_queuespeed(multiprocessing.Process, multiprocessing.Queue(),
                    multiprocessing.Condition())
    print '\n\t##### testing Queue managed by server process\n'
    test_queuespeed(multiprocessing.Process, manager.Queue(),
                    manager.Condition())
    print '\n\t##### testing multiprocessing.Pipe\n'
    test_pipespeed()

    print

    print '\n\t##### testing list\n'
    test_seqspeak(range(10))
    print '\n\t##### testing list managed by server process\n'
    test_seqspeak(manager.list(range(10)))
    print '\n\t##### testing Array("i", ..., lock=False)\n'
    test_seqspeak(multiprocessing.Array('i', range(10), lock=False))

```

(次のページに続く)

(前のページからの続き)

```

print '\n\t##### testing Array("i", ..., lock=True)\n'
test_seqspeak(multiprocessing.Array('i', range(10), lock=True))

print

print '\n\t##### testing threading.Lock\n'
test_lockspeed(threading.Lock())
print '\n\t##### testing threading.RLock\n'
test_lockspeed(threading.RLock())
print '\n\t##### testing multiprocessing.Lock\n'
test_lockspeed(multiprocessing.Lock())
print '\n\t##### testing multiprocessing.RLock\n'
test_lockspeed(multiprocessing.RLock())
print '\n\t##### testing lock managed by server process\n'
test_lockspeed(manager.Lock())
print '\n\t##### testing rlock managed by server process\n'
test_lockspeed(manager.RLock())

print

print '\n\t##### testing threading.Condition\n'
test_conditionspeed(threading.Thread, threading.Condition())
print '\n\t##### testing multiprocessing.Condition\n'
test_conditionspeed(multiprocessing.Process, multiprocessing.Condition())
print '\n\t##### testing condition managed by a server process\n'
test_conditionspeed(multiprocessing.Process, manager.Condition())

gc.enable()

if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()

```

16.7 mmap — メモリマップファイル

メモリにマップされたファイルオブジェクトは、文字列とファイルオブジェクトの両方のように振舞います。しかし通常の文字列オブジェクトとは異なり、これらは可変です。文字列が期待されるほとんどの場所で `mmap` オブジェクトを利用できます。例えば、メモリマップファイルを探索するために `re` モジュールを使うことができます。それらは可変なので、`obj[index] = 'a'` のように文字を変換できますし、スライスを使うことで `obj[i1:i2] = '...'` のように部分文字列を変換することができます。現在のファイル位置をデータの始めとする読み込みや書き込み、ファイルの異なる位置へ `seek()` することもできます。

メモリマップドファイルは Unix と Windows で異なる `mmap` コンストラクタで生成されます。どちらの場合も、更新用に開かれたファイルディスクリプタを渡さなければなりません。既存の Python ファイルオブジェクトをマップしたければ、`fileno()` メソッドを使って `fileno` パラメータの正しい値を取得してください。そうでなければ、`os.open()` 関数を使ってファイルを開けます。この関数はファイルディスクリプタを直接返します (処理が終わったら、やはりファイルを閉じる必要があります)。

注釈: 書き込み可能でバッファされたファイルへのメモリマップファイルを作りたいのであれば、まず最初にファイルの `flush()` を呼び出すべきです。これはバッファへのローカルな修正がマッピングで実際に利用可能になることを保障するために必要です。

Unix バージョンと Windows バージョンのどちらのコンストラクタについても、オプションのキーワード・パラメータとして `access` を指定することになるかもしれません。 `access` は 3 つの値の内の 1 つを受け入れます。 `ACCESS_READ` は読み込み専用、 `ACCESS_WRITE` は書き込み可能、 `ACCESS_COPY` はコピーした上での書き込みです。 `access` は Unix と Windows の両方で使用することができます。 `access` が指定されない場合、Windows の `mmap` は書き込み可能マップを返します。 3 つのアクセス型すべてに対する初期メモリ値は、指定されたファイルから得られます。 `ACCESS_READ` 型のメモリマップに対して書き込むと `TypeError` 例外を送出します。 `ACCESS_WRITE` 型のメモリマップへの書き込みはメモリと元のファイルの両方に影響を与えます。 `ACCESS_COPY` 型のメモリマップへの書き込みはメモリに影響を与えますが、元のファイルを更新することはありません。

バージョン 2.5 で変更: 無名メモリ (anonymous memory) にマップするためには `fileno` として -1 を渡し、 `length` を与えてください。

バージョン 2.6 で変更: `mmap.mmap` はこれまで `mmap` オブジェクトを生成するファクトリ関数でした。これからは `mmap.mmap` がクラスそのものになります。

```
class mmap.mmap (fileno, length[, tagname[, access[, offset ]]])
```

(Windows バージョン) ファイルハンドル `fileno` によって指定されたファイルから `length` バイトをマップして、 `mmap` オブジェクトを生成します。 `length` が現在のファイルサイズより大きな場合、ファイルサイズは `length` を含む大きさにまで拡張されます。 `length` が 0 の場合、マップの最大の長さは現在のファイルサイズになります。ただし、ファイル自体が空のときは Windows が例外を送出します (Windows では空のマップを作成することができません)。

`tagname` は、 `None` 以外で指定された場合、マップのタグ名を与える文字列となります。 Windows は同じファイルに対する様々なマップを持つことを可能にします。既存のタグの名前を指定すればそのタグがオープンされ、そうでなければこの名前新しいタグが作成されます。もしこのパラメータを省略したり `None` を与えたりしたならば、マップは名前なしで作成されます。タグ・パラメータの使用の回避は、あなたのコードを Unix と Windows の間で移植可能にしておくのを助けてくれるでしょう。

`offset` may be specified as a non-negative integer offset. `mmap` references will be relative to the offset from the beginning of the file. `offset` defaults to 0. `offset` must be a multiple of the `ALLOCATIONGRANULARITY`.

```
class mmap.mmap (fileno, length[, flags[, prot[, access[, offset ]]]])
```

(Unix バージョン) ファイルディスクリプタ `fileno` で指定されたファイルから `length` バイトをマップし、 `mmap` オブジェクトを返します。 `length` が 0 の場合、マップの最大の長さは `mmap` が呼ばれた時点でのファイルサイズになります。

`flags` はマップの種類を指定します。 `MAP_PRIVATE` はプライベートな copy-on-write(書き込み時コピー) のマップを作成します。従って、 `mmap` オブジェクトの内容への変更はこのプロセス内にのみ有効です。 `MAP_SHARED` はファイルの同じ領域をマップする他のすべてのプロセスと共有されたマップを作成します。デフォルトは `MAP_SHARED` です。

`prot` が指定された場合、希望のメモリ保護を与えます。2つの最も有用な値は、`PROT_READ` と `PROT_WRITE` です。これは、読み込み可能または書き込み可能を指定するものです。`prot` のデフォルトは `PROT_READ | PROT_WRITE` です。

`access` はオプションのキーワード・パラメータとして、`flags` と `prot` の代わりに指定してもかまいません。`flags`, `prot` と `access` の両方を指定することは間違っています。このパラメーターを使用法についての情報は、先に述べた `access` の記述を参照してください。

`offset` may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. `offset` defaults to 0. `offset` must be a multiple of `ALLOCATIONGRANULARITY` which is equal to `PAGESIZE` on Unix systems.

Mac OS X と OpenVMS において、作成された memory mapping の正当性を確実にするために `fileno` で指定されたファイルディスクリプタは内部で自動的に物理的なストレージ (physical backing store) と同期されます。

この例は `mmap` の簡潔な使い方を示すものです:

```
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write("Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)
    # read content via standard file methods
    print mm.readline() # prints "Hello Python!"
    # read content via slice notation
    print mm[:5] # prints "Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = " world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print mm.readline() # prints "Hello world!"
    # close the map
    mm.close()
```

次の例では無名マップを作り親プロセスと子プロセスの間でデータのやりとりをしてみせます:

```
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write("Hello world!")

pid = os.fork()

if pid == 0: # In a child process
```

(次のページに続く)

(前のページからの続き)

```
mm.seek(0)
print mm.readline()

mm.close()
```

メモリマップファイルオブジェクトは以下のメソッドをサポートしています:

close()

メモリマップファイルを閉じます。この呼出しの後にオブジェクトの他のメソッドの呼出すことは、`ValueError` 例外の送出を引き起こします。このメソッドは開いたファイルのクローズはしません。

find(string[, start[, end]])

オブジェクト内の `[start, end]` の範囲に含まれている部分文字列 `string` が見つかった場所の最も小さいインデックスを返します。オプションの引数 `start` と `end` はスライスに使われるときのように解釈されます。失敗したときには `-1` を返します。

flush([offset, size])

Flushes changes made to the in-memory copy of a file back to disk. Without use of this call there is no guarantee that changes are written back before the object is destroyed. If *offset* and *size* are specified, only changes to the given range of bytes will be flushed to disk; otherwise, the whole extent of the mapping is flushed. *offset* must be a multiple of the `PAGESIZE` or `ALLOCATIONGRANULARITY`.

(Windows バージョン) ゼロ以外の値が返されたら成功を、ゼロは失敗を意味します。

(Unix バージョン) ゼロの値が返されたら成功を意味します。呼び出しが失敗すると例外が送出されます。

move(dest, src, count)

オフセット `src` から始まる `count` バイトをインデックス `dest` の位置へコピーします。もし `mmap` が `ACCESS_READ` で作成されていた場合、`TypeError` 例外を発生させます。

read(num)

現在のファイル位置から最大で `num` バイト分の文字列を返します。ファイル位置は返したバイトの分だけ後ろの位置へ更新されます。

read.byte()

現在のファイル位置から長さ 1 の文字列を返します。ファイル位置は 1 だけ進みます。

readline()

現在のファイル位置から次の改行までの、1 行を返します。

resize(newsize)

マップと元ファイル (がもしあれば) のサイズを変更します。もし `mmap` が `ACCESS_READ` または `ACCESS_COPY` で作成されたならば、マップサイズの変更は `TypeError` 例外を発生させます。

rfind(string[, start[, end]])

オブジェクト内の `[start, end]` の範囲に含まれている部分文字列 `string` が見つかった場所の最も大

きいインデックスを返します。オプションの引数 *start* と *end* はスライスに使われるときのように解釈されます。失敗したときには `-1` を返します。

seek (*pos* [, *whence*])

ファイルの現在位置をセットします。 *whence* 引数はオプションであり、デフォルトは `os.SEEK_SET` つまり 0 (絶対位置) です。その他の値として、 `os.SEEK_CUR` つまり 1 (現在位置からの相対位置) と `os.SEEK_END` つまり 2 (ファイルの終わりからの相対位置) があります。

size ()

ファイルの長さを返します。メモリマップ領域のサイズより大きいかもしれません。

tell ()

ファイルポインタの現在位置を返します。

write (*string*)

メモリ内のファイルポインタの現在位置に *string* のバイト列を書き込みます。ファイル位置はバイト列が書き込まれた後の位置へ更新されます。もし `mmap` が `ACCESS_READ` で作成されていた場合、書き込み時に `TypeError` 例外を発生させるでしょう。

write.byte (*byte*)

メモリ内のファイルポインタの現在位置に単一文字の文字列 *byte* を書き込みます。ファイル位置は 1 だけ進みます。もし `mmap` が `ACCESS_READ` で作成されていた場合、書き込み時に `TypeError` 例外を発生させるでしょう。

16.8 readline — GNU readline のインタフェース

`readline` モジュールでは、補完や Python インタプリタからの履歴ファイルの読み書きを容易にするための多くの関数を定義しています。このモジュールは直接、または `rlcompleter` モジュールを介して使うことができます。`rlcompleter` モジュールは対話的プロンプトで Python 識別子の補完をサポートするものです。このモジュールを使って設定した内容は、インタプリタの対話プロンプトと、組み込み関数の `raw_input()` および `input()` `input()` が提示するプロンプトの両方の挙動に影響します。

注釈: 下層の Readline ライブラリー API は GNU readline ではなく `libedit` ライブラリーで実装される可能性があります。MacOS X では `readline` モジュールはどのライブラリーが使われているかを実行時に検出します。

`libedit` の設定ファイルは GNU readline のものとは異なります。もし設定文字列をプログラムからロードしているなら、GNU readline と `libedit` を区別するために "libedit" という文字列が `readline.__doc__` に含まれているかどうかチェックしてください。

`readline` のキーバインディングは初期化ファイルで設定できます。このファイルは、たいていはホームディレクトリに `.inputrc` という名前で置いてあります。GNU Readline マニュアルの [Readline Init File](#) を参照して、そのファイルの形式や可能な構成、Readline ライブラリ全体の機能を知ってください。

16.8.1 初期化ファイル

以下の関数は初期化ファイルならびにユーザ設定関連のものです:

`readline.parse_and_bind(string)`

string 引数で渡された最初の行を実行します。これにより下層のライブラリーの `rl_parse_and_bind()` が呼ばれます。

`readline.read_init_file([filename])`

`readline` 初期化ファイルを実行します。デフォルトのファイル名は最後に使用されたファイル名です。これにより下層のライブラリーの `rl_read_init_file()` が呼ばれます。

16.8.2 行バッファ

以下の関数は行バッファを操作します:

`readline.get_line_buffer()`

行バッファ (下層のライブラリーの `rl_line_buffer`) の現在の内容を返します。

`readline.insert_text(string)`

テキストをカーサー位置の行バッファに挿入します。これにより下層のライブラリーの `rl_insert_text()` が呼ばれますが、戻り値は無視されます。

`readline.redisplay()`

スクリーンの表示を変更して行バッファの現在の内容を反映させます。これにより下層のライブラリーの `rl_redisplay()` が呼ばれます。

16.8.3 履歴ファイル

以下の関数は履歴ファイルを操作します:

`readline.read_history_file([filename])`

`readline` 履歴ファイルを読み込み、履歴リストに追加します。デフォルトのファイル名は `~/.history` です。これにより下層のライブラリーの `read_history()` が呼ばれます。

`readline.write_history_file([filename])`

履歴リストを `readline` 履歴ファイルに保存します。既存のファイルは上書きされます。デフォルトのファイル名は `~/.history` です。これにより下層のライブラリーの `write_history()` が呼ばれます。

`readline.get_history_length()`

`readline.set_history_length(length)`

Set or return the desired number of lines to save in the history file. The `write_history_file()` function uses this value to truncate the history file, by calling `history_truncate_file()` in the underlying library. Negative values imply unlimited history file size.

16.8.4 履歴リスト

以下の関数はグローバルな履歴リストを操作します:

`readline.clear_history()`

現在の履歴をクリアします。これにより下層のライブラリーの `clear_history()` が呼ばれます。Python がこの機能をサポートするライブラリーのバージョンでコンパイルされたときのみ、この関数は存在します。

バージョン 2.4 で追加.

`readline.get_current_history_length()`

履歴に現在ある項目の数を返します。(`get_history_length()` は履歴ファイルに書かれる最大行数を返します。)

バージョン 2.3 で追加.

`readline.get_history_item(index)`

現在の履歴の `index` 番目の項目を返します。添字は 1 から始まります。これにより下層のライブラリーの `history_get()` が呼ばれます。

バージョン 2.3 で追加.

`readline.remove_history_item(pos)`

履歴から指定された位置の項目を削除します。添字は 0 から始まります。これにより下層のライブラリーの `remove_history()` が呼ばれます。

バージョン 2.4 で追加.

`readline.replace_history_item(pos, line)`

指定された位置の項目を `line` で置き換えます。添字は 0 から始まります。これにより下層のライブラリーの `replace_history_entry()` が呼ばれます。

バージョン 2.4 で追加.

`readline.add_history(line)`

最後に入力したかのように、`line` を履歴バッファに追加します。これにより下層のライブラリーの `add_history()` が呼ばれます。

16.8.5 開始フック

バージョン 2.3 で追加.

`readline.set_startup_hook([function])`

Set or remove the function invoked by the `rl_startup_hook` callback of the underlying library. If `function` is specified, it will be used as the new hook function; if omitted or `None`, any function already installed is removed. The hook is called with no arguments just before `readline` prints the first prompt.

```
readline.set_pre_input_hook([function])
```

Set or remove the function invoked by the `rl_pre_input_hook` callback of the underlying library. If *function* is specified, it will be used as the new hook function; if omitted or `None`, any function already installed is removed. The hook is called with no arguments after the first prompt has been printed and just before readline starts reading input characters. This function only exists if Python was compiled for a version of the library that supports it.

16.8.6 補完

The following functions relate to implementing a custom word completion function. This is typically operated by the Tab key, and can suggest and automatically complete a word being typed. By default, Readline is set up to be used by *rlcompleter* to complete Python identifiers for the interactive interpreter. If the *readline* module is to be used with a custom completer, a different set of word delimiters should be set.

```
readline.set_completer([function])
```

completer 関数を設定または削除します。 *function* が指定された場合、新たな completer 関数として用いられます; 省略された場合や `None` の場合、現在インストールされている completer 関数は削除されます。 completer 関数は `function(text, state)` の形式で、関数が文字列でない値を返すまで *state* を 0, 1, 2, ..., にして呼び出します。この関数は *text* から始まる補完結果として次に来そうなものを返さなければなりません。

The installed completer function is invoked by the *entry_func* callback passed to `rl_completion_matches()` in the underlying library. The *text* string comes from the first parameter to the `rl_attempted_completion_function` callback of the underlying library.

```
readline.get_completer()
```

completer 関数を取得します。 completer 関数が設定されていなければ `None` を返します。

バージョン 2.3 で追加。

```
readline.get_completion_type()
```

Get the type of completion being attempted. This returns the `rl_completion_type` variable in the underlying library as an integer.

バージョン 2.6 で追加。

```
readline.get_begidx()
```

```
readline.get_endidx()
```

Get the beginning or ending index of the completion scope. These indexes are the *start* and *end* arguments passed to the `rl_attempted_completion_function` callback of the underlying library.

```
readline.set_completer_delims(string)
```

```
readline.get_completer_delims()
```

Set or get the word delimiters for completion. These determine the start of the word to be considered for completion (the completion scope). These functions access the `rl_completer_word_break_characters` variable in the underlying library.

```
readline.set_completion_display_matches_hook([function])
```

Set or remove the completion display function. If *function* is specified, it will be used as the new completion display function; if omitted or `None`, any completion display function already installed is removed. This sets or clears the `rl_completion_display_matches_hook` callback in the underlying library. The completion display function is called as `function(substitution, [matches], longest_match_length)` once each time matches need to be displayed.

バージョン 2.6 で追加.

16.8.7 例

以下の例では、ユーザのホームディレクトリにある `.pyhist` という名前の履歴ファイルを自動的に読み書きするために、`readline` モジュールによる履歴の読み書き関数をどのように使うかを例示しています。以下のソースコードは通常、対話セッションの中で `PYTHONSTARTUP` ファイルから読み込まれ自動的に実行されることになります。

```
import os
import readline
histfile = os.path.join(os.path.expanduser("~"), ".pyhist")
try:
    readline.read_history_file(histfile)
    # default history len is -1 (infinite), which may grow unruly
    readline.set_history_length(1000)
except IOError:
    pass
import atexit
atexit.register(readline.write_history_file, histfile)
del os, histfile
```

次の例では `code.InteractiveConsole` クラスを拡張し、履歴の保存・復旧をサポートします。

```
import code
import readline
import atexit
import os

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/console-history")):
        code.InteractiveConsole.__init__(self, locals, filename)
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except IOError:
                pass
```

(次のページに続く)

(前のページからの続き)

```

        atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
        readline.set_history_length(1000)
        readline.write_history_file(histfile)

```

16.9 rlcompleter — GNU readline 向け補完関数

ソースコード: `Lib/rlcompleter.py`

`rlcompleter` モジュールでは Python の識別子やキーワードを定義した `readline` モジュール向けの補完関数を定義しています。

このモジュールが Unix プラットフォームで import され、`readline` が利用できるときには、`Completer` クラスのインスタンスが自動的に作成され、`complete()` メソッドが `readline` 補完に設定されます。

例:

```

>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline. <TAB PRESSED>
readline.__doc__          readline.get_line_buffer(  readline.read_init_file(
readline.__file__        readline.insert_text(      readline.set_completer(
readline.__name__        readline.parse_and_bind(
>>> readline.

```

`rlcompleter` モジュールは Python の対話モードで利用する為にデザインされています。ユーザは以下の命令を初期化ファイル (環境変数 `PYTHONSTARTUP` によって定義されます) に書き込むことで、Tab キーによる補完を利用できます:

```

try:
    import readline
except ImportError:
    print "Module readline not available."
else:
    import rlcompleter
    readline.parse_and_bind("tab: complete")

```

`readline` のないプラットフォームでも、このモジュールで定義される `Completer` クラスは独自の目的に使えます。

16.9.1 Completer オブジェクト

`Completer` オブジェクトは以下のメソッドを持っています:

`Completer.complete(text, state)`

`text` の `state` 番目の補完候補を返します。

もし `text` がピリオド ('.') を含まない場合、`--main--`、`--builtin--` で定義されている名前か、キーワード (`keyword` モジュールで定義されている) から補完されます。

ピリオドを含む名前の場合、副作用を出さずに名前を最後まで評価しようとします (関数を明示的に呼び出しはしませんが、`--getattr--()` を呼んでしまうことはあります) そして、`dir()` 関数でマッチする語を見つけます。式を評価中に発生した全ての例外は補足して無視され、`None` を返します。

第 17 章

プロセス間通信とネットワーク

この章で解説されるモジュールは他のプロセスと通信するメカニズムを提供します。

いくつかのモジュール、たとえば *signal* や *subprocess* は同じマシン上での 2 つのプロセス間でだけ動作します。他のモジュールはネットワークプロトコルをサポートし、2 つかそれ以上のプロセスがマシンをまたいで通信するために利用できます。

この章で解説されるモジュールのリスト:

17.1 subprocess — サブプロセス管理

バージョン 2.4 で追加.

subprocess モジュールは新しいプロセスの開始、入力/出力/エラーパイプの接続、リターンコードの取得を可能とします。このモジュールはいくつかの古いモジュールや関数を置き換えることを意図しています:

```
os.system
os.spawn*
os.popen*
popen2.*
commands.*
```

これらの古いモジュールや関数の代わりに、このモジュールをどのように使うかについては *古い関数を subprocess モジュールで置き換える* で説明します。

参考:

POSIX (Linux, BSD など) ユーザは Python 2.7 にバンドルされているバージョンよりも遥かに新しい *subprocess*³² モジュールをインストールして使うことを強くお勧めします。これは多くの状況においてより良い振る舞いをする差し替えです。

PEP 324 – subprocess モジュールを提案している PEP

17.1.1 subprocess モジュールを使う

サブプロセスを起動するのにお奨めなのは、以下の簡易関数を使うことです。それで満足できないような高度なユースケースがあるなら、背後にある *Popen* インターフェイスを直接使ってください。

`subprocess.call` (*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*)

args で指定された引数でコマンドを実行します。コマンドの完了を待って、`returncode` を返します。

ここでは引数は [よく使われる引数](#) で説明している一番良く使うものだけ示しています (ですのでシグニチャは省略して少々変わった書き方をしています)。完全な関数シグネチャは *Popen* コンストラクタと同じです。この関数は引数を直接そのインターフェースへ渡します。

例:

```
>>> subprocess.call(["ls", "-l"])
0

>>> subprocess.call("exit 1", shell=True)
1
```

警告: `shell=True` を使うことはセキュリティ上の脅威となり得ます。詳細については [よく使われる引数](#) に含まれる警告を参照してください。

注釈: この関数では `stderr=PIPE` および `stderr=PIPE` を使用しないでください。子プロセスが生成する出力の大きさによってはデッドロックしうるからです。パイプが必要なのであれば *Popen* の `communicate()` メソッドを使ってください。

`subprocess.check_call` (*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*)

指定された引数でコマンドを実行し、完了を待ちます。コマンドのリターンコードがゼロならば `return` しますが、そうでなければ *CalledProcessError* 例外を送出します。 *CalledProcessError* オブジェクトにはリターンコードが `returncode` 属性として収められています。

ここでは引数は [よく使われる引数](#) で説明している一番良く使うものだけ示しています (ですのでシグニチャは省略して少々変わった書き方をしています)。完全な関数シグネチャは *Popen* コンストラクタと同じです。この関数は引数を直接そのインターフェースへ渡します。

例:

```
>>> subprocess.check_call(["ls", "-l"])
0

>>> subprocess.check_call("exit 1", shell=True)
Traceback (most recent call last):
```

(次のページに続く)

(前のページからの続き)

```
...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status_
↳1
```

バージョン 2.5 で追加.

警告: `shell=True` を使うことはセキュリティ上の脅威となり得ます。詳細については [よく使われる引数](#) に含まれる警告を参照してください。

注釈: この関数では `stderr=PIPE` および `stdout=PIPE` を使用しないでください。子プロセスが生成する出力の大きさによってはデッドロックしうるからです。パイプが必要なのであれば [Popen](#) の `communicate()` メソッドを使ってください。

`subprocess.check_output(args, *, stdin=None, stderr=None, shell=False, universal_newlines=False)`
 引数でコマンドを実行し、その出力をバイト文字列として返します。

コマンドのリターンコードが非ゼロならば `CalledProcessError` 例外を送出します。`CalledProcessError` オブジェクトには、リターンコードが `returncode` 属性に、コマンドからの出力が `output` 属性に、それぞれ収められています。

ここでは引数は [よく使われる引数](#) で説明している一番良く使うものだけ示しています (ですのでシグネチャは省略して少々変わった書き方をしています)。完全な関数シグネチャは [Popen](#) コンストラクタとほぼ同じで、`stdout` だけはこの関数が内部利用で使うので指定は許されません。その他の引数は [Popen](#) コンストラクタにそのまま渡されます。

例:

```
>>> subprocess.check_output(["echo", "Hello World!"])
'Hello World!\n'

>>> subprocess.check_output("exit 1", shell=True)
Traceback (most recent call last):
...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status_
↳1
```

標準エラーも結果に含めるには、`stderr=subprocess.STDOUT` を使います:

```
>>> subprocess.check_output(
...     "ls non_existent_file; exit 0",
...     stderr=subprocess.STDOUT,
...     shell=True)
'ls: non_existent_file: No such file or directory\n'
```

バージョン 2.7 で追加.

警告: `shell=True` を使うことはセキュリティ上の脅威となり得ます。詳細については [よく使われる引数](#) に含まれる警告を参照してください。

注釈: この関数では `stderr=PIPE` を使用しないでください。子プロセスが生成するエラー出力の大きさによってはデッドロックしうるからです。標準エラー出力のパイプが必要なのであれば [Popen](#) の `communicate()` メソッドを使ってください。

`subprocess.PIPE`

[Popen](#) の `stdin`, `stdout`, `stderr` 引数に渡して、標準ストリームに対するパイプを開くことを指定するための特別な値。

`subprocess.STDOUT`

[Popen](#) の `stderr` 引数に渡して、標準エラーが標準出力と同じハンドルに出力されるように指定するための特殊な値です。

exception `subprocess.CallProcessError`

[check_call\(\)](#) または [check_output\(\)](#) によって実行されるプロセスが非ゼロの終了ステータスを返す場合に送出される例外です。

returncode

子プロセスの終了ステータスです。

cmd

子プロセスを spawn するために使用されるコマンドです。

output

この例外が [check_output\(\)](#) によって送出された場合は子プロセスの出力です。そうでなければ `None` です。

よく使われる引数

幅広い使用例をサポートするために、[Popen](#) コンストラクタ (とその他の簡易関数) は、多くのオプション引数を受け付けます。典型的な使用例については、これらの引数の多くはデフォルト値のままで問題ありません。一般的に必要とされる引数は以下の通りです:

`args` は全ての呼び出しに必要で、文字列あるいはプログラム引数のシーケンスでなければなりません。一般に、引数のシーケンスを渡す方が望ましいです。なぜなら、モジュールが必要な引数のエスケープやクオート (例えばファイル名中のスペースを許すこと) の面倒を見ることができるからです。単一の文字列を渡す場合、`shell` は `True` でなければなりません (以下を参照)。もしくは、その文字列は引数を指定せずに実行される単なるプログラムの名前であればなりません。

`stdin`, `stdout` および `stderr` には、実行するプログラムの標準入力、標準出力、および標準エラー出力の

ファイルハンドルをそれぞれ指定します。有効な値は *PIPE*、既存のファイルデスクリプタ (正の整数)、既存のファイルオブジェクト、そして `None` です。 *PIPE* を指定すると新しいパイプが子プロセスに向けて作られます。デフォルト設定の `None` を指定するとリダイレクトは起こりません。子プロセスのファイルハンドルはすべて親から受け継がれます。加えて、 *stderr* を *STDOUT* にすると、子プロセスの標準エラー出力からの出力は *stdout* と同じファイルハンドルに出力されます。

stdout か *stderr* がパイプで *universal_newlines* が `True` の場合 *open()* 関数への 'U' モードとして説明されている *universal_newlines* モードで動作し、全ての行終端コードが '\n' に変換されます。

shell が `True` なら、指定されたコマンドはシェルによって実行されます。あなたが Python を主として (ほとんどのシステムシェル以上の) 強化された制御フローのために使用していて、さらにシェルパイプ、ファイル名ワイルドカード、環境変数展開、~ のユーザホームディレクトリへの展開のような他のシェル機能への簡単なアクセスを望むなら、これは有用かもしれませんが。しかしながら、Python 自身が多くのシェ尔的な機能の実装を提供していることに注意してください (特に *glob*, *fnmatch*, *os.walk()*, *os.path.expandvars()*, *os.path.expanduser()*, *shutil*)。

警告: 信頼されていないソースからのサニタイズされていない入力を組み込んだシェルコマンドを実行すると、任意のコマンドを実行されることになるセキュリティ上の重大な欠陥 *シェルインジェクション* (en) に対して脆弱になります。この理由から、コマンド文字列が外部入力から構成される場合、 *shell=True* は絶対に使うべきではありません:

```
>>> from subprocess import call
>>> filename = input("What file would you like to display?\n")
What file would you like to display?
non_existent; rm -rf / #
>>> call("cat " + filename, shell=True) # Uh-oh. This will end badly...
```

shell=False はシェルに基づくすべての機能を無効にしますが、この脆弱性の影響を受けません; *shell=False* を動かすのに役立つヒントについては *Popen* コンストラクタのドキュメント中の注釈を参照してください。

shell=True を使用する場合、シェルコマンドを構築するために使用される文字列中の空白とシェルのメタ文字を適切にエスケープするために *pipes.quote()* を使用することができます。

これらのオプションは、他の全てのオプションとともに *Popen* コンストラクタのドキュメンテーションの中で、より詳細に説明されています。

Popen コンストラクタ

このモジュールの中で、根底のプロセス生成と管理は *Popen* クラスによって扱われます。簡易関数によってカバーされないあまり一般的でないケースを開発者が扱えるように、 *Popen* クラスは多くの柔軟性を提供しています。

```
class subprocess.Popen (args, bufsize=0, executable=None, stdin=None, stdout=None,
                        stderr=None, preexec_fn=None, close_fds=False, shell=False,
                        cwd=None, env=None, universal_newlines=False, startupinfo=None,
                        creationflags=0)
```

新しいプロセスで子のプログラムを実行します。Unix においては、子のプログラムを実行するために、このクラスは `os.execvp()` のような振る舞いを使用します。Windows においては、このクラスは Windows の `CreateProcess()` 関数を使用します。 [Popen](#) への引数は以下の通りです。

`args` はプログラム引数のシーケンスか、単一の文字列でなければなりません。デフォルトでは、`args` がシーケンスの場合に実行されるプログラムは `args` の最初の要素です。`args` が文字列の場合、解釈はプラットフォーム依存であり、下記に説明されます。デフォルトの挙動からの追加の違いについては `shell` および `executable` 引数を参照してください。特に明記されない限り、`args` をシーケンスとして渡すことが推奨されます。

Unix 上では、`args` が文字列の場合、その文字列は実行すべきプログラムの名前またはパスとして解釈されます。しかし、これはプログラムに引数を渡さない場合にのみ可能です。

注釈: `args` を正しくトークン化するには、`shlex.split()` が便利です。このメソッドは特に複雑な状況で活躍します:

```
>>> import shlex, subprocess
>>> command_line = raw_input()
/bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
>>> args = shlex.split(command_line)
>>> print args
['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd',
↵ "echo '$MONEY'"]
>>> p = subprocess.Popen(args) # Success!
```

特に注意すべき点は、シェル内でスペースで区切られたオプション (`-input` など) と引数 (`eggs.txt` など) はリストの別々の要素になるのに対し、シェル内で (上記のスペースを含むファイル名や `echo` コマンドのように) クォーティングやバックスラッシュエスケープが必要なものは単一のリスト要素であることです。

Windows 上では、`args` がシーケンスなら [Windows における引数シーケンスから文字列への変換](#) に記述された方法で文字列に変換されます。これは根底の `CreateProcess()` が文字列上で動作するからです。

`shell` 引数 (デフォルトでは `False`) は、実行するプログラムとしてシェルを使用するかどうかを指定します。`shell` が `True` の場合、`args` をシーケンスとしてではなく文字列として渡すことが推奨されます。

Unix で `shell=True` の場合、シェルのデフォルトは `/bin/sh` になります。`args` が文字列の場合、この文字列はシェルを介して実行されるコマンドを指定します。したがって、文字列は厳密にシェルプロンプトで打つ形式と一致しなければなりません。例えば、文字列の中にスペースを含むファイル名がある場合は、クォーティングかバックスラッシュエスケープが必要です。`args` がシーケンスの場合には、最初の要素はコマンド名を表わす文字列として、残りの要素は追加の引数としてシェルに渡されます。つまり、以下の [Popen](#) と等価ということです:


```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

Windows で `shell=True` とすると、COMSPEC 環境変数がデフォルトシェルを指定します。Windows で `shell=True` を指定する必要があるのは、実行したいコマンドがシェルに組み込みの場合だけです (例えば `dir` や `copy`)。バッチファイルやコンソールベースの実行ファイルを実行するために `shell=True` は必要ありません。

警告: `shell=True` を渡すことは、信頼されていない入力と組み合わせるとセキュリティ上の脅威となり得ます。詳細については [よく使われる引数](#) に含まれる警告を参照してください。

`bufsize` は、もしこれが与えられた場合、ビルトインの `open()` 関数の該当する引数と同じ意味をもちます: 0 はバッファされないことを意味し、1 は行ごとにバッファされることを、それ以外の正の値は (ほぼ) その大きさのバッファが使われることを意味します。負の `bufsize` はシステムのデフォルト値が使われることを意味し、通常これはバッファがすべて有効となります。 `bufsize` のデフォルト値は 0 (バッファされない) です。

注釈: パフォーマンス上の問題がある場合、`bufsize` を -1 か十分大きな正の値 (例えば 4096) に設定し、バッファを有効にすることを勧めます。

`executable` 引数は、実行される置換プログラムを指定します。これが必要になるのは極めて稀です。 `shell=False` のときは、`executable` は `args` で指定されている実行プログラムを置換します。しかし、オリジナルの `args` は依然としてプログラムに渡されます。ほとんどのプログラムは、`args` で指定されたプログラムをコマンド名として扱います。そして、それは実際に実行されたプログラムとは異なる可能性があります。Unix において、`ps` のようなユーティリティの中では、`args` 名が実行ファイルの表示名になります。 `shell=True` の場合、Unix において `executable` 引数はデフォルトの `/bin/sh` に対する置換シェルを指定します。

`stdin`, `stdout` および `stderr` には、実行するプログラムの標準入力、標準出力、および標準エラー出力のファイルハンドルをそれぞれ指定します。有効な値は [PIPE](#)、既存のファイルデスクリプタ (正の整数)、既存のファイルオブジェクト、そして `None` です。 [PIPE](#) を指定すると新しいパイプが子プロセスに向けて作られます。デフォルト設定の `None` を指定するとリダイレクトは起こりません。子プロセスのファイルハンドルはすべて親から受け継がれます。加えて、`stderr` を [STDOUT](#) にすると、子プロセスの標準エラー出力からの出力は `stdout` と同じファイルハンドルに出力されます。

`preexec_fn` に callable オブジェクトが指定されている場合、このオブジェクトは子プロセスが起動された後、プログラムが `exec` される直前に呼ばれます。(Unix のみ)

`close_fds` が `true` の場合、子プロセスが実行される前に 0, 1, 2 以外のすべてのファイルデスクリプタが閉じられます (Unix のみ)。Windows では `close_fds` が `true` の場合、すべてのファイルハンドルは子プロセスに引き継がれません。Windows の場合、`close_fds` を `true` にしながら、`stdin`, `stdout`, `stderr` を利用して標準ハンドルをリダイレクトすることはできません。

`cwd` が `None` 以外の場合、子プロセスのカレントディレクトリが実行される前に `cwd` に変更されます。

このディレクトリは実行ファイルを探す段階では考慮されませんので、プログラムのパスを `cwd` に対する相対パスで指定することはできない、ということに注意してください。

`env` が `None` 以外の場合、これは新しいプロセスでの環境変数を定義します。デフォルトでは、子プロセスは現在のプロセスの環境変数を引き継ぎます。

注釈: `env` を特定の値として与える場合、プログラムを実行するのに必要な変数全てを与えなければなりません。Windows で [side-by-side assembly](#) を実行するためには、`env` は正しい `SystemRoot` を含まなければいけません。

`universal_newlines` を `True` にすると、ファイルオブジェクト `stdout`, `stderr` が [universal newlines](#) モードのテキストファイルとして開きます。行は Unix 慣習の `'\r\n'`、古い Macintosh 慣習の `'\r'`、Windows 慣習の `'\r\n'` のいずれでも終端するでしょうが、どの外部化表現であれ Python プログラムからは `'\n'` に見えるようになります。

注釈: この機能は Python に `universal newline` がサポートされている場合 (デフォルト) にのみ有効です。また、`stdout`, `stdin` および `stderr` のファイルオブジェクトの `newlines` 属性は `communicate()` メソッドでは更新されません。

`startupinfo` は、根底の `CreateProcess` 関数に渡される [STARTUPINFO](#) オブジェクトになります。`creationflags` は、与えられるなら、[CREATE_NEW_CONSOLE](#) または [CREATE_NEW_PROCESS_GROUP](#) にできます。(Windows のみ)

例外

子プロセス内で送出された例外は、新しいプログラムの実行開始の前に親プロセスで再送出されます。さらに、この例外オブジェクトには `child_traceback` という属性が追加されています。この属性は子プロセスの視点からの `traceback` 情報が格納された文字列です。

最も一般的に起こる例外は `OSError` です。これは、たとえば存在しないファイルを実行しようとしたときなどに発生します。アプリケーションは `OSError` 例外に備えておかねばなりません。

不正な引数で `Popen` が呼ばれた場合は、`ValueError` が発生します。

呼び出されたプロセスがゼロでないリターンコードを返した場合 `check_call()` や `check_output()` は `CalledProcessError` を送出します。

セキュリティ

ほかの `popen` 関数とは異なり、この実装は決して暗黙のうちにシステムシェルを実行しません。これはシェルのメタ文字を含むすべての文字が子プロセスに安全に渡されるということを意味しています。明らかに、シェルが明示的に起動される場合は、空白とメタキャラクターがすべて適切にクオートされていることを保証するのはアプリケーションの責任です。

17.1.2 Popen オブジェクト

`Popen` クラスのインスタンスには、以下のようなメソッドがあります:

`Popen.poll()`

子プロセスが終了しているかどうかを調べます。 `returncode` 属性を設定して返します。

`Popen.wait()`

子プロセスが終了するまで待ちます。 `returncode` 属性を設定して返します。

警告: これは、子プロセスが十分な出力を生成したのに、出力先が、OS パイプバッファがそれ以上のデータを受け付けるのを待っているような場合に、デッドロックになります。これを避けるために、 `communicate()` を利用してください。

`Popen.communicate(input=None)`

プロセスと通信します: end-of-file に到達するまでデータを `stdin` に送信し、`stdout` および `stderr` からデータを受信します。プロセスが終了するまで待ちます。オプション引数 `input` には子プロセスに送られる文字列か、あるいはデータを送らない場合は `None` を指定します。

`communicate()` はタプル (`stdoutdata`, `stderrdata`) を返します。

子プロセスの標準入力にデータを送りたい場合は、`Popen` オブジェクトを `stdin=PIPE` と指定して作成しなければなりません。同じく、戻り値のタプルから `None` ではない値を取得するためには、`stdout=PIPE` かつ/または `stderr=PIPE` を指定しなければなりません。

注釈: 受信したデータはメモリにバッファされます。そのため、返されるデータが大きいかあるいは制限がないような場合はこのメソッドを使うべきではありません。

`Popen.send_signal(signal)`

`signal` シグナルを子プロセスに送ります。

注釈: Windows では、`SIGTERM` は `terminate()` のエイリアスです。 `CTRL_C_EVENT` と `CTRL_BREAK_EVENT` を、`CREATE_NEW_PROCESS_GROUP` を含む `creationflags` で始まった、プロセスに送れます。

バージョン 2.6 で追加.

`Popen.terminate()`

子プロセスを止めます。Posix OS では、このメソッドは `SIGTERM` シグナルを子プロセスに送ります。Windows では、Win32 API の `TerminateProcess()` 関数を利用して子プロセスを止めます。

バージョン 2.6 で追加.

`Popen.kill()`

子プロセスを kill します。Posix OS では SIGKILL シグナルを子プロセスに送ります。Windows では、`kill()` は `terminate()` のエイリアスです。

バージョン 2.6 で追加。

以下の属性も利用可能です:

警告: `.stdin.write`, `.stdout.read`, `.stderr.read` を利用すると、別のパイプの OS パイプバッファがいっぱいになってデッドロックする恐れがあります。これを避けるためには `communicate()` を利用してください。

`Popen.stdin`

`stdin` 引数が `PIPE` の場合、この属性には子プロセスの入力に使われるファイルオブジェクトになります。そうでない場合は `None` です。

`Popen.stdout`

`stdout` 引数が `PIPE` の場合、この属性には子プロセスの出力に使われるファイルオブジェクトになります。そうでない場合は `None` です。

`Popen.stderr`

`stderr` 引数が `PIPE` の場合、この属性には子プロセスのエラー出力に使われるファイルオブジェクトになります。そうでない場合は `None` です。

`Popen.pid`

子プロセスのプロセス ID が入ります。

`shell` 引数を `True` にセットした場合は、生成されたシェルのプロセス ID になります。

`Popen.returncode`

`poll()` か `wait()` (か、間接的に `communicate()`) から設定された、子プロセスの終了ステータスが入ります。 `None` はまだその子プロセスが終了していないことを示します。

負の値 `-N` は子プロセスがシグナル `N` により中止させられたことを示します (Unix のみ)。

17.1.3 Windows Popen ヘルパ

`STARTUPINFO` クラスと以下の定数は、Windows でいつでも利用できます。

class `subprocess.STARTUPINFO`

`Popen` の生成に使われる Windows `STARTUPINFO` 構造の部分的なサポートです。

dwFlags

特定の `STARTUPINFO` の属性が、プロセスがウィンドウを生成するときに使われるかを決定するビットフィールドです:

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_
↳ USESHOWWINDOW
```

hStdInput

dwFlags が *STARTF_USESTDHANDLES* を指定すれば、この属性がプロセスの標準入力処理です。 *STARTF_USESTDHANDLES* が指定されなければ、標準入力のデフォルトはキーボードバッファです。

hStdOutput

dwFlags が *STARTF_USESTDHANDLES* を指定すれば、この属性がプロセスの標準出力処理です。そうでなければ、この属性は無視され、標準出力のデフォルトはコンソールウィンドウのバッファです。

hStdError

dwFlags が *STARTF_USESTDHANDLES* を指定すれば、この属性がプロセスの標準エラー処理です。そうでなければ、この属性は無視され、標準エラーのデフォルトはコンソールウィンドウのバッファです。

wShowWindow

dwFlags が *STARTF_USESHOWWINDOW* を指定すれば、この属性は *ShowWindow* 関数の *nCmdShow* 引数で指定された値なら、 *SW_SHOWDEFAULT* 以外の任意のものにできます。しかし、この属性は無視されます。

この属性には *SW_HIDE* が提供されています。これは、 *Popen* が *shell=True* として呼び出されたときに使われます。

定数

subprocess モジュールは、以下の定数を公開します。

subprocess.STD_INPUT_HANDLE

標準入力デバイスです。この初期値は、コンソール入力バッファ、 *CONIN\$* です。

subprocess.STD_OUTPUT_HANDLE

標準出力デバイスです。この初期値は、アクティブコンソールスクリーン、 *CONOUT\$* です。

subprocess.STD_ERROR_HANDLE

標準エラーデバイスです。この初期値は、アクティブコンソールスクリーン、 *CONOUT\$* です。

subprocess.SW_HIDE

ウィンドウを隠します。別のウィンドウが活性化します。

subprocess.STARTF_USESTDHANDLES

追加情報を保持する、 *STARTUPINFO.hStdInput*, *STARTUPINFO.hStdOutput*, および *STARTUPINFO.hStdError* 属性を指定します。

subprocess.STARTF_USESHOWWINDOW

追加情報を保持する、 *STARTUPINFO.wShowWindow* 属性を指定します。

`subprocess.CREATE_NEW_CONSOLE`

新しいプロセスが、親プロセスのコンソールを継承する (デフォルト) のではなく、新しいコンソールを持ちます。

`Popen` が `shell=True` として生成されたとき、このフラグは必ず設定されます。

`subprocess.CREATE_NEW_PROCESS_GROUP`

新しいプロセスグループが生成されることを指定する `Popen` `creationflags` パラメタです。このフラグは、サブプロセスで `os.kill()` を使うのに必要です。

`CREATE_NEW_CONSOLE` が指定されていたら、このフラグは無視されます。

17.1.4 古い関数を `subprocess` モジュールで置き換える

この節では、”a が b になる”と書かれているものは a の代替として b が使えるということを表します。

注釈: この節で紹介されている ”a” 関数は全て、実行するプログラムが見つからないときは (概ね) 静かに終了します。それに対して ”b” 代替手段は `OSError` 例外を送出します。

また、要求された操作が非ゼロの終了コードを返した場合、`check_output()` を使用した置き換えは `CalledProcessError` で失敗します。その出力は、送出された例外の `output` 属性として利用可能です。

以下の例では、適切な関数が `subprocess` モジュールからすでにインポートされていることを前提としています。

/bin/sh シェルのバッククォートを置き換える

```
output=`mycmd myarg`
```

これは以下ようになります:

```
output = check_output(["mycmd", "myarg"])
```

シェルのパイプラインを置き換える

```
output=`dmesg | grep hda`
```

これは以下ようになります:

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close() # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
```

p2 を開始した後の p1.stdout.close() の呼び出しは、p1 が p2 の前に存在した場合に、p1 が SIGPIPE を受け取るために重要です。

あるいは、信頼された入力に対しては、シェル自身のパイプラインサポートを直接使用することもできます:

```
output=`dmesg | grep hda`
```

これは以下ようになります:

```
output=check_output("dmesg | grep hda", shell=True)
```

os.system() を置き換える

```
status = os.system("mycmd" + " myarg")
# becomes
status = subprocess.call("mycmd" + " myarg", shell=True)
```

注釈:

- このプログラムは普通シェル経由で呼び出す必要はありません。

より現実的な例ではこうなるでしょう:

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print >>sys.stderr, "Child was terminated by signal", -retcode
    else:
        print >>sys.stderr, "Child returned", retcode
except OSError as e:
    print >>sys.stderr, "Execution failed:", e
```

os.spawn 関数群を置き換える

P_NOWAIT の例:

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

P_WAIT の例:

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

シーケンスを使った例:

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

環境変数を使った例:

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

`os.popen()`, `os.popen2()`, `os.popen3()` を置き換える

```
pipe = os.popen("cmd", 'r', bufsize)
==>
pipe = Popen("cmd", shell=True, bufsize=bufsize, stdout=PIPE).stdout
```

```
pipe = os.popen("cmd", 'w', bufsize)
==>
pipe = Popen("cmd", shell=True, bufsize=bufsize, stdin=PIPE).stdin
```

```
(child_stdin, child_stdout) = os.popen2("cmd", mode, bufsize)
==>
p = Popen("cmd", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

```
(child_stdin,
 child_stdout,
 child_stderr) = os.popen3("cmd", mode, bufsize)
==>
p = Popen("cmd", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin,
 child_stdout,
 child_stderr) = (p.stdin, p.stdout, p.stderr)
```

```
(child_stdin, child_stdout_and_stderr) = os.popen4("cmd", mode,
                                                    bufsize)
==>
p = Popen("cmd", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```

Unix では、`os.popen2`、`os.popen3`、`os.popen4` は実行するコマンドとしてシーケンスも受け入れます。どちらにせよ、引数はシェルの干渉を受けることなく直接渡されます。この使い方は以下のように置き換えられます。


```
(child_stdin, child_stdout) = os.popen2(["/bin/ls", "-l"], mode,
                                         bufsize)

==>
p = Popen(["/bin/ls", "-l"], bufsize=bufsize, stdin=PIPE, stdout=PIPE)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

終了コードハンドリングは以下のように解釈します:

```
pipe = os.popen("cmd", 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print "There were some errors"

==>
process = Popen("cmd", shell=True, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print "There were some errors"
```

popen2 モジュールの関数群を置き換える

```
(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize, mode)

==>
p = Popen("somestring", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

Unix では、`popen2` は実行するコマンドとしてシーケンスも受け入れます。どちらにせよ、引数はシェルの干渉を受けることなく、直接渡されます。この使い方は、以下のように置き換えられます。

```
(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"], bufsize,
                                             mode)

==>
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

`popen2.Popen3` および `popen2.Popen4` は以下の点を除けば、基本的に `subprocess.Popen` と同じです:

- `Popen` は実行が失敗した場合に例外を送出します。
- `capturestderr` 引数は `stderr` 引数に代わりました。
- `stdin=PIPE` および `stdout=PIPE` を指定する必要があります。
- `popen2` はデフォルトですべてのファイル記述子を閉じますが、`Popen` では明示的に `close_fds=True` を指定する必要があります。

17.1.5 注釈

Windows における引数シーケンスから文字列への変換

Windows では、`args` シーケンスは以下の (MS C ランタイムで使われる規則に対応する) 規則を使って解析できる文字列に変換されます:

1. 引数は、スペースかタブのどちらかの空白で分けられます。
2. ダブルクォーテーションマークで囲まれた文字列は、空白が含まれていたとしても 1 つの引数として解釈されます。クオートされた文字列は引数に埋め込めます。
3. バックスラッシュに続くダブルクォーテーションマークは、リテラルのダブルクォーテーションマークと解釈されます。
4. バックスラッシュは、ダブルクォーテーションが続かない限り、リテラルとして解釈されます。
5. 複数のバックスラッシュにダブルクォーテーションマークが続くなら、バックスラッシュ 2 つで 1 つのバックスラッシュ文字と解釈されます。バックスラッシュの数が奇数なら、最後のバックスラッシュは規則 3 に従って続くダブルクォーテーションマークをエスケープします。

17.2 `socket` — 低レベルネットワークインターフェース

このモジュールは、Python で BSD ソケット (*socket*) インターフェースを利用するために使用します。最近の Unix システム、Windows, Max OS X, BeOS, OS/2 など、多くのプラットフォームで利用可能です。

注釈: いくつかの振る舞いはプラットフォームに依存します。オペレーティングシステムのソケット API を呼び出しているためです。

C 言語によるソケットプログラミングの基礎については、以下の資料を参照してください。An Introductory 4.3BSD Interprocess Communication Tutorial (Stuart Sechrest), An Advanced 4.3BSD Interprocess Communication Tutorial (Samuel J. Leffler 他), UNIX Programmer's Manual, Supplementary Documents 1(PS1:7 章 PS1:8 章)。ソケットの詳細については、各プラットフォームのソケット関連システムコールに関するドキュメント (Unix ではマニュアルページ、Windows では WinSock(または WinSock2) 仕様書) も参照してください。IPv6 対応の API については、[RFC 3493](#) "Basic Socket Interface Extensions for IPv6" を参照してください。

Python インターフェースは、Unix のソケット用システムコールとライブラリを、そのまま Python のオブジェクト指向スタイルに変換したものです。各種ソケット関連のシステムコールは、`socket()` 関数で生成する `socket` オブジェクトのメソッドとして実装されています。メソッドのパラメータは C のインターフェースよりも多少高水準で、例えば `read()` や `write()` メソッドではファイルオブジェクトと同様、受信時のバッファ確保や送信時の出力サイズなどは自動的に処理されます。

ソケットのアドレスは以下のように指定します: 単一の文字列は、`AF_UNIX` アドレスファミリーを示します。`(host, port)` のペアは `AF_INET` アドレスファミリーを示し、`host` は `'daring.cwi.nl'` のようなインターネットドメイン形式または `'100.50.200.5'` のような IPv4 アドレスを文字列で、`port` はポート番号を整数で指定します。`AF_INET6` アドレスファミリーは `(host, port, flowinfo, scopeid)`

の長さ 4 のタプルで示し、*flowinfo* と *scopeid* にはそれぞれ C の struct `sockaddr_in6` における `sin6_flowinfo` と `sin6_scopeid` の値を指定します。後方互換性のため、`socket` モジュールのメソッドでは `sin6_flowinfo` と `sin6_scopeid` を省略する事ができますが、*scopeid* を省略するとスコープを持った IPv6 アドレスの処理で問題が発生する場合があります。現在サポートされているアドレスファミリーは以上です。ソケットオブジェクトで利用する事のできるアドレス形式は、ソケットオブジェクトの作成時に指定したアドレスファミリーで決まります。

IPv4 アドレスのホストアドレスが空文字列の場合、`INADDR_ANY` として処理されます。また、'`<broadcast>`' の場合は `INADDR_BROADCAST` として処理されます。IPv6 では後方互換性のためこの機能は用意されていないので、IPv6 をサポートする Python プログラムでは利用しないで下さい。

IPv4/v6 ソケットの *host* 部にホスト名を指定すると、処理結果が一定ではない場合があります。これは Python は DNS から取得したアドレスのうち最初のアドレスを使用するので、DNS の処理やホストの設定によって異なる IPv4/6 アドレスを取得する場合がありますためです。常に同じ結果が必要であれば、*host* に数値のアドレスを指定してください。

バージョン 2.5 で追加: `AF_NETLINK` ソケットが `pid`, `groups` のペアで表現されます。

バージョン 2.6 で追加: Linux のみ、`AF_TIPC` アドレスファミリーを使って TIPC を利用することができます。TIPC はオープンで、IP ベースではないクラスターコンピューター環境向けのネットワークプロトコルです。アドレスはタプルで表現され、その中身はアドレスタイプに依存します。一般的なタプルの形は (`addr_type`, `v1`, `v2`, `v3` [, `scope`]) で、

- *addr_type* は `TIPC_ADDR_NAMESEQ`, `TIPC_ADDR_NAME`, `TIPC_ADDR_ID` の 1 つ。
- *scope* は `TIPC_ZONE_SCOPE`, `TIPC_CLUSTER_SCOPE`, `TIPC_NODE_SCOPE` の 1 つ。
- *addr_type* が `TIPC_ADDR_NAME` の場合、*v1* はサーバータイプ、*v2* はポート ID (the port identifier)、そして *v3* は 0 であるべきです。

addr_type が `TIPC_ADDR_NAMESEQ` の場合、*v1* はサーバータイプ、*v2* はポート番号下位 (lower port number)、*v3* はポート番号上位 (upper port number) です。

addr_type が `TIPC_ADDR_ID` の場合、*v1* はノード、*v2* は参照、*v3* は 0 であるべきです。

エラー時には例外が発生します。引数型のエラーやメモリ不足の場合には通常の例外が発生し、ソケットやアドレス関連のエラーの場合は `socket.error` が発生します。

`setblocking()` メソッドで、非ブロッキングモードを使用することができます。また、より汎用的に `settimeout()` メソッドでタイムアウトを指定する事ができます。

`socket` モジュールでは、以下の定数と関数を提供しています:

exception `socket.error`

この例外は、ソケット関連のエラーが発生した場合に送出されます。例外の値は障害の内容を示す文字列か、または `os.error` と同様な (`errno`, `string`) のペアとなります。オペレーティングシステムで定義されているエラーコードについては `errno` を参照してください。

バージョン 2.6 で変更: `socket.error` は `IOError` の子クラスになりました。

exception `socket.herror`

この例外は、C API の `gethostbyname_ex()` や `gethostbyaddr()` など、`h_errno` のようなアドレス関連のエラーが発生した場合に送出されます。

例外の値は `(h_errno, string)` のペアで、ライブラリの呼び出し結果を返します。`string` は C 関数 `hstrerror()` で取得した、`h_errno` の意味を示す文字列です。

exception `socket.gaierror`

この例外は `getaddrinfo()` と `getnameinfo()` でアドレス関連のエラーが発生した場合に送出されます。例外の値は `(error, string)` のペアで、ライブラリの呼び出し結果を返します。`string` は C 関数 `gai_strerror()` で取得した、`h_errno` の意味を示す文字列です。`error` の値は、このモジュールで定義される `EAI_*` 定数の何れかとなります。

exception `socket.timeout`

この例外は、あらかじめ `settimeout()` を呼び出してタイムアウトを有効にしてあるソケットでタイムアウトが生じた際に送出されます。例外に付属する値は文字列で、その内容は現状では常に `"timed out"` となります。

バージョン 2.3 で追加。

`socket.AF_UNIX`

`socket.AF_INET`

`socket.AF_INET6`

アドレス (およびプロトコル) ファミリーを示す定数で、`socket()` の最初の引数に指定することができます。`AF_UNIX` ファミリーをサポートしないプラットフォームでは、`AF_UNIX` は未定義となります。

`socket.SOCK_STREAM`

`socket.SOCK_DGRAM`

`socket.SOCK_RAW`

`socket.SOCK_RDM`

`socket.SOCK_SEQPACKET`

ソケットタイプを示す定数で、`socket()` の 2 番目の引数に指定することができます。(ほとんどの場合、`SOCK_STREAM` と `SOCK_DGRAM` 以外は必要ありません。)

`SO_*`

`socket.SOMAXCONN`

`MSG_*`

`SOL_*`

`IPPROTO_*`

`IPPORT_*`

`INADDR_*`

`IP_*`

`IPV6_*`

`EAI_*`

`AI_*`

`NI_*`

TCP_*

Unix のソケット・IP プロトコルのドキュメントで定義されている各種定数。ソケットオブジェクトの `setsockopt()` や `getsockopt()` で使用します。ほとんどのシンボルは Unix のヘッダファイルに従っています。一部のシンボルには、デフォルト値を定義してあります。

SIO_***RCVALL_***

Windows の `WSAIoctl()` のための定数です。この定数はソケットオブジェクトの `ioctl()` メソッドに引数として渡されます。

バージョン 2.6 で追加。

TIPC_*

TIPC 関連の定数で、C のソケット API が公開しているものにマッチします。詳しい情報は TIPC のドキュメントを参照してください。

バージョン 2.6 で追加。

socket.has_ipv6

現在のプラットフォームで IPv6 がサポートされているか否かを示す真偽値。

バージョン 2.3 で追加。

socket.create_connection(address[, timeout[, source_address]])

`address` ((`host`, `port`) ペア) で `listen` している TCP サービスに接続し、ソケットオブジェクトを返します。これは `socket.connect()` を高級にした関数です。`host` が数値でないホスト名の場合、`AF_INET` と `AF_INET6` の両方で名前解決を試み、得られた全てのアドレスに対して成功するまで接続を試みます。この関数を使って IPv4 と IPv6 に両対応したクライアントを簡単に書くことができます。

オプションの `timeout` 引数を指定すると、接続を試みる前にソケットオブジェクトのタイムアウトを設定します。`timeout` が指定されない場合、`getdefaulttimeout()` が返すデフォルトのタイムアウト設定値を利用します。

`source_address` は接続する前にバインドするソースアドレスを指定するオプション引数で、指定する場合は (`host`, `port`) の 2 要素タプルでなければなりません。`host` や `port` が "" か 0 だった場合は、OS のデフォルトの動作になります。

バージョン 2.6 で追加。

バージョン 2.7 で変更: `source_address` が追加されました。

socket.getaddrinfo(host, port[, family[, socktype[, proto[, flags]]]])

`host` / `port` 引数の指すアドレス情報を、そのサービスに接続されたソケットを作成するために必要な全ての引数が入った 5 要素のタプルに変換します。`host` はドメイン名、IPv4/v6 アドレスの文字列、または `None` です。`port` は 'http' のようなサービス名文字列、ポート番号を表す数値、または `None` です。`host` と `port` に `None` を指定すると C API に `NULL` を渡せます。

オプションの `family`、`socktype`、`proto` 引数を指定すると、返されるアドレスのリストを絞り込むことができます。デフォルトではこれらの値は 0 で、絞り込まずに全て取得することを意味します。`flags` 引数には `AI_*` 定数のうち 1 つ以上が指定でき、結果の取り方を変えることができます。デフォルト

ではこれは 0 です。例えば、`AI_NUMERICHOST` を指定するとドメイン名解決を行わないようにし、`host` がドメイン名だった場合には例外を送出します。

この関数は以下の構造をとる 5 要素のタプルのリストを返します:

```
(family, socktype, proto, canonname, sockaddr)
```

このタプルにある `family`, `socktype`, `proto` は、`socket()` 関数を呼び出す際に指定する値と同じ整数です。`AI_CANONNAME` を含んだ `flags` を指定した場合、`canonname` は `host` の canonical name を示す文字列です。そうでない場合は `canonname` は空文字列です。`sockaddr` は、ソケットアドレスを `family` に依存した形式で表すタプルで、(`AF_INET` の場合は 2 要素のタプル (`address`, `port`)、`AF_INET6` の場合は 4 要素のタプル (`address`, `port`, `flow info`, `scope id`)) `socket.connect()` メソッドに渡すためのものです。

次の例では `example.org` の 80 番ポートポートへの TCP 接続を得るためのアドレス情報を取得しようとしています。(結果は IPv6 をサポートしているかどうかで変わります):

```
>>> socket.getaddrinfo("example.org", 80, 0, 0, socket.IPPROTO_TCP)
[(10, 1, 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (2, 1, 6, '', ('93.184.216.34', 80))]
```

バージョン 2.2 で追加.

`socket.getfqdn([name])`

`name` の完全修飾ドメイン名を返します。`name` が空または省略された場合、ローカルホストを指定したとみなします。完全修飾ドメイン名の取得にはまず `gethostbyaddr()` でチェックし、次に可能であればエイリアスを調べ、名前にピリオドを含む最初の名前を値として返します。完全修飾ドメイン名を取得できない場合、`gethostbyname()` で返されるホスト名を返します。

バージョン 2.0 で追加.

`socket.gethostbyname(hostname)`

ホスト名を '100.50.200.5' のような IPv4 形式のアドレスに変換します。ホスト名として IPv4 アドレスを指定した場合、その値は変換せずにそのまま返ります。`gethostbyname()` API へのより完全なインターフェースが必要であれば、`gethostbyname_ex()` を参照してください。`gethostbyname()` は、IPv6 名前解決をサポートしていません。IPv4/ v6 のデュアルスタックをサポートする場合は `getaddrinfo()` を使用します。

`socket.gethostbyname_ex(hostname)`

ホスト名から、IPv4 形式の各種アドレス情報を取得します。戻り値は (`hostname`, `aliaslist`, `ipaddrlist`) のタプルで、`hostname` は `ip_address` で指定したホストの正式名、`aliaslist` は同じアドレスの別名のリスト (空の場合もある)、`ipaddrlist` は同じホスト上の同一インターフェースの IPv4 アドレスのリスト (ほとんどの場合は単一のアドレスのみ) を示します。`gethostbyname_ex()` は、IPv6 名前解決をサポートしていません。IPv4/v6 のデュアルスタックをサポートする場合は `getaddrinfo()` を使用します。

`socket.gethostname()`

Python インタープリタを現在実行中のマシンのホスト名を示す文字列を取得します。

実行中マシンの IP アドレスが必要であれば、`gethostbyname(gethostname())` を使用してください。この処理は実行中ホストのアドレス-ホスト名変換が可能であることを前提としていますが、常に変換可能であるとは限りません。

注意: `gethostname()` は完全修飾ドメイン名を返すとは限りません。完全修飾ドメイン名が必要であれば、`gethostbyaddr(gethostname())` としてください(下記参照)。

`socket.gethostbyaddr(ip_address)`

`(hostname, aliaslist, ipaddrlist)` のタプルを返し、`hostname` は `ip_address` で指定したホストの正式名、`aliaslist` は同じアドレスの別名のリスト(空の場合もある)、`ipaddrlist` は同じホスト上の同一インターフェースの IPv4 アドレスのリスト(ほとんどの場合は単一のアドレスのみ)を示します。完全修飾ドメイン名が必要であれば、`getfqdn()` を使用してください。`gethostbyaddr()` は、IPv4/IPv6 の両方をサポートしています。

`socket.getnameinfo(sockaddr, flags)`

ソケットアドレス `sockaddr` から、`(host, port)` のタプルを取得します。`flags` の設定に従い、`host` は完全修飾ドメイン名または数値形式アドレスとなります。同様に、`port` は文字列のポート名または数値のポート番号となります。

バージョン 2.2 で追加。

`socket.getprotobyne(protocolname)`

('icmp' のような) インターネットプロトコル名を、`socket()` の第三引数として指定する事ができる定数に変換します。これは主にソケットを "raw" モード (`SOCK_RAW`) でオープンする場合には必要ですが、通常のソケットモードでは第三引数に 0 を指定するか省略すれば正しいプロトコルが自動的に選択されます。

`socket.getservbyname(servicename[, protocolname])`

インターネットサービス名とプロトコルから、そのサービスのポート番号を取得します。省略可能なプロトコル名として、'tcp' か 'udp' のどちらかを指定することができます。指定がなければどちらのプロトコルにもマッチします。

`socket.getservbyport(port[, protocolname])`

インターネットポート番号とプロトコル名から、サービス名を取得します。省略可能なプロトコル名として、'tcp' か 'udp' のどちらかを指定することができます。指定がなければどちらのプロトコルにもマッチします。

`socket.socket([family[, type[, proto]]])`

アドレスファミリー、ソケットタイプ、プロトコル番号を指定してソケットを作成します。アドレスファミリーには `AF_INET` (デフォルト値)・`AF_INET6`・`AF_UNIX` を指定することができます。ソケットタイプには `SOCK_STREAM` (デフォルト値)・`SOCK_DGRAM`・または他の `SOCK_` 定数の何れかを指定します。プロトコル番号は通常省略するか、または 0 を指定します。

`socket.socketpair([family[, type[, proto]]])`

指定されたアドレスファミリー、ソケットタイプ、プロトコル番号から、接続されたソケットのペアを作成します。アドレスファミリー、ソケットタイプ、プロトコル番号は `socket()` 関数と同様に指定します。デフォルトのアドレスファミリーは、プラットフォームで定義されていれば `AF_UNIX`, そうでなければ `AF_INET` が使われます。

バージョン 2.4 で追加.

`socket.fromfd(fd, family, type[, proto])`

ファイル記述子 (ファイルオブジェクトの `fileno()` メソッドが返す整数) `fd` を複製して、ソケットオブジェクトを構築します。アドレスファミリとプロトコル番号は `socket()` と同様に指定します。ファイル記述子はソケットを指していなければなりません、実際にソケットであるかどうかのチェックは行っていません。このため、ソケット以外のファイル記述子を指定するとその後の処理が失敗する場合があります。この関数が必要な事はあまりありませんが、(Unix の `inet` デーモンに起動されるプログラムのように) ソケットを標準入力や標準出力として使用するプログラムでソケットオプションの取得や設定を行うために使われます。この関数で使用するソケットは、ブロッキングモードと想定しています。 利用可能: Unix

`socket.ntohl(x)`

32 ビットの正の整数のバイトオーダーを、ネットワークバイトオーダーからホストバイトオーダーに変換します。ホストバイトオーダーとネットワークバイトオーダーが一致するマシンでは、この関数は何もしません。それ以外の場合は 4 バイトのスワップを行います。

`socket.ntohs(x)`

16 ビットの正の整数のバイトオーダーを、ネットワークバイトオーダーからホストバイトオーダーに変換します。ホストバイトオーダーとネットワークバイトオーダーが一致するマシンでは、この関数は何もしません。それ以外の場合は 2 バイトのスワップを行います。

`socket.htonl(x)`

32 ビットの正の整数のバイトオーダーを、ホストバイトオーダーからネットワークバイトオーダーに変換します。ホストバイトオーダーとネットワークバイトオーダーが一致するマシンでは、この関数は何もしません。それ以外の場合は 4 バイトのスワップを行います。

`socket.htons(x)`

16 ビットの正の整数のバイトオーダーを、ホストバイトオーダーからネットワークバイトオーダーに変換します。ホストバイトオーダーとネットワークバイトオーダーが一致するマシンでは、この関数は何もしません。それ以外の場合は 2 バイトのスワップを行います。

`socket.inet_aton(ip_string)`

ドット記法による IPv4 アドレス ('123.45.67.89' など) を 32 ビットにパックしたバイナリ形式に変換し、長さ 4 の文字列オブジェクトとして返します。この関数が返す値は、標準 C ライブラリの `struct in_addr` 型を使用する関数に渡す事ができます。

`inet_aton()` はドットが 3 個以下の文字列も受け取ります; 詳細については Unix のマニュアル `inet(3)` を参照してください。

IPv4 アドレス文字列が不正であれば、`socket.error` が発生します。このチェックは、この関数で使用している C の実装 `inet_aton()` で行われます。

`inet_aton()` は、IPv6 をサポートしません。IPv4/v6 のデュアルスタックをサポートする場合は `inet_pton()` を使用します。

`socket.inet_ntoa(packed_ip)`

32 ビットにパックしたバイナリ形式の IPv4 アドレスを、ドット記法による文字列 ('123.45.67.89')

など)に変換します。この関数が返す値は、標準 C ライブラリの `struct in_addr` 型を使用する関数に渡す事ができます。

この関数に渡す文字列の長さが 4 バイト以外であれば、`socket.error` が発生します。`inet_ntoa()` は、IPv6 をサポートしません。IPv4/v6 のデュアルスタックをサポートする場合は `inet_pton()` を使用します。

`socket.inet_pton(address_family, ip_string)`

IP アドレスを、アドレスファミリ固有の文字列からパックしたバイナリ形式に変換します。`inet_pton()` は、`struct in_addr` 型 (`inet_aton()` と同様) や `struct in6_addr` を使用するライブラリやネットワークプロトコルを呼び出す際に使用することができます。

現在サポートされている `address_family` は、`AF_INET` と `AF_INET6` です。`ip_string` に不正な IP アドレス文字列を指定すると、`socket.error` が発生します。有効な `ip_string` は、`address_family` と `inet_pton()` の実装によって異なります。

利用可能: Unix (サポートしていないプラットフォームもあります)。

バージョン 2.3 で追加。

`socket.inet_ntop(address_family, packed_ip)`

パックした IP アドレス (数文字の文字列オブジェクト) を、`'7.10.0.5'` や `'5aef:2b::8'` などの標準的な、アドレスファミリ固有の文字列形式に変換します。`inet_ntop()` は (`inet_ntoa()` と同様に) `struct in_addr` 型や `struct in6_addr` 型のオブジェクトを返すライブラリやネットワークプロトコル等で使用することができます。

現在サポートされている `address_family` は、`AF_INET` と `AF_INET6` です。`packed_ip` の長さが指定したアドレスファミリで適切な長さでなければ、`ValueError` が発生します。`inet_ntop()` でエラーとなると、`socket.error` が発生します。

利用可能: Unix (サポートしていないプラットフォームもあります)。

バージョン 2.3 で追加。

`socket.getdefaulttimeout()`

新規に生成されたソケットオブジェクトの、デフォルトのタイムアウト値を浮動小数点形式の秒数で返します。タイムアウトを使用しない場合には `None` を返します。最初に `socket` モジュールがインポートされた時の初期値は `None` です。

バージョン 2.3 で追加。

`socket.setdefaulttimeout(timeout)`

新規に生成されたソケットオブジェクトの、デフォルトのタイムアウト値を浮動小数点形式の秒数で指定します。タイムアウトを使用しない場合には `None` を指定します。最初に `socket` モジュールがインポートされた時の初期値は `None` です。

バージョン 2.3 で追加。

`socket.SocketType`

ソケットオブジェクトの型を示す型オブジェクト。 `type(socket(...))` と同じです。

参考:

`SocketServer` モジュール ネットワークサーバの開発を省力化するためのクラス群。

Module `ssl` ソケットオブジェクトに対する TLS/SSL ラッパー。

17.2.1 socket オブジェクト

ソケットオブジェクトは以下のメソッドを持ちます。 `makefile()` 以外のメソッドは、Unix のソケット用システムコールに対応しています。

`socket.accept()`

接続を受け付けます。ソケットはアドレスに `bind` 済みで、`listen` 中である必要があります。戻り値は `(conn, address)` のペアで、`conn` は接続を通じてデータの送受信を行うための新しいソケットオブジェクト、`address` は接続先でソケットに `bind` しているアドレスを示します。

`socket.bind(address)`

ソケットを `address` に `bind` します。`bind` 済みのソケットを再バインドする事はできません。(`address` のフォーマットはアドレスファミリによって異なります – 前述。)

注釈: 本来、このメソッドは単一のタプルのみを引数として受け付けますが、以前は `AF_INET` アドレスを示す二つの値を指定する事ができました。これは本来の仕様ではなく、Python 2.0 以降では使用することはできません。

`socket.close()`

ソケットをクローズします。以降、このソケットでは全ての操作が失敗します。リモート端点ではキューに溜まったデータがフラッシュされた後はそれ以上のデータを受信しません。ソケットはガベージコレクション時に自動的にクローズされます。

注釈: `close()` は接続に関連付けられたリソースを解放しますが、接続をすぐに切断するとは限りません。接続を即座に切断したい場合は、 `close()` の前に `shutdown()` を呼び出してください。

`socket.connect(address)`

`address` で示されるリモートソケットに接続します。(`address` のフォーマットはアドレスファミリによって異なります — 前述。)

注釈: 本来、このメソッドは単一のタプルのみを引数として受け付けますが、以前は `AF_INET` アドレスを示す二つの値を指定する事ができました。これは本来の仕様ではなく、Python 2.0 以降では使用することはできません。

`socket.connect_ex(address)`

`connect(address)` と同様ですが、C 言語の `connect()` 関数の呼び出しでエラーが発生した場合

には例外を送出せずにエラーを戻り値として返します。(これ以外の、“host not found,”等のエラーの場合には例外が発生します。) 処理が正常に終了した場合には 0 を返し、エラー時には `errno` の値を返します。この関数は、非同期接続をサポートする場合などに使用することができます。

注釈: 本来、このメソッドは単一のタプルのみを引数として受け付けますが、以前は `AF_INET` アドレスを示す二つの値を指定する事ができました。これは本来の仕様ではなく、Python 2.0 以降では使用することはできません。

`socket.fileno()`

ソケットのファイル記述子を整数型で返します。ファイル記述子は、`select.select()` などを使用します。

Windows ではこのメソッドで返された小整数をファイル記述子を扱う箇所 (`os.fdopen()` など) で利用できません。Unix にはこの制限はありません。

`socket.getpeername()`

ソケットが接続しているリモートアドレスを返します。この関数は、リモート IPv4/v6 ソケットのポート番号を調べる場合などに使用します。`address` のフォーマットはアドレスファミリによって異なります (前述)。この関数をサポートしていないシステムも存在します。

`socket.getsockname()`

ソケット自身のアドレスを返します。この関数は、IPv4/v6 ソケットのポート番号を調べる場合などに使用します。(`address` のフォーマットはアドレスファミリによって異なります — 前述。)

`socket.getsockopt(level, optname[, buflen])`

ソケットに指定されたオプションを返します (Unix のマニュアルページ `getsockopt(2)` を参照)。`SO_*` 等のシンボルは、このモジュールで定義しています。`buflen` を省略した場合、取得するオプションは整数とみなし、整数型の値を戻り値とします。`buflen` を指定した場合、長さ `buflen` のバッファでオプションを受け取り、このバッファを文字列として返します。このバッファは、呼び出し元プログラムで `struct` モジュール等を利用して内容を読み取ることができます。

`socket.ioctl(control, option)`

プラットフォーム Windows

`ioctl()` メソッドは `WSAIoctl` システムインタフェースへの制限されたインタフェースです。詳しい情報については、[Win32 documentation](#) を参照してください。

他のプラットフォームでは一般的な `fcntl.fcntl()` と `fcntl.ioctl()` が使われるでしょう; これらの関数は第 1 引数としてソケットオブジェクトを取ります。

バージョン 2.6 で追加.

`socket.listen(backlog)`

ソケットを Listen し、接続を待ちます。引数 `backlog` には接続キューの最大の長さ (0 以上) を指定します。`backlog` の最大数はシステムに依存します (通常は 5)。最小値は必ず 0 です。

`socket.makefile([mode[, bufsize]])`

ソケットに関連付けられた ファイルオブジェクト を返します (ファイルオブジェクトについては [ファイルオブジェクト](#) を参照)。ファイルオブジェクトはその `close()` メソッドが呼ばれた際もソケットを明示的にクローズせずにソケットオブジェクトへの参照を削除するだけです。このため、ソケットは、どこからも参照されなくなってからクローズされます。

ソケットはブロッキングモードでなければなりません (タイムアウトを設定することもできません)。オプション引数の `mode` と `bufsize` には、`file()` 組み込み関数と同じ値を指定します。

注釈: Windows では、`makefile()` によって作成される file-like オブジェクトは、`subprocess.Popen()` などのファイル記述子のある file オブジェクトを期待している場所で利用することはできません。

`socket.recv(bufsize[, flags])`

ソケットからデータを受信し、文字列として返します。受信する最大バイト数は、`bufsize` で指定します。`flags` のデフォルト値は 0 です。値の意味については Unix マニュアルページの `recv(2)` を参照してください。

注釈: ハードウェアおよびネットワークの現実には最大限マッチするように、`bufsize` の値は比較的小さい 2 の累乗、たとえば 4096、にすべきです。

`socket.recvfrom(bufsize[, flags])`

ソケットからデータを受信し、結果をタプル (`string`, `address`) として返します。`string` は受信データの文字列で、`address` は送信元のアドレスを示します。オプション引数 `flags` については、Unix のマニュアルページ `recv(2)` を参照してください。デフォルトは 0 です。`(address` のフォーマットはアドレスファミリによって異なります (前述))

`socket.recvfrom_into(buffer[, nbytes[, flags]])`

ソケットからデータを受信し、そのデータを新しい文字列として返す代わりに `buffer` に書きます。戻り値は (`nbytes`, `address`) のペアで、`nbytes` は受信したデータのバイト数を、`address` はデータを送信したソケットのアドレスです。オプション引数 `flags` (デフォルト:0) の意味については、Unix マニュアルページ `recv(2)` を参照してください。`(address` のフォーマットは前述のとおりアドレスファミリに依存します。)

バージョン 2.5 で追加。

`socket.recv_into(buffer[, nbytes[, flags]])`

`nbytes` バイトまでのデータをソケットから受信して、そのデータを新しい文字列にするのではなく `buffer` に保存します。`nbytes` が指定されない (あるいは 0 が指定された) 場合、`buffer` の利用可能なサイズまで受信します。受信したバイト数を返り値として返します。オプション引数 `flags` (デフォルト:0) の意味については、Unix マニュアルページ `recv(2)` を参照してください。

バージョン 2.5 で追加。

`socket.send(string[, flags])`

ソケットにデータを送信します。ソケットはリモートソケットに接続済みでなければなりません。オプション引数 *flags* の意味は、上記 `recv()` と同じです。戻り値として、送信したバイト数を返します。アプリケーションでは、必ず戻り値をチェックし、全てのデータが送られた事を確認する必要があります。データの一部だけが送信された場合、アプリケーションで残りのデータを再送信してください。ソケットプログラミング HOWTO に、さらに詳しい情報があります。

`socket.sendall(string[, flags])`

ソケットにデータを送信します。ソケットはリモートソケットに接続済みでなければなりません。オプション引数 *flags* の意味は、上記 `recv()` と同じです。`send()` と異なり、このメソッドは *string* の全データを送信するか、エラーが発生するまで処理を続けます。正常終了の場合は `None` を返し、エラー発生時には例外が発生します。エラー発生時、送信されたバイト数を調べる事はできません。

`socket.sendto(string, address)`

`socket.sendto(string, flags, address)`

ソケットにデータを送信します。このメソッドでは接続先を *address* で指定するので、接続済みではありません。オプション引数 *flags* の意味は、上記 `recv()` と同じです。戻り値として、送信したバイト数を返します。(*address* のフォーマットはアドレスファミリによって異なります — 前述。)

`socket.setblocking(flag)`

ソケットのブロッキング・非ブロッキングモードを指定します。*flag* が 0 の場合は非ブロッキングモード、0 以外の場合はブロッキングモードとなります。全てのソケットは、初期状態ではブロッキングモードです。非ブロッキングモードでは、`recv()` メソッド呼び出し時に読み込みデータが無かったり `send()` メソッド呼び出し時にデータを処理する事ができないような場合に `error` 例外が発生します。しかし、ブロッキングモードでは呼び出しは処理が行われるまでブロックされます。`s.setblocking(0)` は `s.settimeout(0.0)` と、`s.setblocking(1)` は `s.settimeout(None)` とそれぞれ同じ意味を持ちます。

`socket.settimeout(value)`

ソケットのブロッキング処理のタイムアウト値を指定します。*value* には、正の浮動小数点で秒数を指定するか、もしくは `None` を指定します。浮動小数点値を指定した場合、操作が完了する前に *value* で指定した秒数が経過すると `timeout` が発生します。タイムアウト値に `None` を指定すると、ソケットのタイムアウトを無効にします。`s.settimeout(0.0)` は `s.setblocking(0)` と、`s.settimeout(None)` は `s.setblocking(1)` とそれぞれ同じ意味を持ちます。

バージョン 2.3 で追加。

`socket.gettimeout()`

ソケットに指定されたタイムアウト値を取得します。タイムアウト値が設定されている場合には浮動小数点型で秒数が、設定されていなければ `None` が返ります。この値は、最後に呼び出された `setblocking()` または `settimeout()` によって設定されます。

バージョン 2.3 で追加。

ソケットのブロッキングとタイムアウトについて: ソケットオブジェクトのモードは、ブロッキング・非ブロッキング・タイムアウトの何れかとなります。初期状態では常にブロッキングモードです。ブロッキングモードでは、処理が完了するまで、もしくはシステムが(接続タイムアウトなどの)エラーを返すまでブロックされます。非ブロッキングモードでは、処理を行う事ができなければ(不幸にもシステムによって異なる値の)

エラーとなります。タイムアウトモードでは、ソケットに指定したタイムアウトまで、もしくはシステムがエラーを返すまでに完了しなければ処理は失敗となります。 `setblocking()` メソッドは、 `settimeout()` の省略形式です。

内部的には、タイムアウトモードではソケットを非ブロッキングモードに設定します。ブロッキングとタイムアウトの設定は、ソケットと同じネットワーク端点へ接続するファイル記述子にも反映されます。この結果、 `makefile()` で作成したファイルオブジェクトはブロッキングモードでのみ使用することができます。これは非ブロッキングモードとタイムアウトモードでは、即座に完了しないファイル操作はエラーとなるためです。

註: `connect()` はタイムアウト設定に従います。一般的に、 `settimeout()` を `connect()` の前に呼ぶかタイムアウト値を `create_connection()` に渡すことをおすすめします。システムのネットワークスタックは Python のソケットタイムアウトの設定を無視して、自身のコネクションタイムアウトエラーを返すこともあります。

`socket.setsockopt(level, optname, value)`

ソケットのオプションを設定します (Unix のマニュアルページ `setsockopt(2)` を参照)。 `SO_*` 等のシンボルは、このモジュールで定義しています。 `value` には、整数または文字列をバッファとして指定する事ができます。文字列を指定する場合、文字列には適切なビットを設定するようにします。(`struct` モジュールを利用すれば、C の構造体を文字列にエンコードする事ができます。)

`socket.shutdown(how)`

接続の片方向、または両方向を切断します。 `how` が `SHUT_RD` の場合、以降は受信を行えません。 `how` が `SHUT_WR` の場合、以降は送信を行えません。 `how` が `SHUT_RDWR` の場合、以降は送受信を行えません。プラットフォームによっては、接続の片方向をシャットダウンすると相手側も閉じられます。(例えば、Mac OS X では、 `shutdown(SHUT_WR)` をすると、接続の相手側はもう `read` ができなくなります)

`read()` メソッドと `write()` メソッドは存在しませんので注意してください。代わりに `flags` を省略した `recv()` と `send()` を使うことができます。

ソケットオブジェクトには以下の `socket` コンストラクタに渡された値に対応した (読み出し専用) 属性があります。

`socket.family`

ソケットファミリー。

バージョン 2.5 で追加。

`socket.type`

ソケットタイプ。

バージョン 2.5 で追加。

`socket.proto`

ソケットプロトコル。

バージョン 2.5 で追加。

17.2.2 例

以下は TCP/IP プロトコルの簡単なサンプルとして、受信したデータをクライアントにそのまま返送するサーバ (接続可能なクライアントは一件のみ) と、サーバに接続するクライアントの例を示します。サーバでは、`socket() · bind() · listen() · accept()` を実行し (複数のクライアントからの接続を受け付ける場合、`accept()` を複数回呼び出します)、クライアントでは `socket()` と `connect()` だけ呼び出しています。サーバでは `sendall() / recv()` メソッドは listen 中のソケットで実行するのではなく、`accept()` で取得したソケットに対して実行している点にも注意してください。

次のクライアントとサーバは、IPv4 のみをサポートしています。

```
# Echo server program
import socket

HOST = '' # Symbolic name meaning all available interfaces
PORT = 50007 # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.sendall(data)
conn.close()
```

```
# Echo client program
import socket

HOST = 'daring.cwi.nl' # The remote host
PORT = 50007 # The same port as used by the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.sendall('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', repr(data)
```

次のサンプルは上記のサンプルとほとんど同じですが、IPv4 と IPv6 の両方をサポートしています。サーバでは、IPv4/v6 の両方ではなく、利用可能な最初のアドレスファミリーだけを listen しています。ほとんどの IPv6 対応システムでは IPv6 が先に現れるため、サーバは IPv4 には応答しません。クライアントでは名前解決の結果として取得したアドレスに順次接続を試み、最初に接続に成功したソケットにデータを送信しています。

```
# Echo server program
import socket
import sys

HOST = None # Symbolic name meaning all available interfaces
PORT = 50007 # Arbitrary non-privileged port
```

(次のページに続く)

(前のページからの続き)

```

s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                               socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except socket.error as msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except socket.error as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print 'could not open socket'
    sys.exit(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()

```

```

# Echo client program
import socket
import sys

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except socket.error as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except socket.error as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print 'could not open socket'

```

(次のページに続く)

(前のページからの続き)

```

    sys.exit(1)
s.sendall('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', repr(data)

```

最後の例は、Windows で raw socket を利用して非常にシンプルなネットワークスニファーを書きます。このサンプルを実行するには、インタフェースを操作するための管理者権限が必要です。

```

import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a package
print s.recvfrom(65565)

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)

```

この例の実行を、ほとんど間を空けずに何度も実行すると、以下のエラーが起こるかもしれません:

```
socket.error: [Errno 98] Address already in use
```

これは以前の実行がソケットを `TIME_WAIT` 状態のままにし、すぐには再利用出来ないことで起こります。

これを防ぐのに、`socket` フラグの `socket.SO_REUSEADDR` があります:

```

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))

```

`SO_REUSEADDR` フラグは、`TIME_WAIT` 状態にあるローカルソケットをそのタイムアウト期限が自然に切れるのを待つことなく再利用することをカーネルに伝えます。

17.3 ss1 — ソケットオブジェクトに対する TLS/SSL ラッパー

バージョン 2.6 で追加.

Source code: [Lib/ssl.py](#)

このモジュールは Transport Layer Security (よく "Secure Sockets Layer" という名前で知られています) 暗号化と、クライアントサイド、サーバーサイド両方のネットワークソケットのためのピア認証の仕組みを提供しています。このモジュールは OpenSSL ライブラリを利用しています。OpenSSL は、全てのモダンな Unix システム、Windows、Mac OS X、その他幾つかの OpenSSL がインストールされているプラットフォームで利用できます。

バージョン 2.7.13 で変更: Updated to support linking with OpenSSL 1.1.0

注釈: OS のソケット API に対して実装されているので、幾つかの挙動はプラットフォーム依存になるかもしれません。インストールされている OpenSSL のバージョンの違いも挙動の違いの原因になるかもしれません。例えば、TLSv1.1, TLSv1.2 は openssl version 1.0.1 以降でのみ利用出来ます。

警告: [セキュリティで考慮すべき点](#) を読まずにこのモジュールを使用しないでください。SSL のデフォルト設定はアプリケーションに十分ではないので、読まない場合はセキュリティに誤った意識を持ってしまうかもしれません。

このセクションでは、`ssl` モジュールのオブジェクトと関数の解説します。TLS, SSL, certificates に関するより一般的な情報は、末尾にある "See Also" のセクションを参照してください。

このモジュールは 1 つのクラス、`ssl.SSLSocket` を提供します。このクラスは `socket.socket` クラスを継承していて、ソケットで通信されるデータを SSL で暗号化・復号するソケットに似たラッパーになります。また、このクラスは追加で、接続の相手側からの証明書を取得する `getpeercert()` メソッド、セキュア接続で使うための暗号方式を取得する `cipher()` のようなメソッドをサポートしています。

より洗練されたアプリケーションのために、`ssl.SSLContext` クラスが設定と証明書の管理の助けとなるでしょう。それは `SSLContext.wrap_socket()` メソッドを通して SSL ソケットを作成することで引き継がれます。

17.3.1 関数、定数、例外

exception `ssl.SSLError`

(現在のところ OpenSSL ライブラリによって提供されている) 下層の SSL 実装からのエラーを伝えるための例外です。このエラーは、低レベルなネットワークの上に載っている、高レベルな暗号化と認証レイヤーでの問題を通知します。このエラーは `socket.error` のサブタイプで、`socket.error` は `IOError` のサブタイプです。`SSLError` インスタンスのエラーコードとメッセージは OpenSSL ライブラリによるものです。

library

エラーが起こった OpenSSL サブモジュールを示すニーモニック文字列で、SSL, PEM, X509 などです。取り得る値は OpenSSL のバージョンに依存します。

バージョン 2.7.9 で追加.

reason

エラーが起こった原因を示すニーモニック文字列で、`CERTIFICATE_VERIFY_FAILED` などです。取り得る値は OpenSSL のバージョンに依存します。

バージョン 2.7.9 で追加.

exception `ssl.SSLZeroReturnError`

読み出しあるいは書き込みを試みようとした際に SSL コネクションが行儀よく閉じられてしまった場合に送出される *SSL*`Error` サブクラス例外です。これは下層の転送 (read TCP) が閉じたことは意味しないことに注意してください。

バージョン 2.7.9 で追加.

exception `ssl.SSLWantReadError`

読み出しあるいは書き込みを試みようとした際に、リクエストが遂行される前に下層の TCP 転送で受け取る必要があるデータが不足した場合に *non-blocking SSL socket* によって送出される *SSL*`Error` サブクラス例外です。

バージョン 2.7.9 で追加.

exception `ssl.SSLWantWriteError`

読み出しあるいは書き込みを試みようとした際に、リクエストが遂行される前に下層の TCP 転送が送信する必要があるデータが不足した場合に *non-blocking SSL socket* によって送出される *SSL*`Error` サブクラス例外です。

バージョン 2.7.9 で追加.

exception `ssl.SSLSyscallError`

SSL ソケット上で操作を遂行しようとしていてシステムエラーが起こった場合に送出される *SSL*`Error` サブクラス例外です。残念ながら元となった `errno` 番号を調べる簡単な方法はありません。

バージョン 2.7.9 で追加.

exception `ssl.SSLEOFError`

SSL コネクションが唐突に打ち切られた際に送出される *SSL*`Error` サブクラス例外です。一般的に、このエラーが起こったら下層の転送を再利用しようと試みるべきではありません。

バージョン 2.7.9 で追加.

exception `ssl.CertificateError`

証明書のエラー (ホスト名のミスマッチのような) を通知するために送出されます。ただし、OpenSSL によって検出された場合の証明書エラーは *SSL*`Error` です。

ソケットの作成

以下に示す関数は、スタンドアロンでソケットを作りたい場合に使います。Python 2.7.9 からは、これよりもっと柔軟な *SSLContext.wrap_socket()* が使えます。

`ssl.wrap_socket(sock, keyfile=None, certfile=None, server_side=False, cert_reqs=CERT_NONE, ssl_version={see docs}, ca_certs=None, do_handshake_on_connect=True, suppress_ragged_eofs=True, ciphers=None)`
`socket.socket` のインスタンス `sock` を受け取り、`socket.socket` のサブタイプである `ssl.SSLSocket` のインスタンスを返します。 `ssl.SSLSocket` は低レイヤのソケットを SSL コンテキストでラップします。 `sock` は `SOCK_STREAM` ソケットでなければなりません; ほかのタイプのソケットはサポートされていません。

クライアントサイドソケットにおいて、コンテキストの生成は遅延されます。つまり、低レイヤのソケットがまだ接続されていない場合、コンテキストの生成はそのソケットの `connect()` メソッドが呼ばれた後に行われます。サーバーサイドソケットの場合、そのソケットに接続先が居なければそれは `listen` 用ソケットだと判断されます。 `accept()` メソッドで生成されるクライアント接続に対してのサーバーサイド SSL ラップは自動的に行われます。 `wrap_socket()` は `SSLError` を送出することがあります。

オプションの `keyfile` と `certfile` 引数は、接続のこちら側を識別するために利用される証明書を含むファイルを指定します。証明書がどのように `certfile` に格納されるかについてのより詳しい情報は、[証明書](#) を参照してください。

`server_side` 引数は真偽値で、このソケットがサーバーサイドとクライアントサイドのどちらの動作をするのかを指定します。

`cert_reqs` 引数は、接続の相手側からの証明書を必要とするかどうかと、それを検証 (validate) するかどうかを指定します。これは次の 3 つの定数のどれかで無ければなりません: `CERT_NONE` (証明書は無視されます), `CERT_OPTIONAL` (必要としないが、提供された場合は検証する), `CERT_REQUIRED` (証明書を必要とし、検証する)。もしこの引数が `CERT_NONE` 以外だった場合、`ca_certs` 引数は CA 証明書ファイルを指定していなければなりません。

`ca_certs` ファイルは、接続の相手側から渡された証明書を検証するために使う、一連の CA 証明書を結合したものを含んでいます。このファイル内にどう証明書を並べるかについての詳しい情報は [証明書](#) を参照してください。

`ssl_version` 引数は、使用する SSL プロトコルのバージョンを指定します。通常、サーバー側が特定のプロトコルバージョンを選び、クライアント側はサーバーの選んだプロトコルを受け入れなければなりません。ほとんどのバージョンは他のバージョンと互換性がありません。もしこの引数が指定されなかった場合、デフォルトは `PROTOCOL_SSLv23` になります。このバージョンは、できるだけの互換性を確保するように選ばれています。

次のテーブルは、どのクライアント側のバージョンがどのサーバー側のバージョンに接続できるかを示しています:

<i>client / server</i>	SSLv2	SSLv3	SSLv23	TLSv1	TLSv1.1	TLSv1.2
<i>SSLv2</i>	yes	no	yes	no	no	no
<i>SSLv3</i>	no	yes	yes	no	no	no
<i>SSLv23</i> ^{*1}	no	yes	yes	yes	yes	yes
<i>TLSv1</i>	no	no	yes	yes	no	no
<i>TLSv1.1</i>	no	no	yes	no	yes	no
<i>TLSv1.2</i>	no	no	yes	no	no	yes

脚注

注釈: どの接続が成功するかは、OpenSSL のバージョンに依存して大きく変わります。例えば、OpenSSL 1.0.0 以前は、SSLv23 クライアントは常に SSLv2 接続を試みていました。

The *ciphers* parameter sets the available ciphers for this SSL object. It should be a string in the [OpenSSL cipher list format](#).

`do_handshake_on_connect` 引数は、`socket.connect()` の後に自動的に SSL ハンドシェイクを行うか、それともアプリケーションが明示的に `SSLSocket.do_handshake()` メソッドを実行するかを指定します。`SSLSocket.do_handshake()` を明示的に呼び出すことで、ハンドシェイクによるソケット I/O のブロッキング動作を制御できます。

`suppress_ragged_eofs` 引数は、`SSLSocket.read()` メソッドが、接続先から予期しない EOF を受け取った時に通知する方法を指定します。`True` (デフォルト) の場合、下位のソケットレイヤーから予期せぬ EOF エラーが来た場合、通常の EOF (空のバイト列オブジェクト) を返します。`False` の場合、呼び出し元に例外を投げて通知します。

バージョン 2.7 で変更: 新しいオプション引数 *ciphers*

コンテキストの作成

コンビニエンス関数が、共通の目的で使用される `SSLContext` オブジェクトを作成するのに役立ちます。

`ssl.create_default_context` (*purpose*=`Purpose.SERVER_AUTH`, *cafile*=`None`, *capath*=`None`, *cadata*=`None`)

新規の `SSLContext` オブジェクトを、与えられた *purpose* のデフォルト設定で返します。設定は `ssl` モジュールで選択され、通常は `SSLContext` のコンストラクタを直接呼び出すよりも高いセキュリティレベルを表現します。

cafile, *capath*, *cadata* は証明書の検証で信用するオプションの CA 証明書で、`SSLContext.load_verify_locations()` のものと同じです。これら 3 つ全てが `None` であれば、この関数は代わりにシステムのデフォルトの CA 証明書を信用して選択することが出来ます。

^{*1} TLS 1.3 protocol will be available with `PROTOCOL_SSLv23` in OpenSSL \geq 1.1.1. There is no dedicated PROTOCOL constant for just TLS 1.3.

これで作られる設定はこうになります: RC4 を除く、高レベルで、未認証のものを含まない暗号化一式と `PROTOCOL_SSLv23`, `OP_NO_SSLv2`, `OP_NO_SSLv3`。 `purpose` に `SERVER_AUTH` を渡すと、 `verify_mode` には `CERT_REQUIRED` がセットされ、 (`cafile`, `capath`, `cadata` のいずれかが与えられれば) CA 証明書がロードされるかまたはデフォルトの CA 証明書をロードするために `SSLContext.load_default_certs()` が使われます。

注釈: プロトコル、オプション、暗号その他設定はもっと制限された、過去の廃れたものを含まない値にいつでも出来るでしょう。この値は互換性と安全性の公平なバランスを表明しています。

もしもあなたのアプリケーションがそのような設定を必要とするのであれば、 `SSLContext` を作ってあなた自身の設定を適用すべきです。

注釈: ある種の古いクライアントやサーバが接続しようと試みてきた場合に、この関数で作られた `SSLContext` が "Protocol or cipher suite mismatch" で始まるエラーを起こすのを目撃したらそれは、この関数が `OP_NO_SSLv3` を使って除外している SSL 3.0 しかサポートしていないのでしょう。SSL 3.0 は 完璧にぶっ壊れている ことが広く知られています。それでもまだこの関数を使って、ただし SSL 3.0 接続を許可したいと望むならば、これをこのように再有効化出来ます:

```
ctx = ssl.create_default_context(Purpose.CLIENT_AUTH)
ctx.options |= ~ssl.OP_NO_SSLv3
```

バージョン 2.7.9 で追加.

バージョン 2.7.10 で変更: デフォルトの暗号設定から RC4 が除かれました。

バージョン 2.7.13 で変更: デフォルトの暗号化文字列に ChaCha20/Poly1305 が追加されました。

デフォルトの暗号化文字列から 3DES が除かれました。

`ssl.HTTPSVerifyCertificates(enable=True)`

Specifies whether or not server certificates are verified when creating client HTTPS connections without specifying a particular SSL context.

Starting with Python 2.7.9, `httplib` and modules which use it, such as `urllib2` and `xmllrpclib`, default to verifying remote server certificates received when establishing client HTTPS connections. This default verification checks that the certificate is signed by a Certificate Authority in the system trust store and that the Common Name (or Subject Alternate Name) on the presented certificate matches the requested host.

Setting `enable` to `True` ensures this default behaviour is in effect.

Setting `enable` to `False` reverts the default HTTPS certificate handling to that of Python 2.7.8 and earlier, allowing connections to servers using self-signed certificates, servers using certificates signed by a Certificate Authority not present in the system trust store, and servers where the hostname does not match the presented server certificate.

The leading underscore on this function denotes that it intentionally does not exist in any implementation of Python 3 and may not be present in all Python 2.7 implementations. The portable approach to bypassing certificate checks or the system trust store when necessary is for tools to enable that on a case-by-case basis by explicitly passing in a suitably configured SSL context, rather than reverting the default behaviour of the standard library client modules.

バージョン 2.7.12 で追加.

参考:

- [CVE-2014-9365](#) – HTTPS man-in-the-middle attack against Python clients using default settings
- [PEP 476](#) – Enabling certificate verification by default for HTTPS
- [PEP 493](#) – HTTPS verification migration tools for Python 2.7

乱数生成

バージョン 2.7.13 で非推奨: OpenSSL は `ssl.RAND_pseudo_bytes()` を廃止しました。代わりに `ssl.RAND_bytes()` を使用してください。

`ssl.RAND_status()`

SSL 擬似乱数生成器が十分なランダム性 (randomness) を受け取っている時に `True` を、それ以外の場合は `False` を返します。 `ssl.RAND_egd()` と `ssl.RAND_add()` を使って擬似乱数生成機にランダム性を加えることができます。

`ssl.RAND_egd(path)`

もしエントロピー収集デーモン (EGD=entropy-gathering daemon) が動いていて、`path` が EGD へのソケットのパスだった場合、この関数はそのソケットから 256 バイトのランダム性を読み込み、SSL 擬似乱数生成器にそれを渡すことで、生成される暗号鍵のセキュリティを向上させることができます。これは、より良いランダム性のソースが無いシステムでのみ必要です。

エントロピー収集デーモンについては、<http://egd.sourceforge.net/> や <http://prngd.sourceforge.net/> を参照してください。

利用出来る環境: LibreSSL および 1.1.0 を超えるバージョンの OpenSSL では利用できません。

`ssl.RAND_add(bytes, entropy)`

与えられた `bytes` を SSL 擬似乱数生成器に混ぜます。 `entropy` 引数 (float 値) は、その文字列に含まれるエントロピーの下限 (lower bound) です。(なので、いつでも 0.0 を使うことができます。) エントロピーのソースについてのより詳しい情報は、[RFC 1750](#) を参照してください。

証明書の取り扱い

`ssl.match_hostname(cert, hostname)`

(`SSLSocket.getpeercert()` が返してきたようなデコードされたフォーマットの) `cert` が、与えられた `hostname` に合致するかを検証します。HTTPS サーバの身元をチェックするために適用されるルールは [RFC 2818](#), [RFC 6125](#) で概説されているものです。ただし IP アドレスによるものは現在サポー

トされていません。HTTPS に加え、この関数は他の SSL ベースのプロトコル、例えば FTPS, IMAPS, POPS などのサーバの身元をチェックするのに相応しいはずです。

失敗すれば `CertificateError` が送出されます。成功すれば、この関数は何も返しません:

```
>>> cert = {'subject': (((('commonName', 'example.com'),),),)}
>>> ssl.match_hostname(cert, "example.com")
>>> ssl.match_hostname(cert, "example.org")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/py3k/Lib/ssl.py", line 130, in match_hostname
ssl.CertificateError: hostname 'example.org' doesn't match 'example.com'
```

バージョン 2.7.9 で追加.

`ssl.cert_time_to_seconds(cert_time)`

`cert_time` として証明書内の "notBefore" や "notAfter" の "%b %d %H:%M:%S %Y %Z" strftime フォーマット (C locale) 日付を渡すと、エポックからの積算秒を返します。

例です. :

```
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan  5 09:34:43 2018 GMT")
>>> timestamp
1515144883
>>> from datetime import datetime
>>> print(datetime.utcfromtimestamp(timestamp))
2018-01-05 09:34:43
```

"notBefore" や "notAfter" の日付には GMT を使わなければなりません ([RFC 5280](#)).

バージョン 2.7.9 で変更: 入力文字列に指定された 'GMT' タイムゾーンを UTC として解釈するようになりました。以前はローカルタイムで解釈していました。また、整数を返すようになりました (入力に含まれる秒の端数を含まない)。

`ssl.get_server_certificate(addr, ssl_version=PROTOCOL_SSLv23, ca_certs=None)`

SSL で保護されたサーバのアドレス `addr` を (`hostname`, `port-number`) の形で受け取り、そのサーバから証明書を取得し、それを PEM エンコードされた文字列として返します。 `ssl_version` が指定された場合は、サーバに接続を試みるときにそのバージョンの SSL プロトコルを利用します。 `ca_certs` が指定された場合、それは `wrap_socket()` の同名の引数と同じフォーマットで、ルート証明書のリストを含むファイルでなければなりません。この関数はサーバ証明書をルート証明書リストに対して認証し、認証が失敗した場合にこの関数も失敗します。

バージョン 2.7.9 で変更: この関数は IPv6 互換になりました。 `ssl_version` のデフォルトが、最近のサーバへの最大限の互換性のために `PROTOCOL_SSLv3` から `PROTOCOL_SSLv23` に変更されました。

`ssl.DER_cert_to_PEM_cert(DER_cert_bytes)`

DER エンコードされたバイト列として与えられた証明書から、PEM エンコードされたバージョンの同じ証明書を返します。

`ssl.PEM_cert_to_DER_cert (PEM_cert_string)`

PEM 形式の ASCII 文字列として与えられた証明書から、同じ証明書を DER エンコードしたバイト列を返します。

`ssl.get_default_verify_paths ()`

OpenSSL デフォルトの `cafile`, `capath` を指すパスを名前付きタプルで返します。パスは `SSLContext.set_default_verify_paths ()` で使われるものと同じです。戻り値は *named tuple* `DefaultVerifyPaths` です:

- `cafile` - `cafile` の解決済みパス、またはファイルが存在しない場合は `None`
- `capath` - `capath` の解決済みパス、またはディレクトリが存在しない場合は `None`
- `openssl_cafile_env` - `cafile` を指す OpenSSL の環境変数
- `openssl_cafile` - OpenSSL にハードコードされた `cafile` のパス
- `openssl_capath_env` - `capath` を指す OpenSSL の環境変数
- `openssl_capath` - OpenSSL にハードコードされた `capath` のパス

利用出来る環境: LibreSSL では環境変数 `openssl_cafile_env` と `openssl_capath_env` が無視されます

バージョン 2.7.9 で追加.

`ssl.enum_certificates (store_name)`

Windows のシステム証明書ストアより証明書を抽出します。 `store_name` は `CA`, `ROOT`, `MY` のうちどれか一つでしょう。Windows は追加の証明書ストアを提供しているかもしれません。

この関数はタプル (`cert_bytes`, `encoding_type`, `trust`) のリストで返します。 `encoding_type` は `cert_bytes` のエンコーディングを表します。 X.509 ASN.1 に対する `x509_asn` か PKCS#7 ASN.1 データに対する `pkcs_7_asn` のいずれかです。 `trust` は、証明書の目的を、OIDS を内容に持つ `set` として表すか、または証明書が全ての目的で信頼出来るならば `True` です。

例:

```
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509_asn', {'1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2'}),
 (b'data...', 'x509_asn', True)]
```

利用出来る環境: Windows.

バージョン 2.7.9 で追加.

`ssl.enum_crls (store_name)`

Windows のシステム証明書ストアより CRLs を抽出します。 `store_name` は `CA`, `ROOT`, `MY` のうちどれか一つでしょう。Windows は追加の証明書ストアを提供しているかもしれません。

この関数はタプル (`cert_bytes`, `encoding_type`, `trust`) のリストで返します。 `encoding_type` は `cert_bytes` のエンコーディングを表します。 X.509 ASN.1 に対する `x509_asn` か PKCS#7 ASN.1 データに対する `pkcs_7_asn` のいずれかです。

利用出来る環境: Windows.

バージョン 2.7.9 で追加.

定数

`ssl.CERT_NONE`

`SSLContext.verify_mode` または `wrap_socket()` の `cert_reqs` パラメータに使用する値です。このモード (これがデフォルトです) では、ソケット接続先からの証明書やその認証を必要としません。接続先から証明書を受け取っても検証は試みられません。

このドキュメントの下の方の、[セキュリティで考慮すべき点](#) に関する議論を参照してください。

`ssl.CERT_OPTIONAL`

`SSLContext.verify_mode` または `wrap_socket()` の `cert_reqs` パラメータに使用する値です。このモードでは、ソケット接続先からの証明書やその認証を必要としませんが、証明書が提供されれば検証が試みられ、検証失敗時には `SSLError` が送出されます。

この設定では、正当な CA 証明書のセットを `SSLContext.load_verify_locations()` または `wrap_socket()` の `ca_certs` パラメータのどちらかに渡す必要があります。

`ssl.CERT_REQUIRED`

`SSLContext.verify_mode` または `wrap_socket()` の `cert_reqs` パラメータに使用する値です。このモードでは、ソケット接続先からの証明書やその認証を必要とされ、証明書が提供されないかその検証失敗時には `SSLError` が送出されます。

この設定では、正当な CA 証明書のセットを `SSLContext.load_verify_locations()` または `wrap_socket()` の `ca_certs` パラメータのどちらかに渡す必要があります。

`ssl.VERIFY_DEFAULT`

`SSLContext.verify_flags` に渡せる値です。このモードでは、証明書失効リスト (CRLs) はチェックされません。デフォルトでは OpenSSL は CRLs を必要としませんし検証にも使いません。

バージョン 2.7.9 で追加.

`ssl.VERIFY_CRL_CHECK_LEAF`

`SSLContext.verify_flags` に渡せる値です。このモードでは、接続先の証明書のチェックのみで仲介の CA 証明書はチェックしません。接続先証明書の発行者 (その CA の直接の祖先) によって署名された妥当な CRL が必要です。 `SSLContext.load_verify_locations` が相応しいものをロードしていなければ、検証は失敗するでしょう。

バージョン 2.7.9 で追加.

`ssl.VERIFY_CRL_CHECK_CHAIN`

`SSLContext.verify_flags` に渡せる値です。このモードでは、接続先の証明書チェーン内の全ての証明書についての CRLs がチェックされます。

バージョン 2.7.9 で追加.

ssl.VERIFY_X509_STRICT

`SSLContext.verify_flags` に渡せる値で、壊れた X.509 証明書に対するワークアラウンドを無効にします。

バージョン 2.7.9 で追加。

ssl.VERIFY_X509_TRUSTED_FIRST

`SSLContext.verify_flags` に渡せる値です。OpenSSL に対し、証明書検証のために信頼チェーンを構築する際、信頼出来る証明書を選ぶように指示します。これはデフォルトで有効にされています。

バージョン 2.7.10 で追加。

ssl.PROTOCOL_TLS

チャンネル暗号化プロトコルとして、クライアントとサーバの両方がサポートする中の、プロトコルバージョンが最も大きなものを選択します。その名前にも関わらず、このオプションは "SSL" とともに "TLS" プロトコルも選択できます。

バージョン 2.7.13 で追加。

ssl.PROTOCOL_SSLv23

Alias for `PROTOCOL_TLS`.

バージョン 2.7.13 で非推奨: Use `PROTOCOL_TLS` instead.

ssl.PROTOCOL_SSLv2

チャンネル暗号化プロトコルとして SSL バージョン 2 を選択します。

このプロトコルは、OpenSSL が `OPENSSL_NO_SSL2` フラグが有効な状態でコンパイルされている場合には利用できません。

警告: SSL version 2 は非セキュアです。このプロトコルは強く非推奨です。

バージョン 2.7.13 で非推奨: OpenSSL は SSLv2 へのサポートを打ち切りました。

ssl.PROTOCOL_SSLv3

チャンネル暗号化プロトコルとして SSL バージョン 3 を選択します。

このプロトコルは、OpenSSL が `OPENSSL_NO_SSLv3` フラグが有効な状態でコンパイルされている場合には利用できません。

警告: SSL version 3 は非セキュアです。このプロトコルは強く非推奨です。

バージョン 2.7.13 で非推奨: OpenSSL has deprecated all version specific protocols. Use the default protocol with flags like `OP_NO_SSLv3` instead.

ssl.PROTOCOL_TLSv1

チャンネル暗号化プロトコルとして TLS バージョン 1.0 を選択します。

バージョン 2.7.13 で非推奨: OpenSSL has deprecated all version specific protocols. Use the default protocol with flags like `OP_NO_SSLv3` instead.

`ssl.PROTOCOL_TLSv1.1`

チャンネル暗号化プロトコルとして TLS バージョン 1.1 を選択します。openssl version 1.0.1+ のみで利用可能です。

バージョン 2.7.9 で追加.

バージョン 2.7.13 で非推奨: OpenSSL has deprecated all version specific protocols. Use the default protocol with flags like `OP_NO_SSLv3` instead.

`ssl.PROTOCOL_TLSv1.2`

チャンネル暗号化プロトコルとして TLS バージョン 1.2 を選択します。これは最も現代的で、接続の両サイドが利用できる場合は、たぶん最も安全な選択肢です。openssl version 1.0.1+ のみで利用可能です。

バージョン 2.7.9 で追加.

バージョン 2.7.13 で非推奨: OpenSSL has deprecated all version specific protocols. Use the default protocol with flags like `OP_NO_SSLv3` instead.

`ssl.OP_ALL`

相手にする SSL 実装のさまざまなバグを回避するためのワークアラウンドを有効にします。このオプションはデフォルトで有効です。これを有効にする場合 OpenSSL 用の同じ意味のフラグ `SSL_OP_ALL` をセットする必要はありません。

バージョン 2.7.9 で追加.

`ssl.OP_NO_SSLv2`

SSLv2 接続が行われないようにします。このオプションは `PROTOCOL_SSLv23` と組み合わせでのみ意味を持ちます。ピア間で SSLv2 がプロトコルバージョンとして選択されることを防ぎます。

バージョン 2.7.9 で追加.

`ssl.OP_NO_SSLv3`

SSLv3 接続が行われないようにします。このオプションは `PROTOCOL_SSLv23` と組み合わせでのみ意味を持ちます。ピア間で SSLv3 がプロトコルバージョンとして選択されることを防ぎます。

バージョン 2.7.9 で追加.

`ssl.OP_NO_TLSv1`

TLSv1 接続が行われないようにします。このオプションは `PROTOCOL_SSLv23` と組み合わせでのみ意味を持ちます。ピア間で TLSv1 がプロトコルバージョンとして選択されることを防ぎます。

バージョン 2.7.9 で追加.

`ssl.OP_NO_TLSv1.1`

TLSv1.1 接続が行われないようにします。このオプションは `PROTOCOL_SSLv23` と組み合わせでのみ意味を持ちます。ピア間で TLSv1.1 がプロトコルバージョンとして選択されることを防ぎます。openssl version 1.0.1+ でのみ利用できます。

バージョン 2.7.9 で追加.

`ssl.OP_NO_TLSv1_2`

TLSv1.2 接続が行われないようにします。このオプションは `PROTOCOL_SSLv23` と組み合わせでのみ意味を持ちます。ピア間で TLSv1.2 がプロトコルバージョンとして選択されることを防ぎます。openssl version 1.0.1+ でのみ利用できます。

バージョン 2.7.9 で追加.

`ssl.OP_NO_TLSv1_3`

Prevents a TLSv1.3 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing TLSv1.3 as the protocol version. TLS 1.3 is available with OpenSSL 1.1.1 or later. When Python has been compiled against an older version of OpenSSL, the flag defaults to 0.

バージョン 2.7.15 で追加.

`ssl.OP_CIPHER_SERVER_PREFERENCE`

暗号の優先順位として、クライアントのものではなくサーバのものを使います。このオプションはクライアントソケットと SSLv2 のサーバソケットでは効果はありません。

バージョン 2.7.9 で追加.

`ssl.OP_SINGLE_DH_USE`

SSL セッションを区別するのに同じ DH 鍵を再利用しないようにします。これはセキュリティを向上させますが、より多くの計算機リソースを必要とします。このオプションはサーバソケットに適用されます。

バージョン 2.7.9 で追加.

`ssl.OP_SINGLE_ECDH_USE`

SSL セッションを区別するのに同じ ECDH 鍵を再利用しないようにします。これはセキュリティを向上させますが、より多くの計算機リソースを必要とします。このオプションはサーバソケットに適用されます。

バージョン 2.7.9 で追加.

`ssl.OP_ENABLE_MIDDLEBOX_COMPAT`

Send dummy Change Cipher Spec (CCS) messages in TLS 1.3 handshake to make a TLS 1.3 connection look more like a TLS 1.2 connection.

This option is only available with OpenSSL 1.1.1 and later.

バージョン 2.7.16 で追加.

`ssl.OP_NO_COMPRESSION`

SSL チャンネルでの圧縮を無効にします。これはアプリケーションのプロトコルが自身の圧縮方法をサポートする場合に有用です。

このオプションは OpenSSL 1.0.0 以降のみで使用できます。

バージョン 2.7.9 で追加.

`ssl.HAS_ALPN`

OpenSSL ライブラリが、組み込みで [RFC 7301](#) で記述されている *Application-Layer Protocol Negotiation* TLS 拡張をサポートしているかどうか。

バージョン 2.7.10 で追加.

`ssl.HAS_ECDH`

OpenSSL ライブラリが、組み込みの楕円曲線ディフィー・ヘルマン鍵共有をサポートしているかどうか。これは、ディストリビュータが明示的に無効にしていない限りは、真であるはずです。

バージョン 2.7.9 で追加.

`ssl.HAS_SNI`

OpenSSL ライブラリが、組み込みで ([RFC 4366](#) で記述されている) *Server Name Indication* 拡張をサポートしているかどうか。

バージョン 2.7.9 で追加.

`ssl.HAS_NPN`

OpenSSL ライブラリが、組み込みで、[NPN draft specification](#) で記述されている *Next Protocol Negotiation* をサポートしているかどうか。true であれば、サポートしたいプロトコルを `SSLContext.set_npn_protocols()` メソッドで提示することができます。

バージョン 2.7.9 で追加.

`ssl.HAS_TLSv1.3`

Whether the OpenSSL library has built-in support for the TLS 1.3 protocol.

バージョン 2.7.15 で追加.

`ssl.CHANNEL_BINDING_TYPES`

サポートされている TLS のチャンネルバインディングのタイプのリスト。リスト内の文字列は `SSLSocket.get_channel_binding()` の引数に渡せます。

バージョン 2.7.9 で追加.

`ssl.OPENSSL_VERSION`

インタプリタによってロードされた OpenSSL ライブラリのバージョン文字列:

```
>>> ssl.OPENSSL_VERSION
'OpenSSL 0.9.8k 25 Mar 2009'
```

バージョン 2.7 で追加.

`ssl.OPENSSL_VERSION_INFO`

OpenSSL ライブラリのバージョン情報を表す 5 つの整数のタプル:

```
>>> ssl.OPENSSL_VERSION_INFO
(0, 9, 8, 11, 15)
```

バージョン 2.7 で追加.

`ssl.OPENSSL_VERSION_NUMBER`

1 つの整数の形式の、OpenSSL ライブラリの生のバージョン番号:

```
>>> ssl.OPENSSL_VERSION_NUMBER
9470143L
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x9080bfL'
```

バージョン 2.7 で追加.

`ssl.ALERT_DESCRIPTION_HANDSHAKE_FAILURE``ssl.ALERT_DESCRIPTION_INTERNAL_ERROR``ALERT_DESCRIPTION_*`

RFC 5246 その他からのアラートの種類です。IANA TLS Alert Registry にはこのリストとその意味が定義された RFC へのリファレンスが含まれています。

`SSLContext.set_servername_callback()` でのコールバック関数の戻り値として使われます。

バージョン 2.7.9 で追加.

`Purpose.SERVER_AUTH`

`create_default_context()` と `SSLContext.load_default_certs()` に渡すオプションです。この値はコンテキストが Web サーバの認証に使われることを示します (ですので、クライアントサイドのソケットを作るのに使うことになるでしょう)。

バージョン 2.7.9 で追加.

`Purpose.CLIENT_AUTH`

`create_default_context()` と `SSLContext.load_default_certs()` に渡すオプションです。この値はコンテキストが Web クライアントの認証に使われることを示します (ですので、サーバサイドのソケットを作るのに使うことになるでしょう)。

バージョン 2.7.9 で追加.

17.3.2 SSL ソケット

SSL ソケットは `socket` オブジェクト の以下のメソッドを提供します:

- `accept()`
- `bind()`
- `close()`
- `connect()`
- `fileno()`
- `getpeername()`, `getsockname()`
- `getsockopt()`, `setsockopt()`

- `gettimeout()`, `settimeout()`, `setblocking()`
- `listen()`
- `makefile()`
- `recv()`, `recv_into()` (非ゼロの flags は渡せません)
- `send()`, `sendall()` (非ゼロの flags は渡せません)
- `shutdown()`

SSL(および TLS) プロトコルは TCP の上に独自の枠組みを持っているので、SSL ソケットの抽象化は、いくつかの点で通常の OS レベルのソケットの仕様から逸脱することがあります。特に [ノンブロッキングソケットについての注釈](#) を参照してください。

SSL ソケットには、以下に示す追加のメソッドと属性もあります:

`SSLSocket.do_handshake()`

SSL セットアップのハンドシェイクを実行します。

バージョン 2.7.9 で変更: ソケットの `context` の属性 `check_hostname` が真の場合に、ハンドシェイクメソッドが `match_hostname()` を実行するようになりました。

`SSLSocket.getpeercert(binary_form=False)`

接続先に証明書が無い場合、`None` を返します。SSL ハンドシェイクがまだ行われていない場合は、`ValueError` が送出されます。

`binary_form` が `False` で接続先から証明書を取得した場合、このメソッドは `dict` のインスタンスを返します。証明書が認証されていない場合、辞書は空です。証明書が認証されていた場合いくつかのキーを持った辞書を返し、`subject` (証明書が発行された principal), `issuer` (証明書を発行した principal) を含みます。証明書が *Subject Alternative Name* 拡張 ([RFC 3280](#) を参照) のインスタンスを格納していた場合、`subjectAltName` キーも辞書に含まれます。

`subject`, `issuer` フィールドは、証明書のそれぞれのフィールドについてのデータ構造で与えられる RDN (relative distinguished name) のシーケンスを格納したタプルで、各 RDN は name-value ペアのシーケンスです。現実世界での例をお見せします:

```
{'issuer': (((('countryName', 'IL'),),
              (('organizationName', 'StartCom Ltd.'),),
              (('organizationalUnitName',
               'Secure Digital Certificate Signing'),),
              (('commonName',
               'StartCom Class 2 Primary Intermediate Server CA'),)),
 'notAfter': 'Nov 22 08:15:19 2013 GMT',
 'notBefore': 'Nov 21 03:09:52 2011 GMT',
 'serialNumber': '95F0',
 'subject': (((('description', '571208-SLe257oHY9fVQ07Z'),),
               (('countryName', 'US'),),
               (('stateOrProvinceName', 'California'),),
               (('localityName', 'San Francisco'),),
               (('organizationName', 'Electronic Frontier Foundation, Inc.'),),
```

(次のページに続く)

(前のページからの続き)

```

        (('commonName', '*.eff.org'),),
        (('emailAddress', 'hostmaster@eff.org'),)),
    'subjectAltName': (('DNS', '*.eff.org'), ('DNS', 'eff.org')),
    'version': 3}

```

注釈: 特定のサービスのために証明書の検証がしたければ、`match_hostname()` 関数を使うことが出来ます。

`binary_form` 引数が `True` だった場合、証明書が渡されていればこのメソッドは DER エンコードされた証明書全体をバイト列として返し、接続先が証明書を提示しなかった場合は `None` を返します。接続先が証明書を提供するかどうかは SSL ソケットの役割に依存します:

- クライアント側ソケットでは、認証が要求されているかどうかに関わらず、サーバは常に証明書を提供します;
- サーバ側ソケットでは、クライアントはサーバによって認証が要求されている場合にのみ証明書を提供します; ですので、(`CERT_OPTIONAL` や `CERT_REQUIRED` ではなく) `CERT_NONE` を使うと `getpeercert()` は `None` を返します。

バージョン 2.7.9 で変更: 返される辞書に `issuer`, `notBefore` のような追加アイテムを含むようになりました。加えて、ハンドシェイクが済んでいなければ `ValueError` を投げるようになりました。返される辞書に `crlDistributionPoints`, `caIssuers`, `OCSP URI` のような X509v3 拡張アイテムを含むようになりました。

`SSLSocket.cipher()`

利用されている暗号の名前、その暗号の利用を定義している SSL プロトコルのバージョン、利用されている鍵の bit 長の 3 つの値を含むタプルを返します。もし接続が確立されていない場合、`None` を返します。

`SSLSocket.compression()`

使われている圧縮アルゴリズムを文字列で返します。接続が圧縮されていなければ `None` を返します。

上位レベルのプロトコルが自身で圧縮メカニズムをサポートする場合、SSL レベルでの圧縮を `OP_NO_COMPRESSION` を使って無効に出来ます。

バージョン 2.7.9 で追加.

`SSLSocket.get_channel_binding(cb_type="tls-unique")`

現在の接続におけるチャンネルバインディングのデータを取得します。未接続あるいはハンドシェイクが完了していなければ `None` を返します。

`cb_type` パラメータにより、望みのチャンネルバインディングのタイプを選択出来ます。チャンネルバインディングのタイプの妥当なものは `CHANNEL_BINDING_TYPES` でリストされています。現在のところでは **RFC 5929** で定義されている 'tls-unique' のみがサポートされています。未サポートのチャンネルバインディングのタイプが要求された場合、`ValueError` を送出します。

バージョン 2.7.9 で追加.

`SSLSocket.selected_alpn_protocol()`

TLS ハンドシェイクで選択されたプロトコルを返します。 `SSLContext.set_alpn_protocols()` が呼ばれていない場合、相手側が ALPN をサポートしていない場合、クライアントが提案したプロトコルのどれもソケットがサポートしない場合、あるいはハンドシェイクがまだ行われていない場合には、`None` が返されます。

バージョン 2.7.10 で追加.

`SSLSocket.selected_npn_protocol()`

TLS/SSL ハンドシェイクで選択された上位レベルのプロトコルを返します。 `SSLContext.set_npn_protocols()` が呼ばれていない場合、相手側が NPN をサポートしていない場合、あるいはハンドシェイクがまだ行われていない場合には、`None` が返されます。

バージョン 2.7.9 で追加.

`SSLSocket.unwrap()`

SSL シャットダウンハンドシェイクを実行します。これは下位レイヤーのソケットから TLS レイヤーを取り除き、下位レイヤーのソケットオブジェクトを返します。これは暗号化されたオペレーションから暗号化されていない接続に移行するときに利用されます。以降の通信には、オリジナルのソケットではなくこのメソッドが返したソケットのみを利用するべきです。

`SSLSocket.version()`

コネクションによって実際にネゴシエイトされた SSL プロトコルバージョンを文字列で、または、セキュアなコネクションが確立していなければ `None` を返します。これを書いている時点では、`"SSLv2"`, `"SSLv3"`, `"TLSv1"`, `"TLSv1.1"`, `"TLSv1.2"` などが返ります。最新の OpenSSL はもっと色々な値を定義しているかもしれません。

バージョン 2.7.9 で追加.

`SSLSocket.context`

この SSL ソケットに結び付けられた `SSLContext` オブジェクトです。SSL ソケットが (`SSLContext.wrap_socket()` ではなく) トップレベルの `wrap_socket()` 関数を使って作られた場合、これはこの SSL ソケットのために作られたカスタムコンテキストオブジェクトです。

バージョン 2.7.9 で追加.

17.3.3 SSL コンテキスト

バージョン 2.7.9 で追加.

SSL コンテキストは、SSL 構成オプション、証明書 (群) や秘密鍵 (群) などのような、一回の SSL 接続よりも長生きするさまざまなデータを保持します。これはサーバサイドソケットの SSL セッションのキャッシュも管理し、同じクライアントからの繰り返しの接続時の速度向上に一役買います。

class `ssl.SSLContext(protocol)`

新しい SSL コンテキストを作成します。 `protocol` にはこのモジュールで定義されている `PROTOCOL_*` 定数のうち一つを指定しなければなりません。最大限の互換性のためには、現時点での推奨は `PROTOCOL_SSLv23` です。

参考:

`create_default_context()` は `ssl` モジュールに、目的に合ったセキュリティ設定を選ばせます。

バージョン 2.7.16 で変更: このコンテキストは、安全性の高いデフォルト値で作成されます。デフォルト設定されるオプションは、`OP_NO_COMPRESSION`, `OP_CIPHER_SERVER_PREFERENCE`, `OP_SINGLE_DH_USE`, `OP_SINGLE_ECDH_USE`, `OP_NO_SSLv2` (`PROTOCOL_SSLv2` 以外), `OP_NO_SSLv3` (`PROTOCOL_SSLv3` 以外) です。初期の暗号方式スイートリストには HIGH 暗号のみが含まれており、NULL 暗号および MD5 暗号は含まれません (`PROTOCOL_SSLv2` 以外)。

`SSLContext` オブジェクトは以下のメソッドと属性を持っています:

`SSLContext.cert_store_stats()`

ロードされた X.509 証明書の数、CA 証明書で活性の X.509 証明書の数、証明書失効リストの数、についての統計情報を辞書として取得します。

一つの CA と他の一つの証明書を持ったコンテキストでの例です:

```
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

`SSLContext.load_cert_chain(certfile, keyfile=None, password=None)`

秘密鍵と対応する証明書をロードします。 `certfile` は、証明書と、証明書認証で必要とされる任意の数の CA 証明書を含む、PEM フォーマットの単一ファイルへのパスでなければなりません。 `keyfile` を指定する場合は、秘密鍵が含まれるファイルを指さなければなりません。そうでなければ秘密鍵も `certfile` から取られます。 `certfile` に証明書をどのように格納すれば良いかについての詳しい情報は、[証明書の議論](#)を参照してください。

`password` 引数に、秘密鍵を復号するためのパスワードを返す関数を与えることが出来ます。その関数は秘密鍵が暗号化されていて、なおかつパスワードが必要な場合にのみ呼び出されます。その関数は引数なしで呼び出され、string, bytes, または bytearray を返さなければなりません。戻り値が string の場合は鍵を復号化するのに使う前に UTF-8 でエンコードされます。string の代わりに bytes や bytearray を返した場合は `password` 引数に直接供給されます。秘密鍵が暗号化されていなかったりパスワードを必要としない場合は、指定は無視されます。

`password` が与えられず、そしてパスワードが必要な場合には、OpenSSL 組み込みのパスワード問い合わせメカニズムが、ユーザに対話的にパスワードを問い合わせます。

秘密鍵が証明書に合致しなければ、`SSLError` が送出されます。

`SSLContext.load_default_certs(purpose=Purpose.SERVER_AUTH)`

デフォルトの場所から "認証局" (CA=certification authority) 証明書ファイル一式をロードします。Windows では、CA 証明書はシステム記憶域の CA と ROOT からロードします。それ以外のシステムでは、この関数は `SSLContext.set_default_verify_paths()` を呼び出します。将来的にはこのメソッドは、他の場所からも CA 証明書をロードするかもしれません。

`purpose` フラグでどの種類の CA 証明書をロードするかを指定します。デフォルトの `Purpose.SERVER_AUTH` は TLS web サーバの認証のために活性かつ信頼された証明書をロードします (クライ

アントサイドのソケット)。 `Purpose.CLIENT_AUTH` はクライアント証明書の正当性検証をサーバサイドで行うための CA 証明書をロードします。

`SSLContext.load_verify_locations (cafile=None, capath=None, cadata=None)`

`verify_mode` が `CERT_NONE` でない場合に接続先の証明書ファイルの正当性検証に使われる ”認証局” (CA=certification authority) 証明書ファイル一式をロードします。少なくとも `cafile` か `capath` のどちらかは指定しなければなりません。

このメソッドは PEM または DER フォーマットの証明書失効リスト (CRLs=certification revocation lists) もロード出来ます。CRLs のために使うには、 `SSLContext.verify_flags` を適切に設定しなければなりません。

`cafile` を指定する場合は、PEM フォーマットで CA 証明書が結合されたファイルへのパスを指定してください。このファイル内で証明書をどのように編成すれば良いのかについての詳しい情報については、[証明書](#) の議論を参照してください。

The `capath` string, if present, is the path to a directory containing several CA certificates in PEM format, following an [OpenSSL specific layout](#).

`cadata` オブジェクトを指定する場合は、PEM エンコードの証明書一つ以上の ASCII 文字列か、DER エンコードの証明書のバイトライクなオブジェクトのどちらかを指定してください。PEM エンコードの証明書の周囲の余分な行は無視されますが、少なくとも一つの証明書が含まれている必要があります。

`SSLContext.get_ca_certs (binary_form=False)`

ロードされた ”認証局” (CA=certification authority) 証明書のリストを取得します。 `binary_form` パラメータが `False` であれば、リストのそれぞれのエントリは `SSLSocket.getpeercert()` が出力するような辞書になります。そうでない場合このメソッドは、DER エンコード形式の証明書のリストで返します。返却されるリストには、SSL 接続によって要求されてロードされない限りは `capath` からの証明書は含みません。

注釈: `capath` ディレクトリ内の証明書は一度でも使われない限りはロードされません。

`SSLContext.set_default_verify_paths()`

デフォルトの ”認証局” (CA=certification authority) 証明書を、OpenSSL ライブラリがビルドされた際に定義されたファイルシステム上のパスからロードします。残念ながらこのメソッドが成功したかどうかを知るための簡単な方法はありません: 証明書が見つからなくてもエラーは返りません。OpenSSL ライブラリがオペレーティングシステムの一部として提供されている際にはどうやら適切に構成できるようですが。

`SSLContext.set_ciphers (ciphers)`

Set the available ciphers for sockets created with this context. It should be a string in the [OpenSSL cipher list format](#). If no cipher can be selected (because compile-time options or other configuration forbids use of all the specified ciphers), an `SSLError` will be raised.

注釈: 接続時に SSL ソケットの `SSLSocket.cipher()` メソッドが、現在選択されているその暗号を使います。

OpenSSL 1.1.1 has TLS 1.3 cipher suites enabled by default. The suites cannot be disabled with `set_ciphers()`.

`SSLContext.set_alpn_protocols (protocols)`

SSL/TLS ハンドシェイク時にソケットが提示すべきプロトコルを指定します。['http/1.1', 'spdy/2'] のような推奨順に並べた ASCII 文字列のリストでなければなりません。プロトコルの選択は [RFC 7301](#) に従いハンドシェイクの中で行われます。ハンドシェイクが正常に終了後、`SSLSocket.selected_alpn_protocol()` メソッドは合意されたプロトコルを返します。

このメソッドは `HAS_ALPN` が偽の場合 `NotImplementedError` を送出します。

OpenSSL 1.1.0 to 1.1.0e will abort the handshake and raise `SSL_ERROR` when both sides support ALPN but cannot agree on a protocol. 1.1.0f+ behaves like 1.0.2, `SSLSocket.selected_alpn_protocol()` returns None.

バージョン 2.7.10 で追加.

`SSLContext.set_npn_protocols (protocols)`

SSL/TLS ハンドシェイク時にソケットが提示すべきプロトコルを指定します。['http/1.1', 'spdy/2'] のような推奨順に並べた文字列のリストでなければなりません。プロトコルの選択は [NPN draft specification](#) に従いハンドシェイクの中で行われます。ハンドシェイクが正常に終了後、`SSLSocket.selected_alpn_protocol()` メソッドは合意されたプロトコルを返します。

このメソッドは `HAS_NPN` が偽の場合 `NotImplementedError` を送出します。

`SSLContext.set_servername_callback (server_name_callback)`

TLS クライアントがサーバ名表示を指定した際の、SSL/TLS サーバによって TLS Client Hello ハンドシェイクメッセージが受け取られたあとで呼び出されるコールバック関数を登録します。サーバ名表示メカニズムは [RFC 6066](#) セクション 3 - Server Name Indication で述べられています。

`SSLContext` ごとに一つだけコールバックをセット出来ます。 `server_name_callback` を `None` にすればコールバックは無効になります。この関数を続けて呼ぶと、以前に登録されたコールバックを上書きします。

コールバック関数 `server_name_callback` は 3 つの引数で呼び出されます; 最初の引数は `ssl.SSLSocket` です。2 つ目の引数は、クライアントが相手をしようと意図しているサーバ名を表す文字列 (または TLS Client Hello がサーバ名を含まない場合は `None`) です。そして 3 つ目の引数はオリジナルの `SSLContext` です。サーバ名引数は IDNA デコードされたサーバ名です。

このコールバックの典型的な利用方法は、`ssl.SSLSocket` の `SSLSocket.context` 属性を、サーバ名に合致する証明書チェーンを持つ新しい `SSLContext` オブジェクトに変更することです。

TLS 接続の初期ネゴシエーションのフェーズですから、`SSLSocket.selected_alpn_protocol()`, `SSLSocket.context` のような限られたメソッドと属性のみ使えます。 `SSLSocket.getpeercert()`, `SSLSocket.getpeername()`, `SSLSocket.cipher()`, `SSLSocket.compress()` メソッドは TLS 接続が TLS Client Hello よりも先に進行していることを必要としますから、これらは意味のある値を返しませんし、安全に呼び出すことも出来ません。

TLS ネゴシエーションを継続させるならば、`server_name_callback` 関数は `None` を返さなければなりません。TLS が失敗することを必要とするなら、constant `ALERT_DESCRIPTION_*` を返してください。ここにはない値を返すと、致命エラー `ALERT_DESCRIPTION_INTERNAL_ERROR` を引き起こします。

サーバ名に対する IDNA デコードのエラーがあれば、TLS 接続はクライアントに対する TLS の致命的アラートメッセージ `ALERT_DESCRIPTION_INTERNAL_ERROR` とともに終了します。

`server_name_callback` 関数が例外を送出した場合、TLS 接続は TLS の致命的アラートメッセージ `ALERT_DESCRIPTION_HANDSHAKE_FAILURE` とともに終了します。

このメソッドは OpenSSL ライブラリが `OPENSSL_NO_TLSEXT` を定義してビルドされている場合、`NotImplementedError` を送出します。

`SSLContext.load_dh_params (dhfile)`

ディフィー・ヘルマン (DH) 鍵交換のための鍵生成パラメータをロードします。DH 鍵交換を用いることは、(サーバ、クライアントともに) 計算機リソースに高い処理負荷をかけますがセキュリティを向上させます。`dhfile` パラメータは PEM フォーマットの DH パラメータを含んだファイルへのパスでなければなりません。

この設定はクライアントソケットには適用されません。さらにセキュリティを改善するのに `OP_SINGLE_DH_USE` オプションも利用できます。

`SSLContext.set_ecdh_curve (curve_name)`

楕円曲線ディフィー・ヘルマン (ECDH) 鍵交換の曲線名を指定します。ECDH はもとの DH に較べて、ほぼ間違いなく同程度に安全である一方で、顕著に高速です。`curve_name` パラメータは既知の楕円曲線を表す文字列でなければなりません。例えば `prime256v1` が広くサポートされている曲線です。

この設定はクライアントソケットには適用されません。さらにセキュリティを改善するのに `OP_SINGLE_ECDH_USE` オプションも利用できます。

このメソッドは `HAS_ECDH` が `False` の場合は利用できません。

参考:

[SSL/TLS & Perfect Forward Secrecy](#) Vincent Bernat.

`SSLContext.wrap_socket (sock, server_side=False, do_handshake_on_connect=True, suppress_ragged_eofs=True, server_hostname=None)`

既存の Python ソケット `sock` をラップして `ssl.SSLSocket` オブジェクトを返します。`sock` は `SOCK_STREAM` ソケットでなければなりません; ほかのタイプのソケットはサポートされていません。

返される SSL ソケットは、コンテキスト、その設定と証明書に関連付けられます。パラメータ `server_side`, `do_handshake_on_connect`, `suppress_ragged_eofs` はトップレベルの関数 `wrap_socket ()` のものと同じ意味です。

クライアントサイドから接続では、`server_hostname` で接続しようとしているサービスのホスト名を指定出来ます。これは HTTP バーチャルホストにかなり似て、シングルサーバで複数の SSL ベースのサービスを別々の証明書でホストしているようなサーバに対して使えます。`server_side` が真の場合に `server_hostname` を指定すると `ValueError` を送出します。

バージョン 2.7.9 で変更: OpenSSL が SNI をサポートしなくても `server_hostname` を許容するようになりました。

`SSLContext.session_stats()`

Get statistics about the SSL sessions created or managed by this context. A dictionary is returned which maps the names of each [piece of information](#) to their numeric values. For example, here is the total number of hits and misses in the session cache since the context was created:

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

`SSLContext.check_hostname`

`SSLSocket.do_handshake()` 呼び出し時に、`match_hostname()` を使って接続先証明書のホスト名の合致を見るかどうか。コンテキストの `verify_mode` には `CERT_OPTIONAL` か `CERT_REQUIRED` をセットしなければなりません。また `wrap_socket()` にはホスト名の合致をみるための `server_hostname` を渡さなければなりません。

例:

```
import socket, ssl

context = ssl.SSLContext(ssl.PROTOCOL_TLS)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = context.wrap_socket(s, server_hostname='www.verisign.com')
ssl_sock.connect(('www.verisign.com', 443))
```

注釈: この機能には OpenSSL 0.9.8f 以降が必要です。

`SSLContext.options`

このコンテキストで有効になっている SSL オプションを表す整数。デフォルトの値は `OP_ALL` ですが、`OP_NO_SSLv2` のようなほかの値をビット OR 演算で指定出来ます。

注釈: OpenSSL の 0.9.8m より古いバージョンを使う場合、値はセットは出来ませんがクリアが出来ません。オプションを (対応するビットをリセットすることで) クリアしようすると `ValueError` が送出されます。

`SSLContext.protocol`

コンテキストの構築時に選択されたプロトコルバージョン。この属性は読み取り専用です。

`SSLContext.verify_flags`

証明書の検証操作のためのフラグです。 `VERIFY_CRL_CHECK_LEAF` などのフラグをビット OR 演算

でセット出来ます。デフォルトでは OpenSSL は証明書失効リスト (CRLs) を必要としませんし検証にも使いません。openssl version 0.9.8+ でのみ利用可能です。

`SSLContext.verify_mode`

接続先の証明書の検証を試みるかどうか、また、検証が失敗した場合にどのように振舞うべきかを制御します。この属性は `CERT_NONE`、`CERT_OPTIONAL`、`CERT_REQUIRED` のうちどれか一つでなければなりません。

17.3.4 証明書

証明書を大まかに説明すると、公開鍵/秘密鍵システム的一种です。このシステムでは、各 *principal* (これはマシン、人、組織などです) は、ユニークな 2 つの暗号鍵を割り当てられます。1 つは公開され、公開鍵 (*public key*) と呼ばれます。もう一方は秘密にされ、秘密鍵 (*private key*) と呼ばれます。2 つの鍵は関連しており、片方の鍵で暗号化したメッセージは、もう片方の鍵のみで復号できます。

A certificate contains information about two principals. It contains the name of a *subject*, and the subject's public key. It also contains a statement by a second principal, the *issuer*, that the subject is who they claim to be, and that this is indeed the subject's public key. The issuer's statement is signed with the issuer's private key, which only the issuer knows. However, anyone can verify the issuer's statement by finding the issuer's public key, decrypting the statement with it, and comparing it to the other information in the certificate. The certificate also contains information about the time period over which it is valid. This is expressed as two fields, called "notBefore" and "notAfter".

Python において証明書を利用する場合、クライアントもサーバーも自分を証明するために証明書を利用することができます。ネットワーク接続の相手側に証明書の提示を要求する事ができ、そのクライアントやサーバーが認証を必要とするならその証明書を認証することができます。認証が失敗した場合、接続は例外を発生させます。認証は下位層の OpenSSL フレームワークが自動的に行います。アプリケーションは認証機構について意識する必要はありません。しかし、アプリケーションは認証プロセスのために幾つかの証明書を提供する必要がありますがあるかもしれません。

Python は証明書を格納したファイルを利用します。そのファイルは "PEM" ([RFC 1422](#) 参照) フォーマットという、ヘッダー行とフッター行の間に base-64 エンコードされた形をとっている必要があります。

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

証明書チェーン

Python が利用する証明書を格納したファイルは、ときには 証明書チェーン (*certificate chain*) と呼ばれる証明書のシーケンスを格納します。このチェーンは、まずクライアントやサーバー自体の *principal* の証明書で始まらなければなりません。それ以降に続く証明書は、手前の証明書の発行者 (*issuer*) の証明書になり、最後に *subject* と発行者が同じ 自己署名 (*self-signed*) 証明書で終わります。この最後の証明書は ルート証明書 (*root certificate*) と呼ばれます。これらの証明書チェーンは 1 つの証明書ファイルに結合されなければなりません。例えば、3 つの証明書からなる証明書チェーンがあるとします。私たちのサーバーの証明書から、私たちの

サーバーに署名した認証局の証明書、そして認証局の証明書を発行した機関のルート証明書です。

```
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the root certificate for the CA's issuer)...
-----END CERTIFICATE-----
```

CA 証明書

もし相手から送られてきた証明書の認証をしたい場合、信頼している各発行者の証明書チェーンが入った "CA certs" ファイルを提供する必要があります。繰り返しますが、このファイルは単純に、各チェーンを結合しただけのものです。認証のために、Python はそのファイルの中の最初にマッチしたチェーンを利用します。 `SSLContext.load_default_certs()` を呼び出すことでプラットフォームの証明書ファイルも使われますが、これは `create_default_context()` によって自動的行われます。

秘密鍵と証明書を一緒にする

多くの場合、証明書と同じファイルに秘密鍵も格納されています。この場合、 `SSLContext.load_cert_chain()`、`wrap_socket()` には `certfile` 引数だけが必要とされます。秘密鍵が証明書ファイルに格納されている場合、秘密鍵は証明書チェーンの最初の証明書よりも先にないといけません。

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

自己署名証明書

SSL 暗号化接続サービスを提供するサーバーを建てる場合、適切な証明書を取得するには、認証局から買うなどの幾つかの方法があります。また、自己署名証明書を作るケースもあります。OpenSSL を使って自己署名証明書を作るには、次のようにします。

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
Generating a 1024 bit RSA private key
.....+++++
.....+++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
```

(次のページに続く)

(前のページからの続き)

```
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization.com
Email Address []:ops@myserver.mygroup.myorganization.com
%
```

自己署名証明書の欠点は、それ自身がルート証明書であり、他の人はその証明書を持っていない(そして信頼しない)ことです。

17.3.5 例

SSL サポートをテストする

インストールされている Python が SSL をサポートしているかどうかをテストするために、ユーザーコードは次のイディオムを利用することができます。

```
try:
    import ssl
except ImportError:
    pass
else:
    ... # do something that requires SSL support
```

クライアントサイドの処理

この例では、自動的に証明書の検証を行うことを含む望ましいセキュリティ設定でクライアントソケットの SSL コンテキストを作ります:

```
>>> context = ssl.create_default_context()
```

あなた自身でセキュリティ設定を調整したいと望むなら、スクラッチからコンテキストを作ることは出来ます(ですが正しくない設定をしてしまいがちなことに警戒してください)

```
>>> context = ssl.SSLContext(ssl.PROTOCOL_TLS)
>>> context.verify_mode = ssl.CERT_REQUIRED
>>> context.check_hostname = True
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(このスニペットは全ての CA 証明書が /etc/ssl/certs/ca-bundle.crt にバンドルされていることを

仮定しています; もし違っていればエラーになりますので、適宜修正してください)

サーバへの接続にこのコンテキストを使うと、`CERT_REQUIRED` でサーバの証明書の検証が行われます: サーバの証明書が CA 証明書のいずれかに署名されていて、その署名が正しいことを保障します。

```
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
...                             server_hostname="www.python.org")
>>> conn.connect(("www.python.org", 443))
```

そして証明書を持てくることが出来ます:

```
>>> cert = conn.getpeercert()
```

証明書が、期待しているサービス (つまり、HTTPS ホスト `www.python.org`) の身元を特定していることを視覚的に点検してみましょう:

```
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2ExtendedValidationServerCA.
→crt',),
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ev-server-g1.crl',
                           'http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'issuer': (((('countryName', 'US'),),
               (('organizationName', 'DigiCert Inc'),),
               (('organizationalUnitName', 'www.digicert.com'),),
               (('commonName', 'DigiCert SHA2 Extended Validation Server CA'),)),
 'notAfter': 'Sep  9 12:00:00 2016 GMT',
 'notBefore': 'Sep  5 00:00:00 2014 GMT',
 'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
 'subject': (((('businessCategory', 'Private Organization'),),
                (('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
                (('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
                (('serialNumber', '3359300'),),
                (('streetAddress', '16 Allen Rd'),),
                (('postalCode', '03894-4801'),),
                (('countryName', 'US'),),
                (('stateOrProvinceName', 'NH'),),
                (('localityName', 'Wolfeboro,'),),
                (('organizationName', 'Python Software Foundation'),),
                (('commonName', 'www.python.org'),)),
 'subjectAltName': (('DNS', 'www.python.org'),
                    ('DNS', 'python.org'),
                    ('DNS', 'pypi.org'),
                    ('DNS', 'docs.python.org'),
                    ('DNS', 'testpypi.python.org'),
                    ('DNS', 'bugs.python.org'),
                    ('DNS', 'wiki.python.org'),
                    ('DNS', 'hg.python.org'),
                    ('DNS', 'mail.python.org'),
                    ('DNS', 'packaging.python.org'),
                    ('DNS', 'pythonhosted.org'),
                    ('DNS', 'www.pythonhosted.org'),
```

(次のページに続く)

(前のページからの続き)

```

        ('DNS', 'test.pythonhosted.org'),
        ('DNS', 'us.pycon.org'),
        ('DNS', 'id.python.org')),
    'version': 3}

```

SSL チャンネルは今や確立されて証明書が検証されているので、サーバとお喋りを続けることができます:

```

>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
 b'X-Cache: HIT',
 b'X-Cache-Hits: 11',
 b'Vary: Cookie',
 b'Strict-Transport-Security: max-age=63072000; includeSubDomains',
 b'Connection: close',
 b'',
 b'']

```

このドキュメントの下の方の、[セキュリティで考慮すべき点](#) に関する議論を参照してください。

サーバーサイドの処理

サーバーサイドの処理では、通常、サーバー証明書と秘密鍵がそれぞれファイルに格納された形で必要です。最初に秘密鍵と証明書が保持されたコンテキストを作成し、クライアントがあなたの信憑性をチェック出来るようにします。そののちにソケットを開き、ポートにバインドし、そのソケットの `listen()` を呼び、クライアントからの接続を待ちます。

```

import socket, ssl

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="mycertfile", keyfile="mykeyfile")

bindsocket = socket.socket()
bindsocket.bind(('myaddr.mydomain.com', 10023))
bindsocket.listen(5)

```

クライアントが接続してきた場合、`accept()` を呼んで新しいソケットを作成し、接続のためにサーバサイドの SSL ソケットを、コンテキストの `SSLContext.wrap_socket()` メソッドで作ります:

```

while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()

```

そして、`connstream` からデータを読み、クライアントと切断する (あるいはクライアントが切断してくる) まで何か処理をします。

```

def deal_with_client(connstream):
    data = connstream.read()
    # null data means the client is finished with us
    while data:
        if not do_something(connstream, data):
            # we'll assume do_something returns False
            # when we're finished with client
            break
        data = connstream.read()
    # finished with client

```

そして新しいクライアント接続のために `listen` に戻ります。(もちろん現実のサーバは、おそらく個々のクライアント接続ごとに別のスレッドで処理するか、ソケットをノンブロッキングモードにし、イベントループを使うでしょう。)

17.3.6 ノンブロッキングソケットについての注意事項

ノンブロッキングソケットとともに使う場合、いくつか気をつけなければならない事項があります:

- `select()` 呼び出しは OS レベルでのソケットが読み出し可能 (または書き込み可能) になったことを教えてくれますが、上位の SSL レイヤーでの十分なデータがあることを意味するわけではありません。例えば、SSL フレームの一部が届いただけかもしれません。ですから、`SSLSocket.recv()` と `SSLSocket.send()` の失敗を処理することに備え、ほかの `select()` 呼び出し後にリトライしなければなりません。
- 反対に、SSL レイヤーは自身で独自の枠組みを持っているために、読み出せるけれども `select()` が気付くことのないデータを SSL ソケットが持っていることがあります。ですので、潜在的に入手可能なデータを飲み干すために最初に `SSLSocket.recv()` を呼び出すべきであり、そののちでそれでもまだ必要な場合にだけ `select()` でブロックすべきです。

(当然のことながら、ほかのプリミティブ、例えば `poll()` や `selectors` モジュール内のものを使う際にも似た但し書きが付きます)

- SSL ハンドシェイクそのものがノンブロッキングになります: `SSLSocket.do_handshake()` メソッドは成功するまでリトライしなければなりません。 `select()` を用いてソケットの準備が整うのを待つためには、およそ以下のようにします:

```
while True:
    try:
        sock.do_handshake()
        break
    except ssl.SSLWantReadError:
        select.select([sock], [], [])
    except ssl.SSLWantWriteError:
        select.select([], [sock], [])
```

17.3.7 セキュリティで考慮すべき点

最善のデフォルト値

クライアントでの使用 では、あなたのセキュリティポリシーによる特殊な要件がない限りは、SSL コンテキストを作成するためには `create_default_context()` 関数を使用することを強くお勧めします。それはシステムが信頼した CA 証明書をロードし、証明書の検証を有効化し、ホスト名のチェックを行い、そしてセキュアなプロトコルとセキュアな暗号の分別ある設定を試みます。

接続にクライアントの証明書が必要な場合、`SSLContext.load_cert_chain()` によって追加出来ます。

対照的に、`SSLContext` クラスのコンストラクタを自身で呼び出すことによって SSL コンテキストを作ると、デフォルトでは証明書検証もホスト名チェックも行わないものになります。もしそうするのであれば、良いセキュリティレベルを知るために、どうか下の方にあるパラグラフをお読みください。

手動での設定

証明書の検証

`SSLContext` のコンストラクタを直接呼び出した場合、`CERT_NONE` がデフォルトとして使われます。これは接続先の身元特定をしないので安全ではありませんし、特にクライアントモードでは大抵相手となるサーバの信憑性を保障したいでしょう。ですから、クライアントモードでは `CERT_REQUIRED` を強くお勧めします。ですが、それだけでは不十分です; `SSLSocket.getpeercert()` を呼び出してサーバ証明書が望んだサービスと合致するかのチェックもしなければなりません。多くのプロトコルとアプリケーションにとって、サービスはホスト名で特定されます; この場合、`match_hostname()` が使えます。これらの共通的なチェックは `SSLContext.check_hostname` が有効な場合、自動的に行われます。

サーバモードにおいて、(より上位のレベルでの認証メカニズムではなく) SSL レイヤーを使ってあなたのクライアントを認証したいならば、`CERT_REQUIRED` を指定して同じようにクライアントの証明書を検証すべきでしょう。

注釈: クライアントモードでは anonymous ciphers が有効 (デフォルトでは無効) でない限り、`CERT_OPTIONAL` と `CERT_REQUIRED` は同じ意味になります。

プロトコルのバージョン

SSL versions 2 と 3 は非セキュアと考えられており、それゆえその使用は危険です。クライアントとサーバの最大限の互換性が欲しいならば、`SSLContext.options` 属性で明示的に SSLv2, SSLv3 を無効にして `PROTOCOL_SSLv23` を使ってください:

```
context = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
context.options |= ssl.OP_NO_SSLv2
context.options |= ssl.OP_NO_SSLv3
```

上で作った SSL コンテキストは TLSv1 かそれ以降 (あなたのシステムでサポートされていれば) だけの接続を許可します。

暗号の選択

If you have advanced security requirements, fine-tuning of the ciphers enabled when negotiating a SSL session is possible through the `SSLContext.set_ciphers()` method. Starting from Python 2.7.9, the ssl module disables certain weak ciphers by default, but you may want to further restrict the cipher choice. Be sure to read OpenSSL's documentation about the [cipher list format](#). If you want to check which ciphers are enabled by a given cipher list, use the `openssl ciphers` command on your system.

マルチプロセス化

(例えば `multiprocessing` や `concurrent.futures` を使って、) マルチプロセスアプリケーションの一部としてこのモジュールを使う場合、OpenSSL の内部の乱数発生器は fork したプロセスを適切に処理しないことに気を付けて下さい。SSL の機能を `os.fork()` とともに使う場合、アプリケーションは親プロセスの PRNG 状態を変更しなければなりません。 `RAND_add()`, `RAND_bytes()`, `RAND_pseudo_bytes()` のいずれかの呼び出し成功があれば十分です。

17.3.8 LibreSSL support

LibreSSL is a fork of OpenSSL 1.0.1. The ssl module has limited support for LibreSSL. Some features are not available when the ssl module is compiled with LibreSSL.

- LibreSSL >= 2.6.1 no longer supports NPN. The methods `SSLContext.set_npn_protocols()` and `SSLSocket.selected_npn_protocol()` are not available.
- `SSLContext.set_default_verify_paths()` ignores the env vars `SSL_CERT_FILE` and `SSL_CERT_PATH` although `get_default_verify_paths()` still reports them.

参考:

`socket.socket` クラス 下位レイヤーの `socket` クラスのドキュメント

[SSL/TLS Strong Encryption: An Introduction](#) Apache WEB サーバのドキュメンテーションのイントロ

RFC 1422: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management
Steve Kent

RFC 1750: Randomness Recommendations for Security D. Eastlake et. al.

RFC 3280: Internet X.509 Public Key Infrastructure Certificate and CRL Profile Housley et. al.

RFC 4366: Transport Layer Security (TLS) Extensions Blake-Wilson et. al.

RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2 T. Dierks et. al.

RFC 6066: Transport Layer Security (TLS) Extensions D. Eastlake

IANA TLS: Transport Layer Security (TLS) Parameters IANA

RFC 7525: Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security
IETF

Mozilla's Server Side TLS recommendations Mozilla

17.4 `signal` — 非同期イベントにハンドラを設定する

このモジュールでは Python でシグナルハンドラを使うための機構を提供します。シグナルとハンドラを扱う上で一般的なルールがいくつかあります:

- 特定のシグナルに対するハンドラが一度設定されると、明示的にリセットしないかぎり設定されたままになります (Python は背後の実装系に関係なく BSD 形式のインタフェースをエミュレートします)。例外は `SIGCHLD` のハンドラで、この場合は背後の実装系の仕様に従います。
- クリティカルセクションから一時的にシグナルを "ブロック" することはできません (この機能をサポートしない Unix 系システムも存在するためです)。
- Python のシグナルハンドラは Python のユーザが望む限り非同期で呼び出されますが、呼び出されるのは Python インタプリタの "原子的な (atomic)" 命令実行単位の間です。したがって、(巨大なサイズのテキストに対する正規表現の一致検索のような) 純粋に C 言語のレベルで実現されている時間のかかる処理中に到着したシグナルは、不定期間遅延する可能性があります。
- シグナルが I/O 操作中に到着すると、シグナルハンドラが処理を返した後に I/O 操作が例外を送出する可能性があります。これは背後にある Unix システムが割り込みシステムコールにどういう意味付けをしているかに依存します。
- C 言語のシグナルハンドラは常に処理を返すので、`SIGFPE` や `SIGSEGV` のような同期エラーの捕捉はほとんど意味がありません。
- Python は標準でごく少数のシグナルハンドラをインストールしています: `SIGPIPE` は無視されます (したがって、パイプやソケットに対する書き込みで生じたエラーは通常の Python 例外として報告されます) `SIGINT` は `KeyboardInterrupt` 例外に変換されます。これらはどれも上書きすることができます。

- シグナルとスレッドの両方を同じプログラムで使用する場合にはいくつか注意が必要です。シグナルとスレッドを同時に利用する上で基本的に注意すべきことは、`signal()` 命令は常に主スレッド (main thread) の処理中で実行するということです。どのスレッドも `alarm()`、`getsignal()`、`pause()`、`setitimer()`、`getitimer()` を実行することができます。しかし、主スレッドだけが新たなシグナルハンドラを設定することができ、したがってシグナルを受け取ることができるのは主スレッドだけです (これは、背後のスレッド実装が個々のスレッドに対するシグナル送信をサポートしているかに関わらず、Python `signal` モジュールが強制している仕様です)。したがって、シグナルをスレッド間通信の手段として使うことはできません。代わりにロック機構を使ってください。

以下に `signal` モジュールで定義されている変数を示します:

`signal.SIG_DFL`

二つある標準シグナル処理オプションのうちの一つです; 単純にシグナルに対する標準の関数を実行します。例えば、ほとんどのシステムでは、`SIGQUIT` に対する標準の動作はコアダンプと終了で、`SIGCHLD` に対する標準の動作は単にシグナルの無視です。

`signal.SIG_IGN`

もう一つの標準シグナル処理オプションで、受け取ったシグナルを単に無視します。

SIG*

全てのシグナル番号はシンボル定義されています。例えば、ハングアップシグナルは `signal.SIGHUP` で定義されています; 変数名は C 言語のプログラムで使われているのと同じ名前で、`<signal.h>` にあります。'`signal()`' に関する Unix マニュアルページでは、システムで定義されているシグナルを列挙しています (あるシステムではリストは `signal(2)` に、別のシステムでは `signal(7)` に列挙されています)。全てのシステムで同じシグナル名のセットを定義しているわけではないので注意してください; このモジュールでは、システムで定義されているシグナル名だけを定義しています。

`signal.CTRL_C_EVENT`

`CTRL+C` キーストロークに該当するシグナル。このシグナルは `os.kill()` でだけ利用できます。

利用出来る環境: Windows.

バージョン 2.7 で追加.

`signal.CTRL_BREAK_EVENT`

`CTRL+BREAK` キーストロークに該当するシグナル。このシグナルは `os.kill()` でだけ利用できます。

利用出来る環境: Windows.

バージョン 2.7 で追加.

`signal.NSIG`

最も大きいシグナル番号に 1 を足した値です。

`signal.ITIMER_REAL`

実時間でデクリメントするインターバルタイマーです。タイマーが発火したときに `SIGALRM` を送ります。

`signal.ITIMER_VIRTUAL`

プロセスの実行時間だけデクリメントするインターバルタイマーです。タイマーが発火したときに SIGVTALRM を送ります。

`signal.ITIMER_PROF`

プロセスの実行中と、システムがそのプロセスのために実行している時間だけデクリメントするインターバルタイマーです。ITIMER_VIRTUAL と組み合わせて、このタイマーはよくアプリケーションがユーザー空間とカーネル空間で消費した時間のプロファイリングに利用されます。タイマーが発火したときに SIGPROF を送ります。

`signal` モジュールは 1 つの例外を定義しています:

exception `signal.ItimerError`

背後の `setitimer()` または `getitimer()` 実装からエラーを通知するために送出されます。無効なインターバルタイマーや負の時間が `setitimer()` に渡された場合、このエラーを予期してください。このエラーは `IOError` を継承しています。

`signal` モジュールでは以下の関数を定義しています:

`signal.alarm(time)`

`time` がゼロでない値の場合、この関数は `time` 秒後頃に SIGALRM をプロセスに送るように要求します。それ以前にスケジュールしたアラームはキャンセルされます (常に一つのアラームしかスケジュールできません)。この場合、戻り値は以前に設定されたアラームシグナルが通知されるまであと何秒だったかを示す値です。 `time` がゼロの場合、アラームは一切スケジュールされず、現在スケジュールされているアラームがキャンセルされます。戻り値がゼロの場合、現在アラームがスケジュールされていないことを示します。(Unix マニュアルページ `alarm(2)` を参照してください)。利用できる環境: Unix。

`signal.getsignal(signalnum)`

シグナル `signalnum` に対する現在のシグナルハンドラを返します。戻り値は呼び出し可能な Python オブジェクトか、`signal.SIG_IGN`、`signal.SIG_DFL`、および `None` といった特殊な値のいずれかです。ここで `signal.SIG_IGN` は以前そのシグナルが無視されていたことを示し、`signal.SIG_DFL` は以前そのシグナルの標準の処理方法が使われていたことを示し、`None` はシグナルハンドラがまだ Python によってインストールされていないことを示します。

`signal.pause()`

シグナルを受け取るまでプロセスを一時停止します; その後、適切なハンドラが呼び出されます。戻り値はありません。Windows では利用できません。(Unix マニュアルページ `signal(2)` を参照してください。)

`signal.setitimer(which, seconds[, interval])`

`which` で指定されたタイマー (`signal.ITIMER_REAL`、`signal.ITIMER_VIRTUAL`、`signal.ITIMER_PROF` のどれか) を、`seconds` 秒後と (`alarm()` と異なり、float を指定できます)、それから `interval` 秒間隔で起動するように設定します。`seconds` に 0 を指定すると、`which` で指定されたタイマーをクリアすることができます。

インターバルタイマーが起動したとき、シグナルがプロセスに送られます。送られるシグナルは利用されたタイマーの種類に依存します。`signal.ITIMER_REAL` の場合は SIGALRM が、`signal.ITIMER_VIRTUAL` の場合は SIGVTALRM が、`signal.ITIMER_PROF` の場合は SIGPROF が送られます。

以前の値が (delay, interval) のタプルとして返されます。

無効なインターバルタイマーを渡すと `ItimerError` 例外が発生します。利用できる環境: Unix。

バージョン 2.6 で追加。

`signal.getitimer (which)`

`which` で指定されたインターバルタイマーの現在の値を返します。利用できる環境: Unix。

バージョン 2.6 で追加。

`signal.set_wakeup_fd (fd)`

wakeup fd を `fd` に設定します。シグナルを受信したときに、`'\0'` バイトがその fd に書き込まれます。これは、poll や select をしているライブラリを起こして、シグナルの処理をさせるのに利用できます。

The old wakeup fd is returned (or -1 if file descriptor wakeup was not enabled). If `fd` is -1, file descriptor wakeup is disabled. If not -1, `fd` must be non-blocking. It is up to the library to remove any bytes from `fd` before calling poll or select again.

スレッドが有効な場合、この関数はメインスレッドからしか実行できません。それ以外のスレッドからこの関数を実行しようとすると `ValueError` 例外が発生します。

バージョン 2.6 で追加。

`signal.siginterrupt (signalnum, flag)`

システムコールのリスタートの動作を変更します。`flag` が `False` の場合、`signalnum` シグナルに中断されたシステムコールは再実行されます。それ以外の場合、システムコールは中断されます。戻り値はありません。利用できる環境: Unix (詳しい情報についてはマニュアルページ `siginterrupt(3)` を参照してください)。

`signal()` を使ってシグナルハンドラを設定したときに、暗黙のうちに `flag` に `true` を指定して `siginterrupt()` が実行されるため、中断に対するリスタートの動作がリセットされることに注意してください。

バージョン 2.6 で追加。

`signal.signal (signalnum, handler)`

シグナル `signalnum` に対するハンドラを関数 `handler` にします。`handler` は二つの引数 (下記参照) を取る呼び出し可能な Python オブジェクトか、`signal.SIG_IGN` あるいは `signal.SIG_DFL` といった特殊な値にすることができます。以前に使われていたシグナルハンドラが返されます (上記の `getsignal()` の記述を参照してください)。(Unix マニュアルページ `signal(2)` を参照してください。)

スレッドが有効な場合、この関数はメインスレッドからしか実行できません。それ以外のスレッドからこの関数を実行しようとすると `ValueError` 例外が発生します。

`handler` は二つの引数とともに呼び出されます: シグナル番号、および現在のスタックフレーム (`None` またはフレームオブジェクト; フレームオブジェクトについての記述は 標準型の階層における説明 か、`inspect` モジュールの属性の説明を参照してください)。

Windows では、`signal()` は `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV`, `SIGTERM` でのみ利用できます。それ以外の場合は `ValueError` を発生させます。

17.4.1 例

以下は最小限のプログラム例です。この例では `alarm()` を使ってファイルを開く処理を待つのに費やす時間を制限します; 例えば、電源の入っていないシリアルデバイスを開こうとすると、通常 `os.open()` は未定義の期間ハングアップしてしましますが、この方法はそうした場合に便利です。ここではファイルを開くまで 5 秒間のアラームを設定することで解決しています; ファイルを開く処理が長くかかりすぎると、アラームシグナルが送信され、ハンドラが例外を送出するようになっています。

```
import signal, os

def handler(signum, frame):
    print 'Signal handler called with signal', signum
    raise IOError("Couldn't open device!")

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)           # Disable the alarm
```

17.5 popen2 — アクセス可能な I/O ストリームを持つサブプロセス生成

バージョン 2.6 で非推奨: このモジュールは時代遅れです。 `subprocess` モジュールを利用してください。特に [古い関数を subprocess モジュールで置き換える](#) 節を参照してください。

このモジュールにより、Unix および Windows でプロセスを起動し、その入力 / 出力 / エラー出力パイプに接続し、そのリターンコードを取得することができます。

`subprocess` モジュールは、新しいプロセスを実行して結果を取得するためのより強力な機能を提供しています。 `popen2` の代わりに `subprocess` モジュールを利用することが推奨されています。

このモジュールで提供されている第一のインタフェースは 3 つのファクトリ関数です。これらの関数のいずれも、`bufsize` を指定した場合、I/O パイプのバッファサイズを決定します。 `mode` を指定する場合、文字列 'b' または 't' でなければなりません; Windows では、ファイルオブジェクトをバイナリあるいはテキストモードのどちらで開くかを決めなければなりません。 `mode` の標準の値は 't' です。

Unix では `cmd` はシーケンスでもよく、その場合には (`os.spawnv()` のように) 引数はシェルを介在させずプログラムに直接渡されます。 `cmd` が文字列の場合、(`os.system()` のように) シェルに渡されます。

子プロセスからのリターンコードを取得するには、 `Popen3` および `Popen4` クラスの `poll()` あるいは `wait()` メソッドを使うしかありません; これらの機能は Unix でしか利用できません。この情報は

`popen2()`、`popen3()`、および `popen4()` 関数、あるいは `os` モジュールにおける同等の関数の使用によっては得ることができません。(`os` モジュールの関数から返されるタプルは `popen2` モジュールの関数から返されるものとは違う順序です。)

`popen2.popen2(cmd[, bufsize[, mode]])`

`cmd` をサブプロセスとして実行します。ファイルオブジェクト (`child.stdout`, `child.stdin`) を返します。

`popen2.popen3(cmd[, bufsize[, mode]])`

`cmd` をサブプロセスとして実行します。ファイルオブジェクト (`child.stdout`, `child.stdin`, `child.stderr`) を返します。

`popen2.popen4(cmd[, bufsize[, mode]])`

`cmd` をサブプロセスとして実行します。ファイルオブジェクト (`child.stdout_and_stderr`, `child.stdin`) を返します。

バージョン 2.0 で追加。

Unix では、ファクトリ関数によって返されるオブジェクトを定義しているクラスも利用することができます。これらのオブジェクトは Windows 実装で使われていないため、そのプラットフォーム上で使うことはできません。

`class popen2.Popen3(cmd[, capturestderr[, bufsize]])`

このクラスは子プロセスを表現します。通常、`Popen3` インスタンスは上で述べた `popen2()` および `popen3()` ファクトリ関数を使って生成されます。

`Popen3` オブジェクトを生成するためにいずれかのヘルパー関数を使っていないのなら、`cmd` パラメータはサブプロセスで実行するシェルコマンドになります。`capturestderr` フラグに真を与えることで、このオブジェクトが子プロセスの標準エラー出力をキャプチャすべきであることを指示します。標準の値は偽です。`bufsize` パラメータを指定すると、子プロセスへの / からの I/O バッファサイズとして使われます。

`class popen2.Popen4(cmd[, bufsize])`

`Popen3` に似ていますが、標準エラー出力を標準出力と同じファイルオブジェクトでキャプチャします。このオブジェクトは通常 `popen4()` で生成されます。

バージョン 2.0 で追加。

17.5.1 Popen3 および Popen4 オブジェクト

`Popen3` および `Popen4` クラスのインスタンスは以下のメソッドを持ちます:

`Popen3.poll()`

子プロセスがまだ終了していない際には `-1` を、そうでない場合にはステータスコード (`wait()` を参照) を返します。

`Popen3.wait()`

子プロセスの状態コード出力を待機して返します。状態コードでは子プロセスのリターンコードと、プ

プロセスが `exit()` によって終了したか、あるいはシグナルによって死んだかについての情報を符号化しています。状態コードの解釈を助けるための関数は `os` モジュールで定義されています; [プロセス管理](#) 節の `W*` () 関数ファミリーを参照してください。

以下の属性も利用可能です:

`Popen3.fromchild`

子プロセスからの出力を提供するファイルオブジェクトです。 `Popen4` インスタンスの場合、この値は標準出力と標準エラー出力の両方を提供するオブジェクトになります。

`Popen3.tochild`

子プロセスへの入力を提供するファイルオブジェクトです。

`Popen3.childerr`

コンストラクタに `capturestderr` を渡した際には子プロセスからの標準エラー出力を提供するファイルオブジェクトで、そうでない場合 `None` になります。 `Popen4` インスタンスでは、この値は常に `None` になります。

`Popen3.pid`

子プロセスのプロセス番号です。

17.5.2 フロー制御の問題

何らかの形式でプロセス間通信を利用している際には常に、制御フローについて注意深く考える必要があります。これはこのモジュール (あるいは `os` モジュールにおける等価な機能) で生成されるファイルオブジェクトの場合にもあてはまります。

親プロセスが子プロセスの標準出力を読み出している一方で、子プロセスが大量のデータを標準エラー出力に書き込んでいる場合、この子プロセスから出力を読み出そうとするとデッドロックが発生します。同様の状況は読み書きの他の組み合わせでも生じます。本質的な要因は、一方のプロセスが別のプロセスでブロック型の読み出しをしている際に、 `_PC_PIPE_BUF` バイトを超えるデータがブロック型の入出力を行うプロセスによって書き込まれることにあります。

こうした状況を扱うには幾つかのやりかたがあります。

多くの場合、もっとも単純なアプリケーションに対する変更は、親プロセスで以下のようなモデル:

```
import popen2

r, w, e = popen2.popen3('python slave.py')
e.readlines()
r.readlines()
r.close()
e.close()
w.close()
```

に従うようにし、子プロセスで以下のようなコードにすることでしょう:

```
import os
import sys

# note that each of these print statements
# writes a single long string

print >>sys.stderr, 400 * 'this is a test\n'
os.close(sys.stderr.fileno())
print >>sys.stdout, 400 * 'this is another test\n'
```

とりわけ、`sys.stderr` は全てのデータを書き込んだ後に閉じられなければならないということに注意してください。さもなければ、`readlines()` は返ってきません。また、`sys.stderr.close()` が `stderr` を閉じないように `os.close()` を使わなければならないことにも注意してください。(そうでなく `sys.stderr` に関連付けると、暗黙のうちに閉じられてしまうので、それ以降のエラーが出力されません)。

より一般的なアプローチをサポートする必要があるアプリケーションでは、パイプ経由の I/O を `select()` ループでまとめるか、個々の `popen*`() 関数や `Popen*` クラスが提供する各々のファイルに対して、個別のスレッドを使って読み出しを行います。

参考:

`subprocess` モジュール サブプロセスの生成と管理のためのモジュール。

17.6 `asyncore` — 非同期ソケットハンドラ

ソースコード: [Lib/asyncore.py](#)

このモジュールは、非同期ソケットサービスのクライアント・サーバを開発するための基盤として使われます。

CPU が一つしかない場合、プログラムが”二つのことを同時に”実行する方法は二つしかありません。もっとも簡単で一般的なのはマルチスレッドを利用する方法ですが、これとはまったく異なるテクニックで、一つのスレッドだけでマルチスレッドと同じような効果を得られるテクニックがあります。このテクニックは I/O 処理が中心である場合にのみ有効で、CPU 負荷の高いプログラムでは効果が無く、この場合にはプリエンブティブなスケジューリングが可能なスレッドが有効でしょう。しかし、多くの場合、ネットワークサーバでは CPU 負荷よりは IO 負荷が問題となります。

もし OS の I/O ライブラリがシステムコール `select()` をサポートしている場合(ほとんどの場合はサポートされている) I/O 処理は”バックグラウンド”で実行し、その間に他の処理を実行すれば、複数の通信チャネルを同時にこなすことができます。一見、この戦略は奇妙で複雑に思えるかもしれませんが、いろいろな面でマルチスレッドよりも理解しやすく、制御も容易です。`asyncore` は多くの複雑な問題を解決済みなので、洗練され、パフォーマンスにも優れたネットワークサーバとクライアントを簡単に開発することができます。とくに、`asynchat` のような、対話型のアプリケーションやプロトコルには非常に有効でしょう。

基本的には、この二つのモジュールを使う場合は一つ以上のネットワーク チャネル を `asyncore.dispatcher` クラス、または `asynchat.async_chat` のインスタンスとして作成します。作成されたチャネルはグローバルマップに登録され、`loop()` 関数で参照されます。`loop()` には、専用の マップ を渡す事も可能です。

チャンネルを生成後、`loop()` を呼び出すとチャンネル処理が開始し、最後のチャンネル（非同期処理中にマップに追加されたチャンネルを含む）が閉じるまで継続します。

```
asyncore.loop([timeout[, use_poll[, map[, count]]]])
```

ポーリングループを開始し、count 回が過ぎるか、全てのオープン済みチャンネルがクローズされた場合のみ終了します。全ての引数はオプションです。引数 `count` のデフォルト値は `None` で、ループは全てのチャンネルがクローズされた場合のみ終了します。引数 `timeout` は `select()` または `poll()` の引数 `timeout` として渡され、秒単位で指定します。デフォルト値は 30 秒です。引数 `use_poll` が真の場合、`select()` ではなく `poll()` が使われます（デフォルト値は `False` です）。

引数 `map` には、監視するチャンネルをアイテムとして格納した辞書を指定します。チャンネルがクローズされた時に `map` からそのチャンネルが削除されます。`map` が省略された場合、グローバルなマップが使用されます。チャンネル (`asyncore.dispatcher`, `asynchat.async_chat` とそのサブクラス) は自由に混ぜて `map` に入れることができます。

```
class asyncore.dispatcher
```

`dispatcher` クラスは、低レベルソケットオブジェクトの薄いラッパーです。便宜上、非同期ループから呼び出されるイベント処理メソッドを追加していますが、これ以外の点では、non-blocking なソケットと同様です。

非同期ループ内で低レベルイベントが発生した場合、発生タイミングや接続の状態から特定の高レベルイベントへと置き換えることができます。例えばソケットを他のホストに接続する場合、最初の書き込み可能イベントが発生すれば接続が完了した事が分かります（この時点で、ソケットへの書き込みは成功すると考えられる）。このように判定できる高レベルイベントを以下に示します：

Event	説明
<code>handle_connect()</code>	最初に read もしくは write イベントが発生した時
<code>handle_close()</code>	読み込み可能なデータなしで read イベントが発生した時
<code>handle_accepted()</code>	listen 中のソケットで read イベントが発生した時

非同期処理中、マップに登録されたチャンネルの `readable()` メソッドと `writable()` メソッドが呼び出され、`select()` か `poll()` で read/write イベントを検出するリストに登録するか否かを判定します。

このようにして、チャンネルでは低レベルなソケットイベントの種類より多くの種類のイベントを検出する事ができます。以下にあげるイベントは、サブクラスでオーバーライドすることが可能です：

```
handle_read()
```

非同期ループで、チャンネルのソケットの `read()` メソッドの呼び出しが成功した時に呼び出されます。

```
handle_write()
```

非同期ループで、書き込み可能ソケットが実際に書き込み可能になった時に呼び出される。このメソッドは、パフォーマンスの向上のためバッファリングを行う場合などに利用できます。例：

```
def handle_write(self):
    sent = self.send(self.buffer)
    self.buffer = self.buffer[sent:]
```

handle_expt()

out of band (OOB) データが検出された時に呼び出されます。OOB はあまりサポートされておらず、また滅多に使われないので、`handle_expt()` が呼び出されることはほとんどありません。

handle_connect()

ソケットの接続が確立した時に呼び出されます。"welcome" バナーの送信、プロトコルネゴシエーションの初期化などを行います。

handle_close()

ソケットが閉じた時に呼び出されます。

handle_error()

捕捉されない例外が発生した時に呼び出されます。デフォルトでは、短縮したトレースバック情報が出力されます。

handle_accept()

listen 中のチャネル (受動的にオープンしたもの) がリモートホストからの `connect()` で接続され、接続が確立した時に呼び出されます。

readable()

非同期ループ中に呼び出され、read イベントの監視リストに加えるか否かを決定します。デフォルトのメソッドでは `True` を返し、read イベントの発生を監視します。

writable()

非同期ループ中に呼び出され、write イベントの監視リストに加えるか否かを決定します。デフォルトのメソッドでは `True` を返し、write イベントの発生を監視します。

さらに、チャネルにはソケットのメソッドとほぼ同じメソッドがあり、チャネルはソケットのメソッドの多くを委譲・拡張しており、ソケットとほぼ同じメソッドを持っています。

create_socket(family, type)

引数も含め、通常のソケット生成と同じ。 `socket` モジュールを参照のこと。

connect(address)

通常のソケットオブジェクトと同様、`address` には一番目の値が接続先ホスト、2 番目の値がポート番号であるタプルを指定します。

send(data)

リモート側の端点に `data` を送出します。

recv(buffer_size)

リモート側の端点より、最大 `buffer_size` バイトのデータを読み込みます。長さ 0 の文字列が返ってきた場合、チャネルはリモートから切断された事を示します。

`recv()` が `EAGAIN` または `EWOULDBLOCK` による `socket.error` を起こしうることに注意してください。 `select.select()` や `select.poll()` が、ソケットが読み込み出来る状態にあ

ると報告したとしてもです。

listen(*backlog*)

ソケットへの接続を待つ。引数 *backlog* は、キューイングできるコネクションの最大数を指定します (1 以上)。最大数はシステムに依存でします (通常は 5)。

bind(*address*)

ソケットを *address* にバインドします。ソケットはバインド済みであってはなりません。(*address* の形式は、アドレスファミリに依存します。 *socket* モジュールを参照のこと。) ソケットを再利用可能にする (`SO_REUSEADDR` オプションを設定する) には、 *dispatcher* オブジェクトの `set_reuse_addr()` メソッドを呼び出してください。

accept()

接続を受け入れます。ソケットはアドレスにバインド済みであり、`listen()` で接続待ち状態でなければなりません。戻り値は `None` か (`conn`, `address`) のペアで、`conn` はデータの送受信を行う新しいソケットオブジェクト、`address` は接続先ソケットがバインドされているアドレスです。`None` が返された場合、接続が起こらなかったことを意味します。その場合、サーバーはこのイベントを無視して後続の接続を待ち続けるべきです。

close()

ソケットをクローズします。以降の全ての操作は失敗します。リモート端点では、キューに溜まったデータ以外、これ以降のデータ受信は行えません。ソケットはガベージコレクション時に自動的にクローズされます。

class `asyncore.dispatcher.with.send`

dispatcher のサブクラスで、シンプルなバッファされた出力を持ちます。シンプルなクライアントプログラムに適しています。もっと高レベルな場合には `asynchat.async_chat` を利用してください。

class `asyncore.file_dispatcher`

`file_dispatcher` はファイルデスクリプタか **ファイルオブジェクト** とオプションとして `map` を引数にとつて、`poll()` か `loop()` 関数で利用できるようにラップします。与えられたファイルオブジェクトなどが `fileno()` メソッドを持っているとき、そのメソッドが呼び出されて戻り値が *file-wrapper* のコンストラクタに渡されます。利用できるプラットフォーム: UNIX。

class `asyncore.file_wrapper`

`file_wrapper` は整数のファイルデスクリプタを受け取って `os.dup()` を呼び出してハンドルを複製するので、元のハンドルは `file_wrapper` と独立して `close` されます。このクラスは *file_dispatcher* クラスが使うために必要なソケットをエミュレートするメソッドを実装しています。利用できるプラットフォーム: UNIX。

17.6.1 `asyncore` の例: 簡単な HTTP クライアント

基本的なサンプルとして、以下に非常に単純な HTTP クライアントを示します。この HTTP クライアントは *dispatcher* クラスでソケットを利用しています:

```

import asyncore, socket

class HTTPClient(asyncore.dispatcher):

    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.connect( (host, 80) )
        self.buffer = 'GET %s HTTP/1.0\r\n\r\n' % path

    def handle_connect(self):
        pass

    def handle_close(self):
        self.close()

    def handle_read(self):
        print self.recv(8192)

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]

client = HTTPClient('www.python.org', '/')
asyncore.loop()

```

17.6.2 基本的な echo サーバーの例

この例の基本的な echo サーバーは、*dispatcher* を利用して接続を受けつけ、接続をハンドラーにディスパッチします:

```

import asyncore
import socket

class EchoHandler(asyncore.dispatcher_with_send):

    def handle_read(self):
        data = self.recv(8192)
        if data:
            self.send(data)

class EchoServer(asyncore.dispatcher):

    def __init__(self, host, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)

```

(次のページに続く)

(前のページからの続き)

```
self.set_reuse_addr()
self.bind((host, port))
self.listen(5)

def handle_accept(self):
    pair = self.accept()
    if pair is not None:
        sock, addr = pair
        print 'Incoming connection from %s' % repr(addr)
        handler = EchoHandler(sock)

server = EchoServer('localhost', 8080)
asyncore.loop()
```

17.7 `asyncchat` — 非同期ソケットコマンド/レスポンスハンドラ

ソースコード: `Lib/asyncchat.py`

`asyncchat` を使うと、`asyncore` を基盤とした非同期なサーバ・クライアントをより簡単に開発する事ができます。`asyncchat` では、プロトコルの要素が任意の文字列で終了するか、または可変長の文字列であるようなプロトコルを容易に制御できるようになっています。`asyncchat` は、抽象クラス `async_chat` を定義しており、`async_chat` を継承して `collect_incoming_data()` メソッドと `found_terminator()` メソッドを実装すれば使うことができます。`async_chat` と `asyncore` は同じ非同期ループを使用しており、`asyncore.dispatcher` も `asyncchat.async_chat` も同じチャネルマップに登録する事ができます。通常、`asyncore.dispatcher` はサーバチャネルとして使用し、リクエストの受け付け時に `asyncchat.async_chat` オブジェクトを生成します。

`class asyncchat.async_chat`

このクラスは、`asyncore.dispatcher` から継承した抽象クラスです。使用する際には `async_chat` のサブクラスを作成し、`collect_incoming_data()` と `found_terminator()` を定義しなければなりません。`asyncore.dispatcher` のメソッドを使用する事もできますが、メッセージ/レスポンス処理を中心に行う場合には使えないメソッドもあります。

`asyncore.dispatcher` と同様に、`async_chat` も `select()` 呼出し後のソケットの状態からイベントを生成します。ポーリングループ開始後、イベント処理フレームワークが自動的に `async_chat` のメソッドを呼び出しますので、プログラマが処理を記述する必要はありません。

パフォーマンスの向上やメモリの節約のために、2 つのクラス属性を調整することができます。

`ac.in_buffer_size`

非同期入力バッファサイズ (デフォルト値: 4096)。

`ac.out_buffer_size`

非同期出力バッファサイズ (デフォルト値: 4096)。

`asyncore.dispatcher` と違い、`async_chat` では *producer* の first-in-first-out キュー (fifo) を作成する事ができます。producer は `more()` メソッドを必ず持ち、このメソッドでチャンネル上に送出するデータを返します。producer が枯渇状態 (*i.e.* これ以上のデータを持たない状態) にある場合、`more()` は空文字列を返します。この時、`async_chat` は枯渇状態にある producer を fifo から除去し、次の producer が存在すればその producer を使用します。fifo に producer が存在しない場合、`handle_write()` は何もしません。リモート端点からの入力の終了や重要な中断点を検出する場合は、`set_terminator()` に記述します。

`async_chat` のサブクラスでは、入力メソッド `collect_incoming_data()` と `found_terminator()` を定義し、チャンネルが非同期に受信するデータを処理します。これらのメソッドについては後で解説します。

`async_chat.close_when_done()`

producer fifo のトップに `None` をプッシュします。この producer がポップされると、チャンネルがクローズします。

`async_chat.collect_incoming_data(data)`

チャンネルが受信した不定長のデータを `data` に指定して呼び出されます。このメソッドは必ずオーバーライドする必要があり、デフォルトの実装では、`NotImplementedError` 例外を送出します。

`async_chat.discard_buffers()`

非常用のメソッドで、全ての入出力バッファと producer fifo を廃棄します。

`async_chat.found_terminator()`

入力データストリームが、`set_terminator()` で指定した終了条件と一致した場合に呼び出されます。このメソッドは必ずオーバーライドする必要があり、デフォルトの実装では、`NotImplementedError` 例外を送出します。入力データを参照する必要がある場合でも引数としては与えられないため、入力バッファをインスタンス属性として参照しなければなりません。

`async_chat.get_terminator()`

現在のチャンネルの終了条件を返します。

`async_chat.push(data)`

チャンネルの fifo にデータをプッシュして転送します。データをチャンネルに書き出すために必要なのはこれだけですが、データの暗号化やチャンク化などを行う場合には独自の producer を使用する事もできます。

`async_chat.push_with_producer(producer)`

指定した producer オブジェクトをチャンネルの fifo に追加します。これより前に push された producer が全て枯渇した後、チャンネルはこの producer から `more()` メソッドでデータを取得し、リモート端点に送信します。

`async_chat.set_terminator(term)`

チャンネルで検出する終了条件を設定します。`term` は入力プロトコルデータの処理方式によって以下の3つの型の何れかを指定します。

term	説明
<i>string</i>	入力ストリーム中で <i>string</i> が検出された時、 <i>found_terminator()</i> を呼び出します
<i>integer</i>	指定された文字数が読み込まれた時、 <i>found_terminator()</i> を呼び出します
None	永久にデータを読み込みます

終了条件が成立しても、その後に続くデータは、 *found_terminator()* の呼出し後に再びチャンネルを読み込めば取得する事ができます。

17.7.1 *asyncchat* - 補助クラス

class *asyncchat.fifo* (*list=None*)

アプリケーションからプッシュされ、まだチャンネルに書き出されていないデータを保持するための *fifo*。 *fifo* は必要になるまでデータと producer を保持するために使われるリストです。引数 *list* には、チャンネルに出力する producer またはデータを指定する事ができます。

is_empty()

fifo が空のとき (のみ) に True を返します。

first()

fifo に *push()* されたアイテムのうち、最も古いアイテムを返します。

push(*data*)

データ (文字列または producer オブジェクト) を producer *fifo* に追加します。

pop()

fifo が空でなければ、(True, *first()*) を返し、ポップされたアイテムを削除します。 *fifo* が空であれば (False, None) を返します。

17.7.2 *asyncchat* 使用例

以下のサンプルは、 *async_chat* で HTTP リクエストを読み込む処理の一部です。Web サーバは、クライアントからの接続毎に *http_request_handler* オブジェクトを作成します。最初はチャンネルの終了条件に空行を指定して HTTP ヘッダの末尾までを検出し、その後ヘッダ読み込み済みを示すフラグを立てています。

ヘッダ読み込んだ後、リクエストの種類が POST であればデータが入力ストリームに流れるため、Content-Length: ヘッダの値を数値として終了条件に指定し、適切な長さのデータをチャンネルから読み込みます。

必要な入力データを全て入手したら、チャンネルの終了条件に None を指定して残りのデータを無視するようにしています。この後、 *handle_request()* が呼び出されます。

```
class http_request_handler(asyncchat.async_chat):

    def __init__(self, sock, addr, sessions, log):
        asyncchat.async_chat.__init__(self, sock=sock)
```

(次のページに続く)

(前のページからの続き)

```
self.addr = addr
self.sessions = sessions
self.ibuffer = []
self.obuffer = ""
self.set_terminator("\r\n\r\n")
self.reading_headers = True
self.handling = False
self.cgi_data = None
self.log = log

def collect_incoming_data(self, data):
    """Buffer the data"""
    self.ibuffer.append(data)

def found_terminator(self):
    if self.reading_headers:
        self.reading_headers = False
        self.parse_headers("".join(self.ibuffer))
        self.ibuffer = []
        if self.op.upper() == "POST":
            clen = self.headers.getheader("content-length")
            self.set_terminator(int(clen))
        else:
            self.handling = True
            self.set_terminator(None)
            self.handle_request()
    elif not self.handling:
        self.set_terminator(None) # browsers sometimes over-send
        self.cgi_data = parse(self.headers, "".join(self.ibuffer))
        self.handling = True
        self.ibuffer = []
        self.handle_request()
```


第 18 章

インターネット上のデータの操作

この章ではインターネット上で一般的に利用されているデータ形式の操作をサポートするモジュール群について記述します。

18.1 `email` — 電子メールと MIME 処理のためのパッケージ

バージョン 2.2 で追加.

`email` パッケージは電子メールのメッセージを管理するライブラリです。これには MIME やそれ以外の **RFC 2822** ベースのメッセージ文書もふくまれます。このパッケージはいくつかの古い標準パッケージ、`rfc822`, `mimetools`, `multifile` などにふくまれていた機能のほとんどを持ち、加えて標準ではなかった `mimecntl` などの機能も含んでいます。このパッケージは、とくに電子メールのメッセージを SMTP (**RFC 2821**)、NNTP、その他のサーバに送信するために作られているというわけではありません。それは `smtplib`, `nntplib` モジュールなどの機能です。`email` パッケージは **RFC 2822** に加えて、**RFC 2045**, **RFC 2046**, **RFC 2047** および **RFC 2231** など MIME 関連の RFC をサポートしており、できるかぎり RFC に準拠することをめざしています。

`email` パッケージの一番の特徴は、電子メールの内部表現である オブジェクトモデル と、電子メールメッセージの解析および生成とを分離していることです。`email` パッケージを使うアプリケーションは基本的にはオブジェクトを処理することができます。メッセージに子オブジェクトを追加したり、メッセージから子オブジェクトを削除したり、内容を完全に並べかえたり、といったことができます。フラットなテキスト文書からオブジェクトモデルへの変換、またそこからフラットな文書へと戻す変換はそれぞれ別々の解析器 (パーサ) と生成器 (ジェネレータ) が担当しています。また、一般的な MIME オブジェクトタイプのいくつかについては手軽なサブクラスが存在しており、メッセージフィールド値を抽出したり解析したり、RFC 準拠の日付を生成したりなどのよくおこわれるタスクについてはいくつかの雑用ユーティリティもついています。

以下の節では `email` パッケージの機能を説明します。説明の順序は多くのアプリケーションで一般的な使用順序にもとづいています。まず、電子メールメッセージをファイルあるいはその他のソースからフラットなテキスト文書として読み込み、つぎにそのテキストを解析して電子メールのオブジェクト構造を作成し、その構造を操作して、最後にオブジェクトツリーをフラットなテキストに戻す、という順序になっています。

このオブジェクト構造は、まったくのゼロから作りだしたものであってもしっこうにかまいません。この場合も上と似たような作業順序になるでしょう。

またここには `email` パッケージが提供するすべてのクラスおよびモジュールに関する説明と、`email` パッケージを使っていくうえで遭遇するかもしれない例外クラス、いくつかの補助ユーティリティ、そして少々のサンプルも含まれています。古い `mimelib` や前バージョンの `email` パッケージのユーザのために、現行バージョンとの違いと移植についての節も設けてあります。

`email` パッケージ文書の内容

18.1.1 `email.message`: 電子メールメッセージの表現

`Message` クラスは、`email` パッケージの中心となるクラスです。これは `email` オブジェクトモデルの基底クラスになっています。`Message` はヘッダフィールドを検索したりメッセージ本体にアクセスするための核となる機能を提供します。

概念的には、(`email.message` モジュールからインポートされる)`Message` オブジェクトにはヘッダとペイロードが格納されています。ヘッダは、**RFC 2822** 形式のフィールド名およびフィールド値がコロンで区切られたものです。コロンはフィールド名またはフィールド値のどちらにも含まれません。

ヘッダは大文字小文字を区別した形式で保存されますが、ヘッダ名が一致するかどうかの検査は大文字小文字を区別せずにおこなうことができます。`Unix-From` ヘッダまたは `From_` ヘッダとして知られるエンベロープヘッダがひとつ存在することもあります。ペイロードは、単純なメッセージオブジェクトの場合は単なる文字列ですが、MIME コンテナ文書 (`multipart/*` または `message/rfc822` など) の場合は `Message` オブジェクトのリストになっています。

`Message` オブジェクトは、メッセージヘッダにアクセスするためのマップ (辞書) 形式のインタフェースと、ヘッダおよびペイロードの両方にアクセスするための明示的なインタフェースを提供します。これにはメッセージオブジェクトツリーからフラットなテキスト文書を生成したり、一般的に使われるヘッダのパラメータにアクセスしたり、またオブジェクトツリーを再帰的にたどったりするための便利なメソッドを含みます。

`Message` クラスのメソッドは以下のとおりです:

class `email.message.Message`

コンストラクタは引数を取りません。

as_string (`[unixfrom]`)

メッセージ全体をフラットな文字列として返します。オプション `unixfrom` が `True` の場合、返される文字列にはエンベロープヘッダも含まれます。`unixfrom` のデフォルトは `False` です。もし、文字列への変換を完全に行うためにデフォルト値を埋める必要がある場合、メッセージのフラット化は `Message` の変更を引き起こす可能性があります (例えば、MIME の境界が生成される、変更される等)。

このメソッドは手軽に利用する事ができますが、必ずしも期待通りにメッセージをフォーマットするとは限りません。たとえば、これはデフォルトでは `From` で始まる行を変更してしまいます。以下の例のように `Generator` のインスタンスを生成して `flatten()` メソッドを直接呼び出せばより柔軟な処理を行う事ができます。

```
from cStringIO import StringIO
from email.generator import Generator
```

(次のページに続く)

(前のページからの続き)

```
fp = StringIO()
g = Generator(fp, mangle_from=False, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

`__str__()`

`as_string(unixfrom=True)()` と同じです。

`is_multipart()`

メッセージのペイロードが子 *Message* オブジェクトからなるリストであれば `True` を返し、そうでなければ `False` を返します。 `is_multipart()` が `False` を返した場合は、ペイロードは文字列オブジェクトである必要があります。

`set_unixfrom(unixfrom)`

メッセージのエンベロープヘッダを *unixfrom* に設定します。これは文字列である必要があります。

`get_unixfrom()`

メッセージのエンベロープヘッダを返します。エンベロープヘッダが設定されていない場合は `None` が返されます。

`attach(payload)`

与えられた *payload* を現在のペイロードに追加します。この時点でのペイロードは `None` か、あるいは *Message* オブジェクトのリストである必要があります。このメソッドの実行後、ペイロードは必ず *Message* オブジェクトのリストになります。ペイロードにスカラーオブジェクト (文字列など) を格納したい場合は、かわりに `set_payload()` を使ってください。

`get_payload([i[, decode]])`

現在のペイロードへの参照を返します。これは `is_multipart()` が `True` の場合 *Message* オブジェクトのリストになり、`is_multipart()` が `False` の場合は文字列になります。ペイロードがリストの場合、リストを変更することはそのメッセージのペイロードを変更することになります。

オプション引数の *i* がある場合、`is_multipart()` が `True` ならば `get_payload()` はペイロード中で 0 から数えて *i* 番目の要素を返します。*i* が 0 より小さい場合、あるいはペイロードの個数以上の場合は *IndexError* が発生します。ペイロードが文字列 (つまり `is_multipart()` が `False`) にもかかわらず *i* が与えられたときは *TypeError* が発生します。

オプションの *decode* はそのペイロードが *Content-Transfer-Encoding* ヘッダに従ってデコードされるべきかどうかを指示するフラグです。この値が `True` でメッセージが *multipart* ではない場合、ペイロードはこのヘッダの値が *quoted-printable* または *base64* のときにかぎりデコードされます。これ以外のエンコーディングが使われている場合、*Content-Transfer-Encoding* ヘッダがない場合、あるいは曖昧な *base64* データが含まれる場合は、ペイロードはそのまま (デコードされずに) 返されます。もしメッセージが *multipart* で *decode* フラグが `True` の場合は `None` が返されます。*decode* のデフォルト値は `False` です。

`set_payload(payload[, charset])`

メッセージ全体のオブジェクトのペイロードを *payload* に設定します。ペイロードの形式をととの

えるのは呼び出し側の責任です。オプションの `charset` はメッセージのデフォルト文字セットを設定します。詳しくは `set_charset()` を参照してください。

バージョン 2.2.2 で変更: `charset` 引数を追加。

`set_charset(charset)`

ペイロードの文字セットを `charset` に変更します。ここには `Charset` インスタンス (`email.charset` 参照)、文字セット名をあらわす文字列、あるいは `None` のいずれかが指定できます。文字列を指定した場合、これは `Charset` インスタンスに変換されます。`charset` が `None` の場合、`charset` パラメータは `Content-Type` ヘッダから除去されます (それ以外にメッセージの変形は行われません)。これ以外のものを文字セットとして指定した場合、`TypeError` が発生します。

`MIME-Version` ヘッダが存在しなければ、追加されます。`Content-Type` ヘッダが存在しなければ、`text/plain` を値として追加されます。`Content-Type` が存在していてもいなくても、`charset` パラメータは `charset.output_charset` に設定されます。`charset.input_charset` と `charset.output_charset` が異なるなら、ペイロードは `output_charset` に再エンコードされます。`Content-Transfer-Encoding` ヘッダが存在しなければ、ペイロードは、必要なら指定された `Charset` を使って transfer エンコードされ、適切な値のヘッダが追加されます。`Content-Transfer-Encoding` ヘッダがすでに存在すれば、ペイロードはすでにその `Content-Transfer-Encoding` によって正しくエンコードされたものと見なされ、変形されません。

ここでいうメッセージとは、unicode 文字列か `charset.input_charset` でエンコードされたペイロードを持つ `text/*` 形式のものを仮定しています。これは、プレーンテキスト形式を生成する際に `charset.output_charset` と転送 (transfer) エンコードに、エンコードまたは変換されます。MIME ヘッダ (`MIME-Version`, `Content-Type`, `Content-Transfer-Encoding`) は必要に応じて追加されます。

バージョン 2.2.2 で追加。

`get_charset()`

そのメッセージ中のペイロードの `Charset` インスタンスを返します。

バージョン 2.2.2 で追加。

以下のメソッドは、メッセージの **RFC 2822** ヘッダにアクセスするためのマップ (辞書) 形式のインタフェースを実装したものです。これらのメソッドと、通常のマップ (辞書) 型はまったく同じ意味をもつわけではないことに注意してください。たとえば辞書型では、同じキーが複数あることは許されていませんが、ここでは同じメッセージヘッダが複数ある場合があります。また、辞書型では `keys()` で返されるキーの順序は保証されていませんが、`Message` オブジェクト内のヘッダはつねに元のメッセージ中に現れた順序、あるいはそのあとに追加された順序で返されます。削除され、その後ふたたび追加されたヘッダはリストの一番最後に現れます。

こういった意味のちがいは意図的なもので、最大の利便性をもつようにつくられています。

注意: どんな場合も、メッセージ中のエンベロープヘッダはこのマップ形式のインタフェースには含まれません。

`__len__()`

複製されたものもふくめてヘッダ数の合計を返します。

`--contains--(name)`

メッセージオブジェクトが *name* という名前のフィールドを持っていれば `true` を返します。この検査では名前の大文字小文字は区別されません。*name* は最後にコロンをふくんでいてはいけません。このメソッドは以下のように `in` 演算子で使われます:

```
if 'message-id' in myMessage:
    print 'Message-ID:', myMessage['message-id']
```

`--getitem--(name)`

指定された名前のヘッダフィールドの値を返します。*name* は最後にコロンをふくんでいてはいけません。そのヘッダがない場合は `None` が返され、`KeyError` 例外は発生しません。

注意: 指定された名前のフィールドがメッセージのヘッダに 2 回以上現れている場合、どちらの値が返されるかは未定義です。ヘッダに存在するフィールドの値をすべて取り出したい場合は `get_all()` メソッドを使ってください。

`--setitem--(name, val)`

メッセージヘッダに *name* という名前の *val* という値をもつフィールドをあらたに追加します。このフィールドは現在メッセージに存在するフィールドのいちばん後に追加されます。

注意: このメソッドでは、すでに同一の名前で存在するフィールドは上書きされません。もしメッセージが名前 *name* をもつフィールドをひとつしか持たないようにしたければ、最初にそれを除去してください。たとえば:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

`--delitem--(name)`

メッセージのヘッダから、*name* という名前をもつフィールドをすべて除去します。たとえこの名前をもつヘッダが存在していなくても例外は発生しません。

`has_key(name)`

メッセージが *name* という名前をもつヘッダフィールドを持っていれば真を、そうでなければ偽を返します。

`keys()`

メッセージ中にあるすべてのヘッダのフィールド名のリストを返します。

`values()`

メッセージ中にあるすべてのフィールドの値のリストを返します。

`items()`

メッセージ中にあるすべてのヘッダのフィールド名とその値を 2-タプルのリストとして返します。

`get(name[, failobj])`

指定された名前をもつフィールドの値を返します。これは指定された名前がないときにオプション引数の *failobj* (デフォルトでは `None`) を返すことをのぞけば、`--getitem--()` と同じです。

さらに、役に立つメソッドをいくつか紹介します:

get_all (*name* [, *failobj*])

name の名前をもつフィールドのすべての値からなるリストを返します。該当する名前のヘッダがメッセージ中に含まれていない場合は *failobj* (デフォルトでは `None`) が返されます。

add_header (*_name*, *_value*, ***params*)

拡張ヘッダ設定。このメソッドは `__setitem__()` と似ていますが、追加のヘッダ・パラメータをキーワード引数で指定できるところが違ってしています。 *_name* に追加するヘッダフィールドを、 *_value* にそのヘッダの 最初の 値を渡します。

キーワード引数辞書 *params* の各項目ごとに、そのキーがパラメータ名として扱われ、キー名にふくまれるアンダースコアはハイフンに置換されます (アンダースコアは、Python 識別子としてハイフンを使えないための代替です)。ふつう、パラメータの値が `None` 以外のときは、`key="value"` の形で追加されます。パラメータの値が `None` のときはキーのみが追加されます。値が非 ASCII 文字を含むなら、それは (CHARSET, LANGUAGE, VALUE) の形式の 3 タプルでなくてはなりません。ここで CHARSET はその値をエンコードするのに使われる文字セットを指示する文字列で、LANGUAGE は通常 `None` か空文字列にでき (これら以外で指定出来るものについては [RFC 2231](#) を参照してください)、VALUE は非 ASCII コードポイントを含む文字列値です。

以下はこの使用例です:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

こうするとヘッダには以下のように追加されます

```
Content-Disposition: attachment; filename="bud.gif"
```

非 ASCII 文字を使った例:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fu^c3^9fballer.ppt'))
```

は、以下ようになります

```
Content-Disposition: attachment; filename*="iso-8859-1"Fu%DFballer.ppt"
```

replace_header (*_name*, *_value*)

ヘッダの置換。 *_name* と一致するヘッダで最初に見つかったものを置き換えます。このときヘッダの順序とフィールド名の大文字小文字は保存されます。一致するヘッダがない場合、`KeyError` が発生します。

バージョン 2.2.2 で追加.

get_content_type ()

そのメッセージの content-type を返します。返された文字列は強制的に小文字で *maintype/subtype* の形式に変換されます。メッセージ中に *Content-Type* ヘッダがない場合、デフォルトの content-type は `get_default_type()` が返す値によって与えられます。 [RFC 2045](#) によ

ばメッセージはつねにデフォルトの content-type をもっているので、`get_content_type()` はつねになんらかの値を返すはずです。

RFC 2045 はメッセージのデフォルト content-type を、それが *multipart/digest* コンテナに現れているとき以外は *text/plain* に規定しています。あるメッセージが *multipart/digest* コンテナ中にある場合、その content-type は *message/rfc822* になります。もし *Content-Type* ヘッダが適切でない content-type 書式だった場合、**RFC 2045** はそのデフォルトを *text/plain* として扱うよう定めています。

バージョン 2.2.2 で追加。

`get_content_maintype()`

そのメッセージの主 content-type を返します。これは `get_content_type()` によって返される文字列の *maintype* 部分です。

バージョン 2.2.2 で追加。

`get_content_subtype()`

そのメッセージの副 content-type (sub content-type、subtype) を返します。これは `get_content_type()` によって返される文字列の *subtype* 部分です。

バージョン 2.2.2 で追加。

`get_default_type()`

デフォルトの content-type を返します。ほとんどのメッセージではデフォルトの content-type は *text/plain* ですが、メッセージが *multipart/digest* コンテナに含まれているときだけ例外的に *message/rfc822* になります。

バージョン 2.2.2 で追加。

`set_default_type(ctype)`

デフォルトの content-type を設定します。ctype は *text/plain* あるいは *message/rfc822* である必要がありますが、強制ではありません。デフォルトの content-type はヘッダの *Content-Type* には格納されません。

バージョン 2.2.2 で追加。

`get_params([failobj[, header[, unquote]]])`

メッセージの *Content-Type* パラメータをリストとして返します。返されるリストはキー/値の組からなる 2 要素タプルが連なったものであり、これらは '=' 記号で分離されています。 '=' の左側はキーになり、右側は値になります。パラメータ中に '=' がなかった場合、値の部分は空文字列になり、そうでなければその値は `get_param()` で説明されている形式になります。また、オプション引数 *unquote* が `True` (デフォルト) である場合、この値は *unquote* されます。

オプション引数 *failobj* は、*Content-Type* ヘッダが存在しなかった場合に返すオブジェクトです。オプション引数 *header* には *Content-Type* のかわりに検索すべきヘッダを指定します。

バージョン 2.2.2 で変更: *unquote* 引数の追加。

`get_param(param[, failobj[, header[, unquote]]])`

メッセージの *Content-Type* ヘッダ中のパラメータ *param* を文字列として返します。そのメッ

セージ中に *Content-Type* ヘッダが存在しなかった場合、*failobj* (デフォルトは `None`) が返されます。

オプション引数 *header* が与えられた場合、*Content-Type* のかわりにそのヘッダが使用されます。

パラメータのキー比較は常に大文字小文字を区別しません。返り値は文字列か 3 要素のタプルで、タプルになるのはパラメータが **RFC 2231** エンコードされている場合です。3 要素タプルの場合、各要素の値は (*CHARSET*, *LANGUAGE*, *VALUE*) の形式になっています。*CHARSET* と *LANGUAGE* は `None` になることがあり、その場合 *VALUE* は *us-ascii* 文字セットでエンコードされているとみなさなければならないので注意してください。普段は *LANGUAGE* を無視できます。

この関数を使うアプリケーションが、パラメータが **RFC 2231** 形式でエンコードされているかどうかを気にしないのであれば、`email.utils.collapse_rfc2231_value()` に `get_param()` の返り値を渡して呼び出すことで、このパラメータをひとつにまとめることができます。この値がタプルならばこの関数は適切にデコードされた Unicode 文字列を返し、そうでない場合は `unquote` された元の文字列を返します。たとえば:

```
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

いずれの場合もパラメータの値は (文字列であれ 3 要素タプルの *VALUE* 項目であれ) つねに `unquote` されます。ただし、`unquote` が `False` に指定されている場合は `unquote` されません。

バージョン 2.2.2 で変更: `unquote` 引数の追加、3 要素タプルが返り値になる可能性あり。

set_param (*param*, *value*[, *header*[, *requote*[, *charset*[, *language*]]]])

Content-Type ヘッダ中のパラメータを設定します。指定されたパラメータがヘッダ中にすでに存在する場合、その値は *value* に置き換えられます。*Content-Type* ヘッダがまだこのメッセージ中に存在していない場合、**RFC 2045** にしたがってこの値には *text/plain* が設定され、新しいパラメータ値が末尾に追加されます。

オプション引数 *header* が与えられた場合、*Content-Type* のかわりにそのヘッダが使用されます。オプション引数 *requote* が `False` でない限り、この値は `quote` されます (デフォルトは `True`)。

オプション引数 *charset* が与えられると、そのパラメータは **RFC 2231** に従ってエンコードされます。オプション引数 *language* は **RFC 2231** の言語を指定しますが、デフォルトではこれは空文字列となります。*charset* と *language* はどちらも文字列である必要があります。

バージョン 2.2.2 で追加。

del_param (*param*[, *header*[, *requote*]])

指定されたパラメータを *Content-Type* ヘッダ中から完全にとりのぞきます。ヘッダはそのパラメータと値がない状態に書き換えられます。*requote* が `False` でない限り (デフォルトでは `True` です)、すべての値は必要に応じて `quote` されます。オプション変数 *header* が与えられた場合、*Content-Type* のかわりにそのヘッダが使用されます。

バージョン 2.2.2 で追加。

set_type (*type*[, *header*][, *requote*])

Content-Type ヘッダの *maintype* と *subtype* を設定します。 *type* は *maintype/subtype* という形の文字列でなければなりません。それ以外の場合は *ValueError* が発生します。

このメソッドは *Content-Type* ヘッダを置き換えますが、すべてのパラメータはそのままにします。 *requote* が *False* の場合、これはすでに存在するヘッダを quote せず放置しますが、そうでない場合は自動的に quote します (デフォルト動作)。

オプション変数 *header* が与えられた場合、 *Content-Type* のかわりにそのヘッダが使用されます。 *Content-Type* ヘッダが設定される場合には、 *MIME-Version* ヘッダも同時に付加されます。

バージョン 2.2.2 で追加。

get_filename ([*failobj*])

そのメッセージ中の *Content-Disposition* ヘッダにある、 *filename* パラメータの値を返します。目的のヘッダに *filename* パラメータがない場合には *Content-Type* ヘッダにある *name* パラメータを探します。それも無い場合またはヘッダが無い場合には *failobj* が返されます。返される文字列はつねに *email.utils.unquote()* によって *unquote* されます。

get_boundary ([*failobj*])

そのメッセージ中の *Content-Type* ヘッダにある、 *boundary* パラメータの値を返します。目的のヘッダが欠けていたり、 *boundary* パラメータがない場合には *failobj* が返されます。返される文字列はつねに *email.utils.unquote()* によって *unquote* されます。

set_boundary (*boundary*)

メッセージ中の *Content-Type* ヘッダにある、 *boundary* パラメータに値を設定します。 *set_boundary()* は必要に応じて *boundary* を quote します。そのメッセージが *Content-Type* ヘッダを含んでいない場合、 *HeaderParseError* が発生します。

注意: このメソッドを使うのは、古い *Content-Type* ヘッダを削除して新しい *boundary* をもったヘッダを *add_header()* で足すのとは少し違います。 *set_boundary()* は一連のヘッダ中の *Content-Type* ヘッダの位置を保つからです。しかし、これは元の *Content-Type* ヘッダ中に存在していた連続する行の順番までは保ちません。

get_content_charset ([*failobj*])

そのメッセージ中の *Content-Type* ヘッダにある、 *charset* パラメータの値を返します。値はすべて小文字に変換されます。メッセージ中に *Content-Type* がなかったり、このヘッダ中に *charset* パラメータがない場合には *failobj* が返されます。

注意: これは *get_charset()* メソッドとは異なります。こちらのほうは文字列のかわりに、そのメッセージボディのデフォルトエンコーディングの *Charset* インスタンスを返します。

バージョン 2.2.2 で追加。

get_charsets ([*failobj*])

メッセージ中に含まれる文字セットの名前をすべてリストにして返します。そのメッセージが *multipart* である場合、返されるリストの各要素がそれぞれの *subpart* のペイロードに対応します。それ以外の場合、これは長さ 1 のリストを返します。

リスト中の各要素は文字列であり、これは対応する subpart 中のそれぞれの *Content-Type* ヘッダにある charset の値です。しかし、その subpart が *Content-Type* をもってないか、charset がないか、あるいは MIME maintype が *text* でないいずれかの場合には、リストの要素として *failobj* が返されます。

walk()

walk() メソッドは多目的のジェネレータで、これはあるメッセージオブジェクトツリー中のすべての part および subpart をわたり歩くのに使えます。順序は深さ優先です。おそらく典型的な用法は、*walk()* を for ループ中でのイテレータとして使うことでしょう。ループを一回まわるとに、次の subpart が返されるのです。

以下の例は、multipart メッセージのすべての part において、その MIME タイプを表示していくものです。

```
>>> for part in msg.walk():
...     print part.get_content_type()
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
```

バージョン 2.5 で変更: 以前の非推奨メソッド *get_type()*、*get_main_type()*、*get_subtype()* は削除されました。

Message オブジェクトはオプションとして 2 つのインスタンス属性をとることができます。これはある MIME メッセージからプレーンテキストを生成するのに使うことができます。

preamble

MIME ドキュメントの形式では、ヘッダ直後にくる空行と最初の multipart 境界をあらわす文字列のあいだにいくつかのテキスト (訳注: preamble, 序文) を埋めこむことを許しています。このテキストは標準的な MIME の範疇からはみ出しているため、MIME 形式を認識するメールソフトからこれらは通常まったく見えません。しかしメッセージのテキストを生で見る場合、あるいはメッセージを MIME 対応していないメールソフトで見る場合、このテキストは目に見えることになります。

preamble 属性は MIME ドキュメントに加えるこの最初の MIME 範囲外テキストを含んでいます。*Parser* があるテキストをヘッダ以降に発見したが、それはまだ最初の MIME 境界文字列が現れる前だった場合、パーザはそのテキストをメッセージの *preamble* 属性に格納します。*Generator* がある MIME メッセージからプレーンテキスト形式を生成するときメッセージが *preamble* 属性を持つことが発見されれば、これはそのテキストをヘッダと最初の MIME 境界の間に挿入します。詳細は *email.parser* および *email.generator* を参照してください。

注意: そのメッセージに *preamble* がない場合、*preamble* 属性には *None* が格納されます。

epilogue

epilogue 属性はメッセージの最後の MIME 境界文字列からメッセージ末尾までのテキストを含むもので、それ以外は *preamble* 属性と同じです。

バージョン 2.5 で変更: *Generator* でファイル終端に改行を出力するため、*epilogue* に空文字列を設定する必要はなくなりました。

defects

defects 属性はメッセージを解析する途中で検出されたすべての問題点 (defect、障害) のリストを保持しています。解析中に発見される障害についてのより詳細な説明は *email.errors* を参照してください。

バージョン 2.4 で追加.

18.1.2 email.parser: 電子メールメッセージを解析 (パース) する

メッセージオブジェクト構造体をつくるには 2 つの方法があります。ひとつはまったくのスクラッチから *Message* を生成して、これを *attach()* と *set_payload()* 呼び出しを介してつなげていく方法で、もうひとつは電子メールメッセージのフラットなテキスト表現を解析 (parse、パース) する方法です。

email パッケージでは、MIME 文書をふくむ、ほとんどの電子メールの文書構造に対応できる標準的なパーザ (解析器) を提供しています。このパーザに文字列あるいはファイルオブジェクトを渡せば、パーザはそのオブジェクト構造の基底となる (root の) *Message* インスタンスを返します。簡単な非 MIME メッセージであれば、この基底オブジェクトのペイロードはたんにメッセージのテキストを格納する文字列になるでしょう。MIME メッセージであれば、基底オブジェクトはその *is_multipart()* メソッドに対して *True* を返します。そして、その各 subpart に *get_payload()* メソッドおよび *walk()* メソッドを介してアクセスすることができます。

実際には 2 つのパーザインターフェイスが使用可能です。ひとつは旧式の *Parser* API であり、もうひとつはインクリメンタルな *FeedParser* API です。旧式の *Parser* API はメッセージ全体のテキストが文字列としてすでにメモリ上にあるか、それがローカルなファイルシステム上に存在しているときには問題ありません。*FeedParser* はメッセージを読み込むときに、そのストリームが入力待ちのためにブロックされるような場合 (ソケットから email メッセージを読み込む時など) に、より有効です。*FeedParser* はインクリメンタルにメッセージを読み込み、解析します。パーザを close したときには根っこ (root) のオブジェクトのみが返されます^{*1}。

このパーザは、ある制限された方法で拡張できます。また、もちろん自分でご自分のパーザを完全に無から実装することもできます。*email* パッケージについているパーザと *Message* クラスの間に隠された秘密の関係はなにもありませんので、ご自分で実装されたパーザも、それが必要とするやりかたでメッセージオブジェクトツリーを作成することができます。

FeedParser API

バージョン 2.4 で追加.

email.feedparser モジュールからインポートされる *FeedParser* は email メッセージをインクリメンタルに解析するのに向いた API を提供します。これは email メッセージのテキストを (ソケットなどの) 読み込みがブロックされる可能性のある情報源から入力するときに必要となります。もちろん *FeedParser* は文

^{*1} Python 2.4 から導入された *email* パッケージバージョン 3.0 では、旧式の *Parser* は *FeedParser* によって書き直されました。そのためパーザの意味論と得られる結果は 2 つのパーザで同一のものになります。

字列またはファイルにすべて格納されている email メッセージを解析するのにも使うことができますが、このような場合には旧式の *Parser* API のほうが便利かもしれません。これら 2 つのパーザ API の意味論と得られる結果は同一です。

FeedParser API は簡単です。まずインスタンスをつくり、それにテキストを (それ以上テキストが必要なくなるまで) 流しこみます。その後パーザを *close* すると根っこ (root) のメッセージオブジェクトが返されます。標準に従ったメッセージを解析する場合、*FeedParser* は非常に正確であり、標準に従っていないメッセージでもちゃんと動きます。そのさい、これはメッセージがどのように壊れていると認識されたかについての情報を残します。これはメッセージオブジェクトの *defects* 属性にリストとして現れ、メッセージ中に発見された問題が記録されます。パーザが検出できる障害 (defect) については *email.errors* モジュールを参照してください。

以下は *FeedParser* の API です:

```
class email.parser.FeedParser ([_factory])
```

FeedParser インスタンスを作成します。オプション引数 *_factory* には引数なしの callable を指定し、これはつねに新しいメッセージオブジェクトの作成が必要になったときに呼び出されます。デフォルトでは、これは *email.message.Message* クラスになっています。

feed (*data*)

FeedParser にデータを供給します。 *data* は 1 行または複数行からなる文字列を渡します。渡される行は完結していなくてもよく、その場合 *FeedParser* は部分的な行を適切につなぎ合わせます。文字列中の各行は標準的な 3 種類の行末文字 (復帰 CR、改行 LF、または CR+LF) どれかの組み合わせでよく、これらが混在してもかまいません。

close ()

FeedParser を *close* し、それまでに渡されたすべてのデータの解析を完了させ根っこ (root) のメッセージオブジェクトを返します。 *FeedParser* を *close* したあとにさらにデータを *feed* した場合の挙動は未定義です。

Parser クラス API

email.parser モジュールからインポートされる *Parser* クラスは、メッセージを表すテキストが文字列またはファイルの形で完全に使用可能なときメッセージを解析するのに使われる API を提供します。 *email.Parser* モジュールはまた、 *HeaderParser* と呼ばれる 2 番目のクラスも提供しています。これはメッセージのヘッダのみを処理したい場合に使うことができ、ずっと高速な処理がおこなえます。なぜならこれはメッセージ本体を解析しようとはしないからです。かわりに、そのペイロードにはメッセージ本体の生の文字列が格納されます。 *HeaderParser* クラスは *Parser* クラスと同じ API をもっています。

```
class email.parser.Parser ([_class])
```

Parser クラスのコンストラクタです。オプション引数 *_class* をとることができます。これは呼び出し可能なオブジェクト (関数やクラス) でなければならず、メッセージ内コンポーネント (sub-message object) が作成されるときは常にそのファクトリクラスとして使用されます。デフォルトではこれは *Message* になっています (*email.message* 参照)。このファクトリクラスは引数なしで呼び出されます。

オプション引数 *strict* は無視されます。

バージョン 2.4 で非推奨: `Parser` は Python 2.4 で新しく導入された `FeedParser` の後方互換性のための API ラッパで、すべての解析が事実上 non-strict です。 `Parser` コンストラクタに `strict` フラグを渡す必要はありません。

バージョン 2.2.2 で変更: `strict` フラグが追加されました。

バージョン 2.4 で変更: `strict` フラグは推奨されなくなりました。

それ以外の `Parser` メソッドは以下のとおりです:

`parse` (`fp`[, `headersonly`])

ファイルなどストリーム形式 (file-like) のオブジェクト `fp` からすべてのデータを読み込み、得られたテキストを解析して基底 (root) メッセージオブジェクト構造を返します。 `fp` はストリーム形式のオブジェクトで `readline()` および `read()` 両方のメソッドをサポートしている必要があります。

`fp` に格納されているテキスト文字列は、一連の **RFC 2822** 形式のヘッダおよびヘッダ継続行 (header continuation lines) によって構成されている必要があります。オプションとして、最初にエンペローブヘッダが来ることもできます。ヘッダ部分はデータの終端か、ひとつの空行によって終了したとみなされます。ヘッダ部分に続くデータはメッセージ本体となります (MIME エンコードされた subpart を含んでいるかもしれません)。

オプション引数 `headersonly` はヘッダを読み終えた後にパースを止めるかを指定するフラグです。デフォルトは `False` で、ファイルの内容全体をパースします。

バージョン 2.2.2 で変更: `headersonly` フラグが追加されました。

`parsestr` (`text`[, `headersonly`])

`parse()` メソッドに似ていますが、ファイルなどのストリーム形式のかわりに文字列を引数としてとるところが違います。文字列に対してこのメソッドを呼ぶことは、`text` を `StringIO` インスタンスとして作成して `parse()` を適用するのと同じです。

オプション引数 `headersonly` は `parse()` メソッドと同じです。

バージョン 2.2.2 で変更: `headersonly` フラグが追加されました。

ファイルや文字列からメッセージオブジェクト構造を作成するのはかなりよくおこなわれる作業なので、便宜上次のような 2 つの関数が提供されています。これらは `email` パッケージのトップレベルの名前空間で使用できます。

`email.message_from_string` (`s`[, `_class`[, `strict`]])

文字列からメッセージオブジェクト構造を作成し返します。これは `Parser().parsestr(s)` とまったく同じです。オプション引数 `_class` および `strict` は `Parser` クラスのコンストラクタと同様に解釈されます。

バージョン 2.2.2 で変更: `strict` フラグが追加されました。

`email.message_from_file` (`fp`[, `_class`[, `strict`]])

Open されたファイルオブジェクトからメッセージオブジェクト構造を作成し返します。これは `Parser().parse(fp)` とまったく同じです。オプション引数 `_class` および `strict` は `Parser` クラスのコンストラクタと同様に解釈されます。

バージョン 2.2.2 で変更: *strict* フラグが追加されました。

対話的な Python プロンプトでこの関数を使用するとすれば、このようになります:

```
>>> import email
>>> msg = email.message_from_string(myString)
```

追記事項

以下はテキスト解析の際に適用されるいくつかの規約です:

- ほとんどの非 *multipart* 形式のメッセージは単一の文字列ペイロードをもつ単一のメッセージオブジェクトとして解析されます。このオブジェクトは *is_multipart()* に対して *False* を返します。このオブジェクトに対する *get_payload()* メソッドは文字列オブジェクトを返します。
- *multipart* 形式のメッセージはすべてメッセージ内コンポーネント (sub-message object) のリストとして解析されます。外側のコンテナメッセージオブジェクトは *is_multipart()* に対して *True* を返し、このオブジェクトに対する *get_payload()* メソッドは *Message* subpart のリストを返します。
- *message/** の Content-Type をもつほとんどのメッセージ (例: *message/delivery-status* や *message/rfc822* など) もコンテナメッセージオブジェクトとして解析されますが、ペイロードのリストの長さは 1 になります。このオブジェクトは *is_multipart()* メソッドに対して *True* を返し、リスト内にあるひとつだけの要素がメッセージ内のコンポーネントオブジェクトになります。
- いくつかの標準的でないメッセージは、*multipart* の使い方に統一がとれていない場合があります。このようなメッセージは *Content-Type* ヘッダに *multipart* を指定しているものの、その *is_multipart()* メソッドは *False* を返すことがあります。もしこのようなメッセージが *FeedParser* によって解析されると、その *defects* 属性のリスト中には *MultipartInvariantViolationDefect* クラスのインスタンスが現れます。詳しい情報については *email.errors* を参照してください。

脚注

18.1.3 email.generator: MIME 文書を生成する

よくある作業のひとつは、メッセージオブジェクト構造からフラットな電子メールテキストを生成することです。この作業は *smtplib* や *ntplib* モジュールを使ってメッセージを送信したり、メッセージをコンソールに出力したりするときに必要になります。あるメッセージオブジェクト構造をとってきて、そこからフラットなテキスト文書を生成するのは *Generator* クラスの仕事です。

繰り返しになりますが、*email.parser* モジュールと同じく、この機能はここでの定義済みジェネレータに制限されるわけではありません。これらはご自身でゼロから作りあげることもできます。しかしながら、定義済みのジェネレータはほとんどの電子メールを標準に沿ったやり方で生成する方法を知っていますし、MIME メッセージも非 MIME メッセージもとても上手く扱えます。さらにこれはフラットなテキストから *Parser* クラスを使ってメッセージ構造に変換し、それをまたフラットなテキストに戻しても、結果が冪等 (入力が出

力と同じになる)^{*1} になるよう設計されています。一方で、プログラムによって構成された `Message` のジェネレータを使う場合、デフォルトの挿入によって `Message` オブジェクトを変えてしまうかもしれません。

`email.generator` モジュールからインポートされる `Generator` クラスで公開されているメソッドには、以下のようなものがあります:

```
class email.generator.Generator (outfp[, mangle_from_[, maxheaderlen ]])
```

`Generator` クラスのコンストラクタは `outfp` と呼ばれるストリーム形式 (file-like) のオブジェクトひとつを引数にとります。 `outfp` は `write()` メソッドをサポートし、Python 拡張 print 文の出力ファイルとして使えるようになっている必要があります。

オプション引数 `mangle_from_` はフラグで、`True` のときはメッセージ本体の行で厳密に `From` で始まるもの、つまり行の先頭が `From` でその後に空白が続く行の前に `>` という文字を挿入します。これは、このような行が Unix の mailbox 形式のエンベロープヘッダ区切り文字列として誤認識されるのを防ぐための、移植性ある唯一の方法です (詳しくは [WHY THE CONTENT-LENGTH FORMAT IS BAD \(なぜ Content-Length 形式が有害か\)](#) を参照してください)。デフォルトでは `mangle_from_` は `True` になっていますが、Unix の mailbox 形式ファイルに出力しないのならばこれは `False` に設定してもかまいません。

オプション引数 `maxheaderlen` は連続していないヘッダの最大長を指定します。ひとつのヘッダ行が `maxheaderlen` (これは文字数です、tab は空白 8 文字に展開されます) よりも長い場合、ヘッダは `Header` クラスで定義されているように途中で折り返され、間にはセミコロンが挿入されます。もしセミコロンが見つからない場合、そのヘッダは放置されます。ヘッダの折り返しを禁止するにはこの値にゼロを指定してください。デフォルトは 78 文字で、[RFC 2822](#) で推奨されている (ですが強制ではありません) 値です。

これ以外のパブリックな `Generator` メソッドは以下のとおりです:

```
flatten (msg[, unixfrom ])
```

`msg` を基点とするメッセージオブジェクト構造体の文字表現を出力します。出力先のファイルにはこの `Generator` インスタンスが作成されたときに指定されたものが使われます。各 subpart は深さ優先順序 (depth-first) で出力され、得られるテキストは適切に MIME エンコードされたものになっています。

オプション引数 `unixfrom` は、基点となるメッセージオブジェクトの最初の [RFC 2822](#) ヘッダが現れる前に、エンベロープヘッダ区切り文字列を出力することを強制するフラグです。そのメッセージオブジェクトがエンベロープヘッダをもたない場合、標準的なエンベロープヘッダが自動的に作成されます。デフォルトではこの値は `False` に設定されており、エンベロープヘッダ区切り文字列は出力されません。

注意: 各 subpart に関しては、エンベロープヘッダは出力されません。

バージョン 2.2.2 で追加.

```
clone (fp)
```

この `Generator` インスタンスの独立したクローンを生成し返します。オプションはすべて同一

^{*1} ここではあなたが `unixfrom` 引数に適切な設定を使い、`maxheaderlen=0` (入力行がどんな長さでも元のものを持続します) をセットしていることを前提にしています。これでもなお厳密には正しくありません。多くの場合ヘッダの連続する空白が単一の空白に置き換えられるからです。これはいずれは修正されるべきバグです。

になっています。

バージョン 2.2.2 で追加.

write(*s*)

文字列 *s* を既定のファイルに出力します。ここでいう出力先は *Generator* コンストラクタに渡した *outfp* のことをさします。この関数はただ単に拡張 `print` 文で使われる *Generator* インスタンスに対してファイル操作風の API を提供するためだけのものです。

ユーザの便宜をはかるため、メソッド `Message.as_string()` と `str(aMessage)` (つまり `Message.__str__()` のことです) をつかえばメッセージオブジェクトを特定の書式でフォーマットされた文字列に簡単に変換することができます。詳細は `email.message` を参照してください。

`email.generator` モジュールはひとつの派生クラスも提供しています。これは *DecodedGenerator* と呼ばれるもので、*Generator* 基底クラスと似ていますが、非 *text* 型の subpart を特定の書式でフォーマットされた表現形式で置きかえるところが違います。

class `email.generator.DecodedGenerator` (*outfp*[, *mangle_from_*[, *maxheaderlen*[, *fmt*]]])

このクラスは *Generator* から派生したもので、メッセージの subpart をすべて渡り歩きます。subpart の主形式が *text* だった場合、これはその subpart のペイロードをデコードして出力します。オプション引数 *mangle_from_* および *maxheaderlen* の意味は基底クラス *Generator* のそれと同じです。

Subpart の主形式が *text* ではない場合、オプション引数 *fmt* がそのメッセージペイロードのかわりのフォーマット文字列として使われます。*fmt* は `%(keyword)s` のような形式を展開し、以下のキーワードを認識します:

- *type* – 非 *text* 型 subpart の MIME 形式
- *maintype* – 非 *text* 型 subpart の MIME 主形式 (maintype)
- *subtype* – 非 *text* 型 subpart の MIME 副形式 (subtype)
- *filename* – 非 *text* 型 subpart のファイル名
- *description* – 非 *text* 型 subpart につけられた説明文字列
- *encoding* – 非 *text* 型 subpart の Content-transfer-encoding

fmt のデフォルト値は `None` です。こうすると以下の形式で出力します

`[Non-text %(type)s part of message omitted, filename %(filename)s]`

バージョン 2.2.2 で追加.

バージョン 2.5 で変更: 以前の非推奨メソッド `__call__()` は削除されました。

18.1.4 `email.mime`: ゼロからのメールと MIME オブジェクトの作成

ふつう、メッセージオブジェクト構造はファイルまたは何がしかのテキストをパーザに通すことで得られます。パーザは与えられたテキストを解析し、基底となる *root* のメッセージオブジェクトを返します。しかし、

完全なメッセージオブジェクト構造を何も無いところから作成することもまた可能です。個別の *Message* を手で作成することさえできます。実際には、すでに存在するメッセージオブジェクト構造をとってきて、そこに新たな *Message* オブジェクトを追加したり、あるものを別のところへ移動させたりできます。これは MIME メッセージを切ったりおろしたりするために非常に便利なインターフェイスを提供します。

新しいメッセージオブジェクト構造は *Message* インスタンスを作成することにより作れます。ここに添付ファイルやその他適切なものをすべて手で加えてやればよいのです。MIME メッセージの場合、*email* パッケージはこれらを簡単におこなえるようにするためにいくつかの便利なサブクラスを提供しています。

以下がそのサブクラスです:

```
class email.mime.base.MIMEBase(_maintype, _subtype, **_params)
```

モジュール: *email.mime.base*

これはすべての *Message* の MIME 用サブクラスの基底となるクラスです。とくに *MIMEBase* のインスタンスを直接作成することは (可能ではありますが) ふつうはしないでしょう。 *MIMEBase* は単により特化された MIME 用サブクラスのための便宜的な基底クラスとして提供されています。

_maintype は *Content-Type* の主形式 (maintype) であり (*text* や *image* など)、 *_subtype* は *Content-Type* の副形式 (subtype) です (*plain* や *gif* など)。 *_params* は各パラメータのキーと値を格納した辞書であり、これは直接 *Message.add_header* に渡されます。

MIMEBase クラスはつねに (*_maintype*、 *_subtype*、 および *_params* にもとづいた) *Content-Type* ヘッダと、 *MIME-Version* ヘッダ (必ず 1.0 に設定される) を追加します。

```
class email.mime.nonmultipart.MIMENonMultipart
```

モジュール: *email.mime.nonmultipart*

MIMEBase のサブクラスで、これは *multipart* 形式でない MIME メッセージのための中間的な基底クラスです。このクラスのおもな目的は、通常 *multipart* 形式のメッセージに対してのみ意味をなす *attach()* メソッドの使用をふせぐことです。もし *attach()* メソッドが呼ばれた場合、これは *MultipartConversionError* 例外を発生します。

バージョン 2.2.2 で追加。

```
class email.mime.multipart.MIMEMultipart([_subtype[, boundary[, _subparts[, _params]]]])
```

モジュール: *email.mime.multipart*

MIMEBase のサブクラスで、これは *multipart* 形式の MIME メッセージのための中間的な基底クラスです。オプション引数 *_subtype* はデフォルトでは *mixed* になっていますが、そのメッセージの副形式 (subtype) を指定するのに使うことができます。メッセージオブジェクトには *multipart/_subtype* という値をもつ *Content-Type* ヘッダとともに、 *MIME-Version* ヘッダが追加されるでしょう。

オプション引数 *boundary* は *multipart* の境界文字列です。これが *None* の場合 (デフォルト)、境界は必要に応じて計算されます (例えばメッセージがシリアライズされるときなど)。

_subparts はそのペイロードの subpart の初期値からなるシーケンスです。このシーケンスはリストに変換できるようになっている必要があります。新しい subpart はつねに *Message.attach* メソッドを

使ってそのメッセージに追加できるようになっています。

Content-Type ヘッダに対する追加のパラメータはキーワード引数 *_params* を介して取得あるいは設定されます。これはキーワード辞書になっています。

バージョン 2.2.2 で追加.

```
class email.mime.application.MIMEApplication(_data[, _subtype[, _encoder[,  
                                              **_params]])
```

モジュール: `email.mime.application`

MIMENonMultipart のサブクラスである *MIMEApplication* クラスは MIME メッセージオブジェクトのメジャータイプ *application* を表します。 *_data* は生のバイト列が入った文字列です。オプション引数 *_subtype* は MIME のサブタイプを設定します。サブタイプのデフォルトは *octet-stream* です。

オプション引数の *_encoder* は呼び出し可能なオブジェクト (関数など) で、データの転送に使う実際のエンコード処理を行います。この呼び出し可能なオブジェクトは引数を 1 つ取り、それは *MIMEApplication* のインスタンスです。ペイロードをエンコードされた形式に変更するために *get_payload()* と *set_payload()* を使い、必要に応じて *Content-Transfer-Encoding* やその他のヘッダをメッセージオブジェクトに追加するべきです。デフォルトのエンコードは *base64* です。組み込みのエンコーダの一覧は *email.encoders* モジュールを見てください。

_params は基底クラスのコンストラクタにそのまま渡されます。

バージョン 2.5 で追加.

```
class email.mime.audio.MIMEAudio(_audiodata[, _subtype[, _encoder[, **_params]])
```

モジュール: `email.mime.audio`

MIMEAudio クラスは *MIMENonMultipart* のサブクラスで、主形式 (maintype) が *audio* の MIME オブジェクトを作成するのに使われます。 *_audiodata* は実際の音声データを格納した文字列です。もしこのデータが標準の Python モジュール *sndhdr* によって認識できるものであれば、*Content-Type* ヘッダの副形式 (subtype) は自動的に決定されます。そうでない場合はその画像の形式 (subtype) を *_subtype* で明示的に指定する必要があります。副形式が自動的に決定できず、*_subtype* の指定もない場合は、*TypeError* が発生します。

オプション引数の *_encoder* は呼び出し可能なオブジェクト (関数など) で、オーディオデータの転送に使う実際のエンコード処理を行います。この呼び出し可能なオブジェクトは引数を 1 つ取り、それは *MIMEAudio* のインスタンスです。ペイロードをエンコードされた形式に変更するために *get_payload()* と *set_payload()* を使い、必要に応じて *Content-Transfer-Encoding* やその他のヘッダをメッセージオブジェクトに追加するべきです。デフォルトのエンコードは *base64* です。組み込みのエンコーダの一覧は *email.encoders* モジュールを見てください。

_params は基底クラスのコンストラクタにそのまま渡されます。

```
class email.mime.image.MIMEImage(_imagedata[, _subtype[, _encoder[, **_params]])
```

モジュール: `email.mime.image`

MIMEImage クラスは *MIMENonMultipart* のサブクラスで、主形式 (maintype) が *image* の MIME

オブジェクトを作成するのに使われます。 `_imagedata` は実際の画像データを格納した文字列です。もしこのデータが標準の Python モジュール `imghdr` によって認識できるものであれば、 `Content-Type` ヘッダの副形式 (subtype) は自動的に決定されます。そうでない場合はその画像の形式 (subtype) を `_subtype` で明示的に指定する必要があります。副形式が自動的に決定できず、 `_subtype` の指定もない場合は、 `TypeError` が発生します。

オプション引数の `_encoder` は呼び出し可能なオブジェクト (関数など) で、画像データの転送に使う実際のエンコード処理を行います。この呼び出し可能なオブジェクトは引数を 1 つ取り、それは `MIMEImage` のインスタンスです。ペイロードをエンコードされた形式に変更するために `get_payload()` と `set_payload()` を使い、必要に応じて `Content-Transfer-Encoding` やその他のヘッダをメッセージオブジェクトに追加するべきです。デフォルトのエンコードは base64 です。組み込みのエンコーダの一覧は `email.encoders` モジュールを見てください。

`_params` は `MIMEBase` コンストラクタに直接渡されます。

```
class email.mime.message.MIMEMessage(_msg[, _subtype])
モジュール: email.mime.message
```

`MIMEMessage` クラスは `MIMENonMultipart` のサブクラスで、主形式 (maintype) が `message` の MIME オブジェクトを作成するのに使われます。ペイロードとして使われるメッセージは `_msg` になります。これは `Message` クラス (あるいはそのサブクラス) のインスタンスでなければいけません。そうでない場合、この関数は `TypeError` を発生します。

オプション引数 `_subtype` はそのメッセージの副形式 (subtype) を設定します。デフォルトではこれは `rfc822` になっています。

```
class email.mime.text.MIMEText(_text[, _subtype[, _charset]])
モジュール: email.mime.text
```

`MIMEText` クラスは `MIMENonMultipart` のサブクラスで、主形式 (maintype) が `text` の MIME オブジェクトを作成するのに使われます。ペイロードの文字列は `_text` になります。 `_subtype` には副形式 (subtype) を指定し、デフォルトは `plain` です。 `_charset` はテキストの文字セットで、 `MIMENonMultipart` コンストラクタに引数として渡されます。デフォルトではこの値は `us-ascii` になっています。 `_text` が unicode の場合には `_charset` の `output_charset` でエンコードされ、それ以外の場合にはそのまま使われます。

バージョン 2.4 で変更: 以前、推奨されない引数であった `_encoding` は削除されました。 `_charset` 引数を基にして Content Transfer Encodnig が暗黙に決定されます。

Unless the `_charset` parameter is explicitly set to `None`, the `MIMEText` object created will have both a `Content-Type` header with a `charset` parameter, and a `Content-Transfer-Encoding` header. This means that a subsequent `set_payload` call will not result in an encoded payload, even if a charset is passed in the `set_payload` command. You can "reset" this behavior by deleting the `Content-Transfer-Encoding` header, after which a `set_payload` call will automatically encode the new payload (and add a new `Content-Transfer-Encoding` header).

18.1.5 email.header: 国際化されたヘッダ

RFC 2822 は電子メールメッセージの形式を規定する基本規格です。これはほとんどの電子メールが ASCII 文字のみで構成されていたころ普及した **RFC 822** 標準から発展したものです。 **RFC 2822** は電子メールがすべて 7-bit ASCII 文字のみから構成されていると仮定して作られた仕様です。

もちろん、電子メールが世界的に普及するにつれ、この仕様は国際化されてきました。今では電子メールに言語依存の文字セットを使うことができます。基本規格では、まだ電子メールメッセージを 7-bit ASCII 文字のみを使って転送するよう要求していますので、多くの RFC でどうやって非 ASCII の電子メールを **RFC 2822** 準拠な形式にエンコードするかが記述されています。これらの RFC は以下のものを含みます: **RFC 2045**、**RFC 2046**、**RFC 2047**、および **RFC 2231**。 `email` パッケージは、`email.header` および `email.charset` モジュールでこれらの規格をサポートしています。

ご自分の電子メールヘッダ、たとえば *Subject* や *To* などのフィールドに非 ASCII 文字を入れたい場合、`Header` クラスを使う必要があります。 `Message` オブジェクトの該当フィールドに文字列ではなく、`Header` インスタンスを使うのです。 `Header` クラスは `email.header` モジュールからインポートしてください。たとえば:

```
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('p\xfb6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> print msg.as_string()
Subject: =?iso-8859-1?q?p=F6stal?=
```

Subject フィールドに非 ASCII 文字をふくめていることに注目してください。ここでは、含めたいバイト列がエンコードされている文字セットを指定して `Header` インスタンスを作成することによって実現しています。のちにこの `Message` インスタンスからフラットなテキストを生成するさいに、この *Subject* フィールドは **RFC 2047** 準拠の適切な形式にエンコードされます。MIME 機能のついているメーラなら、このヘッダに埋めこまれた ISO-8859-1 文字をただしく表示するでしょう。

バージョン 2.2.2 で追加。

以下は `Header` クラスの説明です:

```
class email.header.Header([s[, charset[, maxlinelen[, header_name[, continuation_ws[, errors]]]]])
```

別の文字セットの文字列をふくむ MIME 準拠なヘッダを作成します。

オプション引数 *s* はヘッダの値の初期値です。これが `None` の場合 (デフォルト)、ヘッダの初期値は設定されません。この値はあとから `append()` メソッドを呼び出すことによって追加することができます。 *s* はバイト文字列か、あるいは Unicode 文字列でもかまいませんが、この意味については `append()` の項を参照してください。

オプション引数 *charset* には 2 つの目的があります。ひとつは `append()` メソッドにおける *charset* 引数と同じものです。もうひとつの目的は、これ以降 *charset* 引数を省略した `append()` メソッド呼び出しすべてにおける、デフォルト文字セットを決定するものです。コンストラクタに *charset* が与えられない場合 (デフォルト)、初期値の *s* および以後の `append()` 呼び出しにおける文字セットとして

`us-ascii` が使われます。

行の最大長は `maxlinelen` によって明示的に指定できます。最初の行を (`Subject` などの `s` に含まれないフィールドヘッダの責任をとるため) 短く切りとる場合、`header_name` にそのフィールド名を指定してください。`maxlinelen` のデフォルト値は 76 であり、`header_name` のデフォルト値は `None` です。これはその最初の行を長い、切りとられたヘッダとして扱わないことを意味します。

オプション引数 `continuation_ws` は [RFC 2822](#) 準拠の折り返し用余白文字で、ふつうこれは空白か、ハードタブ文字 (hard tab) である必要があります。ここで指定された文字は複数にわたる行の行頭に挿入されます。`continuation_ws` のデフォルト値は 1 つのスペース文字です (" ")。

オプション引数 `errors` は、`append()` メソッドにそのまま渡されます。

`append(s[, charset[, errors]])`

この MIME ヘッダに文字列 `s` を追加します。

オプション引数 `charset` がもし与えられた場合、これは `Charset` インスタンス (`email.charset` を参照) か、あるいは文字セットの名前でなければなりません。この場合は `Charset` インスタンスに変換されます。この値が `None` の場合 (デフォルト)、コンストラクタで与えられた `charset` が使われます。

`s` はバイト文字列か、Unicode 文字列です。これがバイト文字列 (`isinstance(s, str)` が真) の場合、`charset` はその文字列のエンコーディングであり、これが与えられた文字セットでうまくデコードできないときは `UnicodeError` が発生します。

一方 `s` が Unicode 文字列の場合、`charset` はその文字列の文字セットを決定するためのヒントとして使われます。この場合、[RFC 2822](#) 準拠のヘッダを [RFC 2047](#) の規則を用いて生成する際、Unicode 文字列は以下の文字セットを (この優先順位で) 適用してエンコードされます: `us-ascii`、`charset` で与えられたヒント、それもなければ `utf-8`。最初の文字セットは `UnicodeError` をなるべく防ぐために使われます。

オプション引数 `errors` は `unicode()` または `unicode.encode()` の呼び出し時に使用し、デフォルト値は "strict" です。

`encode([splitchars])`

メッセージヘッダを RFC に沿ったやり方でエンコードします。おそらく長い行は折り返され、非 ASCII 部分は base64 または quoted-printable エンコーディングで包含されるでしょう。オプション引数 `splitchars` には長い ASCII 行を分割する区切り文字群を文字列として指定し、[RFC 2822](#) の *highest level syntactic breaks* の大まかなサポートの為に使用します。この引数は [RFC 2047](#) でエンコードされた行には影響しません。

`Header` クラスは、標準の演算子や組み込み関数をサポートするためのメソッドもいくつか提供しています。

`__str__()`

`Header.encode()` と同じです。 `str(aHeader)` などとすると有用でしょう。

`__unicode__()`

組み込みの `unicode()` 関数の補助です。ヘッダを Unicode 文字列として返します。

`__eq__(other)`

このメソッドは、ふたつの `Header` インスタンスどうしが等しいかどうか判定するのに使えます。

`__ne__(other)`

このメソッドは、ふたつの `Header` インスタンスどうしが異なっているかどうかを判定するのに使えます。

さらに、`email.header` モジュールは以下のような便宜的な関数も提供しています。

`email.header.decode_header(header)`

文字セットを変換することなしに、メッセージのヘッダをデコードします。ヘッダの値は `header` に渡します。

この関数はヘッダのそれぞれのデコードされた部分ごとに、`(decoded_string, charset)` という形式の 2 要素タプルからなるリストを返します。`charset` はヘッダのエンコードされていない部分に対しては `None` を、それ以外の場合はエンコードされた文字列が指定している文字セットの名前を小文字からなる文字列で返します。

以下はこの使用例です:

```
>>> from email.header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[('p\xfb6stal', 'iso-8859-1')]
```

`email.header.make_header(decoded_seq[, maxlinelen[, header_name[, continuation_ws]]])`

`decode_header()` によって返される 2 要素タプルのリストから `Header` インスタンスを作成します。

`decode_header()` はヘッダの値をとってきて、`(decoded_string, charset)` という形式の 2 要素タプルからなるリストを返します。ここで `decoded_string` はデコードされた文字列、`charset` はその文字セットです。

この関数はこれらのリストの項目から、`Header` インスタンスを返します。オプション引数 `maxlinelen`、`header_name` および `continuation_ws` は `Header` コンストラクタに与えるものと同じです。

18.1.6 email.charset: 文字集合の表現

このモジュールは文字セットを表現する `Charset` クラスと電子メールメッセージにふくまれる文字セット間の変換、および文字セットのレジストリとこのレジストリを操作するためのいくつかの便宜的なメソッドを提供します。`Charset` インスタンスは `email` パッケージ中にあるほかのいくつかのモジュールで使用されます。

このクラスは `email.charset` モジュールから import してください。

バージョン 2.2.2 で追加.

```
class email.charset.Charset([input_charset])
```

文字セットを email のプロパティに写像する。

このクラスはある特定の文字セットに対し、電子メールに課される制約の情報を提供します。また、与えられた適用可能な codec をつかって、文字セット間の変換をおこなう便宜的なルーチンも提供します。またこれは、ある文字セットが与えられたときに、その文字セットを電子メールメッセージのなかでどうやって RFC に準拠したやり方で使用するかに関する、できうるかぎりの情報も提供します。

文字セットによっては、それらの文字を電子メールのヘッダあるいはメッセージ本体で使う場合は quoted-printable 形式あるいは base64 形式でエンコードする必要があります。またある文字セットはむきだしのまま変換する必要があり、電子メールの中では使用できません。

以下ではオプション引数 *input_charset* について説明します。この値はつねに小文字に強制的に変換されます。そして文字セットの別名が正規化されたあと、この値は文字セットのレジストリ内を検索し、ヘッダのエンコーディングとメッセージ本体のエンコーディング、および出力時の変換に使われる codec をみつけるのに使われます。たとえば *input_charset* が iso-8859-1 の場合、ヘッダおよびメッセージ本体は quoted-printable でエンコードされ、出力時の変換用 codec は必要ありません。もし *input_charset* が euc-jp ならば、ヘッダは base64 でエンコードされ、メッセージ本体はエンコードされませんが、出力されるテキストは euc-jp 文字セットから iso-2022-jp 文字セットに変換されます。

Charset インスタンスは以下のようなデータ属性をもっています:

input_charset

最初に指定される文字セットです。一般に通用している別名は、正式な 電子メール用の名前に変換されます (たとえば、latin-1 は iso-8859-1 に変換されます)。デフォルトは 7-bit の us-ascii です。

header_encoding

この文字セットが電子メールヘッダに使われる前にエンコードされる必要がある場合、この属性は *Charset.QP* (quoted-printable エンコーディング)、*Charset.BASE64* (base64 エンコーディング)、あるいは最短の QP または BASE64 エンコーディングである *Charset.SHORTEST* に設定されます。そうでない場合、この値は *None* になります。

body_encoding

header_encoding と同じですが、この値はメッセージ本体のためのエンコーディングを記述します。これはヘッダ用のエンコーディングとは違うかもしれません。*body_encoding* では、*Charset.SHORTEST* を使うことはできません。

output_charset

文字セットによっては、電子メールのヘッダあるいはメッセージ本体に使う前にそれを変換する必要があります。もし *input_charset* がそれらの文字セットのどれかをさしていたら、この *output_charset* 属性はそれが出力時に変換される文字セットの名前をあらわしています。それ以外の場合、この値は *None* になります。

input_codec

input_charset を Unicode に変換するための Python 用 codec 名です。変換用の codec が必要ないときは、この値は *None* になります。

output_codec

Unicode を *output_charset* に変換するための Python 用 codec 名です。変換用の codec が必要ない

ときは、この値は `None` になります。この属性は `input_codec` と同じ値をもつことになるでしょう。

`Charset` インスタンスは、以下のメソッドも持っています:

get_body_encoding()

メッセージ本体のエンコードに使われる `content-transfer-encoding` の値を返します。

この値は使用しているエンコーディングの文字列 `quoted-printable` または `base64` か、あるいは関数のどちらかです。後者の場合、これはエンコードされる `Message` オブジェクトを単一の引数として取るような関数である必要があります。この関数は変換後 `Content-Transfer-Encoding` ヘッダ自体を、なんであれ適切な値に設定する必要があります。

このメソッドは `body_encoding` が `QP` の場合 `quoted-printable` を返し、`body_encoding` が `BASE64` の場合 `base64` を返します。それ以外の場合は文字列 `7bit` を返します。

convert(s)

文字列 `s` を `input_codec` から `output_codec` に変換します。

toSplittable(s)

マルチバイトの可能性のある文字列を、安全に `split` できる形式に変換します。`s` には `split` する文字列を渡します。

これは `input_codec` を使って文字列を `Unicode` にすることで、文字と文字の境界で (たとえそれがマルチバイト文字であっても) 安全に `split` できるようにします。

`input_charset` の文字列 `s` をどうやって `Unicode` に変換すればいいかが不明な場合、このメソッドは与えられた文字列そのものを返します。

`Unicode` に変換できなかった文字は、`Unicode` 置換文字 (`Unicode replacement character`) `'U+FFFD'` に置換されます。

fromSplittable(ustr[, to_output])

`split` できる文字列をエンコードされた文字列に変換しなおします。`ustr` は "unsplit" するための `Unicode` 文字列です。

このメソッドでは、文字列を `Unicode` からエンコード形式に戻すために適切な `codec` を使用します。与えられた文字列が `Unicode` ではなかった場合、あるいはそれをどうやって `Unicode` から変換するか不明だった場合は、与えられた文字列そのものが返されます。

`Unicode` から正しく変換できなかった文字については、適当な文字 (通常は `'?'`) に置き換えられます。

`to_output` が `True` の場合 (デフォルト)、このメソッドは `output_codec` をエンコードの形式として使用します。`to_output` が `False` の場合、これは `input_codec` を使用します。

get_output_charset()

出力用の文字セットを返します。

これは `output_charset` 属性が `None` でなければその値になります。それ以外の場合、この値は `input_charset` と同じです。

encoded_header_len()

エンコードされたヘッダ文字列の長さを返します。これは quoted-printable エンコーディングあるいは base64 エンコーディングに対しても正しく計算されます。

header_encode(s[, convert])

文字列 *s* をヘッダ用にエンコードします。

convert が `True` の場合、文字列は入力用文字セットから出力用文字セットに自動的に変換されます。これは行の長さ問題のあるマルチバイトの文字セットに対しては役に立ちません (マルチバイト文字はバイト境界ではなく、文字ごとの境界で `split` する必要があります)。これらの問題を扱うには、高水準のクラスである `Header` クラスを使ってください (`email.header` を参照)。*convert* の値はデフォルトでは `False` です。

エンコーディングの形式 (base64 または quoted-printable) は、*header_encoding* 属性に基づきます。

body_encode(s[, convert])

文字列 *s* をメッセージ本体用にエンコードします。

convert が `True` の場合 (デフォルト)、文字列は入力用文字セットから出力用文字セットに自動的に変換されます。`header_encode()` とは異なり、メッセージ本体にはふつうバイト境界の問題やマルチバイト文字セットの問題がないので、これはきわめて安全におこなえます。

エンコーディングの形式 (base64 または quoted-printable) は、*body_encoding* 属性に基づきます。

`Charset` クラスには、標準的な演算と組み込み関数をサポートするいくつかのメソッドがあります。

__str__()

input_charset を小文字に変換された文字列型として返します。`__repr__()` は、`__str__()` の別名となっています。

__eq__(other)

このメソッドは、2 つの `Charset` インスタンスが同じかどうかをチェックするのに使います。

__ne__(other)

このメソッドは、2 つの `Charset` インスタンスが異なるかどうかをチェックするのに使います。

また、`email.charset` モジュールには、グローバルな文字セット、文字セットの別名 (エイリアス) および codec 用のレジストリに新しいエントリを追加する以下の関数もふくまれています:

`email.charset.add_charset(charset[, header_enc[, body_enc[, output_charset]])`

文字の属性をグローバルなレジストリに追加します。

charset は入力用の文字セットで、その文字セットの正式名称を指定する必要があります。

オプション引数 *header_enc* および *body_enc* は quoted-printable エンコーディングをあらわす `Charset.QP` か、base64 エンコーディングをあらわす `Charset.BASE64`、最短の quoted-printable または base64 エンコーディングをあらわす `Charset.SHORTEST`、あるいはエンコーディングなしの `None` のどれかになります。`SHORTEST` が使えるのは *header_enc* だけです。デフォルトの値はエンコーディングなしの `None` になっています。

オプション引数 `output_charset` には出力用の文字セットが入ります。 `Charset.convert()` が呼ばれたときの変換はまず入力用の文字セットを Unicode に変換し、それから出力用の文字セットに変換されます。デフォルトでは、出力は入力と同じ文字セットになっています。

`input_charset` および `output_charset` はこのモジュール中の文字セット-codec 対応表にある Unicode codec エントリである必要があります。モジュールがまだ対応していない codec を追加するには、 `add_codec()` を使ってください。より詳しい情報については `codecs` モジュールの文書を参照してください。

グローバルな文字セット用のレジストリは、モジュールの global 辞書 `CHARSETS` 内に保持されています。

`email.charset.add_alias(alias, canonical)`

文字セットの別名 (エイリアス) を追加します。 `alias` はその別名で、たとえば `latin-1` のように指定します。 `canonical` はその文字セットの正式名称で、たとえば `iso-8859-1` のように指定します。

文字セットのグローバルな別名用レジストリは、モジュールの global 辞書 `ALIASES` 内に保持されています。

`email.charset.add_codec(charset, codecname)`

与えられた文字セットの文字と Unicode との変換をおこなう codec を追加します。

`charset` はある文字セットの正式名称、 `codecname` は Python 用 codec の名前です。後者は組み込み関数 `unicode()` の第 2 引数か、あるいは Unicode 文字列型の `encode()` メソッドに適した形式になっていなければなりません。

18.1.7 email.encoders: エンコーダ

何もないところから `Message` を作成するときしばしば必要になるのが、ペイロードをメールサーバに通すためにエンコードすることです。これはとくにバイナリデータを含んだ `image/*` や `text/*` タイプのメッセージで必要です。

`email` パッケージでは、 `encoders` モジュールにおいていくかの便宜的なエンコーディングをサポートしています。実際にはこれらのエンコーダは `MIMEAudio` および `MIMEImage` クラスのコンストラクタでデフォルトエンコーダとして使われています。すべてのエンコーディング関数は、エンコードするメッセージオブジェクトひとつだけを引数にとります。これらはふつうペイロードを取りだし、それをエンコードして、ペイロードをエンコードされたものにセットしなおします。これらはまた `Content-Transfer-Encoding` ヘッダを適切な値に設定します。

マルチパートメッセージにこれら関数を使うことは全く無意味です。それらは各々のサブパートごとに適用されるべきものです。メッセージがマルチパートのものを渡すと `TypeError` が発生します。

提供されているエンコーディング関数は以下のとおりです:

`email.encoders.encode_quopri(msg)`

ペイロードを quoted-printable 形式にエンコードし、 `Content-Transfer-Encoding` ヘッダを

`quoted-printable`^{*1} に設定します。これはそのペイロードのほとんどが通常の印刷可能な文字からなっているが、印刷不可能な文字がすこしだけあるときのエンコード方法として適しています。

`email.encoders.encode_base64(msg)`

ペイロードを base64 形式でエンコードし、*Content-Transfer-Encoding* ヘッダを base64 に変更します。これはペイロード中のデータのほとんどが印刷不可能な文字である場合に適しています。quoted-printable 形式よりも結果としてはコンパクトなサイズになるからです。base64 形式の欠点は、これが人間にはまったく読めないテキストになってしまうことです。

`email.encoders.encode_7or8bit(msg)`

これは実際にはペイロードを変更はしませんが、ペイロードの形式に応じて *Content-Transfer-Encoding* ヘッダを 7bit あるいは 8bit に適した形に設定します。

`email.encoders.encode_noop(msg)`

これは何もしないエンコーダです。*Content-Transfer-Encoding* ヘッダを設定さえしません。

18.1.8 email.errors: 例外及び欠陥クラス

`email.errors` モジュールでは、以下の例外クラスが定義されています:

exception `email.errors.MessageError`

これは `email` パッケージが発生しうるすべての例外の基底クラスです。これは標準の *Exception* クラスから派生しており、追加のメソッドはまったく定義されていません。

exception `email.errors.MessageParseError`

これは *Parser* クラスが発生しうる例外の基底クラスです。*MessageError* から派生しています。

exception `email.errors.HeaderParseError`

メッセージの **RFC 2822** ヘッダを解析している途中にある条件でエラーがおこると発生します。これは *MessageParseError* から派生しています。この例外が起こる可能性があるのは *Parser.parse* メソッドまたは *Parser.parsestr* メソッドです。

この例外が発生するのはメッセージ中で最初の **RFC 2822** ヘッダが現れたあとにエンベロープヘッダが見つかったとか、最初の **RFC 2822** ヘッダが現れる前に前のヘッダからの継続行が見つかったとかいう状況を含みます。あるいはヘッダでも継続行でもない行がヘッダ中に見つかった場合でもこの例外が発生します。

exception `email.errors.BoundaryError`

メッセージの **RFC 2822** ヘッダを解析している途中にある条件でエラーがおこると発生します。これは *MessageParseError* から派生しています。この例外が起こる可能性があるのは *Parser.parse* メソッドまたは *Parser.parsestr* メソッドです。

この例外が発生するのは、厳格なパーズ方式が用いられているときに、*multipart/** 形式の開始あるいは終了の文字列が見つからなかった場合などです。

^{*1} 注意: `encode_quopri()` を使ってエンコードすると、データ中のタブ文字や空白文字もエンコードされます。

exception `email.errors.MultipartConversionError`

この例外は、`Message` オブジェクトに `add_payload()` メソッドを使ってペイロードを追加するとき、そのペイロードがすでにスカラー値である (訳注: リストでない) にもかかわらず、そのメッセージの `Content-Type` ヘッダのメインタイプがすでに設定されていて、それが `multipart` 以外になってしまっている場合にこの例外が発生します。`MultipartConversionError` は `MessageError` と組み込みの `TypeError` を両方継承しています。

`Message.add_payload()` はもはや推奨されないメソッドのため、この例外はめったに発生しません。しかしこの例外は `attach()` メソッドが `MIMENonMultipart` から派生したクラスのインスタンス (例: `MIMEImage` など) に対して呼ばれたときにも発生することがあります。

以下は `FeedParser` がメッセージの解析中に検出する障害 (defect) の一覧です。これらの障害は、問題が見つかったメッセージに追加されるため、たとえば `multipart/alternative` 内にあるネストしたメッセージが異常なヘッダをもっていた場合には、そのネストしたメッセージが障害を持っているが、その親メッセージには障害はないとみなされることに注意してください。

すべての障害クラスは `email.errors.MessageDefect` のサブクラスですが、これは例外とは違いますので注意してください！

バージョン 2.4 で追加: 全ての障害クラスが追加されました。

- `NoBoundaryInMultipartDefect` – メッセージが `multipart` だと宣言されているのに、`boundary` パラメータがない。
- `StartBoundaryNotFoundDefect` – `Content-Type` ヘッダで宣言された開始境界がない。
- `FirstHeaderLineIsContinuationDefect` – メッセージの最初のヘッダが継続行から始まっている。
- `MisplacedEnvelopeHeaderDefect` – ヘッダブロックの途中に "Unix From" ヘッダがある。
- `MalformedHeaderDefect` – コロンのないヘッダがある、あるいはそれ以外の異常なヘッダである。
- `MultipartInvariantViolationDefect` – メッセージが `multipart` だと宣言されているのに、サブパートが存在しない。注意: メッセージがこの障害を持っているとき、`is_multipart()` メソッドはたとえその `content-type` が `multipart` であっても `false` を返すことがあります。

18.1.9 `email.utils`: 多方面のユーティリティ

`email.utils` モジュールではいくつかの便利なユーティリティを提供しています:

`email.utils.quote(str)`

文字列 `str` 内のバックスラッシュをバックスラッシュ 2 つに置換した新しい文字列を返します。また、ダブルクォートはバックスラッシュ + ダブルクォートに置換されます。

`email.utils.unquote(str)`

文字列 `str` の クォートを外した 新しい文字列を返します。`str` の先頭と末尾がダブルクォートだった場合、それらは取り除かれます。同様に `str` の先頭と末尾が角ブラケット (<, >) だった場合もそれらは取り除かれます。

`email.utils.parseaddr(address)`

`To` や `Cc` のようなフィールドを持つアドレスをパースして、構成要素の 実名 と 電子メールアドレスを取り出します。パースに成功した場合、これらの情報持つタプルを返します。失敗した場合は 2 要素のタプル ('', '') を返します。

`email.utils.formataddr(pair)`

`parseaddr()` の逆で、2 要素のタプル (realname, email_address) を取って `To` や `Cc` ヘッダに適した文字列を返します。タプル `pair` の第 1 要素が偽である場合、第 2 要素の値をそのまま返します。

`email.utils.getaddresses(fieldvalues)`

このメソッドは `parseaddr()` が返す形式の 2 要素タプルのリストを返します。`fieldvalues` は `Message.get_all` が返すような一連のヘッダフィールドです。以下はメッセージの全ての受信者を得る簡単な例です:

```
from email.utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

`email.utils.parsedate(date)`

RFC 2822 の規則に基づいて日付の解析を試みます。しかしながらメイラーによっては指定された形式に従っていないものがあるので、その場合 `parsedate()` は正しく推測しようとします。`date` は "Mon, 20 Nov 1995 19:12:08 -0500" のような **RFC 2822** 形式の日付を含む文字列です。日付の解析に成功した場合、`parsedate()` は関数 `time.mktime()` に直接渡せる形式の 9 要素からなるタプルを返します。失敗した場合は `None` を返します。返されるタプルの 6、7、8 番目の添字は使用不可なので注意してください。

`email.utils.parsedate_tz(date)`

`parsedate()` と同様に機能しますが、`None` か 10 要素のタプルを返します。最初の 9 要素は `time.mktime()` に直接渡すことの出来るタプルを成し、10 番目はその日付のタイムゾーンの UTC (グリニッジ標準時の公式用語) からの差です^{*1}。入力文字列にタイムゾーンがない場合、返されるタプルの最後の要素は `None` です。返されるタプルの 6、7、8 番目の添字は使用不可なので注意してください。

`email.utils.mktime_tz(tuple)`

`parsedate_tz()` が返す 10 要素のタプルを UTC のタイムスタンプ (エポックからの秒数) に変換します。与えられた時間帯が `None` である場合、時間帯として現地時間 (localtime) が仮定されます。

`email.utils.formatdate([timeval[, localtime[, usegmt]])`

日付を **RFC 2822** 形式の文字列で返します。例:

^{*1} 注意: この時間帯のオフセット値は `time.timezone` の値と符号が逆です。これは `time.timezone` が POSIX 標準に準拠しているのに対して、こちらは **RFC 2822** に準拠しているからです。

Fri, 09 Nov 2001 01:08:47 -0000

与えられた場合、オプションの *timeval* は `time.gmtime()` や `time.localtime()` に渡すことの出来る浮動小数の時刻です。それ以外の場合、現在時刻が使われます。

オプション引数 *localtime* はフラグです。True の場合、この関数は *timeval* を解析して UTC の代わりに現地のタイムゾーンに対する日付を返します。おそらく夏時間も考慮するでしょう。デフォルトは False で、UTC が使われます。

オプション引数 *usegmt* はフラグです。True の場合、この関数はタイムゾーンを数値の -0000 ではなく ascii 文字列 GMT として日付を出力します。これはプロトコルによっては (例えば HTTP) 必要です。これは *localtime* が False のときのみ適用されます。デフォルトは False です。

バージョン 2.4 で追加。

`email.utils.make_msgid([idstring])`

RFC 2822 準拠形式の *Message-ID* ヘッダに適した文字列を返します。オプション引数 *idstring* が文字列として与えられた場合、これはメッセージ ID の一意性を高めるのに利用されます。

`email.utils.decode_rfc2231(s)`

RFC 2231 に従って文字列 *s* をデコードします。

`email.utils.encode_rfc2231(s[, charset[, language]])`

RFC 2231 に従って *s* をエンコードします。オプション引数 *charset* および *language* が与えられた場合、これらは文字セット名と言語名として使われます。もしこれらのどちらも与えられていない場合、*s* はそのまま返されます。*charset* は与えられているが *language* が与えられていない場合、文字列 *s* は *language* の空文字列を使ってエンコードされます。

`email.utils.collapse_rfc2231_value(value[, errors[, fallback_charset]])`

ヘッダのパラメータが **RFC 2231** 形式でエンコードされている場合、`Message.get_param` は 3 要素からなるタプルを返すことがあります。ここには、そのパラメータの文字セット、言語、および値の順に格納されています。`collapse_rfc2231_value()` はこのパラメータをひとつの Unicode 文字列にまとめます。オプション引数 *errors* は built-in である `unicode()` 関数の引数 *errors* に渡されます。このデフォルト値は `replace` となっています。オプション引数 *fallback_charset* は、もし **RFC 2231** ヘッダの使用している文字セットが Python の知っているものではなかった場合の非常用文字セットとして使われます。デフォルトでは、この値は `us-ascii` です。

便宜上、`collapse_rfc2231_value()` に渡された引数 *value* がタプルでない場合には、これは文字列でなければなりません。その場合にはクォートを除いた文字列を返します。

`email.utils.decode_params(params)`

RFC 2231 に従って引数のリストをデコードします。*params* は (content-type, string-value) のような形式の 2 要素タプルです。

バージョン 2.4 で変更: `dump_address_pair()` 関数は削除されました。かわりに `format_addr()` 関数を使ってください。

バージョン 2.4 で変更: `decode()` 関数は削除されました。かわりに `Header.decode_header` メソッドを使ってください。

バージョン 2.4 で変更: `encode()` 関数は削除されました。かわりに `Header.encode` メソッドを使ってください。

脚注

18.1.10 `email.iterators`: イテレータ

`Message.walk` メソッドを使うと、簡単にメッセージオブジェクトツリー内を次から次へとたどる (iteration) ことができます。 `email.iterators` モジュールはこのための高水準イテレータをいくつか提供します。

`email.iterators.body_line_iterator(msg[, decode])`

このイテレータは `msg` 中のすべてのサブパートに含まれるペイロードをすべて順にたどっていき、ペイロード内の文字列を 1 行ずつ返します。サブパートのヘッダはすべて無視され、Python 文字列でないペイロードからなるサブパートも無視されます。これは `readline()` を使って、ファイルからメッセージを (ヘッダだけとばして) フラットなテキストとして読むのにいくぶん似ているかもしれません。

オプション引数 `decode` は、 `Message.get_payload` にそのまま渡されます。

`email.iterators.typed_subpart_iterator(msg[, maintype[, subtype]])`

このイテレータは `msg` 中のすべてのサブパートをたどり、それらの中で指定された MIME 形式 `maintype` と `subtype` をもつようなパートのみを返します。

`subtype` は省略可能であることに注意してください。これが省略された場合、サブパートの MIME 形式は `maintype` のみがチェックされます。じつは `maintype` も省略可能で、その場合にはデフォルトは `text` です。

つまり、デフォルトでは `typed_subpart_iterator()` は MIME 形式 `text/*` をもつサブパートを順に返していくというわけです。

以下の関数は役に立つデバッグ用ツールとして追加されたもので、パッケージとして公式なサポートのあるインターフェイスではありません。

`email.iterators._structure(msg[, fp[, level]])`

そのメッセージオブジェクト構造の content-type をインデントつきで表示します。たとえば:

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
  text/plain
  text/plain
  multipart/digest
    message/rfc822
      text/plain
    message/rfc822
      text/plain
    message/rfc822
      text/plain
    message/rfc822
      text/plain
```

(次のページに続く)

(前のページからの続き)

```
message/rfc822
    text/plain
text/plain
```

オプション引数 *fp* は出力を渡すためのストリーム (訳注: 原文では file-like) オブジェクトです。これは Python の拡張 print 文が対応できるようになっている必要があります。 *level* は内部的に使用されます。

18.1.11 email: 使用例

ここでは *email* パッケージを使って電子メールメッセージを読む・書く・送信するいくつかの例を紹介します。より複雑な MIME メッセージについても扱います。

最初に、テキスト形式の単純なメッセージを作成・送信する方法です:

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.mime.text import MIMEText

# Open a plain text file for reading.  For this example, assume that
# the text file contains only ASCII characters.
fp = open(textfile, 'rb')
# Create a text/plain message
msg = MIMEText(fp.read())
fp.close()

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = 'The contents of %s' % textfile
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server, but don't include the
# envelope header.
s = smtplib.SMTP('localhost')
s.sendmail(me, [you], msg.as_string())
s.quit()
```

Parser() クラスの parse(filename) が parsestr(message.as_string) メソッドを使って簡単に RFC822 ヘッダの解析ができます:

```
# Import the email modules we'll need
from email.parser import Parser

# If the e-mail headers are in a file, uncomment this line:
#headers = Parser().parse(open(messagefile, 'r'))

# Or for parsing headers in a string, use:
```

(次のページに続く)

(前のページからの続き)

```
headers = Parser().parsestr('From: <user@example.com>\n'
    'To: <someone_else@example.com>\n'
    'Subject: Test message\n'
    '\n'
    'Body would go here\n')

# Now the header items can be accessed as a dictionary:
print 'To: %s' % headers['to']
print 'From: %s' % headers['from']
print 'Subject: %s' % headers['subject']
```

つぎに、あるディレクトリ内にある何枚かの家族写真をひとつの MIME メッセージに収めて送信する例です:

```
# Import smtplib for the actual sending function
import smtplib

# Here are the email package modules we'll need
from email.mime.image import MIMEImage
from email.mime.multipart import MIMEMultipart

COMMASPACE = ', '

# Create the container (outer) email message.
msg = MIMEMultipart()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = COMMASPACE.join(family)
msg.preamble = 'Our family reunion'

# Assume we know that the image files are all in PNG format
for file in pngfiles:
    # Open the files in binary mode. Let the MIMEImage class automatically
    # guess the specific image type.
    fp = open(file, 'rb')
    img = MIMEImage(fp.read())
    fp.close()
    msg.attach(img)

# Send the email via our own SMTP server.
s = smtplib.SMTP('localhost')
s.sendmail(me, family, msg.as_string())
s.quit()
```

つぎはあるディレクトリに含まれている内容全体をひとつの電子メールメッセージとして送信するやり方です:^{*1}

^{*1} 最初の思いつきと用例は Matthew Dixon Cowles のおかげです。

```
#!/usr/bin/env python

"""Send the contents of a directory as a MIME message."""

import os
import sys
import smtplib
# For guessing MIME type based on file name extension
import mimetypes

from optparse import OptionParser

from email import encoders
from email.message import Message
from email.mime.audio import MIMEAudio
from email.mime.base import MIMEBase
from email.mime.image import MIMEImage
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

COMMASPACE = ', '

def main():
    parser = OptionParser(usage="""\
Send the contents of a directory as a MIME message.

Usage: %prog [options]

Unless the -o option is given, the email is sent by forwarding to your local
SMTP server, which then does the normal delivery process.  Your local machine
must be running an SMTP server.
""")
    parser.add_option('-d', '--directory',
                      type='string', action='store',
                      help="""Mail the contents of the specified directory,
otherwise use the current directory.  Only the regular
files in the directory are sent, and we don't recurse to
subdirectories.""")
    parser.add_option('-o', '--output',
                      type='string', action='store', metavar='FILE',
                      help="""Print the composed message to FILE instead of
sending the message to the SMTP server.""")
    parser.add_option('-s', '--sender',
                      type='string', action='store', metavar='SENDER',
                      help='The value of the From: header (required)')
    parser.add_option('-r', '--recipient',
                      type='string', action='append', metavar='RECIPIENT',
                      default=[], dest='recipients',
                      help='A To: header value (at least one required)')
    opts, args = parser.parse_args()
    if not opts.sender or not opts.recipients:
```

(次のページに続く)

(前のページからの続き)

```

    parser.print_help()
    sys.exit(1)
directory = opts.directory
if not directory:
    directory = '.'
# Create the enclosing (outer) message
outer = MIMEMultipart()
outer['Subject'] = 'Contents of directory %s' % os.path.abspath(directory)
outer['To'] = COMMASPACE.join(opts.recipients)
outer['From'] = opts.sender
outer.preamble = 'You will not see this in a MIME-aware mail reader.\n'

for filename in os.listdir(directory):
    path = os.path.join(directory, filename)
    if not os.path.isfile(path):
        continue
    # Guess the content type based on the file's extension. Encoding
    # will be ignored, although we should check for simple things like
    # gzip'd or compressed files.
    ctype, encoding = mimetypes.guess_type(path)
    if ctype is None or encoding is not None:
        # No guess could be made, or the file is encoded (compressed), so
        # use a generic bag-of-bits type.
        ctype = 'application/octet-stream'
    maintype, subtype = ctype.split('/', 1)
    if maintype == 'text':
        fp = open(path)
        # Note: we should handle calculating the charset
        msg = MIMEText(fp.read(), _subtype=subtype)
        fp.close()
    elif maintype == 'image':
        fp = open(path, 'rb')
        msg = MIMEImage(fp.read(), _subtype=subtype)
        fp.close()
    elif maintype == 'audio':
        fp = open(path, 'rb')
        msg = MIMEAudio(fp.read(), _subtype=subtype)
        fp.close()
    else:
        fp = open(path, 'rb')
        msg = MIMEBase(maintype, subtype)
        msg.set_payload(fp.read())
        fp.close()
        # Encode the payload using Base64
        encoders.encode_base64(msg)
        # Set the filename parameter
        msg.add_header('Content-Disposition', 'attachment', filename=filename)
    outer.attach(msg)

# Now send or store the message
composed = outer.as_string()
if opts.output:

```

(次のページに続く)

(前のページからの続き)

```
fp = open(opts.output, 'w')
fp.write(composed)
fp.close()

else:
    s = smtplib.SMTP('localhost')
    s.sendmail(opts.sender, opts.recipients, composed)
    s.quit()

if __name__ == '__main__':
    main()
```

つぎに、上のような MIME メッセージをどうやって展開してひとつのディレクトリ上の複数ファイルにするかを示します:

```
#!/usr/bin/env python

"""Unpack a MIME message into a directory of files."""

import os
import sys
import email
import errno
import mimetypes

from optparse import OptionParser

def main():
    parser = OptionParser(usage="""\
Unpack a MIME message into a directory of files.

Usage: %prog [options] msgfile
""")
    parser.add_option('-d', '--directory',
                      type='string', action='store',
                      help="""Unpack the MIME message into the named
                      directory, which will be created if it doesn't already
                      exist.""")
    opts, args = parser.parse_args()
    if not opts.directory:
        parser.print_help()
        sys.exit(1)

    try:
        msgfile = args[0]
    except IndexError:
        parser.print_help()
        sys.exit(1)

    try:
```

(次のページに続く)

(前のページからの続き)

```

    os.mkdir(opts.directory)
except OSError as e:
    # Ignore directory exists error
    if e.errno != errno.EEXIST:
        raise

fp = open(msgfile)
msg = email.message_from_file(fp)
fp.close()

counter = 1
for part in msg.walk():
    # multipart/* are just containers
    if part.get_content_maintype() == 'multipart':
        continue
    # Applications should really sanitize the given filename so that an
    # email message can't be used to overwrite important files
    filename = part.get_filename()
    if not filename:
        ext = mimetypes.guess_extension(part.get_content_type())
        if not ext:
            # Use a generic bag-of-bits extension
            ext = '.bin'
        filename = 'part-%03d%s' % (counter, ext)
    counter += 1
    fp = open(os.path.join(opts.directory, filename), 'wb')
    fp.write(part.get_payload(decode=True))
    fp.close()

if __name__ == '__main__':
    main()

```

つぎの例は、HTML メッセージを代替プレーンテキスト版付きで作るやりかたです。^{*2}

```

#!/usr/bin/env python

import smtplib

from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

# me == my email address
# you == recipient's email address
me = "my@email.com"
you = "your@email.com"

# Create message container - the correct MIME type is multipart/alternative.
msg = MIMEMultipart('alternative')

```

(次のページに続く)

^{*2} Martin Matejek が教えてくれました。

(前のページからの続き)

```
msg['Subject'] = "Link"
msg['From'] = me
msg['To'] = you

# Create the body of the message (a plain-text and an HTML version).
text = "Hi!\nHow are you?\nHere is the link you wanted:\nhttps://www.python.org"
html = """\
<html>
  <head></head>
  <body>
    <p>Hi!<br>
      How are you?<br>
      Here is the <a href="https://www.python.org">link</a> you wanted.
    </p>
  </body>
</html>
"""

# Record the MIME types of both parts - text/plain and text/html.
part1 = MIMEText(text, 'plain')
part2 = MIMEText(html, 'html')

# Attach parts into message container.
# According to RFC 2046, the last part of a multipart message, in this case
# the HTML message, is best and preferred.
msg.attach(part1)
msg.attach(part2)

# Send the message via local SMTP server.
s = smtplib.SMTP('localhost')
# sendmail function takes 3 arguments: sender's address, recipient's address
# and message to send - here it is sent as one string.
s.sendmail(me, you, msg.as_string())
s.quit()
```

参考:

`smtplib` モジュール SMTP プロトコルクライアント

`nntplib` モジュール NNTP プロトコルクライアント

18.1.12 パッケージの履歴

このテーブルは email パッケージのリリース履歴を表しています。それぞれのバージョンと、それが同梱された Python のバージョンとの関連が示されています。このドキュメントでの、追加/変更されたバージョンの表記は email パッケージのバージョンではなく、Python のバージョンです。このテーブルは Python の各バージョン間の email パッケージの互換性も示しています。

email バージョン	配布	互換
1.x	Python 2.2.0 から Python 2.2.1 まで	もうサポートされません
2.5	Python 2.2.2+ と Python 2.3	Python 2.1 から 2.5 まで
3.0	Python 2.4	Python 2.3 から 2.5 まで
4.0	Python 2.5	Python 2.3 から 2.5 まで

以下は `email` バージョン 4 と 3 の間のおもな差分です:

- 全モジュールが **PEP 8** 標準にあわせてリネームされました。たとえば、version 3 でのモジュール `email.Message` は version 4 では `email.message` になりました。
- 新しいサブパッケージ `email.mime` が追加され、version 3 の `email.MIME*` は、`email.mime` のサブパッケージにまとめられました。たとえば、version 3 での `email.MIMEText` は、`email.mime.text` になりました。

Python 2.6 までは *version 3* の名前も有効です。

- `email.mime.application` モジュールが追加されました。これは `MIMEApplication` クラスを含んでいます。
- version 3 で推奨されないとされた機能は削除されました。これらは `Generator.__call__()`、`Message.get_type()`、`Message.get_main_type()`、`Message.get_subtype()` を含みます。
- **RFC 2331** サポートの修正が追加されました。これは `Message.get_param` などの関数の戻り値を変更します。いくつかの環境では、3 つ組のタプルで返されていた値が 1 つの文字列で返されます (とくに、全ての拡張パラメータセグメントがエンコードされていなかった場合、予測されていた language や charset の指定がないと、戻り値は単純な文字列になります)。過去の版では % デコードがエンコードされているセグメントおよびエンコードされていないセグメントに対して行われましたが、エンコードされたセグメントのみで行われるようになりました。

`email` バージョン 3 とバージョン 2 との違いは以下のようなものです:

- `FeedParser` クラスが新しく導入され、`Parser` クラスは `FeedParser` を使って実装されるようになりました。このパーザは non-strict なものであり、解析はベストエフォート方式でおこなわれ解析中に例外を発生させることはありません。解析中に発見された問題はそのメッセージの `defect` (障害) 属性に保存されます。
- バージョン 2 で `DeprecationWarning` を発生していた API はすべて削除されました。以下のものが含まれています: `MIMEText` コンストラクタに渡す引数 `_encoder`、`Message.add_payload()` メソッド、`Utils.dump_address_pair()` 関数、そして `Utils.decode()` と `Utils.encode()` です。
- 新しく以下の関数が `DeprecationWarning` を発生するようになりました: `Generator.__call__()`、`Message.get_type()`、`Message.get_main_type()`、`Message.get_subtype()`、そして `Parser` クラスに対する `strict` 引数です。これらは `email` の将来のバージョンで撤廃される予定です。
- Python 2.3 以前はサポートされなくなりました。

`email` バージョン 2 とバージョン 1 との違いは以下のようなものです:

- `email.Header` モジュールおよび `email.Charset` モジュールが追加されています。
- `Message` インスタンスの Pickle 形式が変わりました。が、これは正式に定義されたことは一度もないので (そしてこれからも)、この変更は互換性の欠如とはみなされていません。ですがもしお使いのアプリケーションが `Message` インスタンスを pickle あるいは unpickle しているなら、現在 `email` バージョン 2 では `Message` インスタンスがプライベート変数 `_charset` および `_default_type` を含むようになったということに注意してください。
- `Message` クラス中のいくつかのメソッドは推奨されなくなったか、あるいは呼び出し形式が変更になっています。また、多くの新しいメソッドが追加されています。詳しくは `Message` クラスの文書を参照してください。これらの変更は完全に下位互換になっているはずです。
- `message/rfc822` 形式のコンテナは、見た目上のオブジェクト構造が変わりました。 `email` バージョン 1 ではこの content type はスカラー形式のペイロードとして表現されていました。つまり、コンテナメッセージの `is_multipart()` は `false` を返し、 `get_payload()` はリストオブジェクトではなく単一の `Message` インスタンスを直接返すようになっていたのです。

この構造はパッケージ中のほかの部分と整合がとれていなかったため、 `message/rfc822` 形式のオブジェクト表現形式が変更されました。 `email` バージョン 2 では、コンテナは `is_multipart()` で `True` を返し*ます。また `get_payload()` はひとつの `Message` インスタンスを要素とするリストを返すようになりました。

注意: ここは下位互換が完全には成りたたなくなっている部分のひとつです。けれどもあらかじめ `get_payload()` が返すタイプをチェックするようになっていれば問題にはなりません。ただ `message/rfc822` 形式のコンテナを `Message` インスタンスにじかに `set_payload()` しないようにさえすればよいのです。

- `Parser` コンストラクタに `strict` 引数が追加され、 `parse()` および `parsestr()` メソッドには `headersonly` 引数がつきました。 `strict` フラグはまた `email.message_from_file()` と `email.message_from_string()` にも追加されています。
- `Generator.__call__()` はもはや推奨されなくなりました。かわりに `Generator.flatten` を使ってください。また、 `Generator` クラスには `clone()` メソッドが追加されています。
- `email.generator` モジュールに `DecodedGenerator` クラスが加わりました。
- 中間的な基底クラスである `MIMENonMultipart` および `MIMEMultipart` がクラス階層の中に追加され、ほとんどの MIME 関係の派生クラスがこれを介するようになっています。
- `MIMEText` コンストラクタの `_encoder` 引数は推奨されなくなりました。いまやエンコーダは `_charset` 引数にもとづいて暗黙のうちに決定されます。
- `email.utils` モジュールにおける以下の関数は推奨されなくなりました: `dump_address_pairs()`, `decode()`, および `encode()`。また、このモジュールには以下の関数が追加されています: `make_msgid()`, `decode_rfc2231()`, `encode_rfc2231()` そして `decode_params()`。
- Public ではない関数 `email.iterators._structure()` が追加されました。

18.1.13 mimelib との違い

`email` パッケージはもともと `mimelib` と呼ばれる個別のライブラリからつくられたものです。その後変更が加えられ、メソッド名がより一貫したものになり、いくつかのメソッドやモジュールが加えられたりはずされたりしました。いくつかのメソッドでは、その意味も変更されています。しかしほとんどの部分において、`mimelib` パッケージで使うことのできた機能は、ときどきその方法が変わってはいるものの `email` パッケージでも使用可能です。`mimelib` パッケージと `email` パッケージの間の下位互換性はあまり優先はされませんでした。

以下では `mimelib` パッケージと `email` パッケージにおける違いを簡単に説明し、それに沿ってアプリケーションを移植するさいの指針を述べています。

おそらく 2 つのパッケージのもっとも明らかな違いは、パッケージ名が `email` に変更されたことでしょう。さらにトップレベルのパッケージが以下のように変更されました:

- `messageFromString()` は `message_from_string()` に名前が変更されました。
- `messageFromFile()` は `message_from_file()` に名前が変更されました。

`Message` クラスでは、以下のような違いがあります:

- `asString()` メソッドは `as_string()` に名前が変更されました。
- `ismultipart()` メソッドは `is_multipart()` に名前が変更されました。
- `get_payload()` メソッドはオプション引数として `decode` をとるようになりました。
- `getall()` メソッドは `get_all()` に名前が変更されました。
- `addheader()` メソッドは `add_header()` に名前が変更されました。
- `gettype()` メソッドは `get_type()` に名前が変更されました。
- `getmaintype()` メソッドは `get_main_type()` に名前が変更されました。
- `getsubtype()` メソッドは `get_subtype()` に名前が変更されました。
- `getparams()` メソッドは `get_params()` に名前が変更されました。また、従来の `getparams()` は文字列のリストを返していましたが、`get_params()` は 2-タブルのリストを返すようになっています。これはそのパラメータのキーと値の組が、`'='` 記号によって分離されたものです。
- `getparam()` メソッドは `get_param()` に名前が変更されました。
- `getcharsets()` メソッドは `get_charsets()` に名前が変更されました。
- `getfilename()` メソッドは `get_filename()` に名前が変更されました。
- `getboundary()` は `get_boundary()` に名前が変更されました。
- `setboundary()` メソッドは `set_boundary()` に名前が変更されました。
- `getdecodedpayload()` メソッドは廃止されました。これと同様の機能は `get_payload()` メソッドの `decode` フラグに 1 を渡すことで実現できます。

- `getpayloadastext()` メソッドは廃止されました。これと同様の機能は `email.generator` モジュールの `DecodedGenerator` クラスによって提供されます。
- `getbodyastext()` メソッドは廃止されました。これと同様の機能は `email.iterators` モジュールにある `typed_subpart_iterator()` を使ってイテレータを作ることにより実現できます。

`Parser` クラスは、その public なインターフェイスは変わっていませんが、これはより一層かしこくなって `message/delivery-status` 形式のメッセージを認識するようになりました。これは配送状態通知^{*1} において、各ヘッダブロックを表す独立した `Message` サブパートを含むひとつの `Message` インスタンスとして表現されます。

`Generator` クラスは、その public なインターフェイスは変わっていませんが、`email.generator` モジュールに新しいクラスが加わりました。`DecodedGenerator` と呼ばれるこのクラスは以前 `Message`。`getpayloadastext()` メソッドで使われていた機能のほとんどを提供します。

また、以下のモジュールおよびクラスが変更されています:

- `MIMEBase` クラスのコンストラクタ引数 `_major` と `_minor` は、それぞれ `_maintype` と `_subtype` に変更されています。
- `Image` クラスおよびモジュールは `MIMEImage` に名前が変更されました。`_minor` 引数も `_subtype` に名前が変更されています。
- `Text` クラスおよびモジュールは `MIMEText` に名前が変更されました。`_minor` 引数も `_subtype` に名前が変更されています。
- `MessageRFC822` クラスおよびモジュールは `MIMEMessage` に名前が変更されました。注意: 従来バージョンの `mimelib` では、このクラスおよびモジュールは `RFC822` という名前でしたが、これは大文字小文字を区別しないファイルシステムでは Python の標準ライブラリモジュール `rfc822` と名前がかち合っていました。

また、`MIMEMessage` クラスはいまや `message main type` をもつあらゆる種類の MIME メッセージを表現できるようになりました。これはオプション引数として、MIME subtype を指定する `_subtype` 引数をとることができるようになっています。デフォルトでは、`_subtype` は `rfc822` になります。

`mimelib` では、`address` および `date` モジュールでいくつかのユーティリティ関数が提供されていました。これらの関数はすべて `email.utils` モジュールの中に移されています。

`MsgReader` クラスおよびモジュールは廃止されました。これにもっとも近い機能は `email.iterators` モジュール中の `body_line_iterator()` 関数によって提供されています。

脚注

18.2 json — JSON エンコーダおよびデコーダ

バージョン 2.6 で追加.

^{*1} 配送状態通知 (Delivery Status Notifications, DSN) は [RFC 1894](#) によって定義されています。

JSON (JavaScript Object Notation) は、**RFC 7159** (**RFC 4627** を obsolete) と **ECMA-404** によって定義された軽量のデータ交換用のフォーマットです。JavaScript のオブジェクトリテラル記法に由来しています (JavaScript の厳密なサブセットではありませんが^{*1})。

`json` の API は標準ライブラリの `marshal` や `pickle` のユーザに馴染み深いものです。

基本的な Python オブジェクト階層のエンコーディング:

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}})
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print json.dumps("\foo\bar")
"\foo\bar"
>>> print json.dumps(u'\u1234')
"\u1234"
>>> print json.dumps('\'')
"\'"
>>> print json.dumps({'c': 0, "b": 0, "a": 0}, sort_keys=True)
{"a": 0, "b": 0, "c": 0}
>>> from StringIO import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

コンパクトなエンコーディング:

```
>>> import json
>>> json.dumps([1,2,3,{ '4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
```

見やすい表示:

```
>>> import json
>>> print json.dumps({'4': 5, '6': 7}, sort_keys=True,
...                  indent=4, separators=(',', ' '))
{
    "4": 5,
    "6": 7
}
```

JSON のデコーディング:

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
[u'foo', {u'bar': [u'baz', None, 1.0, 2]}]
>>> json.loads('"\foo\bar"')
u'foo\x08ar'
>>> from StringIO import StringIO
```

(次のページに続く)

^{*1} RFC 7159 正誤表 で述べられている通り、JSON は (ECMAScript Edition 5.1 の) JavaScript とは逆に、U+2028 (LINE SEPARATOR) と U+2029 (PARAGRAPH SEPARATOR) が文字列内に含まれることを許容しています。

(前のページからの続き)

```
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
[u'streaming API']
```

JSON オブジェクトのデコーディング方法を眺める:

```
>>> import json
>>> def as_complex(dct):
...     if '__complex__' in dct:
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...     object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads('1.1', parse_float=decimal.Decimal)
Decimal('1.1')
```

JSONEncoder の拡張:

```
>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         # Let the base class default method raise the TypeError
...         return json.JSONEncoder.default(self, obj)
...
>>> json.dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[', '2.0', ',', '1.0', ']']
```

シェルから `json.tool` を使って妥当性チェックをして見やすく表示:

```
$ echo '{"json":"obj"}' | python -m json.tool
{
  "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

注釈: JSON は [YAML 1.2](#) のサブセットです。このモジュールのデフォルト設定 (特に、デフォルトの セパレータ 値) で生成される JSON は [YAML 1.0](#) および [1.1](#) のサブセットでもあります。このモジュールは [YAML シリアライザ](#) としても使えます。

18.2.1 基本的な使い方

```
json.dump(obj, fp, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True,
          cls=None, indent=None, separators=None, encoding="utf-8", default=None,
          sort_keys=False, **kw)
```

この変換表を使って、*obj* を JSON 形式の *fp* (*fp.write()* がサポートされている *file-like object*) へのストリームとして直列化します。

skipkeys が真 (デフォルトは *False*) ならば、基本型 (*str*, *unicode*, *int*, *long*, *float*, *bool*, *None*) 以外の辞書のキーは *TypeError* を送出せずに読み飛ばされます。

ensure_ascii が真 (これがデフォルトです) ならば全ての非 ASCII 文字は出力において `\uXXXX` エスケープされて、結果は ASCII 文字のみからなる *str* インスタンスになります。 *ensure_ascii* が偽の場合、*fp* へ書き込まれるチャンクは *unicode* インスタンスになることがあります。これは普通は入力に Unicode 文字列を含むか、*encoding* パラメータを使う場合に起こります。 *fp.write()* が (*codecs.getwriter()* でのように) *unicode* であると明示的に理解していない限り、これはエラーを引き起こすかもしれません。

check_circular が *false* (デフォルトは *True*) ならば、コンテナ型の循環参照チェックが省かれ、循環参照があれば *OverflowError* (またはもっと悪い結果) に終わります。

allow_nan が偽 (デフォルトは *True*) の場合、許容範囲外の *float* 値 (*nan*, *inf*, *-inf*) を JSON 仕様を厳格に守って直列化すると、*ValueError* になります。 *allow_nan* が真の場合は、JavaScript の等価なもの (*NaN*, *Infinity*, *-Infinity*) が使われます。

indent が非負の整数であれば、JSON の配列要素とオブジェクトメンバはそのインデントレベルで見やすく表示されます。インデントレベルが 0 か負であれば改行だけが挿入されます。 *None* (デフォルト) では最もコンパクトな表現が選択されます。

注釈: デフォルトの要素のセパレータが `' ', ' '` なので、*indent* が指定されたときは出力の末尾に空白文字が付くかもしれません。これを避けるために *separators=(',', ':')* が使えます。

separators を指定する場合は、(*item_separator*, *key_separator*) というタプルでなければなりません。デフォルトでは `(' ', ' ', ' ': '')` が使われます。最もコンパクトな JSON の表現を得たければ空白を削った `(' ', ' ', ' ': '')` を指定すればいいでしょう。

encoding は文字列のエンコーディングで、デフォルトは UTF-8 です。

default を指定する場合は関数を指定して、この関数はそれ以外では直列化できないオブジェクトに対して呼び出されます。その関数は、オブジェクトを JSON でエンコードできるバージョンにして返すか、さもなければ *TypeError* を送出しなければなりません。指定しない場合は、*TypeError* が送出されます。

sort_keys が *true* (デフォルトでは *False* です) であれば、辞書の出力がキーでソートされます。

カスタマイズされた *JSONEncoder* のサブクラス (たとえば追加の型を直列化するように *default()* メソッドをオーバーライドしたもの) を使うには、*cls* キーワード引数に指定します; 指定しなければ

`JSONEncoder` が使われます。

注釈: `pickle` や `marshal` と違って、JSON はフレーム付きのプロトコル (framed protocol) ではないので、同じ `fp` に対して繰り返し `dump()` を呼び、たくさんのオブジェクトを直列化しようとすると、不正な JSON ファイルが作られてしまいます。

```
json.dumps(obj, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True,
            cls=None, indent=None, separators=None, encoding="utf-8", default=None,
            sort_keys=False, **kw)
```

この変換表を使って、`obj` を JSON 形式の `str` オブジェクトに直列化します。`ensure_ascii` が偽の場合は、結果は非 ASCII を含むかもしれず、戻り値が `unicode` のインスタンスになることがあります。

引数は `dump()` のものと同じ意味になります。

注釈: JSON のキー値ペアのキーは、常に `str` 型です。辞書が JSON に変換されるとき、辞書の全てのキーは文字列へ強制的に変換が行われます。この結果として、辞書が JSON に変換され、それから辞書に戻された場合、辞書は元のものと同じではありません。つまり文字列ではないキーを持っている場合、`loads(dumps(x)) != x` となるということです。

```
json.load(fp[, encoding[, cls[, object_hook[, parse_float[, parse_int[, parse_constant[,
            object_pairs_hook[, **kw]]]]]]]]))
```

この変換表を使い、`fp.read()` をサポートし JSON ドキュメントを含んでいる *file-like object* を Python オブジェクトへ脱直列化します。

`fp` の内容が ASCII に基づいたしかし UTF-8 ではないエンコーディング (たとえば latin-1) を使っているならば、適切な `encoding` 名が指定されなければなりません。エンコーディングが ASCII に基づかないもの (UCS-2 など) であることは許されないので、`codecs.getreader(encoding)(fp)` というように包むか、または単に `unicode` オブジェクトにデコードしたものを `loads()` に渡して下さい。

`object_hook` はオプションの関数で、任意のオブジェクトリテラルがデコードされた結果 (`dict`) に対し呼び出されます。`object_hook` の戻り値は `dict` の代わりに使われます。この機能は独自のデコーダ (例えば JSON-RPC クラスヒンティング) を実装するのに使えます。

`object_pairs_hook` はオプションで渡す関数で、ペアの順序付きリストのデコード結果に対して呼ばれます。`object_pairs_hook` の戻り値は `dict` の代わりに使われます。この機能はキーと値のデコードされる順序に依存する独自のデコーダ (例えば `collections.OrderedDict()` は挿入の順序を記憶します) を実装するのに使えます。`object_hook` も定義されている場合は、`object_pairs_hook` が優先して使用されます。

バージョン 2.7 で変更: `object_pairs_hook` のサポートが追加されました。

`parse_float` は、もし指定されれば、全てのデコードされる JSON の浮動小数点数文字列に対して呼ばれます。デフォルトでは、`float(num_str)` と等価です。これは JSON 浮動小数点数に対して他のデータ型やパーサ (たとえば `decimal.Decimal`) を使うのに使えます。

`parse_int` は、もし指定されれば、全てのデコードされる JSON の整数文字列に対して呼ばれます。デ

フォルトでは、`int(num_str)` と等価です。これは JSON 整数に対して他のデータ型やパーサ (たとえば `float`) を使うのに使えます。

`parse_constant` は、もし指定されれば、次の文字列に対して呼ばれます: `'-Infinity'`, `'Infinity'`, `'NaN'`, `'null'`, `'true'`, `'false'`。これは不正な JSON 数値に遭遇したときに例外を送出するのに使えます。

バージョン 2.7 で変更: `'null'`, `'true'`, `'false'` に対して `parse_constant` は呼びされません。

カスタマイズされた `JSONDecoder` のサブクラスを使うには、`cls` キーワード引数に指定します; 指定しなかった場合は `JSONDecoder` が使われます。追加のキーワード引数はこのクラスのコンストラクタに引き渡されます。

```
json.loads(s[, encoding[, cls[, object_hook[, parse_float[, parse_int[, parse_constant[, ob-
ject_pairs_hook[, **kw]]]]]]]])
```

この変換表を使い、`s` (JSON ドキュメントを含んでいる `str` または `unicode` のインスタンス) を Python オブジェクトへ脱直列化します。

`s` が ASCII に基づいたしかし UTF-8 ではないエンコーディング (たとえば latin-1) でエンコードされた `str` ならば、適切な `encoding` 名が指定されなければなりません。エンコーディングが ASCII に基づかないもの (UCS-2 など) であることは許されないの、まず `unicode` にデコードして下さい。

その他の引数は `load()` のものと同じ意味です。

18.2.2 エンコーダとデコーダ

```
class json.JSONDecoder([encoding[, object_hook[, parse_float[, parse_int[, parse_constant[,
strict[, object_pairs_hook]]]]]]])
```

単純な JSON デコーダ。

デフォルトではデコーディングの際、以下の変換を行います:

JSON	Python
object	dict
array	list
string	unicode
number (int)	int, long
number (real)	float
true	True
false	False
null	None

また、このデコーダは `NaN`, `Infinity`, `-Infinity` を対応する `float` の値として、JSON の仕様からは外れますが、理解します。

`encoding` はこのインスタンスでデコードされる `str` オブジェクトを解釈するために使われるエンコーディング (デフォルトは UTF-8) を定めます。 `unicode` オブジェクトのデコーディングには影響を与

えません。

注意して欲しいのは現状では ASCII のスーパーセットであるようなエンコーディングだけうまく動くということです。他のエンコーディングの文字列は `unicode` にして渡して下さい。

`object_hook` は、もし指定されれば、全てのデコードされた JSON オブジェクトに対して呼ばれその返値は与えられた `dict` の代わりに使われます。この機能は独自の脱直列化 (たとえば JSON-RPC クラスヒンテイングをサポートするような) を提供するのに使えます。

`object_pairs_hook` は、もし指定されれば、全てのペアの順序付きリストにデコードされた JSON オブジェクトに対して呼ばれます。 `object_pairs_hook` の返り値は `dict` の代わりに使われます。この機能はキーと値のデコードされる順序に依存する独自のデコーダ (たとえば `collections.OrderedDict()` は挿入の順序を記憶します) を実装するのに使えます。 `object_hook` も定義されている場合は、 `object_pairs_hook` が優先して使用されます。

バージョン 2.7 で変更: `object_pairs_hook` のサポートが追加されました。

`parse_float` は、もし指定されれば、全てのデコードされる JSON の浮動小数点数文字列に対して呼ばれます。デフォルトでは、 `float(num_str)` と等価です。これは JSON 浮動小数点数に対して他のデータ型やパーサ (たとえば `decimal.Decimal`) を使うのに使えます。

`parse_int` は、もし指定されれば、全てのデコードされる JSON の整数文字列に対して呼ばれます。デフォルトでは、 `int(num_str)` と等価です。これは JSON 整数に対して他のデータ型やパーサ (たとえば `float`) を使うのに使えます。

`parse_constant` は、もし指定されれば、次の文字列に対して呼ばれます: `'-Infinity'`, `'Infinity'`, `'NaN'`, `'null'`, `'true'`, `'false'`。これは不正な JSON 数値に遭遇したときに例外を送出するのに使えます。

`strict` が `false` (デフォルトは `True`) の場合、制御文字を文字列に含めることができます。ここで言う制御文字とは、`'\t'` (タブ)、`'\n'`、`'\r'`、`'\0'` を含む 0-31 の範囲のコードを持つ文字のことです。

脱直列化しようとしているデータが不正な JSON ドキュメントだった場合、 `ValueError` が送出されます。

decode (*s*)

s (*str* または `unicode` インスタンスで JSON 文書を含むもの) の Python 表現を返します。

raw_decode (*s*)

s (*str* または `unicode` インスタンスで JSON 文書で始まるもの) から JSON 文書をデコードし、Python 表現と *s* の文書の終わるところのインデックスからなる 2 要素のタプルを返します。

このメソッドは後ろに余分なデータを従えた文字列から JSON 文書をデコードするのに使えます。

class `json.JSONEncoder` (`[skipkeys[, ensure_ascii[, check_circular[, allow_nan[, sort_keys[, indent[, separators[, encoding[, default]]]]]]]]])`
Python データ構造に対する拡張可能な JSON エンコーダ。

デフォルトでは以下のオブジェクトと型をサポートします:

Python	JSON
dict	object
list, tuple	array
str, unicode	string
int, long, float	number
True	true
False	false
None	null

このクラスを拡張して他のオブジェクトも認識するようにするには、サブクラスを作って `default()` メソッドを次のように実装します。もう一つ別のメソッドでオブジェクト `o` に対する直列化可能なオブジェクトを返すものを呼び出すようにします。変換できない時はスーパークラスの実装を (`TypeError` を送出させるために) 呼ばなければなりません。

`skipkeys` が偽 (デフォルト) ならば、str, int, long, float, None 以外のキーをエンコードする試みは `TypeError` に終わります。 `skipkeys` が真の場合は、それらのアイテムは単に読み飛ばされます。

`ensure_ascii` が真 (これがデフォルトです) の場合、全ての非 ASCII 文字は出力において `\uXXXX` エスケープされて、結果は ASCII 文字のみからなる `str` インスタンスになります。 `ensure_ascii` が偽の場合、結果は `unicode` インスタンスになることがあります。これは普通は入力に Unicode 文字列を含むか、 `encoding` パラメータを使う場合に起こります。

`check_circular` が true (デフォルト) ならば、リスト、辞書および自作でエンコードしたオブジェクトは循環参照がないかエンコード中にチェックされ、無限再帰 (これは `OverflowError` を引き起こします) を防止します。 True でない場合は、そういったチェックは施されません。

`allow_nan` が true (デフォルト) ならば、 NaN, Infinity, -Infinity はそのままエンコードされます。この振る舞いは JSON 仕様に従っていませんが、大半の JavaScript ベースのエンコーダ、デコーダと矛盾しません。 True でない場合は、そのような浮動小数点数をエンコードすると `ValueError` が送出されます。

`sort_keys` が true (デフォルトは False) ならば、辞書の出力がキーでソートされます。これは JSON の直列化がいつでも比較できるようになるので回帰試験の際に便利です。

`indent` が非負の整数であれば (デフォルトは None です)、JSON の配列要素とオブジェクトメンバはそのインデントレベルで見やすく表示されます。インデントレベルが 0 であれば 改行だけが挿入されます。 None では最もコンパクトな表現が選択されます。

注釈: デフォルトの要素のセパレータが `' , '` なので、 `indent` が指定されたときは出力の末尾に空白文字が付くかもしれません。これを避けるために `separators=(',', ':')` が使えます。

`separators` を指定する場合は、 `(item_separator, key_separator)` というタプルでなければなりません。デフォルトでは `(' ', ':')` が使われます。最もコンパクトな JSON の表現を得たければ空白を削った `(':', ':')` を指定すればいいでしょう。

`default` を指定する場合は関数を指定して、この関数はそれ以外では直列化できないオブジェクトに対して呼び出されます。その関数は、オブジェクトを JSON でエンコードできるバージョンにして返すか、さもなければ `TypeError` を送出しなければなりません。指定しない場合は、`TypeError` が送出されます。

`encoding` が `None` でなければ、入力文字列は全て JSON エンコーディングに先立ってこのエンコーディングで Unicode に変換されます。デフォルトは UTF-8 です。

`default(o)`

このメソッドをサブクラスで実装する際には `o` に対して直列化可能なオブジェクトを返すか、基底クラスの実装を (`TypeError` を送出するために) 呼び出すかします。

たとえば、任意のイテレータをサポートするために、次のように実装します:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

`encode(o)`

Python データ構造 `o` の JSON 文字列表現を返します。たとえば:

```
>>> JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

`iterencode(o)`

与えられたオブジェクト `o` をエンコードし、得られた文字列表現ごとに yield します。たとえば:

```
for chunk in JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

18.2.3 標準への準拠と互換性

JSON 形式の仕様は [RFC 7159](#) と [ECMA-404](#) で規定されています。この節では、このモジュールの RFC への準拠水準について詳しく述べます。簡単のために、`JSONEncoder` および `JSONDecoder` の子クラスと、明示的に触れられていないパラメータについては考慮しないことにします。

このモジュールは、JavaScript では正しいが JSON では不正ないくつかの拡張が実装されているため、厳密な意味では RFC に準拠していません。特に:

- 無限および NaN の数値を受け付け、また出力します;
- あるオブジェクト内での同じ名前の繰り返しを受け付け、最後の名前と値のペアの値のみを使用します。

この RFC は、RFC 準拠のパーサが RFC 準拠でない入力テキストを受け付けることを許容しているので、このモジュールの脱直列化は技術的に言えば、デフォルトの設定では RFC に準拠しています。

文字エンコーディング

RFC は、UTF-8、UTF-16、UTF-32 のいずれかで JSON を表現するように要求しており、UTF-8 が最大の互換性を確保するために推奨されるデフォルトです。このため、このモジュールは *encoding* パラメータのデフォルトに UTF-8 を使います。

このモジュールのデシリアライザは直接的には、ASCII 互換のエンコーディングでのみ動作します。UTF-16、UTF-32 やほかの ASCII 非互換のエンコーディングには、このドキュメント内でデシリアライザの *encoding* パラメータについて説明した回避方法を取る必要があります。

RFC で要求ではなく許可されている通り、このモジュールのシリアライザはデフォルトで *ensure_ascii=True* という設定を用い、従って、結果の文字列が ASCII 文字しか含まないように出力をエスケープします。

RFC は JSON テキストの最初にバイトオーダーマーク (BOM) を追加することを禁止していますので、このモジュールはその出力に BOM を追加しません。RFC は JSON デシリアライザが入力の一番最初の BOM を無視することを、許容はしますが求めてはいません。このモジュールのデシリアライザが一番最初の BOM を見つけると *ValueError* を送出します。

RFC は JSON 文字列に正当な Unicode 文字に対応付かないバイト列 (例えばベアにならない UTF-16 サロゲートのかたわれ) が含まれることを明示的に禁止してはならず、もちろんこれは相互運用性の問題を引き起こします。デフォルトでは、このモジュールは (オリジナルの *str* にある場合) そのようなシーケンスのコードポイントを受け取り、出力します。

無限および NaN の数値

RFC は、無限もしくは NaN の数値の表現は許可していません。それにも関わらずデフォルトでは、このモジュールは Infinity、-Infinity、NaN を正しい JSON の数値リテラルの値であるかのように受け付け、出力します:

```
>>> # Neither of these calls raises an exception, but the results are not valid_
↪ JSON
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # Same when deserializing
>>> json.loads('-Infinity')
-inf
>>> json.loads('NaN')
nan
```

シリアライザでは、この振る舞いを変更するのに *allow_nan* パラメータが使えます。デシリアライザでは、この振る舞いを変更するのに *parse_constant* パラメータが使えます。

オブジェクト中に重複した名前の扱い

RFC は JSON オブジェクト中の名前はユニークでなければならないと規定していますが、JSON オブジェクトで名前が繰り返された場合の扱いについて指定していません。デフォルトでは、このモジュールは例外を出せず、かわりに重複した名前のうち、最後に出現した名前と値のペア以外を無視します。

```
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{'x': 3}
```

`object_pairs_hook` パラメータでこの動作を変更できます。

トップレベルの非オブジェクト、非配列の値の扱い

廃止された [RFC 4627](#) によって規定された古いバージョンの JSON では、JSON テキストのトップレベルの値は JSON オブジェクトか配列 (Python での `dict` か `list`) であることを要求していたが、JSON の `null`, `boolean`, `number`, `string` であることは許されていませんでしたが、この制限は [RFC 7159](#) により取り払われました。このモジュールはこの制限を持っていませんし、シリアライザでもデシリアライズでも、一度としてこの制限で実装されたことはありません。

それに関わらず、相互運用可能性を最大化したいならば、あなた自身の手で自発的にその制約に忠実に従いたいと思うでしょう。

実装の制限

いくつかの JSON デシリアライザの実装は、以下の制限を設定することがあります。

- 受け入れられる JSON テキストのサイズ
- JSON オブジェクトと配列のネストの最大の深さ
- JSON 数値の範囲と精度
- JSON 文字列の内容と最大の長さ

このモジュールは関連する Python データ型や Python インタプリタ自身の制約の世界を超えたそのような制約を強要はしません。

JSON にシリアライズする際には、あなたの JSON を消費する側のアプリケーションが持つ当該制約に思いを馳せてください。とりわけ JSON 数値を IEEE 754 倍精度浮動小数にデシリアライズする際の問題はありがちで、すなわちその有効桁数と精度の制限の影響を受けます。これは、極端に大きな値を持った Python `int` をシリアライズするとき、あるいは `decimal.Decimal` のような”風変わりな”数値型をシリアライズするとき、に特に関係があります。

18.3 mailcap — mailcap ファイルの操作

ソースコード: [Lib/mailcap.py](#)

mailcap ファイルは、メールリーダや Web ブラウザのような MIME 対応のアプリケーションが、異なる MIME タイプのファイルにどのように反応するかを設定するために使われます ("mailcap" の名前は "mail capability" から取られました)。例えば、ある mailcap ファイルに `video/mpeg; xmpeg %s` のような行が入っているとします。ユーザが email メッセージや Web ドキュメント上でその MIME タイプ `video/mpeg` に遭遇すると、`%s` はファイル名 (通常テンポラリファイルに属するものになります) に置き換えられ、ファイルを閲覧するために `xmpeg` プログラムが自動的に起動されます。

mailcap の形式は [RFC 1524](#), "A User Agent Configuration Mechanism For Multimedia Mail Format Information" で文書化されていますが、この文書はインターネット標準ではありません。しかしながら、mailcap ファイルはほとんどの Unix システムでサポートされています。

`mailcap.findmatch(caps, MIMETYPE[, key[, filename[, plist]]])`

2 要素のタプルを返します; 最初の要素は文字列で、実行すべきコマンド (`os.system()` に渡されます) が入っています。二つめの要素は与えられた MIME タイプに対する mailcap エントリです。一致する MIME タイプが見つからなかった場合、`(None, None)` が返されます。

`key` は desired フィールドの値で、実行すべき動作のタイプを表現します; ほとんどの場合、単に MIME 形式のデータ本体を見たいと思うので、標準の値は `'view'` になっています。与えられた MIME 型をもつ新たなデータ本体を作成した場合や、既存のデータ本体を置き換えたい場合には、`'view'` の他に `'compose'` および `'edit'` を取ることもできます。これらフィールドの完全なリストについては [RFC 1524](#) を参照してください。

`filename` はコマンドライン中で `%s` に代入されるファイル名です; 標準の値は `'/dev/null'` で、たいていこの値を使いたいわけではないはずです。従って、ファイル名を指定してこのフィールドを上書きする必要があるでしょう。

`plist` は名前付けされたパラメタのリストです; 標準の値は単なる空のリストです。リスト中の各エントリはパラメタ名を含む文字列、等号 (`'='`)、およびパラメタの値でなければなりません。mailcap エントリには `%{foo}` といったような名前つきのパラメタを含めることができ、`'foo'` と名づけられたパラメタの値に置き換えられます。例えば、コマンドライン `showpartial %{id} %{number} %{total}` が mailcap ファイルにあり、`plist` が `['id=1', 'number=2', 'total=3']` に設定されていれば、コマンドラインは `'showpartial 1 2 3'` になります。

mailcap ファイル中では、オプションの `"test"` フィールドを使って、(計算機アーキテクチャや、利用しているウィンドウシステムといった) 何らかの外部条件をテストするよう指定することができます。`findmatch()` はこれらの条件を自動的にチェックし、チェックが失敗したエントリを読み飛ばします。

`mailcap.getcaps()`

MIME タイプを mailcap ファイルのエントリに対応付ける辞書を返します。この辞書は `findmatch()` 関数に渡されるべきものです。エントリは辞書のリストとして記憶されますが、この表現形式の詳細について知っておく必要はないでしょう。

mailcap 情報はシステム上で見つかった全ての mailcap ファイルから導出されます。ユーザ設定の mailcap ファイル `$HOME/.mailcap` はシステムの mailcap ファイル `/etc/mailcap`、`/usr/etc/mailcap`、および `/usr/local/etc/mailcap` の内容を上書きします。

以下に使用例を示します:

```
>>> import mailcap
>>> d = mailcap.getcaps()
>>> mailcap.findmatch(d, 'video/mpeg', filename='tmp1223')
('xmpeg tmp1223', {'view': 'xmpeg %s'})
```

18.4 mailbox — 様々な形式のメールボックス操作

このモジュールでは二つのクラス *Mailbox* および *Message* をディスク上のメールボックスとそこに収められたメッセージへのアクセスと操作のために定義しています。 *Mailbox* は辞書のようなキーからメッセージへの対応付けを提供しています。 *Message* は *email.message* モジュールの *Message* を拡張して形式ごとの状態と振る舞いを追加しています。サポートされるメールボックスの形式は Maildir, mbox, MH, Babyl, MMDF です。

参考:

email モジュール メッセージの表現と操作。

18.4.1 Mailbox オブジェクト

class mailbox.Mailbox

メールボックス。内容を確認したり変更したりできます。

Mailbox 自体はインタフェースを定義し形式ごとのサブクラスに継承されるように意図されたもので、インスタンス化されることは想定されていません。インスタンス化したいならばサブクラスを代わりに使うべきです。

Mailbox のインタフェースは辞書風で、小さなキーがメッセージに対応します。キーは対象となる *Mailbox* インスタンスが発行するもので、そのインスタンスに対してのみ意味を持ちます。一つのキーは一つのメッセージにひも付けられ、その対応はメッセージが他のメッセージで置き換えられるような更新をされたあとも続きます。

メッセージを *Mailbox* インスタンスに追加するには集合風のメソッド *add()* を使います。また削除は *del* 文または集合風の *remove()* や *discard()* を使って行ないます。

Mailbox インタフェースのセマンティクスと辞書のそれとは注意すべき違いがあります。メッセージは、要求されるたびに新しい表現 (典型的には *Message* インスタンス) が現在のメールボックスの状態に基づいて生成されます。同様に、メッセージが *Mailbox* インスタンスに追加される時も、渡されたメッセージ表現の内容がコピーされます。どちらの場合も *Mailbox* インスタンスにメッセージ表現への参照は保たれません。

デフォルトの *Mailbox* イテレータはメッセージ表現ごとに繰り返すもので、辞書のイテレータのようにキーごとの繰り返しではありません。さらに、繰り返し中のメールボックスを変更することは安全であり整合的に定義されています。イテレータが作られた後にメールボックスに追加されたメッセージはそのイテレータからは見えません。そのイテレータが *yield* するまえにメールボックスから削除されたメッセージは黙ってスキップされますが、イテレータからのキーを使ったときにはそのキーに対応する

メッセージが削除されているならば `KeyError` を受け取ることになります。

警告: 十分な注意を、何か他のプロセスによっても同時に変更される可能性のあるメールボックスを更新する時は、払わなければなりません。そのようなタスクをこなすのに最も安全なメールボックス形式は Maildir で、mbox のような単一ファイルの形式を並行した書き込みに利用するのは避けるように努力しましょう。メールボックスを更新する場面では、必ず `lock()` と `unlock()` メソッドを、ファイル内のメッセージを読んだり書き込んだり削除したりといった操作をする前に、呼び出してロックします。メールボックスをロックし損なうと、メッセージを失ったりメールボックス全体をぐちゃぐちゃにしたりする羽目に陥ります。

`Mailbox` インスタンスには次のメソッドがあります:

add(*message*)

メールボックスに *message* を追加し、それに割り当てられたキーを返します。

引数 *message* は `Message` インスタンス、`email.message.Message` インスタンス、文字列、ファイル風オブジェクト (テキストモードで開かれていなければなりません) を使えます。*message* が適切な形式に特化した `Message` サブクラスのインスタンス (例えばメールボックスが `mbox` インスタンスのときの `mboxMessage` インスタンス) であれば、形式ごとの情報が利用されます。そうでなければ、形式ごとに必要な情報は適当なデフォルトが使われます。

remove(*key*)

__delitem__(*key*)

discard(*key*)

メールボックスから *key* に対応するメッセージを削除します。

対応するメッセージが無い場合、メソッドが `remove()` または `__delitem__()` として呼び出されている時は `KeyError` 例外が送出されます。しかし、`discard()` として呼び出されている場合は例外は発生しません。基づいているメールボックス形式が別のプロセスからの平行した変更をサポートしているならば、この `discard()` の振る舞いの方が好まれるかもしれません。

__setitem__(*key*, *message*)

key に対応するメッセージを *message* で置き換えます。*key* に対応しているメッセージが既に無くなっている場合 `KeyError` 例外が送出されます。

`add()` と同様に、引数の *message* には `Message` インスタンス、`email.message.Message` インスタンス、文字列、ファイル風オブジェクト (テキストモードで開かれていなければなりません) を使えます。*message* が適切な形式に特化した `Message` サブクラスのインスタンス (例えばメールボックスが `mbox` インスタンスのときの `mboxMessage` インスタンス) であれば、形式ごとの情報が利用されます。そうでなければ、現在 *key* に対応するメッセージの形式ごとの情報が変更されずに残ります。

iterkeys()

keys()

`iterkeys()` として呼び出されると全てのキーについてのイテレータを返しますが、`keys()` として呼び出されるとキーのリストを返します。

itervalues()**__iter__()****values()**

`itervalues()` または `__iter__()` として呼び出されると全てのメッセージの表現についてのイテレータを返しますが、`values()` として呼び出されるとその表現のリストを返します。メッセージは適切な形式ごとの `Message` サブクラスのインスタンスとして表現されるのが普通ですが、`Mailbox` インスタンスが初期化されるときに指定すればお好みのメッセージファクトリを使うこともできます。

注釈: `__iter__()` は辞書のそれのようにキーについてのイテレータではありません。

iteritems()**items()**

`(key, message)` ペア、ただし `key` はキーで `message` はメッセージ表現、のイテレータ (`iteritems()` として呼び出された場合)、またはリスト (`items()` として呼び出された場合) を返します。メッセージは適切な形式ごとの `Message` サブクラスのインスタンスとして表現されるのが普通ですが、`Mailbox` インスタンスが初期化されるときに指定すればお好みのメッセージファクトリを使うこともできます。

get(key, default=None)**__getitem__(key)**

`key` に対応するメッセージの表現を返します。対応するメッセージが存在しない場合、`get()` として呼び出されたなら `default` を返しますが、`__getitem__()` として呼び出されたなら `KeyError` 例外が送出されます。メッセージは適切な形式ごとの `Message` サブクラスのインスタンスとして表現されるのが普通ですが、`Mailbox` スタンスが初期化されるときに指定すればお好みのメッセージファクトリを使うこともできます。

get_message(key)

`key` に対応するメッセージの表現を形式ごとの `Message` サブクラスのインスタンスとして返します。もし対応するメッセージが存在しなければ `KeyError` 例外が送出されます。

get_string(key)

`key` に対応するメッセージの文字列を返すか、そのようなメッセージが存在しない場合は `KeyError` 例外を送出します。

get_file(key)

`key` に対応するメッセージの表現をファイル風表現として返します。もし対応するメッセージが存在しなければ `KeyError` 例外が送出されます。ファイル風オブジェクトはバイナリモードで開かれているように振る舞います。このファイルは必要がなくなったら閉じなければなりません。

注釈: 他の表現方法とは違い、ファイル風オブジェクトはそれを作り出した `Mailbox` インスタンスやそれが基づいているメールボックスと独立である必要がありません。より詳細な説明は各サブクラスごとにあります。

has_key (*key*)

__contains__ (*key*)

key がメッセージに対応していれば `True` を、そうでなければ `False` を返します。

__len__ ()

メールボックス中のメッセージ数を返します。

clear ()

メールボックスから全てのメッセージを削除します。

pop (*key* [, *default*])

key に対応するメッセージの表現を返し、そのメッセージを削除します。もし対応するメッセージが存在しなければ *default* が供給されていればその値を返し、そうでなければ `KeyError` 例外を送出します。メッセージは適切な形式ごとの `Message` サブクラスのインスタンスとして表現されるのが普通ですが、`Mailbox` インスタンスが初期化されるときに指定すれば好みのメッセージファクトリを使うこともできます。

popitem ()

任意に選んだ (*key*, *message*) ペアを返します。ただしここで *key* はキーで *message* はメッセージ表現です。もしメールボックスが空ならば、`KeyError` 例外を送出します。メッセージは適切な形式ごとの `Message` サブクラスのインスタンスとして表現されるのが普通ですが、`Mailbox` インスタンスが初期化されるときに指定すれば好みのメッセージファクトリを使うこともできます。

update (*arg*)

引数 *arg* は *key* から *message* へのマッピングまたは (*key*, *message*) ペアのイテレート可能オブジェクトでなければなりません。メールボックスは、各 *key* と *message* のペアについて `__setitem__()` を使ったかのように *key* に対応するメッセージが *message* になるように更新されます。`__setitem__()` と同様に、*key* は既存のメールボックス中のメッセージに対応しているものでなければならず、そうでなければ `KeyError` が送出されます。ですから、一般的には *arg* に `Mailbox` インスタンスを渡すのは間違いです。

注釈: 辞書と違い、キーワード引数はサポートされていません。

flush ()

保留されている変更をファイルシステムに書き込みます。`Mailbox` のサブクラスによっては変更はいつも直ちにファイルに書き込まれ `flush()` は何もしないということもありますが、それでもこのメソッドを呼ぶように習慣付けておきましょう。

lock ()

メールボックスの排他的アドバイザリロックを取得し、他のプロセスが変更しないようにします。ロックが取得できない場合 `ExternalClashError` が送出されます。ロック機構はメールボックス形式によって変わります。メールボックスの内容に変更を加えるときはいつもロックを掛けるべきです。

unlock ()

メールボックスのロックが存在する場合は解放します。

`close()`

メールボックスをフラッシュし、必要ならばアンロックし、開いているファイルを閉じます。

`Mailbox` サブクラスによっては何もしないこともあります。

`Maildir`

`class mailbox.Maildir (dirname, factory=rfc822.Message, create=True)`

Maildir 形式のメールボックスのための `Mailbox` のサブクラス。パラメータ `factory` は呼び出し可能オブジェクトで (バイナリモードで開かれているかのように振る舞う) ファイル風メッセージ表現を受け付けて好みの表現を返すものです。 `factory` が `None` ならば、 `MaildirMessage` がデフォルトのメッセージ表現として使われます。 `create` が `True` ならばメールボックスが存在しないときには作成します。

`factory` のデフォルトが `rfc822.Message` であつたり、 `path` ではなく `dirname` という名前であつたりというのは歴史的理由によるものです。 `Maildir` インスタンスが他の `Mailbox` サブクラスと同じように振る舞わせるためには、 `factory` に `None` をセットしてください。

Maildir はディレクトリ型のメールボックス形式でメール転送エージェント qmail 用に発明され、現在では多くの他のプログラムでもサポートされているものです。Maildir メールボックス中のメッセージは共通のディレクトリ構造の下で個別のファイルに保存されます。このデザインにより、Maildir メールボックスは複数の無関係のプログラムからデータを失うことなくアクセスしたり変更したりできます。そのためロックは不要です。

Maildir メールボックスには三つのサブディレクトリ `tmp`, `new`, `cur` があります。メッセージはまず `tmp` サブディレクトリに瞬間的に作られた後、 `new` サブディレクトリに移動されて配送を完了します。メールユーザエージェントが引き続いて `cur` サブディレクトリにメッセージを移動しメッセージの状態についての情報をファイル名に追加される特別な "info" セクションに保存することができます。

Courier メール転送エージェントによって導入されたスタイルのフォルダもサポートされます。主たるメールボックスのサブディレクトリは `'.'` がファイル名の先頭であればフォルダと見なされます。フォルダ名は `Maildir` によって先頭の `'.'` を除いて表現されます。各フォルダはまた Maildir メールボックスですがさらにフォルダを含むことはできません。その代わり、論理的包含関係は例えば `"Archived.2005.07"` のような `'.'` を使ったレベル分けで表わされます。

注釈: 本来の Maildir 仕様ではある種のメッセージのファイル名にコロン (`:`) を使う必要があります。しかしながら、オペレーティングシステムによってはこの文字をファイル名に含めることができないことがあります。そういった環境で Maildir のような形式を使いたい場合、代わりに使われる文字を指定する必要があります。感嘆符 (`!`) を使うのが一般的な選択です。以下の例を見てください:

```
import mailbox
mailbox.Maildir.colon = '!'
```

`colon` 属性はインスタンスごとにセットしても構いません。

`Maildir` インスタンスには `Mailbox` の全てのメソッドに加え以下のメソッドもあります:

list_folders()

全てのフォルダ名のリストを返します。

get_folder(folder)

名前が *folder* であるフォルダを表わす *Maildir* インスタンスを返します。そのようなフォルダが存在しなければ *NoSuchMailboxError* 例外が送出されます。

add_folder(folder)

名前が *folder* であるフォルダを作り、それを表わす *Maildir* インスタンスを返します。

remove_folder(folder)

名前が *folder* であるフォルダを削除します。もしフォルダに一つでもメッセージが含まれていれば *NotEmptyError* 例外が送出されフォルダは削除されません。

clean()

過去 36 時間以内にアクセスされなかったメールボックス内の一時ファイルを削除します。Maildir 仕様はメールを読むプログラムはときどきこの作業をすべきだとしています。

Maildir で実装された *Mailbox* のいくつかのメソッドには特別な注意が必要です:

add(message)

__setitem__(key, message)

update(arg)

警告: これらのメソッドは一意的なファイル名をプロセス ID に基づいて生成します。複数のスレッドを使う場合は、同じメールボックスを同時に操作しないようにスレッド間で調整しておかないと検知されない名前の衝突が起こりメールボックスを壊すかもしれません。

flush()

Maildir メールボックスへの変更は即時に適用されるので、このメソッドは何もしません。

lock()

unlock()

Maildir メールボックスはロックをサポート (または要求) しないので、このメソッドは何もしません。

close()

Maildir インスタンスは開いたファイルを保持しませんしメールボックスはロックをサポートしませんので、このメソッドは何もしません。

get_file(key)

ホストのプラットフォームによっては、返されたファイルが開いている間、元になったメッセージを変更したり削除したりできない場合があります。

参考:

[gmail の maildir マニュアルページ](#) Maildir 形式のオリジナルの仕様。

[Using maildir format](#) Maildir 形式の発明者による注意書き。更新された名前生成規則と ”info” の解釈についても含まれます。

[Courier の maildir マニュアルページ](#) Maildir 形式のもう一つの仕様。フォルダをサポートする一般的な拡張について記述されています。

mbbox

class mailbox.mbox (*path*, *factory*=None, *create*=True)

mbbox 形式のメールボックスのための *Mailbox* のサブクラス。パラメータ *factory* は呼び出し可能オブジェクトで (バイナリモードで開かれているかのように振る舞う) ファイル風メッセージ表現を受け付けて好みの表現を返すものです。 *factory* が None ならば、 *mboxMessage* がデフォルトのメッセージ表現として使われます。 *create* が True ならばメールボックスが存在しないときには作成します。

mbbox 形式は Unix システム上でメールを保存する古くからある形式です。mbbox メールボックスでは全てのメッセージが一つのファイルに保存されておりそれぞれのメッセージは ”From ” という 5 文字で始まる行を先頭に付けられています。

mbbox 形式には幾つかのバリエーションがあり、それぞれオリジナルの形式にあった欠点を克服すると主張しています。互換性のために、 *mbbox* はオリジナルの (時に *mboxo* と呼ばれる) 形式を実装しています。すなわち、 *Content-Length* ヘッダはもしあっても無視され、メッセージのボディにある行頭の ”From ” はメッセージを保存する際に ”>From ” に変換されますが、この ”>From ” は読み出し時にも ”From ” に変換されません。

mbbox で実装された *Mailbox* のいくつかのメソッドには特別な注意が必要です:

get_file (*key*)

mbbox インスタンスに対し *flush()* や *close()* を呼び出した後でファイルを使用すると予期しない結果を引き起こしたり例外が送出されたりすることがあります。

lock()

unlock()

3 種類のロック機構が使われます — ドットロッキングと、もし使用可能ならば *flock()* と *lockf()* システムコールです。

参考:

[gmail の mbox マニュアルページ](#) mbox 形式の仕様および種々のバリエーション。

[tin の mbox マニュアルページ](#) もう一つの mbox 形式の仕様でロックについての詳細を含む。

[Configuring Netscape Mail on Unix: Why The Content-Length Format is Bad](#) バリエーションの一つではなくオリジナルの mbox を使う理由。

”mbox” は相互に互換性を持たないいくつかのメールボックスフォーマットの集まりです mbox バリエーションの歴史。

MH

class mailbox.**MH** (*path*, *factory*=None, *create*=True)

MH 形式のメールボックスのための *Mailbox* のサブクラス。パラメータ *factory* は呼び出し可能オブジェクトで (バイナリモードで開かれているかのように振る舞う) ファイル風メッセージ表現を受け付けて好みの表現を返すものです。 *factory* が None ならば、 *MHMessage* がデフォルトのメッセージ表現として使われます。 *create* が True ならばメールボックスが存在しないときには作成します。

MH はディレクトリに基づいたメールボックス形式で MH Message Handling System というメールユーザエージェントのために発明されました。 MH メールボックス中のそれぞれのメッセージは一つのファイルとして収められています。 MH メールボックスにはメッセージの他に別の MH メールボックス (フォルダ と呼ばれます) を含んでもかまいません。フォルダは無限にネストできます。 MH メールボックスにはもう一つ シーケンス という名前付きのリストでメッセージをサブフォルダに移動することなく論理的に分類するものがサポートされています。シーケンスは各フォルダの *.mh_sequences* というファイルで定義されます。

MH クラスは MH メールボックスを操作しますが、 *mh* の動作の全てを模倣しようとはしていません。特に、 *mh* が状態と設定を保存する *context* や *.mh_profile* といったファイルは書き換えませんし影響も受けません。

MH インスタンスには *Mailbox* の全てのメソッドの他に次のメソッドがあります:

list_folders ()

全てのフォルダ名のリストを返します。

get_folder (*folder*)

folder という名前のフォルダを表わす *MH* インスタンスを返します。もしフォルダが存在しなければ *NoSuchMailboxError* 例外が送出されます。

add_folder (*folder*)

folder という名前のフォルダを作成し、それを表わす *MH* インスタンスを返します。

remove_folder (*folder*)

名前が *folder* であるフォルダを削除します。もしフォルダに一つでもメッセージが含まれていれば *NotEmptyError* 例外が送出されフォルダは削除されません。

get_sequences ()

folder という名前のフォルダを削除します。フォルダにメッセージが一つでも残っていれば、 *NotEmptyError* 例外が送出されフォルダは削除されません。

set_sequences (*sequences*)

シーケンス名をキーのリストに対応付ける辞書を返します。シーケンスが一つもなければ空の辞書を返します。

pack ()

メールボックス中のシーケンスを *get_sequences* () で返されるような名前とキーのリストに対応付ける辞書 *sequences* に基づいて再定義します。

注釈: 番号付けの間隔を詰める必要に応じてメールボックス中のメッセージの名前を付け替えま

す。シーケンスのリストのエントリもそれに応じて更新されます。

既に発行されたキーはこの操作によって無効になるのでそれ以降使ってはなりません:

`remove(key)`

`__delitem__(key)`

`discard(key)`

これらのメソッドはメッセージを直ちに削除します。名前の前にコンマを付加してメッセージに削除の印を付けるという MH の規約は使いません。

`lock()`

`unlock()`

3 種類のロック機構が使われます — ドットロックと、もし使用可能ならば `flock()` と `lockf()` システムコールです。MH メールボックスに対するロックとは `.mh_sequences` のロックと、それが影響を与える操作中だけの個々のメッセージファイルに対するロックを意味します。

`get_file(key)`

ホストプラットフォームにより、ファイルが開かれたままの場合はメッセージを削除することができない場合があります。

`flush()`

MH メールボックスへの変更は即時に適用されますのでこのメソッドは何もしません。

`close()`

MH インスタンスは開いたファイルを保持しませんのでこのメソッドは `unlock()` と同じです。

参考:

[nmh - Message Handling System](#) `mh` の改良版である `nmh` のホームページ。

[MH & nmh: Email for Users & Programmers](#) GPL ライセンスの `mh` および `nmh` の本で、このメールボックス形式についての情報があります。

Babyl

`class mailbox.Babyl(path, factory=None, create=True)`

Babyl 形式のメールボックスのための *Mailbox* のサブクラス。パラメータ *factory* は呼び出し可能オブジェクトで (バイナリモードで開かれているかのように振る舞う) ファイル風メッセージ表現を受け付けて好みの表現を返すものです。 *factory* が `None` ならば、 *BabylMessage* がデフォルトのメッセージ表現として使われます。 *create* が `True` ならばメールボックスが存在しないときには作成します。

Babyl は単一ファイルのメールボックス形式で Emacs に付属している Rmail メールユーザエージェントで使われているものです。メッセージの開始は Control-Underscore (`'\037'`) および Control-L (`'\014'`) の二文字を含む行で示されます。メッセージの終了は次のメッセージの開始または最後のメッセージの場合には Control-Underscore を含む行で示されます。

Babyl メールボックス中のメッセージには二つのヘッダのセット、オリジナルヘッダといわれる可視ヘッダ、があります。可視ヘッダは典型的にはオリジナルヘッダの一部を分り易いように再整形したり短くしたりしたものです。Babyl メールボックス中のそれぞれのメッセージには ラベル というそのメッセージについての追加情報を記録する短い文字列のリストを伴い、メールボックス中に見出されるユーザが定義した全てのラベルのリストは Babyl オプションセクションに保持されます。

Babyl インスタンスには *Mailbox* の全てのメソッドの他に次のメソッドがあります:

get_labels()

メールボックスで使われているユーザが定義した全てのラベルのリストを返します。

注釈: メールボックスにどのようなラベルが存在するかを決めるのに、Babyl オプションセクションのリストを参考にせず、実際のメッセージを探索しますが、Babyl セクションもメールボックスが変更されたときにはいつでも更新されます。

Babyl で実装された *Mailbox* のいくつかのメソッドには特別な注意が必要です:

get_file(key)

Babyl メールボックスにおいて、メッセージのヘッダはボディと繋がって格納されていません。ファイル風の表現を生成するために、ヘッダとボディが (*StringIO* モジュールの) ファイルと同じ API を持つ *StringIO* インスタンスと一緒にコピーされます。その結果、ファイル風オブジェクトは本当に元にしているメールボックスとは独立していますが、文字列表現と比べてメモリーを節約することにもなりません。

lock()

unlock()

3 種類のロック機構が使われます — ドットロッキングと、もし使用可能ならば *flock()* と *lockf()* システムコールです。

参考:

[Format of Version 5 Babyl Files](#) Babyl 形式の仕様。

[Reading Mail with Rmail](#) Rmail のマニュアルで Babyl のセマンティクスについての情報も少しある。

MMDF

class mailbox.MMDF (*path*, *factory*=None, *create*=True)

MMDF 形式のメールボックスのための *Mailbox* のサブクラス。パラメータ *factory* は呼び出し可能オブジェクトで (バイナリモードで開かれているかのように振る舞う) ファイル風メッセージ表現を受け付けて好みの表現を返すものです。 *factory* が None ならば、 *MMDFMessage* がデフォルトのメッセージ表現として使われます。 *create* が True ならばメールボックスが存在しないときには作成します。

MMDF は単一ファイルのメールボックス形式で Multichannel Memorandum Distribution Facility というメール転送エージェント用に発明されたものです。各メッセージは mbox と同様の形式で収められますが、前後を 4 つの Control-A ('\001') を含む行で挟んであります。mbox 形式と同じようにそれぞ

れのメッセージの開始は "From " の 5 文字を含む行で示されますが、それ以外の場所での "From " は格納の際 ">From " には変えられません。それは追加されたメッセージ区切りによって新たなメッセージの開始と見間違ふことが避けられるからです。

MMDF で実装された *Mailbox* のいくつかのメソッドには特別な注意が必要です:

get_file (*key*)

MMDF インスタンスに対し *flush()* や *close()* を呼び出した後でファイルを使用すると予期しない結果を引き起こしたり例外が送出されたりすることがあります。

lock ()

unlock ()

3 種類のロック機構が使われます — ドットロッキングと、もし使用可能ならば *flock()* と *lockf()* システムコールです。

参考:

mmdf man page from tin ニュースリーダ *tin* のドキュメント中の *MMDF* 形式仕様。

MMDF Multichannel Memorandum Distribution Facility についてのウィキペディアの記事。

18.4.2 Message objects

class mailbox.**Message** ([*message*])

email.Message モジュールの *Message* のサブクラス。 *mailbox.Message* のサブクラスはメールボックス形式ごとの状態と動作を追加します。

message が省略された場合、新しいインスタンスはデフォルトの空の状態で生成されます。 *message* が *email.message.Message* インスタンスならばその内容がコピーされます。さらに、 *message* が *Message* インスタンスならば、形式固有の情報も可能な限り変換されます。 *message* が文字列またはファイルならば、読まれ解析されるべき **RFC 2822** 準拠のメッセージを含んでいなければなりません。

サブクラスにより提供される形式ごとの状態と動作は様々ですが、一般に或るメールボックスに固有のものでないプロパティだけがサポートされます (おそらくプロパティのセットはメールボックス形式ごとに固有でしょう)。例えば、単一ファイルメールボックス形式におけるファイルオフセットやディレクトリ式メールボックス形式におけるファイル名は保持されません、というのもそれらは元々のメールボックスにしか適用できないからです。しかし、メッセージがユーザに読まれたかどうかあるいは重要だとマークされたかどうかという状態は保持されます、というのはそれらはメッセージ自体に適用されるからです。

Mailbox インスタンスを使って取得したメッセージを表現するのに *Message* インスタンスが使われなければならないとは要求していません。ある種の状況では *Message* による表現を生成するのに必要な時間やメモリが受け入れられないこともあります。そういった状況では *Mailbox* インスタンスは文字列やファイル風オブジェクトの表現も提供できますし、 *Mailbox* インスタンスを初期化する際にメッセージファクトリーを指定することもできます。

MaildirMessage

class mailbox.**Mai**ldirMessage ([*message*])

Mai

ldir 固有の動作をするメッセージ。引数 *message* は *Message* のコンストラクタと同じ意味を持ちます。

通常、メールユーザエージェントは *new* サブディレクトリにある全てのメッセージをユーザが最初にメールボックスを開くか閉じるかした後で *cur* サブディレクトリに移動し、メッセージが実際に読まれたかどうかを記録します。 *cur* にある各メッセージには状態情報を保存するファイル名に付け加えられた "info" セクションがあります。(メールリーダーの中には "info" セクションを *new* にあるメッセージに付けることもあります。) "info" セクションには二つの形式があります。一つは "2," の後に標準化されたフラグのリストを付けたもの(たとえば "2,FR")、もう一つは "1," の後にいわゆる実験的情報を付け加えるものです。Maildir の標準的なフラグは以下の通りです:

フラグ	意味	説明
D	ドラフト (Draft)	作成中
F	フラグ付き (Flagged)	重要とされたもの
P	通過 (Passed)	転送、再送またはバウンス
R	返答済み (Replied)	返答されたもの
S	既読 (Seen)	読んだもの
T	ごみ (Trashed)	削除予定とされたもの

*Mai*ldirMessage インスタンスは以下のメソッドを提供します:

get_subdir ()

"new" (メッセージが *new* サブディレクトリに保存されるべき場合) または "cur" (メッセージが *cur* サブディレクトリに保存されるべき場合) のどちらかを返します。

注釈: メッセージは通常メールボックスがアクセスされた後、メッセージが読まれたかどうかに関わらず *new* から *cur* に移動されます。メッセージ *msg* は "S" in *msg.get_flags*() が True ならば読まれています。

set_subdir (*subdir*)

メッセージが保存されるべきサブディレクトリをセットします。パラメータ *subdir* は "new" または "cur" のいずれかでなければなりません。

get_flags ()

現在セットされているフラグを特定する文字列を返します。メッセージが標準 Maildir 形式に準拠しているならば、結果はアルファベット順に並べられたゼロまたは 1 回の 'D'、'F'、'P'、'R'、'S'、'T' をつなげたものです。空文字列が返されるのはフラグが一つもない場合、または "info" が実験的セマンティクスを使っている場合です。

set_flags (*flags*)

flags で指定されたフラグをセットし、他のフラグは下ろします。

add_flag (*flag*)

flag で指定されたフラグをセットしますが他のフラグは変えません。一度に二つ以上のフラグをセットすることは、*flag* に 2 文字以上の文字列を指定すればできます。現在の "info" はフラグの代わりに実験的情報を使っている場合でも上書きされます。

remove_flag (*flag*)

flag で指定されたフラグを下ろしますが他のフラグは変えません。一度に二つ以上のフラグを取り除くことは、*flag* に 2 文字以上の文字列を指定すればできます。"info" がフラグの代わりに実験的情報を使っている場合は現在の "info" は書き換えられません。

get_date ()

メッセージの配送日時をエポックからの秒数を表わす浮動小数点数で返します。

set_date (*date*)

メッセージの配送日時を *date* にセットします。*date* はエポックからの秒数を表わす浮動小数点数です。

get_info ()

メッセージの "info" を含む文字列を返します。このメソッドは実験的 (即ちフラグのリストでない) "info" にアクセスし、また変更するのに役立ちます。

set_info (*info*)

"info" に文字列 *info* をセットします。

MaildirMessage インスタンスが *mboxMessage* や *MMDFMessage* のインスタンスに基づいて生成されるとき、*Status* および *X-Status* ヘッダは省かれ以下の変換が行われます:

結果の状態	<i>mboxMessage</i> または <i>MMDFMessage</i> の状態
"cur" サブディレクトリ	O フラグ
F フラグ	F フラグ
R フラグ	A フラグ
S フラグ	R フラグ
T フラグ	D フラグ

MaildirMessage インスタンスが *MHMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	<i>MHMessage</i> の状態
"cur" サブディレクトリ	"unseen" シーケンス
"cur" サブディレクトリおよび S フラグ	"unseen" シーケンス無し
F フラグ	"flagged" シーケンス
R フラグ	"replied" シーケンス

MaildirMessage インスタンスが *BabylMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	<i>BabylMessage</i> の状態
"cur" サブディレクトリ	"unseen" ラベル
"cur" サブディレクトリおよび S フラグ	"unseen" ラベル無し
P フラグ	"forwarded" または "resent" ラベル
R フラグ	"answered" ラベル
T フラグ	"deleted" ラベル

mailboxMessage

class mailbox.**mailboxMessage** ([*message*])

mailbox 固有の動作をするメッセージ。引数 *message* は *Message* のコンストラクタと同じ意味を持ちます。

mailbox メールボックス中のメッセージは単一ファイルにまとめて格納されています。送り主のエンベロープアドレスおよび配送日時は通常メッセージの開始を示す "From " から始まる行に記録されますが、正確なフォーマットに関しては mailbox の実装ごとに大きな違いがあります。メッセージの状態を示すフラグ、たとえば読んだかどうかあるいは重要だとマークを付けられているかどうかといったようなもの、は典型的には *Status* および *X-Status* に収められます。

規定されている mailbox メッセージのフラグは以下の通りです:

フラグ	意味	説明
R	読んだもの	読んだもの
O	古い (Old)	以前に MUA に発見された
D	削除 (Deleted)	削除予定とされたもの
F	フラグ付き (Flagged)	重要とされたもの
A	返答済み (Answered)	返答されたもの

"R" および "O" フラグは *Status* ヘッダに記録され、"D"、"F"、"A" フラグは *X-Status* ヘッダに記録されます。フラグとヘッダは通常記述された順番に出現します。

mailboxMessage インスタンスは以下のメソッドを提供します:

get_from ()

mailbox メールボックスのメッセージの開始を示す "From " 行を表わす文字列を返します。先頭の "From " および末尾の改行は含まれません。

set_from (*from_*, *time_* = None)

"From " 行を *from_* にセットします。 *from_* は先頭の "From " や末尾の改行を含まない形で指定しなければなりません。利便性のために、 *time_* を指定して適切に整形して *from_* に追加させることができます。 *time_* を指定する場合、それは *time.struct_time* インスタンス、 *time.strftime* () に渡すのに適したタプル、または True (この場合 *time.gmtime* () を使います) のいずれかでなければなりません。

get_flags ()

現在セットされているフラグを特定する文字列を返します。メッセージが規定された形式に準拠しているならば、結果は次の順に並べられた 0 回か 1 回の 'R'、'O'、'D'、'F'、'A' です。

set_flags (*flags*)

flags で指定されたフラグをセットして、他のフラグは下ろします。*flags* は並べられたゼロまたは 1 回の 'R'、'O'、'D'、'F'、'A' です。

add_flag (*flag*)

flag で指定されたフラグをセットしますが他のフラグは変えません。一度に二つ以上のフラグをセットすることは、*flag* に 2 文字以上の文字列を指定すればできます。

remove_flag (*flag*)

flag で指定されたフラグを下ろしますが他のフラグは変えません。一二つ以上のフラグを取り除くことは、*flag* に 2 文字以上の文字列を指定すればできます。

mbboxMessage インスタンスが *MaildirMessage* インスタンスに基づいて生成されるとき、*MaildirMessage* インスタンスの配送日時に基づいて "From" 行が作り出され、次の変換が行われます：

結果の状態	<i>MaildirMessage</i> の状態
R フラグ	S フラグ
O フラグ	"cur" サブディレクトリ
D フラグ	T フラグ
F フラグ	F フラグ
A フラグ	R フラグ

mbboxMessage インスタンスが *MHMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます：

結果の状態	<i>MHMessage</i> の状態
R フラグおよび O フラグ	"unseen" シーケンス無し
O フラグ	"unseen" シーケンス
F フラグ	"flagged" シーケンス
A フラグ	"replied" シーケンス

mbboxMessage インスタンスが *BabylMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます：

結果の状態	<i>BabylMessage</i> の状態
R フラグおよび O フラグ	"unseen" ラベル無し
O フラグ	"unseen" ラベル
D フラグ	"deleted" ラベル
A フラグ	"answered" ラベル

mbboxMessage インスタンスが *MMDFMessage* インスタンスに基づいて生成されるとき、"From" 行はコ

ピーされ全てのフラグは直接対応します:

結果の状態	<i>MMDFMessage</i> の状態
R フラグ	R フラグ
O フラグ	O フラグ
D フラグ	D フラグ
F フラグ	F フラグ
A フラグ	A フラグ

MHMessage

class mailbox.MHMessage ([*message*])

MH 固有の動作をするメッセージ。引数 *message* は *Message* のコンストラクタと同じ意味を持ちます。

MH メッセージは伝統的な意味あいにおいてマークやフラグをサポートしません。しかし、MH メッセージにはシーケンスがあり任意のメッセージを論理的にグループ分けできます。いくつかのメールソフト (標準の *mh* や *nmh* はそうではありませんが) は他の形式におけるフラグとほぼ同じようにシーケンスを使います:

シーケンス	説明
unseen	読んではいないが既に MUA に見つけられている
replied	返答されたもの
flagged	重要とされたもの

MHMessage インスタンスは以下のメソッドを提供します:

get_sequences ()

このメッセージを含むシーケンスの名前のリストを返す。

set_sequences (*sequences*)

このメッセージを含むシーケンスのリストをセットする。

add_sequence (*sequence*)

sequence をこのメッセージを含むシーケンスのリストに追加する。

remove_sequence (*sequence*)

sequence をこのメッセージを含むシーケンスのリストから除く。

MHMessage インスタンスが *MaildirMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	<i>MaildirMessage</i> の状態
"unseen" シーケンス	S フラグ無し
"replied" シーケンス	R フラグ
"flagged" シーケンス	F フラグ

MHMessage インスタンスが *mboxMessage* や *MMDFMessage* のインスタンスに基づいて生成されるとき、*Status* および *X-Status* ヘッダは省かれ以下の変換が行われます:

結果の状態	<i>mboxMessage</i> または <i>MMDFMessage</i> の状態
"unseen" シーケンス	R フラグ無し
"replied" シーケンス	A フラグ
"flagged" シーケンス	F フラグ

MHMessage インスタンスが *BabylMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	<i>BabylMessage</i> の状態
"unseen" シーケンス	"unseen" ラベル
"replied" シーケンス	"answered" ラベル

BabylMessage

`class mailbox.BabylMessage([message])`

Babyl 固有の動作をするメッセージ。引数 *message* は *Message* のコンストラクタと同じ意味を持ちます。

ある種のメッセージラベルは アトリビュート と呼ばれ、規約により特別な意味が与えられています。アトリビュートは以下の通りです:

ラベル	説明
unseen	読んではいないが既に MUA に見つけられている
deleted	削除予定とされたもの
filed	他のファイルまたはメールボックスにコピーされた
answered	返答されたもの
forwarded	転送された
edited	ユーザによって変更された
resent	再送された

デフォルトでは Rmail は可視ヘッダのみ表示します。 *BabylMessage* クラスはしかし、オリジナルヘッダをより完全だという理由で使います。可視ヘッダは望むならそのように指示してアクセスすることができます。

BabylMessage インスタンスは以下のメソッドを提供します:

get_labels()

メッセージに付いているラベルのリストを返します。

set_labels(labels)

メッセージに付いているラベルのリストを *labels* にセットします。

add_label(label)

メッセージに付いているラベルのリストに *label* を追加します。

remove_label(label)

メッセージに付いているラベルのリストから *label* を削除します。

get_visible()

ヘッダがメッセージの可視ヘッダでありボディが空であるような *Message* インスタンスを返します。

set_visible(visible)

メッセージの可視ヘッダを *visible* のヘッダと同じにセットします。引数 *visible* は *Message* インスタンスまたは *email.message.Message* インスタンス、文字列、ファイル風オブジェクト (テキストモードで開かれてなければなりません) のいずれかです。

update_visible()

BabylMessage インスタンスのオリジナルヘッダが変更されたとき、可視ヘッダは自動的に対応して変更されるわけではありません。このメソッドは可視ヘッダを以下のように更新します。対応するオリジナルヘッダのある可視ヘッダはオリジナルヘッダの値がセットされます。対応するオリジナルヘッダの無い可視ヘッダは除去されます。そして、オリジナルヘッダにあって可視ヘッダに無い *Date*、*From*、*Reply-To*、*To*、*CC*、*Subject* は可視ヘッダに追加されます。

BabylMessage インスタンスが *MaildirMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	<i>MaildirMessage</i> の状態
"unseen" ラベル	S フラグ無し
"deleted" ラベル	T フラグ
"answered" ラベル	R フラグ
"forwarded" ラベル	P フラグ

BabylMessage インスタンスが *mboxMessage* や *MMDFMessage* のインスタンスに基づいて生成されるとき、*Status* および *X-Status* ヘッダは省かれ以下の変換が行われます:

結果の状態	<i>mboxMessage</i> または <i>MMDFMessage</i> の状態
"unseen" ラベル	R フラグ無し
"deleted" ラベル	D フラグ
"answered" ラベル	A フラグ

`BabylMessage` インスタンスが `MHMessage` インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	<code>MHMessage</code> の状態
"unseen" ラベル	"unseen" シーケンス
"answered" ラベル	"replied" シーケンス

`MMDFMessage`

```
class mailbox.MMDFMessage([message])
```

MMDF 固有の動作をするメッセージ。引数 `message` は `Message` のコンストラクタと同じ意味を持ちます。

mbox メールボックスのメッセージと同様に、MMDF メッセージは送り主のアドレスと配送日時が最初の "From" で始まる行に記録されています。同様に、メッセージの状態を示すフラグは通常 `Status` および `X-Status` ヘッダに収められています。

よく使われる MMDF メッセージのフラグは mbox メッセージのものと同一で以下の通りです:

フラグ	意味	説明
R	読んだもの	読んだもの
O	古い (Old)	以前に MUA に発見された
D	削除 (Deleted)	削除予定とされたもの
F	フラグ付き (Flagged)	重要とされたもの
A	返答済み (Answered)	返答されたもの

"R" および "O" フラグは `Status` ヘッダに記録され、"D"、"F"、"A" フラグは `X-Status` ヘッダに記録されます。フラグとヘッダは通常記述された順番に出現します。

`MMDFMessage` インスタンスは `mboxMessage` インスタンスと同一の以下のメソッドを提供します:

`get_from()`

mbox メールボックスのメッセージの開始を示す "From" 行を表わす文字列を返します。先頭の "From" および末尾の改行は含まれません。

`set_from(from_, time_=None)`

"From" 行を `from_` にセットします。 `from_` は先頭の "From" や末尾の改行を含まない形で指定しなければなりません。利便性のために、 `time_` を指定して適切に整形して `from_` に追加させることができます。 `time_` を指定する場合、それは `time.struct_time` インスタンス、 `time.strftime()` に渡すのに適したタプル、または `True` (この場合 `time.gmtime()` を使います) のいずれかでなければなりません。

`get_flags()`

現在セットされているフラグを特定する文字列を返します。メッセージが規定された形式に準拠しているならば、結果は次の順に並べられた 0 回か 1 回の 'R'、'O'、'D'、'F'、'A' です。

set_flags (*flags*)

flags で指定されたフラグをセットして、他のフラグは下ろします。*flags* は並べられたゼロまたは 1 回の 'R'、'O'、'D'、'F'、'A' です。

add_flag (*flag*)

flag で指定されたフラグをセットしますが他のフラグは変えません。一度に二つ以上のフラグをセットすることは、*flag* に 2 文字以上の文字列を指定すればできます。

remove_flag (*flag*)

flag で指定されたフラグを下ろしますが他のフラグは変えません。一二つ以上のフラグを取り除くことは、*flag* に 2 文字以上の文字列を指定すればできます。

MMDFMessage インスタンスが *MaiIdirMessage* インスタンスに基づいて生成されるとき、“From”行が *MaiIdirMessage* インスタンスの配信日をもとに生成され、以下の変換が行われます:

結果の状態	<i>MaiIdirMessage</i> の状態
R フラグ	S フラグ
O フラグ	”cur” サブディレクトリ
D フラグ	T フラグ
F フラグ	F フラグ
A フラグ	R フラグ

MMDFMessage インスタンスが *MHMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	<i>MHMessage</i> の状態
R フラグおよび O フラグ	”unseen” シーケンス無し
O フラグ	”unseen” シーケンス
F フラグ	”flagged” シーケンス
A フラグ	”replied” シーケンス

MMDFMessage インスタンスが *BabylMessage* インスタンスに基づいて生成されるとき、以下の変換が行われます:

結果の状態	<i>BabylMessage</i> の状態
R フラグおよび O フラグ	”unseen” ラベル無し
O フラグ	”unseen” ラベル
D フラグ	”deleted” ラベル
A フラグ	”answered” ラベル

MMDFMessage インスタンスが *mbxMessage* インスタンスに基づいて生成されるとき、“From”行がコピーされ、全てのフラグが直接対応します:

結果の状態	<i>mboxMessage</i> の状態
R フラグ	R フラグ
O フラグ	O フラグ
D フラグ	D フラグ
F フラグ	F フラグ
A フラグ	A フラグ

18.4.3 例外

mailbox モジュールでは以下の例外クラスが定義されています:

exception mailbox.Error

他の全てのモジュール固有の例外の基底クラス。

exception mailbox.NoSuchMailboxError

メールボックスがあると思っていたが見つからなかった場合に送出されます。これはたとえば *Mailbox* のサブクラスを存在しないパスでインスタンス化しようとしたとき (かつ *create* パラメータは *False* であった場合)、あるいは存在しないフォルダを開こうとした時などに発生します。

exception mailbox.NotEmptyError

メールボックスが空であることを期待されているときに空でない場合、たとえばメッセージの残っているフォルダを削除しようとした時などに送出されます。

exception mailbox.ExternalClashError

メールボックスに関係したある条件がプログラムの制御を外れてそれ以上作業を続けられなくなった場合、たとえば他のプログラムが既に保持しているロックを取得しようとして失敗したとき、あるいは一意的に生成されたファイル名が既に存在していた場合などに送出されます。

exception mailbox.FormatError

ファイル中のデータが解析できない場合、たとえば *MH* インスタンスが壊れた *.mh.sequences* ファイルを読もうと試みた場合などに送出されます。

18.4.4 非推奨のクラスとモジュール

バージョン 2.6 で非推奨。

古いバージョンの *mailbox* モジュールはメッセージの追加や削除といったメールボックスの変更をサポートしていませんでした。また形式ごとのメッセージプロパティを表現するクラスも提供していませんでした。後方互換性のために、古いメールボックスクラスもまだ使うことができますが、できるだけ新しいクラスを使うべきです。古いクラスは Python 3 で削除されました。

古いメールボックスオブジェクトはイテレーションと一つの公開メソッドだけを提供していました:

```
oldmailbox.next()
```

メールボックスオブジェクトのコンストラクタに渡された、オプションの *factory* 引数を使って、メー

ルボックス中の次のメッセージを生成して返します。標準の設定では、`factory` は `rfc822.Message` オブジェクトです (`rfc822` モジュールを参照してください)。メールボックスの実装により、このオブジェクトの `fp` 属性は真のファイルオブジェクトかもしれないし、複数のメールメッセージが単一のファイルに収められているなどの場合に、メッセージ間の境界を注意深く扱うためにファイルオブジェクトをシミュレートするクラスのインスタンスであるかもしれません。次のメッセージがない場合、このメソッドは `None` を返します。

ほとんどの古いメールボックスクラスは現在のメールボックスクラスと違う名前ですが、`Maildir` だけは例外です。そのため、新しい方の `Maildir` クラスには `next()` メソッドが定義され、コンストラクタも他の新しいメールボックスクラスとは少し異なります。

古いメールボックスのクラスで名前が新しい対応物と同じでないものは以下の通りです:

```
class mailbox.UnixMailbox (fp[,factory])
```

全てのメッセージが単一のファイルに収められ、`From` (`From_` として知られています) 行によって分割されているような、旧来の Unix 形式のメールボックスにアクセスします。ファイルオブジェクト `fp` はメールボックスファイルを指します。オプションの `factory` パラメタは新たなメッセージオブジェクトを生成するような呼び出し可能オブジェクトです。`factory` は、メールボックスオブジェクトに対して `next()` メソッドを実行した際に、単一の引数、`fp` を伴って呼び出されます。この引数の標準の値は `rfc822.Message` クラスです (`rfc822` モジュール – および以下 – を参照してください)。

注釈: このモジュールの実装上の理由により、`fp` オブジェクトはバイナリモードで開くようにしてください。特に Windows 上では注意が必要です。

可搬性を最大限にするために、Unix 形式のメールボックス内にあるメッセージは、正確に 'From ' (末尾の空白に注意してください) で始まる文字列が、直前の正しく二つの改行の後にくるような行で分割されます。現実的には広範なバリエーションがあるため、それ以外の `From_` 行について考慮すべきではないのですが、現在の実装では先頭の二つの改行をチェックしていません。これはほとんどのアプリケーションでうまく動作します。

`UnixMailbox` クラスでは、ほぼ正確に `From_` デリミタにマッチするような正規表現を用いることで、より厳密に `From_` 行のチェックを行うバージョンを実装しています。`UnixMailbox` ではデリミタ行が `From name time` の行に分割されるものと考えます。可搬性を最大限にするためには、代わりに `PortableUnixMailbox` クラスを使ってください。このクラスは `UnixMailbox` と同じですが、個々のメッセージは `From` 行だけで分割されるものとみなします。

```
class mailbox.PortableUnixMailbox (fp[,factory])
```

厳密性の低い `UnixMailbox` のバージョンで、メッセージを分割する行は `From` のみであると見なします。実際に見られるメールボックスのバリエーションに対応するため、`From` 行における "name time" 部分は無視されます。メール処理ソフトウェアはメッセージ中の 'From ' で始まる行をクオートするため、この分割はうまく動作します。

```
class mailbox.MmdfMailbox (fp[,factory])
```

全てのメッセージが単一のファイルに収められ、4 つの control-A 文字によって分割されているような、MMDf 形式のメールボックスにアクセスします。ファイルオブジェクト `fp` はメールボックスファイルをさします。オプションの `factory` は `UnixMailbox` クラスにおけるのと同様です。

```
class mailbox.MHMailbox (dirname[, factory])
```

数字で名前のつけられた別々のファイルに個々のメッセージを収めたディレクトリである、MH メールボックスにアクセスします。メールボックスディレクトリの名前は *dirname* で渡します。 *factory* は *UnixMailbox* クラスにおけるのと同様です。

```
class mailbox.BabylMailbox (fp[, factory])
```

MMDf メールボックスと似ている、Babyl メールボックスにアクセスします。Babyl 形式では、各メッセージは二つのヘッダからなるセット、 *original* ヘッダおよび *visible* ヘッダを持っています。original ヘッダは '*** EOOH ***' (End-Of-Original-Headers) だけを含む行の前にあり、visible ヘッダは EOOH 行の後にあります。Babyl 互換のメールリーダーは visible ヘッダのみを表示し、 *BabylMailbox* オブジェクトは visible ヘッダのみを含むようなメッセージを返します。メールメッセージは EOOH 行で始まり、 '\037\014' だけを含む行で終わります。 *factory* は *UnixMailbox* クラスにおけるのと同様です。

古いメールボックスクラスを撤廃された *rfc822* モジュールではなく、 *email* モジュールと使いたければ、以下のようにできます:

```
import email
import email.Errors
import mailbox

def msgfactory(fp):
    try:
        return email.message_from_file(fp)
    except email.Errors.MessageParseError:
        # Don't return None since that will
        # stop the mailbox iterator
        return ''

mbox = mailbox.UnixMailbox(fp, msgfactory)
```

一方、メールボックス内には正しい形式の MIME メッセージしか入っていないと分かっているのなら、単に以下のようにします:

```
import email
import mailbox

mbox = mailbox.UnixMailbox(fp, email.message_from_file)
```

18.4.5 例

メールボックス中の面白そうなメッセージのサブジェクトを全て印字する簡単な例:

```
import mailbox
for message in mailbox.mbox('~/.mbox'):
    subject = message['subject'] # Could possibly be None.
    if subject and 'python' in subject.lower():
        print subject
```

Babyl メールボックスから MH メールボックスへ全てのメールをコピーし、変換可能な全ての形式固有の情報を変換する:

```
import mailbox
destination = mailbox.MH('~Mail')
destination.lock()
for message in mailbox.Babyl('~RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

この例は幾つかのメーリングリストのメールをソートするものです。他のプログラムと平行して変更を加えることでメールが破損したり、プログラムを中断することでメールを失ったり、はたまた半端なメッセージがメールボックス中にあることで途中で終了してしまう、といったことを避けるように注意深く扱っています:

```
import mailbox
import email.errors

list_names = ('python-list', 'python-dev', 'python-bugs')

boxes = dict((name, mailbox.mbox('~email/%s' % name)) for name in list_names)
inbox = mailbox.Maildir('~Maildir', factory=None)

for key in inbox.iterkeys():
    try:
        message = inbox[key]
    except email.errors.MessageParseError:
        continue           # The message is malformed. Just leave it.

    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            # Get mailbox to use
            box = boxes[name]

            # Write copy to disk before removing original.
            # If there's a crash, you might duplicate a message, but
            # that's better than losing a message completely.
            box.lock()
            box.add(message)
            box.flush()
            box.unlock()

            # Remove original message
            inbox.lock()
            inbox.discard(key)
            inbox.flush()
            inbox.unlock()
            break           # Found destination, so stop looking.

for box in boxes.itervalues():
    box.close()
```


18.5 `mhlib` — MH のメールボックスへのアクセス機構

バージョン 2.6 で非推奨: `mhlib` モジュールは Python 3 で削除されました。代わりに `mailbox` モジュールを使ってください。

`mhlib` モジュールは MH フォルダおよびその内容に対する Python インタフェースを提供します。

このモジュールには、あるフォルダの集まりを表現する `MH`、単一のフォルダを表現する `Folder`、単一のメッセージを表現する `Message`、の 3 つのクラスが入っています。

```
class mhlib.MH([path[, profile]])
```

`MH` は MH フォルダの集まりを表現します。

```
class mhlib.Folder(mh, name)
```

`Folder` クラスは単一のフォルダとフォルダ内のメッセージ群を表現します。

```
class mhlib.Message(folder, number[, name])
```

`Message` オブジェクトはフォルダ内の個々のメッセージを表現します。メッセージクラスは `mimertools.Message` から派生しています。

18.5.1 MH オブジェクト

`MH` インスタンスは以下のメソッドを持っています:

```
MH.error(format[, ...])
```

エラーメッセージを出力します – 上書きすることができます。

```
MH.getprofile(key)
```

プロファイルエントリ (設定されていなければ `None`) を返します。

```
MH.getpath()
```

メールボックスのパス名を返します。

```
MH.getcontext()
```

現在のフォルダ名を返します。

```
MH.setcontext(name)
```

現在のフォルダ名を設定します。

```
MH.listfolders()
```

トップレベルフォルダのリストを返します。

```
MH.listallfolders()
```

全てのフォルダのリストを返します。

```
MH.listsubfolders(name)
```

指定したフォルダの直下にあるサブフォルダのリストを返します。

```
MH.listallsubfolders(name)
```

指定したフォルダの下にある全てのサブフォルダのリストを返します。

`MH.makefolder (name)`

新しいフォルダを生成します。

`MH.deletefolder (name)`

フォルダを削除します – サブフォルダが入っているはいけません。

`MH.openfolder (name)`

新たな開かれたフォルダオブジェクトを返します。

18.5.2 Folder オブジェクト

`Folder` インスタンスは開かれたフォルダを表現し、以下のメソッドを持っています:

`Folder.error (format[, ...])`

エラーメッセージを出力します – 上書きすることができます。

`Folder.getfullname ()`

フォルダの完全なパス名を返します。

`Folder.getsequencesfilename ()`

フォルダ内のシーケンスファイルの完全なパス名を返します。

`Folder.getmessagefilename (n)`

フォルダ内のメッセージ *n* の完全なパス名を返します。

`Folder.listmessages ()`

フォルダ内のメッセージの (番号の) リストを返します。

`Folder.getcurrent ()`

現在のメッセージ番号を返します。

`Folder.setcurrent (n)`

現在のメッセージ番号を *n* に設定します。

`Folder.parsesequence (seq)`

msgs 文を解釈して、メッセージのリストにします。

`Folder.getlast ()`

最新のメッセージを取得します。メッセージがフォルダにない場合には 0 を返します。

`Folder.setlast (n)`

最新のメッセージを設定します (内部使用のみ)。

`Folder.getsequences ()`

フォルダ内のシーケンスからなる辞書を返します。シーケンス名がキーとして使われ、値はシーケンスに含まれるメッセージ番号のリストになります。

`Folder.putsequences (dict)`

フォルダ内のシーケンスからなる辞書 name: list を返します。

`Folder.remove_messages(list)`

リスト中のメッセージをフォルダから削除します。

`Folder.refile_messages(list, tofolder)`

リスト中のメッセージを他のフォルダに移動します。

`Folder.move_message(n, tofolder, ton)`

一つのメッセージを他のフォルダの指定先に移動します。

`Folder.copy_message(n, tofolder, ton)`

一つのメッセージを他のフォルダの指定先にコピーします。

18.5.3 Message オブジェクト

`Message` クラスは `mimertools.Message` のメソッドに加え、一つメソッドを持っています:

`Message.open_message(n)`

新たな開かれたメッセージオブジェクトを返します (ファイル記述子を一つ消費します)。

18.6 mimertools — MIME メッセージを解析するためのツール

バージョン 2.3 で非推奨: `email` パッケージを `mimertools` モジュールより優先して使うべきです。このモジュールは、下位互換性維持のためにのみ存在しています。Python 3.x では削除されています。

このモジュールは、`rfc822` モジュールの `Message` クラスのサブクラスと、マルチパート MIME や符合理化メッセージの操作に役に立つ多くのユーティリティ関数を定義しています。

このモジュールでは以下の内容を定義しています:

`class mimertools.Message(fp[, seekable])`

`Message` クラスの新しいインスタンスを返します。これは、`rfc822.Message` クラスのサブクラスで、いくつかの追加のメソッドがあります (以下を参照のこと)。`seekable` 引数は、`rfc822.Message` のものと同じ意味を持ちます。

`mimertools.choose_boundary()`

パートの境界として使うことができる見込みが高いユニークな文字列を返します。その文字列は、`'hostipaddr.uid.pid.timestamp.random'` の形をしています。

`mimertools.decode(input, output, encoding)`

オープンしたファイルオブジェクト `input` から、許される MIME `encoding` を使って符号化されたデータを読んで、オープンされたファイルオブジェクト `output` に復号化されたデータを書きます。`encoding` に許される値は、`'base64'`, `'quoted-printable'`, `'uuencode'`, `'x-uuencode'`, `'uue'`, `'x-uue'`, `'7bit'`, および `'8bit'` です。`'7bit'` あるいは `'8bit'` で符号化されたメッセージを復号化しても何も効果がありません。入力が出力に単純にコピーされるだけです。

`mimertools.encode(input, output, encoding)`

オープンしたファイルオブジェクト `input` からデータを読んで、それを許される MIME `encoding` を

使って符号化して、オープンしたファイルオブジェクト *output* に書きます。 *encoding* に許される値は、 *decode()* のものと同じです。

`mimetools.copyliteral(input, output)`

オープンしたファイル *input* から行を EOF まで読んで、それらをオープンしたファイル *output* に書きます。

`mimetools.copybinary(input, output)`

オープンしたファイル *input* からブロックを EOF まで読んで、それらをオープンしたファイル *output* に書きます。ブロックの大きさは現在 8192 に固定されています。

参考:

email モジュール 包括的な e-mail 処理パッケージです。 *mimetools* に取って代わります。

rfc822 モジュール *mimetools.Message* のベースクラスを提供しています。

multifile モジュール MIME データのような、別個のパーツを含むファイルの読み込みをサポート。

<http://faqs.cs.uu.nl/na-dir/mail/mime-faq/.html> MIME でよく訊ねられる質問と回答。MIME の概要に関しては、この文書の Part 1 の質問 1.1 への回答参照。

18.6.1 Message オブジェクトの追加メソッド

Message クラスは、 *rfc822.Message* メソッドに加えて、以下のメソッドを定義しています:

`Message.getplist()`

Content-Type ヘッダのパラメータリストを返します。これは文字列のリストです。 *key=value* の形のパラメータに対しては、 *key* は小文字に変換されますが、 *value* は変換されません。たとえば、もしメッセージに、ヘッダ *Content-type: text/html; spam=1; Spam=2; Spam* が含まれていれば、 *getplist()* は、Python リスト `['spam=1', 'spam=2', 'Spam']` を返すでしょう。

`Message.getparam(name)`

与えられた *name* の (*name=value* の形に対して *getplist()* が返す) 第 1 パラメータの *value* を返します。もし *value* が、 `'<...>'` あるいは `'"..."'` のように引用符で囲まれていれば、これらは除去されます。

`Message.getencoding()`

Content-Transfer-Encoding メッセージヘッダで指定された符号化方式を返します。もしそのようなヘッダが存在しなければ、 `'7bit'` を返します。符号化方式文字列は小文字に変換されます。

`Message.gettype()`

Content-Type ヘッダで指定された (*type/subtype* の形での) メッセージタイプを返します。もしそのようなヘッダが存在しなければ、 `'text/plain'` を返します。タイプ文字列は小文字に変換されます。

`Message.getmaintype()`

Content-Type ヘッダで指定されたメインタイプを返します。もしそのようなヘッダが存在しなければ、 `'text'` を返します。メインタイプ文字列は小文字に変換されます。

`Message.getsubtype()`

Content-Type ヘッダで指定されたサブタイプを返します。もしそのようなヘッダが存在しなければ、`'plain'` を返します。サブタイプ文字列は小文字に変換されます。

18.7 mimetypes — ファイル名を MIME 型へマップする

ソースコード: [Lib/mimetypes.py](#)

`mimetypes` モジュールは、ファイル名あるいは URL と、ファイル名拡張子に関連付けられた MIME 型とを変換します。ファイル名から MIME 型へと、MIME 型からファイル名拡張子への変換が提供されます; 後者の変換では符号化方式はサポートされていません。

このモジュールは、一つのクラスと多くの便利な関数を提供します。これらの関数がこのモジュールへの標準のインターフェースですが、アプリケーションによっては、そのクラスにも関係するかもしれません。

以下で説明されている関数は、このモジュールへの主要なインターフェースを提供します。たとえモジュールが初期化されていなくても、もしこれらの関数が、`init()` がセットアップする情報に依存していれば、これらの関数は、`init()` を呼びます。

`mimetypes.guess_type(url, strict=True)`

`url` で与えられるファイル名あるいは URL に基づいて、ファイルの型を推定します。戻り値は、タプル (`type`, `encoding`) です、ここで `type` は、もし型が (拡張子がないあるいは未定義のため) 推定できない場合は、`None` を、あるいは、MIME *content-type* ヘッダ に利用できる、`'type/subtype'` の形の文字列です。

`encoding` は、符号化方式がない場合は `None` を、あるいは、符号化に使われるプログラムの名前 (たとえば、`compress` あるいは `gzip`) です。符号化方式は *Content-Encoding* ヘッダとして使うのに適しており、*Content-Transfer-Encoding* ヘッダには適していません。マッピングはテーブルドリブンです。符号化方式のサフィックスは大/小文字を区別します; データ型サフィックスは、最初大/小文字を区別して試し、それから大/小文字を区別せずに試します。

省略可能な `strict` 引数は、既知の MIME 型のリストとして認識されるものが、IANA に登録された 正式な型のみに限定されるかどうかを指定するフラグです。`strict` が `True` (デフォルト) の時は、IANA 型のみがサポートされます; `strict` が `False` のときは、いくつかの追加の、非標準ではあるが、一般的に使用される MIME 型も認識されます。

`mimetypes.guess_all_extensions(type, strict=True)`

`type` で与えられる MIME 型に基づいてファイルの拡張子を推定します。戻り値は、先頭のドット (`'.'`) を含む、可能なファイル拡張子すべてを与える文字列のリストです。拡張子と特別なデータストリームとの関連付けは保証されませんが、`guess_type()` によって MIME 型 `type` とマップされます。

省略可能な `strict` 引数は `guess_type()` 関数のものと同じ意味を持ちます。

`mimetypes.guess_extension(type, strict=True)`

`type` で与えられる MIME 型に基づいてファイルの拡張子を推定します。戻り値は、先頭のドット (`'.'`) を含む、ファイル拡張子を与える文字列のリストです。拡張子と特別なデータストリームとの関連付け

は保証されませんが、`guess_type()` によって MIME 型 `type` とマップされます。もし `type` に対して拡張子が推定できない場合は、`None` が返されます。

省略可能な `strict` 引数は `guess_type()` 関数のものと同じ意味を持ちます。

モジュールの動作を制御するために、いくつかの追加の関数とデータ項目が利用できます。

`mimetypes.init(files=None)`

内部のデータ構造を初期化します。もし `files` が与えられていれば、これはデフォルトの `type map` を増やすために使われる、一連のファイル名でなければなりません。もし省略されていれば、使われるファイル名は `knownfiles` から取られます。Windows であれば、現在のレジストリの設定が読み込まれます。`files` あるいは `knownfiles` 内の各ファイル名は、それ以前に現れる名前より優先されます。繰り返し `init()` を呼び出すことは許されています。

`files` に空リストを与えることで、システムのデフォルトが適用されるのを避けることが出来ます; 組み込みのリストから well-known な値だけが取り込まれます。

バージョン 2.7 で変更: 前のバージョンでは、Windows のレジストリの設定は無視されていました。

`mimetypes.read_mime_types(filename)`

ファイル `filename` で与えられた型のマップが、もしあればロードします。型のマップは、先頭の dot ('.') を含むファイル名拡張子を、'type/subtype' の形の文字列にマッピングする辞書として返されます。もしファイル `filename` が存在しないか、読み込めなければ、`None` が返されます。

`mimetypes.add_type(type, ext, strict=True)`

MIME 型 `type` からのマッピングを拡張子 `ext` に追加します。拡張子がすでに既知であれば、新しい型が古いものに置き替わります。その型がすでに既知であれば、その拡張子が、既知の拡張子のリストに追加されます。

`strict` が `True` の時 (デフォルト) は、そのマッピングは正式な MIME 型に、そうでなければ、非標準の MIME 型に追加されます。

`mimetypes.inited`

グローバルなデータ構造が初期化されているかどうかを示すフラグ。これは `init()` により `True` に設定されます。

`mimetypes.knownfiles`

共通にインストールされた型マップファイル名のリスト。これらのファイルは、普通 `mime.types` という名前であり、パッケージごとに異なる場所にインストールされます。

`mimetypes.suffix_map`

サフィックスをサフィックスにマップする辞書。これは、符号化方式と型が同一拡張子で示される符号化ファイルが認識できるように使用されます。例えば、`.tgz` 拡張子は、符号化と型が別個に認識できるように `.tar.gz` にマップされます。

`mimetypes.encodings_map`

ファイル名拡張子を符号化方式型にマッピングする辞書。

`mimetypes.types_map`

ファイル名拡張子を MIME 型にマップする辞書。

`mimetypes.common.types`

ファイル名拡張子を非標準ではあるが、一般に使われている MIME 型にマップする辞書。

モジュールの使用例:

```
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

18.7.1 Mime 型オブジェクト

MimeTypes クラスは一つ以上の MIME 型データベースが欲しいアプリケーションにとって有用でしょう。これは *mimetypes* モジュールのそれと似たインターフェースを提供します。

class `mimetypes.MimeTypes` (*filenames=()*, *strict=True*)

このクラスは、MIME-型データベースを表現します。デフォルトでは、このモジュールの他のものと同じデータベースへのアクセスを提供します。初期データベースは、このモジュールによって提供されるもののコピーで、追加の `mime.types` -形式のファイルを、`read()` あるいは `readfp()` メソッドを使って、データベースにロードすることで拡張されます。マッピング辞書も、もしデフォルトのデータが望むものでなければ、追加のデータをロードする前にクリアされます。

省略可能な *filenames* パラメータは、追加のファイルを、デフォルトデータベースの”トップに”ロードさせるのに使うことができます。

suffix_map

サフィックスをサフィックスにマップする辞書。これは、符号化方式と型が同一拡張子で示されるような符号化ファイルが認識できるように使用されます。例えば、`.tgz` 拡張子は、符号化方式と型が別個に認識できるように `.tar.gz` に対応づけられます。これは、最初はモジュールで定義されたグローバルな *suffix_map* のコピーです。

encodings_map

ファイル名拡張子を符号化型にマッピングする辞書。これは、最初はモジュールで定義されたグローバルな *encodings_map* のコピーです。

types_map

ファイル名拡張子を MIME 型にマッピングする 2 種類の辞書のタプル; 最初の辞書は非標準型、二つ目は標準型の辞書です。初期状態ではそれぞれ *common.types* と *types_map* です。

types_map_inv

MIME 型をファイル名拡張子のリストにマッピングする 2 種類の辞書のタプル; 最初の辞書は非標準型、二つ目は標準型の辞書です。初期状態ではそれぞれ *common.types* と *types_map* です。

guess.extension (*type*, *strict=True*)

guess.extension() 関数と同様ですが、オブジェクトに保存されたテーブルを使用します。

guess.type (*url*, *strict=True*)

guess.type() 関数と同様ですが、オブジェクトに保存されたテーブルを使用します。

guess.all_extensions (*type*, *strict=True*)

guess.all_extensions() と同様ですが、オブジェクトに保存されたテーブルを参照します。

read (*filename*, *strict=True*)

MIME 情報を、*filename* という名のファイルからロードします。これはファイルを解析するのに *readfp()* を使用します。

strict が *True* の時 (デフォルト) は、そのマッピングは標準 MIME 型のリストに、そうでなければ、非標準 MIME 型のリストに追加されます。

readfp (*fp*, *strict=True*)

MIME 型情報を、オープンしたファイル *fp* からロードします。ファイルは、標準の `mime.types` ファイルの形式でなければなりません。

strict が *True* の時 (デフォルト) は、そのマッピングは標準 MIME 型のリストに、そうでなければ、非標準 MIME 型のリストに追加されます。

read.windows_registry (*strict=True*)

MIME type 情報を Windows のレジストリから読み込みます。Windows でのみ利用できます。

strict が *True* の時 (デフォルト) は、そのマッピングは標準 MIME 型のリストに、そうでなければ、非標準 MIME 型のリストに追加されます。

バージョン 2.7 で追加.

18.8 MimeWriter — 汎用 MIME ファイルライター

バージョン 2.3 で非推奨: *email* パッケージを、*MimeWriter* モジュールよりも優先して使用すべきです。このモジュールは、下位互換性維持のためだけに存在します。

このモジュールは、クラス *MimeWriter* を定義します。この *MimeWriter* クラスは、MIME マルチパートファイルを作成するための基本的なフォーマットを実装します。これは出力ファイル内をあちこち移動することも、大量のバッファスペースを使うこともありません。最終的にファイルに並ぶべき順番に、パートを書かなければなりません。 *MimeWriter* は、あなたが追加するヘッダをバッファして、それらの順番を並び替えることができるようにします。

class *MimeWriter.MimeWriter* (*fp*)

MimeWriter クラスの新しいインスタンスを返します。渡される唯一の引数 *fp* は、出力先ファイルオブジェクトです。 *StringIO* オブジェクトも使えることに注意して下さい。

18.8.1 MimeWriter オブジェクト

`MimeWriter` インスタンスには以下のメソッドがあります:

`MimeWriter.addheader(key, value[, prefix])`

MIME メッセージに新しいヘッダ行を追加します。 *key* は、そのヘッダの名前であり、そして *value* で、そのヘッダの値を明示的に与えます。省略可能な引数 *prefix* は、ヘッダが挿入される場所を決定します; 0 は最後に追加することを意味し、1 は先頭への挿入です。デフォルトは最後に追加することです。

`MimeWriter.flushheaders()`

今まで集められたヘッダすべてが書かれ (そして忘れられ) るようにします。これは、もし全く本体が必要でない場合に役に立ちます。例えば、ヘッダのような情報を保管するために (誤って) 使用された、型 *message/rfc822* のサブパート用など。

`MimeWriter.startbody(ctype[, plist[, prefix]])`

メッセージの本体に書くのに使用できるファイルのようなオブジェクトを返します。 *content-type* は与えられた *ctype* に設定され、省略可能なパラメータ *plist* は、 *content-type* 宣言のための追加のパラメータを与えます。 *prefix* は、そのデフォルトが先頭への挿入であること以外は *addheader()* のように働きます。

`MimeWriter.startmultipartbody(subtype[, boundary[, plist[, prefix]]])`

メッセージ本体を書くのに使うことができるファイルのようなオブジェクトを返します。更に、このメソッドはマルチパートのコードを初期化します。ここで、 *subtype* が、そのマルチパートのサブタイプを、 *boundary* がユーザ定義の境界指定を、そして *plist* が、そのサブタイプ用の省略可能なパラメータを定義します。 *prefix* は、 *startbody()* のように働きます。サブパートは、 *nextpart()* を使って作成するべきです。

`MimeWriter.nextpart()`

マルチパートメッセージの個々のパートを表す、 *MimeWriter* の新しいインスタンスを返します。これは、そのパートを書くのにも、また複雑なマルチパートを再帰的に作成するのにも使えます。メッセージは、 *nextpart()* を使う前に最初に *startmultipartbody()* で初期化しなければなりません。

`MimeWriter.lastpart()`

これは、マルチパートメッセージの最後のパートを指定するのに使うことができ、マルチパートメッセージを書くときはいつでも使うべきです。

18.9 mimify — 電子メールメッセージの MIME 処理

バージョン 2.3 で非推奨: *mimify* モジュールを使うよりも *email* パッケージを使うべきです。このモジュールは以前のバージョンとの互換性のために保守されているにすぎません。

mimify モジュールでは電子メールメッセージから MIME へ、および MIME から電子メールメッセージへの変換を行うための二つの関数を定義しています。電子メールメッセージは単なるメッセージでも、MIME 形式でもかまいません。各パートは個別に扱われます。メッセージ (の一部) の MIME 化 (*mimify*) の際、7 ビット ASCII 文字を使って表現できない何らかの文字が含まれていた場合、メッセージの *quoted-printable* への符

号化が伴います。メッセージが送信される前に編集しなければならない場合、MIME 化および非 MIME 化は特に便利です。典型的な使用法は以下のようになります:

```
unmimify message
edit message
mimify message
send message
```

モジュールでは以下のユーザから呼び出し可能な関数と、ユーザが設定可能な変数を定義しています:

`mimify.mimify(infile, outfile)`

infile を *outfile* にコピーします。その際、パートを quoted-printable に変換し、必要なら MIME メールヘッダを追加します。 *infile* および *outfile* はファイルオブジェクト (実際には、`readline()` メソッドを持つ (*infile*) か、`write()` (*outfile*) メソッドを持つあらゆるオブジェクト) か、ファイル名を指す文字列を指定することができます。 *infile* および *outfile* が両方とも文字列の場合、同じ値にすることができます。

`mimify.unmimify(infile, outfile[, decode_base64])`

infile を *outfile* にコピーします。その際、全ての quoted-printable 化されたパートを復号化します。 *infile* および *outfile* はファイルオブジェクト (実際には、`readline()` メソッドを持つ (*infile*) か、`write()` (*outfile*) メソッドを持つあらゆるオブジェクト) か、ファイル名を指す文字列を指定することができます。 *infile* および *outfile* が両方とも文字列の場合、同じ値にすることができます。 *decode_base64* 引数が与えられており、その値が真である場合、base64 符号で符号化されているパートも同様に復号化されます。

`mimify.mime_decode_header(line)`

line 内の符号化されたヘッダ行が復号化されたものを返します。ISO 8859-1 文字セット (Latin-1) だけをサポートします。

`mimify.mime_encode_header(line)`

line 内のヘッダ行が MIME 符号化されたものを返します。

`mimify.MAXLEN`

デフォルトでは、非 ASCII 文字 (8 ビット目がセットされている文字) を含むか、`MAXLEN` 文字 (デフォルトの値は 200 です) よりも長い部分は quoted-printable 形式で符号化されます。

`mimify.CHARSET`

文字セットがメールヘッダで指定されていない場合指定しなければなりません。使われている文字セットを表す文字列は `CHARSET` に記憶されます。デフォルトの値は ISO-8859-1 (Latin1 (latin-one)) としても知られています。

このモジュールはコマンドラインから利用することもできます。以下のような使用法:

```
mimify.py -e [-l length] [infile [outfile]]
mimify.py -d [-b] [infile [outfile]]
```

で、それぞれ符号化 (mimify) および復号化 (unmimify) を行います。標準の設定では *infile* は標準入力で、*putfile* は標準出力です。入出力に同じファイルを指定することもできます。

符号化の際に `-l` オプションを与えた場合、`length` で指定した長さより長い行があれば、長い部分の内容が符号化されます。

復号化の際に `-b` オプションが与えられていれば、base64 パートも同様に復号化されます。

参考:

`quopri` モジュール MIME quoted-printable 形式ファイルのエンコードおよびデコード。

18.10 `multifile` — 個別の部分を含んだファイル群のサポート

バージョン 2.5 で非推奨: `multifile` モジュールよりも `email` パッケージを使うべきです。このモジュールは後方互換性のためだけに存在しています。

`MultiFile` オブジェクトは、テキストファイルのセクション分割をファイル類似の入力オブジェクトのように扱えるようにします。指定したデリミタのパターンに遭遇すると `readline()` が `''` を返します。このクラスの標準設定は MIME マルチパートメッセージを解釈する上で便利となるように設計されていますが、サブクラス化を行って幾つかのメソッドを上書きすることで、簡単に汎用目的に対応させることができます。

class `multifile.MultiFile` (`fp`[, `seekable`])

マルチファイルを生成します。`fp` 引数には、例えば `open()` が返すファイルオブジェクトのような、`MultiFile` インスタンスが行データを取得出来るオブジェクトを渡す必要があります。

`MultiFile` は入力オブジェクトの `readline()`、`seek()`、および `tell()` メソッドしか参照せず、後者の二つのメソッドは個々の MIME パートにランダムアクセスしたい場合にのみ必要です。`MultiFile` を `seek` できないストリームオブジェクトで使うには、オプションの `seekable` 引数の値を偽にしてください; これにより、入力オブジェクトの `seek()` および `tell()` メソッドを使わないようになります。

`MultiFile` の視点から見ると、テキストは三種類の行データ: データ、セクション分割子、終了マーカ、からなることを知っていると役に立つでしょう。`MultiFile` は、多重入れ子構造になっている可能性のある、それぞれが独自のセクション分割子および終了マーカのパターンを持つメッセージパートをサポートするように設計されています。

参考:

`email` モジュール 包括的な e-mail 処理パッケージです。`multifile` に取って代わります。

18.10.1 `MultiFile` オブジェクト

`MultiFile` インスタンスは以下のメソッドを持っています:

`MultiFile.readline` (`str`)

一行データを読みます。その行が (セクション分割子や終了マーカや本物の EOF でない) データの場合、行データを返します。その行がもっとも最近スタックにプッシュされた境界パターンにマッチした場合、`''` を返し、マッチした内容が終了マーカかそうでないかによって `self.last` を 1 か 0 に設定します。行がその他のスタックされている境界パターンにマッチした場合、エラーが送出されます。

背後のストリームオブジェクトにおけるファイルの終端に到達した場合、全ての境界がスタックから除去されていない限りこのメソッドは `Error` を送出します。

`MultiFile.readlines(str)`

このパートの残りの全ての行を文字列のリストとして返します。

`MultiFile.read()`

次のセクションまでの全ての行を読みます。読んだ内容を単一の (複数行にわたる) 文字列として返します。このメソッドには `size` 引数をとらないので注意してください!

`MultiFile.seek(pos[, whence])`

ファイルを `seek` します。 `seek` する際のインデックスは現在のセクションの開始位置からの相対位置になります。 `pos` および `whence` 引数はファイルの `seek` における引数と同じように解釈されます。

`MultiFile.tell()`

現在のセクションの先頭に対して相対的なファイル位置を返します。

`MultiFile.next()`

次のセクションまで行を読み飛ばします (すなわち、セクション分割子または終了マーカが消費されるまで行データを読みます)。次のセクションがあった場合には真を、終了マーカが発見された場合には偽を返します。最も最近スタックにプッシュされた境界パターンを最有効化します。

`MultiFile.is_data(str)`

`str` がデータの場合に真を返し、セクション分割子の可能性がある場合には偽を返します。このメソッドは行の先頭が (全ての MIME 境界が持っている) `'--'` 以外になっているかを調べるように実装されていますが、派生クラスで上書きできるように宣言されています。

このテストは実際の境界テストにおいて高速性を保つために使われているので注意してください; このテストが常に `false` を返す場合、テストが失敗するのではなく、単に処理が遅くなるだけです。

`MultiFile.push(str)`

境界文字列をスタックにプッシュします。この境界文字列の修飾されたバージョンが入力行に見つかった場合、セクション分割子または終了マーカであると解釈されます (どちらであるかは修飾に依存します。 [RFC 2045](#) を参照してください)。それ以降の全てのデータ読み出しは、 `pop()` を呼んで境界文字列を除去するか、 `next()` を呼んで境界文字列を再有効化しないかぎり、ファイル終端を示す空文字列を返します。

一つ以上の境界をプッシュすることは可能です。もっとも最近プッシュされた境界に遭遇すると EOF が返ります; その他の境界に遭遇するとエラーが送出されます。

`MultiFile.pop()`

セクション境界をポップします。この境界はもはや EOF として解釈されません。

`MultiFile.section_divider(str)`

境界をセクション分割子にします。標準では、このメソッドは (全ての MIME 境界が持っている) `'--'` を境界文字列の先頭に追加しますが、これは派生クラスで上書きできるように宣言されています。末尾の空白は無視されることから考えて、このメソッドでは LF や CR-LF を追加する必要はありません。

`MultiFile.end_marker(str)`

境界文字列を終了マーカ行にします。標準では、このメソッドは (MIME マルチパートデータのメッ

セージ終了マーカのように) '---' を境界文字列の先頭に追加し、かつ '---' を境界文字列の末尾に追加しますが、これは派生クラスで上書きできるように宣言されています。末尾の空白は無視されることから考えて、このメソッドでは LF や CR-LF を追加する必要はありません。

最後に、`MultiFile` インスタンスは二つの公開されたインスタンス変数を持っています:

`MultiFile.level`

現在のパートにおける入れ子の深さです。

`MultiFile.last`

最後に見つかったファイル終了イベントがメッセージ終了マーカであった場合に真となります。

18.10.2 `MultiFile` の例

```
import mimetools
import multifile
import StringIO

def extract_mime_part_matching(stream, mimetype):
    """Return the first element in a multipart MIME message on stream
    matching mimetype."""

    msg = mimetools.Message(stream)
    msgtype = msg.gettype()
    params = msg.getplist()

    data = StringIO.StringIO()
    if msgtype[:10] == "multipart/":

        file = multifile.MultiFile(stream)
        file.push(msg.getparam("boundary"))
        while file.next():
            submsg = mimetools.Message(file)
            try:
                data = StringIO.StringIO()
                mimetools.decode(file, data, submsg.getencoding())
            except ValueError:
                continue
            if submsg.gettype() == mimetype:
                break
        file.pop()
    return data.getvalue()
```

18.11 `rfc822` — RFC 2822 準拠のメールヘッダ読み出し

バージョン 2.3 で非推奨: `rfc822` モジュールを使うよりも `email` パッケージを使うべきです。このモジュールは以前のバージョンとの互換性のために保守されているにすぎません。Python 3 ではこのモジュールは削除されました。

このモジュールでは、インターネット標準 [RFC 2822](#)^{*1} で定義されている ”電子メールメッセージ” を表現するクラス、`Message` を定義しています。このメッセージはメッセージヘッダ群とメッセージボディの集まりからなります。このモジュールではまた、ヘルパークラス [RFC 2822](#) アドレス群を解釈するための `AddressList` クラスを定義しています。RFC 2822 メッセージ固有の構文に関する情報は RFC を参照してください。

`mailbox` モジュールでは、多くのエンドユーザメールプログラムによって生成されるメールボックスを読み出すためのクラスを提供しています。

```
class rfc822.Message (file[, seekable ])
```

`Message` インスタンスは入力オブジェクトをパラメタに与えてインスタンス化します。入力オブジェクトのメソッドのうち、`Message` が依存するのは `readline()` だけです; 通常のファイルオブジェクトは適格です。インスタンス化を行うと、入力オブジェクトからデリミタ行 (通常は空行 1 行) に到達するまでヘッダを読み出し、それらをインスタンス中に保持します。ヘッダの後のメッセージ本体は読み出しません。

このクラスは `readline()` メソッドをサポートする任意の入力オブジェクトを扱うことができます。入力オブジェクトが `seek` および `tell` できる場合、`rewindbody()` メソッドが動作します。また、不正な行データを入力ストリームにプッシュバックできます。入力オブジェクトが `seek` できない一方で、入力行をプッシュバックする `unread()` メソッドを持っている場合、`Message` は不正な行データにこのプッシュバックを使います。こうして、このクラスはバッファされているストリームから来るメッセージを解釈するのに使うことができます。

オプションの `seekable` 引数は、`lseek()` システムコールが動作しないと分かるまでは `tell()` がバッファされたデータを無視するような、ある種の `stdio` ライブラリでの回避手段として提供されています。可搬性を最大にするために、`socket` オブジェクトによって生成されたファイルのような、`seek` できないオブジェクトを渡す際には、最初に `tell()` が呼び出されないようにするために `seekable` 引数をゼロに設定すべきです。

ファイルとして読み出された入力行データは CR-LF と単一の改行 (line feed) のどちらで終端されていてもかまいません; 行データを記憶する前に、終端の CR-LF は単一の改行と置き換えられます。

ヘッダに対するマッチは全て大小文字に依存しません。例えば、`m['From']`、`m['from']`、および `m['FROM']` は全て同じ結果になります。

```
class rfc822.AddressList (field)
```

[RFC 2833](#) アドレスがカンマで区切られたものとして解釈できる単一の文字列パラメタを使って、`AddressList` ヘルパークラスをインスタンス化することができます。(パラメタ `None` は空のリストを表します。)

```
rfc822.quote (str)
```

`str` 中のバックスラッシュが 2 つのバックスラッシュに置き換えられ、二重引用符がバックスラッシュ付きの二重引用符に置き換えられた、新たな文字列を返します。

```
rfc822.unquote (str)
```

^{*1} このモジュールはもともと [RFC 822](#) に適合していたので、そういう名前になっています。その後、[RFC 2822](#) が [RFC 822](#) に対する更新としてリリースされました。このモジュールは [RFC 2822](#) 適合であり、特に [RFC 822](#) からの構文や意味付けに対する変更がなされています。

文字列 *str* を 逆クオート した新しい文字列を返します。もし *str* の先頭あるいは末尾がダブルクオート だった場合、これらは単に切り落とされます。同様にもし *str* の先頭あるいは末尾が角ブラケット (<, >) だった場合も切り落とされます。

`rfc822.parseaddr(address)`

To や *Cc* といった、アドレスが入っているフィールドの値 *address* を解析し、含まれている ”実名 (realname)” 部分および ”電子メールアドレス” 部分に分けます。それらの情報からなるタプルを返します。解析が失敗した場合には 2 要素のタプル (`None`, `None`) を返します。

`rfc822.dump_address_pair(pair)`

`parseaddr()` の逆で、実名と電子メールアドレスからなる 2 要素のタプル (`realname`, `email_address`) を引数にとり、*To* あるいは *Cc* ヘッダに適した形式の文字列を返します。タプル *pair* の第 1 要素が偽である場合、第 2 要素の値をそのまま返します。

`rfc822.parsedate(date)`

RFC 2822 の規則に従っている日付を解析しようと試みます。しかしながら、メーラによっては **RFC 2822** で指定されているような書式に従わないため、そのような場合には `parsedata()` は正しい日付を推測しようと試みます。*date* は 'Mon, 20 Nov 1995 19:12:08 -0500' のような **RFC 2822** 様式の日付を収めた文字列です。日付の解析に成功した場合、`parsedate()` は `time.mktime()` にそのまま渡すことができるような 9 要素のタプルを返します; そうでない場合には `None` を返します。結果のインデックス 6、7、および 8 は有用な情報ではありません。

`rfc822.parsedate_tz(date)`

`parsedate()` と同じ機能を実現しますが、`None` または 10 要素のタプルを返します; 最初の 9 要素は `time.mktime()` に直接渡すことができるようなタプルで、10 番目の要素はその日のタイムゾーンにおける UTC (グリニッチ標準時の公式名称) からのオフセットです。(タイムゾーンオフセットの符号は、同じタイムゾーンにおける `time.timezone` 変数の符号と反転しています; 後者の変数が POSIX 標準に従っている一方、このモジュールは **RFC 2822** に従っているからです。) 入力文字列がタイムゾーン情報を持たない場合、タプルの最後の要素は `None` になります。結果のインデックス 6、7、および 8 は有用な情報ではありません。

`rfc822.mktime_tz(tuple)`

`parsedata_tz()` が返す 10 要素のタプルを UTC タイムスタンプに変換します。タプル内のタイムゾーン要素が `None` の場合、地域の時刻を表しているものと仮定します。些細な欠陥: この関数はまず最初の 8 要素を地域における時刻として変換し、次にタイムゾーンの違いに対する補償を行います; これにより、夏時間の切り替え日前後でちょっとしたエラーが生じるかもしれません。通常の利用に関しては心配ありません。

参考:

email モジュール 包括的な e-mail 処理パッケージです。 *rfc822* に取って代わります。

mailbox モジュール エンドユーザのメールプログラムによって生成される、様々な mailbox 形式を読み出すためのクラス群。

mimertools モジュール MIME エンコードされたメッセージを処理する *rfc822.Message* のサブクラス。

18.11.1 Message オブジェクト

`Message` インスタンスは以下のメソッドを持っています:

`Message.rewindbody()`

メッセージ本体の先頭に seek します。このメソッドはファイルオブジェクトが seek 可能である場合にのみ動作します。

`Message.isheader(line)`

ある行が正しい **RFC 2822** ヘッダである場合、その行の正規化されたフィールド名 (インデックス指定の際に使われる辞書キー) を返します; そうでない場合 `None` を返します (解析をここで一度中断し、行データを入力ストリームに押し戻すことを意味します)。このメソッドをサブクラスで上書きすると便利ことがあります。

`Message.islast(line)`

与えられた `line` が `Message` の区切りとなるデリミタであった場合に真を返します。このデリミタ行は消費され、ファイルオブジェクトの読み位置はその直後になります。標準ではこのメソッドは単にその行が空行かどうかをチェックしますが、サブクラスで上書きすることもできます。

`Message.iscomment(line)`

与えられた行全体を無視し、単に読み飛ばすときに真を返します。標準では、これは控えメソッド (stub) であり、常に `False` を返しますが、サブクラスで上書きすることもできます。

`Message.getallmatchingheaders(name)`

`name` に一致するヘッダ全てで構成される行のリストを返します。各物理行は継続行であるか否かに関係なく別々のリスト要素になります。 `name` に一致するヘッダがない場合、空のリストを返します。

`Message.getfirstmatchingheader(name)`

`name` に一致する最初のヘッダと、その行に継続する (複数) 行からなる行データのリストを返します。 `name` に一致するヘッダがない場合 `None` を返します。

`Message.getrawheader(name)`

`name` に一致する最初のヘッダにおけるコロン以降のテキストが入った単一の文字列を返します。このテキストには、先頭の空白、末尾の改行、また継続の行がある場合には途中の改行と空白が含まれます。 `name` に一致するヘッダが存在しない場合には `None` を返します。

`Message.getheader(name[, default])`

`name` に一致する最後のヘッダから先頭および末尾の空白を剥ぎ取った単一の文字列を返します。途中にある空白は剥ぎ取られません。オプションの `default` 引数は、 `name` に一致するヘッダが存在しない場合に、別のデフォルト値を返すように指定するために使われます。デフォルトは `None` です。パースされたヘッダを得る方法としてはこれが好ましいでしょう。

`Message.get(name[, default])`

正規の辞書との互換性をより高めるための `getheader()` の別名です。

`Message.getaddr(name)`

`getheader(name)` が返した文字列を解析して、 (`full name`, `email address`) からなるペアを返します。 `name` に一致するヘッダが無い場合、 (`None`, `None`) が返されます; そうでない場合、 `full name` および `address` は (空文字列をとりうる) 文字列になります。

例: `m` に最初の *From* ヘッダに文字列 `'jack@cwil.nl (Jack Jansen)'` が入っている場合、`m.getaddr('From')` はペア `('Jack Jansen', 'jack@cwil.nl')` になります。また、`'Jack Jansen <jack@cwil.nl>'` であっても、全く同じ結果になります。

`Message.getaddrlist(name)`

`getaddr(list)` に似ていますが、複数のメールアドレスからなるリストが入ったヘッダ (例えば *To* ヘッダ) を解析し、`(full name, email address)` のペアからなるリストを (たとえヘッダには一つしかアドレスが入っていなかったとしても) 返します。`name` に一致するヘッダが無かった場合、空のリストを返します。

指定された名前に一致する複数のヘッダが存在する場合 (例えば、複数の *Cc* ヘッダが存在する場合)、全てのアドレスを解析します。指定されたヘッダが継続行で収められている場合も解析されます。

`Message.getdate(name)`

`getheader()` を使ってヘッダを取得して解析し、`time.mktime()` と互換な 9 要素のタプルにします; フィールド 6、7、および 8 は有用な値ではないので注意して下さい。`name` に一致するヘッダが存在しなかったり、ヘッダが解析不能であった場合、`None` を返します。

日付の解析は黒魔術のようなものであり、全てのヘッダが標準に従っているとは限りません。このメソッドは多くの発信源から集められた膨大な数の電子メールでテストされており、正しく動作することが分かっていますが、間違った結果を出力してしまう可能性はまだあります。

`Message.getdate_tz(name)`

`getheader()` を使ってヘッダを取得して解析し、10 要素のタプルにします; 最初の 9 要素は `time.mktime()` と互換性のあるタプルを形成し、10 番目の要素はその日におけるタイムゾーンの UTC からのオフセットを与える数字になります。フィールド 6、7、および 8 は有用な値ではないので注意して下さい。`getdate()` と同様に、`name` に一致するヘッダがなかったり、解析不能であった場合、`None` を返します。

`Message` インスタンスはまた、限定的なマップ型のインタフェースを持っています。すなわち: `m[name]` は `m.getheader(name)` に似ていますが、一致するヘッダがない場合 `KeyError` を送出します; `len(m)`、`m.get(name[, default])`、`name in m`、`m.keys()`、`m.values()`、`m.items()`、および `m.setdefault(name[, default])` は期待通りに動作します。ただし `setdefault()` は標準の設定値として空文字列をとります。`Message` インスタンスはまた、マップ型への書き込みを行えるインタフェース `m[name] = value` および `del m[name]` をサポートしています。`Message` オブジェクトでは、`clear()`、`copy()`、`popitem()`、あるいは `update()` といったマップ型インタフェースのメソッドはサポートしていません。(`get()` および `setdefault()` のサポートは Python 2.2 でしか追加されていません。)

最後に、`Message` インスタンスはいくつかの public なインスタンス変数を持っています:

`Message.headers`

ヘッダ行のセット全体が、(`setitem` を呼び出して変更されない限り) 読み出された順番に入れられたリストです。各行は末尾の改行を含んでいます。ヘッダを終端する空行はリストに含まれません。

`Message.fp`

インスタンス化の際に渡されたファイルまたはファイル類似オブジェクトです。この値はメッセージ本体を読み出すために使うことができます。

`Message.unixfrom`

メッセージに `Unix From` 行がある場合はその行、そうでなければ空文字列になります。この値は例えば `mbox` 形式のメールボックスファイルのような、あるコンテキスト中のメッセージを再生成するために必要です。

18.11.2 AddressList オブジェクト

`AddressList` インスタンスは以下のメソッドを持ちます:

`AddressList.__len__()`

アドレスリスト中のアドレスの数を返します。

`AddressList.__str__()`

アドレスリストの正規化 (canonicalize) された文字列表現を返します。アドレスはカンマで分割された "name" <host@domain> 形式になります。

`AddressList.__add__(alist)`

二つの `AddressList` 被演算子中の双方に含まれるアドレスについて、重複を除いた (集合和の) 全てのアドレスを含む新たな `AddressList` インスタンスを返します。

`AddressList.__iadd__(alist)`

`__add__()` のインプレース演算版です; `AddressList` インスタンスと右辺値 `alist` との集合和をとり、その結果をインスタンス自体と置き換えます。

`AddressList.__sub__(alist)`

左辺値の `AddressList` インスタンスのアドレスのうち、右辺値中に含まれていないもの全てを含む (集合差分の) 新たな `AddressList` インスタンスを返します。

`AddressList.__isub__(alist)`

`__sub__()` のインプレース演算版で、`alist` にも含まれているアドレスを削除します。

最後に、`AddressList` インスタンスは public なインスタンス変数を持つます:

`AddressList.addresslist`

アドレスあたり一つの文字列ペアで構成されるタプルからなるリストです。各メンバ中では、最初の要素は正規化された名前部分で、二つ目は実際の配送アドレス ('@' で分割されたユーザ名とホスト、ドメインからなるペア) です。

脚注

18.12 base64 — RFC 3548: Base16, Base32, Base64 データの符号化

このモジュールは、[RFC 3548](#) で定められた仕様によるデータの符号化 (エンコード、encoding) および復元 (デコード、decoding) を提供します。この RFC 標準では Base16, Base32 および Base64 が定義されており、これはバイナリ文字列とテキスト文字列とをエンコードあるいはデコードするためのアルゴリズムです。変換

されたテキスト文字列は email で確実に送信したり、URL の一部として使用したり、HTTP POST リクエストの一部に含めることができます。これらの符号化アルゴリズムは **uuencode** で使われているものとは別物です。

There are two interfaces provided by this module. The modern interface supports encoding and decoding string objects using both base-64 alphabets defined in [RFC 3548](#) (normal, and URL- and filesystem-safe). The legacy interface provides for encoding and decoding to and from file-like objects as well as strings, but only using the Base64 standard alphabet.

Python 2.4 で導入された現代的なインターフェイスは以下のものを提供します：

`base64.b64encode(s[, altchars])`

Base64 をつかって、文字列をエンコードします。

s はエンコードする文字列です。オプション引数 *altchars* は最低でも 2 の長さをもつ文字列で (これ以降の文字は無視されます)、これは + と / の代わりに使われる代替アルファベットを指定します。これにより、アプリケーションはたとえば URL やファイルシステムの影響をうけない Base64 文字列を生成することができます。デフォルトの値は `None` で、これは標準の Base64 アルファベット集合が使われることを意味します。

エンコードされた文字列が返されます。

`base64.b64decode(s[, altchars])`

Base64 文字列をデコード (復元) します。

s にはデコードする文字列を渡します。オプション引数の *altchars* は最低でも 2 の長さをもつ文字列で (これ以降の文字は無視されます)、これは + と / の代わりに使われる代替アルファベットを指定します。

The decoded string is returned. A `TypeError` is raised if *s* is incorrectly padded. Characters that are neither in the normal base-64 alphabet nor the alternative alphabet are discarded prior to the padding check.

`base64.standard_b64encode(s)`

標準の Base64 アルファベット集合をもちいて文字列 *s* をエンコード (符号化) します。

`base64.standard_b64decode(s)`

標準の Base64 アルファベット集合をもちいて文字列 *s* をデコード (復元) します。

`base64.urlsafe_b64encode(s)`

Encode string *s* using the URL- and filesystem-safe alphabet, which substitutes `-` instead of `+` and `_` instead of `/` in the standard Base64 alphabet. The result can still contain `=`.

`base64.urlsafe_b64decode(s)`

Decode string *s* using the URL- and filesystem-safe alphabet, which substitutes `-` instead of `+` and `_` instead of `/` in the standard Base64 alphabet.

`base64.b32encode(s)`

Base32 をつかって、文字列をエンコード (符号化) します。*s* にはエンコードする文字列を渡し、エンコードされた文字列が返されます。

`base64.b32decode(s[, casefold[, map01]])`

Base32 をつかって、文字列をデコード (復元) します。

s にはエンコードする文字列を渡します。オプション引数 *casefold* は小文字のアルファベットを受けつけるかどうかを指定します。セキュリティ上の理由により、デフォルトではこれは `False` になっています。

RFC 3548 は付加的なマッピングとして、数字の 0 (零) をアルファベットの O (オー) に、数字の 1 (壹) をアルファベットの I (アイ) または L (エル) に対応させることを許しています。オプション引数は *map01* は、`None` でないときは、数字の 1 をどの文字に対応づけるかを指定します (*map01* が `None` でないとき、数字の 0 はつねにアルファベットの O (オー) に対応づけられます)。セキュリティ上の理由により、これはデフォルトでは `None` になっているため、0 および 1 は入力として許可されていません。

デコードされた文字列が返されます。 *s* が正しくパディングされていなかったり、規定のアルファベット以外の文字が含まれていた場合には `TypeError` が発生します。

`base64.b16encode(s)`

Base16 をつかって、文字列をエンコード (符号化) します。

s にはエンコードする文字列を渡し、エンコードされた文字列が返されます。

`base64.b16decode(s[, casefold])`

Base16 をつかって、文字列をデコード (復元) します。

s にはエンコードする文字列を渡します。オプション引数 *casefold* は小文字のアルファベットを受けつけるかどうかを指定します。セキュリティ上の理由により、デフォルトではこれは `False` になっています。

デコードされた文字列が返されます。 *s* が正しくパディングされていなかったり、規定のアルファベット以外の文字が含まれていた場合には `TypeError` が発生します。

レガシーなインターフェイスは以下のものを提供します:

`base64.decode(input, output)`

input の中身をデコードした結果を *output* に出力します。 *input*、*output* とともにファイルオブジェクトか、ファイルオブジェクトと同じインターフェースを持ったオブジェクトである必要があります。 *input* は `input.read()` が空文字列を返すまで読まれます。

`base64.decodestring(s)`

文字列 *s* をデコードして結果のバイナリデータを返します。 *s* には一行以上の base64 形式でエンコードされたデータが含まれている必要があります。

`base64.encode(input, output)`

input の中身を base64 形式でエンコードした結果を *output* に出力します。 *input*、*output* とともにファイルオブジェクトか、ファイルオブジェクトと同じインターフェースを持ったオブジェクトである必要があります。 *input* は `input.read()` が空文字列を返すまで読まれます。 `encode()` はエンコードされたデータと改行文字 (`'\n'`) を出力します。

`base64.encodestring(s)`

文字列 *s* (任意のバイナリデータを含むことができます) を r base64 形式でエンコードした結果の (1 行以上の文字列) データを返します。 `encodestring()` はエンコードされた一行以上のデータと改行文字 ('\\n') を出力します。

モジュールの使用例:

```
>>> import base64
>>> encoded = base64.b64encode('data to be encoded')
>>> encoded
'ZGF0YSB0byBiZSB1bmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
'data to be encoded'
```

参考:

モジュール `binascii` ASCII からバイナリへ、バイナリから ASCII への変換をサポートするモジュール。

RFC 1521 - MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies
Section 5.2, "Base64 Content-Transfer-Encoding," provides the definition of the base64 encoding.

18.13 binhex — binhex4 形式ファイルのエンコードおよびデコード

このモジュールは binhex4 形式のファイルに対するエンコードやデコードを行います。 binhex4 は Macintosh のファイルを ASCII で表現できるようにしたものです。 Macintosh 上では、ファイルと finder 情報の両方のフォークがエンコード (またはデコード) されます。他のプラットフォームではデータフォークだけが処理されます。

注釈: Python 3.x で特別な Macintosh サポートは削除されました。

`binhex` モジュールでは以下の関数を定義しています:

`binhex.binhex(input, output)`

ファイル名 *input* のバイナリファイルをファイル名 *output* の binhex 形式ファイルに変換します。 *output* パラメタはファイル名でも (`write()` および `close()` メソッドをサポートするような) ファイル様オブジェクトでもかまいません。

`binhex.hexbin(input[, output])`

binhex 形式のファイル *input* をデコードします。 *input* はファイル名でも、 `write()` および `close()` メソッドをサポートするようなファイル様オブジェクトでもかまいません。変換結果のファイルはファイル名 *output* になります。この引数が省略された場合、出力ファイルは binhex ファイルの中から復元されます。

以下の例外も定義されています:

exception binhex.Error

binhex 形式を使ってエンコードできなかった場合 (例えば、ファイル名が filename フィールドに収まらないくらい長かった場合など) や、入力が正しくエンコードされた binhex 形式のデータでなかった場合に送出される例外です。

参考:

モジュール *binascii* ASCII からバイナリへ、バイナリから ASCII への変換をサポートするモジュール。

18.13.1 注釈

別のより強力なエンコーダおよびデコーダへのインタフェースが存在します。詳しくはソースを参照してください。

非 Macintosh プラットフォームでテキストファイルをエンコードしたりデコードしたりする場合でも、古い Macintosh の改行文字変換 (行末をキャリッジリターンとする) が行われます。

このドキュメントを書いている時点では、*hexbin()* はいつも正しく動作するわけではないようです。

18.14 binascii — バイナリデータと ASCII データとの間での変換

binascii モジュールにはバイナリと ASCII コード化されたバイナリ表現との間の変換を行うための多数のメソッドが含まれています。通常、これらの関数を直接使う必要はなく、*uu*、*base64* や *binhex* といった、ラッパ (wrapper) モジュールを使うことになるでしょう。*binascii* モジュールは、高レベルなモジュールで利用される、高速な C で書かれた低レベル関数を提供しています。

binascii モジュールでは以下の関数を定義します:

binascii.a2b_uu (*string*)

uuencode された 1 行のデータをバイナリに変換し、変換後のバイナリデータを返します。最後の行を除いて、通常 1 行には (バイナリデータで) 45 バイトが含まれます。入力データの先頭には空白文字が連続していてもかまいません。

binascii.b2a_uu (*data*)

バイナリデータを uuencode して 1 行の ASCII 文字列に変換します。戻り値は変換後の 1 行の文字列で、改行を含みます。*data* の長さは 45 バイト以下でなければなりません。

binascii.a2b_base64 (*string*)

base64 でエンコードされたデータのブロックをバイナリに変換し、変換後のバイナリデータを返します。一度に 1 行以上のデータを与えてもかまいません。

binascii.b2a_base64 (*data*)

バイナリデータを base64 でエンコードして 1 行の ASCII 文字列に変換します。戻り値は変換後の 1 行の文字列で、改行文字を含みます。出力には改行コードが追加されます。これはこの関数の元々のユースケースが、MIME-base64 標準に準拠するための出力行を得るために 57 バイトずつ *data* を読んで都度出力を供給する、というものであったためです。このユースケースでの利用でないならば、出力は [RFC 3548](#) に準拠します (—訳注: MIME-base64: [RFC 1521](#)、base64: [RFC 3548](#) —)。

`binascii.a2b_qp(string[, header])`

quoted-printable 形式のデータをバイナリに変換し、バイナリデータを返します。一度に 1 行以上のデータを渡すことができます。オプション引数 *header* が与えられており、かつその値が真であれば、アンダースコアは空白文字にデコードされます。

`binascii.b2a_qp(data[, quotetabs, istext, header])`

バイナリデータを quoted-printable 形式でエンコードして 1 行から複数行の ASCII 文字列に変換します。変換後の文字列を返します。オプション引数 *quotetabs* が存在し、かつその値が真であれば、全てのタブおよび空白文字もエンコードされます。オプション引数 *istext* が存在し、かつその値が真であれば、改行はエンコードされませんが、行末の空白文字はエンコードされます。オプション引数 *header* が存在し、かつその値が真である場合、空白文字は RFC1522 にしたがってアンダースコアにエンコードされます。オプション引数 *header* が存在し、かつその値が偽である場合、改行文字も同様にエンコードされます。そうでない場合、復帰 (linefeed) 文字の変換によってバイナリデータストリームが破損してしまうかもしれません。

`binascii.a2b_hqx(string)`

binhex4 形式の ASCII 文字列データを RLE 展開を行わないでバイナリに変換します。文字列はバイナリのバイトデータを完全に含むような長さか、または (binhex4 データの最後の部分の場合) 余白のビットがゼロになっていなければなりません。

`binascii.rledecode_hqx(data)`

data に対し、binhex4 標準に従って RLE 展開を行います。このアルゴリズムでは、あるバイトの後ろに 0x90 がきた場合、そのバイトの反復を指示しており、さらにその後ろに反復カウントが続きます。カウントが 0 の場合 0x90 自体を示します。このルーチンは入力データの末端における反復指定が不完全でないかぎり解凍されたデータを返しますが、不完全な場合、例外 *Incomplete* が送出されます。

`binascii.rlecode_hqx(data)`

binhex4 方式の RLE 圧縮を *data* に対して行い、その結果を返します。

`binascii.b2a_hqx(data)`

バイナリを hexbin4 エンコードして ASCII 文字列に変換し、変換後の文字列を返します。引数の *data* はすでに RLE エンコードされていなければならない、その長さは (最後のフラグメントを除いて) 3 で割り切れなければなりません。

`binascii.crc_hqx(data, crc)`

crc を初期値として *data* の 16 ビット CRC 値を計算し、その結果を返します。この関数は、よく 0x1021 と表現される CRC-CCITT 多項式 $x^{16} + x^{12} + x^5 + 1$ を使います。この CRC は binhex4 形式で使われています。

`binascii.crc32(data[, crc])`

32 ビットチェックサムである CRC-32 を *data* に対して計算します。初期値は *crc* です。これは ZIP ファイルのチェックサムと同じです。このアルゴリズムはチェックサムアルゴリズムとして設計されたもので、一般的なハッシュアルゴリズムには向きません。以下のように使います:

```
print binascii.crc32("hello world")
# Or, in two pieces:
crc = binascii.crc32("hello")
```

(次のページに続く)

(前のページからの続き)

```
crc = binascii.crc32(" world", crc) & 0xffffffff
print 'crc32 = 0x%08x' % crc
```

注釈: 全ての Python のバージョン、全てのプラットフォームに渡って同じ数値を生成しようとするならば、`crc32(data) & 0xffffffff` を使って下さい。チェックサムをバイナリ形式そのままだけで扱うならばこのような細工は必要ありません。返値は符号に関係なく正しい 32 ビットのバイナリ表現だからです。

バージョン 2.6 で変更: 返値はどのプラットフォームでも `[-2**31, 2**31-1]` の範囲の値です。過去においては返値はあるプラットフォームでは符号付きでまた別のところでは符号無しでした。3.0 における振る舞いに合わせるためには `& 0xffffffff` を施して下さい。

バージョン 3.0 で変更: 戻り値の範囲は、プラットフォームに関係なく `[0, 2**32-1]` の範囲の符号無しです。

`binascii.b2a_hex(data)`

`binascii.hexlify(data)`

バイナリデータ *data* の 16 進数表現を返します。*data* の各バイトは対応する 2 桁の 16 進数表現に変換されます。従って、変換結果の文字列は *data* の 2 倍の長さになります。

`binascii.a2b_hex(hexstr)`

`binascii.unhexlify(hexstr)`

16 進数表記の文字列 *hexstr* の表すバイナリデータを返します。この関数は `b2a_hex()` の逆です。*hexstr* は 16 進数数字 (大文字でも小文字でもかまいません) を偶数個含んでいなければなりません。そうでないばあい、例外 `TypeError` が送出されます。

exception `binascii.Error`

エラーが発生した際に送出される例外です。通常はプログラムのエラーです。

exception `binascii.Incomplete`

変換するデータが不完全な場合に送出される例外です。通常はプログラムのエラーではなく、多少追加読み込みを行って再度変換を試みることで対処できます。

参考:

`base64` モジュール RFC 準拠の base64 形式の、底が 16、32、64 のエンコーディング。

`binhex` モジュール Macintosh で使われる binhex フォーマットのサポート。

`uu` モジュール Unix で使われる UU エンコードのサポート。

`quopri` モジュール MIME 電子メールメッセージで使われる quoted-printable エンコードのサポート。

18.15 quopri — MIME quoted-printable 形式データのエンコードおよびデコード

ソースコード: `Lib/quopri.py`

このモジュールは [RFC 1521](#): "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies" で定義されている quoted-printable による伝送のエンコードおよびデコードを行います。quoted-printable エンコーディングは比較的印字不可能な文字の少ないデータのために設計されています; 画像ファイルを送るときのように印字不可能な文字がたくさんある場合には、[base64](#) モジュールで利用できる base64 エンコーディングのほうがよりコンパクトになります。

`quopri.decode(input, output[, header])`

ファイル *input* の内容をデコードして、デコードされたバイナリデータをファイル *output* に書き出します。*input* および *output* はファイルか、ファイルオブジェクトのインタフェースを真似たオブジェクトでなければなりません。*input* は `input.readline()` が空文字列を返すまで読みつづけられます。オプション引数 *header* が存在し、かつその値が真である場合、アンダースコアは空白文字にデコードされます。これは [RFC 1522](#): "MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text" で記述されているところの "Q"-エンコードされたヘッダをデコードするのに使われます。

`quopri.encode(input, output, quotetabs)`

ファイル *input* の内容をエンコードして、quoted-printable 形式にエンコードされたデータをファイル *output* に書き出します。*input* および *output* はファイルか、ファイルオブジェクトのインタフェースを真似たオブジェクトでなければなりません。*input* は `input.readline()` が空文字列を返すまで読みつづけられます。*quotetabs* はデータ中に埋め込まれた空白文字やタブを変換するかどうか制御するフラグです; この値が真なら、それらの空白をエンコードします。偽ならエンコードせずそのままにしておきます。行末のスペースやタブは [RFC 1521](#) に従って常にエンコードされるので注意してください。

`quopri.decodestring(s[, header])`

`decode()` に似ていますが、文字列を入力として受け取り、デコードされた文字列を返します。

`quopri.encodestring(s[, quotetabs])`

`encode()` に似ていますが、文字列を入力として受け取り、エンコードされた文字列を返します。*quotetabs* はオプション (デフォルトは 0 です) で、この値はそのまま `encode()` に渡されます。

参考:

[mimify](#) モジュール MIME メッセージを処理するための汎用ユーティリティ。

[base64](#) モジュール MIME base64 形式データのエンコードおよびデコード

18.16 uu — uuencode 形式のエンコードとデコード

ソースコード: [Lib/uu.py](#)

このモジュールではファイルを uuencode 形式 (任意のバイナリデータを ASCII 文字列に変換したもの) にエンコード、デコードする機能を提供します。引数としてファイルが仮定されている所では、ファイルのようなオブジェクトが利用できます。後方互換性のために、パス名を含む文字列も利用できるようにして、対応

するファイルを開いて読み書きします。しかし、このインタフェースは利用しないでください。呼び出し側でファイルを開いて (Windows では 'rb' か 'wb' のモードで) 利用する方法が推奨されます。

このコードは Lance Ellinghouse によって提供され、Jack Jansen によって更新されました。

`uu` モジュールでは以下の関数を定義しています:

`uu.encode(in_file, out_file[, name[, mode]])`

`in_file` を `out_file` にエンコードします。エンコードされたファイルには、デフォルトでデコード時に利用される `name` と `mode` を含んだヘッダがつきます。省略された場合には、`in_file` から取得された名前が '-' という文字と、0666 がそれぞれデフォルト値として与えられます。

`uu.decode(in_file[, out_file[, mode[, quiet]]])`

uuencode 形式でエンコードされた `in_file` をデコードして `varout_file` に書き出します。もし `out_file` がパス名でかつファイルを作る必要があるときには、`mode` がパーミッションの設定に使われます。`out_file` と `mode` のデフォルト値は `in_file` のヘッダから取得されます。しかし、ヘッダで指定されたファイルが既に存在していた場合は、`uu.Error` が送出されます。

誤った実装の uuencoder による入力で、エラーから復旧できた場合、`decode()` は標準エラー出力に警告を表示するかもしれません。`quiet` を真にすることでこの警告を抑制することができます。

exception `uu.Error`

`Exception` のサブクラスで、`uu.decode()` によって、さまざまな状況で送出される可能性があります。上で紹介された場合以外にも、ヘッダのフォーマットが間違っている場合や、入力ファイルが途中で区切れた場合にも送出されます。

参考:

モジュール `binascii` ASCII からバイナリへ、バイナリから ASCII への変換をサポートするモジュール。

第 19 章

構造化マークアップツール

Python は様々な構造化データマークアップ形式を扱うための、様々なモジュールをサポートしています。これらは標準化一般マークアップ言語 (SGML) およびハイパーテキストマークアップ言語 (HTML)、そして可拡張性マークアップ言語 (XML) を扱うためのいくつかのインタフェースからなります。

注意すべき重要な点として、`xml` パッケージは少なくとも一つの SAX に対応した XML パーザが利用可能でなければなりません。Python 2.3 からは Expat パーザが Python に取り込まれているので、`xml.parsers.expat` モジュールは常に利用できます。また、`PyXML 追加パッケージ` についても知りたいと思うかもしれませんが、このパッケージは Python 用の拡張された XML ライブラリセットを提供します。

`xml.dom` および `xml.sax` パッケージのドキュメントは Python による DOM および SAX インタフェースへのバインディングに関する定義です。

19.1 HTMLParser — HTML および XHTML のシンプルなパーサー

注釈: `HTMLParser` モジュールは Python 3 で `html.parser` に改名されました。ソースを 3 用に移行する際には `2to3` がインポートを自動的に直してくれます。

バージョン 2.2 で追加.

ソースコード: [Lib/HTMLParser.py](#)

このモジュールでは `HTMLParser` クラスを定義します。このクラスは HTML (ハイパーテキスト記述言語、HyperText Mark-up Language) および XHTML で書式化されているテキストファイルを解釈するための基礎となります。`htmlilib` にあるパーサーと違って、このパーサーは `sgmllib` の SGML パーサーに基づいてはいません。

class `HTMLParser.HTMLParser`

`HTMLParser` インスタンスは、HTML データが入力されると、開始タグ、終了タグ、およびその他の要素が見つかる度にハンドラーメソッドを呼び出します。各メソッドの挙動を実装するには `HTMLParser` サブクラスを使ってそれぞれを上書きして行います。

`HTMLParser` クラスは引数なしでインスタンス化します。

`htmlplib` のパーサーと違い、このパーサーは終了タグが開始タグと一致しているか調べたり、外側のタグ要素が閉じるときに内側で明示的に閉じられていないタグ要素のタグ終了ハンドラを呼び出したりはしません。

例外も定義されています:

exception `HTMLParser.HTMLParseError`

`HTMLParser` は正しくないマークアップを処理することが出来ますが、あるケースにおいてはこの例外が発生します。これは、パース中にエラーに遭遇した場合の例外です。この例外は三つの属性を提供しています: `msg` はエラーの内容を説明する簡単なメッセージ、`lineno` は壊れたマークアップ構造を検出した場所の行番号、`offset` は問題のマークアップ構造の行内での開始位置を示す文字数です。

19.1.1 HTML パーサーアプリケーションの例

基礎的な例として、`HTMLParser` クラスを使い、発見した開始タグ、終了タグ、およびデータを出力する、シンプルな HTML パーサーを以下に示します:

```
from HTMLParser import HTMLParser

# create a subclass and override the handler methods
class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print "Encountered a start tag:", tag

    def handle_endtag(self, tag):
        print "Encountered an end tag :", tag

    def handle_data(self, data):
        print "Encountered some data  :", data

# instantiate the parser and fed it some HTML
parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>'
          '<body><h1>Parse me!</h1></body></html>')
```

出力は以下のようになります:

```
Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data  : Test
Encountered an end tag : title
Encountered an end tag : head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data  : Parse me!
Encountered an end tag : h1
```

(次のページに続く)

(前のページからの続き)

```
Encountered an end tag : body
Encountered an end tag : html
```

19.1.2 HTMLParser メソッド

`HTMLParser` インスタンスは以下のメソッドを提供します:

`HTMLParser.feed(data)`

パーサーにテキストを入力します。入力が完全なタグ要素で構成されている場合に限り処理が行われます; 不完全なデータであった場合、新たにデータが入力されるか、`close()` が呼び出されるまでバッファされます。`data` は `unicode` でも `str` でも構いませんが、`unicode` の方が良いです。

`HTMLParser.close()`

全てのバッファされているデータについて、その後にファイル終了マークが続いているとみなして強制的に処理を行います。このメソッドは入力データの終端で行うべき追加処理を定義するために派生クラスで上書きすることができますが、再定義を行ったクラスでは常に、`HTMLParser` 基底クラスのメソッド `close()` を呼び出さなくてはなりません。

`HTMLParser.reset()`

インスタンスをリセットします。未処理のデータはすべて失われます。インスタンス化の際に暗黙的に呼び出されます。

`HTMLParser.getpos()`

現在の行番号およびオフセット値を返します。

`HTMLParser.get_starttag_text()`

最も最近開かれた開始タグのテキスト部分を返します。このテキストは必ずしも元データを構造化する上で必須ではありませんが、"広く知られている (as deployed)" HTML を扱ったり、入力を最小限の変更で再生成 (属性間の空白をそのままにする、など) したりする場合に便利ことがあります。

以下のメソッドはデータまたはマークアップ要素が見つかる度に呼び出されます。これらはサブクラスで上書きされることを想定されています。基底クラスの実装は (`handle_startendtag()` を除き) 何もしません:

`HTMLParser.handle_starttag(tag, attrs)`

このメソッドは開始タグを扱うために呼び出されます (例: `<div id="main">`)。

引数 `tag` はタグの名前で、小文字に変換されます。引数 `attrs` は (`name`, `value`) のペアからなるリストで、タグの `<>` 括弧内にある属性が収められています。`name` は小文字に変換され、`value` 内の引用符は取り除かれ、文字参照と実態参照は置き換えられます。

例えば、タグ `` を処理する場合、このメソッドは `handle_starttag('a', [('href', 'https://www.cwi.nl/')])` として呼び出されます。

バージョン 2.6 で変更: 属性中の `htmlentitydefs` の全てのエンティティ参照が置換されるようになりました。

`HTMLParser.handle_endtag(tag)`

このメソッドは要素の終了タグを扱うために呼び出されます (例: `</div>`)。

引数 *tag* はタグの名前で、小文字に変換されます。

`HTMLParser.handle_startendtag(tag, attrs)`

`handle_starttag()` と似ていますが、パーサーが XHTML 形式の空タグ (``) を見つける度に呼び出されます。この特定の字句情報が必要な場合にこのメソッドをサブクラスで上書きすることができます; 既定の実装では、単に `handle_starttag()` および `handle_endtag()` を呼び出します。

`HTMLParser.handle_data(data)`

このメソッドは任意のデータを処理するために呼び出されます (例: テキストノードおよび `<script>...</script> a` や `<style>...</style>` の内容)。

`HTMLParser.handle_entityref(name)`

このメソッドは `&name;` 形式の名前指定文字参照 (例: `>`) を処理するために呼び出されます。 *name* は一般実体参照になります (例: `'gt'`)。

`HTMLParser.handle_charref(name)`

このメソッドは `&#NNN;` あるいは `&#xNNN;` 形式の 10 進および 16 進数値文字参照を処理するために呼び出されます。例えば、`>` と等価な 10 進数は `>` で、16 進数は `>` になります。この場合、メソッドは `'62'` あるいは `'x3E'` を受け取ります。

`HTMLParser.handle_comment(data)`

このメソッドはコメントが見つかった場合に呼び出されます (例: `<!--comment-->`)。

例えば、コメント `<!-- comment -->` があると。このメソッドを引数 `'comment'` で呼び出されます。

Internet Explorer 独自拡張の条件付きコメント (condcoms) はこのメソッドに送ることができます。 `<!--[if IE 9]>IE9-specific content<![endif]-->` の場合、このメソッドは `'[if IE 9]>IE9-specific content<![endif]'` を受け取ります。

`HTMLParser.handle_decl(decl)`

このメソッドは HTML doctype 宣言を扱うために呼び出されます (例: `<!DOCTYPE html>`)。

引数 *decl* は `<!...>` マークアップ内の宣言の内容全体になります (例: `'DOCTYPE html'`)。

`HTMLParser.handle_pi(data)`

処理指令 (processing instruction) が見つかった場合に呼び出されます。 *data* には、処理指令全体が含まれ、例えば `<?proc color='red'>` という処理指令の場合、`handle_pi("proc color='red'")` のように呼び出されます。

注釈: The `HTMLParser` クラスでは、処理指令に SGML の構文を使用します。末尾に `'?'` がある XHTML の処理指令では、`'?'` が *data* に含まれることになります。

`HTMLParser.unknown_decl(data)`

このメソッドはパーサーが未知の宣言を読み込んだ時に呼び出されます。

パラメーター `data` は `<![...]>` マークアップ内の宣言の内容全体になります。これは派生クラスで上書きする時に役立つ場合があります。

19.1.3 例

以下のクラスは、より多くの例を示すのに用いられるパーサーの実装です:

```
from HTMLParser import HTMLParser
from htmlentitydefs import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print "Start tag:", tag
        for attr in attrs:
            print "    attr:", attr

    def handle_endtag(self, tag):
        print "End tag  :", tag

    def handle_data(self, data):
        print "Data      :", data

    def handle_comment(self, data):
        print "Comment  :", data

    def handle_entityref(self, name):
        c = unichr(name2codepoint[name])
        print "Named ent:", c

    def handle_charref(self, name):
        if name.startswith('x'):
            c = unichr(int(name[1:], 16))
        else:
            c = unichr(int(name))
        print "Num ent  :", c

    def handle_decl(self, data):
        print "Decl      :", data

parser = MyHTMLParser()
```

doctype をパースします:

```
>>> parser.feed('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" '
...             '"http://www.w3.org/TR/html4/strict.dtd">')
Decl      : DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
↳html4/strict.dtd"
```

要素のタイトルと一部属性をパースします:

```
>>> parser.feed('')
Start tag: img
      attr: ('src', 'python-logo.png')
      attr: ('alt', 'The Python logo')
>>>
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
Data      : Python
End tag   : h1
```

それ以上のパースを行わずに、script と style 要素の内容をそのまま返します:

```
>>> parser.feed('<style type="text/css">#python { color: green }</style>')
Start tag: style
      attr: ('type', 'text/css')
Data      : #python { color: green }
End tag   : style

>>> parser.feed('<script type="text/javascript">'
...             'alert("<strong>hello!</strong>");</script>')
Start tag: script
      attr: ('type', 'text/javascript')
Data      : alert("<strong>hello!</strong>");
End tag   : script
```

コメントをパースします:

```
>>> parser.feed('<!-- a comment -->'
...             '<!--[if IE 9]>IE-specific content<![endif]-->')
Comment   : a comment
Comment   : [if IE 9]>IE-specific content<![endif]
```

名前指定および数値文字参照をパースし、正しい文字に変換します (注: これら 3 個の参照はすべて '>' と等価です):

```
>>> parser.feed('&gt;&#62;&#x3E;')
Named ent: >
Num ent   : >
Num ent   : >
```

不完全なチャンクを *feed()* に処理させても、*handle_data()* は 1 回以上呼び出される場合があります:

```
>>> for chunk in ['<sp', 'an>buff', 'ered ', 'text</s', 'pan>']:
...     parser.feed(chunk)
...
Start tag: span
Data      : buff
Data      : ered
Data      : text
End tag   : span
```

不正な HTML (例えば属性が引用符で括られていない) のパースも動作します:

```
>>> parser.feed('<p><a class=link href=#main>tag soup</p ></a>')
Start tag: p
Start tag: a
  attr: ('class', 'link')
  attr: ('href', '#main')
Data      : tag soup
End tag   : p
End tag   : a
```

19.2 sgmllib — 単純な SGML パーザ

バージョン 2.6 で非推奨: *sgmllib* モジュールは Python 3 で削除されました。

このモジュールでは SGML (Standard Generalized Mark-up Language: 汎用マークアップ言語標準) で書式化されたテキストファイルを解析するための基礎として働く *SGMLParser* クラスを定義しています。実際には、このクラスは完全な SGML パーザを提供しているわけではありません — このクラスは HTML で用いられているような SGML だけを解析し、モジュール自体も *htmlib* モジュールの基礎にするためだけに存在しています。XHTML をサポートし、少し異なったインタフェースを提供しているもう一つの HTML パーザは、*HTMLParser* モジュールで使うことができます。

class *sgmllib.SGMLParser*

SGMLParser クラスは引数無しでインスタンス化されます。このパーザは以下の構成を認識するようにハードコードされています:

- `<tag attr="value" ...>` と `</tag>` で表されるタグの開始部と終了部。
- `&#name;` 形式をとる文字の数値参照。
- `&name;` 形式をとるエンティティ参照。
- `<!--text-->` 形式をとる SGML コメント。末尾の `>` とその直前にある `--` の間にはスペース、タブ、改行を入れることができます。

一つの例外が以下のように定義されます:

exception *sgmllib.SGMLParseError*

SGMLParser クラスで構文解析中にエラーに出逢うとこの例外が発生します。

バージョン 2.1 で追加.

SGMLParser インスタンスは以下のメソッドを持っています:

SGMLParser.**reset**()

インスタンスをリセットします。未処理のデータはすべて失われます。インスタンス化の際に暗黙的に呼び出されます。

SGMLParser.**setnomoretags**()

タグの処理を停止します。以降の入力をリテラル入力 (CDATA) として扱います。(この機能は HTML

タグ <PAINTTEXT> を実装できるようにするためだけに提供されています)

`SGMLParser.setliteral()`

リテラルモード (CDATA モード) に移行します。

`SGMLParser.feed(data)`

テキストをパーザに入力します。入力是完全なエレメントから成り立つ場合に限り処理されます; 不完全なデータは追加のデータが入力されるか、`close()` が呼び出されるまでバッファに蓄積されます。

`SGMLParser.close()`

バッファに蓄積されている全てのデータについて、直後に EOF が来た時のようにして強制的に処理します。このメソッドは派生クラスで再定義して、入力の終了時に追加の処理を行うよう定義することができますが、このメソッドの再定義されたバージョンでは常に `close()` を呼び出さなければなりません。

`SGMLParser.get_starttag_text()`

最も最近開かれた開始タグのテキスト部分を返します。このテキストは必ずしも元データを構造化する上で必須ではありませんが、”広く知られている (as deployed)” HTML を扱ったり、入力を最小限の変更で再生成 (属性間の空白をそのままにするなど) したりする場合に便利ことがあります。

`SGMLParser.handle_starttag(tag, method, attributes)`

このメソッドは `start_tag()` か `do_tag()` のどちらかのメソッドが定義されている開始タグを処理するために呼び出されます。 `tag` 引数はタグの名前で、小文字に変換されています。 `method` 引数は開始タグの意味解釈をサポートするために用いられるバインドされたメソッドです。 `attributes` 引数は `(name, value)` のペアからなるリストで、タグの <> 括弧内にある属性が収められています。

`name` は小文字に変換されます。 `value` 内の二重引用符とバックスラッシュも変換され、と同時に知られている文字参照および知られているエンティティ参照でセミコロンで終端されているものも変換されます (通常、エンティティ参照は任意の非英数文字で終端されてよいのですが、これを許すと非常に一般的な `` において `eggs` が正当なエンティティ参照であるようなケースを破綻させます)。

例えば、タグ `` を処理する場合、このメソッドは `unknown_starttag('a', [('href', 'http://www.cwi.nl/')])` として呼び出されます。基底クラスの実装では、単に `method` を単一の引数 `attributes` と共に呼び出します。

バージョン 2.5 で追加: 属性値中のエンティティおよび文字参照の扱い。

`SGMLParser.handle_endtag(tag, method)`

このメソッドは `end_tag()` メソッドの定義されている終了タグを処理するために呼び出されます。 `tag` 引数はタグの名前で、小文字に変換されており、 `method` 引数は終了タグの意味解釈をサポートするために使われるバインドされたメソッドです。 `end_tag()` メソッドが終了エレメントとして定義されていない場合、このハンドラは呼び出されません。基底クラスの実装では単に `method` を呼び出します。

`SGMLParser.handle_data(data)`

このメソッドは何らかのデータを処理するために呼び出されます。派生クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。

`SGMLParser.handle_charref(ref)`

このメソッドは `&#ref;` 形式の文字参照 (character reference) を処理するために呼び出されます。基底クラスの実装は、`convert_charref()` を使って参照を文字列に変換します。もしそのメソッドが文字列を返せば `handle_data()` を呼び出します。そうでなければ、エラーを処理するために `unknown_charref(ref)` が呼び出されます。

バージョン 2.5 で変更: ハードコードされた変換ではなく `convert_charref()` を使うようになりました。

`SGMLParser.convert_charref(ref)`

文字参照を文字列に変換するか、`None` を返します。`ref` は文字列として渡される参照です。基底クラスでは `ref` は 0-255 の範囲の十進数でなければなりません。そしてコードポイントをメソッド `convert_codepoint()` を使って変換します。もし `ref` が不正もしくは範囲外ならば、`None` を返します。このメソッドはデフォルト実装の `handle_charref()` と属性値パーザから呼び出されます。

バージョン 2.5 で追加。

`SGMLParser.convert_codepoint(codepoint)`

コードポイントを `str` の値に変換します。もしそれが適切ならばエンコーディングをここで扱うこともできますが、`sgml-lib` の残りの部分はこの問題に関知しません。

バージョン 2.5 で追加。

`SGMLParser.handle_entityref(ref)`

このメソッドは `ref` を一般エンティティ参照として、`&ref;` 形式のエンティティ参照を処理するために呼び出されます。このメソッドは、`ref` を `convert_entityref()` に渡して変換します。変換結果が返ってきた場合、変換された文字を引数にして `handle_data()` を呼び出します; そうでない場合、`unknown_entityref(ref)` を呼び出します。標準では `entitydefs` は `&`、`'`、`>`、`<`、および `"` の変換を定義しています。

バージョン 2.5 で変更: ハードコードされた変換ではなく `convert_entityref()` を使うようになりました。

`SGMLParser.convert_entityref(ref)`

名前付きエンティティ参照を `str` の値に変換するか、または `None` を返します。変換結果は再パースしません。`ref` はエンティティの名前部分だけです。デフォルトの実装ではインスタンス (またはクラス) 変数の `entitydefs` というエンティティ名から対応する文字列へのマッピングから `ref` を探します。もし `ref` に対応する文字列が見つからなければメソッドは `None` を返します。このメソッドは `handle_entityref()` のデフォルト実装からおよび属性値パーザから呼び出されます。

バージョン 2.5 で追加。

`SGMLParser.handle_comment(comment)`

このメソッドはコメントに遭遇した場合に呼び出されます。`comment` 引数は文字列で、`<!--` と `-->` デリミタ間の、デリミタ自体を除いたテキストが収められています。例えば、コメント `<!--text-->` があると、このメソッドは引数 `'text'` で呼び出されます。基底クラスの実装では何も行いません。

`SGMLParser.handle_decl(data)`

パーザが SGML 宣言を読み出した際に呼び出されるメソッドです。実際には、DOCTYPE は HTML だけに見られる宣言ですが、パーザは宣言間の相違 (や誤った宣言) を判別しません。DOCTYPE の内部

サブセット宣言はサポートされていません。 *data* パラメタは `<!...>` 記述内の宣言内容全体になります。基底クラスの実装では何も行いません。

`SGMLParser.report_unbalanced(tag)`

このメソッドは対応する開始エレメントのない終了タグが発見された時に呼び出されます。

`SGMLParser.unknown_starttag(tag, attributes)`

未知の開始タグを処理するために呼び出されるメソッドです。派生クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。

`SGMLParser.unknown_endtag(tag)`

未知の終了タグを処理するために呼び出されるメソッドです。派生クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。

`SGMLParser.unknown_charref(ref)`

このメソッドは解決不能な文字参照数値を処理するために呼び出されます。標準で何が処理可能かは `handle_charref()` を参照してください。派生クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。

`SGMLParser.unknown_entityref(ref)`

未知のエンティティ参照を処理するために呼び出されるメソッドです。派生クラスで上書きするためのメソッドです; 基底クラスの実装では何も行いません。

上に挙げたメソッドを上書きしたり拡張したりするのは別に、派生クラスでは以下の形式のメソッドを定義して、特定のタグを処理することもできます。入力ストリーム中のタグ名は大小文字の区別に依存しません; メソッド名中の *tag* は小文字でなければなりません:

`SGMLParser.start_tag(attributes)`

このメソッドは開始タグ *tag* を処理するために呼び出されます。 `do_tag()` よりも高い優先順位があります。 *attributes* 引数は上の `handle_starttag()` で記述されているのと同じ意味です。

`SGMLParser.do_tag(attributes)`

このメソッドは `start_tag()` メソッドが定義されていない開始タグ *tag* を処理するために呼び出されます。 *attributes* 引数は上の `handle_starttag()` で記述されているのと同じ意味です。

`SGMLParser.end_tag()`

このメソッドは終了タグ *tag* を処理するために呼び出されます。

パーザは開始されたエレメントのうち、終了タグがまだ見つからないもののスタックを維持しているので注意してください。 `start_tag()` で処理されたタグだけがスタックにプッシュされます。それらのタグに対する `end_tag()` メソッドの定義はオプションです。 `do_tag()` や `unknown_tag()` で処理されるタグについては、 `end_tag()` を定義してはいけません; 定義されていても使われることはありません。あるタグに対して `start_tag()` および `do_tag()` メソッドの両方が存在する場合、 `start_tag()` が優先されます。

19.3 `htmllib` — HTML 文書の解析器

バージョン 2.6 で非推奨: `htmllib` モジュールは Python 3 で削除されました。Python 2 では代わりに `HTMLParser` を使ってください。なお、`HTMLParser` は Python 3 では `html.parser` に移動しています。

このモジュールでは、ハイパーテキストマークアップ言語 (HTML, HyperText Mark-up Language) 形式でフォーマットされたテキストファイルを解析するための基盤として役立つクラスを定義しています。このクラスは I/O と直接的には接続されません — このクラスにはメソッドを介して文字列形式の入力を提供する必要があり、出力を生成するには "フォーマッタ (formatter)" オブジェクトのメソッドを何度か呼び出さなくてはなりません。`HTMLParser` クラスは、機能を追加するために他のクラスの基底クラスとして利用するように設計されており、ほとんどのメソッドが拡張したり上書きしたりできるようになっています。さらにこのクラスは `sgmlib` モジュールで定義されている `SGMLParser` クラスから派生しており、その機能を拡張しています。`HTMLParser` の実装は、[RFC 1866](#) で解説されている HTML 2.0 記述言語をサポートします。`formatter` では 2 つのフォーマッタオブジェクト実装が提供されています; フォーマッタのインタフェースについての情報は `formatter` モジュールのドキュメントを参照してください。

以下は `sgmlib.SGMLParser` で定義されているインタフェースの概要です:

- インスタンスにデータを与えるためのインタフェースは `feed()` メソッドで、このメソッドは文字列を引数に取ります。このメソッドに一度に与えるテキストは必要に応じて多くも少なくもできます; というのは `p.feed(a)`; `p.feed(b)` は `p.feed(a+b)` と同じ効果を持つからです。与えられたデータが完全な HTML マークアップ文を含む場合、それらの文は即座に処理されます; 不完全なマークアップ構造はバッファに保存されます。全ての未処理データを強制的に処理させるには、`close()` メソッドを呼び出します。

例えば、ファイルの全内容を解析するには:

```
parser.feed(open('myfile.html').read())
parser.close()
```

- HTML タグに対して意味付けを定義するためのインタフェースはとても単純です: サブクラスを派生して、`start_tag()`、`end_tag()`、あるいは `do_tag()` といったメソッドを定義するだけです。パーザはこれらのメソッドを適切なタイミングで呼び出します: `start_tag()` や `do_tag()` は `<tag ...>` の形式の開始タグに遭遇した時に呼び出されます; `end_tag()` は `</tag>` の形式の終了タグに遭遇した時に呼び出されます。 `<H1> ... </H1>` のように開始タグが終了タグと対応している必要がある場合、クラス中で `start_tag()` が定義されていなければなりません; `<P>` のように終了タグが必要ない場合、クラス中では `do_tag()` を定義しなければなりません。

このモジュールではパーザクラスと例外を一つずつ定義しています:

class `htmllib.HTMLParser(formatter)`

基底となる HTML パーザクラスです。XHTML 1.0 仕様 (<https://www.w3.org/TR/xhtml1>) 勧告で要求されている全てのエンティティ名をサポートしています。また、全ての HTML 2.0 の要素および HTML 3.0、3.2 の多くの要素のハンドラを定義しています。

exception `htmllib.HTMLParseError`

`HTMLParser` クラスがパーズ処理中にエラーに遭遇した場合に送出する例外です。

バージョン 2.4 で追加.

参考:

`formatter` モジュール 抽象化された書式イベントの流れを `writer` オブジェクト上の特定の出力イベントに変換するためのインターフェース。

`HTMLParser` モジュール HTML パーザのひとつです。やや低いレベルでしか入力を取扱えませんが、XHTML を扱うことができるように設計されています。”広く知られている HTML (HTML as deployed)” では使われておらずかつ XHTML では正しくないと言われる SGML 構文のいくつかは実装されていません。

`htmlentitydefs` モジュール XHTML 1.0 エンティティに対する置換テキストの定義。

`sgmllib` モジュール `HTMLParser` の基底クラス。

19.3.1 HTMLParser オブジェクト

タグメソッドに加えて、`HTMLParser` クラスではタグメソッドで利用するためのいくつかのメソッドとインスタンス変数を提供しています。

`HTMLParser.formatter`

パーザに関連付けられているフォーマッタインスタンスです。

`HTMLParser.nofill`

ブール値のフラグで、空白文字を縮約したくないときには真、縮約するときには偽にします。一般的には、この値を真にするのは、`<PRE>` 要素の中のテキストのように、文字列データが”書式化済みの (preformatted)” 場合だけです。標準の値は偽です。この値は `handle_data()` および `save_end()` の操作に影響します。

`HTMLParser.anchor_bgn(href, name, type)`

このメソッドはアンカー領域の先頭で呼び出されます。引数は `<A>` タグの属性で同じ名前を持つものに対応します。標準の実装では、ドキュメント内のハイパーリンク (`<A>` タグの `HREF` 属性) を列挙したリストを維持しています。ハイパーリンクのリストはデータ属性 `anchorlist` で手に入れることができます。

`HTMLParser.anchor_end()`

このメソッドはアンカー領域の末尾で呼び出されます。標準の実装では、テキストの脚注マーカを追加します。マーカは `anchor_bgn()` で作られたハイパーリンクリストのインデクス値です。

`HTMLParser.handle_image(source, alt[, ismap[, align[, width[, height]]]])`

このメソッドは画像を扱うために呼び出されます。標準の実装では、単に `handle_data()` に `alt` の値を渡すだけです。

`HTMLParser.save_bgn()`

文字列データをフォーマッタオブジェクトに送らずにバッファに保存する操作を開始します。保存されたデータは `save_end()` で取得してください。 `save_bgn()` / `save_end()` のペアを入れ子構造にすることはできません。

`HTMLParser.save_end()`

文字列データのバッファリングを終了し、以前 `save_bgn()` を呼び出した時点から保存されている全てのデータを返します。 `nofill` フラグが偽の場合、空白文字は全てスペース文字一文字に置き換えられます。予め `save_bgn()` を呼ばないでこのメソッドを呼び出すと `TypeError` 例外が送出されます。

19.4 `htmlentitydefs` — HTML 一般エンティティの定義

注釈: Python 3 で `htmlentitydefs` モジュールは `html.entities` と改名されました。ソースを 3 用に変換する際には `2to3` ツールが自動的に `import` を直してくれます。

ソースコード: [Lib/htmlentitydefs.py](#)

このモジュールでは `name2codepoint`、`codepoint2name`、`entitydefs` の三つの辞書を定義しています。 `entitydefs` は `htmllib` モジュールで `HTMLParser` クラスの `entitydefs` メンバを定義するために使われます。このモジュールでは XHTML 1.0 で定義された全てのエンティティを提供しており、Latin-1 文字集合 (ISO-8859-1) の簡単なテキスト置換を行う事ができます。

`htmlentitydefs.entitydefs`

各 XHTML 1.0 実体定義と ISO Latin-1 における置換テキストとを対応付ける辞書です。

`htmlentitydefs.name2codepoint`

HTML 実体名と Unicode コードポイントとを対応付ける辞書です。

バージョン 2.3 で追加.

`htmlentitydefs.codepoint2name`

Unicode コードポイントと HTML 実体名とを対応付ける辞書です。

バージョン 2.3 で追加.

19.5 XML を扱うモジュール群

Python の XML を扱うインタフェースは `xml` パッケージにまとめられています。

警告: XML モジュール群は不正なデータや悪意を持って生成されたデータに対して安全ではありません。信頼できないデータをパースする必要がある場合は、[XML の脆弱性](#) を参照してください。

注意すべき重要な点として、`xml` パッケージは少なくとも一つの SAX に対応した XML パーザが利用可能でなければなりません。Expat パーザが Python に取り込まれているので、`xml.parsers.expat` モジュールは常に利用できます。

`xml.dom` および `xml.sax` パッケージのドキュメントは Python による DOM および SAX インタフェースへのバインディングに関する定義です。

XML に関連するサブモジュール:

- `xml.etree.ElementTree`: ElementTree API、シンプルで軽量な XML プロセッサ
- `xml.dom`: DOM API の定義
- `xml.dom.minidom`: 最小限の DOM の実装
- `xml.dom.pulldom`: 部分的な DOM ツリー構築のサポート
- `xml.sax`: SAX2 基底クラスと便利関数群
- `xml.parsers.expat`: Expat parser バインディング

19.6 XML の脆弱性

XML 処理モジュールは悪意を持って生成されたデータに対して安全ではありません。攻撃者は脆弱性を使って、DoS 攻撃、ローカルファイルへのアクセス、他マシンへのネットワーク接続、ファイアーウォールの迂回などが可能になります。XML に対する攻撃は、エンティティに対するインライン DTD (Document Type Definition) などのマイナーな機能を悪用します。

以下のテーブルは知られている攻撃と、各モジュールがそれに対して脆弱であるかどうかを示しています。

種類	sax	etree	minidom	pulldom	xmlrpc
billion laughs	脆弱	脆弱	脆弱	脆弱	脆弱
quadratic blowup	脆弱	脆弱	脆弱	脆弱	脆弱
external entity expansion	脆弱	安全 (1)	安全 (2)	脆弱	安全 (3)
DTD retrieval	脆弱	安全	安全	脆弱	安全
decompression bomb	安全	安全	安全	安全	脆弱

1. `xml.etree.ElementTree` は外部のエンティティを展開せず、`ParserError` 例外を発生させます。
2. `xml.dom.minidom` は外部エンティティを展開せず、展開前のエンティティをそのまま返します。
3. `xmlrpclib` は外部エンティティを展開せず、除外します。

billion laughs / exponential entity expansion **Billion Laughs** 攻撃 – exponential entity expansion と呼ばれる – は、複数レベルにネストしたエンティティを用います。各エンティティは他のエンティティを複数回参照し、最後のエンティティ定義は小さい文字列を含みます。最終的に、その小さい文字列は数ギガバイトにまで展開されます。exponential expansion は大量の CPU 時間も消費します。

quadratic blowup entity expansion quadratic blowup 攻撃は **Billion Laughs** 攻撃と似ています。こちらもエンティティの展開を利用します。ネストしたエンティティを用いる代わりに、数千文字のエンティティ 1 つを繰り返し利用します。この攻撃は指数関数的な効率はありませんが、パーサーのエンティティのネスト対策を回避します。

external entity expansion エンティティの定義はただのテキスト置換以上のことができます。public identifier や system identifier を使って外部のリソースを参照することもできます。system identifier は標準的な URI やローカルファイルを参照できます。XML パーサーはリソースを HTTP や FTP リクエストを用いて取得し、その内容を XML ドキュメントに埋め込みます。

DTD retrieval Python の `xml.dom.pulldom` のような XML ライブラリは DTD をリモートやローカルの場所から読み込みます。この機能には外部エンティティ展開の問題と同じことが予想されます。

decompression bomb decompression bomb (ZIP bomb と呼ばれる) 問題は、gzipped HTTP ストリームや LZMA ファイルなどの圧縮された XML ストリームをパースできる全ての XML ライブラリに適用されます。攻撃者は送信するデータ量を 1/3 かそれ以下に減らすことができます。

PyPI にある `defusedxml` のドキュメントは、全ての知られている攻撃手法についてのさらなる情報を、例や参照つきで提供しています。

19.6.1 defused packages

これらの拡張パッケージは、信頼出来ない XML データを解析するあらゆるコードでの使用に推奨されます。

`defusedxml` は、潜在的に悪意のある操作を防止するように修正された標準 XML パーザのサブクラスを含む pure Python パッケージです。このパッケージには、さらに exploit の例や xpath インジェクションのようなより多くの XML exploit についての幅広いドキュメントが付属しています。

`defusedexpat` は libexpat を改造したものと、`pyexpat` にパッチを当てた拡張モジュールで、エンティティ展開を使った DoS 攻撃の対策をしています。`defusedexpat` は、安全で設定可能な量のエンティティ展開を許可します。この改造は将来のバージョンの Python にマージされる予定です。

これらのワークアラウンドや改造は、後方互換性のためにパッチレベルのリリースには含められません。また、インライン DTD とエンティティ展開は、正しく定義された XML の機能です。

19.7 xml.etree.ElementTree — ElementTree XML API

バージョン 2.5 で追加.

Source code: [Lib/xml/etree/ElementTree.py](#)

`Element` 型は柔軟性のあるコンテナオブジェクトで、階層的データ構造をメモリーに格納するようにデザインされています。この型は言わばリストと辞書の間の子のようなものです。

警告: `xml.etree.ElementTree` モジュールは悪意を持って構築されたデータに対して安全ではありません。信頼できないデータや認証されていないデータを解析する必要がある場合は、[XML の脆弱性](#) を参照してください。

各要素は関連する多くのプロパティを持っています:

- この要素がどういう種類のデータを表現しているかを同定する文字列であるタグ (別の言い方をすれば要素の型)
- 幾つもの属性 (Python 辞書に収められます)
- テキスト文字列
- オプションの末尾文字列
- 幾つもの子要素 (Python シーケンスに収められます)

element のインスタンスを作るには、`Element` コンストラクタや `SubElement()` ファクトリー関数を使います。

`ElementTree` クラスは要素の構造を包み込み、それと XML を行き来するのに使えます。

この API の C 実装である `xml.etree.cElementTree` も使用可能です。

チュートリアルその他のドキュメントへのリンクについては <http://effbot.org/zone/element-index.htm> を参照して下さい。Fredrik Lundh のページも `xml.etree.ElementTree` の開発バージョンの置き場所です。

バージョン 2.7 で変更: `ElementTree` API が 1.3 に更新されました。より詳しい情報については、[Introducing ElementTree 1.3](#) を参照してください。

19.7.1 チュートリアル

これは `xml.etree.ElementTree` (略して ET) を使用するための短いチュートリアルで、ブロックの構築およびモジュールの基本コンセプトを紹介することを目的としています。

XML 木構造と要素

XML は本質的に階層データ形式で、木構造で表すのが最も自然な方法です。ET はこの目的のために 2 つのクラス - XML 文書全体を木で表す `ElementTree` および木構造内の単一ノードを表す `Element` - を持っています。文書全体とのやりとり (ファイルの読み書き) は通常 `ElementTree` レベルで行います。単一 XML 要素およびその子要素とのやりとりは `Element` レベルで行います。

XML の解析

このセクションでは例として以下の XML 文書を使います:

```
<?xml version="1.0"?>
<data>
  <country name="Liechtenstein">
    <rank>1</rank>
    <year>2008</year>
    <gdppc>141100</gdppc>
    <neighbor name="Austria" direction="E"/>
    <neighbor name="Switzerland" direction="W"/>
  </country>
```

(次のページに続く)

(前のページからの続き)

```

<country name="Singapore">
  <rank>4</rank>
  <year>2011</year>
  <gdppc>59900</gdppc>
  <neighbor name="Malaysia" direction="N"/>
</country>
<country name="Panama">
  <rank>68</rank>
  <year>2011</year>
  <gdppc>13600</gdppc>
  <neighbor name="Costa Rica" direction="W"/>
  <neighbor name="Colombia" direction="E"/>
</country>
</data>

```

データを取り込むのにはいくつかの方があります。ディスクから読むにはこうします:

```

import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()

```

文字列から取り込むにはこうします:

```
root = ET.fromstring(country_data_as_string)
```

`fromstring()` は XML を文字列から *Element* に直接パースします。*Element* はパースされた木のルート要素です。他のパース関数は *ElementTree* を作成するかもしれません。ドキュメントをきちんと確認してください。

Element として、`root` はタグと属性の辞書を持ちます:

```

>>> root.tag
'data'
>>> root.attrib
{}

```

さらにイテレート可能な子ノードも持ちます:

```

>>> for child in root:
...     print child.tag, child.attrib
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}

```

子ノードは入れ子になっており、インデックスで子ノードを指定してアクセスできます:

```

>>> root[0][1].text
'2008'

```

関心ある要素の検索

Element は、例えば、*Element.iter()* などの、配下 (その子ノードや孫ノードなど) の部分木全体を再帰的にイテレートするいくつかの役立つメソッドを持っています:

```
>>> for neighbor in root.iter('neighbor'):
...     print neighbor.attrib
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

Element.findall() はタグで現在の要素の直接の子要素のみ検索します。 *Element.find()* は特定のタグで 最初の子要素を検索し、 *Element.text* は要素のテキストコンテンツにアクセスします。 *Element.get()* は要素の属性にアクセスします:

```
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print name, rank
...
Liechtenstein 1
Singapore 4
Panama 68
```

XPath の使用は、検索したい要素を指定するより洗練された方法です。

XML ファイルの編集

ElementTree は XML 文書を構築してファイルに出力する簡単な方法を提供しています。 *ElementTree.write()* メソッドはこの目的に適います。

Element オブジェクトを作成すると、そのフィールドの直接変更 (*Element.text* など) や、属性の追加および変更 (*Element.set()* メソッド)、あるいは新しい子ノードの追加 (例えば *Element.append()* など) によってそれを操作できます。

例えば各 country の rank に 1 を足して、rank 要素に updated 属性を追加したい場合:

```
>>> for rank in root.iter('rank'):
...     new_rank = int(rank.text) + 1
...     rank.text = str(new_rank)
...     rank.set('updated', 'yes')
...
>>> tree.write('output.xml')
```

XML はこのようになります:

```
<?xml version="1.0"?>
<data>
  <country name="Liechtenstein">
    <rank updated="yes">2</rank>
    <year>2008</year>
    <gdppc>141100</gdppc>
    <neighbor name="Austria" direction="E"/>
    <neighbor name="Switzerland" direction="W"/>
  </country>
  <country name="Singapore">
    <rank updated="yes">5</rank>
    <year>2011</year>
    <gdppc>59900</gdppc>
    <neighbor name="Malaysia" direction="N"/>
  </country>
  <country name="Panama">
    <rank updated="yes">69</rank>
    <year>2011</year>
    <gdppc>13600</gdppc>
    <neighbor name="Costa Rica" direction="W"/>
    <neighbor name="Colombia" direction="E"/>
  </country>
</data>
```

`Element.remove()` を使って要素を削除することが出来ます。例えば rank が 50 より大きい全ての country を削除したい場合:

```
>>> for country in root.findall('country'):
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...
>>> tree.write('output.xml')
```

XML はこのようになります:

```
<?xml version="1.0"?>
<data>
  <country name="Liechtenstein">
    <rank updated="yes">2</rank>
    <year>2008</year>
    <gdppc>141100</gdppc>
    <neighbor name="Austria" direction="E"/>
    <neighbor name="Switzerland" direction="W"/>
  </country>
  <country name="Singapore">
    <rank updated="yes">5</rank>
    <year>2011</year>
    <gdppc>59900</gdppc>
    <neighbor name="Malaysia" direction="N"/>
  </country>
</data>
```

XML 文書の構築

`SubElement()` 関数は、与えられた要素に新しい子要素を作成する便利な手段も提供しています:

```
>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
>>> c = ET.SubElement(a, 'c')
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /><c><d /></c></a>
```

名前空間のある XML の解析

XML 入力 が **名前空間** を持っている場合、`prefix:sometag` の形式で修飾されたタグと属性が、その *prefix* が完全な URI で置換された `{uri}sometag` の形に展開されます。さらに、**デフォルトの XML 名前空間** があると、修飾されていない全てのタグにその完全 URI が前置されます。

ひとつは接頭辞 "fictional" でもうひとつがデフォルト名前空間で提供された、2 つの名前空間を組み込んだ XML の例をここにお見せします:

```
<?xml version="1.0"?>
<actors xmlns:fictional="http://characters.example.com"
  xmlns="http://people.example.com">
  <actor>
    <name>John Cleese</name>
    <fictional:character>Lancelot</fictional:character>
    <fictional:character>Archie Leach</fictional:character>
  </actor>
  <actor>
    <name>Eric Idle</name>
    <fictional:character>Sir Robin</fictional:character>
    <fictional:character>Gunther</fictional:character>
    <fictional:character>Commander Clement</fictional:character>
  </actor>
</actors>
```

この XML の例を、検索し、渡り歩くためのひとつの方法としては、`find()` や `findall()` に渡す xpath では全てのタグや属性に手作業で URI を付けてまわる手があります:

```
root = fromstring(xml_text)
for actor in root.findall('{http://people.example.com}actor'):
    name = actor.find('{http://people.example.com}name')
    print name.text
    for char in actor.findall('{http://characters.example.com}character'):
        print ' |-->', char.text
```

もっと良い方法があります。接頭辞の辞書を作り、これを検索関数で使うことです:

```
ns = {'real_person': 'http://people.example.com',
      'role': 'http://characters.example.com'}
```

(次のページに続く)

(前のページからの続き)

```

for actor in root.findall('real_person:actor', ns):
    name = actor.find('real_person:name', ns)
    print name.text
    for char in actor.findall('role:character', ns):
        print ' |-->', char.text

```

どちらのアプローチでも同じ結果です:

```

John Cleese
 |--> Lancelot
 |--> Archie Leach
Eric Idle
 |--> Sir Robin
 |--> Gunther
 |--> Commander Clement

```

その他の情報

<http://effbot.org/zone/element-index.htm> にはチュートリアルと他のドキュメントへのリンクがあります。

19.7.2 XPath サポート

このモジュールは木構造内の要素の位置決めのための [XPath 表現](#) を限定的にサポートしています。その目指すところは短縮構文のほんの一部だけのサポートであり、XPath エンジンのフルセットは想定していません。

例

以下はこのモジュールの XPath 機能の一部を紹介する例です。[XML の解析](#) 節から XML 文書 `countrydata` を使用します:

```

import xml.etree.ElementTree as ET

root = ET.fromstring(countrydata)

# Top-level elements
root.findall(".")

# All 'neighbor' grand-children of 'country' children of the top-level
# elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall("./year/..[@name='Singapore']")

# 'year' nodes that are children of nodes with name='Singapore'
root.findall("./*[@name='Singapore']/year")

```

(次のページに続く)

(前のページからの続き)

```
# All 'neighbor' nodes that are the second child of their parent
root.findall("./neighbor[2]")
```

サポートされている XPath 構文

操作	意味
tag	与えられたタグのすべての子要素を選択します。例えば、spam は spam と名付けられた子要素すべてを選択し、spam/egg は spam と名付けられた全子要素の中から egg と名付けられた孫要素をすべて選択します。
*	すべての子要素を選択します。例えば、*/egg は egg と名付けられた孫要素をすべて選択します。
.	現在のノードを選択します。これはパスの先頭に置くことで相対パスであることを示すのに役立ちます。
//	現在の要素配下のすべてのレベル上のすべての子要素を選択します。例えば、./egg は木全体から egg 要素を選択します。
..	親要素を選択します。
[@attrib]	与えられた属性を持つすべての要素を選択します。
[@attrib='value']	与えられた属性が、与えられた値を持つすべての要素を選択します。値に引用符が含まれてはなりません。
[tag]	tag と名付けられた子要素を持つすべての要素を選択します。隣接した子要素のみサポートしています。
[tag='text']	子孫を含む完全なテキストコンテンツと与えられた text が一致する、tag と名付けられた子要素を持つすべての要素を選択します。
[position]	与えられた位置にあるすべての要素を選択します。位置は整数 (1 が先頭)、表現 last() (末尾)、あるいは末尾からの相対位置 (例: last()-1) のいずれかで指定できます。

述部 (角括弧内の表現) の前にはタグ名、アスタリスク、あるいはその他の述部がなければなりません。
position 述部の前にはタグ名がなければなりません。

19.7.3 リファレンス

関数

xml.etree.ElementTree.**Comment** (text=None)

コメント要素のファクトリです。このファクトリ関数は、標準のシリアライザでは XML コメントにシリアライズされる特別な要素を作ります。コメント文字列はバイト文字列でも Unicode 文字列でも構いません。text はそのコメント文字列を含んだ文字列です。コメントを表わす要素のインスタンスを返します。

xml.etree.ElementTree.**dump** (elem)

要素の木もしくは要素の構造を `sys.stdout` に出力します。この関数はデバッグ目的のみに使用してください。

出力される形式の正確なところは実装依存です。このバージョンでは、通常の XML ファイルとして出力されます。

`elem` は要素の木もしくは個別の要素です。

`xml.etree.ElementTree.fromstring(text)`

文字列定数で与えられた XML 断片を解析します。 `XML()` 関数と同じです。 `text` は XML データの文字列です。 `Element` インスタンスを返します。

`xml.etree.ElementTree.fromstringlist(sequence, parser=None)`

文字列フラグメントのシーケンスから XML ドキュメントを解析します。 `sequence` は XML データのフラグメントを格納した、リストかその他のシーケンスです。 `parser` はオプションのパーザインスタンスです。パーザが指定されない場合、標準の `XMLParser` パーザが使用されます。 `Element` インスタンスを返します。

バージョン 2.7 で追加。

`xml.etree.ElementTree.iselement(element)`

オブジェクトが正当な要素オブジェクトであるかをチェックします。 `element` は要素インスタンスです。引数が要素オブジェクトならば真値を返します。

`xml.etree.ElementTree.iterparse(source, events=None, parser=None)`

XML 断片を構文解析して要素の木を漸増的に作っていき、その間進行状況をユーザーに報告します。 `source` は XML データを含むファイル名またはファイル風オブジェクト。 `events` は報告すべきイベントのリスト。省略された場合は "end" イベントだけが報告されます。 `parser` はオプションの引数で、パーサーのインスタンスを指定します。指定されなかった場合は標準の `XMLParser` が利用されます。 `parser` は `cElementTree` ではサポートされていません。 `(event, elem)` ペアのイテレータ (`iterator`) を返します。

注釈: `iterparse()` は "start" イベントを発行した時に開始タグの文字 ">" が現れたことだけを保証します。そのため、属性は定義されますが、その時点ではテキストの内容も `tail` 属性も定義されていません。同じことは子要素にも言えて、その時点ではあるともないとも言えません。

全部揃った要素が必要ならば、"end" イベントを探してください。

`xml.etree.ElementTree.parse(source, parser=None)`

XML 断片を解析して要素の木にします。 `source` には XML データを含むファイル名またはファイルオブジェクトを指定します。 `parser` はオプションでパーザインスタンスを指定します。パーザが指定されない場合、標準の `XMLParser` パーザが使用されます。 `ElementTree` インスタンスを返します。

`xml.etree.ElementTree.ProcessingInstruction(target, text=None)`

PI 要素のファクトリです。このファクトリ関数は XML の処理命令としてシリアル化された特別な要素を作成します。 `target` は PI ターゲットを含んだ文字列です。 `text` を指定する場合は PI コンテンツを含む文字列にします。PI を表わす要素インスタンスを返します。

`xml.etree.ElementTree.register_namespace(prefix, uri)`

名前空間の接頭辞を登録します。レジストリはグローバルで、与えられた接頭辞が名前空間 URI のどちらかの既存のマッピングはすべて削除されます。 *prefix* には名前空間の接頭辞を指定します。 *uri* には名前空間の URI を指定します。この名前空間のタグや属性は、可能な限り与えられた接頭辞をつけてシリアライズされます。

バージョン 2.7 で追加。

`xml.etree.ElementTree.SubElement(parent, tag, attrib={}, **extra)`

子要素のファクトリです。この関数は要素インスタンスを作成し、それを既存の要素に追加します。

要素名、属性名、および属性値はバイト文字列でも Unicode 文字列でも構いません。 *parent* には親要素を指定します。 *tag* には要素名を指定します。 *attrib* はオプションで要素の属性を含む辞書を指定します。 *extra* は追加の属性で、キーワード引数として与えます。要素インスタンスを返します。

`xml.etree.ElementTree.tostring(element, encoding="us-ascii", method="xml")`

XML 要素を全ての子要素を含めて表現する文字列を生成します。 *element* は *Element* のインスタンスです。 *encoding*^{*1} は出力エンコーディング (デフォルトは US-ASCII) です。 *method* は "xml", "html", "text" のいずれか (デフォルトは "xml") です。XML データを含んだエンコードされた文字列を返します。

`xml.etree.ElementTree.tostringlist(element, encoding="us-ascii", method="xml")`

XML 要素を全ての子要素を含めて表現する文字列を生成します。 *element* は *Element* のインスタンスです。 *encoding*^{*1} は出力エンコーディング (デフォルトは US-ASCII) です。 *method* は "xml", "html", "text" のいずれか (デフォルトは "xml") です。XML データを含んだエンコードされた文字列のリストを返します。これは、 `"".join(tostringlist(element)) == tostring(element)` であること以外、なにか特定のシーケンスになることは保証していません。

バージョン 2.7 で追加。

`xml.etree.ElementTree.XML(text, parser=None)`

文字列定数で与えられた XML 断片を解析します。この関数は Python コードに "XML リテラル" を埋め込むのに使えます。 *text* には XML データを含む文字列を指定します。 *parser* はオプションで、パーザのインスタンスを指定します。指定されなかった場合、標準の *XMLParser* パーザを使用します。 *Element* インスタンスを返します。

`xml.etree.ElementTree.XMLID(text, parser=None)`

文字列定数で与えられた XML 断片を解析し、要素 ID と要素を対応付ける辞書を返します。 *text* には XML データを含んだ文字列を指定します。 *parser* はオプションで、パーザのインスタンスを指定します。指定されなかった場合、標準の *XMLParser* パーザを使用します。 *Element* のインスタンスと辞書のタプルを返します。

^{*1} XML 出力に含まれるエンコーディング文字列は適切な規格に従っていなければなりません。例えば、"UTF-8" は有効ですが、"UTF8" はそうではありません。 <https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl> と <https://www.iana.org/assignments/character-sets/character-sets.xhtml> を参照してください。

Element オブジェクト

class xml.etree.ElementTree.**Element** (*tag*, *attrib*={}, ***extra*)

要素クラスです。この関数は Element インタフェースを定義すると同時に、そのリファレンス実装を提供します。

要素名、属性名、および属性値はバイト文字列でも Unicode 文字列でも構いません。 *tag* には要素名を指定します。 *attrib* はオプションで、要素と属性を含む辞書を指定します。 *extra* は追加の属性で、キーワード引数として与えます。要素インスタンスを返します。

tag

この要素が表すデータの種類を示す文字列です (言い替えると、要素の型です)。

text

tail

これらの属性は要素に結びつけられた付加的なデータを保持するのに使われます。これらの属性値はたいてい文字列ですが、アプリケーション固有のオブジェクトであって構いません。要素が XML ファイルから作られる場合、 *text* 属性は要素の開始タグとその最初の子要素または終了タグまでのテキストか、あるいは `None` を保持し、 *tail* 属性は要素の終了タグと次のタグまでのテキストか、あるいは `None` を保持します。このような XML データ

```
<a><b>1<c>2<d/>3</c></b>4</a>
```

の場合、 *a* 要素は *text*, *tail* 属性ともに `None`, *b* 要素は *text* に "1" で *tail* に "4", *c* 要素は *text* に "2" で *tail* は `None`, *d* 要素は *text* が `None` で *tail* に "3" をそれぞれ保持します。

要素の内側のテキストを収集するためには、 `itertext()` を参照してください。例えば `"".join(element.itertext())` のようにします。

アプリケーションはこれらの属性に任意のオブジェクトを格納できます。

attrib

要素の属性を保持する辞書です。 *attrib* の値は常に書き換え可能な Python 辞書ですが、ElementTree の実装によっては別の内部表現を使用し、要求されたときにだけ辞書を作るようにしているかもしれません。そうした実装の利益を享受するために、可能な限り下記の辞書メソッドを通じて使用してください。

以下の辞書風メソッドが要素の属性に対して動作します。

clear()

要素をリセットします。この関数は全ての子要素を削除し、全属性を消去し、テキストとテール属性を `None` に設定します。

get (*key*, *default=None*)

要素の *key* という名前の属性を取得します。

属性の値、または属性がない場合は *default* を返します。

items()

要素の属性を (名前, 値) ペアのシーケンスとして返します。返される属性の順番は決まっていま

せん。

keys ()

要素の属性名をリストとして返します。返される名前の順番は決まっています。

set (*key*, *value*)

要素の属性 *key* に *value* をセットします。

以下のメソッドは要素の子要素 (副要素) に対して動作します。

append (*subelement*)

要素 *subelement* をこの要素の内部にあるサブ要素のリストの最後に追加します。

extend (*subelements*)

シーケンスオブジェクト *subelements* から 0 個以上のサブ要素を追加します。サブ要素が有効なオブジェクトでない場合は `AssertionError` を発生させます。

バージョン 2.7 で追加.

find (*match*)

match にマッチする最初のサブ要素を探します。 *match* はタグ名かパス (path) です。要素・インスタンスまたは `None` を返します。

findall (*match*)

タグ名かパスにマッチする全てのサブ要素を探します。全てのマッチする要素を、ドキュメント上の順序で含むリストを返します。

findtext (*match*, *default=None*)

match にマッチする最初のサブ要素のテキストを探します。 *match* はタグ名かパスです。最初にマッチする要素の `text` を返すか、要素が見あたらなかった場合 *default* を返します。マッチした要素に `text` がなければ空文字列が返されるので気を付けましょう。

getchildren ()

バージョン 2.7 で非推奨: `list(elem)` がイテレーションを使用してください。

getiterator (*tag=None*)

バージョン 2.7 で非推奨: 代わりに `Element.iter()` メソッドを使用してください。

insert (*index*, *element*)

サブ要素をこの要素の与えられた位置に挿入します。

iter (*tag=None*)

現在の要素を根とする木の **イテレータ** を作成します。イテレータは現在の要素とそれ以下のすべての要素を、文書内での出現順 (深さ優先順) でイテレートします。 *tag* が `None` または `'*'` でない場合、与えられたタグに等しいものについてのみイテレータから返されます。イテレート中に木構造が変更された場合の結果は未定義です。

バージョン 2.7 で追加.

iterfind (*match*)

タグ名かパスにマッチする全てのサブ要素を探します。全てのマッチする要素をドキュメント上の

順序で `yield` するイテレート可能オブジェクトを返します。

バージョン 2.7 で追加.

itertext()

テキストのイテレータを作成します。イテレータは、この要素とすべての子要素を文書上の順序で巡回し、すべての内部のテキストを返します。

バージョン 2.7 で追加.

makeelement (*tag, attrib*)

現在の要素と同じ型の新しい要素オブジェクトを作成します。このメソッドは呼び出さずに、`SubElement()` ファクトリ関数を使って下さい。

remove (*subelement*)

要素から *subelement* を削除します。find* メソッド群と異なり、このメソッドは要素をインスタンスの同一性で比較します。タグや内容では比較しません。

`Element` オブジェクトは以下のシーケンス型のメソッドを、サブ要素を操作するためにサポートします: `object.__delitem__()`, `object.__getitem__()`, `object.__setitem__()`, `object.__len__()`.

注意: 子要素を持たない要素の真偽値は `False` になります。この挙動は将来のバージョンで変更されるかもしれません。直接真偽値をテストするのではなく、`len(elem)` が `elem is None` を利用してください。

```
element = root.find('foo')

if not element: # careful!
    print "element not found, or element has no subelements"

if element is None:
    print "element not found"
```

ElementTree オブジェクト

class `xml.etree.ElementTree.ElementTree` (*element=None, file=None*)

`ElementTree` ラッパークラスです。このクラスは要素の全階層を表現し、さらに標準 XML との相互変換を追加しています。

element は根要素です。木は *file* が与えられればその XML の内容により初期化されます。

setroot (*element*)

この木の根要素を置き換えます。従って現在の木の内容は破棄され、与えられた要素が代わりに使われます。注意して使ってください。 *element* は要素インスタンスです。

find (*match*)

`Element.find()` と同じで、木の根要素を起点とします。

findall (*match*)

Element.findall() と同じで、木の根要素を起点とします。

findtext (*match*, *default=None*)

Element.findtext() と同じで、木の根要素を起点とします。

getiterator (*tag=None*)

バージョン 2.7 で非推奨: 代わりに *ElementTree.iter()* メソッドを使用してください。

getroot ()

この木のルート要素を返します。

iter (*tag=None*)

根要素に対する、木を巡回するイテレータを返します。イテレータは木のすべての要素に渡ってセクション順にループします。*tag* は探したいタグです (デフォルトではすべての要素を返します)。

iterfind (*match*)

タグ名かパスにマッチする全てのサブ要素を返します。 *getroot().iterfind(match)* と同じです。全てのマッチする要素をドキュメント順に *yield* するイテレート可能オブジェクトを返します。

バージョン 2.7 で追加.

parse (*source*, *parser=None*)

外部の XML 断片をこの要素の木に読み込みます。 *source* は XML データを含むファイル名またはファイル風オブジェクト。 *parser* はオプションの構文解析器インスタンスです。これが与えられない場合、標準の XMLParser パーサーが使われます。断片の根要素を返します。

write (*file*, *encoding="us-ascii"*, *xml_declaration=None*, *default_namespace=None*, *method="xml"*)

要素の木をファイルに XML として書き込みます。 *file* はファイル名またはファイル風オブジェクトで書き込み用に開かれたもの。 *encoding*^{*1} は出力エンコーディング (デフォルトは US-ASCII) です。 *xml_declaration* は、XML 宣言がファイルに書かれるかどうかを制御します。 *False* の場合は常に書かれず、 *True* の場合は常に書かれ、 *None* の場合は US-ASCII か UTF-8 以外の場合に書かれます (デフォルトは *None* です)。 *default_namespace* でデフォルトの XML 名前空間 ("xmlns" 用) を指定します。 *method* は "xml", "html", "text" のいずれかです (デフォルトは "xml" です)。エンコードされた文字列を返します。

以下はこれから操作する XML ファイルです:

```
<html>
  <head>
    <title>Example page</title>
  </head>
  <body>
    <p>Moved to <a href="http://example.org/">example.org</a>
    or <a href="http://example.com/">example.com</a>.</p>
  </body>
</html>
```

第 1 段落のすべてのリンクの "target" 属性を変更する例:

```
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")      # Finds first occurrence of tag p in body
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))    # Returns list of all links
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77ec1cc>]
>>> for i in links:              # Iterates through all found links
...     i.attrib["target"] = "blank"
...
>>> tree.write("output.xhtml")
```

QName オブジェクト

class xml.etree.ElementTree.QName(*text_or_uri*, *tag=None*)

QName ラッパーです。このクラスは QName 属性値をラップし、出力時に適切な名前空間の扱いを得るために使われます。*text_or_uri* は {uri}local という形式の QName 値を含む文字列、または tag 引数を与えられた場合には QName の URI 部分の文字列です。*tag* が与えられた場合、一つめの引数は URI と解釈され、この引数はローカル名と解釈されます。QName インスタンスは不透明です。

TreeBuilder オブジェクト

class xml.etree.ElementTree.TreeBuilder(*element_factory=None*)

汎用の要素構造ビルダー。これは start, data, end のメソッド呼び出しの列を整形式の要素構造に変換します。このクラスを使うと、好みの XML 構文解析器、または他の XML に似た形式の構文解析器を使って、要素構造を作り出すことができます。*element_factory* が与えられた場合には、新しい *Element* インスタンスを作る際にこれ呼び出します。

close()

ビルダのバッファをフラッシュし、最上位の文書要素を返します。戻り値は *Element* インスタンスになります。

data(*data*)

現在の要素にテキストを追加します。*data* は文字列です。バイト文字列もしくは Unicode 文字列でなければなりません。

end(*tag*)

現在の要素を閉じます。*tag* は要素の名前です。閉じられた要素を返します。

start(*tag*, *attrs*)

新しい要素を開きます。*tag* は要素の名前です。*attrs* は要素の属性を保持した辞書です。開かれた要素を返します。

加えて、カスタムの *TreeBuilder* オブジェクトは以下のメソッドを提供できます:

doctype (*name, pubid, system*)

doctype 宣言を処理します。 *name* は doctype 名です。 *pubid* は公式の識別子です。 *system* はシステム識別子です。このメソッドはデフォルトの *TreeBuilder* クラスには存在しません。

バージョン 2.7 で追加.

XMLParser オブジェクト

class xml.etree.ElementTree.XMLParser (*html=0, target=None, encoding=None*)

expat パーサーを利用した、XML ソースデータからの *Element* 構造ビルダー。 *html* は定義済みの HTML エンティティです。このフラグは現在の実装ではサポートされていません。 *target* はターゲットオブジェクトです。省略された場合、ビルダーは標準の *TreeBuilder* クラスのインスタンスを利用します。 *encoding*^{*1} はオプションで、与えられた場合は XML ファイルで指定されたエンコーディングをオーバーライドします。

close ()

構文解析器にデータを供給するのを終わりにします。要素構造を返します。

doctype (*name, pubid, system*)

バージョン 2.7 で非推奨: カスタムの *TreeBuilder* *target* で *TreeBuilder.doctype()* メソッドを定義してください。

feed (*data*)

パーザへデータを入力します。 *data* はエンコードされたデータです。

XMLParser.feed() は *target* の *start()* メソッドをそれぞれの開始タグに対して呼び、また *end()* メソッドを終了タグに対して呼び、そしてデータは *data()* メソッドで処理されます。 *XMLParser.close()* は *target* の *close()* メソッドを呼びます。 *XMLParser* は木構造を構築する以外にも使えます。以下の例は、XML ファイルの最高の深さを数えます:

```
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:                                # The target object of the parser
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):               # Called for each opening tag.
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):                           # Called for each closing tag.
...         self.depth -= 1
...     def data(self, data):
...         pass                                     # We do not need to do anything with data.
...     def close(self):                             # Called when all data has been parsed.
...         return self.maxDepth
...
>>> target = MaxDepth()
>>> parser = XMLParser(target=target)
>>> exampleXml = """
... <a>
```

(次のページに続く)

(前のページからの続き)

```

...     <b>
...     </b>
...     <b>
...         <c>
...             <d>
...             </d>
...         </c>
...     </b>
... </a>"""
>>> parser.feed(exampleXml)
>>> parser.close()

```

4

19.8 xml.dom — 文書オブジェクトモデル (DOM) API

バージョン 2.0 で追加.

文書オブジェクトモデル (Document Object Model)、すなわち "DOM" は、ワールドワイドウェブコンソーシアム (World Wide Web Consortium, W3C) による、XML ドキュメントにアクセスしたり変更を加えたりするための、プログラミング言語間共通の API です。DOM 実装によって、XML ドキュメントはツリー構造として表現されます。また、クライアントコード側でツリー構造をゼロから構築できるようになります。さらに、前述の構造に対して、よく知られたインタフェースをもつ一連のオブジェクトを通したアクセス手段も提供します。

DOM はランダムアクセスを行うアプリケーションで非常に有用です。SAX では、一度に閲覧することができるのはドキュメントのほんの一部です。ある SAX 要素に注目している際には、別の要素にアクセスすることはできません。またテキストノードに注目しているときには、その中に入っている要素にアクセスすることができません。SAX によるアプリケーションを書くときには、プログラムがドキュメント内のどこを処理しているのかを追跡するよう、コードのどこかに記述する必要があります。SAX 自体がその作業を行ってくれることはありません。さらに、XML ドキュメントに対する先読み (look ahead) が必要だとすると不運なことになります。

アプリケーションによっては、ツリーにアクセスできなければイベント駆動モデルを実現できません。もちろん、何らかのツリーを SAX イベントに応じて自分で構築することもできるでしょうが、DOM ではそのようなコードを書かなくてもよくなります。DOM は XML データに対する標準的なツリー表現なのです。

文書オブジェクトモデルは、W3C によっていくつかの段階、W3C の用語で言えば "レベル (level)" で定義されています。Python においては、DOM API への対応付けは実質的には DOM レベル 2 勧告に基づいています。

DOM アプリケーションは、普通は XML を DOM に解析するところから始まります。どのようにして解析を行うかについては DOM レベル 1 では全くカバーしておらず、レベル 2 では限定的な改良だけが行われました: レベル 2 では `Document` を生成するメソッドを提供する `DOMImplementation` オブジェクトクラスがありますが、実装に依存しない方法で XML リーダ (reader)/パーザ (parser)/文書ビルダ (Document builder) にアクセスする方法はありません。また、既存の `Document` オブジェクトなしにこれらのメソッドにアクセスするような、よく定義された方法もありません。Python では、各々の DOM 実装で `getDOMImplementation()` が定義されているはずです。DOM レベル 3 ではロード (Load)/ストア (Store) 仕様が追加され、リーダーのイン

タフフェイスにを定義していますが、Python 標準ライブラリではまだ利用することができません。

DOM 文書オブジェクトを生成したら、そのプロパティとメソッドを使って XML 文書の一部にアクセスできます。これらのプロパティは DOM 仕様で定義されています; 本リファレンスマニュアルでは、Python において DOM 仕様がどのように解釈されているかを記述しています。

W3C から提供されている仕様は、DOM API を Java、ECMAScript、および OMG IDL で定義しています。ここで定義されている Python での対応づけは、大部分がこの仕様の IDL 版に基づいていますが、厳密な準拠は必要とされていません (実装で IDL の厳密な対応付けをサポートするのは自由ですが)。API への対応付けに関する詳細な議論は [適合性](#) を参照してください。

参考:

[Document Object Model \(DOM\) Level 2 Specification](#) Python DOM API が準拠している W3C 勧告。

[Document Object Model \(DOM\) Level 1 Specification](#) `xml.dom.minidom` でサポートされている W3C の DOM に関する勧告。

[Python Language Mapping Specification](#) このドキュメントでは OMG IDL から Python への対応付けを記述しています。

19.8.1 モジュールコンテンツ

`xml.dom` には以下の関数があります:

`xml.dom.registerDOMImplementation(name, factory)`

ファクトリ関数 (factory function) `factory` を名前 `name` で登録します。ファクトリ関数は `DOMImplementation` インタフェースを実装するオブジェクトを返さなければなりません。特定の実装 (例えば実装が何らかのカスタマイズをサポートしている場合) に適切となるように、ファクトリ関数は毎回同じオブジェクトを返したり、呼び出しごとに新しいオブジェクトを返したりすることが出来ます。

`xml.dom.getDOMImplementation([name[, features]])`

適切な DOM 実装を返します。`name` は、よく知られた DOM 実装のモジュール名か、`None` になります。`None` でない場合、対応するモジュールを `import` して、`import` が成功した場合 `DOMImplementation` オブジェクトを返します。`name` が与えられておらず、環境変数 `PYTHON_DOM` が設定されていた場合、DOM 実装を見つけるのに環境変数が使われます。

`name` が与えられない場合、利用可能な実装を調べて、指定された機能 (feature) セットを持つものを探します。実装が見つからなければ `ImportError` を送出します。`features` のリストは (feature, version) のペアからなるシーケンスで、利用可能な `DOMImplementation` オブジェクトの `hasFeature()` メソッドに渡されます。

いくつかの便利な定数も提供されています:

`xml.dom.EMPTY_NAMESPACE`

DOM 内のノードに名前空間が何も関連づけられていないことを示すために使われる値です。この

値は通常、ノードの `namespaceURI` の値として見つかったり、名前空間特有のメソッドに対する `namespaceURI` パラメタとして使われます。

バージョン 2.2 で追加.

`xml.dom.XML_NAMESPACE`

[Namespaces in XML](#) (4 節) で定義されている、予約済みプレフィクス (reserved prefix) `xml` に関連付けられた名前空間 URI です。

バージョン 2.2 で追加.

`xml.dom.XMLNS_NAMESPACE`

[Document Object Model \(DOM\) Level 2 Core Specification](#) (1.1.8 節) で定義されている、名前空間宣言への名前空間 URI です。

バージョン 2.2 で追加.

`xml.dom.XHTML_NAMESPACE`

[XHTML 1.0: The Extensible HyperText Markup Language](#) (3.1.1 節) で定義されている、XHTML 名前空間 URI です。

バージョン 2.2 で追加.

加えて、`xml.dom` には基底となる `Node` クラスと DOM 例外クラスが収められています。このモジュールで提供されている `Node` クラスは DOM 仕様で定義されているメソッドや属性は何ら実装していません; これらは具体的な DOM 実装において提供しなければなりません。このモジュールの一部として提供されている `Node` クラスでは、具体的な `Node` オブジェクトの `nodeType` 属性として使う定数を提供しています; これらの定数は、DOM 仕様に適合するため、クラスではなくモジュールのレベルに配置されています。

19.8.2 DOM 内のオブジェクト

DOM について最も明確に限定しているドキュメントは W3C による DOM 仕様です。

DOM 属性は単純な文字列としてだけではなく、ノードとして操作されるかもしれないので注意してください。とはいえ、そうしなければならない場合はかなり稀なので、今のところ記述されていません。

インタフェース	節	目的
<code>DOMImplementation</code>	<code>DOMImplementation</code> オブジェクト	根底にある実装へのインタフェース。
<code>Node</code>	<code>Node</code> オブジェクト	ドキュメント内の大部分のオブジェクトに対する基底インタフェース。
<code>NodeList</code>	<code>NodeList</code> オブジェクト	ノード列に対するインタフェース。
<code>DocumentType</code>	<code>DocumentType</code> オブジェクト	ドキュメントの処理に必要な宣言についての情報。
<code>Document</code>	<code>Document</code> オブジェクト	ドキュメント全体を表現するオブジェクト。
<code>Element</code>	<code>Element</code> オブジェクト	ドキュメント階層内の要素ノード。
<code>Attr</code>	<code>Attr</code> オブジェクト	要素ノード上の属性値ノード。
<code>Comment</code>	<code>Comment</code> オブジェクト	ソースドキュメント内のコメント表現。
<code>Text</code>	<code>Text</code> オブジェクトおよび <code>CDATASection</code> オブジェクト	ドキュメント内のテキスト記述を含むノード。
<code>ProcessingInstruction</code>	<code>ProcessingInstruction</code> オブジェクト	処理命令 (processing instruction) 表現。

さらに追加の節として、Python で DOM を利用するために定義されている例外について記述しています。

DOMImplementation オブジェクト

`DOMImplementation` インタフェースは、利用している DOM 実装において特定の機能が利用可能かどうかを決定するための方法をアプリケーションに提供します。DOM レベル 2 では、`DOMImplementation` を使って新たな `Document` オブジェクトや `DocumentType` オブジェクトを生成する機能も追加しています。

`DOMImplementation.hasFeature(feature, version)`

機能名 *feature* とバージョン番号 *version* で識別される機能 (feature) が実装されていれば真を返します。

`DOMImplementation.createDocument(namespaceUri, qualifiedName, doctype)`

新たな (DOM のスーパークラスである) `Document` クラスのオブジェクトを返します。このクラスは *namespaceUri* と *qualifiedName* が設定された子クラス `Element` のオブジェクトを所有しています。*doctype* は `createDocumentType()` によって生成された `DocumentType` クラスのオブジェクト、または `None` である必要があります。Python DOM API では、子クラスである `Element` を作成しないことを示すために、はじめの 2 つの引数を `None` に設定することができます。

`DOMImplementation.createDocumentType(qualifiedName, publicId, systemId)`

新たな `DocumentType` クラスのオブジェクトを返します。このオブジェクトは *qualifiedName*、*publicId*、そして *systemId* 文字列をふくんでおり、XML 文書の形式情報を表現しています。

Node オブジェクト

XML 文書の全ての構成要素は `Node` のサブクラスです。

`Node.nodeType`

ノード (node) の型を表現する整数値です。型に対応する以下のシンボル定

数: `ELEMENT_NODE`、`ATTRIBUTE_NODE`、`TEXT_NODE`、`CDATA_SECTION_NODE`、`ENTITY_NODE`、`PROCESSING_INSTRUCTION_NODE`、`COMMENT_NODE`、`DOCUMENT_NODE`、`DOCUMENT_TYPE_NODE`、`NOTATION_NODE`、が `Node` オブジェクトで定義されています。読み出し専用の属性です。

`Node.parentNode`

現在のノードの親ノードか、文書ノードの場合には `None` になります。この値は常に `Node` オブジェクトか `None` になります。Element ノードの場合、この値はルート要素 (root element) の場合を除き親要素 (parent element) となり、ルート要素の場合には `Document` オブジェクトとなります。Attr ノードの場合、この値は常に `None` となります。読み出し専用の属性です。

`Node.attributes`

属性オブジェクトの `NamedNodeMap` です。要素だけがこの属性に実際の値を持ちます; その他のオブジェクトでは、この属性を `None` にします。読み出し専用の属性です。

`Node.previousSibling`

このノードと同じ親ノードを持ち、直前にくるノードです。例えば、*self* 要素の開始タグの直前にくる終了タグを持つ要素です。もちろん、XML 文書は要素だけで構成されているだけではないので、直前にくる兄弟関係にある要素 (sibling) はテキストやコメント、その他になる可能性があります。このノードが親ノードにおける先頭の子ノードである場合、属性値は `None` になります。読み出し専用の属性です。

`Node.nextSibling`

このノードと同じ親ノードを持ち、直後にくるノードです。例えば、*previousSibling* も参照してください。このノードが親ノードにおける末尾頭の子ノードである場合、属性値は `None` になります。読み出し専用の属性です。

`Node.childNodes`

このノード内に収められているノードからなるリストです。読み出し専用の属性です。

`Node.firstChild`

このノードに子ノードがある場合、その先頭のノードです。そうでない場合 `None` になります。読み出し専用の属性です。

`Node.lastChild`

このノードに子ノードがある場合、その末尾のノードです。そうでない場合 `None` になります。読み出し専用の属性です。

`Node.localName`

`tagName` にコロンがあれば、コロン以降の部分に、なければ `tagName` 全体になります。値は文字列です。

`Node.prefix`

`tagName` のコロンがあれば、コロン以前の部分に、なければ空文字列になります。値は文字列か、`None` になります。

`Node.namespaceURI`

要素名に関連付けられた名前空間です。文字列か `None` になります。読み出し専用の属性です。

Node.nodeName

この属性はノード型ごとに異なる意味を持ちます; 詳しくは DOM 仕様を参照してください。この属性で得られることになる情報は、全てのノード型では `tagName`、属性では `name` プロパティといったように、常に他のプロパティで得ることができます。全てのノード型で、この属性の値は文字列か `None` になります。読み出し専用の属性です。

Node.nodeValue

この属性はノード型ごとに異なる意味を持ちます; 詳しくは DOM 仕様を参照してください。その序今日は `nodeName` と似ています。この属性の値は文字列か `None` になります。

Node.hasAttributes()

ノードが何らかの属性を持っている場合に真を返します。

Node.hasChildNodes()

ノードが何らかの子ノードを持っている場合に真を返します。

Node.isSameNode(*other*)

other がこのノードと同じノードを参照している場合に真を返します。このメソッドは、何らかのプロキシ (proxy) 機構を利用するような DOM 実装で特に便利です (一つ以上のオブジェクトが同じノードを参照するかもしれないからです)。

注釈: このメソッドは DOM レベル 3 API で提案されており、まだ "ワーキングドラフト (working draft)" の段階です。しかし、このインタフェースだけは議論にはならないと考えられます。W3C による変更は必ずしも Python DOM インタフェースにおけるこのメソッドに影響するとは限りません (ただしこのメソッドに対する何らかの新たな W3C API もサポートされるかもしれません)。

Node.appendChild(*newChild*)

現在のノードの子ノードリストの末尾に新たな子ノードを追加し、*newChild* を返します。もしノードが既にツリーにあれば、最初に削除されます。

Node.insertBefore(*newChild*, *refChild*)

新たな子ノードを既存の子ノードの前に挿入します。 *refChild* は現在のノードの子ノードである場合に限られます; そうでない場合、 `ValueError` が送出されます。 *newChild* が返されます。もし *refChild* が `None` なら、 *newChild* を子ノードリストの最後に挿入します。

Node.removeChild(*oldChild*)

子ノードを削除します。 *oldChild* はこのノードの子ノードでなければなりません。 そうでない場合、 `ValueError` が送出されます。 成功した場合 *oldChild* が返されます。 *oldChild* をそれ以降使わない場合、 `unlink()` メソッドを呼び出さなければなりません。

Node.replaceChild(*newChild*, *oldChild*)

既存のノードと新たなノードを置き換えます。この操作は *oldChild* が現在のノードの子ノードである場合に限られます; そうでない場合、 `ValueError` が送出されます。

Node.normalize()

一続きのテキスト全体を一個の `Text` インスタンスとして保存するために隣接するテキストノードを

結合します。これにより、多くのアプリケーションで DOM ツリーからのテキスト処理が簡単になります。

バージョン 2.1 で追加。

`Node.cloneNode(deep)`

このノードを複製 (clone) します。 *deep* を設定すると、子ノードも同様に複製することを意味します。複製されたノードを返します。

NodeList オブジェクト

`NodeList` はノードのシーケンスを表現します。これらのオブジェクトは DOM コア勧告 (DOM Core recommendation) において、二通りに使われています: `Element` オブジェクトでは、子ノードのリストを提供するのに `NodeList` を利用します。また、このインタフェースにおける `Node` の `getElementsByTagName()` および `getElementsByTagNameNS()` メソッドは、クエリに対する結果を表現するのに `NodeList` を利用します。

DOM レベル 2 勧告では、これらのオブジェクトに対し、メソッドと属性を一つずつ定義しています:

`NodeList.item(i)`

存在する場合シーケンスの *i* 番目の要素を、そうでない場合 `None` を返します。*i* は 0 未満やシーケンスの長さ以上であってはなりません。

`NodeList.length`

シーケンス中のノードの数です。

この他に、Python の DOM インタフェースでは、`NodeList` オブジェクトを Python のシーケンスとして使えるようにするサポートが追加されていることが必要です。`NodeList` の実装では、全て `__len__()` と `__getitem__()` をサポートしなければなりません; このサポートにより、`for` 文内で `NodeList` にわたる繰り返しと、組み込み関数 `len()` の適切なサポートができるようになります。

DOM 実装が文書の変更をサポートしている場合、`NodeList` の実装でも `__setitem__()` および `__delitem__()` メソッドをサポートしなければなりません。

DocumentType オブジェクト

文書で宣言されている記法 (notation) やエンティティ (entity) に関する (外部サブセット (external subset) がパーザから利用でき、情報を提供できる場合にはそれも含めた) 情報は、`DocumentType` オブジェクトから手に入れることができます。文書の `DocumentType` は、`Document` オブジェクトの `doctype` 属性で入手することができます; 文書の `DOCTYPE` 宣言がない場合、文書の `doctype` 属性は、このインタフェースを持つインスタンスの代わりに `None` に設定されます。

`DocumentType` は `Node` を特殊化したもので、以下の属性を加えています:

`DocumentType.publicId`

文書型定義 (document type definition) の外部サブセットに対する公開識別子 (public identifier) です。文字列または `None` になります。

`DocumentType.systemId`

文書型定義 (document type definition) の外部サブセットに対するシステム識別子 (system identifier) です。文字列の URI または `None` になります。

`DocumentType.internalSubset`

ドキュメントの完全な内部サブセットを与える文字列です。サブセットを囲むブラケットは含みません。ドキュメントが内部サブセットを持たない場合、この値は `None` です。

`DocumentType.name`

DOCTYPE 宣言でルート要素の名前が与えられている場合、その値になります。

`DocumentType.entities`

外部エンティティの定義を与える `NamedNodeMap` です。複数回定義されているエンティティに対しては、最初の定義だけが提供されます (その他は XML 勧告での要求仕様によって無視されます)。パーザによって情報が提供されないか、エンティティが定義されていない場合には、この値は `None` になることがあります。

`DocumentType.notations`

記法の定義を与える `NamedNodeMap` です。複数回定義されている記法名に対しては、最初の定義だけが提供されます (その他は XML 勧告での要求仕様によって無視されます)。パーザによって情報が提供されないか、エンティティが定義されていない場合には、この値は `None` になることがあります。

Document オブジェクト

`Document` は XML ドキュメント全体を表現し、その構成要素である要素、属性、処理命令、コメント等を持ちます。`Document` は `Node` からプロパティを継承していることを思い出してください。

`Document.documentElement`

ドキュメントの唯一無二のルート要素です。

`Document.createElement(tagName)`

新たな要素ノードを生成して返します。要素は、生成された時点ではドキュメント内に挿入されません。`insertBefore()` や `appendChild()` のような他のメソッドの一つを使って明示的に挿入を行う必要があります。

`Document.createElementNS(namespaceURI, tagName)`

名前空間を伴う新たな要素ノードを生成して返します。`tagName` には接頭辞 (prefix) があってもかまいません。要素は、生成された時点では文書内に挿入されません。`insertBefore()` や `appendChild()` のような他のメソッドの一つを使って明示的に挿入を行う必要があります。`appendChild()`。

`Document.createTextNode(data)`

引数として渡されたデータの入ったテキストノードを生成して返します。他の生成 (create) メソッドと同じく、このメソッドは生成されたノードをツリーに挿入しません。

`Document.createComment(data)`

引数として渡されたデータの入ったコメントノードを生成して返します。他の生成 (create) メソッドと同じく、このメソッドは生成されたノードをツリーに挿入しません。

`Document.createProcessingInstruction(target, data)`

引数として渡された *target* および *data* の入った処理命令ノードを生成して返します。他の生成 (`create`) メソッドと同じく、このメソッドは生成されたノードをツリーに挿入しません。

`Document.createAttribute(name)`

属性ノードを生成して返します。このメソッドは属性ノードを特定の要素に関連づけることはしません。新たに生成された属性インスタンスを使うには、適切な `Element` オブジェクトの `setAttributeNode()` を使わなければなりません。

`Document.createAttributeNS(namespaceURI, qualifiedName)`

名前空間を伴う新たな属性ノードを生成して返します。 *tagName* には接頭辞 (prefix) があってもかまいません。このメソッドは属性ノードを特定の要素に関連づけることはしません。新たに生成された属性インスタンスを使うには、適切な `Element` オブジェクトの `setAttributeNode()` を使わなければなりません。

`Document.getElementsByTagName(tagName)`

全ての下位要素 (直接の子要素、子要素の子要素等) から特定の要素型名を持つものを検索します。

`Document.getElementsByTagNameNS(namespaceURI, localName)`

全ての下位要素 (直接の子要素、子要素の子要素等) から特定の名前空間 URI とローカル名 (local name) を持つものを検索します。ローカル名は名前空間における接頭辞以降の部分です。

Element オブジェクト

`Element` は `Node` のサブクラスです。このため `Node` クラスの全ての属性を継承します。

`Element.tagName`

要素型名です。名前空間使用の文書では、要素型名中にコロンがあるかもしれません。値は文字列です。

`Element.getElementsByTagName(tagName)`

`Document` クラス内における同名のメソッドと同じです。

`Element.getElementsByTagNameNS(namespaceURI, localName)`

`Document` クラス内における同名のメソッドと同じです。

`Element.hasAttribute(name)`

指定要素に *name* で渡した名前の属性が存在していれば真を返します。

`Element.hasAttributeNS(namespaceURI, localName)`

指定要素に *namespaceURI* と *localName* で指定した名前の属性が存在していれば真を返します。

`Element.getAttribute(name)`

name で指定した属性の値を文字列として返します。もし、属性が存在しない、もしくは属性に値が設定されていない場合、空の文字列が返されます。

`Element.getAttributeNode(attrname)`

attrname で指定された属性の `Attr` ノードを返します。

`Element.getAttributeNS(namespaceURI, localName)`

`namespaceURI` と `localName` によって指定した属性の値を文字列として返します。もし、属性が存在しない、もしくは属性に値が設定されていない場合、空の文字列が返されます。

`Element.getAttributeNodeNS(namespaceURI, localName)`

指定した `namespaceURI` および `localName` を持つ属性値をノードとして返します。

`Element.removeAttribute(name)`

名前で指定された属性を削除します。該当する属性がない場合、`NotFoundError` が送出されます。

`Element.removeAttributeNode(oldAttr)`

`oldAttr` が属性リストにある場合、削除して返します。`oldAttr` が存在しない場合、`NotFoundError` が送出されます。

`Element.removeAttributeNS(namespaceURI, localName)`

名前で指定された属性を削除します。このメソッドは `qname` ではなく `localName` を使うので注意してください。該当する属性がなくても例外は送出されません。

`Element.setAttribute(name, value)`

文字列を使って属性値を設定します。

`Element.setAttributeNode(newAttr)`

新たな属性ノードを要素に追加します。`name` 属性が既存の属性に一致した場合、必要に応じて属性を置換します。置換が行われると古い属性ノードが返されます。`newAttr` がすでに使われていれば、`InuseAttributeErr` が送出されます。

`Element.setAttributeNodeNS(newAttr)`

新たな属性ノードを要素に追加します。`namespaceURI` および `localName` 属性が既存の属性に一致した場合、必要に応じて属性を置き換えます。置換が生じると、古い属性ノードが返されます。`newAttr` がすでに使われていれば、`InuseAttributeErr` が送出されます。

`Element.setAttributeNS(namespaceURI, qname, value)`

指定された `namespaceURI` および `qname` で与えられた属性の値を文字列で設定します。`qname` は属性の完全な名前であり、この点が上記のメソッドと違うので注意してください。

Attr オブジェクト

`Attr` は `Node` を継承しており、全ての属性を継承しています。

`Attr.name`

要素型名です。名前空間使用の文書では、要素型名中にコロンが含まれるかもしれません。

`Attr.localName`

名前にコロンがあればコロン以降の部分に、なければ名前全体になります。

`Attr.prefix`

名前にコロンがあればコロン以前の部分に、なければ空文字列になります。

`Attr.value`

その要素の text value. これは `nodeValue` 属性の別名です。

NamedNodeMap Objects

NamedNodeMap は Node を継承していません。

NamedNodeMap.length

属性リストの長さです。

NamedNodeMap.item(index)

特定のインデックスを持つ属性を返します。属性の並び方は任意ですが、DOM 文書が生成されている間は一定になります。各要素は属性ノードです。属性値はノードの value 属性で取得してください。

このクラスをよりマップ型の動作ができるようにする実験的なメソッドもあります。そうしたメソッドを使うこともできますし、Element オブジェクトに対して、標準化された getAttribute*() ファミリのメソッドを使うこともできます。

Comment オブジェクト

Comment は XML 文書中のコメントを表現します。Comment は Node のサブクラスですが、子ノードを持つことはありません。

Comment.data

文字列によるコメントの内容です。この属性には、コメントの先頭にある <!-- と末尾にある --> 間の全ての文字が入っていますが、<!-- と --> 自体は含みません。

Text オブジェクトおよび CDATASection オブジェクト

Text インタフェースは XML 文書内のテキストを表現します。パーザおよび DOM 実装が DOM の XML 拡張をサポートしている場合、CDATA でマークされた区域 (section) に入れられている部分テキストは CDATASection オブジェクトに記憶されます。これら二つのインタフェースは同一ののですが、nodeType 属性が異なります。

これらのインタフェースは Node インタフェースを拡張したものです。しかし子ノードを持つことはできません。

Text.data

文字列によるテキストノードの内容です。

注釈: CDATASection ノードの利用は、ノードが完全な CDATA マーク区域を表現するという意味ではなく、ノードの内容が CDATA 区域の一部であるということの意味するだけです。単一の CDATA セクションは文書ツリー内で複数のノードとして表現されることがあります。二つの隣接する CDATASection ノードが、異なる CDATA マーク区域かどうかを決定する方法はありません。

ProcessingInstruction オブジェクト

XML 文書内の処理命令を表現します; Node インタフェースを継承していますが、子ノードを持つことはできません。

`ProcessingInstruction.target`

最初の空白文字までの処理命令の内容です。読み出し専用の属性です。

`ProcessingInstruction.data`

最初の空白文字以降の処理命令の内容です。

例外

バージョン 2.1 で追加.

DOM レベル 2 勧告では、単一の例外 *DOMException* と、どの種のエラーが発生したかをアプリケーションが決定できるようにする多くの定数を定義しています。 *DOMException* インスタンスは、特定の例外に関する適切な値を提供する *code* 属性を伴っています。

Python DOM インタフェースでは、上記の定数を提供していますが、同時に一連の例外を拡張して、DOM で定義されている各例外コードに対して特定の例外が存在するようにしています。DOM の実装では、適切な特定の例外を送出しなければならない、各例外は *code* 属性に対応する適切な値を伴わなければならない。

exception `xml.dom.DOMException`

全ての特定の DOM 例外で使われている基底例外クラスです。この例外クラスを直接インスタンス化することはできません。

exception `xml.dom.DomstringSizeErr`

指定された範囲のテキストが文字列に収まらない場合に送出されます。この例外は Python の DOM 実装で使われるかどうかは判っていませんが、Python で書かれていない DOM 実装から送出される場合があります。

exception `xml.dom.HierarchyRequestErr`

挿入できない型のノードを挿入しようと試みたときに送出されます。

exception `xml.dom.IndexSizeErr`

メソッドに与えたインデックスやサイズパラメタが負の値や許容範囲の値を超えた際に送出されます。

exception `xml.dom.InuseAttributeErr`

文書中にすでに存在する `Attr` ノードを挿入しようと試みた際に送出されます。

exception `xml.dom.InvalidAccessErr`

パラメタまたは操作が根底にあるオブジェクトでサポートされていない場合に送出されます。

exception `xml.dom.InvalidCharacterErr`

この例外は、文字列パラメタが、現在使われているコンテキストで XML 1.0 勧告によって許可されていない場合に送出されます。例えば、要素型に空白の入った `Element` ノードを生成しようとすると、このエラーが送出されます。

exception `xml.dom.InvalidModificationErr`

ノードの型を変更しようと試みた際に送出されます。

exception `xml.dom.InvalidStateErr`

定義されていないオブジェクトや、もはや利用できなくなったオブジェクトを使おうと試みた際に送出されます。

exception `xml.dom.NamespaceErr`

[Namespaces in XML](#) に照らして許可されていない方法でオブジェクトを変更しようと試みた場合、この例外が送出されます。

exception `xml.dom.NotFoundErr`

参照しているコンテキスト中に目的のノードが存在しない場合に送出される例外です。例えば、`NamedNodeMap.removeNamedItem()` は渡されたノードがノードマップ中に存在しない場合にこの例外を送出します。

exception `xml.dom.NotSupportedErr`

要求された方のオブジェクトや操作が実装でサポートされていない場合に送出されます。

exception `xml.dom.NoDataAllowedErr`

データ属性をサポートしないノードにデータを指定した際に送出されます。

exception `xml.dom.NoModificationAllowedErr`

オブジェクトに対して (読み出し専用ノードに対する修正のように) 許可されていない修正を行おうと試みた際に送出されます。

exception `xml.dom.SyntaxErr`

無効または不正な文字列が指定された際に送出されます。

exception `xml.dom.WrongDocumentErr`

ノードが現在属している文書と異なる文書に挿入され、かつある文書から別の文書へのノードの移行が実装でサポートされていない場合に送出されます。

DOM 勧告で定義されている例外コードは、以下のテーブルに従って上記の例外と対応付けられます:

定数	例外
<code>DOMSTRING_SIZE_ERR</code>	<i>DomstringSizeErr</i>
<code>HIERARCHY_REQUEST_ERR</code>	<i>HierarchyRequestErr</i>
<code>INDEX_SIZE_ERR</code>	<i>IndexSizeErr</i>
<code>INUSE_ATTRIBUTE_ERR</code>	<i>InuseAttributeErr</i>
<code>INVALID_ACCESS_ERR</code>	<i>InvalidAccessErr</i>
<code>INVALID_CHARACTER_ERR</code>	<i>InvalidCharacterErr</i>
<code>INVALID_MODIFICATION_ERR</code>	<i>InvalidModificationErr</i>
<code>INVALID_STATE_ERR</code>	<i>InvalidStateErr</i>
<code>NAMESPACE_ERR</code>	<i>NamespaceErr</i>
<code>NOT_FOUND_ERR</code>	<i>NotFoundErr</i>
<code>NOT_SUPPORTED_ERR</code>	<i>NotSupportedErr</i>
<code>NO_DATA_ALLOWED_ERR</code>	<i>NoDataAllowedErr</i>
<code>NO_MODIFICATION_ALLOWED_ERR</code>	<i>NoModificationAllowedErr</i>
<code>SYNTAX_ERR</code>	<i>SyntaxErr</i>
<code>WRONG_DOCUMENT_ERR</code>	<i>WrongDocumentErr</i>

19.8.3 適合性

この節では適合性に関する要求と、Python DOM API、W3C DOM 勧告、および OMG IDL の Python API への対応付けとの間の関係について述べます。

型の対応付け

DOM 仕様で使われているプリミティブな IDL 型は、以下のテーブルに従って Python の型に対応付けられています。

IDL 型	Python の型
boolean	IntegerType (値 0 または 1) による
int	IntegerType
long int	IntegerType
unsigned int	IntegerType

さらに、勧告で定義されている `DOMString` は、Python 文字列または Unicode 文字列に対応付けられます。アプリケーションでは、DOM から文字列が返される際には常に Unicode を扱えなければなりません。

IDL の `null` 値は `None` に対応付けられており、API で `null` の使用が許されている場所では常に受理されるか、あるいは実装によって提供されるはずです。

アクセサメソッド

OMG IDL から Python への対応付けは、IDL *attribute* 宣言へのアクセサ関数の定義を、Java による対応付けが行うのとほとんど同じように行います。IDL 宣言の対応付け

```
readonly attribute string someValue;  
    attribute string anotherValue;
```

は、三つのアクセサ関数: `someValue` に対する `"get"` メソッド (`_get_someValue()`)、そして `anotherValue` に対する `"get"` および `"set"` メソッド (`_get_anotherValue()` および `_set_anotherValue()`) を生成します。とりわけ、対応付けでは、IDL 属性が通常の Python 属性としてアクセス可能であることは必須ではありません: `object.someValue` が動作することは必須ではなく、`AttributeError` を送出してもかまいません。

しかしながら、Python DOM API では、通常の属性アクセスが動作することが必須です。これは、Python IDL コンパイラによって生成された典型的なサロゲーションはまず動作することではなく、DOM オブジェクトが CORBA を解してアクセスされる場合には、クライアント上でラップオブジェクトが必要であることを意味します。CORBA DOM クライアントでは他にもいくつか考慮すべきことがある一方で、CORBA を介して DOM を使った経験を持つ実装者はこのことを問題視していません。readonly であると宣言された属性は、全ての DOM 実装で書き込みアクセスを制限しているとは限りません。

Python DOM API では、アクセサ関数は必須ではありません。アクセサ関数が提供された場合、Python IDL 対応付けによって定義された形式をとらなければなりませんが、属性は Python から直接アクセスすることが

できるので、それらのメソッドは必須ではないと考えられます。readonly であると宣言された属性に対しては、”set” アクセサを提供してはなりません。

この IDL での定義は W3C DOM API の全ての要件を実装しているわけではありません。例えば、一部のオブジェクトの概念や `getElementsByTagName()` が ”live” であることなどです。Python DOM API はこれらの要件を実装することを強制しません。

19.9 xml.dom.minidom — 最小限の DOM の実装

バージョン 2.0 で追加.

ソースコード: `Lib/xml/dom/minidom.py`

`xml.dom.minidom` は、Document Object Model インタフェースの最小の実装です。他言語の実装と似た API を持ちます。このモジュールは、完全な DOM に比べて単純で、非常に小さくなるように意図されています。DOM について既に熟知しているユーザを除き、XML 処理には代わりに `xml.etree.ElementTree` モジュールを使うことを検討すべきです。

警告: `xml.dom.minidom` モジュールは悪意を持って構築されたデータに対して安全ではありません。信頼できないデータや認証されていないデータを解析する必要がある場合は、[XML の脆弱性](#) を参照してください。

DOM アプリケーションは通常、XML を DOM に解析 (parse) することで開始します。 `xml.dom.minidom` では、以下のような解析用の関数を介して行います:

```
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\\temp\\mydata.xml') # parse an XML file by name

datasource = open('c:\\temp\\mydata.xml')
dom2 = parse(datasource) # parse an open file

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')
```

`parse()` 関数はファイル名か、開かれたファイルオブジェクトを引数にとることができます。

`xml.dom.minidom.parse(filename_or_file[, parser[, bufsize]])`

与えられた入力から Document を返します。 `filename_or_file` はファイル名でもファイルオブジェクトでもかまいません。 `parser` を指定する場合、SAX2 パーザオブジェクトでなければなりません。この関数はパーザの文書ハンドラを変更し、名前空間サポートを有効にします; (エンティティリゾルバ (entity resolver) のような) 他のパーザ設定は前もっておこなわなければなりません。

XML データを文字列で持っている場合、 `parseString()` を代わりに使うことができます:

```
xml.dom.minidom.parseString(string[, parser])
```

string を表わす Document を返します。このメソッドは、文字列に対する *StringIO* オブジェクトを作成し、それを *parse()* に渡します。

これらの関数は両方とも、文書の内容を表現する Document オブジェクトを返します。

parse() や *parseString()* といった関数が行うのは、XML パーザを、何らかの SAX パーザからくる解析イベント (parse event) を受け取って DOM ツリーに変換できるような "DOM ビルダ (DOM builder)" に結合することです。関数は誤解を招くような名前になっているかもしれませんが、インタフェースについて学んでいるときには理解しやすいでしょう。文書の解析はこれらの関数が戻るより前に完結します; 要するに、これらの関数自体はパーザ実装を提供しないということです。

"DOM 実装" オブジェクトのメソッドを呼び出して Document を生成することもできます。このオブジェクトは、*xml.dom* パッケージ、または *xml.dom.minidom* モジュールの *getDOMImplementation()* 関数を呼び出して取得できます。*xml.dom.minidom* モジュールの実装を使うと、常に *minidom* 実装の Document インスタンスを返します。一方、*xml.dom* 版の関数では、別の実装によるインスタンスを返すかもしれません (PyXML package がインストールされているとそうなるでしょう)。Document を取得したら、DOM を構成するために子ノードを追加していくことができます:

```
from xml.dom.minidom import getDOMImplementation

impl = getDOMImplementation()

newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement
text = newdoc.createTextNode('Some textual content.')
top_element.appendChild(text)
```

DOM 文書オブジェクトを手にしたら、XML 文書のプロパティやメソッドを使って、文書の一部にアクセスすることができます。これらのプロパティは DOM 仕様で定義されています。文書オブジェクトの主要なプロパティは *documentElement* プロパティです。このプロパティは XML 文書の主要な要素: 他の全ての要素を保持する要素、を与えます。以下にプログラム例を示します:

```
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

DOM ツリーを使い終えたとき、*unlink()* メソッドを呼び出して不要になったオブジェクトが早く片付けられるように働きかけることができます。*unlink()* は、DOM API に対する *xml.dom.minidom* 特有の拡張です。ノードに対して *unlink()* を呼び出した後は、ノードとその下位ノードは本質的には無意味なものとなります。このメソッドを呼び出さなくても、Python のガベージコレクタがいつかはツリーのオブジェクトを後片付けします。

参考:

Document Object Model (DOM) Level 1 Specification *xml.dom.minidom* でサポートされている W3C の DOM に関する勧告。

19.9.1 DOM オブジェクト

Python の DOM API 定義は `xml.dom` モジュールドキュメントの一部として与えられています。この節では、`xml.dom` の API と `xml.dom.minidom` との違いについて列挙します。

`Node.unlink()`

DOM との内部的な参照を破壊して、循環参照ガベージコレクションを持たないバージョンの Python でもガベージコレクションされるようにします。循環参照ガベージコレクションが利用できる場合でも、このメソッドを使えば大量のメモリをすぐに使えるようにできるため、不要になったらすぐに DOM オブジェクトに対してこのメソッドを呼ぶのが良い習慣です。このメソッドは `Document` オブジェクトに対して呼び出すだけでよいのですが、あるノードの子ノードを破棄するために子ノードに対して呼び出してもかまいません。

`Node.writexml(writer, indent="", addindent="", newl="")`

XML を `writer` オブジェクトに書き込みます。 `writer` は、ファイルオブジェクトインタフェースの `write()` に該当するメソッドを持たなければなりません。 `indent` 引数には現在のノードのインデントを指定します。 `addindent` 引数には現在のノードの下にサブノードを追加する際のインデント増分を指定します。 `newl` には、改行時に行末を終端する文字列を指定します。

`Document` ノードでは、追加のキーワード引数 `encoding` を使って XML ヘッダの `encoding` フィールドを指定することができます。

バージョン 2.1 で変更: 美しい出力をサポートするため、新たなキーワード引数 `indent`、`addindent`、および `newl` が追加されました。

バージョン 2.3 で変更: `Document` ノードでは、追加のキーワード引数 `encoding` を使って XML ヘッダの `encoding` フィールドを指定することができます。

`Node.toxml([encoding])`

DOM が表現している XML を文字列にして返します。

引数がなければ、XML ヘッダは `encoding` を指定せず、文書内の全ての文字をデフォルトエンコード方式で表示できない場合、結果は Unicode 文字列となります。この文字列を UTF-8 以外のエンコード方式でエンコードするのは不正であり、なぜなら UTF-8 が XML のデフォルトエンコード方式だからです。

明示的な `encoding`^{*1} 引数があると、結果は指定されたエンコード方式によるバイト文字列となります。引数を常に指定するよう推奨します。表現不可能なテキストデータの場合に `UnicodeError` が送出されるのを避けるため、`encoding` 引数は `"utf-8"` に指定するべきです。

バージョン 2.3 で変更: `encoding` が追加されました。 `writexml()` を参照して下さい。

`Node.toprettyxml(indent="\t", newl="\n", encoding=None)`

文書の整形されたバージョンを返します。 `indent` はインデントを行うための文字で、デフォルトはタブです; `newl` には行末で出力される文字列を指定し、デフォルトは `\n` です。

^{*1} XML 出力に含まれるエンコーディング文字列は適切な規格に従っていなければなりません。例えば、`"UTF-8"` は有効ですが、`"UTF8"` はそうではありません。 <https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl> と <https://www.iana.org/assignments/character-sets/character-sets.xhtml> を参照してください。

バージョン 2.1 で追加.

バージョン 2.3 で変更: encoding 引数が追加されました。 `writexml()` を参照して下さい。

以下の標準 DOM メソッドは、 `xml.dom.minidom` では特別な注意をする必要があります:

`Node.cloneNode(deep)`

このメソッドは Python 2.0 にパッケージされているバージョンの `xml.dom.minidom` にはありましたが、これには深刻な障害があります。以降のリリースでは修正されています。

19.9.2 DOM の例

以下のプログラム例は、単純なプログラムのかなり現実的な例です。特にこの例に関しては、DOM の柔軟性をあまり活用してはいません。

```
import xml.dom.minidom

document = """\
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point>
</slide>

<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>
"""

dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = []
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc.append(node.data)
    return ''.join(rc)

def handleSlideshow(slideshow):
    print "<html>"
    handleSlideshowTitle(slideshow.getElementsByTagName("title")[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print "</html>"

def handleSlides(slides):
```

(次のページに続く)

(前のページからの続き)

```

    for slide in slides:
        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print "<title>%s</title>" % getText(title.childNodes)

def handleSlideTitle(title):
    print "<h2>%s</h2>" % getText(title.childNodes)

def handlePoints(points):
    print "<ul>"
    for point in points:
        handlePoint(point)
    print "</ul>"

def handlePoint(point):
    print "<li>%s</li>" % getText(point.childNodes)

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print "<p>%s</p>" % getText(title.childNodes)

handleSlideshow(dom)

```

19.9.3 minidom と DOM 標準

`xml.dom.minidom` モジュールは、本質的には DOM 1.0 互換の DOM に、いくつかの DOM 2 機能 (主に名前空間機能) を追加したものです。

Python における DOM インタフェースは率直なものです。以下の対応付け規則が適用されます:

- インタフェースはインスタンスオブジェクトを介してアクセスされます。アプリケーション自身から、クラスをインスタンス化してはなりません; `Document` オブジェクト上で利用可能な生成関数 (creator function) を使わなければなりません。派生インタフェースでは基底インタフェースの全ての演算 (および属性) に加え、新たな演算をサポートします。
- 演算はメソッドとして使われます。DOM では `in` パラメタのみを使うので、引数は通常の順番 (左から右へ) で渡されます。オプション引数はありません。void 演算は `None` を返します。
- IDL 属性はインスタンス属性に対応付けられます。OMG IDL 言語における Python への対応付けとの互換性のために、属性 `foo` はアクセサメソッド `_get_foo()` および `_set_foo()` でもアクセスできます。readonly 属性は変更してはなりません; とはいえ、これは実行時には強制されません。
- `short int`、`unsigned int`、`unsigned long long`、および `boolean` 型は、全て Python

整数オブジェクトに対応付けられます。

- `DOMString` 型は Python 文字列型に対応付けられます。 `xml.dom.minidom` ではバイト文字列 (byte string) および Unicode 文字列のどちらかに対応づけられますが、通常 Unicode 文字列を生成します。 `DOMString` 型の値は、W3C の DOM 仕様で、IDL `null` 値になってもよいとされている場所では `None` になることもあります。
- `const` 宣言を行うと、(`xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE` のように) 対応するスコープ内の変数に対応付けを行います; これらは変更してはなりません。
- `DOMException` は現状では `xml.dom.minidom` でサポートされていません。その代わり、`xml.dom.minidom` は、`TypeError` や `AttributeError` といった標準の Python 例外を使います。
- `NodeList` オブジェクトは Python の組み込みリスト型を使って実装されています。Python 2.2 からは、これらのオブジェクトは DOM 仕様で定義されたインタフェースを提供していますが、それ以前のバージョンの Python では、公式の API をサポートしていません。しかしながら、これらの API は W3C 勧告で定義されたインタフェースよりも "Python 的な" ものになっています。

以下のインタフェースは `xml.dom.minidom` では全く実装されていません:

- `DOMTimeStamp`
- `DocumentType` (added in Python 2.1)
- `DOMImplementation` (added in Python 2.1)
- `CharacterData`
- `CDATASection`
- `Notation`
- `Entity`
- `EntityReference`
- `DocumentFragment`

これらの大部分は、ほとんどの DOM のユーザにとって一般的な用途として有用とはならないような XML 文書内の情報を反映しています。

19.10 `xml.dom.pulldom` — 部分的な DOM ツリー構築のサポート

バージョン 2.0 で追加.

Source code: <Lib/xml/dom/pulldom.py>

`xml.dom.pulldom` では、SAX イベントから、文書の文書オブジェクトモデル表現の選択された一部分だけを構築できるようにします。

警告: `xml.dom.pulldom` モジュールは悪意を持って構築されたデータに対して安全ではありません。信頼できないデータや認証されていないデータを解析する必要がある場合は、[XML の脆弱性](#) を参照してください。

```
class xml.dom.pulldom.PullDOM([documentFactory])
    xml.sax.handler.ContentHandler 実装です ...

class xml.dom.pulldom.DOMEventStream(stream, parser, bufsize)
    ...

class xml.dom.pulldom.SAX2DOM([documentFactory])
    xml.sax.handler.ContentHandler 実装です ...

xml.dom.pulldom.parse(stream_or_string[, parser[, bufsize]])
    ...

xml.dom.pulldom.parseString(string[, parser])
    ...

xml.dom.pulldom.default_bufsize
    parse() の bufsize パラメタのデフォルト値です。
```

バージョン 2.1 で変更: この変数の値は `parse()` を呼び出す前に変更することができ、その場合新たな値が効果を持つようになります。

19.10.1 DOMEventStream オブジェクト

```
DOMEventStream.getEvent()
    ...

DOMEventStream.expandNode(node)
    ...

DOMEventStream.reset()
    ...
```

19.11 xml.sax — SAX2 パーサのサポート

バージョン 2.0 で追加。

`xml.sax` パッケージは Python 用の Simple API for XML (SAX) インターフェースを実装した数多くのモジュールを提供しています。またパッケージには SAX 例外と SAX API 利用者が頻繁に利用するであろう有用な関数群も含まれています。

警告: `xml.sax` モジュールは悪意を持って構築されたデータに対して安全ではありません。信頼できないデータや認証されていないデータを解析する必要があるばあいは、[XML の脆弱性](#) を参照してください。

その関数群は以下の通りです:

`xml.sax.make_parser([parser_list])`

SAX `XMLReader` オブジェクトを生成し、返します。最初に見つかったパーサが使用されます。`parser_list` を与える場合、それは `create_parser()` という名前の関数をもつモジュール名のリストでなければなりません。`parser_list` に列挙されているモジュールは、パーサのデフォルトリストにあるモジュールよりも先に使われます。

`xml.sax.parse(filename_or_stream, handler[, error_handler])`

SAX パーサを生成し、ドキュメントの解析に使用します。`filename_or_stream` として与えられるドキュメントは、ファイル名でもファイルオブジェクトでもかまいません。`handler` パラメータは SAX `ContentHandler` のインスタントである必要があります。もし `error_handler` が与える場合は、それは SAX `ErrorHandler` のインスタンスでなければなりません。引数で指定しない場合は、例外 `SAXParseException` を発生します。戻り値はありません。すべての動作は `handler` が行われなければなりません。

`xml.sax.parseString(string, handler[, error_handler])`

`parse()` に似ていますが、こちらはパラメータ `string` で指定されたバッファをパースします。

典型的な SAX アプリケーションでは 3 種類のオブジェクト (リーダ、ハンドラ、入力元) が用いられます (ここで言うリーダとはパーサを指しています)。言い換えると、プログラムはまず入力元からバイト列、あるいは文字列を読み込み、一連のイベントを発生させます。発生したイベントはハンドラ・オブジェクトによって振り分けられます。さらに言い換えると、リーダがハンドラのメソッドを呼び出すわけです。つまり SAX アプリケーションには、リーダ・オブジェクト、(作成またはオープンされる) 入力元のオブジェクト、ハンドラ・オブジェクト、そしてこれら 3 つのオブジェクトを連携させることが必須なのです。前処理の最後の段階でリーダは入力をパースするために呼び出されます。パースの過程で入力データの構造、構文にもとづいたイベントにより、ハンドラ・オブジェクトのメソッドが呼び出されます。

これらのオブジェクトでは、インタフェースだけに関係があります。通常は、これらはアプリケーション自体はインスタンス化しません。Python は明示的なインタフェースの概念を持たないので、形式的にはクラスとして導入されます。しかし、アプリケーションは提供されたクラスを継承せずに実装をしてもかまいません。インタフェース `InputSource`, `Locator`, `Attributes`, `AttributesNS`, `XMLReader` はモジュール `xml.sax.xmlreader`。The handler interfaces are defined in `xml.sax.handler` で定義されています。ハンドラインタフェースは `xml.sax.handler` で定義されています。便利なので、`InputSource` (よく直接インスタンス化されるクラス) とハンドラクラスは `xml.sax` からアクセスできます。これらのインタフェースは下記で説明します。

このほかに `xml.sax` は次の例外クラスも提供しています。

exception `xml.sax.SAXException(msg[, exception])`

XML エラーと警告をカプセル化します。このクラスには XML パーサとアプリケーションで発生するエラーおよび警告の基本的な情報を持たせることができます。また機能追加や地域化のためにサブクラス化することも可能です。なお `ErrorHandler` で定義されているハンドラがこの例外のインスタ

スを受け取ることに注意してください。実際に例外を発生させることは必須でなく、情報のコンテナとして利用されることもあるからです。

インスタンスを作成する際 `msg` はエラー内容を示す可読データにしてください。オプションの `exception` パラメータは `None` もしくはパース用コードで補足、渡って来る情報でなければなりません。

このクラスは SAX 例外の基底クラスになります。

exception `xml.sax.SAXParseException` (`msg`, `exception`, `locator`)

パースエラー時に発生する `SAXException` のサブクラスです。パースエラーに関する情報として、このクラスのインスタンスが SAX `ErrorHandler` インターフェースのメソッドに渡されます。このクラスは `SAXException` 同様 SAX `Locator` インターフェースもサポートしています。

exception `xml.sax.SAXNotRecognizedException` (`msg`[, `exception`])

SAX `XMLReader` が認識できない機能やプロパティに遭遇したとき発生させる `SAXException` のサブクラスです。SAX アプリケーションや拡張モジュールにおいて同様の目的にこのクラスを利用することもできます。

exception `xml.sax.SAXNotSupportedException` (`msg`[, `exception`])

SAX `XMLReader` が要求された機能をサポートしていないとき発生させる `SAXException` のサブクラスです。SAX アプリケーションや拡張モジュールにおいて同様の目的にこのクラスを利用することもできます。

参考:

SAX: The Simple API for XML SAX API 定義に関し中心となっているサイトです。Java による実装とオンライン・ドキュメントが提供されています。実装と SAX API の歴史に関する情報のリンクも掲載されています。

`xml.sax.handler` モジュール アプリケーションが提供するオブジェクトのインターフェース定義。

`xml.sax.saxutils` モジュール SAX アプリケーション向けの有用な関数群。

`xml.sax.xmlreader` モジュール パーサが提供するオブジェクトのインターフェース定義。

19.11.1 SAXException オブジェクト

`SAXException` 例外クラスは以下のメソッドをサポートしています:

`SAXException.getMessage()`

エラー状態を示す可読メッセージを返します。

`SAXException.getException()`

カプセル化した例外オブジェクトまたは `None` を返します。

19.12 xml.sax.handler — SAX ハンドラの基底クラス

バージョン 2.0 で追加.

SAX API はコンテンツ・ハンドラ、DTD ハンドラ、エラー・ハンドラ、エンティティ・リゾルバという 4 つのハンドラを規定しています。通常アプリケーション側で実装する必要があるのは、これらのハンドラが発生させるイベントのうち、処理したいものへのインターフェースだけです。インターフェースは 1 つのオブジェクトにまとめることも、複数のオブジェクトに分けることも可能です。ハンドラはすべてのメソッドがデフォルトで実装されるように、`xml.sax.handler` で提供される基底クラスを継承しなくてはなりません。

class `xml.sax.handler.ContentHandler`

アプリケーションにとって最も重要なメインの SAX コールバック・インターフェースです。このインターフェースで発生するイベントの順序はドキュメント内の情報の順序を反映しています。

class `xml.sax.handler.DTDHandler`

DTD イベントのハンドラです。

パースされていないエンティティや属性など、基本的なパースに必要な DTD イベントの指定だけを行うインターフェースです。

class `xml.sax.handler.EntityResolver`

エンティティ解決用の基本インターフェースです。このインターフェースを実装したオブジェクトを作成しパーサに登録することで、パーサはすべての外部エンティティを解決するメソッドを呼び出すようになります。

class `xml.sax.handler.ErrorHandler`

エラーや警告メッセージをアプリケーションに通知するためにパーサが使用するインターフェースです。このオブジェクトのメソッドが、エラーをただちに例外に変換するか、あるいは別の方法で処理するかを制御をしています。

これらのクラスに加え、`xml.sax.handler` は機能やプロパティ名のシンボル定数を提供しています。

`xml.sax.handler.feature_namespaces`

値: "http://xml.org/sax/features/namespaces"

真: 名前空間を処理します。

偽: オプションで名前空間を処理しません (暗黙に名前空間接頭辞も無効にします; デフォルト)。

アクセス: (パース時) 読み込み専用; (パース時以外) 読み書き可

`xml.sax.handler.feature_namespace_prefixes`

値: "http://xml.org/sax/features/namespace-prefixes"

真: 名前空間宣言で用いられている元々の接頭辞付きの名前と属性を報告します。

偽: 名前空間宣言で用いられている属性を報告しません。オプションで元々の接頭辞付きの名前も報告しません (デフォルト)。

アクセス: (パース時) 読み込み専用; (パース時以外) 読み書き可

`xml.sax.handler.feature_string_interning`

値: "http://xml.org/sax/features/string-interning"

真: 全ての要素名、接頭辞、属性名、名前空間 URI、ローカル名を組み込みの `intern` 関数を使ってシンボルに登録します。

偽: 名前を必ずしもシンボルに登録しませんが、されるかもしれません (デフォルト)。

アクセス: (パース時) 読み込み専用; (パース時以外) 読み書き可

`xml.sax.handler.feature_validation`

値: `"http://xml.org/sax/features/validation"`

真: 全ての妥当性検査エラーを報告します (外部一般エンティティと外部変数エンティティが暗示します)。

偽: 妥当性検査エラーを報告しません。

アクセス: (パース時) 読み込み専用; (パース時以外) 読み書き可

`xml.sax.handler.feature_external_ges`

値: `"http://xml.org/sax/features/external-general-entities"`

真: 全ての外部一般 (テキスト) エンティティを取り込みます。

偽: 外部一般エンティティを取り込みません。

アクセス: (パース時) 読み込み専用; (パース時以外) 読み書き可

`xml.sax.handler.feature_external_pes`

値: `"http://xml.org/sax/features/external-parameter-entities"`

真: 外部 DTD サブセットを含む全ての外部変数エンティティを取り込みます。

偽: 外部 DTD サブセットであっても外部変数エンティティを取り込みません。

アクセス: (パース時) 読み込み専用; (パース時以外) 読み書き可

`xml.sax.handler.all_features`

全機能のリスト。

`xml.sax.handler.property_lexical_handler`

値: `"http://xml.org/sax/properties/lexical-handler"`

データ型: `xml.sax.sax2lib.LexicalHandler` (Python 2 では未サポート)

説明: コメントなど字句解析イベント用のオプション拡張ハンドラ。

アクセス: 読み書き可

`xml.sax.handler.property_declaration_handler`

値: `"http://xml.org/sax/properties/declaration-handler"`

データ型: `xml.sax.sax2lib.DeclHandler` (Python 2 では未サポート)

説明: 表記や未解析エンティティをのぞく DTD 関連イベント用のオプション拡張ハンドラ。

アクセス: 読み書き可

`xml.sax.handler.property_dom_node`

値: "http://xml.org/sax/properties/dom-node"

データ型: `org.w3c.dom.Node` (Python 2 では未サポート)

説明: パース時は DOM イテレータならば現在の DOM ノードです。非パース時はイテレータのルート DOM ノードです。

アクセス: (パース時) 読み込み専用; (パース時以外) 読み書き可

`xml.sax.handler.property_xml_string`

値: "http://xml.org/sax/properties/xml-string"

データ型: 文字列

説明: 現在のイベントの元になったリテラル文字列。

アクセス: 読み込み専用

`xml.sax.handler.all_properties`

既知の全プロパティ名のリスト。

19.12.1 ContentHandler オブジェクト

`ContentHandler` はアプリケーション側でサブクラス化して利用することが前提になっています。パーサは入力ドキュメントのイベントにより、それぞれに対応する以下のメソッドを呼び出します:

`ContentHandler.setDocumentLocator(locator)`

アプリケーションにドキュメントイベントの発生位置を指すロケータを与えるためにパーサから呼び出されます。

SAX パーサによるロケータの提供は強く推奨されています (必須ではありません)。もし提供する場合は、`DocumentHandler` インターフェースのどのメソッドよりも先にこのメソッドが呼び出されるようにしなければなりません。

パーサがエラーを報告しない場合でも、ロケータによってアプリケーションは全てのドキュメント関連イベントの終了位置を知ることが出来ます。通常、アプリケーションは自身のエラー (例えば文字コンテンツがアプリケーションの規則に適合しない場合) を報告するためにこれを使用します。ロケータが返す情報は検索エンジンでの利用にはおそらく不十分です。

ロケータが正しい情報を返すのは、このインターフェースからイベントの呼出しが実行されている間だけです。それ以外のときは使用すべきではありません。

`ContentHandler.startDocument()`

ドキュメントの開始通知を受け取ります。

SAX パーサはこのインターフェースや `DTDHandler` のどのメソッド (`setDocumentLocator()` を除く) よりも先にこのメソッドを一度だけ呼び出します。

`ContentHandler.endDocument()`

ドキュメントの終了通知を受け取ります。

SAX パーサはこのメソッドを一度だけ呼び出します。パース中に呼び出す最後のメソッドです。パーサは (回復不能なエラーで) パース処理を中断するか、あるいは入力のために到達するまでこのメソッドを呼び出しません。

`ContentHandler.startPrefixMapping(prefix, uri)`

接頭辞と URI 名前空間の関連付けのスコープを開始します。

このイベントからの情報は名前空間処理に必須ではありません。SAX XML リーダは `feature.namespaces` 機能が有効な場合 (デフォルト)、要素と属性名の接頭辞を自動的に置換します。

しかしながら、接頭辞の自動展開を安全に行えないために、アプリケーションが文字データや属性値の中で接頭辞を使わなければならない場合があります。必要ならば `startPrefixMapping()` や `endPrefixMapping()` イベントはアプリケーションにコンテキスト自身の中で接頭辞を展開するための情報を提供します。

`startPrefixMapping()` と `endPrefixMapping()` イベントは相互に正しい入れ子関係になることが保証されていないので注意が必要です。すべての `startPrefixMapping()` は対応する `startElement()` の前に発生し、`endPrefixMapping()` イベントは対応する `endElement()` の後で発生しますが、その順序は保証されていません。

`ContentHandler.endPrefixMapping(prefix)`

接頭辞と URI の関連付けのスコープを終了します。

詳しくは `startPrefixMapping()` を参照してください。このイベントは常に対応する `endElement()` の後で発生しますが、複数の `endPrefixMapping()` イベントの順序は特に保証されません。

`ContentHandler.startElement(name, attrs)`

非名前空間モードでの要素の開始を通知します。

`name` 引数は要素型の生の XML 1.0 名を文字列として持ち、`attrs` 引数は要素の属性を持つ `Attributes` インターフェイス (`Attributes` インタフェース を参照) のオブジェクトを保持します。`attrs` として渡されたオブジェクトはパーサに再利用されるかもしれません。そのため、それへの参照を確保するのは属性のコピーを保持する確実な方法ではありません。属性のコピーを保持するには `attrs` 属性の `copy()` メソッドを使用してください。

`ContentHandler.endElement(name)`

非名前空間モードでの要素の終了を通知します。

`name` 引数は `startElement()` イベントとまったく同じ要素型名を持ちます。

`ContentHandler.startElementNS(name, qname, attrs)`

名前空間モードでの要素の開始を通知します。

`name` 引数は要素型の名前を `(uri, localname)` というタプルとして持ち、`qname` 引数は元の文書で使われている生の XML 1.0 名を持ち、要素の属性を持つ `AttributesNS` インターフェイス

([AttributesNS インタフェース](#) を参照) のインスタンスを保持します。名前空間が要素に関連付けられていない場合、`name` の `uri` 要素は `None` です。`attrs` として渡されたオブジェクトはパーサに再利用されるかもしれません。そのため、それへの参照を確保するのは属性のコピーを保持する確実な方法ではありません。属性のコピーを保持するには `attrs` 属性の `copy()` メソッドを使用してください。

`feature.namespace_prefixes` 機能が有効でなければ、パーサで `qname` を `None` に設定することも可能です。

`ContentHandler.endElementNS(name, qname)`

名前空間モードでの要素の終了を通知します。

`name` 引数は `startElementNS()` イベントとまったく同じ要素型を持ちます。`qname` 引数も同様です。

`ContentHandler.characters(content)`

文字データの通知を受け取ります。

パーサはこのメソッドを呼び出して文字データの各チャンクを報告します。SAX パーサは一連の文字データを単一のチャンクとして返す場合と複数のチャンクに分けて返す場合がありますが、ロケータの情報が正しく保たれるように、一つのイベントの文字データは常に同じ外部エンティティのものでなければなりません。

`content` は Unicode 文字列、バイト列のどちらでもかまいませんが、`expat` リーダ・モジュールは常に Unicode 文字列を生成するようになっています。

注釈: Python XML SIG が提供していた初期 SAX 1 では、このメソッドにもっと JAVA 風のインターフェースが用いられています。しかし Python で採用されている大半のパーサでは古いインターフェースを有効に使うことができないため、よりシンプルなものに変更されました。古いコードを新しいインターフェースに変更するには、古い `offset` と `length` パラメータでスライスせずに、`content` を指定するようにしてください。

`ContentHandler.ignorableWhitespace(whitespace)`

要素内容中の無視できる空白文字の通知を受け取ります。

妥当性検査を行うパーサはこのメソッドを使って、無視できる空白文字 (W3C XML 1.0 勧告の 2.10 節参照) の各チャンクを報告しなければなりません。妥当性検査をしないパーサもコンテンツモデルの利用とパースが可能な場合、このメソッドを利用することが可能です。

SAX パーサは一連の空白文字を単一のチャンクとして返す場合と複数のチャンクに分けて返す場合がありますが、ロケータの情報が正しく保たれるように、一つのイベントの文字データは常に同じ外部エンティティのものでなければなりません。

`ContentHandler.processingInstruction(target, data)`

処理命令の通知を受け取ります。

パーサは処理命令が見つかるたびにこのメソッドを呼び出します。処理命令はメインのドキュメント要素の前や後にも発生することがあるので注意してください。

SAX パーサがこのメソッドを使って XML 宣言 (XML 1.0 のセクション 2.8) やテキスト宣言 (XML 1.0 のセクション 4.3.1) の通知をすることはありません。

`ContentHandler.skippedEntity(name)`

スキップしたエンティティの通知を受け取ります。

パーサはエンティティをスキップするたびにこのメソッドを呼び出します。妥当性検査をしないプロセッサは (外部 DTD サブセットで宣言されているなどの理由で) 宣言が見当たらないエンティティをスキップします。すべてのプロセッサは `feature_external_ges` および `feature_external_pes` 属性の値によっては外部エンティティをスキップすることがあります。

19.12.2 DTDHandler オブジェクト

`DTDHandler` インスタンスは以下のメソッドを提供します:

`DTDHandlernotationDecl(name, publicId, systemId)`

表記宣言イベントの通知を扱います。

`DTDHandlerunparsedEntityDecl(name, publicId, systemId, ndata)`

未パースのエンティティ宣言イベントの通知を扱います。

19.12.3 EntityResolver オブジェクト

`EntityResolver.resolveEntity(publicId, systemId)`

エンティティのシステム識別子を解決し、文字列として読み込んだシステム識別子あるいは `InputSource` オブジェクトのいずれかを返します。デフォルトの実装では `systemId` を返します。

19.12.4 ErrorHandler オブジェクト

このインターフェイスのあるオブジェクトを使って `XMLReader` からエラーと警告情報を受け取ります。このインターフェイスを実装しているオブジェクトを作った場合、それを `XMLReader` に登録して、パーサはオブジェクトのメソッドを呼んで全警告とエラーを報告します。エラーには利用可能な 3 つのレベルがあります: 警告、(おそらく) 回復可能なエラー、回復不能なエラーです。全てのメソッドは `SAXParseException` を引数としてのみ受け取ります。警告とエラーは渡された例外オブジェクトによって例外に変換されるかもしれません。

`ErrorHandler.error(exception)`

パーサが回復可能なエラーに遭遇すると呼び出されます。このメソッドが例外を送出しない場合パースは継続されますが、アプリケーションは更なるドキュメント情報を期待すべきではありません。パーサの処理を継続を認めることで入力ドキュメント内の他のエラーを見つけることができます。

`ErrorHandler.fatalError(exception)`

パーサが回復不能なエラーに遭遇すると呼び出されます。このメソッドが `return` したとき、パースの停止が求められています。

`ErrorHandler.warning` (*exception*)

パーサが軽微な警告情報をアプリケーションに通知するときに呼び出されます。このメソッドが `return` したときはパースの継続が求められ、ドキュメント情報はアプリケーションに送り返されます。このメソッドでの例外の送出はパースを終了します。

19.13 `xml.sax.saxutils` — SAX ユーティリティ

バージョン 2.0 で追加.

モジュール `xml.sax.saxutils` には SAX アプリケーションの作成に役立つ多くの関数やクラスも含まれており、直接利用したり、基底クラスとして使うことができます。

`xml.sax.saxutils.escape` (`data`[, `entities`])

文字列データ内の `'&'`, `'<'`, `'>'` をエスケープします。

オプションの `entities` パラメータに辞書を渡すことで、そのほかの文字をエスケープさせることも可能です。辞書のキーと値はすべて文字列で、キーに指定された文字は対応する値に置換されます。`'&'`, `'<'`, `'>'` は `entities` が与えられるかどうかに関わらず、常にエスケープします。

`xml.sax.saxutils.unescape` (`data`[, `entities`])

エスケープされた文字列 `'&'`, `'<'`, `'>'` を元の文字に戻します。

オプションの `entities` パラメータに辞書を渡すことで、そのほかの文字をエスケープさせることも可能です。辞書のキーと値はすべて文字列で、キーに指定された文字は対応する値に置換されます。`'&'`, `'<'`, `'>'` は `entities` が与えられるかどうかに関わらず、常に元の文字に戻します。

バージョン 2.3 で追加.

`xml.sax.saxutils.quoteattr` (`data`[, `entities`])

`escape()` に似ていますが、`data` は属性値の作成に使われます。戻り値はクオート済みの `data` で、置換する文字の追加も可能です。`quoteattr()` はクオートすべき文字を `data` の文脈から判断し、クオートすべき文字を残さないように文字列をエンコードします。`data` の中にシングル・クオート、ダブル・クオートがあれば、両方ともエンコードし、全体をダブルクオートで囲みます。戻り値の文字列はそのまま属性値として利用できます:

```
>>> print "<element attr=%s>" % quoteattr("ab ' cd \" ef")
<element attr="ab ' cd &quot; ef">
```

この関数は参照具象構文を使って、HTML や SGML の属性値を生成するのに便利です。

バージョン 2.2 で追加.

`class xml.sax.saxutils.XMLGenerator` ([`out`[, `encoding`]])

このクラスは `ContentHandler` インターフェースの実装で、SAX イベントを XML ドキュメントに書き戻します。つまり、`XMLGenerator` をコンテンツ・ハンドラとして用いると、パースしたオリジナル・ドキュメントの複製が作れるのです。`out` に指定するのはファイル風のオブジェクトで、デフォルトは `sys.stdout` です。`encoding` は出力ストリームのエンコーディングで、デフォルトは `'iso-8859-1'` です。

```
class xml.sax.saxutils.XMLFilterBase (base)
```

このクラスは *XMLReader* とクライアント・アプリケーションのイベント・ハンドラとの間に位置するものとして設計されています。デフォルトでは何もせず、ただリクエストをリーダに、イベントをハンドラに、それぞれ加工せず渡すだけです。しかし、サブクラスでメソッドをオーバーライドすると、イベント・ストリームやリクエストを加工してから渡すように変更可能です。

```
xml.sax.saxutils.prepare_input_source (source[, base])
```

この関数は引き数に入力ソース、オプションとして URL を取り、読み取り可能な解決済み *InputSource* オブジェクトを返します。入力ソースは文字列、ファイル風オブジェクト、*InputSource* のいずれでも良く、この関数を使うことで、パーサは様々な *source* パラメータを *parse()* に渡すことが可能になります。

19.14 xml.sax.xmlreader — XML パーサのインタフェース

バージョン 2.0 で追加.

各 SAX パーサは Python モジュールとして *XMLReader* インタフェースを実装しており、関数 *create_parser()* を提供しています。この関数は新たなパーサ・オブジェクトを生成する際、*xml.sax.make_parser()* から引数なしで呼び出されます。

```
class xml.sax.xmlreader.XMLReader
```

SAX パーサが継承可能な基底クラスです。

```
class xml.sax.xmlreader.IncrementalParser
```

入力ソースを一度にパースせずに、利用可能なように文書のチャンクを入力したい場合があります。SAX リーダは、通常はファイル全体を読み込みませんが、チャンクで読み込むことに注意してください。それでも *parse()* は文書全体が処理されるまで *return* しません。そのため、*parse()* のブロックする挙動を望まない場合はこれらのインターフェイスを使用してください。

パーサのインスタンスが作成されると、*feed* メソッドを通じてすぐに、データを受け入れられるようになります。*close* メソッドの呼出しでパースが終わると、パーサは新しいデータを受け入れられるように、*reset* メソッドを呼び出されなければなりません。

これらのメソッドをパース処理の途中で呼び出すことはできません。つまり、パースが実行された後で、パーサから *return* する前に呼び出す必要があるのです。

デフォルトでは、SAX 2.0 ドライバを書く人のために、このクラスは *IncrementalParser* の *feed*、*close*、*reset* メソッドを使って *XMLReader* インタフェースの *parse* メソッドを実装しています。

```
class xml.sax.xmlreader.Locator
```

SAX イベントと文書の位置を関連付けるインタフェースです。*locator* オブジェクトは *DocumentHandler* メソッドを呼び出している間だけ正しい結果を返し、それ以外とのときは、予測できない結果を返します。情報を利用できない場合、メソッドは *None* を返すこともあります。

```
class xml.sax.xmlreader.InputSource ([systemId])
```

XMLReader がエンティティを読み込むために必要な情報をカプセル化します。

このクラスには公開識別子、システム識別子、(場合によっては文字エンコーディング情報を含む) バイト・ストリーム、そしてエンティティの文字ストリームなどの情報が含まれます。

アプリケーションは `XMLReader.parse()` メソッドでの使用や `EntityResolver.resolveEntity` の戻り値としてこのオブジェクトを作成します。

`InputSource` はアプリケーションに属します。 `XMLReader` はアプリケーションから渡された `InputSource` オブジェクトの変更を許可されていませんが、コピーを作ってそれを変更することは可能です。

class `xml.sax.xmlreader.AttributesImpl(attrs)`

`Attributes` インタフェース ([Attributes インタフェース](#) 参照) の実装です。これは辞書風のオブジェクトで、`startElement()` 内で要素の属性を表示します。最も有用な辞書操作に加え、インタフェースに記述されているメソッドを多数サポートしています。このクラスのオブジェクトはリーダによってインスタンス化されなければなりません。 `attrs` は属性名と属性値の対応付けを含む辞書風オブジェクトでなければなりません。

class `xml.sax.xmlreader.AttributesNSImpl(attrs, qnames)`

`AttributesImpl` を名前空間認識型に改良したクラスで、 `startElementNS()` に渡されます。 `AttributesImpl` の派生クラスですが、 `namespaceURI` と `localname` の 2 要素のタプルを解釈します。さらに、元の文書に出てくる修飾名を返す多くのメソッドを提供します。このクラスは `AttributesNS` インタフェース ([AttributesNS インタフェース](#) 参照) の実装です。

19.14.1 XMLReader オブジェクト

`XMLReader` は次のメソッドをサポートします:

`XMLReader.parse(source)`

入力ソースを処理し、SAX イベントを作成します。 `source` オブジェクトはシステム識別子 (入力ソースを特定する文字列 – 一般にファイル名や URL)、ファイル様オブジェクト、または `InputSource` オブジェクトです。 `parse()` が return したとき、入力データの処理は完了し、パーサ・オブジェクトは破棄ないしリセットされます。制限として、現在の実装はバイト・ストリームのみをサポートしており、文字ストリームの処理は将来の課題になっています。

`XMLReader.getContentHandler()`

現在の `ContentHandler` を返します。

`XMLReader.setContentHandler(handler)`

現在の `ContentHandler` を設定します。 `ContentHandler` が設定されていない場合、内容イベントは破棄されます。

`XMLReader.getDTDHandler()`

現在の `DTDHandler` を返します。

`XMLReader.setDTDHandler(handler)`

現在の `DTDHandler` を返します。 `DTDHandler` が設定されていない場合、DTD イベントは破棄されます。

`XMLReader.getEntityResolver()`

現在の *EntityResolver* を返します。

`XMLReader.setEntityResolver(handler)`

現在の *EntityResolver* を返します。*EntityResolver* が設定されていない場合、外部エンティティの解決を試行することでエンティティのシステム識別子が開かれます。利用出来ない場合は失敗します。

`XMLReader.getErrorHandler()`

現在の *ErrorHandler* を返します。

`XMLReader.setErrorHandler(handler)`

現在のエラーハンドラを設定します。*ErrorHandler* が設定されていない場合、エラーが例外として送出され、警告が表示されます。

`XMLReader.setLocale(locale)`

アプリケーションにエラーや警告のロケール設定を許可します。

SAX パーサにとって、エラーや警告の地域化は必須ではありません。しかし、パーサが要求されたロケールをサポートしていない場合、SAX 例外を送出しなければなりません。アプリケーションはパースの途中でロケールの変更を要求することができます。

`XMLReader.getFeature(featurename)`

機能 *featurename* の現在の設定を返します。その機能が認識できないときは、*SAXNotRecognizedException* を送出します。有名な機能名はモジュール *xml.sax.handler* に列挙されています。

`XMLReader.setFeature(featurename, value)`

機能名 *featurename* に値 *value* を設定します。その機能が認識できないときは、*SAXNotRecognizedException* を送出します。また、パーサが指定された機能や設定をサポートしていないときは、*SAXNotSupportedException* を送出します。

`XMLReader.getProperty(propertyname)`

属性名 *propertyname* の現在の値を返します。その属性が認識できないときは、*SAXNotRecognizedException* を送出します。有名な属性名はモジュール *xml.sax.handler* に列挙されています。

`XMLReader.setProperty(propertyname, value)`

属性名 *propertyname* に値 *value* を設定します。その機能が認識できないときは、*SAXNotRecognizedException* を送出します。また、パーサが指定された機能や設定をサポートしていないときは、*SAXNotSupportedException* を送出します。

19.14.2 IncrementalParser オブジェクト

IncrementalParser のインスタンスは次の追加メソッドを提供します:

`IncrementalParser.feed(data)`

data のチャンクを処理します。

`IncrementalParser.close()`

文書の終端を決定します。文書の適格性を調べ (終端でのみ可能)、ハンドラを起動し、パース時に割り当てた資源を解放します。

`IncrementalParser.reset()`

このメソッドは `close` が呼び出された後、新しい文書をパースできるように、パーサをリセットするのに呼び出されます。`close` 後 `reset` を呼び出さずに `parse` や `feed` を呼び出した場合の戻り値は未定義です。

19.14.3 Locator オブジェクト

Locator のインスタンスは次のメソッドを提供します:

`Locator.getColumnNumber()`

現在のイベントが開始する列番号を返します。

`Locator.getLineNumber()`

現在のイベントが開始する行番号を返します。

`Locator.getPublicId()`

現在の文書イベントの公開識別子を返します。

`Locator.getSystemId()`

現在のイベントのシステム識別子を返します。

19.14.4 InputSource オブジェクト

`InputSource.setPublicId(id)`

この *InputSource* の公開識別子を設定します。

`InputSource.getPublicId()`

この *InputSource* の公開識別子を返します。

`InputSource.setSystemId(id)`

この *InputSource* のシステム識別子を設定します。

`InputSource.getSystemId()`

この *InputSource* のシステム識別子を返します。

`InputSource.setEncoding(encoding)`

この *InputSource* の文字エンコーディングを設定します。

エンコーディングは XML エンコーディング宣言として受け入れられる文字列でなければなりません (XML 勧告の 4.3.3 節を参照)。

InputSource も文字ストリームを含んでいた場合、*InputSource* のエンコーディング属性は無視されます。

`InputSource.getEncoding()`

この *InputSource* の文字エンコーディングを取得します。

`InputSource.setByteStream(bytefile)`

この入力ソースのバイトストリーム (Python のファイル様オブジェクトですが、バイト列と文字の相互変換はサポートしません) を設定します。

指定された文字ストリームもあった場合 SAX パーサはこれを無視しますが、URI 接続開始自身に優先してバイトストリームを使います。

アプリケーションがバイトストリームの文字エンコーディングを知っている場合は、`setEncoding` メソッドで設定する必要があります。

`InputSource.getBytesStream()`

この入力ソースのバイトストリームを取得します。

`getEncoding` メソッドは、このバイトストリームの文字エンコーディングを返します。不明なときは `None` を返します。

`InputSource.setCharacterStream(charfile)`

この入力ソースの文字ストリームをセットします (ストリームは Python 1.6 の Unicode-wrapped なファイル風オブジェクトで、Unicode 文字列への変換をサポートしていなければなりません)。

文字ストリームが指定された場合、SAX パーサは全バイトストリームを無視し、システム識別子への URI 接続の開始を試みません。

`InputSource.setCharacterStream()`

この入力ソースの文字ストリームを取得します。

19.14.5 Attributes インタフェース

Attributes オブジェクトは `copy()`, `get()`, `has_key()`, `items()`, `keys()`, `values()` を含むマッピングプロトコルの一部を実装しています。以下のメソッドも提供されています:

`Attributes.getLength()`

属性の数を返します。

`Attributes.getNames()`

属性の名前を返します。

`Attributes.getType(name)`

属性名 *name* のタイプを返します。通常は 'CDATA' です。

`Attributes.getValue(name)`

属性 *name* の値を返します。

19.14.6 `AttributesNS` インタフェース

このインタフェースは `Attributes` インタフェース ([Attributes インタフェース](#) 参照) のサブタイプです。 `Attributes` インタフェースがサポートしているすべてのメソッドは `AttributesNS` オブジェクトでも利用可能です。

次のメソッドもサポートされています:

`AttributesNS.getValueByQName(name)`

修飾名の値を返します。

`AttributesNS.getNameByQName(name)`

修飾名 `name` に対応する `(namespace, localname)` のペアを返します。

`AttributesNS.getQNameByName(name)`

`(namespace, localname)` のペアに対応する修飾名を返します。

`AttributesNS.getQNames()`

全ての属性の修飾名を返します。

19.15 `xml.parsers.expat` — Expat を使った高速な XML 解析

警告: `pyexpat` モジュールは悪意を持って構築されたデータに対して安全ではありません。信頼できないデータや認証されていないデータを解析する必要があるばあいは、[XML の脆弱性](#) を参照してください。

バージョン 2.0 で追加。

`xml.parsers.expat` モジュールは、検証 (validation) を行わない XML パーザ (parser, 解析器)、Expat への Python インタフェースです。モジュールは一つの拡張型 `xmlparser` を提供します。これは XML パーザの現在の状況を表します。一旦 `xmlparser` オブジェクトを生成すると、オブジェクトの様々な属性をハンドラ関数 (handler function) に設定できます。その後、XML 文書をパーザに入力すると、XML 文書の文字列とマークアップに応じてハンドラ関数が呼び出されます。

このモジュールでは、Expat パーザへのアクセスを提供するために `pyexpat` モジュールを使用します。`pyexpat` モジュールの直接使用は撤廃されています。

このモジュールは、例外を一つと型オブジェクトを一つ提供しています:

exception `xml.parsers.expat.ExpatError`

Expat がエラーを報告したときに例外を送出します。Expat のエラーを解釈する上での詳細な情報は、[ExpatError 例外](#) を参照してください。

exception `xml.parsers.expat.error`

[ExpatError](#) への別名です。

`xml.parsers.expat.XMLParserType`

`ParserCreate()` 関数から返された戻り値の型を示します。

`xml.parsers.expat` モジュールには以下の 2 つの関数が収められています:

`xml.parsers.expat.ErrorString(errno)`

与えられたエラー番号 `errno` を解説する文字列を返します。

`xml.parsers.expat.ParserCreate([encoding[, namespace_separator]])`

新しい `xmlparser` オブジェクトを作成し、返します。 `encoding` が指定されていた場合、XML データで使われている文字列のエンコード名でなければなりません。 Expat は、Python のように多くのエンコードをサポートしておらず、またエンコーディングのレパートリを拡張することはできません; サポートするエンコードは、UTF-8, UTF-16, ISO-8859-1 (Latin1), ASCII です。 `encoding`^{*1} が指定されると、文書に対する明示的、非明示的なエンコード指定を上書き (override) します。

Expat はオプションで XML 名前空間の処理を行うことができます。これは引数 `namespace_separator` に値を指定することで有効になります。この値は、1 文字の文字列でなければなりません; 文字列が誤った長さを持つ場合には `ValueError` が送出されます (None は値の省略と見なされます)。名前空間の処理が可能なとき、名前空間に属する要素と属性が展開されます。要素のハンドラである `StartElementHandler` と `EndElementHandler` に渡された要素名は、名前空間の URI、名前空間の区切り文字、要素名のローカル部を連結したものになります。名前空間の区切り文字が 0 バイト (`chr(0)`) の場合、名前空間の URI とローカル部は区切り文字なしで連結されます。

たとえば、`namespace_separator` に空白文字 (' ') がセットされ、次のような文書が解析されるとします:

```
<?xml version="1.0"?>
<root xmlns      = "http://default-namespace.org/"
      xmlns:py   = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

`StartElementHandler` は各要素ごとに次のような文字列を受け取ります:

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

`pyexpat` が使っている Expat ライブラリの制限により、返される `xmlparser` インスタンスは単独の XML ドキュメントの解析にしか使えません。それぞれのドキュメントごとに別々のパーサのインスタンスを作るために `ParserCreate` を呼び出してください。

参考:

[The Expat XML Parser](#) Expat プロジェクトのホームページ。

^{*1} XML 出力に含まれるエンコーディング文字列は適切な規格に従っていなければなりません。例えば、"UTF-8" は有効ですが、"UTF8" はそうではありません。 <https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl> と <https://www.iana.org/assignments/character-sets/character-sets.xhtml> を参照してください。

19.15.1 XMLParser Objects

`xmlparser` オブジェクトは以下のようなメソッドを持ちます:

`xmlparser.Parse(data[, isfinal])`

文字列 *data* の内容を解析し、解析されたデータを処理するための適切な関数を呼び出します。このメソッドを最後に呼び出す時は *isfinal* を真にしなければなりません; 単体ファイルを細切れに渡して解析出来ることを意味しますが、複数ファイルは扱えません。 *data* にはいつでも空の文字列を渡せます。

`xmlparser.ParseFile(file)`

file オブジェクトから読み込んだ XML データを解析します。 *file* には *read(nbytes)* メソッドのみが必要です。このメソッドはデータがなくなった場合に空文字列を返さねばなりません。

`xmlparser.SetBase(base)`

(XML) 宣言中のシステム識別子中の相対 URI を解決するための、基底 URI を設定します。相対識別子の解決はアプリケーションに任せられます: この値は関数 *ExternalEntityRefHandler()* や *NotationDeclHandler()*, *UnparsedEntityDeclHandler()* に引数 *base* としてそのまま渡されます。

`xmlparser.GetBase()`

以前の *SetBase()* によって設定された基底 URI を文字列の形で返します。 *SetBase()* が呼ばれていないときには *None* を返します。

`xmlparser.GetInputContext()`

現在のイベントを発生させた入力データを文字列として返します。データはテキストの入っているエンティティが持っているエンコードになります。イベントハンドラがアクティブでないときに呼ばれると、戻り値は *None* となります。

バージョン 2.1 で追加.

`xmlparser.ExternalEntityParserCreate(context[, encoding])`

親となるパーザで解析された内容が参照している、外部で解析されるエンティティを解析するために使える "子の" パーザを作成します。 *context* パラメータは、以下に記すように *ExternalEntityRefHandler()* ハンドラ関数に渡される文字列でなければなりません。子のパーザは *ordered_attributes*, *returns_unicode*, *specified_attributes* が現在のパーザの値に設定されて生成されます。

`xmlparser.SetParamEntityParsing(flag)`

パラメーターエンティティ (外部 DTD サブセットを含む) の解析を制御します。 *flag* の有効な値は、*XML_PARAM_ENTITY_PARSING_NEVER*, *XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE*, *XML_PARAM_ENTITY_PARSING_ALWAYS* です。 *flag* の設定をしたら *true* を返します。

`xmlparser.UseForeignDTD([flag])`

flag の値をデフォルトの *true* にすると、Expat は代替の DTD をロードするため、すべての引数に *None* を設定して *ExternalEntityRefHandler* を呼び出します。XML 文書が文書型定義を持っていないければ、*ExternalEntityRefHandler* が呼び出しますが、*StartDoctypeDeclHandler* と *EndDoctypeDeclHandler* は呼び出されません。

`flag` に `false` を与えると、メソッドが前回呼ばれた時の `true` の設定が解除されますが、他には何も起こりません。

このメソッドは `Parse()` または `ParseFile()` メソッドが呼び出される前にだけ呼び出されます; これら 2 つのメソッドのどちらかが呼び出されたあとにメソッドが呼ばれると、`code` に定数 `errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING` が設定されて例外 `ExpatError` が送出されます。

バージョン 2.3 で追加.

`xmlparser` オブジェクトは次のような属性を持ちます:

`xmlparser.buffer_size`

`buffer.text` が真の時に使われるバッファのサイズです。この属性に新しい整数値を代入することで違うバッファサイズにできます。サイズが変えられるときにバッファはフラッシュされます。

バージョン 2.3 で追加.

バージョン 2.6 で変更: バッファサイズが変えられるようになりました。

`xmlparser.buffer_text`

この値を真にすると、`xmlparser` オブジェクトが Expat から返されたもとの内容をバッファに保持するようになります。これにより可能なときに何度も `CharacterDataHandler()` を呼び出してしまふようなことを避けることができます。Expat は通常、文字列のデータを行末ごと大量に破棄するため、かなりパフォーマンスを改善できるはずですが、この属性はデフォルトでは偽で、いつでも変更可能です。

バージョン 2.3 で追加.

`xmlparser.buffer_used`

`buffer.text` が利用可能なとき、バッファに保持されたバイト数です。これらのバイトは UTF-8 でエンコードされたテキストを表します。この属性は `buffer.text` が偽の時には意味がありません。

バージョン 2.3 で追加.

`xmlparser.ordered_attributes`

この属性をゼロ以外の整数にすると、報告される (XML ノードの) 属性を辞書型ではなくリスト型にします。属性は文書のテキスト中の出現順で示されます。それぞれの属性は、2 つのリストのエントリ: 属性名とその値、が与えられます。(このモジュールの古いバージョンでも、同じフォーマットが使われています。) デフォルトでは、この属性はデフォルトでは偽となりますが、いつでも変更可能です。

バージョン 2.1 で追加.

`xmlparser.returns_unicode`

この属性をゼロ以外の整数にすると、ハンドラ関数に Unicode 文字列が渡されます。`returns_unicode` が `False` の時には、UTF-8 でエンコードされたデータを含む 8 ビット文字列がハンドラに渡されます。Python がユニコードサポートつきでビルドされている場合、この値はデフォルトで `True` です。

バージョン 1.6 で変更: 戻り値の型がいつでも変更できるように変更されたはずですが、

xmlparser.specified_attributes

ゼロ以外の整数にすると、パーザは文書のインスタンスで特定される属性だけを報告し、属性宣言から導出された属性は報告しないようになります。この属性が指定されたアプリケーションでは、XML プロセッサの振る舞いに関する標準に従うために必要とされる (文書型) 宣言によって、どのような付加情報が利用できるのかということについて特に注意を払わなければなりません。デフォルトで、この属性は偽となりますが、いつでも変更可能です。

バージョン 2.1 で追加.

以下の属性には、`xmlparser` オブジェクトで最も最近に起きたエラーに関する値が入っており、また `Parse()` または `ParseFile()` メソッドが `xml.parsers.expat.ExpatError` 例外を送出した際にのみ正しい値となります。

xmlparser.ErrorByteIndex

エラーが発生したバイトのインデックスです。

xmlparser.ErrorCode

エラーを特定する数値によるコードです。この値は `ErrorString()` に渡したり、`errors` オブジェクトで定義された内容と比較できます。

xmlparser.ErrorColumnNumber

エラーの発生したカラム番号です。

xmlparser.ErrorLineNumber

エラーの発生した行番号です。

以下の属性は `xmlparser` オブジェクトがその時パースしている位置に関する値を保持しています。コールバックがパースイベントを報告している間、これらの値はイベントの生成した文字列の先頭の位置を指し示します。コールバックの外から参照された時には、(対応するコールバックであるかにかかわらず) 直前のパースイベントの位置を示します。

バージョン 2.4 で追加.

xmlparser.CurrentByteIndex

パーサへの入力、現在のバイトインデックス。

xmlparser.CurrentColumnNumber

パーサへの入力、現在のカラム番号。

xmlparser.CurrentLineNumber

パーサへの入力、現在の行番号。

以下に指定可能なハンドラのリストを示します。`xmlparser` オブジェクト `o` にハンドラを指定するには、`o.handlername = func` を使用します。`handlername` は、以下のリストに挙げた値をとらなければならない、また `func` は正しい数の引数を受理する呼び出し可能なオブジェクトでなければなりません。引数は特に明記しない限り、すべて文字列となります。

xmlparser.XmlDeclHandler (*version, encoding, standalone*)

XML 宣言が解析された時に呼ばれます。XML 宣言とは、XML 勧告の適用バージョン (オプション)、文書テキストのエンコード、そしてオプションの "スタンドアロン" の宣言です。*version* と *encoding*

は `returns.unicode` 属性によって指示された型を示す文字列となり、`standalone` は、文書がスタンドアロンであると宣言される場合には 1 に、文書がスタンドアロンでない場合には 0 に、スタンドアロン宣言を省略する場合には -1 になります。このハンドラは Expat のバージョン 1.95.0 以降のみ使用できます。

バージョン 2.1 で追加。

`xmlparser.StartDoctypeDeclHandler (doctypeName, systemId, publicId, has_internal_subset)`

Expat が文書型宣言 `<!DOCTYPE ...`) を解析し始めたときに呼び出されます。 `doctypeName` は、与えられた値がそのまま Expat に提供されます。 `systemId` と `publicId` パラメタが指定されている場合、それぞれシステムと公開識別子を与えます。省略する時には `None` にします。文書が内部的な文書宣言のサブセット (internal document declaration subset) を持つか、サブセット自体の場合、 `has_internal_subset` は `true` になります。このハンドラには、Expat version 1.2 以上が必要です。

`xmlparser.EndDoctypeDeclHandler ()`

Expat が文書型宣言の解析を終えたときに呼び出されます。このハンドラには、Expat version 1.2 以上が必要です。

`xmlparser.ElementDeclHandler (name, model)`

それぞれの要素型宣言ごとに呼び出されます。 `name` は要素型の名前であり、 `model` は内容モデル (content model) の表現です。

`xmlparser.AttrlistDeclHandler (elname, attname, type, default, required)`

ひとつの要素型で宣言される属性ごとに呼び出されます。属性リストの宣言が 3 つの属性を宣言したとすると、このハンドラはひとつの属性に 1 度ずつ、3 度呼び出されます。 `elname` は要素名であり、これに対して宣言が適用され、 `attname` が宣言された属性名となります。属性型は文字列で、 `type` として渡されます; 取りえる値は、 `'CDATA'`, `'ID'`, `'IDREF'`, ... です。 `default` は、属性が文書のインスタンスによって指定されていないときに使用されるデフォルト値を与えます。デフォルト値 (`#IMPLIED` values) が存在しないときには `None` を与えます。文書のインスタンスによって属性値が与えられる必要のあるときには `required` が `true` になります。このメソッドは Expat version 1.95.0 以上が必要です。

`xmlparser.StartElementHandler (name, attributes)`

要素の開始を処理するごとに呼び出されます。 `name` は要素名を格納した文字列で、 `attributes` はその値に属性名を対応付ける辞書型です。

`xmlparser.EndElementHandler (name)`

要素の終端を処理するごとに呼び出されます。

`xmlparser.ProcessingInstructionHandler (target, data)`

処理命令を処理するごとに呼び出されます。

`xmlparser.CharacterDataHandler (data)`

文字データを処理するときに呼びだされます。このハンドラは通常の文字データ、 `CDATA` セクション、無視できる空白文字列のために呼び出されます。これらを識別しなければならないアプリケーションは、要求された情報を収集するために `StartCdataSectionHandler`, `EndCdataSectionHandler`, and `ElementDeclHandler` コールバックメソッドを使用できます。

`xmlparser.UnparsedEntityDeclHandler (entityName, base, systemId, publicId, notation-Name)`

解析されていない (NDATA) エンティティ宣言を処理するために呼び出されます。このハンドラは Expat ライブラリのバージョン 1.2 のためだけに存在します; より最近のバージョンでは、代わりに `EntityDeclHandler` を使用してください (根底にある Expat ライブラリ内の関数は、撤廃されたものであると宣言されています)。

`xmlparser.EntityDeclHandler` (*entityName*, *is_parameter_entity*, *value*, *base*, *systemId*, *publicId*,
notationName)

エンティティ宣言ごとに呼び出されます。パラメタと内部エンティティについて、*value* はエンティティ宣言の宣言済みの内容を与える文字列となります; 外部エンティティの時には `None` となります。解析済みエンティティの場合、*notationName* パラメタは `None` となり、解析されていないエンティティの時には記法 (notation) 名となります。*is_parameter_entity* は、エンティティがパラメタエンティティの場合真に、一般エンティティ (general entity) の場合には偽になります (ほとんどのアプリケーションでは、一般エンティティのことは気にする必要がありません)。このハンドラは Expat ライブラリのバージョン 1.95.0 以降でのみ使用できます。

バージョン 2.1 で追加。

`xmlparser.NotationDeclHandler` (*notationName*, *base*, *systemId*, *publicId*)

記法の宣言 (notation declaration) で呼び出されます。*notationName*, *base*, *systemId*, および *publicId* を与える場合、文字列にします。public な識別子が省略された場合、*publicId* は `None` になります。

`xmlparser.StartNamespaceDeclHandler` (*prefix*, *uri*)

要素が名前空間宣言を含んでいる場合に呼び出されます。名前空間宣言は、宣言が配置されている要素に対して `StartElementHandler` が呼び出される前に処理されます。

`xmlparser.EndNamespaceDeclHandler` (*prefix*)

名前空間宣言を含んでいたエレメントの終了タグに到達したときに呼び出されます。このハンドラは、要素に関する名前空間宣言ごとに、`StartNamespaceDeclHandler` とは逆の順番で一度だけ呼び出され、各名前空間宣言のスコープが開始されたことを示します。このハンドラは、要素が終了する際、対応する `EndElementHandler` が呼ばれた後に呼び出されます。

`xmlparser.CommentHandler` (*data*)

コメントで呼び出されます。*data* はコメントのテキストで、先頭の '`<!--`' と末尾の '`-->`' を除きます。

`xmlparser.StartCdataSectionHandler` ()

CDATA セクションの開始時に呼び出されます。CDATA セクションの構文的な開始と終了位置を識別できるようにするには、このハンドラと `EndCdataSectionHandler` が必要です。

`xmlparser.EndCdataSectionHandler` ()

CDATA セクションの終了時に呼び出されます。

`xmlparser.DefaultHandler` (*data*)

XML 文書中で、適用可能なハンドラが指定されていない文字すべてに対して呼び出されます。この文字とは、検出されたことが報告されるが、ハンドラは指定されていないようなコンストラクト (construct) の一部である文字を意味します。

`xmlparser.DefaultHandlerExpand` (*data*)

`DefaultHandler` () と同じですが、内部エンティティの展開を禁止しません。エンティティ参照は

デフォルトハンドラに渡されません。

`xmlparser.NotStandaloneHandler()`

XML 文書がスタンドアロンの文書として宣言されていない場合に呼び出されます。外部サブセットやパラメタエンティティへの参照が存在するが、XML 宣言が XML 宣言中で `standalone` 変数を `yes` に設定していない場合に起きます。このハンドラが 0 を返すと、パーザは `XML_ERROR_NOT_STANDALONE` を発生させます。このハンドラが設定されていなければ、パーザは前述の事態で例外を送出しません。

`xmlparser.ExternalEntityRefHandler(context, base, systemId, publicId)`

外部エンティティの参照時に呼び出されます。 `base` は現在の基底 (base) で、以前の `SetBase()` で設定された値になっています。 `public`、および `system` の識別子である、 `systemId` と `publicId` が指定されている場合、値は文字列です; `public` 識別子が指定されていない場合、 `publicId` は `None` になります。 `context` の値は不明瞭なものであり、以下に記述するようにしか使ってはなりません。

外部エンティティが解析されるようにするには、このハンドラを実装しなければなりません。このハンドラは、 `ExternalEntityParserCreate(context)` を使って適切なコールバックを指定し、子パーザを生成して、エンティティを解析する役割を担います。このハンドラは整数を返さなければなりません; 0 を返した場合、パーザは `XML_ERROR_EXTERNAL_ENTITY_HANDLING` エラーを送出します。そうでない場合、解析を継続します。

このハンドラが与えられておらず、 `DefaultHandler` コールバックが指定されていれば、外部エンティティは `DefaultHandler` で報告されます。

19.15.2 ExpatError 例外

`ExpatError` 例外はいくつかの興味深い属性を備えています:

`ExpatError.code`

特定のエラーにおける Expat の内部エラー番号です。この値はこのモジュールの `errors` オブジェクトで定義されている定数のいずれかに一致します。

バージョン 2.1 で追加.

`ExpatError.lineno`

エラーが検出された場所の行番号です。最初の行の番号は 1 です。

バージョン 2.1 で追加.

`ExpatError.offset`

エラーが発生した場所の行内でのオフセットです。最初のカラムの番号は 0 です。

バージョン 2.1 で追加.

19.15.3 例

以下のプログラムでは、与えられた引数を入力するだけの三つのハンドラを定義しています。

```
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print 'Start element:', name, attrs
def end_element(name):
    print 'End element:', name
def char_data(data):
    print 'Character data:', repr(data)

p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""<?xml version="1.0"?>
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""", 1)
```

このプログラムの出力は以下のようになります:

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

19.15.4 内容モデルの記述

内容モデルは入れ子になったタプルを使って記述されています。各タプルには以下の 4 つの値が収められています: 型、限定詞 (quantifier)、名前、そして子のタプル。子のタプルは単に内容モデルを記述したものです。

最初の二つのフィールドの値は `xml.parsers.expat` モジュールの `model` オブジェクトで定義されている定数です。これらの定数は二つのグループ: モデル型 (model type) グループと限定子 (quantifier) グループ、に取りまとめられます。

以下にモデル型グループにおける定数を示します:

`xml.parsers.expat.XML_CTYPE_ANY`

モデル名で指定された要素は ANY の内容モデルを持つと宣言されます。

`xml.parsers.expat.XML_CTYPE_CHOICE`

指定されたエレメントはいくつかのオプションから選択できるようになっています; (A | B | C) の

ような内容モデルで用いられます。

`xml.parsers.expat.XML_CTYPE_EMPTY`

EMPTY であると宣言されている要素はこのモデル型を持ちます。

`xml.parsers.expat.XML_CTYPE_MIXED`

`xml.parsers.expat.XML_CTYPE_NAME`

`xml.parsers.expat.XML_CTYPE_SEQ`

順々に続くようなモデルの系列を表すモデルがこのモデル型で表されます。(A, B, C) のようなモデルで用いられます。

限定子グループにおける定数を以下に示します:

`xml.parsers.expat.XML_CQUANT_NONE`

修飾子 (modifier) が指定されていません。従って A のように、厳密に一つだけです。

`xml.parsers.expat.XML_CQUANT_OPT`

このモデルはオプションです: A? のように、一つか全くないかです。

`xml.parsers.expat.XML_CQUANT_PLUS`

このモデルは (A+ のように) 一つかそれ以上あります。

`xml.parsers.expat.XML_CQUANT_REP`

このモデルは A* のようにゼロ回以上あります。

19.15.5 Expat エラー定数

以下の定数は `xml.parsers.expat` モジュールにおける `errors` オブジェクトで提供されています。これらの定数は、エラーが発生した際に送出される `ExpatError` 例外オブジェクトのいくつかの属性を解釈する上で便利です。

`errors` オブジェクトは以下の属性を持ちます:

`xml.parsers.expat.XML_ERROR_ASYNC_ENTITY`

`xml.parsers.expat.XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF`

属性値中のエンティティ参照が、内部エンティティではなく外部エンティティを参照しました。

`xml.parsers.expat.XML_ERROR_BAD_CHAR_REF`

文字参照が、XML では正しくない (illegal) 文字を参照しました (例えば 0 や '�')。

`xml.parsers.expat.XML_ERROR_BINARY_ENTITY_REF`

エンティティ参照が、記法 (notation) つきで宣言されているエンティティを参照したため、解析できません。

`xml.parsers.expat.XML_ERROR_DUPLICATE_ATTRIBUTE`

一つの属性が一つの開始タグ内に一度より多く使われています。

`xml.parsers.expat.XML_ERROR_INCORRECT_ENCODING`

`xml.parsers.expat.XML_ERROR_INVALID_TOKEN`

入力されたバイトが文字に適切に関連付けできない際に送出されます; 例えば、UTF-8 入力ストリームにおける NUL バイト (値 0) などです。

`xml.parsers.expat.XML_ERROR_JUNK_AFTER_DOC_ELEMENT`

空白以外の何かがドキュメント要素の後にあります。

`xml.parsers.expat.XML_ERROR_MISPLACED_XML_PI`

入力データの先頭以外の場所に XML 定義が見つかりました。

`xml.parsers.expat.XML_ERROR_NO_ELEMENTS`

このドキュメントには要素が入っていません (XML では全てのドキュメントは確実にトップレベルの要素を一つ持つよう要求しています)。

`xml.parsers.expat.XML_ERROR_NO_MEMORY`

Expat が内部メモリを確保できませんでした。

`xml.parsers.expat.XML_ERROR_PARAM_ENTITY_REF`

パラメタエンティティが許可されていない場所で見つかりました。

`xml.parsers.expat.XML_ERROR_PARTIAL_CHAR`

入力に不完全な文字が見つかりました。

`xml.parsers.expat.XML_ERROR_RECURSIVE_ENTITY_REF`

エンティティ参照中に、同じエンティティへの別の参照が入っていました; おそらく違う名前で参照しているか、間接的に参照しています。

`xml.parsers.expat.XML_ERROR_SYNTAX`

何らかの仕様化されていない構文エラーに遭遇しました。

`xml.parsers.expat.XML_ERROR_TAG_MISMATCH`

終了タグが最も内側で開かれている開始タグに一致しません。

`xml.parsers.expat.XML_ERROR_UNCLOSED_TOKEN`

何らかの (開始タグのような) トークンが閉じられないまま、ストリームの終端や次のトークンに遭遇しました。

`xml.parsers.expat.XML_ERROR_UNDEFINED_ENTITY`

定義されていないエンティティへの参照が行われました。

`xml.parsers.expat.XML_ERROR_UNKNOWN_ENCODING`

ドキュメントのエンコードが Expat でサポートされていません。

`xml.parsers.expat.XML_ERROR_UNCLOSED_CDATA_SECTION`

CDATA セクションが閉じられていません。

`xml.parsers.expat.XML_ERROR_EXTERNAL_ENTITY_HANDLING`

`xml.parsers.expat.XML_ERROR_NOT_STANDALONE`

XML 文書が "standalone" だと宣言されており `NotStandaloneHandler` が設定され 0 が返されているにもかかわらず、パーサは "standalone" ではないと判別しました。

`xml.parsers.expat.XML_ERROR_UNEXPECTED_STATE`

`xml.parsers.expat.XML_ERROR_ENTITY_DECLARED_IN_PE`

`xml.parsers.expat.XML_ERROR_FEATURE_REQUIRES_XML_DTD`

その操作を完了するには DTD のサポートが必要ですが、Expat が DTD のサポートをしない設定になっています。これは `xml.parsers.expat` モジュールの標準的なビルドでは報告されません。

`xml.parsers.expat.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING`

パースが始まったあとで動作の変更が要求されました。これはパースが開始される前にのみ変更可能です。(現在のところ) `UseForeignDTD()` によってのみ送出されます。

`xml.parsers.expat.XML_ERROR_UNBOUND_PREFIX`

名前空間の処理を有効すると宣言されていないプレフィックスが見つかります。

`xml.parsers.expat.XML_ERROR_UNDECLARING_PREFIX`

XML 文書はプレフィックスに対応した名前空間宣言を削除しようとしてしました。

`xml.parsers.expat.XML_ERROR_INCOMPLETE_PE`

パラメータエンティティは不完全なマークアップを含んでいます。

`xml.parsers.expat.XML_ERROR_XML_DECL`

XML 文書中に要素がありません。

`xml.parsers.expat.XML_ERROR_TEXT_DECL`

外部エンティティ中のテキスト宣言にエラーがあります。

`xml.parsers.expat.XML_ERROR_PUBLICID`

パブリック ID 中に許可されていない文字があります。

`xml.parsers.expat.XML_ERROR_SUSPENDED`

要求された操作は一時停止されたパーサで行われていますが、許可されていない操作です。このエラーは追加の入力を行なおうとしている場合、もしくはパーサが停止しようとしている場合にも送出されます。

`xml.parsers.expat.XML_ERROR_NOT_SUSPENDED`

パーサを一時停止しようとしてしましたが、停止されませんでした。

`xml.parsers.expat.XML_ERROR_ABORTED`

Python アプリケーションには通知されません。

`xml.parsers.expat.XML_ERROR_FINISHED`

要求された操作で、パース対象となる入力完了したと判断しましたが、入力は受理されませんでした。このエラーは追加の入力を行なおうとしている場合、もしくはパーサが停止しようとしている場合に送出されます。

`xml.parsers.expat.XML_ERROR_SUSPEND_PE`

第 20 章

インターネットプロトコルとサポート

この章で記述されるモジュールは、インターネットプロトコルを実装し、関連技術をサポートします。それらは全て Python で実装されています。これらのモジュールの大部分は、システム依存のモジュール `socket` が存在することが必要ですが、これは現在ではほとんどの一般的なプラットフォーム上でサポートされています。ここに概観を示します：

20.1 `webbrowser` — 便利なウェブブラウザコントローラー

ソースコード: [Lib/webbrowser.py](#)

`webbrowser` モジュールにはウェブベースのドキュメントを表示するための、とてもハイレベルなインターフェースが定義されています。たいいていの環境では、このモジュールの `open()` を呼び出すだけで正しく動作します。

Unix では、X11 上でグラフィカルなブラウザが選択されますが、グラフィカルなブラウザが利用できなかったり、X11 が利用できない場合はテキストモードのブラウザが使われます。もしテキストモードのブラウザが使われたら、ユーザがブラウザから抜け出すまでプロセスの呼び出しはブロックされます。

環境変数 `BROWSER` が存在する場合、プラットフォームのデフォルトであるブラウザのリストをオーバーライドし、`os.pathsep` で区切られたリストの順にブラウザの起動を試みます。リストの中の値に `%s` が含まれていたら、テキストモードのブラウザのコマンドラインとして `%s` の代わりに URL が引数として解釈されます；もし `%s` が含まれなければ、起動するブラウザの名前として単純に解釈されます。^{*1}

非 Unix プラットフォームあるいは Unix 上でリモートブラウザが利用可能な場合、制御プロセスはユーザがブラウザを終了するのを待ちませんが、ディスプレイにブラウザのウィンドウを表示させたままにします。Unix 上でリモートブラウザが利用可能でない場合、制御プロセスは新しいブラウザを立ち上げ、待ちます。

`webbrowser` スクリプトをこのモジュールのコマンドライン・インタフェースとして使うことができます。スクリプトは引数に 1 つの URL を受け付けます。また次のオプション引数を受け付けます。 `-n` により可能ならば新しいブラウザウィンドウで指定された URL を開きます。一方、 `-t` では新しいブラウザのページ（「タブ」）で開きます。当然ながらこれらのオプションは排他的です。使用例は次の通りです：

^{*1} ここでブラウザの名前が絶対パスで書かれていない場合は `PATH` 環境変数で与えられたディレクトリから探し出されます。

```
python -m webbrowser -t "http://www.python.org"
```

以下の例外が定義されています:

exception `webbrowser.Error`

ブラウザのコントロールエラーが起こると発生する例外。

以下の関数が定義されています:

`webbrowser.open(url, new=0, autoraise=True)`

デフォルトのブラウザで `url` を表示します。`new` が 0 なら、`url` はブラウザの今までと同じウィンドウで開きます。`new` が 1 なら、可能であればブラウザの新しいウィンドウが開きます。`new` が 2 なら、可能であればブラウザの新しいタブが開きます。`autoraise` が `True` なら、可能であればウィンドウが前面に表示されます (多くのウィンドウマネージャではこの変数の設定に関わらず、前面に表示されます)。

幾つかのプラットフォームにおいて、ファイル名をこの関数で開こうとすると、OS によって関連付けられたプログラムが起動されます。しかし、この動作はポータブルではありませんし、サポートされていません。

バージョン 2.5 で変更: `new` を 2 にもできるようになりました。

`webbrowser.open_new(url)`

可能であれば、デフォルトブラウザの新しいウィンドウで `url` を開きますが、そうでない場合はブラウザのただ 1 つのウィンドウで `url` を開きます。

`webbrowser.open_new_tab(url)`

可能であれば、デフォルトブラウザの新しいページ (「タブ」) で `url` を開きますが、そうでない場合は `open_new()` と同様に振る舞います。

バージョン 2.5 で追加。

`webbrowser.get([name])`

ブラウザの種類 `name` のコントローラオブジェクトを返します。もし `name` が空文字列 なら、呼び出した環境に適したデフォルトブラウザのコントローラを返します。

`webbrowser.register(name, constructor[, instance])`

ブラウザの種類 `name` を登録します。ブラウザの種類が登録されたら、`get()` でそのブラウザのコントローラを呼び出すことができます。`instance` が指定されなかったり、`None` なら、インスタンスが必要な時には `constructor` がパラメータなしに呼び出されて作られます。`instance` が指定されたら、`constructor` は呼び出されないの、`None` でかまいません。

この登録は、変数 `BROWSER` を設定するか、`get()` を空文字列でなく、宣言したハンドラの名前と一致する引数とともに呼び出すときだけ、役に立ちます。

いくつかの種類のブラウザがあらかじめ定義されています。このモジュールで定義されている、関数 `get()` に与えるブラウザの名前と、それぞれのコントローラクラスのインスタンスを以下の表に示します。

Type Name	Class Name	注釈
'mozilla'	Mozilla('mozilla')	
'firefox'	Mozilla('mozilla')	
'netscape'	Mozilla('netscape')	
'galeon'	Galeon('galeon')	
'epiphany'	Galeon('epiphany')	
'skipstone'	BackgroundBrowser('skipstone')	
'kfmclient'	Konqueror()	(1)
'konqueror'	Konqueror()	(1)
'kfm'	Konqueror()	(1)
'mosaic'	BackgroundBrowser('mosaic')	
'opera'	Opera()	
'grail'	Grail()	
'links'	GenericBrowser('links')	
'elinks'	Elinks('elinks')	
'lynx'	GenericBrowser('lynx')	
'w3m'	GenericBrowser('w3m')	
'windows-default'	WindowsDefault	(2)
'macosx'	MacOSX('default')	(3)
'safari'	MacOSX('safari')	(3)
'google-chrome'	Chrome('google-chrome')	(4)
'chrome'	Chrome('chrome')	(4)
'chromium'	Chromium('chromium')	(4)
'chromium-browser'	Chromium('chromium-browser')	(4)

注釈:

- (1) "Konqueror"は Unix の KDE デスクトップ環境のファイルマネージャで、KDE が動作している時にだけ意味を持ちます。何か信頼できる方法で KDE を検出するのがいいでしょう; 変数 `KDEDIR` では十分ではありません。また、KDE 2 で `konqueror` コマンドを使うときにも、"kfm"が使われます — Konqueror を動作させるのに最も良い方法が実装によって選択されます。
- (2) Windows プラットフォームのみ。
- (3) Mac OS X プラットフォームのみ。
- (4) Chrome/Chromium のサポートがバージョン 2.7.5 で追加されました。

簡単な例を示します:

```
url = 'http://www.python.org/'

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url + 'doc/')
```

(次のページに続く)

(前のページからの続き)

```
# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```

20.1.1 ブラウザコントローラーオブジェクト

ブラウザコントローラーには以下のメソッドが定義されていて、モジュールレベルの便利な 3 つの関数に相当します:

`controller.open(url, new=0, autoraise=True)`

このコントローラーでハンドルされたブラウザで `url` を表示します。`new` が 1 なら、可能であればブラウザの新しいウィンドウが開きます。`new` が 2 なら、可能であればブラウザの新しいページ(「タブ」)が開きます。

`controller.open_new(url)`

可能であれば、このコントローラーでハンドルされたブラウザの新しいウィンドウで `url` を開きますが、そうでない場合はブラウザのただ 1 つのウィンドウで `url` を開きます。`open_new()` の別名。

`controller.open_new_tab(url)`

可能であれば、このコントローラーでハンドルされたブラウザの新しいページ(「タブ」)で `url` を開きますが、そうでない場合は `open_new()` と同じです。

バージョン 2.5 で追加。

20.2 cgi — CGI (ゲートウェイインタフェース規格) のサポート

ソースコード: `Lib/cgi.py`

ゲートウェイインタフェース規格 (CGI) に準拠したスクリプトをサポートするためのモジュールです。

このモジュールでは、Python で CGI スクリプトを書く際に使える様々なユーティリティを定義しています。

20.2.1 はじめに

CGI スクリプトは、HTTP サーバによって起動され、通常は HTML の `<FORM>` または `<ISINDEX>` エレメントを通じてユーザが入力した内容を処理します。

ほとんどの場合、CGI スクリプトはサーバ上の特殊なディレクトリ `cgi-bin` の下に置きます。HTTP サーバは、まずスクリプトを駆動するためのシェルの環境変数に、リクエストの全ての情報(クライアントのホスト名、リクエストされている URL、クエリ文字列、その他諸々)を設定し、スクリプトを実行した後、スクリプトの出力をクライアントに送信します。

スクリプトの入力端もクライアントに接続されていて、この経路を通じてフォームデータを読み込むこともあります。それ以外の場合には、フォームデータは URL の一部分である「クエリ文字列」を介して渡されます。

このモジュールでは、上記のケースの違いに注意しつつ、Python スクリプトに対しては単純なインタフェースを提供しています。このモジュールではまた、スクリプトをデバッグするためのユーティリティも多数提供しています。また、最近ではフォームを経由したファイルのアップロードをサポートしています (ブラウザ側がサポートしていればです)。

CGI スクリプトの出力は 2 つのセクションからなり、空行で分割されています。最初のセクションは複数のヘッダからなり、後続するデータがどのようなものをクライアントに通知します。最小のヘッダセクションを生成するための Python のコードは以下のようなものです:

```
print "Content-Type: text/html"      # HTML is following
print                               # blank line, end of headers
```

二つ目のセクションは通常、ヘッダやインラインイメージ等の付属したテキストをうまくフォーマットして表示できるようにした HTML です。以下に単純な HTML を出力する Python コードを示します:

```
print "<TITLE>CGI script output</TITLE>"
print "<H1>This is my first CGI script</H1>"
print "Hello, world!"
```

20.2.2 cgi モジュールを使う

先頭には `import cgi` と書いてください。 `from cgi import *` と書いてはなりません — このモジュールでは、以前のバージョンとの互換性を持たせるため、内部で呼び出す名前を多数定義しており、それらをユーザの名前空間に存在させる必要はないからです。

新たにスクリプトを書く際には、以下の行を付加するかどうか検討してください:

```
import cgitb
cgitb.enable()
```

これによって、特別な例外処理が有効にされ、エラーが発生した際にブラウザ上に詳細なレポートを出力するようになります。ユーザにスクリプトの内部を見せたくないのなら、以下のようにしてレポートをファイルに保存できます:

```
import cgitb
cgitb.enable(display=0, logdir="/path/to/logdir")
```

スクリプトを開発する際には、この機能はとても役に立ちます。 `cgitb` が生成する報告はバグを追跡するためにかかる時間を大幅に減らせるような情報を提供してくれます。スクリプトをテストし終わり、正確に動作することを確認したら、いつでも `cgitb` の行を削除できます。

入力されたフォームデータを取得するには、 `FieldStorage` クラスを使うのが最良の方法です。このモジュールで定義されている他のクラスのほとんどは以前のバージョンとの互換性のためのものです。インスタンス生成は引数なしで必ず 1 度だけ行います。これにより、標準入力または環境変数からフォームの内容を読み出します (どちらから読み出すかは、複数の環境変数の値が CGI 標準に従ってどう設定されているかで決まります)。インスタンスが標準入力を消費しうるので、インスタンス生成を行うのは一度だけにしなければなりません。

FieldStorage のインスタンスは Python の辞書型のように添え字アクセスが可能です。in を使用することによって要素が含まれているかの判定も出来ますし、標準の辞書メソッド `keys()` 及び組み込み関数 `len()` もサポートしています。空の文字列を含むフォーム要素は無視され、辞書には現れません。そのような値を保持するには、FieldStorage のインスタンス作成の際にオプションのキーワードパラメータ `keep_blank_values` に true を指定してください。

例えば、以下のコード (*Content-Type* ヘッダと空行はすでに出力された後とします) は name および addr フィールドが両方とも空の文字列に設定されていないか調べます:

```
form = cgi.FieldStorage()
if "name" not in form or "addr" not in form:
    print "<H1>Error</H1>"
    print "Please fill in the name and addr fields."
    return
print "<p>name:", form["name"].value
print "<p>addr:", form["addr"].value
...further form processing here...
```

ここで、`form[key]` で参照される各フィールドはそれ自体が FieldStorage (または MiniFieldStorage。フォームのエンコードによって変わります) のインスタンスです。インスタンスの属性 `value` の内容は対応するフィールドの値で、文字列になります。`getvalue()` メソッドはこの文字列値を直接返します。`getvalue()` の 2 つめの引数にオプションの値を与えると、リクエストされたキーが存在しない場合に返すデフォルトの値になります。

入力されたフォームデータに同じ名前のフィールドが二つ以上あれば、`form[key]` で得られるオブジェクトは FieldStorage や MiniFieldStorage のインスタンスではなく、そうしたインスタンスのリストになります。この場合、`form.getvalue(key)` も同様に、文字列からなるリストを返します。もしこうした状況が起きうと思うなら (HTML のフォームに同じ名前をもったフィールドが複数含まれているのなら)、`getlist()` メソッドを使ってください。これは常に値のリストを返します (単一要素のケースを特別扱いする必要はありません)。例えば、以下のコードは任意の数のユーザ名フィールドを結合し、コンマで分割された文字列にします:

```
value = form.getlist("username")
usernames = ",".join(value)
```

フィールドがアップロードされたファイルを表している場合、`value` 属性や `getvalue()` メソッドを使ってフィールドの値にアクセスすると、ファイルの内容を全て文字列としてメモリ上に読み込んでしまいます。これは望ましくない機能かもしれません。アップロードされたファイルがあるかどうかは `filename` 属性および `file` 属性のいずれかで調べられます。その後、以下のようにして `file` 属性から落착いてデータを読み出せます:

```
fileitem = form["userfile"]
if fileitem.file:
    # It's an uploaded file; count lines
    linecount = 0
    while 1:
        line = fileitem.file.readline()
        if not line: break
```

(次のページに続く)

(前のページからの続き)

```
linecount = linecount + 1
```

アップロードされたファイルの内容を取得している間にエラーが発生した場合 (例えば、ユーザーが戻るボタンやキャンセルボタンで submit を中断した場合)、そのフィールドのオブジェクトの `done` 属性には `-1` が設定されます。

現在ドラフトとなっているファイルアップロードの標準仕様では、一つのフィールドから (再帰的な `multipart/*` エンコーディングを使って) 複数のファイルがアップロードされる可能性を受け入れています。この場合、アイテムは辞書形式の `FieldStorage` アイテムとなります。複数ファイルかどうかは `type` 属性が `multipart/form-data` (または `multipart/*` にマッチする他の MIME 型) になっているかどうかを調べれば判別できます。この場合、トップレベルのフォームオブジェクトと同様にして再帰的に個別処理できます。

フォームが「古い」形式で入力された場合 (クエリ文字列または単一の `application/x-www-form-urlencoded` データで入力された場合)、データ要素の実体は `MiniFieldStorage` クラスのインスタンスになります。この場合、`list`、`file`、および `filename` 属性は常に `None` になります。

フォームが POST によって送信され、クエリー文字列も持っていた場合、`FieldStorage` と `MiniFieldStorage` の両方が含まれます。

20.2.3 高水準インタフェース

バージョン 2.2 で追加.

前節では CGI フォームデータを `FieldStorage` クラスを使って読み出す方法について解説しました。この節では、フォームデータを分かりやすく直感的な方法で読み出せるようにするために追加された、より高水準のインタフェースについて記述します。このインタフェースは前節で説明した技術を撤廃するものではありません — 例えば、前節の技術は依然としてファイルのアップロードを効率的に行う上で便利です。

このインタフェースは 2 つの単純なメソッドからなります。このメソッドを使えば、一般的な方法でフォームデータを処理でき、ある名前のフィールドに入力された値が一つなのかそれ以上なのかを心配する必要がなくなります。

前節では、一つのフィールド名に対して二つ以上の値が入力されるかもしれない場合には、常に以下のようなコードを書くよう学びました:

```
item = form.getvalue("item")
if isinstance(item, list):
    # The user is requesting more than one item.
else:
    # The user is requesting only one item.
```

こういった状況は、例えば以下のように、同じ名前を持った複数のチェックボックスからなるグループがフォームに入っているような場合によく起きます:


```
<input type="checkbox" name="item" value="1" />
<input type="checkbox" name="item" value="2" />
```

しかしながら、ほとんどの場合、あるフォーム中で特定の名前を持ったコントロールはただ一つしかないの
で、その名前に関連付けられた値はただ一つしかないはずだと考えるでしょう。そこで、スクリプトには例え
ば以下のようなコードを書くでしょう:

```
user = form.getvalue("user").upper()
```

このコードの問題点は、クライアント側がスクリプトにとって常に有効な入力を提供するとは期待できない
ところにあります。例えば、もし好奇心旺盛なユーザがもう一つの `user=foo` ペアをクエリ文字列に追加し
たら、`getvalue("user")` メソッドは文字列ではなくリストを返すため、このスクリプトはクラッシュす
るでしょう。リストに対して `upper()` メソッドを呼び出すと、引数が有効でない(リスト型はその名前のメ
ソッドを持っていない)ため、例外 `AttributeError` を送出します。

従って、フォームデータの値を読み出しには、得られた値が単一の値なのか値のリストなのかを常に調べる
コードを使うのが適切でした。これでは煩わしく、より読みにくいスクリプトになってしまいます。

ここで述べる高水準のインタフェースで提供している `getfirst()` や `getlist()` メソッドを使うと、もっ
と便利にアプローチできます。

`FieldStorage.getfirst(name[, default])`

フォームフィールド `name` に関連付けられた値をつねに一つだけ返す軽量メソッドです。同じ名前で 1
つ以上の値がポストされている場合、このメソッドは最初の値だけを返します。フォームから値を受
信する際の値の並び順はブラウザ間で異なる可能性があり、特定の順番であるとは期待できないので
注意してください。^{*1} 指定したフォームフィールドや値がない場合、このメソッドはオプションの引数
`default` を返します。このパラメタを指定しない場合、標準の値は `None` に設定されます。

`FieldStorage.getlist(name)`

このメソッドはフォームフィールド `name` に関連付けられた値を常にリストにして返します。`name` に
指定したフォームフィールドや値が存在しない場合、このメソッドは空のリストを返します。値が一つ
だけ存在する場合、要素を一つだけ含むリストを返します。

これらのメソッドを使うことで、以下のようにナイスでコンパクトにコードを書けます:

```
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", "").upper()      # This way it's safe.
for item in form.getlist("item"):
    do_something(item)
```

^{*1} 最近のバージョンの HTML 仕様ではフィールドの値を供給する順番を取り決めてはいますが、ある HTTP リクエストがその取り
決めに準拠したブラウザから受信したものかどうか、そもそもブラウザから送信されたものかどうかの判別は退屈で間違いやすい
ので注意してください。

20.2.4 古いクラス群

バージョン 2.6 で非推奨: これらのクラスは、`cgi` モジュールの以前のバージョンに入っており、以前のバージョンとの互換性のために現在もサポートされています。新しいアプリケーションでは `FieldStorage` クラスを使うべきです。

`SvFormContentDict` は単一の値しか持たないフォームデータの内容を辞書として記憶します; このクラスでは、各フィールド名はフォーム中に一度しか現れないと仮定しています。

`FormContentDict` は複数の値を持つフォームデータの内容を辞書として記憶します (フォーム要素は値のリストです); フォームが同じ名前を持ったフィールドを複数含む場合に便利です。

他のクラス (`FormContent`、`InterpFormContentDict`) は非常に古いアプリケーションとの後方互換性のためだけにあります。

20.2.5 関数

より細かく CGI をコントロールしたり、このモジュールで実装されているアルゴリズムを他の状況で利用したい場合には、以下の関数が便利です。

`cgi.parse(fp[, environ[, keep_blank_values[, strict_parsing]]])`

環境変数、またはファイルからクエリを解釈します (ファイルは標準で `sys.stdin`、環境変数は標準で `os.environ` になります) `keep_blank_values` および `strict_parsing` パラメタはそのまま `urlparse.parse_qs()` に渡されます。

`cgi.parse_qs(qs[, keep_blank_values[, strict_parsing[, max_num_fields]]])`

この関数はこのモジュールでは廃止予定です。代わりに `urlparse.parse_qs()` を利用してください。この関数は後方互換性のためだけに残されています。

`cgi.parse_qs1(qs[, keep_blank_values[, strict_parsing[, max_num_fields]]])`

この関数はこのモジュールでは廃止予定です。代わりに `urlparse.parse_qs1()` を利用してください。この関数は後方互換性のためだけに残されています。

`cgi.parse_multipart(fp, pdict)`

(ファイル入力のための) `multipart/form-data` 型の入力を解釈します。引数は入力ファイルを示す `fp` と `Content-Type` ヘッダ内の他のパラメタを含む辞書 `pdict` です。

`urlparse.parse_qs()` と同じく辞書を返します。辞書のキーはフィールド名で、対応する値は各フィールドの値でできたリストです。この関数は簡単に使えますが、数メガバイトのデータがアップロードされると考えられる場合にはあまり適していません — その場合、より柔軟性のある `FieldStorage` を代りに使ってください。

マルチパートデータがネストしている場合、各パートを解釈できないので注意してください — 代りに `FieldStorage` を使ってください。

`cgi.parse_header(string)`

(`Content-Type` のような) MIME ヘッダを解釈し、ヘッダの主要値と各パラメタからなる辞書にします。

`cgi.test()`

メインプログラムから利用できる堅牢性テストを行う CGI スクリプトです。最小の HTTP ヘッダと、HTML フォームからスクリプトに供給された全ての情報を書式化して出力します。

`cgi.print_environ()`

シェル変数を HTML に書式化して出力します。

`cgi.print_form(form)`

フォームを HTML に初期化して出力します。

`cgi.print_directory()`

現在のディレクトリを HTML に書式化して出力します。

`cgi.print_environ_usage()`

意味のある (CGI の使う) 環境変数を HTML で出力します。

`cgi.escape(s[, quote])`

文字列 *s* 中の文字 '&', '<', および '>' を HTML で正しく表示できる文字列に変換します。それらの文字が中に入っているかもしれないようなテキストを出力する必要があるときに使ってください。オプションの引数 *quote* の値が真であれば、二重引用符文字 (") も変換します; この機能は `` のように二重引用符で区切られた HTML の属性値を出力に含めるのに役立ちます。単引用符は変換されないことに注意して下さい。

クオートされる値が単引用符か二重引用符、またはその両方を含む可能性がある場合は、代わりに `xml.sax.saxutils.quoteattr()` 関数を検討してください。

20.2.6 セキュリティへの配慮

重要なルールが一つあります: (関数 `os.system()` または `os.popen()`、またはその他の同様の機能によって) 外部プログラムを呼び出すなら、クライアントから受信した任意の文字列をシェルに渡していないことをよく確かめてください。これはよく知られているセキュリティホールであり、これによって Web のどこかにいる悪賢いハッカーが、だまされやすい CGI スクリプトに任意のシェルコマンドを実行させてしまいます。URL の一部やフィールド名でさえも信用してはいけません。CGI へのリクエストはあなたの作ったフォームから送信されるとは限らないからです!

安全な方法をとるために、フォームから入力された文字をシェルに渡す場合、文字列に入っているのが英数文字、ダッシュ、アンダースコア、およびピリオドだけかどうかを確認してください。

20.2.7 CGI スクリプトを Unix システムにインストールする

あなたの使っている HTTP サーバのドキュメントを読んでください。そしてローカルシステムの管理者と一緒にどのディレクトリに CGI スクリプトをインストールすべきかを調べてください; 通常これはサーバのファイルシステムツリー内の `cgi-bin` ディレクトリです。

あなたのスクリプトが "others" によって読み取り可能および実行可能であることを確認してください; Unix ファイルモードは 8 進表記で 0755 です (`chmod 0755 filename` を使ってください)。スクリプトの最初

の行の 1 カラム目が、`#!` で開始し、その後に Python インタプリタへのパス名が続いていることを確認してください。例えば:

```
#!/usr/local/bin/python
```

Python インタプリタが存在し、“others” によって実行可能であることを確かめてください。

あなたのスクリプトが読み書きしなければならないファイルが全て “others” によって読み出しや書き込み可能であることを確かめてください — 読み出し可能のファイルモードは `0644` で、書き込み可能のファイルモードは `0666` になるはずですが、これは、セキュリティ上の理由から、HTTP サーバがあなたのスクリプトを特権を全く持たないユーザ “nobody” の権限で実行するからです。この権限下では、誰でもが読める (書ける、実行できる) ファイルしか読み出し (書き込み、実行) できません。スクリプト実行時のディレクトリや環境変数のセットもあなたがログインしたときの設定と異なります (カレントディレクトリは普通サーバの `cgi-bin` ディレクトリです)。特に、実行ファイルに対するシェルの検索パス (`PATH`) や Python のモジュール検索パス (`PYTHONPATH`) が何らかの値に設定されていると期待してはいけません。

モジュールを Python の標準設定におけるモジュール検索パス上にないディレクトリからロードする必要がある場合、他のモジュールを取り込む前にスクリプト内で検索パスを変更できます。例えば:

```
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
sys.path.insert(0, "/usr/local/lib/python")
```

(この方法では、最後に挿入されたディレクトリが最初に検索されます!)

非 Unix システムにおける説明は変わるでしょう; あなたの使っている HTTP サーバのドキュメントを調べてください (普通は CGI スクリプトに関する節があります)。

20.2.8 CGI スクリプトをテストする

残念ながら、CGI スクリプトは普通、コマンドラインから起動しようとしても動きません。また、コマンドラインから起動した場合には完璧に動作するスクリプトが、不思議なことにサーバからの起動では失敗することがあります。しかし、スクリプトをコマンドラインから実行してみなければならない理由が一つあります: もしスクリプトが文法エラーを含んでいれば、Python インタプリタはそのプログラムを全く実行しないため、HTTP サーバはほとんどの場合クライアントに謎めいたエラーを送信するからです。

スクリプトが構文エラーを含まないのにうまく動作しないなら、次の節に進むしかありません。

20.2.9 CGI スクリプトをデバッグする

何よりもまず、些細なインストール関連のエラーでないか確認してください — 上の CGI スクリプトのインストールに関する節を注意深く読めば時間を大いに節約できます。もしインストールの手続きを正しく理解しているか不安なら、このモジュールのファイル (`cgi.py`) をコピーして、CGI スクリプトとしてインストールしてみてください。このファイルはスクリプトとして呼び出すと、スクリプトの実行環境とフォームの内容を HTML 形式で出力します。ファイルに正しいモードを設定するなどして、リクエストを送ってみてください。

標準的な `cgi-bin` ディレクトリにインストールされていれば、以下のような URL をブラウザに入力してリクエストを送信できるはずです:

```
http://yourhostname/cgi-bin/cgi.py?name=Joe+Blow&addr=At+Home
```

もしタイプ 404 のエラーになるなら、サーバはスクリプトを発見できないでいます – おそらくあなたはスクリプトを別のディレクトリに入れる必要があるのでしょう。他のエラーになるなら、先に進む前に解決しなければならないインストール上の問題があります。もし実行環境の情報とフォーム内容 (この例では、各フィールドはフィールド名 "addr" に対して値 "At Home"、およびフィールド名 "name" に対して "Joe Blow") が綺麗にフォーマットされて表示されるなら、`cgi.py` スクリプトは正しくインストールされています。同じ操作をあなたの自作スクリプトに対して行えば、スクリプトをデバッグできるようになるはずです。

次のステップでは `cgi` モジュールの `test()` 関数を呼び出すことになります: メインプログラムコードを以下の 1 文と置き換えてください

```
cgi.test()
```

この操作で `cgi.py` ファイル自体をインストールした時と同じ結果を出力するはずです。

通常の Python スクリプトが例外を処理しきれずに送出した場合 (様々な理由: モジュール名のタイプミス、ファイルが開けなかった、など)、Python インタプリタはナイスなトレースバックを出力して終了します。Python インタプリタはあなたの CGI スクリプトが例外を送出した場合にも同様に振舞うので、トレースバックは大抵 HTTP サーバのいずれかのログファイルに残るかまったく無視されるかです。

幸運なことに、あなたが自作のスクリプトで何らかのコードを実行できるようになったら、`cgitb` モジュールを使って簡単にトレースバックをブラウザに送信できます。まだそうでないなら、以下の 2 行:

```
import cgitb
cgitb.enable()
```

をスクリプトの先頭に追加してください。そしてスクリプトを再度走らせます; 問題が発生すれば、クラッシュの原因を見出せるような詳細な報告を読めます。

`cgitb` モジュールのインポートに問題がありそうだと思うなら、(組み込みモジュールだけを使った) もっと堅牢なアプローチを取れます:

```
import sys
sys.stderr = sys.stdout
print "Content-Type: text/plain"
print
...your code here...
```

このコードは Python インタプリタがトレースバックを出力することに依存しています。出力のコンテンツ型はプレーンテキストに設定されており、全ての HTML 処理を無効にしています。スクリプトがうまく動作する場合、生の HTML コードがクライアントに表示されます。スクリプトが例外を送出する場合、最初の 2 行が出力された後、トレースバックが表示されます。HTML の解釈は行われないので、トレースバックを読むはずですが。

20.2.10 よくある問題と解決法

- ほとんどの HTTP サーバはスクリプトの実行が完了するまで CGI からの出力をバッファします。このことは、スクリプトの実行中にクライアントが進捗状況報告を表示できないことを意味します。
- 上のインストールに関する説明を調べましょう。
- HTTP サーバのログファイルを調べましょう。(別のウィンドウで `tail -f logfile` を実行すると便利かもしれません！)
- 常に `python script.py` などとして、スクリプトが構文エラーでないか調べましょう。
- スクリプトに構文エラーがないなら、`import cgitb; cgitb.enable()` をスクリプトの先頭に追加してみましょう。
- 外部プログラムを起動するときには、スクリプトがそのプログラムを見つけられるようにしましょう。これは通常、絶対パス名を使うことを意味します — `PATH` は普通、あまり CGI スクリプトにとって便利でない値に設定されています。
- 外部のファイルを読み書きする際には、CGI スクリプトを動作させるときに使われる `userid` でファイルを読み書きできるようになっているか確認しましょう: `userid` は通常、Web サーバを動作させている `userid` か、Web サーバの `suexec` 機能で明示的に指定している `userid` になります。
- CGI スクリプトを `set-uid` モードにしてはいけません。これはほとんどのシステムで動作せず、セキュリティ上の信頼性もありません。

20.3 cgitb — CGI スクリプトのトレースバック管理機構

バージョン 2.2 で追加.

`cgitb` モジュールでは、Python スクリプトのための特殊な例外処理を提供します。(実はこの説明は少し的外れです。このモジュールはもともと徹底的なトレースバック情報を CGI スクリプトで生成した HTML 内に表示するための設計されました。その後この情報を平文テキストでも表示できるように一般化されています。) このモジュールの有効化後に捕捉されない例外が生じた場合、詳細で書式化された報告が Web ブラウザに送信されます。この報告には各レベルにおけるソースコードの抜粋が示されたトレースバックと、現在動作している関数の引数やローカルな変数が収められており、問題のデバッグを助けます。オプションとして、この情報をブラウザに送信する代わりにファイルに保存することもできます。

この機能を有効化するためには、単に自作の CGI スクリプトの最初に以下の 2 行を追加します:

```
import cgitb
cgitb.enable()
```

`enable()` 関数のオプションは、報告をブラウザに表示するかどうかと、後で解析するためにファイルに報告をログ記録するかどうかを制御します。

```
cgitb.enable([display[, logdir[, context[, format]]]])
```

この関数は、`sys.excepthook` を設定することで、インタプリタの標準の例外処理を `cgitb` モ

ジュールに肩代わりさせるようにします。

オプションの引数 `display` は標準で 1 になっており、この値は 0 にしてトレースバックをブラウザに送らないように抑制することもできます。引数 `logdir` が存在すれば、トレースバックレポートはそのファイルに書き込まれます。`logdir` の値はログファイルを配置するディレクトリです。オプションの引数 `context` は、トレースバックの中で現在の行の周辺の何行を表示するかです; この値は標準で 5 です。オプションの引数 `format` が "html" の場合、出力は HTML に書式化されます。その他の値を指定すると平文テキストの出力を強制します。デフォルトの値は "html" です。

`cgitb.handler([info])`

この関数は標準の設定 (ブラウザに報告を表示しますがファイルにはログを書き込みません) を使って例外を処理します。この関数は、例外を捕捉した際に `cgitb` を使って報告したい場合に使うことができます。オプションの `info` 引数は、例外の型、例外の値、トレースバックオブジェクトからなる 3 要素のタプルでなければなりません。これは `sys.exc_info()` によって返される値と全く同じです。`info` 引数が与えられていない場合、現在の例外は `sys.exc_info()` から取得されます。

20.4 wsgiref — WSGI ユーティリティとリファレンス実装

バージョン 2.5 で追加.

Web Server Gateway Interface (WSGI) は、Web サーバソフトウェアと Python で記述された Web アプリケーションとの標準インターフェースです。標準インターフェースを持つことで、WSGI をサポートするアプリケーションを幾つもの異なる Web サーバで使うことが容易になります。

Web サーバとプログラミングフレームワークの作者だけが、WSGI デザインのあらゆる細部や特例などを知る必要があります。WSGI アプリケーションをインストールしたり、既存のフレームワークを使ったアプリケーションを記述するだけの皆さんは、全てについて理解する必要はありません。

`wsgiref` は WSGI 仕様のリファレンス実装で、これは Web サーバやフレームワークに WSGI サポートを加えるのに利用できます。これは WSGI 環境変数やレスポンスヘッダを操作するユーティリティ、WSGI サーバ実装時のベースクラス、WSGI アプリケーションを提供する デモ用 HTTP サーバ、それと WSGI サーバとアプリケーションの WSGI 仕様 ([PEP 333](#)) 準拠のバリデーションツールを提供します。

<https://wsgi.readthedocs.org/> に、WSGI に関するさらなる情報と、チュートリアルやその他のリソースへのリンクがあります。

20.4.1 wsgiref.util — WSGI 環境のユーティリティ

このモジュールは WSGI 環境で使う様々なユーティリティ関数を提供します。WSGI 環境は [PEP 333](#) で記述されているような HTTP リクエスト変数を含む辞書です。全ての `environ` パラメータを取る関数は WSGI 準拠の辞書を与えられることを期待しています; 細かい仕様については [PEP 333](#) を参照してください。

`wsgiref.util.guess_scheme(environ)`

`environ` 辞書の HTTPS 環境変数を調べることで `wsgi.url_scheme` が "http" か "https" のどちらであるべきか推測し、その結果を返します。戻り値は文字列です。

この関数は、CGI や FastCGI のような CGI に似たプロトコルをラップするゲートウェイを作成する場合に便利です。典型的には、それらのプロトコルを提供するサーバが SSL 経由でリクエストを受け取った場合には HTTPS 変数に値 "1" "yes"、または "on" を持つでしょう。そのため、この関数はそのような値が見つかった場合には "https" を返し、そうでなければ "http" を返します。

`wsgiref.util.request_uri (environ, include_query=1)`

リクエスト URI 全体 (オプションでクエリ文字列を含む) を、[PEP 333](#) の "URL 再構築 (URL Reconstruction)" にあるアルゴリズムを使って返します。 `include_query` が `false` の場合、クエリ文字列は結果となる文字列には含まれません。

`wsgiref.util.application_uri (environ)`

`PATH_INFO` と `QUERY_STRING` 変数が無視されることを除けば `request_uri()` に似ています。結果はリクエストによって指定されたアプリケーションオブジェクトのベース URI です。

`wsgiref.util.shift_path_info (environ)`

`PATH_INFO` から `SCRIPT_NAME` に一つの名前をシフトしてその名前を返します。 `environ` 辞書は 変更されます。 `PATH_INFO` や `SCRIPT_NAME` のオリジナルをそのまま残したい場合にはコピーを使ってください。

`PATH_INFO` にパスセグメントが何も残っていなければ、`None` が返されます。

典型的なこのルーチンの使い方はリクエスト URI のそれぞれの要素の処理で、例えばパスを一連の辞書のキーとして取り扱う場合です。このルーチンは、渡された環境を、ターゲット URL で示される別の WSGI アプリケーションの呼び出しに合うように調整します。例えば、`/foo` に WSGI アプリケーションがあったとして、そしてリクエスト URL パスが `/foo/bar/baz` で、`/foo` の WSGI アプリケーションが `shift_path_info()` を呼んだ場合、これは "bar" 文字列を受け取り、`environ` は `/foo/bar` の WSGI アプリケーションへの受け渡しに適するように更新されます。つまり、`SCRIPT_NAME` は `/foo` から `/foo/bar` に変わって、`PATH_INFO` は `/bar/baz` から `/baz` に変化するのです。

`PATH_INFO` が単に `"/` の場合、このルーチンは空の文字列を返し、`SCRIPT_NAME` の末尾にスラッシュを加えます、これはたとえ空のパスセグメントが通常は無視され、そして `SCRIPT_NAME` は通常スラッシュで終わる事が無かったとしてもです。これは意図的な振る舞いで、このルーチンでオブジェクト巡回 (object traversal) をした場合に `/x` で終わる URI と `/x/` で終わるものをアプリケーションが識別できることを保証するためのものです。

`wsgiref.util.setup_testing_defaults (environ)`

`environ` をテスト用に自明なデフォルト値で更新します。

このルーチンは WSGI に必要な様々なパラメータを追加します。そのようなパラメータとして `HTTP_HOST`、`SERVER_NAME`、`SERVER_PORT`、`REQUEST_METHOD`、`SCRIPT_NAME`、`PATH_INFO`、そして [PEP 333](#) で定義されている `wsgi.*` 変数群が含まれます。このルーチンはデフォルト値を提供するだけで、これらの変数の既存設定は一切置きかえません。

このルーチンは、ダミー環境をセットアップすることによって WSGI サーバとアプリケーションのユニットテストを容易にすることを意図しています。これは実際の WSGI サーバやアプリケーションで使うべきではありません。なぜならこのデータは偽物なのです！

使い方の例:

```
from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# A relatively simple WSGI application. It's going to print out the
# environment dictionary after being updated by setup_testing_defaults
def simple_app(environ, start_response):
    setup_testing_defaults(environ)

    status = '200 OK'
    headers = [('Content-type', 'text/plain')]

    start_response(status, headers)

    ret = ["%s: %s\n" % (key, value)
            for key, value in environ.iteritems()]
    return ret

httpd = make_server('', 8000, simple_app)
print "Serving on port 8000..."
httpd.serve_forever()
```

上記の環境用関数に加えて、`wsgiref.util` モジュールも以下のようなその他のユーティリティを提供します:

`wsgiref.util.is_hop_by_hop(header_name)`

‘header_name’ が **RFC 2616** で定義されている HTTP/1.1 の “Hop-by-Hop” ヘッダの場合に `true` を返します。

class `wsgiref.util.FileWrapper(filelike, blksize=8192)`

ファイル風オブジェクトをイテレータ (*iterator*) に変換するラップです。結果のオブジェクトは `__getitem__()` と `__iter__()` 両方をサポートしますが、これは Python 2.1 と Jython の互換性のためです。オブジェクトがイテレートされる間、オプションの `blksize` パラメータがくり返し `filelike` オブジェクトの `read()` メソッドに渡されて受け渡す文字列を取得します。 `read()` が空文字列を返した場合、イテレーションは終了して再開されることはありません。

`filelike` に `close()` メソッドがある場合、返されたオブジェクトも `close()` メソッドを持ち、これが呼ばれた場合には `filelike` オブジェクトの `close()` メソッドを呼び出します。

使い方の例:

```
from StringIO import StringIO
from wsgiref.util import FileWrapper

# We're using a StringIO-buffer for as the file-like object
filelike = StringIO("This is an example file-like object"*10)
wrapper = FileWrapper(filelike, blksize=5)

for chunk in wrapper:
    print chunk
```

20.4.2 wsgiref.headers – WSGI レスponsヘッダツール群

このモジュールは単一のクラス、`Headers` を提供し、WSGI レスponsヘッダの操作をマップに似たインターフェースで便利にします。

class `wsgiref.headers.Headers` (*headers*)

`headers` をラップするマップ風オブジェクトを生成します。これは [PEP 333](#) に定義されるようなヘッダの名前 / 値のタプルのリストです。新しい `Headers` オブジェクトに与えられた変更は、一緒に作成された `headers` リストを直接更新します。

`Headers` オブジェクトは典型的なマッピング操作をサポートし、これには `__getitem__()`、`get()`、`__setitem__()`、`setdefault()`、`__delitem__()`、`__contains__()`、`has_key()` を含みます。これらメソッドのそれぞれにおいて、キーはヘッダ名で（大文字小文字は区別しません）、値はそのヘッダ名に関連づけられた最初の値です。ヘッダを設定すると既存のヘッダ値は削除され、ラップされたヘッダのリストの末尾に新しい値が加えられます。既存のヘッダの順番は一般に維持され、ラップされたリストの最後に新しいヘッダが追加されます。

辞書とは違って、`Headers` オブジェクトはラップされたヘッダリストに存在しないキーを取得または削除しようとした場合にもエラーを発生しません。単に、存在しないヘッダの取得は `None` を返し、存在しないヘッダの削除は何もしません。

`Headers` オブジェクトは `keys()`、`values()`、`items()` メソッドもサポートします。複数の値を持つヘッダがある場合には、`keys()` と `items()` で返されるリストは同じキーを一つ以上含むことがあります。`Headers` オブジェクトの `len()` は、その `items()` の長さと同じであり、ラップされたヘッダリストの長さと同じです。実際、`items()` メソッドは単にラップされたヘッダリストのコピーを返しているだけです。

`Headers` オブジェクトに対して `str()` を呼ぶと、HTTP レスponsヘッダとして送信するのに適した形に整形された文字列を返します。それぞれのヘッダはコロンとスペースで区切られた値と共に一列に並んでいます。それぞれの行はキャリッジリターンとラインフィードで終了し、文字列は空行で終了しています。

これらのマッピングインターフェースと整形機能に加えて、`Headers` オブジェクトは複数の値を持つヘッダの取得と追加、MIME パラメータでヘッダを追加するための以下のようなメソッド群も持っています：

get_all (*name*)

指定されたヘッダの全ての値のリストを返します。

返されるリストは、元々のヘッダリストに現れる順、またはこのインスタンスに追加された順に並んでいて、重複を含む場合があります。削除されて加えられたフィールドは全てヘッダリストの末尾に付きます。与えられた `name` に対するフィールドが何もなければ、空のリストが返ります。

add_header (*name*, *value*, ***params*)

(複数の値を持つ可能性のある) ヘッダを、キーワード引数を通じて指定するオプションの MIME パラメータと共に追加します。

`name` は追加するヘッダフィールドです。このヘッダフィールドに MIME パラメータを設定するためにキーワード引数を使うことができます。それぞれのパラメータは文字列か `None` でなければ

いけません。パラメータ中のアンダースコアはダッシュ (-) に変換されます。これは、ダッシュが Python の識別子としては不正なのですが、多くの MIME パラメータはダッシュを含むためです。パラメータ値が文字列の場合、これはヘッダ値のパラメータに `name="value"` の形で追加されます。この値がもし `None` の場合、パラメータ名だけが追加されます。(これは値なしの MIME パラメータの場合に使われます。) 使い方の例は:

```
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

上記はこのようなヘッダを追加します:

```
Content-Disposition: attachment; filename="bud.gif"
```

20.4.3 `wsgiref.simple_server` – シンプルな WSGI HTTP サーバ

このモジュールは WSGI アプリケーションを提供するシンプルな HTTP サーバです (`BaseHTTPServer` がベースです)。個々のサーバインスタンスは単一の WSGI アプリケーションを、特定のホストとポート上で提供します。もし一つのホストとポート上で複数のアプリケーションを提供したいならば、`PATH_INFO` をパースして個々のリクエストでどのアプリケーションを呼び出すか選択するような WSGI アプリケーションを作る必要があります。(例えば、`wsgiref.util` から `shift_path_info()` を利用します。)

`wsgiref.simple_server.make_server(host, port, app, server_class=WSGIServer, handler_class=WSGIRequestHandler)`

`host` と `port` 上で待機し、`app` へのコネクションを受け付ける WSGI サーバを作成します。戻り値は与えられた `server_class` のインスタンスで、指定された `handler_class` を使ってリクエストを処理します。`app` は [PEP 333](#) で定義されるところの WSGI アプリケーションでなければいけません。

使い方の例:

```
from wsgiref.simple_server import make_server, demo_app

httpd = make_server(' ', 8000, demo_app)
print "Serving HTTP on port 8000..."

# Respond to requests until process is killed
httpd.serve_forever()

# Alternative: serve one request, then exit
httpd.handle_request()
```

`wsgiref.simple_server.demo_app(envIRON, start_response)`

この関数は小規模ながら完全な WSGI アプリケーションで、`"Hello world!"` メッセージと、`envIRON` パラメータに提供されているキー/値のペアを含むテキストページを返します。これは WSGI サーバ (`wsgiref.simple_server` のような) がシンプルな WSGI アプリケーションを正しく実行できるかを確かめるのに便利です。

`class wsgiref.simple_server.WSGIServer(server_address, RequestHandlerClass)`

`WSGIServer` インスタンスを作成します。 `server_address` は `(host, port)` のタプル

ル、そして `RequestHandlerClass` はリクエストの処理に使われる `BaseHTTPServer.BaseHTTPRequestHandler` のサブクラスでなければいけません。

`make_server()` が細かい調整をしてくれるので、通常はこのコンストラクタを呼ぶ必要はありません。

`WSGIServer` は `BaseHTTPServer.HTTPServer` のサブクラスなので、その全てのメソッド (`serve_forever()` や `handle_request()` のような) が利用できます。 `WSGIServer` も以下のような WSGI 固有メソッドを提供します:

set_app(application)

呼び出し可能 (callable) な `application` をリクエストを受け取る WSGI アプリケーションとして設定します。

get_app()

現在設定されている呼び出し可能 (callable) アプリケーションを返します。

しかしながら、通常はこれらの追加されたメソッドを使う必要はありません。 `set_app()` は普通は `make_server()` によって呼ばれ、 `get_app()` は主にリクエストハンドラインスタンスの便宜上存在するからです。

class `wsgiref.simple_server.WSGIRequestHandler(request, client_address, server)`

与えられた `request` (すなわちソケット) の HTTP ハンドラ、 `client_address` ((`host`, `port`) のタプル)、 `server` (`WSGIServer` インスタンス) の HTTP ハンドラを作成します。

このクラスのインスタンスを直接生成する必要はありません; これらは必要に応じて `WSGIServer` オブジェクトによって自動的に生成されます。しかしながら、このクラスをサブクラス化し、 `make_server()` 関数に `handler_class` として与えることは可能でしょう。サブクラスにおいてオーバーライドする意味のありそうなものは:

get_environ()

リクエストに対する WSGI 環境を含む辞書を返します。デフォルト実装では `WSGIServer` オブジェクトの `base_environ` 辞書属性のコンテンツをコピーし、それから HTTP リクエスト由来の様々なヘッダを追加しています。このメソッド呼び出し毎に、 **PEP 333** に指定されている関連する CGI 環境変数を全て含む新規の辞書を返さなければいけません。

get_stderr()

`wsgi.errors` ストリームとして使われるオブジェクトを返します。デフォルト実装では単に `sys.stderr` を返します。

handle()

HTTP リクエストを処理します。デフォルト実装では実際の WGI アプリケーションインターフェースを実装するのに `wsgiref.handlers` クラスを使ってハンドラインスタンスを作成します。

20.4.4 wsgiref.validate — WSGI 準拠チェッカー

WSGI アプリケーションのオブジェクト、フレームワーク、サーバまたはミドルウェアの作成時には、その新規のコードを `wsgiref.validate` を使って準拠の検証をすると便利です。このモジュールは WSGI サーバやゲートウェイと WSGI アプリケーションオブジェクト間の通信を検証する WSGI アプリケーションオブジェクトを作成する関数を提供し、双方のプロトコル準拠をチェックします。

このユーティリティは完全な [PEP 333](#) 準拠を保証するものでないことは注意してください; このモジュールでエラーが出ないことは必ずしもエラーが存在しないことを意味しません。しかしこのモジュールがエラーを出したならば、ほぼ確実にサーバかアプリケーションのどちらかが 100% 準拠ではありません。

このモジュールは Ian Bicking の "Python Paste" ライブラリの `paste.lint` モジュールをベースにしています。

`wsgiref.validate.validator(application)`

`application` をラップし、新しい WSGI アプリケーションオブジェクトを返します。返されたアプリケーションは全てのリクエストを元々の `application` に転送し、`application` とそれを呼び出すサーバの両方が WSGI 仕様と RFC 2616 の両方に準拠しているかをチェックします。

何らかの非準拠が検出されると、`AssertionError` 例外が送出されます; しかし、このエラーがどう扱われるかはサーバ依存であることに注意してください。例えば、`wsgiref.simple_server` とその他 `wsgiref.handlers` ベースのサーバ (エラー処理メソッドが他のことをするようにオーバライドしていないもの) は単純にエラーが発生したというメッセージとトレースバックのダンプを `sys.stderr` やその他のエラーストリームに出力します。

このラップは、疑わしいものの実際には [PEP 333](#) で禁止されていないかもしれない挙動を指摘するために `warnings` モジュールを使って出力を生成します。これらは Python のコマンドラインオプションや `warnings` API で抑制されなければ、`sys.stderr` (`wsgi.errors` ではありません。ただし、たまたま同一のオブジェクトだった場合を除く) に書き出されます。

使い方の例:

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# Our callable object which is intentionally not compliant to the
# standard, so the validator is going to break
def simple_app(environ, start_response):
    status = '200 OK' # HTTP Status
    headers = [('Content-type', 'text/plain')] # HTTP Headers
    start_response(status, headers)

    # This is going to break because we need to return a list, and
    # the validator is going to inform us
    return "Hello World"

# This is the application wrapped in a validator
validator_app = validator(simple_app)
```

(次のページに続く)

(前のページからの続き)

```
httpd = make_server('', 8000, validator_app)
print "Listening on port 8000...."
httpd.serve_forever()
```

20.4.5 wsgiref.handlers – サーバ/ゲートウェイのベースクラス

このモジュールは WSGI サーバとゲートウェイ実装のベースハンドラクラスを提供します。これらのベースクラスは、CGI 風の環境と、それに加えて入力、出力そしてエラーストリームが与えられることで、WSGI アプリケーションとの通信の大部分を処理します。

class wsgiref.handlers.CGIHandler

`sys.stdin`、`sys.stdout`、`sys.stderr` そして `os.environ` 経由での CGI ベースの呼び出しです。これは、もしあなたが WSGI アプリケーションを持っていて、これを CGI スクリプトとして実行したい場合に有用です。単に `CGIHandler().run(app)` を起動してください。app はあなたが起動したい WSGI アプリケーションオブジェクトです。

このクラスは *BaseCGIHandler* のサブクラスで、これは `wsgi.run_once` を `true`、`wsgi.multithread` を `false`、そして `wsgi.multiprocess` を `true` にセットし、常に `sys` と `os` を、必要な CGI ストリームと環境を取得するために使用します。

class wsgiref.handlers.BaseCGIHandler(*stdin, stdout, stderr, environ, multithread=True, multiprocess=False*)

CGIHandler に似ていますが、`sys` と `os` モジュールを使う代わりに CGI 環境と I/O ストリームを明示的に指定します。`multithread` と `multiprocess` の値は、ハンドラインスタンスにより実行されるアプリケーションの `wsgi.multithread` と `wsgi.multiprocess` フラグの設定に使われます。

このクラスは *SimpleHandler* のサブクラスで、HTTP の ”本サーバ” でないソフトウェアと使うことを意図しています。もしあなたが `Status: ヘッダ` を HTTP ステータスを送信するのに使うようなゲートウェイプロトコルの実装 (CGI、FastCGI、SCGI など) を書いている場合、おそらく *SimpleHandler* ではなくこのクラスをサブクラス化するとよいでしょう。

class wsgiref.handlers.SimpleHandler(*stdin, stdout, stderr, environ, multithread=True, multiprocess=False*)

BaseCGIHandler と似ていますが、HTTP の本サーバと使うためにデザインされています。もしあなたが HTTP サーバ実装を書いている場合、おそらく *BaseCGIHandler* ではなくこのクラスをサブクラス化するとよいでしょう。

このクラスは *BaseHandler* のサブクラスです。これは `__init__()`、`get_stdin()`、`get_stderr()`、`add_cgi_vars()`、`_write()`、`_flush()` をオーバーライドして、コンストラクタから明示的に環境とストリームを設定するようにしています。与えられた環境とストリームは `stdin`、`stdout`、`stderr` それに `environ` 属性に保存されています。

class wsgiref.handlers.BaseHandler

これは WSGI アプリケーションを実行するための抽象ベースクラスです。それぞれのインスタンスは一つの HTTP リクエストを処理します。しかし原理上は複数のリクエスト用に再利用可能なサブクラスを作成することができます。

`BaseHandler` インスタンスは外部から利用されるたった一つのメソッドを持ちます:

run(*app*)

指定された WSGI アプリケーション、*app* を実行します。

その他の全ての `BaseHandler` のメソッドはアプリケーション実行プロセスでこのメソッドから呼ばれます。したがって、それらは主にそのプロセスのカスタマイズのために存在しています。

以下のメソッドはサブクラスでオーバーライドされなければいけません:

_write(*data*)

文字列の *data* をクライアントへの転送用にバッファします。このメソッドが実際にデータを転送しても OK です: 下部システムが実際にそのような区別をしている場合に効率をより良くするために、`BaseHandler` は書き出しとフラッシュ操作を分けているからです。

_flush()

バッファされたデータをクライアントに強制的に転送します。このメソッドは何もしなくても OK です (すなわち、`_write()` が実際にデータを送る場合)。

get_stdin()

現在処理中のリクエストの `wsgi.input` としての利用に適当な入力ストリームオブジェクトを返します。

get_stderr()

現在処理中のリクエストの `wsgi.errors` としての利用に適当な出力ストリームオブジェクトを返します。

add_cgi_vars()

現在のリクエストの CGI 変数を `environ` 属性に追加します。

オーバーライドされることの多いメソッド及び属性を以下に挙げます。しかし、このリストは単にサマリーであり、オーバーライド可能な全てのメソッドは含んでいません。カスタマイズした `BaseHandler` サブクラスを作成しようとする前に docstring やソースコードでさらなる情報を調べてください。

WSGI 環境のカスタマイズのための属性とメソッド:

wsgi.multithread

`wsgi.multithread` 環境変数で使われる値。 `BaseHandler` ではデフォルトが `true` ですが、別のサブクラスではデフォルトで (またはコンストラクタによって設定されて) 異なる値を持つことがあります。

wsgi.multiprocess

`wsgi.multiprocess` 環境変数で使われる値。 `BaseHandler` ではデフォルトが `true` ですが、別のサブクラスではデフォルトで (またはコンストラクタによって設定されて) 異なる値を持つことがあります。

wsgi.run_once

`wsgi.run_once` 環境変数で使われる値。 `BaseHandler` ではデフォルトが `false` ですが、`CGIHandler` はデフォルトでこれを `true` に設定します。

os.environ

全てのリクエストの WSGI 環境に含まれるデフォルトの環境変数。デフォルトでは `wsgiref.handlers` がインポートされた時点の `os.environ` のコピーですが、サブクラスはクラスまたはインスタンスレベルでそれら自身のものを作ることができます。デフォルト値は複数のクラスとインスタンスで共有されるため、この辞書は読み取り専用と考えるべきだという点に注意してください。

server_software

`origin_server` 属性が設定されている場合、この属性の値がデフォルトの `SERVER_SOFTWARE` WSGI 環境変数の設定や HTTP レスポンス中のデフォルトの `Server:` ヘッダの設定に使われます。これは (`BaseCGIHandler` や `CGIHandler` のような) HTTP オリジンサーバでないハンドラでは無視されます。

get_scheme()

現在のリクエストで使われている URL スキームを返します。デフォルト実装は `wsgiref.util` の `guess_scheme()` を使い、現在のリクエストの `environ` 変数に基づいてスキームが "http" が "https" かを推測します。

setup_environ()

`environ` 属性を、フル実装 (fully-populated) の WSGI 環境に設定します。デフォルトの実装は、上記全てのメソッドと属性、加えて `get_stdin()`、`get_stderr()`、`add_cgi_vars()` メソッドと `wsgi_file_wrapper` 属性を利用します。これは、キーが存在せず、`origin_server` 属性が `true` 値で `server_software` 属性も設定されている場合に `SERVER_SOFTWARE` を挿入します。

例外処理のカスタマイズのためのメソッドと属性:

log_exception(exc_info)

`exc_info` タプルをサーバログに記録します。`exc_info` は (`type`, `value`, `traceback`) のタプルです。デフォルトの実装は単純にトレースバックをリクエストの `wsgi.errors` ストリームに書き出してフラッシュします。サブクラスはこのメソッドをオーバーライドしてフォーマットを変更したり出力先の変更、トレースバックを管理者にメールしたりその他適切と思われるいかなるアクションも取ることができます。

traceback_limit

デフォルトの `log_exception()` メソッドで出力されるトレースバック出力に含まれる最大のフレーム数です。None ならば、全てのフレームが含まれます。

error_output(environ, start_response)

このメソッドは、ユーザに対してエラーページを出力する WSGI アプリケーションです。これはクライアントにヘッダが送出される前にエラーが発生した場合にのみ呼び出されます。

このメソッドは `sys.exc_info()` を使って現在のエラー情報にアクセスでき、その情報はこれを呼ぶときに `start_response` に渡すべきです ([PEP 333](#) の "Error Handling" セクションに記述があります)。

デフォルト実装は単に `error_status`、`error_headers`、`error_body` 属性を出力ページの生成に使います。サブクラスではこれをオーバーライドしてもっと動的なエラー出力をすること

ができます。

しかし、セキュリティの観点からは診断をあらゆるユーザに吐き出すことは推奨されないことに気をつけてください; 理想的には、診断的な出力を有効にするには何らかの特別なことをする必要があります。あるようにすべきで、これがデフォルト実装では何も含まれていない理由です。

error_status

エラーレスポンスで使われる HTTP ステータスです。これは [PEP 333](#) で定義されているステータス文字列です; デフォルトは 500 コードとメッセージです。

error_headers

エラーレスポンスで使われる HTTP ヘッダです。これは [PEP 333](#) で述べられているような、WSGI レスポンスヘッダ ((name, value) タプル) のリストであるべきです。デフォルトのリストはコンテンツタイプを `text/plain` にセットしているだけです。

error_body

エラーレスポンスボディ。これは HTTP レスポンスのボディ文字列であるべきです。これはデフォルトではプレーンテキストで "A server error occurred. Please contact the administrator." です。

[PEP 333](#) の "オプションのプラットフォーム固有のファイルハンドリング" 機能のためのメソッドと属性:

wsgi_file_wrapper

`wsgi.file.wrapper` ファクトリ、または `None` です。この属性のデフォルト値は `wsgiref.util.FileWrapper` クラスです。

sendfile()

オーバーライドしてプラットフォーム固有のファイル転送を実装します。このメソッドはアプリケーションの戻り値が `wsgi.file.wrapper` 属性で指定されたクラスのインスタンスの場合にのみ呼ばれます。これはファイルの転送が成功できた場合には `true` を返して、デフォルトの転送コードが実行されないようにするべきです。このデフォルトの実装は単に `false` 値を返します。

その他のメソッドと属性:

origin_server

この属性はハンドラの `_write()` と `_flush()` が、特別に `Status:` ヘッダに HTTP ステータスを求めるような CGI 風のゲートウェイプロトコル経由でなく、クライアントと直接通信をするような場合には `true` 値に設定されているべきです。

この属性のデフォルト値は `BaseHandler` では `true` ですが、`BaseCGIHandler` と `CGIHandler` では `false` です。

http_version

`origin_server` が `true` の場合、この文字列属性はクライアントへのレスポンスセットの HTTP バージョンの設定に使われます。デフォルトは "1.0" です。

20.4.6 例

これは動作する "Hello World" WSGI アプリケーションです:

```
from wsgiref.simple_server import make_server

# Every WSGI application must have an application object - a callable
# object that accepts two arguments. For that purpose, we're going to
# use a function (note that you're not limited to a function, you can
# use a class for example). The first argument passed to the function
# is a dictionary containing CGI-style environment variables and the
# second variable is the callable object (see PEP 333).
def hello_world_app(environ, start_response):
    status = '200 OK' # HTTP Status
    headers = [('Content-type', 'text/plain')] # HTTP Headers
    start_response(status, headers)

    # The returned object is going to be printed
    return ["Hello World"]

httpd = make_server('', 8000, hello_world_app)
print "Serving on port 8000..."

# Serve until process is killed
httpd.serve_forever()
```

20.5 urllib — URL による任意のリソースへのアクセス

注釈: `urllib` モジュールは、Python 3 で `urllib.request`, `urllib.parse`, `urllib.error` に分割されて名称変更されました。 *2to3* ツールが自動的にソースコードの `import` を修正します。また、Python 3 の `urllib.request.urlopen()` 関数は `urllib2.urlopen()` を移動したもので、`urllib.urlopen()` のほうは削除されています。

このモジュールはワールドワイドウェブ (World Wide Web) を介してデータを取り寄せるための高レベルのインタフェースを提供します。特に、関数 `urlopen()` は組み込み関数 `open()` と同様に動作し、ファイル名の代わりにファイルユニバーサルリソースロケータ (URL) を指定することができます。いくつかの制限はあります — URL は読み出し専用でしか開けませんし、`seek` 操作を行うことはできません。

参考:

より高水準の HTTP クライアントインターフェイスとしては `Requests` パッケージがお奨めです。

バージョン 2.7.9 で変更: HTTPS URI の場合、`urllib` は必要な証明書検証とホスト名チェックを全てデフォルトで行います。

警告: Python の 2.7.9 より前のバージョンでは `urllib` は HTTPS URI のサーバ証明書を検証しようとはしません。自己責任でお使いください!

20.5.1 高レベルインタフェース

`urllib.urlopen(url[, data[, proxies[, context]]])`

URL で表されるネットワーク上のオブジェクトを読み込み用に開きます。URL がスキーム識別子を持たないか、スキーム識別子が `file:` である場合、ローカルシステムのファイルが (*universal newlines* なしで) 開かれます。それ以外の場合はネットワーク上のどこかにあるサーバへのソケットを開きます。接続を作ることができない場合、例外 `IOError` が送出されます。全ての処理がうまくいけば、ファイル類似のオブジェクトが返されます。このオブジェクトは以下のメソッド: `read()`, `readline()`, `readlines()`, `fileno()`, `close()`, `info()`, `getcode()`, `geturl()` をサポートします。また、*iterator* プロトコルも正しくサポートしています。注意: `read()` の引数を省略または負の値を指定しても、データストリームの最後まで読みこむ訳ではありません。ソケットからすべてのストリームを読み込んだことを決定する一般的な方法は存在しません。

`info()`, `getcode()`, `geturl()` メソッドを除き、これらのメソッドはファイルオブジェクトと同じインタフェースを持っています — このマニュアルの [ファイルオブジェクト](#) セクションを参照してください。(このオブジェクトは組み込みのファイルオブジェクトではありませんが、まれに本物の組み込みファイルオブジェクトが必要な場所で使うことができません)

`info()` メソッドは開いた URL に関連付けられたメタ情報を含む `mimertools.Message` クラスのインスタンスを返します。URL へのアクセスメソッドが HTTP である場合、メタ情報中のヘッダ情報はサーバが HTML ページを返すときに先頭に付加するヘッダ情報です (Content-Length および Content-Type を含みます)。アクセスメソッドが FTP の場合、ファイル取得リクエストに応答してサーバがファイルの長さを返したときには (これは現在では普通になりましたが) Content-Length ヘッダがメタ情報に含められます。Content-type ヘッダは MIME タイプが推測可能なときにメタ情報に含められます。アクセスメソッドがローカルファイルの場合、返されるヘッダ情報にはファイルの最終更新日時を表す Date エントリ、ファイルのサイズを示す Content-Length エントリ、そして推測されるファイル形式の Content-Type エントリが含まれます。 `mimertools` モジュールを参照してください。

`geturl()` メソッドはページの実際の URL を返します。場合によっては、HTTP サーバはクライアントの要求を他の URL に振り向け (redirect、リダイレクト) します。関数 `urlopen()` はユーザに対してリダイレクトを透過的に行いますが、呼び出し側にとってクライアントがどの URL にリダイレクトされたかを知りたいときがあります。 `geturl()` メソッドを使うと、このリダイレクトされた URL を取得できます。

`getcode()` メソッドは、レスポンスと共に送られてきた HTTP ステータスコードを返します。URL が HTTP URL でなかった場合は、`None` を返します。

`url` に `http:` スキーム識別子を使う場合、`data` 引数を与えて POST 形式のリクエストを行うことができます (通常リクエストの形式は GET です)。引数 `data` は標準の `application/x-www-form-urlencoded` 形式でなければなりません; 以下の `urlencode()` 関数を参照してください。

`urlopen()` 関数は認証を必要としないプロキシ (proxy) に対して透過的に動作します。Unix または Windows 環境では、Python を起動する前に、環境変数 `http_proxy`, `ftp_proxy` にそれぞれのプロキシサーバを指定する URL を設定してください。例えば ('%' はコマンドプロンプトです):

```
% http_proxy="http://www.someproxy.com:3128"
% export http_proxy
% python
...
```

`no_proxy` 環境変数は、`proxy` を利用せずにアクセスすべきホストを指定するために利用されます。設定する場合は、カンマ区切りの、ホストネーム `suffix` のリストで、オプションとして `:port` を付けることができます。例えば、`cern.ch,ncsa.uiuc.edu,some.host:8080`.

Windows 環境では、プロキシを指定する環境変数が設定されていない場合、プロキシの設定値はレジストリの Internet Settings セクションから取得されます。

Mac OS X では、`urlopen()` はプロキシの情報をシステム設定フレームワーク (Mac OS X System Configuration Framework) から取得します。これはシステム環境設定のネットワークパネルから設定できます。

別の方法として、オプション引数 `proxies` を使って明示的にプロキシを設定することができます。この引数はスキーム名をプロキシの URL にマップする辞書型のオブジェクトでなくてはなりません。空の辞書を指定するとプロキシを使いません。None (デフォルトの値です) を指定すると、上で述べたように環境変数で指定されたプロキシ設定を使います。例えば:

```
# Use http://www.someproxy.com:3128 for HTTP proxying
proxies = {'http': 'http://www.someproxy.com:3128'}
filehandle = urllib.urlopen(some_url, proxies=proxies)
# Don't use any proxies
filehandle = urllib.urlopen(some_url, proxies={})
# Use proxies from environment - both versions are equivalent
filehandle = urllib.urlopen(some_url, proxies=None)
filehandle = urllib.urlopen(some_url)
```

認証を必要とするプロキシは現在のところサポートされていません。これは実装上の制限 (implementation limitation) と考えています。

`context` パラメータには `ssl.SSLContext` インスタンスをセットします。これは `urlopen()` が HTTPS 接続をするのに使う SSL 設定を構成します。

バージョン 2.3 で変更: `proxies` のサポートを追加しました。

バージョン 2.6 で変更: 結果オブジェクトに `getcode()` を追加し、`no_proxy` 環境変数に対応しました。

バージョン 2.7.9 で変更: The `context` parameter was added. All the necessary certificate and hostname checks are done by default.

バージョン 2.6 で非推奨: `urlopen()` 関数は Python 3 では `urllib2.urlopen()` を採用のため撤廃されています。


```
urllib.urlretrieve(url[, filename[, reporthook[, data[, context]]]])
```

URL で表されるネットワーク上のオブジェクトを、必要に応じてローカルなファイルにコピーします。URL がローカルなファイルを指定していたり、オブジェクトのコピーが正しくキャッシュされている場合、そのオブジェクトはコピーされません。タプル (filename, headers) を返し、filename はローカルで見つかったオブジェクトに対するファイル名で、headers は `urlopen()` が返した (おそらくキャッシュされているリモートの) オブジェクトに `info()` を適用して得られるものになります。`urlopen()` と同じ例外を送出します。

The second argument, if present, specifies the file location to copy to (if absent, the location will be a tempfile with a generated name). The third argument, if present, is a callable that will be called once on establishment of the network connection and once after each block read thereafter. The callable will be passed three arguments; a count of blocks transferred so far, a block size in bytes, and the total size of the file. The third argument may be `-1` on older FTP servers which do not return a file size in response to a retrieval request.

`url` が `http`: スキーム識別子を使っていた場合、オプション引数 `data` を与えることで POST リクエストを行うよう指定することができます (通常リクエストの形式は GET です)。`data` 引数は標準の `application/x-www-form-urlencoded` 形式でなくてはなりません; 以下の `urlencode()` 関数を参照してください。

The `context` parameter may be set to a `ssl.SSLContext` instance to configure the SSL settings that are used if `urlretrieve()` makes a HTTPS connection.

バージョン 2.5 で変更: `urlretrieve()` は、予想 (これは `Content-Length` ヘッダにより通知されるサイズです) よりも取得できるデータ量が少ないことを検知した場合、`ContentTooShortError` を発生します。これは、例えば、ダウンロードが中断された場合などに発生します。

`Content-Length` は下限として扱われます: より多いデータがある場合、`urlretrieve()` はそのデータを読みますが、より少ないデータしか取得できない場合、これは exception を発生します。

このような場合にもダウンロードされたデータを取得することは可能で、これは exception インスタンスの `content` 属性に保存されています。

`Content-Length` ヘッダが無い場合、`urlretrieve()` はダウンロードされたデータのサイズをチェックできず、単にそれを返します。この場合は、ダウンロードは成功したと見なす必要があります。

バージョン 2.7.9 で変更: The `context` parameter was added. All the necessary certificate and hostname checks are done by default.

```
urllib.url opener
```

パブリック関数 `urlopen()` および `urlretrieve()` は `FancyURLopener` クラスのインスタンスを生成します。インスタンスは要求された動作に応じて使用されます。この機能をオーバーライドするために、プログラマは `URLopener` または `FancyURLopener` のサブクラスを作り、そのクラスから生成したインスタンスを変数 `urllib.url opener` に代入した後、呼び出したい関数を呼ぶことができます。例えば、アプリケーションが `URLopener` が定義しているのとは異なった `User-Agent` ヘッダを指定したい場合があるかもしれません。この機能は以下のコードで実現できます:


```
import urllib

class AppURLopener(urllib.FancyURLopener):
    version = "App/1.7"

urllib._urlopener = AppURLopener()
```

`urllib.urlcleanup()`

以前の `urlretrieve()` で生成された可能性のあるキャッシュを消去します。

20.5.2 ユーティリティー関数

`urllib.quote(string[, safe])`

`string` に含まれる特殊文字を `%xx` エスケープで置換 (quote) します。アルファベット、数字、および文字 `'_.-'` には対しては quote 処理を行いません。この関数はデフォルトでは URL の path セクションに対するクォートを想定しています。オプションのパラメタ `safe` は quote 処理しない追加の文字を指定します — デフォルトの値は `'/'` です。

例: `quote('/~connolly/')` は `'/%7econnolly/'` になります。

`urllib.quote_plus(string[, safe])`

`quote()` と似ていますが、加えて空白文字をプラス記号 (`'+'`) に置き換えます。これは HTML フォームの値を URL に付加するクエリ文字列にする際に必要な機能です。もとの文字列におけるプラス記号は `safe` に含まれていない限りエスケープ置換されます。上と同様に、`safe` のデフォルトの値は `'/'` です。

`urllib.unquote(string)`

`%xx` エスケープをエスケープが表す 1 文字に置き換えます。

例: `unquote('/%7Econnolly/')` は `'/~connolly/'` になります。

`urllib.unquote_plus(string)`

`unquote()` と似ていますが、加えてプラス記号を空白文字に置き換えます。これは quote 処理された HTML フォームの値を元に戻すのに必要な機能です。

`urllib.urlencode(query[, doseq])`

マップ型オブジェクト、または 2 要素のタプルからなるシーケンスを、”パーセントエンコードされた (percent-encoded)” 文字列に変換して、上述の `urlopen()` のオプション引数 `data` に適した形式にします。この関数はフォームのフィールド値でできた辞書を POST 型のリクエストに渡すときに便利です。返される文字列は `key=value` のペアを `'&'` で区切ったシーケンスで、`key` と `value` の双方は上の `quote_plus()` でクォートされます。2 つの要素をもったタプルからなるシーケンスが引数 `query` として使われた場合、各タプルの最初の値が `key` で、2 番目の値が `value` になります。どちらのケースでも `value` にはシーケンスを入れることができ、その場合オプションのパラメタ `doseq` の評価結果が `True` であったなら、その `key` の各々の `value` に対して `'&'` で区切られた `key=value` のペアが生成されます。このときエンコードされた文字列中のパラメタの順番はシーケンス中のタプルの順番と同じになります。`urlparse` モジュールでは、関数 `parse_qs()` および `parse_qsl()` を提供しており、クエリ文字列を解析して Python のデータ構造にするのに利用できます。

`urllib.pathname2url (path)`

ローカルシステムにおける記法で表されたパス名 *path* を、URL におけるパス部分の形式に変換します。この関数は完全な URL を生成するわけではありません。返される値は常に `quote()` を使って `quote` 処理されたものになります。

`urllib.url2pathname (path)`

URL のパスの部分 *path* をパーセントエンコードされた URL の形式からローカルシステムにおけるパス記法に変換します。この関数は *path* をデコードするために `unquote()` を使います。

`urllib.getproxies ()`

このヘルパー関数はスキーマからプロキシサーバーの URL へのマッピングを行う辞書を返します。この関数はまず、どの OS でも最初に `<scheme>.proxy` という名前の環境変数を大文字小文字を区別せずにスキャンします。そこで見つからなかった場合、Max OS X の場合は Mac OS X システム環境設定を、Windows の場合はシステムレジストリを参照します。もし小文字と大文字の環境変数が両方存在する (そして値が一致しない) なら、小文字の環境変数が優先されます。

注釈: もし環境変数 `REQUEST_METHOD` が設定されていたら (これは通常スクリプトが CGI 環境で動いていることを示しています)、環境変数 `HTTP_PROXY` (大文字の `_PROXY`) は無視されます。その理由は、クライアントが "Proxy:" HTTP ヘッダーを使ってこの環境変数を注入できるからです。もし CGI 環境で HTTP プロキシを使う必要があれば、`ProxyHandler` を明示的に使用するか、環境変数名を小文字にしてください (あるいは、少なくともサフィックスを `_proxy` にしてください)。

注釈: `urllib` はある種のユーティリティ関数、`splitttype`, `splithost` やその他 URL をパースしてコンポーネントに分解するようなものを剥き出しにしておりますが、これらを使うのではなくて、URL のパースには `urlparse` を使ってください。Python 3 では `urllib.parse` でこれらヘルパ関数は曝していません。

20.5.3 URL Opener オブジェクト

`class urllib.URLOpener ([proxies[, context[, **x509]]])`

URL をオープンし、読み出すためのクラスの基底クラスです。http:, ftp:, file: 以外のスキームを使ったオブジェクトのオープンをサポートしたいのでないかぎり、`FancyURLOpener` を使おうと思うことになるでしょう。

デフォルトでは、`URLOpener` クラスは `User-Agent` ヘッダとして `urllib/VVV` を送信します。ここで VVV は `urllib` のバージョン番号です。アプリケーションで独自の `User-Agent` ヘッダを送信したい場合は、`URLOpener` かまたは `FancyURLOpener` のサブクラスを作成し、サブクラス定義においてクラス属性 `version` を適切な文字列値に設定することで行うことができます。

オプションのパラメーター `proxies` はスキーム名をプロキシの URL にマップする辞書でなければなりません。空の辞書はプロキシ機能を完全にオフにします。デフォルトの値は `None` で、この場合、`urlopen()` の定義で述べたように、プロキシを設定する環境変数が存在するならそれを使います。

`context` パラメータは指定する場合 `ssl.SSLContext` インスタンスです。与えられた場合、これは URL をオープンするオブジェクトが HTTPS 接続をするのに使う SSL 設定を定義します。

追加のキーワードパラメーターは `x509` に集められますが、これは `https:` スキームを使った際のクライアント認証に使われることがあります。キーワード引数 `key_file` および `cert_file` が SSL 鍵と証明書を設定するためにサポートされています; クライアント認証をするには両方が必要です。

`URLopener` オブジェクトは、サーバがエラーコードを返した時には `IOError` を発生します。

バージョン 2.7.9 で変更: The `context` parameter was added. All the necessary certificate and hostname checks are done by default.

open (*fullurl* [, *data*])

適切なプロトコルを使って *fullurl* を開きます。このメソッドはキャッシュとプロキシ情報を設定し、その後適切な open メソッドを入力引数つきで呼び出します。認識できないスキームが与えられた場合、`open_unknown()` が呼び出されます。 *data* 引数は `urlopen()` の引数 *data* と同じ意味を持っています。

open_unknown (*fullurl* [, *data*])

オーバーライド可能な、未知のタイプの URL を開くためのインタフェースです。

retrieve (*url* [, *filename* [, *reporthook* [, *data*]]])

url のコンテンツを取得し、*filename* に書き込みます。返り値はタプルで、ローカルシステムにおけるファイル名と、応答ヘッダを含む `mimertools.Message` オブジェクト (URL がリモートを指している場合)、または `None` (URL がローカルを指している場合) からなります。呼び出し側の処理はその後 *filename* を開いて内容を読み出さなくてはなりません。 *filename* が与えられており、かつ URL がローカルシステム上のファイルを示している場合、入力ファイル名が返されます。URL がローカルのファイルを示しておらず、かつ *filename* が与えられていない場合、ファイル名は入力 URL の最後のパス構成要素につけられた拡張子と同じ拡張子を `tempfile.mktemp()` につけたものになります。 *reporthook* を与える場合、この変数は 3 つの数値パラメータを受け取る関数でなくてはなりません。この関数はデータの塊 (chunk) がネットワークから読み込まれるたびに呼び出されます。ローカルの URL を与えた場合 *reporthook* は無視されます。

url が `http:` スキーム識別子を使っていた場合、オプション引数 *data* を与えることで POST リクエストを行うよう指定することができます (通常リクエストの形式は GET です)。 *data* 引数は標準の `application/x-www-form-urlencoded` 形式でなくてはなりません; 以下の `urlencode()` 関数を参照してください。

version

URL をオープンするオブジェクトのユーザエージェントを指定する変数です。 `urllib` を特定のユーザエージェントであるとサーバに通知するには、サブクラスの中でこの値をクラス変数として値を設定するか、コンストラクタの中でベースクラスを呼び出す前に値を設定してください。

class `urllib.FancyURLopener` (...)

`FancyURLopener` は `URLopener` のサブクラスで、以下の HTTP レスポンスコード: 301、302、303、307、および 401 を取り扱う機能を提供します。レスポンスコード 30x に対しては、`Location` ヘッダを使って実際の URL を取得します。レスポンスコード 401 (認証が要求されていることを示す) に対しては、BASIC 認証 (basic HTTP authentication) が行われます。レスポンスコード 30x に対しては、最

大で *maxtries* 属性に指定された数だけ再帰呼び出しを行うようになっています。この値はデフォルトで 10 です。

その他のレスポンスコードについては、`http_error_default()` が呼ばれます。これはサブクラスでエラーを適切に処理するようにオーバーライドすることができます。

注釈: **RFC 2616** によると、POST 要求に対する 301 および 302 応答はユーザの承認無しに自動的にリダイレクトしてはなりません。実際は、これらの応答に対して自動リダイレクトを許すブラウザでは POST を GET に変更しており、*urllib* でもこの動作を再現します。

コンストラクタに与えるパラメーターは *URLopener* と同じです。

注釈: 基本的な HTTP 認証を行う際、*FancyURLopener* インスタンスは `prompt_user_passwd()` メソッドを呼び出します。このメソッドはデフォルトでは実行を制御している端末上で認証に必要な情報を要求するように実装されています。必要ならば、このクラスのサブクラスにおいてより適切な動作をサポートするために `prompt_user_passwd()` メソッドをオーバーライドしてもかまいません。

FancyURLopener クラスはオーバーライド可能な追加のメソッドを提供しており、適切な振る舞いをさせることができます:

prompt_user_passwd (*host*, *realm*)

指定されたセキュリティ領域 (security realm) 下にある与えられたホストにおいて、ユーザー認証に必要な情報を返すための関数です。この関数が返す値は (*user*, *password*) からなるタプルでなければなりません。値は Basic 認証で使われます。

このクラスでの実装では、端末に情報を入力するようプロンプトを出します; ローカルの環境において適切な形で対話型モデルを使うには、このメソッドをオーバーライドしなければなりません。

exception `urllib.ContentTooShortError` (*msg*[, *content*])

この例外は `urlretrieve()` 関数が、ダウンロードされたデータの量が予期した量 (*Content-Length* ヘッダで与えられる) よりも少ないことを検知した際に発生します。 *content* 属性には (恐らく途中までの) ダウンロードされたデータが格納されています。

バージョン 2.5 で追加.

20.5.4 urllib の制限

- 現在のところ、以下のプロトコルだけがサポートされています: HTTP、(バージョン 0.9 および 1.0)、FTP、およびローカルファイル。
- `urlretrieve()` のキャッシュ機能は、有効期限ヘッダ (Expiration time header) を正しく処理できるようにハックするための時間を取れるまで、無効にしています。

- ある URL がキャッシュにあるかどうか調べるような関数があればと思っています。
- 後方互換性のため、URL がローカルシステム上のファイルを指しているように見えるにも関わらずファイルを開くことができれば、URL は FTP プロトコルを使って再解釈されます。この機能は時として混乱を招くエラーメッセージを引き起こします。
- 関数 `urlopen()` および `urlretrieve()` は、ネットワーク接続が確立されるまでの間、一定でない長さの遅延を引き起こすことがあります。このことは、これらの関数を使ってインタラクティブな Web クライアントを構築するのはスレッドなしには難しいことを意味します。
- `urlopen()` または `urlretrieve()` が返すデータはサーバが返す生のデータです。このデータはバイナリデータ (画像データ等)、生テキスト (plain text)、または (例えば) HTML でもかまいません。HTTP プロトコルはリプライヘッダ (reply header) にデータのタイプに関する情報を返します。タイプは `Content-Type` ヘッダを見ることで推測できます。返されたデータが HTML であれば、`htmllib` を使ってパースすることができます。
- FTP プロトコルを扱うコードでは、ファイルとディレクトリを区別できません。このことから、アクセスできないファイルを指している URL からデータを読み出そうとすると、予期しない動作を引き起こす場合があります。URL が / で終わっていれば、ディレクトリを指しているものとみなして、それに適した処理を行います。しかし、ファイルの読み出し操作が 550 エラー (URL が存在しないか、主にパーミッションの理由でアクセスできない) になった場合、URL がディレクトリを指していて、末尾の / を忘れたケースを処理するため、パスをディレクトリとして扱います。このために、パーミッションのためにアクセスできないファイルを fetch しようとする、FTP コードはそのファイルを開こうとして 550 エラーに陥り、次にディレクトリ一覧を表示しようとするため、誤解を生むような結果を引き起こす可能性があるのです。よく調整された制御が必要なら、`ftplib` モジュールを使うか、`FancyURLopener` をサブクラス化するか、`urloper` を変更して目的に合わせるよう検討してください。
- このモジュールは認証を必要とするプロキシをサポートしません。将来実装されるかもしれません。
- `urllib` モジュールは URL 文字列を解釈したり構築したりする (ドキュメント化されていない) ルーチンを含んでいますが、URL を操作するためのインタフェースとしては、`urlparse` モジュールをお勧めします。

20.5.5 使用例

以下は GET メソッドを使ってパラメータを含む URL を取得するセッションの例です:

```
>>> import urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> f = urllib.urlopen("http://www.musi-cal.com/cgi-bin/query?%s" % params)
>>> print f.read()
```

以下は POST メソッドを代わりに使った例です:

```
>>> import urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
```

(次のページに続く)

(前のページからの続き)

```
>>> f = urllib.urlopen("http://www.musi-cal.com/cgi-bin/query", params)
>>> print f.read()
```

以下の例では、環境変数による設定内容に対して上書きする形で HTTP プロキシを明示的に設定しています:

```
>>> import urllib
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.FancyURLopener(proxies)
>>> f = opener.open("http://www.python.org")
>>> f.read()
```

以下の例では、環境変数による設定内容に対して上書きする形で、まったくプロキシを使わないよう設定しています:

```
>>> import urllib
>>> opener = urllib.FancyURLopener({})
>>> f = opener.open("http://www.python.org/")
>>> f.read()
```

20.6 urllib2 — URL を開くための拡張可能なライブラリ

注釈: `urllib2` モジュールは、Python 3 で `urllib.request`, `urllib.error` に分割されました。 *2to3* ツールが自動的にソースコードの `import` を修正します。

`urllib2` モジュールは基本的な認証、暗号化認証、リダイレクション、クッキー、その他の介在する複雑なアクセス環境において (大抵は HTTP で) URL を開くための関数とクラスを定義します。

参考:

より高水準の HTTP クライアントインターフェイスとしては [Requests package](#) がお奨めです。

`urllib2` モジュールでは以下の関数を定義しています:

```
urllib2.urlopen(url[, data[, timeout[, cafile[, capath[, cadefault[, context]]]])
```

URL `url` を開きます。 `url` は文字列でも `Request` オブジェクトでもかまいません。

`data` はサーバに送信する追加のデータを示す文字列か、そのようなデータが無ければ `None` を指定します。現時点で HTTP リクエストは `data` をサポートする唯一のリクエスト形式です; `data` パラメタが指定が指定された場合、HTTP リクエストは GET でなく POST になります。 `data` は標準的な `application/x-www-form-urlencoded` 形式のバッファでなくてはなりません。 `urllib.urlencode()` 関数はマップ型か 2 タブルのシーケンスを取り、この形式の文字列を返します。 `urllib2` モジュールは HTTP/1.1 リクエストを `Connection:close` ヘッダ付きで送信します。

オプションの `timeout` 引数は、接続開始などのブロックする操作におけるタイムアウト時間を秒数で指定します。(指定されなかった場合、グローバルのデフォルトタイムアウト時間が利用されます) この

引数は、HTTP, HTTPS, FTP 接続でのみ有効です。

`context` を指定する場合、`ssl.SSLContext` インスタンスでなければなりません。それはさまざまな SSL オプションを記述しています。詳細は `HTTPSConnection` を参照してください。

任意のパラメーター `cafile` および `capath` には HTTPS リクエストのための CA 証明書のセットを指定します。`cafile` には CA 証明書のリストを含む 1 個のファイルを指定し、`capath` にはハッシュ化された証明書ファイルが格納されたディレクトリを指定しなければなりません。より詳しい情報は `ssl.SSLContext.load_verify_locations()` を参照してください。

`cadefault` パラメータは無視されます。

この関数は以下の 3 つのメソッドを持つファイル類似のオブジェクトを返します:

- `geturl()` — 取得されたリソースの URL を返します。主に、リダイレクトが発生したかどうかを確認するために利用します。
- `info()` — 取得されたページのヘッダーなどのメタ情報を、`mimetools.Message` インスタンスとして返します。(Quick Reference to HTTP Headers を参照してください)
- `getcode()` — レスポンスの HTTP ステータスコードを返します。

エラーが発生した場合 `URLError` を送出します。

どのハンドラもリクエストを処理しなかった場合には `None` を返すことがあるので注意してください (デフォルトでインストールされる グローバルハンドラの `OpenerDirector` は、`UnknownHandler` を使って上記の問題が起きないようにしています)。

さらに、プロキシ設定が検出された場合 (例えば `http-proxy` のような `*_proxy` 環境変数がセットされているなど) には `ProxyHandler` がデフォルトでインストールされ、これがプロキシを通してリクエストを処理するようにしています。

バージョン 2.6 で変更: `timeout` 引数が追加されました。

バージョン 2.7.9 で変更: `cafile`, `capath`, `cadefault`, `context` が追加されました。

`urllib2.install_opener(opener)`

指定された `OpenerDirector` のインスタンスを、デフォルトで利用されるグローバルの opener としてインストールします。opener のインストールは、`urlopen` にその opener を使って欲しいとき以外必要ありません。普段は単に `urlopen()` の代わりに `OpenerDirector.open()` を利用してください。この関数は引数が本当に `OpenerDirector` のインスタンスであるかどうかはチェックしません。適切なインタフェースを持った任意のクラスを利用することができます。

`urllib2.build_opener([handler, ...])`

与えられた順番に URL ハンドラを連鎖させる `OpenerDirector` のインスタンスを返します。`handler` は `BaseHandler` または `BaseHandler` のサブクラスのインスタンスのどちらかです (どちらの場合も、コンストラクトは引数無しで呼び出せるようになっていなければなりません)。クラス `ProxyHandler` (proxy 設定が検出された場合)、`UnknownHandler`, `HTTPHandler`, `HTTPDefaultErrorHandler`, `HTTPRedirectHandler`, `FTPHandler`, `FileHandler`, `HTTPErrorProcessor` については、そのクラスのインスタンスか、そのサブクラスのインスタンスが `handler` に含まれていない限り、`handler` よりも先に連鎖します。

Python が SSL をサポートするように設定してインストールされている場合 (すなわち、`ssl` モジュールを `import` できる場合) `HTTPSHandler` も追加されます。

Python 2.3 からは、`BaseHandler` サブクラスでも `handler_order` メンバ変数を変更して、ハンドラリスト内での場所を変更できるようになりました。

状況に応じて、以下の例外が送出されます:

exception `urllib2.URLError`

ハンドラが何らかの問題に遭遇した場合、この例外 (またはこの例外から派生した例外) を送出します。この例外は `IOError` のサブクラスです。

reason

このエラーの原因。メッセージ文字列か、他の例外のインスタンス (リモート URL の場合は `socket.error`, ローカル URL の場合は `OSError`)。

exception `urllib2.HTTPError`

これは例外 (`URLError` のサブクラス) ですが、このオブジェクトは例外でないファイル類似のオブジェクトとして返り値に使うことができます (`urlopen()` が返すのと同じものです)。この機能は、例えばサーバからの認証リクエストのように、変わった HTTP エラーを処理するのに役立ちます。

code

RFC 2616 に定義されている HTTP ステータスコード。この数値型の値は、`BaseHTTPServer.BaseHTTPRequestHandler.responses` の辞書に登録されているコードに対応します。

reason

このエラーの理由。メッセージ文字列あるいは他の例外インスタンスです。

以下のクラスが提供されています:

class `urllib2.Request` (`url`[, `data`][, `headers`][, `origin_req_host`][, `unverifiable`])

このクラスは URL リクエストを抽象化したものです。

`url` は有効な URL を指す文字列でなくてはなりません。

`data` はサーバに送信する追加のデータを示す文字列か、そのようなデータが無ければ `None` を指定します。現時点で HTTP リクエストは `data` をサポートする唯一のリクエスト形式です; `data` パラメタが指定が指定された場合、HTTP リクエストは GET でなく POST になります。 `data` は標準的な `application/x-www-form-urlencoded` 形式のバッファでなくてはなりません。 `urllib.urlencode()` 関数はマップ型か 2 タプルシーケンスを取り、この形式の文字列を返します。

`headers` は辞書である必要があり、辞書のそれぞれのキーと値を引数として `add_header()` を呼び出したのと同じように扱われます。この引数は、多くの場合ブラウザが何であるかを特定する User-Agent ヘッダーの値を "偽装" するために用いられます。これは一部の HTTP サーバーが、スクリプトからのアクセスを禁止するために一般的なブラウザの User-Agent ヘッダーしか許可しないためです。例えば、Mozilla Firefox は User-Agent に "Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11" のように設定し、`urllib2` はデフォルトで "Python-urllib/2.6" (Python 2.6 の場合) と設定します。

最後の二つの引数は、サードパーティの HTTP クッキーを正しく扱いたい場合にのみ関係してきます:

`origin_req_host` は、[RFC 2965](#) で定義されている元のトランザクションにおけるリクエストホスト (request-host of the origin transaction) です。デフォルトの値は `cookielib.request_host(self)` です。この値は、ユーザによって開始された元々のリクエストにおけるホスト名や IP アドレスです。例えば、もしリクエストがある HTML ドキュメント内の画像を指していれば、この値は画像を含んでいるページへのリクエストにおけるリクエストホストになるはずです。

`unverifiable` は、[RFC 2965](#) の定義において、該当するリクエストが証明不能 (unverifiable) であるかどうかを示します。デフォルトの値は `False` です。証明不能なリクエストとは、ユーザが受け入れの可否を選択できないような URL を持つリクエストのことです。例えば、リクエストが HTML ドキュメント中の画像であり、ユーザがこの画像を自動的に取得するかどうかを選択できない場合には、証明不能フラグは `True` になります。

class `urllib2.OpenerDirector`

`OpenerDirector` クラスは、`BaseHandler` の連鎖的に呼び出して URL を開きます。このクラスはハンドラをどのように連鎖させるか、またどのようにエラーをリカバリするかを管理します。

class `urllib2.BaseHandler`

このクラスはハンドラ連鎖に登録される全てのハンドラがベースとしているクラスです – このクラスでは登録のための単純なメカニズムだけを扱います。

class `urllib2.HTTPDefaultErrorHandler`

HTTP エラー応答のための標準のハンドラを定義します; 全てのレスポンスに対して、例外 `HTTPError` を送出します。

class `urllib2.HTTPRedirectHandler`

リダイレクションを扱うクラスです。

class `urllib2.HTTPCookieProcessor` (`[cookiejar]`)

HTTP Cookie を扱うためのクラスです。

class `urllib2.ProxyHandler` (`[proxies]`)

このクラスはプロキシを通過してリクエストを送らせます。引数 `proxies` を与える場合、プロトコル名からプロキシの URL へ対応付ける辞書でなくてはなりません。標準では、プロキシのリストを環境変数 `<protocol>-proxy` から読み出します。プロキシ環境変数が設定されていない場合は、Windows 環境では、レジストリのインターネット設定セクションからプロキシ設定を手に入れ、Mac OS X 環境では、OS X システム設定フレームワーク (System Configuration Framework) からプロキシ情報を取得します。

自動検出された proxy を無効にするには、空の辞書を渡してください。

注釈: 変数 `REQUEST_METHOD` が設定されている場合、`HTTP_PROXY` は無視されます; `getproxies()` のドキュメンテーションを参照してください。

class `urllib2.HTTPPasswordMgr`

(`realm, uri`) -> (`user, password`) の対応付けデータベースを保持します。

class `urllib2.HTTPPasswordMgrWithDefaultRealm`

(realm, uri) -> (user, password) の対応付けデータベースを保持します。レルム None はその他諸々のレルムを表し、他のレルムが該当しない場合に検索されます。

class urllib2.**AbstractBasicAuthHandler** ([password_mgr])

このクラスは HTTP 認証を補助するための混ぜ込みクラス (mixin class) です。遠隔ホストとプロキシの両方に対応しています。password_mgr を与える場合、[HTTPPasswordMgr](#) と互換性がなければなりません; 互換性のためにサポートしなければならないインタフェースについての情報はセクション [HTTPPasswordMgr オブジェクト](#) を参照してください。

class urllib2.**HTTPBasicAuthHandler** ([password_mgr])

遠隔ホストとの間での認証を扱います。password_mgr を与える場合、[HTTPPasswordMgr](#) と互換性がなければなりません; 互換性のためにサポートしなければならないインタフェースについての情報はセクション [HTTPPasswordMgr オブジェクト](#) を参照してください。

class urllib2.**ProxyBasicAuthHandler** ([password_mgr])

プロキシとの間での認証を扱います。password_mgr を与える場合、[HTTPPasswordMgr](#) と互換性がなければなりません; 互換性のためにサポートしなければならないインタフェースについての情報はセクション [HTTPPasswordMgr オブジェクト](#) を参照してください。

class urllib2.**AbstractDigestAuthHandler** ([password_mgr])

このクラスは HTTP 認証を補助するための混ぜ込みクラス (mixin class) です。遠隔ホストとプロキシの両方に対応しています。password_mgr を与える場合、[HTTPPasswordMgr](#) と互換性がなければなりません; 互換性のためにサポートしなければならないインタフェースについての情報はセクション [HTTPPasswordMgr オブジェクト](#) を参照してください。

class urllib2.**HTTPDigestAuthHandler** ([password_mgr])

遠隔ホストとの間での認証を扱います。password_mgr を与える場合、[HTTPPasswordMgr](#) と互換性がなければなりません; 互換性のためにサポートしなければならないインタフェースについての情報はセクション [HTTPPasswordMgr オブジェクト](#) を参照してください。

class urllib2.**ProxyDigestAuthHandler** ([password_mgr])

プロキシとの間での認証を扱います。password_mgr を与える場合、[HTTPPasswordMgr](#) と互換性がなければなりません; 互換性のためにサポートしなければならないインタフェースについての情報はセクション [HTTPPasswordMgr オブジェクト](#) を参照してください。

class urllib2.**HTTPHandler**

HTTP の URL を開きます。

class urllib2.**HTTPSHandler** ([debuglevel[, context]])

HTTPS の URL のオープンを処理するクラスです。context は [httplib.HTTPSConnection](#) のものと同じです。

バージョン 2.7.9 で変更: context が追加されました。

class urllib2.**FileHandler**

ローカルファイルを開きます。

class urllib2.**FTPHandler**

FTP の URL を開きます。

class urllib2.CacheFTPHandler

FTP の URL を開きます。遅延を最小限にするために、開かれている FTP 接続に対するキャッシュを保持します。

class urllib2.UnknownHandler

その他諸々のためのクラスで、未知のプロトコルの URL を開きます。

class urllib2.HTTPErrorProcessor

HTTP エラー応答の処理をします。

20.6.1 Request オブジェクト

以下のメソッドは *Request* の全ての公開インタフェースを記述します。従ってサブクラスではこれら全てのメソッドをオーバーライドしなければなりません。

Request.add_data(data)

Request のデータを *data* に設定します。この値は HTTP ハンドラ以外のハンドラでは無視されます。HTTP ハンドラでは、データはバイト文字列でなくてはなりません。このメソッドを使うとリクエストの形式が GET から POST に変更されます。

Request.get_method()

HTTP リクエストメソッドを示す文字列を返します。このメソッドは HTTP リクエストだけに対して意味があり、現状では常に 'GET' か 'POST' のいずれかの値を返します。

Request.has_data()

インスタンスが None でないデータを持つかどうかを返します。

Request.get_data()

インスタンスのデータを返します。

Request.add_header(key, val)

リクエストに新たなヘッダを追加します。ヘッダは HTTP ハンドラ以外のハンドラでは無視されます。HTTP ハンドラでは、引数はサーバに送信されるヘッダのリストに追加されます。同じ名前を持つヘッダを 2 つ以上持つことはできず、*key* の衝突が生じた場合、後で追加したヘッダが前に追加したヘッダを上書きします。現時点では、この機能は HTTP の機能を損ねることはありません。というのは、複数呼び出したときに意味を持つようなヘッダには、どれもただ一つのヘッダを使って同じ機能を果たすための(ヘッダ特有の)方法があるからです。

Request.add_unredirected_header(key, header)

リダイレクトされたリクエストには追加されないヘッダを追加します。

バージョン 2.4 で追加.

Request.has_header(header)

インスタンスが名前つきヘッダであるかどうかを(通常のヘッダと非リダイレクトヘッダの両方を調べて)返します。

バージョン 2.4 で追加.

`Request.get_full_url()`

コンストラクタで与えられた URL を返します。

`Request.get_type()`

URL のタイプ — いわゆるスキーム (scheme) — を返します。

`Request.get_host()`

接続を行う先のホスト名を返します。

`Request.get_selector()`

セレクタ — サーバに送られる URL の一部分 — を返します。

`Request.get_header(header_name, default=None)`

指定されたヘッダの値を返します。ヘッダがない場合は、*default* の値を返します。

`Request.header_items()`

リクエストヘッダの値を、タプル (header_name, header_value) のリストで返します。

`Request.set_proxy(host, type)`

リクエストがプロキシサーバを経由するように準備します。*host* および *type* はインスタンスのもとの設定と置き換えられます。インスタンスのセレクタはコンストラクタに与えたもともとの URL になります。

`Request.get_origin_req_host()`

RFC 2965 の定義による、始原トランザクションのリクエストホストを返します。*Request* コンストラクタのドキュメントを参照してください。

`Request.is_unverifiable()`

リクエストが **RFC 2965** の定義における証明不能リクエストであるかどうかを返します。*Request* コンストラクタのドキュメントを参照してください。

20.6.2 OpenerDirector オブジェクト

OpenerDirector インスタンスは以下のメソッドを持っています:

`OpenerDirector.add_handler(handler)`

handler は *BaseHandler* のインスタンスでなければなりません。以下のメソッドを使った検索が行われ、URL を取り扱うことが可能なハンドラの連鎖が追加されます (HTTP エラーは特別扱いされているので注意してください)。

- *protocol_open* — ハンドラが *protocol* の URL を開く方法を知っているかどうかを調べます。
- *http_error_type* — ハンドラが HTTP エラーコード *type* の処理方法を知っていることを示すシグナルです。
- *protocol_error* — ハンドラが (http でない) *protocol* のエラー を処理する方法を知っていることを示すシグナルです。

- `protocol_request` — ハンドラが `protocol` リクエストのプリプロセス方法 を知っていることを示すシグナルです。
- `protocol_response` — ハンドラが `protocol` リクエストのポストプロセス方法 を知っていることを示すシグナルです。

`OpenerDirector.open(url[, data][, timeout])`

与えられた `url` (リクエストオブジェクトでも文字列でもかまいません) を開きます。オプションとして `data` を与えることができます。引数、戻り値、および送出される例外は `urlopen()` と同じです (`urlopen()` の場合、標準でインストールされているグローバルな `OpenerDirector` の `open()` メソッドを呼び出します)。オプションの `timeout` 引数は、接続開始のようなブロックする処理におけるタイムアウト時間を 秒数で指定します。(指定しなかった場合は、グローバルのデフォルト設定が利用されます) タイムアウト機能は、HTTP, HTTPS, FTP 接続でのみ有効です。

バージョン 2.6 で変更: `timeout` 引数が追加されました。

`OpenerDirector.error(proto[, arg[, ...]])`

与えられたプロトコルにおけるエラーを処理します。このメソッドは与えられたプロトコルにおける登録済みのエラーハンドラを (プロトコル固有の) 引数で呼び出します。HTTP プロトコルは特殊なケースで、特定のエラーハンドラを選び出すのに HTTP レスポンスコードを使います; ハンドラクラスの `http_error_*()` メソッドを参照してください。

戻り値および送出される例外は `urlopen()` と同じものです。

`OpenerDirector` オブジェクトは、以下の 3 つのステージに分けて URL を開きます:

各ステージで `OpenerDirector` オブジェクトのメソッドがどのような順で呼び出されるかは、ハンドラインスタンスの並び方で決まります。

1. `protocol_request` 形式のメソッドを持つ全てのハンドラに対してそのメソッドを呼び出し、リクエストのプリプロセスを行います。
2. `protocol_open` 形式のメソッドを持つハンドラを呼び出し、リクエストを処理します。このステージは、ハンドラが `None` でない値 (すなわちレスポンス) を返すか、例外 (通常は `URLError`) を送出した時点で終了します。例外は伝播 (propagate) できます。

実際には、上のアルゴリズムではまず `default_open()` という名前のメソッドを呼び出します。このメソッドが全て `None` を返す場合、同じアルゴリズムを繰り返して、今度は `protocol_open` 形式のメソッドを試します。メソッドが全て `None` を返すと、さらに同じアルゴリズムを繰り返して `unknown_open()` を呼び出します。

これらのメソッドの実装には、親となる `OpenerDirector` インスタンスの `open()` や `error()` といったメソッド呼び出しが入る場合があるので注意してください。

3. `protocol_response` 形式のメソッドを持つ全てのハンドラに対してそのメソッドを呼び出し、リクエストのポストプロセスを行います。

20.6.3 BaseHandler オブジェクト

BaseHandler オブジェクトは直接的に役に立つ 2 つのメソッドと、その他として派生クラスで使われることを想定したメソッドを提供します。以下は直接的に使うためのメソッドです:

`BaseHandler.add_parent (director)`

親オブジェクトとして、`director` を追加します。

`BaseHandler.close ()`

全ての親オブジェクトを削除します。

以下の属性およびメソッドは *BaseHandler* から派生したクラスでのみ使われます:

注釈: 慣習的に、`protocol_request ()` や `protocol_response ()` といったメソッドを定義しているサブクラスは `*Processor` と名づけ、その他は `*Handler` と名づけることになっています

`BaseHandler.parent`

有効な *OpenerDirector* です。この値は違うプロトコルを使って URL を開く場合やエラーを処理する際に使われます。

`BaseHandler.default_open (req)`

このメソッドは *BaseHandler* では定義されていません。しかし、全ての URL をキャッチさせたいなら、サブクラスで定義する必要があります。

このメソッドが定義されていた場合、*OpenerDirector* から呼び出されます。このメソッドは *OpenerDirector* のメソッド `open ()` が返す値について記述されているようなファイル類似のオブジェクトか、`None` を返さなくてはなりません。このメソッドが送出する例外は、真に例外的なことが起きない限り、*URLLError* を送出しなければなりません (例えば、*MemoryError* を *URLLError* をマップしてはいけません)。

このメソッドはプロトコル固有のオープンメソッドが呼び出される前に呼び出されます。

`BaseHandler.protocol_open (req)`

(“protocol” は実際にはプロトコル名です)

このメソッドは *BaseHandler* では定義されていません。しかし *protocol* の URL をキャッチしたいなら、サブクラスで定義する必要があります。

このメソッドが定義されていた場合、*OpenerDirector* から呼び出されます。戻り値は `default_open ()` と同じでなければなりません。

`BaseHandler.unknown_open (req)`

このメソッドは *BaseHandler* では定義されていません。しかし URL を開くための特定のハンドラが登録されていないような URL をキャッチしたいなら、サブクラスで定義する必要があります。

このメソッドが定義されていた場合、*OpenerDirector* から呼び出されます。戻り値は `default_open ()` と同じでなければなりません。

`BaseHandler.http_error_default (req, fp, code, msg, hdrs)`

このメソッドは `BaseHandler` では定義されていません。しかしその他の処理されなかった HTTP エラーを処理する機能をもたせたいなら、サブクラスで定義する必要があります。このメソッドはエラーに遭遇した `OpenerDirector` から自動的に呼び出されます。その他の状況では普通呼び出すべきではありません。

`req` は `Request` オブジェクトで、`fp` は HTTP エラー本体を読み出せるようなファイル類似のオブジェクトになります。`code` は 3 桁の 10 進数からなるエラーコードで、`msg` ユーザ向けのエラーコード解説です。`hdrs` はエラー応答のヘッダをマップしたオブジェクトです。

返される値および送出される例外は `urlopen()` と同じものでなければなりません。

`BaseHandler.http_error_nnn (req, fp, code, msg, hdrs)`

`nnn` は 3 桁の 10 進数からなる HTTP エラーコードでなくてはなりません。このメソッドも `BaseHandler` では定義されていませんが、サブクラスのインスタンスで定義されていた場合、エラーコード `nnn` の HTTP エラーが発生した際に呼び出されます。

特定の HTTP エラーに対する処理を行うためには、このメソッドをサブクラスでオーバーライドする必要があります。

引数、返される値、および送出される例外は `http_error_default()` と同じものでなければなりません。

`BaseHandler.protocol_request (req)`

(“protocol” は実際にはプロトコル名です)

このメソッドは `BaseHandler` では定義されていませんが、サブクラスで特定の `protocol` のリクエストのプリプロセスを行いたい場合には定義する必要があります。

このメソッドが定義されていると、親となる `OpenerDirector` から呼び出されます。その際、`req` は `Request` オブジェクトになります。戻り値は `Request` オブジェクトでなければなりません。

`BaseHandler.protocol_response (req, response)`

(“protocol” は実際にはプロトコル名です)

このメソッドは `BaseHandler` では定義されていませんが、サブクラスで特定の `protocol` のリクエストのポストプロセスを行いたい場合には定義する必要があります。

このメソッドが定義されていると、親となる `OpenerDirector` から呼び出されます。その際、`req` は `Request` オブジェクトになります。`response` は `urlopen()` の戻り値と同じインタフェースを実装したオブジェクトになります。戻り値もまた、`urlopen()` の戻り値と同じインタフェースを実装したオブジェクトでなければなりません。

20.6.4 HTTPRedirectHandler オブジェクト

注釈: HTTP リダイレクトによっては、このモジュールのクライアントコード側での処理を必要とします。その場合、`HTTPError` が送出されます。様々なリダイレクトコードの厳密な意味に関する詳細は [RFC 2616](#)

を参照してください。

`HTTPRedirectHandler.redirect_request(req, fp, code, msg, hdrs, newurl)`

リダイレクトの通知に応じて、`Request` または `None` を返します。このメソッドは `http_error_30*()` メソッドにおいて、リダイレクトの通知をサーバから受信した際に、デフォルトの実装として呼び出されます。リダイレクトを起こす場合、新たな `Request` を生成して、`http_error_30*()` が `newurl` へリダイレクトを実行できるようにします。そうでない場合、他のどのハンドラにもこの URL を処理させたくなければ `HTTPError` を送出し、リダイレクト処理を行うことはできないが他のハンドラなら可能かもしれない場合には `None` を返します。

注釈: このメソッドのデフォルトの実装は、**RFC 2616** に厳密に従ったものではありません。**RFC 2616** では、POST リクエストに対する 301 および 302 応答が、ユーザの承認なく自動的にリダイレクトされてはならないと述べています。現実には、ブラウザは POST を GET に変更することで、これらの応答に対して自動的にリダイレクトを行えるようにしています。デフォルトの実装でも、この挙動を再現しています。

`HTTPRedirectHandler.http_error_301(req, fp, code, msg, hdrs)`

Location: か URI: の URL にリダイレクトします。このメソッドは HTTP における 'moved permanently' レスポンスを取得した際に 親オブジェクトとなる `OpenerDirector` によって呼び出されます。

`HTTPRedirectHandler.http_error_302(req, fp, code, msg, hdrs)`

`http_error_301()` と同じですが、'found' レスポンスに対して呼び出されます。

`HTTPRedirectHandler.http_error_303(req, fp, code, msg, hdrs)`

`http_error_301()` と同じですが、'see other' レスポンスに対して呼び出されます。

`HTTPRedirectHandler.http_error_307(req, fp, code, msg, hdrs)`

`http_error_301()` と同じですが、'temporary redirect' レスポンスに対して呼び出されます。

20.6.5 HTTPCookieProcessor オブジェクト

バージョン 2.4 で追加。

`HTTPCookieProcessor` インスタンスは属性をひとつだけ持ちます:

`HTTPCookieProcessor.cookiejar`

クッキーの入っている `cookielib.CookieJar` オブジェクトです。

20.6.6 ProxyHandler オブジェクト

`ProxyHandler.protocol_open(request)`

("protocol" は実際にはプロトコル名です)

`ProxyHandler` は、コンストラクタで与えた辞書 `proxies` にプロキシが設定されているような `protocol` 全てについて、メソッド `protocol.open` を持つことになります。このメソッドは `request.set_proxy()` を呼び出して、リクエストがプロキシを通過できるように修正します。その後連鎖するハンドラの中から次のハンドラを呼び出して実際にプロトコルを実行します。

20.6.7 HTTPPasswordMgr オブジェクト

以下のメソッドは `HTTPPasswordMgr` および `HTTPPasswordMgrWithDefaultRealm` オブジェクトで利用できます。

`HTTPPasswordMgr.add_password(realm, uri, user, passwd)`

`uri` は単一の URI でも複数の URI からなるシーケンスでもかまいません。 `realm`、 `user` および `passwd` は文字列でなくてはなりません。このメソッドによって、 `realm` と与えられた URI の上位 URI に対して (`user`, `passwd`) が認証トークンとして使われるようになります。

`HTTPPasswordMgr.find_user_password(realm, authuri)`

与えられたレルムおよび URI に対するユーザ名またはパスワードがあればそれを取得します。該当するユーザ名/パスワードが存在しない場合、このメソッドは (`None`, `None`) を返します。

`HTTPPasswordMgrWithDefaultRealm` オブジェクトでは、与えられた `realm` に対して該当するユーザ名/パスワードが存在しない場合、レルム `None` が検索されます。

20.6.8 AbstractBasicAuthHandler オブジェクト

`AbstractBasicAuthHandler.http_error_auth_reqcd(authreq, host, req, headers)`

ユーザ名/パスワードを取得し、再度サーバへのリクエストを試みることで、サーバからの認証リクエストを処理します。 `authreq` はリクエストにおいてレルムに関する情報が含まれているヘッダの名前、 `host` は認証を行う対象の URL とパスを指定します、 `req` は (失敗した) `Request` オブジェクト、そして `headers` はエラーヘッダでなくてはなりません。

`host` は、オーソリティ (例 "python.org") か、オーソリティコンポーネントを含む URL (例 "http://python.org") です。どちらの場合も、オーソリティはユーザ情報コンポーネントを含んではいけません (なので、 "python.org" や "python.org:80" は正しく、 "joe:password@python.org" は不正です)。

20.6.9 HTTPBasicAuthHandler オブジェクト

`HTTPBasicAuthHandler.http_error_401(req, fp, code, msg, hdrs)`

認証情報がある場合、認証情報付きで再度リクエストを試みます。

20.6.10 ProxyBasicAuthHandler オブジェクト

`ProxyBasicAuthHandler.http_error_407(req, fp, code, msg, hdrs)`

認証情報がある場合、認証情報付きで再度リクエストを試みます。

20.6.11 AbstractDigestAuthHandler オブジェクト

`AbstractDigestAuthHandler.http_error_auth_reged(authreq, host, req, headers)`

`authreq` はリクエストにおいてレルムに関する情報が含まれているヘッダの名前、`host` は認証を行う対象のホスト名、`req` は (失敗した) `Request` オブジェクト、そして `headers` はエラーヘッダでなくてはなりません。

20.6.12 HTTPDigestAuthHandler オブジェクト

`HTTPDigestAuthHandler.http_error_401(req, fp, code, msg, hdrs)`

認証情報がある場合、認証情報付きで再度リクエストを試みます。

20.6.13 ProxyDigestAuthHandler オブジェクト

`ProxyDigestAuthHandler.http_error_407(req, fp, code, msg, hdrs)`

認証情報がある場合、認証情報付きで再度リクエストを試みます。

20.6.14 HTTPHandler オブジェクト

`HTTPHandler.http_open(req)`

HTTP リクエストを送ります。 `req.has_data()` に応じて、GET または POST のどちらでも送ることができます。

20.6.15 HTTPSHandler オブジェクト

`HTTPSHandler.https_open(req)`

HTTPS リクエストを送ります。 `req.has_data()` に応じて、GET または POST のどちらでも送ることができます。

20.6.16 FileHandler オブジェクト

`FileHandler.file_open(req)`

ホスト名がない場合、またはホスト名が `'localhost'` の場合にファイルをローカルでオープンします。そうでない場合、プロトコルを `ftp` に切り替え、`parent` を使って再度オープンを試みます。

20.6.17 FTPHandler オブジェクト

`FTPHandler.ftp_open(req)`

`req` で表されるファイルを FTP 越しにオープンします。ログインは常に空のユーザー名およびパスワードで行われます。

20.6.18 CacheFTPHandler オブジェクト

`CacheFTPHandler` オブジェクトは `FTPHandler` オブジェクトに以下のメソッドを追加したものです:

`CacheFTPHandler.setTimeout(t)`

接続のタイムアウトを t 秒に設定します。

`CacheFTPHandler.setMaxConns(m)`

キャッシュ付き接続の最大接続数を m に設定します。

20.6.19 UnknownHandler オブジェクト

`UnknownHandler.unknown_open()`

例外 `URLError` を送出します。

20.6.20 HTTPErrorProcessor オブジェクト

バージョン 2.4 で追加.

`HTTPErrorProcessor.http_response()`

HTTP エラー応答の処理をします。

エラーコード 200 の場合、レスポンスオブジェクトを即座に返します。

200 以外のエラーコードの場合、`OpenerDirector.error()` を介して `protocol.error_code` メソッドに仕事を引き渡します。最終的にどのハンドラもエラーを処理しなかった 場合、`urllib2.HTTPDefaultErrorHandler` が `HTTPError` を送出します。

`HTTPErrorProcessor.https_response()`

HTTPS エラー応答の処理をします。

振る舞いは `http_response()` と同じです。

20.6.21 例

以下の例の他に `urllib-howto` に多くの例があります。

以下の例では、`python.org` のメインページを取得して、その最初の 100 バイト分を表示します:

```
>>> import urllib2
>>> f = urllib2.urlopen('http://www.python.org/')
>>> print f.read(100)
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<?xml-stylesheet href="/css/ht2html
```

今度は CGI の標準入力にデータストリームを送信し、CGI が返すデータを読み出します。この例は Python が SSL をサポートしている場合にのみ動作することに注意してください。

```
>>> import urllib2
>>> req = urllib2.Request(url='https://localhost/cgi-bin/test.cgi',
...                        data='This data is passed to stdin of the CGI')
>>> f = urllib2.urlopen(req)
>>> print f.read()
Got Data: "This data is passed to stdin of the CGI"
```

上の例で使われているサンプルの CGI は以下のようにになっています:

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print 'Content-type: text-plain\n\nGot Data: "%s"' % data
```

以下はベーシック HTTP 認証の例です:

```
import urllib2
# Create an OpenerDirector with support for Basic HTTP Authentication...
auth_handler = urllib2.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-updates.py',
                          user='klem',
                          passwd='kadidd!ehopper')
opener = urllib2.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib2.install_opener(opener)
urllib2.urlopen('http://www.example.com/login.html')
```

`build_opener()` はデフォルトで沢山のハンドラを提供しており、その中に `ProxyHandler` があります。デフォルトでは、`ProxyHandler` は `<scheme>.proxy` という環境変数を使います。ここで `<scheme>` は URL スキームです。例えば、HTTP プロキシの URL を得るには、環境変数 `http.proxy` を読み出します。

この例では、デフォルトの `ProxyHandler` を置き換えてプログラマ的に作成したプロキシ URL を使うようにし、`ProxyBasicAuthHandler` でプロキシ認証サポートを追加します。

```
proxy_handler = urllib2.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib2.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib2.build_opener(proxy_handler, proxy_auth_handler)
```

(次のページに続く)

(前のページからの続き)

```
# This time, rather than install the OpenerDirector, we use it directly:
opener.open('http://www.example.com/login.html')
```

以下は HTTP ヘッダを追加する例です:

`headers` 引数を使って `Request` コンストラクタを呼び出す方法の他に、以下のようにできます:

```
import urllib2
req = urllib2.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
# Customize the default User-Agent header value:
req.add_header('User-Agent', 'urllib-example/0.1 (Contact: . . .)')
r = urllib2.urlopen(req)
```

`OpenerDirector` は全ての `Request` に `User-Agent` ヘッダを自動的に追加します。これを変更するには以下のようにします:

```
import urllib2
opener = urllib2.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/')
```

また、`Request` が `urlopen()` (や `OpenerDirector.open()`) に渡される際には、いくつかの標準ヘッダ (`Content-Length`, `Content-Type` および `Host`) も追加されることを忘れないでください。

20.7 httpplib — HTTP プロトコルクライアント

注釈: `httpplib` モジュールは、Python 3.0 では `http.client` にリネームされました。2to3 ツールが自動的にソースコードの `import` を修正します。

Source code: [Lib/httpplib.py](#)

このモジュールでは HTTP および HTTPS プロトコルのクライアント側を実装しているクラスを定義しています。通常、このモジュールは直接使いません — `urllib` モジュールが HTTP や HTTPS を使った URL を扱う上でこのモジュールを使います。

参考:

より高水準の HTTP クライアントインターフェイスとしては [Requests package](#) がおすすめです。

注釈: HTTPS のサポートは、`socket` モジュールが SSL サポート付きでコンパイルされている場合にのみ利用できます。

注釈: このモジュールは Python 2.0 で大幅に公開インターフェイスを変更しました。HTTP クラスは 1.5.2 との後方互換性のためだけに残されています。新しいコードではこれは使ってはいけません。このドキュメントでは説明しないので、docstring をみてください。

このモジュールでは以下のクラスを提供しています:

```
class httplib.HTTPConnection (host[, port[, strict[, timeout[, source_address]]]])
```

`HTTPConnection` インスタンスは、HTTP サーバとの一回のトランザクションを表現します。インスタンスの生成はホスト名とオプションのポート番号を与えて行います。ポート番号を指定しなかった場合、ホスト名文字列が `host:port` の形式であれば、ホスト名からポート番号を導き、そうでない場合には標準の HTTP ポート番号 (80) を使います。オプションパラメータ `strict` (デフォルトは偽) を真にすると、ステータス行が HTTP/1.0 あるいは HTTP/1.1 として解釈出来ない場合に `BadStatusLine` を送出します。オプションの引数 `timeout` が渡された場合、ブロックする処理 (コネクション接続など) のタイムアウト時間 (秒数) として利用されます (渡されなかった場合は、グローバルのデフォルトタイムアウト設定が利用されます)。オプションの引数 `source_address` を (host, port) という形式のタプルにすると HTTP 接続の接続元アドレスとして使用します。

例えば、以下の呼び出しは全て同じサーバの同じポートに接続するインスタンスを生成します:

```
>>> h1 = httplib.HTTPConnection('www.cwi.nl')
>>> h2 = httplib.HTTPConnection('www.cwi.nl:80')
>>> h3 = httplib.HTTPConnection('www.cwi.nl', 80)
>>> h3 = httplib.HTTPConnection('www.cwi.nl', 80, timeout=10)
```

バージョン 2.0 で追加。

バージョン 2.6 で変更: `timeout` が追加されました。

バージョン 2.7 で変更: `source_address` が追加されました。

```
class httplib.HTTPSConnection (host[, port[, key_file[, cert_file[, strict[, timeout[,
                                                                    source_address[, context]]]]]]])
```

`HTTPConnection` のサブクラスはセキュア・サーバとやりとりする為の SSL を使う場合に用います。デフォルトのポート番号は 443 です。`context` が指定されれば、それは様々な SSL オプションを記述する `ssl.SSLContext` インスタンスでなければなりません。

`key_file` と `cert_file` は廃止されたので、代わりに `ssl.SSLContext.load_cert_chain()` を使うか、または `ssl.create_default_context()` が、システムが信頼する CA 証明書をあなたのために選んでくれます。

ベストプラクティスに関するより良い情報が [セキュリティで考慮すべき点](#) にありますのでお読みください。

バージョン 2.0 で追加。

バージョン 2.6 で変更: `timeout` が追加されました。

バージョン 2.7 で変更: `source_address` が追加されました。

バージョン 2.7.9 で変更: *context* が追加されました。

このクラスは今や全ての必要な証明書とホスト名の検証をデフォルトで行うようになりました。昔の、検証を行わない振る舞いに戻したければ、*context* に `ssl._create_unverified_context()` を渡すことで出来ます。

class `httplib.HTTPResponse` (*sock, debuglevel=0, strict=0*)

コネクションに成功したときに、このクラスのインスタンスが返されます。ユーザーから直接利用されることはありません。

バージョン 2.0 で追加.

class `httplib.HTTPMessage`

HTTPMessage のインスタンスは、HTTP レスポンスヘッダを格納するために利用されます。*mimetools.Message* クラスを利用して実装されていて、HTTP ヘッダを扱うための便利な関数を提供しています。このクラスはユーザーが直接インスタンス生成するものではありません。

必要に応じて以下の例外が送出されます:

exception `httplib.HTTPException`

このモジュールにおける他の例外クラスの基底クラスです。 *Exception* のサブクラスです。

バージョン 2.0 で追加.

exception `httplib.NotConnected`

HTTPException サブクラスです。

バージョン 2.0 で追加.

exception `httplib.InvalidURL`

HTTPException のサブクラスです。ポート番号を指定したものの、その値が数字でなかったり空のオブジェクトであった場合に送出されます。

バージョン 2.3 で追加.

exception `httplib.UnknownProtocol`

HTTPException サブクラスです。

バージョン 2.0 で追加.

exception `httplib.UnknownTransferEncoding`

HTTPException サブクラスです。

バージョン 2.0 で追加.

exception `httplib.UnimplementedFileMode`

HTTPException サブクラスです。

バージョン 2.0 で追加.

exception `httplib.IncompleteRead`

HTTPException サブクラスです。

バージョン 2.0 で追加.

exception `httplib.ImproperConnectionState`

HTTPException サブクラスです。

バージョン 2.0 で追加.

exception `httplib.CannotSendRequest`

ImproperConnectionState のサブクラスです。

バージョン 2.0 で追加.

exception `httplib.CannotSendHeader`

ImproperConnectionState のサブクラスです。

バージョン 2.0 で追加.

exception `httplib.ResponseNotReady`

ImproperConnectionState のサブクラスです。

バージョン 2.0 で追加.

exception `httplib.BadStatusLine`

HTTPException のサブクラスです。サーバが理解できない HTTP 状態コードで応答した場合に送出されます。

バージョン 2.0 で追加.

このモジュールで定義されている定数は以下の通りです:

`httplib.HTTP_PORT`

HTTP プロトコルの標準のポート (通常は 80) です。

`httplib.HTTPS_PORT`

HTTPS プロトコルの標準のポート (通常は 443) です。

また、整数の状態コードについて以下の定数が定義されています:

定数	値	定義
CONTINUE	100	HTTP/1.1, RFC 2616, Section 10.1.1
SWITCHING_PROTOCOLS	101	HTTP/1.1, RFC 2616, Section 10.1.2
PROCESSING	102	WEBDAV, RFC 2518, Section 10.1
OK	200	HTTP/1.1, RFC 2616, Section 10.2.1
CREATED	201	HTTP/1.1, RFC 2616, Section 10.2.2
ACCEPTED	202	HTTP/1.1, RFC 2616, Section 10.2.3
NON_AUTHORITATIVE_INFORMATION	203	HTTP/1.1, RFC 2616, Section 10.2.4
NO_CONTENT	204	HTTP/1.1, RFC 2616, Section 10.2.5
RESET_CONTENT	205	HTTP/1.1, RFC 2616, Section 10.2.6
PARTIAL_CONTENT	206	HTTP/1.1, RFC 2616, Section 10.2.7

次のページに続く

表 1 – 前のページからの続き

定数	値	定義
MULTI_STATUS	207	WEBDAV RFC 2518, Section 10.2
IM_USED	226	Delta encoding in HTTP, RFC 3229 , Section 10.4.1
MULTIPLE_CHOICES	300	HTTP/1.1, RFC 2616, Section 10.3.1
MOVED_PERMANENTLY	301	HTTP/1.1, RFC 2616, Section 10.3.2
FOUND	302	HTTP/1.1, RFC 2616, Section 10.3.3
SEE_OTHER	303	HTTP/1.1, RFC 2616, Section 10.3.4
NOT_MODIFIED	304	HTTP/1.1, RFC 2616, Section 10.3.5
USE_PROXY	305	HTTP/1.1, RFC 2616, Section 10.3.6
TEMPORARY_REDIRECT	307	HTTP/1.1, RFC 2616, Section 10.3.8
BAD_REQUEST	400	HTTP/1.1, RFC 2616, Section 10.4.1
UNAUTHORIZED	401	HTTP/1.1, RFC 2616, Section 10.4.2
PAYMENT_REQUIRED	402	HTTP/1.1, RFC 2616, Section 10.4.3
FORBIDDEN	403	HTTP/1.1, RFC 2616, Section 10.4.4
NOT_FOUND	404	HTTP/1.1, RFC 2616, Section 10.4.5
METHOD_NOT_ALLOWED	405	HTTP/1.1, RFC 2616, Section 10.4.6
NOT_ACCEPTABLE	406	HTTP/1.1, RFC 2616, Section 10.4.7
PROXY_AUTHENTICATION_REQUIRED	407	HTTP/1.1, RFC 2616, Section 10.4.8
REQUEST_TIMEOUT	408	HTTP/1.1, RFC 2616, Section 10.4.9
CONFLICT	409	HTTP/1.1, RFC 2616, Section 10.4.10
GONE	410	HTTP/1.1, RFC 2616, Section 10.4.11
LENGTH_REQUIRED	411	HTTP/1.1, RFC 2616, Section 10.4.12
PRECONDITION_FAILED	412	HTTP/1.1, RFC 2616, Section 10.4.13
REQUEST_ENTITY_TOO_LARGE	413	HTTP/1.1, RFC 2616, Section 10.4.14
REQUEST_URI_TOO_LONG	414	HTTP/1.1, RFC 2616, Section 10.4.15
UNSUPPORTED_MEDIA_TYPE	415	HTTP/1.1, RFC 2616, Section 10.4.16
REQUESTED_RANGE_NOT_SATISFIABLE	416	HTTP/1.1, RFC 2616, Section 10.4.17
EXPECTATION_FAILED	417	HTTP/1.1, RFC 2616, Section 10.4.18
UNPROCESSABLE_ENTITY	422	WEBDAV, RFC 2518, Section 10.3
LOCKED	423	WEBDAV RFC 2518, Section 10.4
FAILED_DEPENDENCY	424	WEBDAV, RFC 2518, Section 10.5
UPGRADE_REQUIRED	426	HTTP Upgrade to TLS, RFC 2817 , Section 6
INTERNAL_SERVER_ERROR	500	HTTP/1.1, RFC 2616, Section 10.5.1
NOT_IMPLEMENTED	501	HTTP/1.1, RFC 2616, Section 10.5.2
BAD_GATEWAY	502	HTTP/1.1 RFC 2616, Section 10.5.3
SERVICE_UNAVAILABLE	503	HTTP/1.1, RFC 2616, Section 10.5.4
GATEWAY_TIMEOUT	504	HTTP/1.1 RFC 2616, Section 10.5.5
HTTP_VERSION_NOT_SUPPORTED	505	HTTP/1.1, RFC 2616, Section 10.5.6
INSUFFICIENT_STORAGE	507	WEBDAV, RFC 2518, Section 10.6
NOT_EXTENDED	510	An HTTP Extension Framework, RFC 2774 , Section 7

`httplib.responses`

このディクショナリは、HTTP 1.1 ステータスコードを W3C の名前にマップしたものです。

例: `httplib.responses[httplib.NOT_FOUND]` は 'Not Found' を示します。

バージョン 2.5 で追加.

20.7.1 HTTPConnection オブジェクト

`HTTPConnection` インスタンスには以下のメソッドがあります:

`HTTPConnection.request(method, url[, body[, headers]])`

このメソッドは、HTTP 要求メソッド `method` およびセクタ `url` を使って、要求をサーバに送ります。`body` 引数を指定する場合、ヘッダが終了した後に送信する文字列データでなければなりません。もしくは、開いているファイルオブジェクトを `body` に渡すこともできます。その場合、そのファイルの内容が送信されます。このファイルオブジェクトは、`fileno()` と `read()` メソッドをサポートしている必要があります。`headers` 引数は要求と同時に送信される拡張 HTTP ヘッダの内容からなるマップ型でなくてはなりません。

`headers` で Content-Length が提供されない場合、全てのメソッドにおいて、`body` の長さがわかるならば、つまり `str` としての長さあるいはファイルのディスク上のサイズをもとに、Content-Length ヘッダは自動的に正しい値にセットされます。`body` が `None` の場合はそのヘッダは、`body` がないメソッドではセットされず、`body` が期待されているメソッド (PUT, POST, PATCH) では 0 にセットします。

バージョン 2.6 で変更: `body` にファイルオブジェクトを渡せるようになりました

`HTTPConnection.getresponse()`

サーバに対して HTTP 要求を送り出した後に呼び出されなければなりません。要求に対する応答を取得します。`HTTPResponse` インスタンスを返します。

注釈: すべての応答を読み込んでからでなければ新しい要求をサーバに送ることはできないことに注意しましょう。

`HTTPConnection.set_debuglevel(level)`

デバッグレベル (印字されるデバッグ出力の量) を設定します。デフォルトのデバッグレベルは 0 で、デバッグ出力を全く印字しません。

`HTTPConnection.set_tunnel(host, port=None, headers=None)`

Set the host and the port for HTTP Connect Tunnelling. Normally used when it is required to do HTTPS Connection through a proxy server.

ヘッダのパラメータは CONNECT リクエストで送信するために他の HTTP ヘッダにマッピングされます。

バージョン 2.7 で追加.

`HTTPConnection.connect()`

オブジェクトを生成するときに指定したサーバに接続します。

`HTTPConnection.close()`

サーバへの接続を閉じます。

上で説明した `request()` メソッドを使うかわりに、以下の 4 つの関数を使用して要求をステップバイステップで送信することもできます。

`HTTPConnection.putrequest(request, selector[, skip_host[, skip_accept_encoding]])`

サーバへの接続が確立したら、最初にこのメソッドを呼び出さなくてはなりません。このメソッドは `request` 文字列、`selector` 文字列、そして HTTP バージョン (HTTP/1.1) からなる一行を送信します。Host: や Accept-Encoding: ヘッダの自動送信を無効にしたい場合 (例えば別のコンテンツエンコーディングを受け入れたい場合) には、`skip_host` や `skip_accept_encoding` を偽でない値に設定してください。

バージョン 2.4 で変更: `skip_accept_encoding` 引数が追加されました。

`HTTPConnection.putheader(header, argument[, ...])`

RFC 822 形式のヘッダをサーバに送ります。この処理では、`header`、コロンとスペース、そして最初の引数からなる 1 行をサーバに送ります。追加の引数を指定した場合、継続して各行にタブ一つと引数の入った引数行が送信されます。

`HTTPConnection.endheaders(message_body=None)`

サーバに空行を送り、ヘッダ部が終了したことを通知します。オプションの `message_body` 引数を、リクエストに関連したメッセージボディを渡すのに使うことができます。

バージョン 2.7 で変更: `message_body` が追加されました。

`HTTPConnection.send(data)`

サーバにデータを送ります。このメソッドは `endheaders()` が呼び出された直後で、かつ `getresponse()` が呼び出される前に使わなくてはなりません。

20.7.2 HTTPResponse オブジェクト

`HTTPResponse` インスタンスは以下のメソッドと属性を持っています:

`HTTPResponse.read([amt])`

応答の本体全体か、`amt` バイトまで読み出して返します。

`HTTPResponse.getheader(name[, default])`

ヘッダ `name` の内容を取得して返すか、該当するヘッダがない場合には `default` を返します。

`HTTPResponse.getheaders()`

(header, value) のタプルからなるリストを返します。

バージョン 2.4 で追加.

`HTTPResponse.fileno()`

下層のソケットの `fileno` を返します。

`HTTPResponse.msg`

応答ヘッダを含む *mimetools.Message* インスタンスです。

`HTTPResponse.version`

サーバが使用した HTTP プロトコルバージョンです。10 は HTTP/1.0 を、11 は HTTP/1.1 を表します。

`HTTPResponse.status`

サーバから返される状態コードです。

`HTTPResponse.reason`

サーバから返される応答の理由文です。

20.7.3 例

以下は GET リクエストの送信方法を示した例です:

```
>>> import httplib
>>> conn = httplib.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print r1.status, r1.reason
200 OK
>>> data1 = r1.read()
>>> conn.request("GET", "/")
>>> r2 = conn.getresponse()
>>> print r2.status, r2.reason
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

次の例のセッションでは、HEAD メソッドを利用しています。HEAD メソッドは全くデータを返さないことに注目してください。

```
>>> import httplib
>>> conn = httplib.HTTPSConnection("www.python.org")
>>> conn.request("HEAD", "/")
>>> res = conn.getresponse()
>>> print res.status, res.reason
200 OK
>>> data = res.read()
>>> print len(data)
0
>>> data == ''
True
```

以下は POST リクエストの送信方法を示した例です:

```
>>> import httplib, urllib
>>> params = urllib.urlencode({'@number': 12524, '@type': 'issue', '@action': 'show
↵'})
```

(次のページに続く)

(前のページからの続き)

```
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...             "Accept": "text/plain"}
>>> conn = httplib.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
>>> print response.status, response.reason
302 Found
>>> data = response.read()
>>> data
'Redirecting to <a href="http://bugs.python.org/issue12524">http://bugs.python.org/
↪issue12524</a>'
>>> conn.close()
```

HTTP PUT リクエストは POST リクエストととても似ています。違いは、HTTP サーバが PUT リクエストによりリソースの作成をサーバ側に行うことだけです。httplib を使って PUT リクエストを行うセッションの例です:

```
>>> # This creates an HTTP message
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/foobar
...
>>> import httplib
>>> BODY = "***filecontents***"
>>> conn = httplib.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print response.status, response.reason
200, OK
```

20.8 ftplib — FTP プロトコルクライアント

Source code: [Lib/ftplib.py](#)

このモジュールでは *FTP* クラスと、それに関連するいくつかの項目を定義しています。 *FTP* クラスは、FTP プロトコルのクライアント側の機能を備えています。このクラスを使うと FTP のいろいろな機能の自動化、例えば他の FTP サーバのミラーリングといったことを実行する Python プログラムを書くことができます。また、 *urllib* モジュールも FTP を使う URL を操作するのにこのクラスを使っています。FTP (File Transfer Protocol) についての詳しい情報は Internet **RFC 959** を参照して下さい。

ftplib モジュールを使ったサンプルを以下に示します:

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.debian.org')      # connect to host, default port
>>> ftp.login()                      # user anonymous, passwd anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')                # change into "debian" directory
```

(次のページに続く)

(前のページからの続き)

```
>>> ftp.retrlines('LIST')           # list directory contents
-rw-rw-r-- 1 1176 1176 1063 Jun 15 10:18 README
...
drwxr-sr-x 5 1176 1176 4096 Dec 19 2000 pool
drwxr-sr-x 4 1176 1176 4096 Nov 17 2008 project
drwxr-xr-x 3 1176 1176 4096 Oct 10 2012 tools
'226 Directory send OK.'
>>> ftp.retrbinary('RETR README', open('README', 'wb').write)
'226 Transfer complete.'
>>> ftp.quit()
```

このモジュールは以下の項目を定義しています:

class `ftplib.FTP` (`[host[, user[, passwd[, acct[, timeout]]]]]`)

FTP クラスの新しいインスタンスを返します。 `host` が与えられると、 `connect(host)` メソッドが実行されます。 `user` が与えられると、さらに `login(user, passwd, acct)` メソッドが実行されます (この `passwd` と `acct` は指定されなければデフォルトでは空文字列です)。

バージョン 2.6 で変更: `timeout` が追加されました。

class `ftplib.FTP_TLS` (`[host[, user[, passwd[, acct[, keyfile[, certfile[, context[, timeout]]]]]]]`)

RFC 4217 に記述されている TLS サポートを FTP に加えた *FTP* のサブクラスです。認証の前に FTP コントロール接続を暗黙にセキュアにし、通常通りに port 21 に接続します。データ接続をセキュアにするには、ユーザが `prot_p()` メソッドを呼び出してそれを明示的に要求しなければなりません。 `context` は SSL 設定オプション、証明書、秘密鍵を一つの (潜在的に長生きの) 構造にまとめた `ssl.SSLContext` オブジェクトです。ベストプラクティスについての [セキュリティで考慮すべき点](#) をお読みください。

`keyfile` と `certfile` は `context` のレガシー版です – これらは、SSL 接続のための、PEM フォーマットの秘密鍵と証明書チェーンファイル名 (前者が `keyfile`、後者が `certfile`) を含むことができます。

バージョン 2.7 で追加。

バージョン 2.7.10 で変更: `context` パラメータが追加されました。

FTP_TLS クラスを使ったサンプルセッションはこちらです:

```
>>> from ftplib import FTP_TLS
>>> ftps = FTP_TLS('ftp.python.org')
>>> ftps.login()           # login anonymously before securing control channel
>>> ftps.prot_p()         # switch to secure data connection
>>> ftps.retrlines('LIST') # list directory content securely
total 9
drwxr-xr-x 8 root wheel 1024 Jan 3 1994 .
drwxr-xr-x 8 root wheel 1024 Jan 3 1994 ..
drwxr-xr-x 2 root wheel 1024 Jan 3 1994 bin
drwxr-xr-x 2 root wheel 1024 Jan 3 1994 etc
d-wxrw-r-x 2 ftp wheel 1024 Sep 5 13:43 incoming
drwxr-xr-x 2 root wheel 1024 Nov 17 1993 lib
```

(次のページに続く)

(前のページからの続き)

```
drwxr-xr-x   6 1094   wheel      1024 Sep 13 19:07 pub
drwxr-xr-x   3 root    wheel      1024 Jan  3  1994 usr
-rw-r--r--   1 root    root        312 Aug  1  1994 welcome.msg
'226 Transfer complete.'
>>> ftps.quit()
>>>
```

exception `ftplib.error_reply`

サーバから想定外の応答があった時に発生する例外。

exception `ftplib.error_temp`

一時的エラーを表すエラーコード (400–499 の範囲の応答コード) を受け取った時に発生する例外。

exception `ftplib.error_perm`

永久エラーを表すエラーコード (500–599 の範囲の応答コード) を受け取った時に発生する例外。

exception `ftplib.error_proto`

File Transfer Protocol の応答仕様に適合しない、すなわち 1–5 の数字で始まらない応答コードをサーバから受け取った時に発生する例外。

`ftplib.all_errors`

FTP インスタンスのメソッド実行時、FTP 接続で (プログラミングのエラーと考えられるメソッドの実行によって) 発生する全ての例外 (タプル形式)。この例外には以上の 4 つのエラーはもちろん、*socket.error* と *IOError* も含まれます。

参考:

netrc モジュール *.netrc* ファイルフォーマットのパーザ。 *.netrc* ファイルは、FTP クライアントがユーザにプロンプトを出す前に、ユーザ認証情報をロードするのによく使われます。

Python のソースディストリビューションの `Tools/scripts/ftpmirror.py` ファイルは、FTP サイトあるいはその一部をミラーリングするスクリプトで、*ftplib* モジュールを使っています。このモジュールを適用した応用例として使うことができます。

20.8.1 FTP オブジェクト

いくつかのコマンドは 2 つのタイプについて実行します: 1 つはテキストファイルで、もう 1 つはバイナリファイルを扱います。これらのメソッドのテキストバージョンでは *lines*、バイナリバージョンでは *binary* の語がメソッド名の終わりについています。

FTP インスタンスには以下のメソッドがあります:

`FTP.set_debuglevel (level)`

インスタンスのデバッグレベルを設定します。この設定によってデバッグ時に出力される量を調節します。デフォルトは 0 で、何も出力されません。1 なら、一般的に 1 つのコマンドあたり 1 行の適当な量のデバッグ出力を行います。2 以上なら、コントロール接続で受信した各行を出力して、最大のデバッグ出力をします。

`FTP.connect(host[, port[, timeout]])`

指定されたホストとポートに接続します。ポート番号のデフォルト値は FTP プロトコルの仕様で定められた 21 です。他のポート番号を指定する必要はめったにありません。この関数はひとつのインスタンスに対して一度だけ実行すべきです；インスタンスが作られた時にホスト名が与えられていたら、呼び出すべきではありません。これ以外の他の全てのメソッドは接続された後で実行可能となります。

オプションの *timeout* 引数は、コネクションの接続におけるタイムアウト時間を秒数で指定します。*timeout* が渡されなかった場合、グローバルのデフォルトタイムアウト設定が利用されます。

バージョン 2.6 で変更: *timeout* が追加されました。

`FTP.getwelcome()`

サーバに最初に接続した際に送信される応答中のウェルカムメッセージを返します。(このメッセージには時に、ユーザにとって重要な免責事項や ヘルプ情報が入っています。)

`FTP.login([user[, passwd[, acct]]])`

与えられた *user* でログインします。 *passwd* と *acct* のパラメータは省略可能で、デフォルトでは空文字列です。もし *user* が指定されないなら、デフォルトで 'anonymous' になります。もし *user* が 'anonymous' なら、デフォルトの *passwd* は 'anonymous@' になります。この関数は各インスタンスについて一度だけ、接続が確立した後に呼び出さなければなりません。インスタンスが作られた時にホスト名とユーザ名が与えられていたら、このメソッドを実行すべきではありません。ほとんどの FTP コマンドはクライアントがログインした後に実行可能になります。 *acct* 引数は "accounting information" を提供します。ほとんどのシステムはこれを実装していません。

`FTP.abort()`

実行中のファイル転送を中止します。これはいつも機能するわけではありませんが、やってみる価値があります。

`FTP.sendcmd(command)`

シンプルなコマンド文字列をサーバに送信して、受信した文字列を返します。

`FTP.voidcmd(command)`

シンプルなコマンド文字列をサーバに送信して、その応答を扱います。応答コードが成功に関係するものの (200–299 の範囲にあるコード) なら何も返しません。それ以外は `error_reply` を発生します。

`FTP.retrbinary(command, callback[, maxblocksize[, rest]])`

バイナリ転送モードでファイルを受信します。 *command* は適切な RETR コマンド: 'RETR filename' でなければなりません。関数 *callback* は、受信したデータブロックのそれぞれに対して、データブロックを 1 つの文字列の引数として呼び出されます。省略可能な引数 *maxblocksize* は、実際の転送を行うのに作られた低レベルのソケットオブジェクトから読み込む最大のチャンクサイズを指定します (これは *callback* に与えられるデータブロックの最大サイズにもなります)。妥当なデフォルト値が設定されます。 *rest* は、 `transfercmd()` メソッドと同じものです。

`FTP.retrlines(command[, callback])`

ASCII 転送モードでファイルとディレクトリのリストを受信します。 *command* は、適切な RETR コマンド (`retrbinary()` を参照) あるいは LIST, NLST, MLSD のようなコマンド (通常は文字列 'LIST') でなければなりません。LIST は、ファイルのリストとそれらのファイルに関する情報を受信します。NLST は、ファイル名のリストを受信します。サーバによっては、MLSD は機械で読めるリ

ストとそれらのファイルに関する情報を受信します。関数 `callback` は末尾の CRLF を取り除いた各行を引数にして実行されます。デフォルトでは `callback` は `sys.stdout` に各行を表示します。

`FTP.set_pasv(val)`

`val` が真なら”パッシブモード”をオンにし、そうでないならパッシブモードをオフにします。(Python 2.0 以前ではデフォルトでパッシブモードはオフにされていましたが、Python 2.1 以後ではデフォルトでオンになっています。)

`FTP.storbinary(command, fp[, blocksize, callback, rest])`

バイナリ転送モードでファイルを転送します。`command` は適切な STOR コマンド: "STOR filename" でなければなりません。`fp` は開かれたファイルオブジェクトで、`read()` メソッドで EOF まで読み込まれ、ブロックサイズ `blocksize` でデータが転送されます。引数 `blocksize` のデフォルト値は 8192 です。`callback` はオプションの引数で、引数を 1 つとる呼び出し可能オブジェクトを渡します。各データブロックが送信された後に、そのブロックを引数にして呼び出されます。`rest` は、`transfercmd()` メソッドにあるものと同じ意味です。

バージョン 2.1 で変更: `blocksize` のデフォルト値が追加されました。

バージョン 2.6 で変更: `callback` 引数が追加されました。

バージョン 2.7 で変更: `rest` パラメタが追加されました。

`FTP.storlines(command, fp[, callback])`

ASCII 転送モードでファイルを転送します。`command` は適切な STOR コマンドでなければなりません(`storbinary()` を参照)。`fp` は開かれたファイルオブジェクトで、`readline()` メソッドで EOF まで読み込まれ、各行がデータが転送されます。`callback` はオプションの引数で、引数を 1 つとる呼び出し可能オブジェクトを渡します。各行が送信された後に、その行数を引数にして呼び出されます。

バージョン 2.6 で変更: `callback` 引数が追加されました。

`FTP.transfercmd(cmd[, rest])`

データ接続中に転送を初期化します。もし転送中なら、EPRT あるいは PORT コマンドと、`cmd` で指定したコマンドを送信し、接続を続けます。サーバがパッシブなら、EPSV あるいは PASV コマンドを送信して接続し、転送コマンドを開始します。どちらの場合も、接続のためのソケットを返します。

省略可能な `rest` が与えられたら、REST コマンドがサーバに送信され、`rest` を引数として与えます。`rest` は普通、要求したファイルのバイトオフセット値で、最初のバイトをとばして指定したオフセット値からファイルのバイト転送を再開するよう伝えます。しかし、RFC 959 では `rest` が印字可能な ASCII コード 33 から 126 の範囲の文字列からなることを要求していることに注意して下さい。したがって、`transfercmd()` メソッドは `rest` を文字列に変換しますが、文字列の内容についてチェックしません。もし REST コマンドをサーバが認識しないなら、例外 `error_reply` が発生します。この例外が発生したら、引数 `rest` なしに `transfercmd()` を実行します。

`FTP.nttransfercmd(cmd[, rest])`

`transfercmd()` と同様ですが、データと予想されるサイズとのタプルを返します。もしサイズが計算できないなら、サイズの代わりに `None` が返されます。`cmd` と `rest` は `transfercmd()` ののと同じです。

`FTP.nlst(argument[, ...])`

NLIST コマンドで返されるファイル名のリストを返します。省略可能な *argument* は、リストアップするディレクトリです（デフォルトではサーバのカレントディレクトリです）。NLST コマンドに非標準である複数の引数を渡すことができます。

FTP.**dir** (*argument*[, ...])

LIST コマンドで返されるディレクトリ内のリストを作り、標準出力へ出力します。省略可能な *argument* は、リストアップするディレクトリです（デフォルトではサーバのカレントディレクトリです）。LIST コマンドに非標準である複数の引数を渡すことができます。もし最後の引数が関数なら、`retrlines()` のように *callback* として使われます; デフォルトでは `sys.stdout` に印字します。このメソッドは `None` を返します。

FTP.**rename** (*fromname*, *toname*)

サーバ上のファイルのファイル名 *fromname* を *toname* へ変更します。

FTP.**delete** (*filename*)

サーバからファイル *filename* を削除します。成功したら応答のテキストを返し、そうでないならパーミッションエラーでは `error_perm` を、他のエラーでは `error_reply` を返します。

FTP.**cwd** (*pathname*)

サーバのカレントディレクトリを設定します。

FTP.**mkd** (*pathname*)

サーバ上に新たにディレクトリを作ります。

FTP.**pwd** ()

サーバ上のカレントディレクトリのパスを返します。

FTP.**rmd** (*dirname*)

サーバ上のディレクトリ *dirname* を削除します。

FTP.**size** (*filename*)

サーバ上のファイル *filename* のサイズを尋ねます。成功したらファイルサイズが整数で返され、そうでないなら `None` が返されます。SIZE コマンドは標準化されていませんが、多くの普通のサーバで実装されていることに注意して下さい。

FTP.**quit** ()

サーバに QUIT コマンドを送信し、接続を閉じます。これは接続を閉じるのに”礼儀正しい”方法ですが、QUIT コマンドに反応してサーバの例外が発生するかもしれません。この例外は、`close()` メソッドによって FTP インスタンスに対するその後のコマンド使用が不可になっていることを示しています（下記参照）。

FTP.**close** ()

接続を一方的に閉じます。既に閉じた接続に対して実行すべきではありません（例えば `quit()` を呼び出して成功した後など）。この実行の後、FTP インスタンスはもう使用すべきではありません（`close()` あるいは `quit()` を呼び出した後で、`login()` メソッドをもう一度実行して再び接続を開くことはできません）。

20.8.2 FTP_TLS オブジェクト

`FTP_TLS` クラスは `FTP` を継承し、さらにオブジェクトを定義します:

`FTP_TLS.ssl_version`

使用する SSL のバージョン (デフォルトは `ssl.PROTOCOL_SSLv23`) です。

`FTP_TLS.auth()`

`ssl_version()` 属性で指定されたものに従って、TLS または SSL を使い、セキュアコントロール接続をセットアップします。

`FTP_TLS.prot_p()`

セキュアデータ接続をセットアップします。

`FTP_TLS.prot_c()`

平文データ接続をセットアップします。

20.9 poplib — POP3 プロトコルクライアント

ソースコード: [Lib/poplib.py](#)

このモジュールは、`POP3` クラスを定義します。これは POP3 サーバへの接続と、[RFC 1725](#) に定められたプロトコルを実装します。`POP3` クラスは `minimal` と `optimal` という 2 つのコマンドセットをサポートします。モジュールは `POP3_SSL` クラスも提供します。このクラスは下位のプロトコルレイヤーに SSL を使った POP3 サーバへの接続を提供します。

POP3 についての注意事項は、それが広くサポートされているにもかかわらず、既に時代遅れだということです。幾つも実装されている POP3 サーバーの品質は、貧弱なものが多数を占めています。もし、お使いのメールサーバーが IMAP をサポートしているなら、`imaplib.IMAP4` クラスが使えます。IMAP サーバーは、より良く実装されている傾向があります。

`poplib` モジュールは二つのクラスを提供します:

class `poplib.POP3` (`host` [, `port` [, `timeout`]])

このクラスが、実際に POP3 プロトコルを実装します。インスタンスが初期化されるときに、接続が作成されます。`port` が省略されると、POP3 標準のポート (110) が使われます。オプションの `timeout` 引数は、接続時のタイムアウト時間を秒数で指定します (指定されなかった場合は、グローバルのデフォルトタイムアウト設定が利用されます)。

バージョン 2.6 で変更: `timeout` が追加されました。

class `poplib.POP3_SSL` (`host` [, `port` [, `keyfile` [, `certfile`]]])

`POP3` クラスのサブクラスで、SSL でカプセル化されたソケットによる POP サーバへの接続を提供します。`port` が指定されていない場合、POP3-over-SSL 標準の 995 番ポートが使われます。`keyfile` と `certfile` もオプションです - SSL 接続に使われる PEM フォーマットの秘密鍵と証明書チェーンを指定出来ます。

バージョン 2.4 で追加。

1 つの例外が、`poplib` モジュールのアトリビュートとして定義されています:

exception `poplib.error_proto`

例外は、このモジュール内で起こったすべてのエラーで発生します。(`socket` モジュールからのエラーは捕まえず、そのまま伝播します) 例外の理由は文字列としてコンストラクタに渡されます。

参考:

モジュール `imaplib` The standard Python IMAP module.

Frequently Asked Questions About Fetchmail POP/IMAP クライアント `fetchmail` の FAQ。POP プロトコルをベースにしたアプリケーションを書くときに有用な、POP3 サーバの種類や RFC への適合度といった情報を収集しています。

20.9.1 POP3 オブジェクト

POP3 コマンドはすべて、それと同じ名前のメソッドとして lower-case で表現されます。そしてそのほとんどは、サーバからのレスポンスとなるテキストを返します。

`POP3` クラスのインスタンスは以下のメソッドを持ちます:

`POP3.set_debuglevel (level)`

インスタンスのデバッグレベルを設定します。この設定によってデバッグ時に出力される量を調節します。デフォルトは 0 で、何も出力されません。1 なら、一般的に 1 つのコマンドあたり 1 行の適当な量のデバッグ出力を行います。2 以上なら、コントロール接続で受信した各行を出力して、最大のデバッグ出力をします。

`POP3.getwelcome ()`

POP3 サーバーから送られるグリーティングメッセージを返します。

`POP3.user (username)`

user コマンドを送出します。応答はパスワード要求を表示します。

`POP3.pass_ (password)`

パスワードを送出します。応答は、メッセージ数とメールボックスのサイズを含みます。注意: サーバー上のメールボックスは `quit ()` が呼ばれるまでロックされます。

`POP3.apop (user, secret)`

POP3 サーバーにログオンするのに、よりセキュアな APOP 認証を使用します。

`POP3.rpop (user)`

POP3 サーバーにログオンするのに、(UNIX の r-コマンドと同様の) RPOP 認証を使用します。

`POP3.stat ()`

メールボックスの状態を得ます。結果は 2 つの integer からなるタプルとなります。(message count, mailbox size)。

`POP3.list ([which])`

メッセージのリストを要求します。結果は (response, ['mesg_num octets', ...],

octets) という形式で表されます。 *which* が与えられると、それによりメッセージを指定します。

POP3.**retr** (*which*)

which 番のメッセージ全体を取り出し、そのメッセージに既読フラグを立てます。結果は (response, ['line', ...], octets) という形式で表されます。

POP3.**dele** (*which*)

which 番のメッセージに削除のためのフラグを立てます。ほとんどのサーバで、QUIT コマンドが実行されるまでは実際の削除は行われません (もっとも良く知られた例外は Eudora QPOP で、その配送メカニズムは RFC に違反しており、どんな切断状況でも削除操作を未解決にしています)。

POP3.**rset** ()

メールボックスの削除マークすべてを取り消します。

POP3.**noop** ()

何もしません。接続保持のために使われます。

POP3.**quit** ()

Signoff: commit changes, unlock mailbox, drop connection. サインオフ: 変更をコミットし、メールボックスをアンロックして、接続を破棄します。

POP3.**top** (*which*, *howmuch*)

メッセージヘッダと *howmuch* で指定した行数のメッセージを、*which* で指定したメッセージ分取り出します。結果は以下のような形式となります。 (response, ['line', ...], octets)。

このメソッドは POP3 の TOP コマンドを利用し、RETR コマンドのように、メッセージに既読フラグをセットしません。残念ながら、TOP コマンドは RFC では貧弱な仕様しか定義されておらず、しばしばノーブランドのサーバーでは (その仕様が) 守られていません。このメソッドを信用してしまう前に、実際に使用する POP サーバーでテストをしてください。

POP3.**uidl** ([*which*])

(ユニーク ID による) メッセージダイジェストのリストを返します。 *which* が設定されている場合、結果はユニーク ID を含みます。それは 'response msgnum uid という形式のメッセージ、または (response, ['msgnum uid', ...], octets) という形式のリストとなります。

POP3_SSL クラスのインスタンスは追加のメソッドを持ちません。このサブクラスのインターフェイスは親クラスと同じです。

20.9.2 POP3 の例

以下にメールボックスを開き、全てのメッセージを取得して印刷する最小の (エラーチェックをしない) 使用例を示します:

```
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
```

(次のページに続く)

(前のページからの続き)

```
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print j
```

モジュールの末尾に、より拡張的な使用例が収められたテストセクションがあります。

20.10 imaplib — IMAP4 プロトコルクライアント

ソースコード: [Lib/imaplib.py](#)

このモジュールでは三つのクラス、`IMAP4`、`IMAP4_SSL` と `IMAP4_stream` を定義します。これらのクラスは IMAP4 サーバへの接続をカプセル化し、[RFC 2060](#) に定義されている IMAP4rev1 クライアントプロトコルの大規模なサブセットを実装しています。このクラスは IMAP4 ([RFC 1730](#)) 準拠のサーバと後方互換性がありますが、`STATUS` コマンドは IMAP4 ではサポートされていないので注意してください。

`imaplib` モジュール内では三つのクラスを提供しており、`IMAP4` は基底クラスとなります:

```
class imaplib.IMAP4([host[, port]])
```

このクラスは実際の IMAP4 プロトコルを実装しています。インスタンスが初期化された際に接続が生成され、プロトコルバージョン (IMAP4 または IMAP4rev1) が決定されます。`host` が指定されていない場合、`''` (ローカルホスト) が用いられます。`port` が省略された場合、標準の IMAP4 ポート番号 (143) が用いられます。

例外は `IMAP4` クラスの属性として定義されています:

exception `IMAP4.error`

何らかのエラー発生の際に送出される例外です。例外の理由は文字列としてコンストラクタに渡されます。

exception `IMAP4.abort`

IMAP4 サーバのエラーが生じると、この例外が送出されます。この例外は `IMAP4.error` のサブクラスです。通常、インスタンスを閉じ、新たなインスタンスを再び生成することで、この例外から復旧できます。

exception `IMAP4.readonly`

この例外は書き込み可能なメールボックスの状態がサーバによって変更された際に送出されます。この例外は `IMAP4.error` のサブクラスです。他の何らかのクライアントが現在書き込み権限を獲得しており、メールボックスを開きなおして書き込み権限を再獲得する必要があります。

このモジュールではもう一つ、安全 (secure) な接続を使ったサブクラスがあります:

```
class imaplib.IMAP4_SSL([host[, port[, keyfile[, certfile]]]])
```

`IMAP4` から派生したサブクラスで、SSL 暗号化ソケットを介して接続を行います (このクラスを利用するためには SSL サポート付きでコンパイルされた socket モジュールが必要です)。`host` が指定されていない場合、`''` (ローカルホスト) が用いられます。`port` が省略された場合、標準の IMAP4-over-SSL

ポート番号 (993) が用いられます。 *keyfile* および *certfile* もオプションです - これらは SSL 接続のための PEM 形式の秘密鍵 (private key) と認証チェーン (certificate chain) ファイルです。

さらにもう一つのサブクラスは、子プロセスで確立した接続を使用する場合に使用します:

```
class imaplib.IMAP4_stream(command)
```

IMAP4 から派生したサブクラスで、*command* を `os.popen2()` に渡して作成される `stdin/stdout` デスクリプタと接続します。

バージョン 2.3 で追加.

以下のユーティリティ関数が定義されています:

```
imaplib.Internaldate2tuple(datestr)
```

IMAP4 の INTERNALDATE 文字列を解析してそれに相当するローカルタイムを返します。戻り値は `time.struct_time` のインスタンスか、文字列のフォーマットが不正な場合は `None` です。

```
imaplib.Int2AP(num)
```

整数を [A .. P] からなる文字集合を用いて表現した文字列に変換します。

```
imaplib.ParseFlags(flagstr)
```

IMAP4 FLAGS 応答を個々のフラグからなるタプルに変換します。

```
imaplib.Time2Internaldate(date_time)
```

date_time を IMAP4 の INTERNALDATE 表現形式に変換します。戻り値は "DD-Mmm-YYYY HH:MM:SS +HHMM" (ダブルクォートを含む) の形をした文字列です。 *date_time* 引数は (`time.time()` が返す) epoch からの経過秒数を表す数値 (int か float) か、(`time.localtime()` が返す) ローカルタイムを表現する 9 要素タプルか、ダブルクォートされた文字列です。文字列だった場合、それがすでに正しいフォーマットになっていると仮定されます。

IMAP4 メッセージ番号は、メールボックスに対する変更が行われた後には変化します; 特に、EXPUNGE 命令はメッセージの削除を行いますが、残ったメッセージには再度番号を振りなおします。従って、メッセージ番号ではなく、UID 命令を使い、その UID を利用するよう強く勧めます。

モジュールの末尾に、より拡張的な使用例が収められたテストセクションがあります。

参考:

プロトコルに関する記述、およびプロトコルを実装したサーバのソースとバイナリは、全てワシントン大学の IMAP Information Center (<https://www.washington.edu/imap/>) にあります。

20.10.1 IMAP4 オブジェクト

全ての IMAP4rev1 命令は、同じ名前のメソッドで表されており、大文字のものも小文字のものもあります。

命令に対する引数は全て文字列に変換されます。例外は AUTHENTICATE の引数と APPEND の最後の引数で、これは IMAP4 リテラルとして渡されます。必要に応じて (IMAP4 プロトコルが感知対象としている文字が文字列に入っており、かつ丸括弧か二重引用符で囲われていなかった場合) 文字列はクォートされます。し

かし、LOGIN 命令の *password* 引数は常にクオートされます。文字列がクオートされないようにしたい (例えば STORE 命令の *flags* 引数) 場合、文字列を丸括弧で囲んでください (例: `r'(\Deleted)'`)。

各命令はタプル: (*type*, [*data*, ...]) を返し、*type* は通常 'OK' または 'NO' です。*data* は命令に対する応答をテキストにしたものか、命令に対する実行結果です。各 *data* は文字列かタプルとなります。タブルの場合、最初の要素はレスポンスのヘッダで、次の要素にはデータが格納されます (ie: 'literal' value)。

以下のコマンドにおける *message_set* オプションは、操作の対象となるひとつあるいは複数のメッセージを指す文字列です。単一のメッセージ番号 ('1') かメッセージ番号の範囲 ('2:4')、あるいは連続していないメッセージをカンマでつなげたもの ('1:3,6:9') となります。範囲指定でアスタリスクを使用すると、上限を無限とすることができます ('3:*')。

IMAP4 のインスタンスは以下のメソッドを持っています:

IMAP4.**append** (*mailbox*, *flags*, *date_time*, *message*)

指定された名前のメールボックスに *message* を追加します。

IMAP4.**authenticate** (*mechanism*, *authobject*)

認証命令です — 応答の処理が必要です。

mechanism は利用する認証メカニズムを与えます。認証メカニズムはインスタンス変数 *capabilities* の中に AUTH=*mechanism* という形式で現れる必要があります。

authobject は呼び出し可能なオブジェクトである必要があります:

```
data = authobject(response)
```

これはサーバで継続応答を処理するためによべれます。これは (おそらく) 暗号化されて、サーバへ送られた *data* を返します。もしクライアントが中断応答 * を送信した場合にはこれは None を返します。

IMAP4.**check** ()

サーバ上のメールボックスにチェックポイントを設定します。

IMAP4.**close** ()

現在選択されているメールボックスを閉じます。削除されたメッセージは書き込み可能メールボックスから除去されます。LOGOUT 前に実行することを勧めます。

IMAP4.**copy** (*message_set*, *new_mailbox*)

message_set で指定したメッセージ群を *new_mailbox* の末尾にコピーします。

IMAP4.**create** (*mailbox*)

mailbox と名づけられた新たなメールボックスを生成します。

IMAP4.**delete** (*mailbox*)

mailbox と名づけられた古いメールボックスを削除します。

IMAP4.**deleteacl** (*mailbox*, *who*)

mailbox における *who* についての ACL を削除 (権限を削除) します。

バージョン 2.4 で追加.

IMAP4.expunge()

選択されたメールボックスから削除された要素を永久に除去します。各々の削除されたメッセージに対して、EXPUNGE 応答を生成します。返されるデータには EXPUNGE メッセージ番号を受信した順番に並べたリストが入っています。

IMAP4.fetch (*message_set*, *message_parts*)

メッセージ (の一部) を取りよせます。 *message_parts* はメッセージパートの名前を表す文字列を丸括弧で囲ったもので、例えば: "(UID BODY[TEXT])" のようになります。返されるデータはメッセージパートのエンベロープ情報とデータからなるタプルです。

IMAP4.getacl (*mailbox*)

mailbox に対する ACL を取得します。このメソッドは非標準ですが、Cyrus サーバでサポートされています。

IMAP4.getannotation (*mailbox*, *entry*, *attribute*)

mailbox に対する ANNOTATION を取得します。このメソッドは非標準ですが、Cyrus サーバでサポートされています。

バージョン 2.5 で追加.

IMAP4.getquota (*root*)

quota root により、リソース使用状況と制限値を取得します。このメソッドは [RFC 2087](#) で定義されている IMAP4 QUOTA 拡張の一部です。

バージョン 2.3 で追加.

IMAP4.getquotaroot (*mailbox*)

mailbox に対して *quota root* を実行した結果のリストを取得します。このメソッドは [RFC 2087](#) で定義されている IMAP4 QUOTA 拡張の一部です。

バージョン 2.3 で追加.

IMAP4.list ([*directory*[, *pattern*]])

pattern にマッチする *directory* メールボックス名を列挙します。 *directory* の標準の設定値は最上レベルのメールフォルダで、 *pattern* は標準の設定では全てにマッチします。返されるデータには LIST 応答のリストが入っています。

IMAP4.login (*user*, *password*)

平文パスワードを使ってクライアントを照合します。 *password* はクオートされます。

IMAP4.login_cram_md5 (*user*, *password*)

パスワードの保護のため、クライアント認証時に CRAM-MD5 だけを使用します。これは、CAPABILITY レスポンスに AUTH=CRAM-MD5 が含まれる場合のみ有効です。

バージョン 2.3 で追加.

IMAP4.logout ()

サーバへの接続を遮断します。サーバからの BYE 応答を返します。

IMAP4.lsub ([*directory*[, *pattern*]])

購読しているメールボックス名のうち、ディレクトリ内でパターンにマッチするものを列挙します。

directory の標準の設定値は最上レベルのメールフォルダで、*pattern* は標準の設定では全てにマッチします。返されるデータには返されるデータはメッセージパートエンベロープ情報とデータからなるタプルです。

IMAP4.**myrights** (*mailbox*)

mailbox における自分の ACL を返します (すなわち自分が *mailbox* で持っている権限を返します)。

バージョン 2.4 で追加.

IMAP4.**namespace** ()

RFC2342 で定義される IMAP 名前空間を返します。

バージョン 2.3 で追加.

IMAP4.**noop** ()

サーバに NOOP を送信します。

IMAP4.**open** (*host*, *port*)

host 上の *port* に対するソケットを開きます。このメソッドは *IMAP4* のコンストラクタから暗黙的に呼び出されます。このメソッドで確立された接続オブジェクトは *IMAP4.read()*, *IMAP4.readline()*, *IMAP4.send()*, *IMAP4.shutdown()* メソッドで使われます。このメソッドはオーバーライドすることができます。

IMAP4.**partial** (*message_num*, *message_part*, *start*, *length*)

メッセージの後略された部分を取り寄せます。返されるデータはメッセージパートエンベロープ情報とデータからなるタプルです。

IMAP4.**proxyauth** (*user*)

user として認証されたものとします。認証された管理者がユーザの代理としてメールボックスにアクセスする際に使用します。

バージョン 2.3 で追加.

IMAP4.**read** (*size*)

遠隔のサーバから *size* バイト読み出します。このメソッドはオーバーライドすることができます。

IMAP4.**readline** ()

遠隔のサーバから一行読み出します。このメソッドはオーバーライドすることができます。

IMAP4.**recent** ()

サーバに更新を促します。新たなメッセージがない場合応答は `None` になり、そうでない場合 `RECENT` 応答の値になります。

IMAP4.**rename** (*oldmailbox*, *newmailbox*)

oldmailbox という名前のメールボックスを *newmailbox* に名称変更します。

IMAP4.**response** (*code*)

応答 *code* を受信していれば、そのデータを返し、そうでなければ `None` を返します。通常の形式 (usual type) ではなく指定したコードを返します。

IMAP4.**search** (*charset*, *criterion*[, ...])

条件に合致するメッセージをメールボックスから検索します。*charset* は `None` でもよく、この場合にはサーバへの要求内に `CHARSET` は指定されません。IMAP プロトコルは少なくとも一つの条件 (*criterion*) が指定されるよう要求しています; サーバがエラーを返した場合、例外が送出されます。

例:

```
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

IMAP4.**select** ([*mailbox*[, *readonly*]])

メールボックスを選択します。返されるデータは *mailbox* 内のメッセージ数 (`EXISTS` 応答) です。標準の設定では *mailbox* は `'INBOX'` です。*readonly* が設定された場合、メールボックスに対する変更はできません。

IMAP4.**send** (*data*)

遠隔のサーバに *data* を送信します。このメソッドはオーバーライドすることができます。

IMAP4.**setacl** (*mailbox*, *who*, *what*)

ACL を *mailbox* に設定します。このメソッドは非標準ですが、Cyrus サーバでサポートされています。

IMAP4.**setannotation** (*mailbox*, *entry*, *attribute*[, ...])

ANNOTATION を *mailbox* に設定します。このメソッドは非標準ですが、Cyrus サーバでサポートされています。

バージョン 2.5 で追加.

IMAP4.**setquota** (*root*, *limits*)

quota *root* のリソースを *limits* に設定します。このメソッドは [RFC 2087](#) で定義されている IMAP4 QUOTA 拡張の一部です。

バージョン 2.3 で追加.

IMAP4.**shutdown** ()

`open` で確立された接続を閉じます。 `IMAP4.logout()` は暗黙的にこのメソッドを呼び出します。このメソッドはオーバーライドすることができます。

IMAP4.**socket** ()

サーバへの接続に使われているソケットインスタンスを返します。

IMAP4.**sort** (*sort_criteria*, *charset*, *search_criterion*[, ...])

`sort` 命令は `search` に結果の並べ替え (`sort`) 機能をつけた変種です。返されるデータには、条件に合致するメッセージ番号をスペースで分割したリストが入っています。

`sort` 命令は *search_criterion* の前に二つの引数を持ちます; *sort_criteria* のリストを丸括弧で囲ったものと、検索時の *charset* です。 `search` と違って、検索時の *charset* は必須です。 `uid sort` 命令もあり、`search` に対する `uid search` と同じように `sort` 命令に対応します。 `sort` 命令はまず、*charset*

引数の指定に従って `searching criteria` の文字列を解釈し、メールボックスから与えられた検索条件に合致するメッセージを探します。次に、合致したメッセージの数を返します。

IMAP4rev1 拡張命令です。

IMAP4.**status** (*mailbox, names*)

mailbox の指定ステータス名の状態情報を要求します。

IMAP4.**store** (*message_set, command, flag_list*)

メールボックス内のメッセージ群のフラグ設定を変更します。 *command* は RFC 2060 のセクション 6.4.6 で指定されているもので、"FLAGS", "+FLAGS", あるいは "-FLAGS" のいずれかとなります。オプションで末尾に ".SILENT" がつくこともあります。

たとえば、すべてのメッセージに削除フラグを設定するには次のようにします:

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    M.store(num, '+FLAGS', '\\Deleted')
M.expunge()
```

IMAP4.**subscribe** (*mailbox*)

新たなメールボックスを購読 (subscribe) します。

IMAP4.**thread** (*threading_algorithm, charset, search_criterion[, ...]*)

`thread` コマンドは `search` にスレッドの概念を加えた変形版です。返されるデータは空白で区切られたスレッドメンバのリストを含んでいます。

各スレッドメンバは 0 以上のメッセージ番号からなり、空白で区切られており、親子関係を示しています。

`thread` コマンドは *search_criterion* 引数の前に 2 つの引数を持っています。 *threading_algorithm* と *charset* です。 `search` コマンドとは違い、 *charset* は必須です。 `search` に対する `uid search` と同様に、 `thread` にも `uid thread` があります。 `thread` コマンドはまずメールボックス中のメッセージを、 *charset* を用いた検索条件で検索します。その後マッチしたメッセージを指定されたスレッドアルゴリズムでスレッド化して返します。

IMAP4rev1 拡張命令です。

バージョン 2.4 で追加。

IMAP4.**uid** (*command, arg[, ...]*)

command args を、メッセージ番号ではなく UID で指定されたメッセージ群に対して実行します。命令内容に応じた応答を返します。少なくとも一つの引数を与えなくてはなりません; 何も与えない場合、サーバはエラーを返し、例外が送出されます。

IMAP4.**unsubscribe** (*mailbox*)

古いメールボックスの購読を解除 (unsubscribe) します。

IMAP4.**xatom** (*name[, arg[, ...]]*)

サーバから CAPABILITY 応答で通知された単純な拡張命令を許容 (allow) します。

`IMAP4_SSL` のインスタンスは追加のメソッドを一つだけ持ちます:

`IMAP4_SSL.ssl()`

サーバへの安全な接続に使われる `SSLObject` インスタンスを返します。

以下の属性が `IMAP4` のインスタンス上で定義されています:

`IMAP4.PROTOCOL_VERSION`

サーバから返された `CAPABILITY` 応答にある、サポートされている最新のプロトコルです。

`IMAP4.debug`

デバッグ出力を制御するための整数値です。初期値はモジュール変数 `Debug` から取られます。3 以上の値にすると各命令をトレースします。

20.10.2 IMAP4 の使用例

以下にメールボックスを開き、全てのメッセージを取得して印刷する最小の (エラーチェックをしない) 使用例を示します:

```
import getpass, imaplib

M = imaplib.IMAP4()
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print 'Message %s\n%s\n' % (num, data[0][1])
M.close()
M.logout()
```

20.11 nntplib — NNTP プロトコルクライアント

Source code: [Lib/nntplib.py](#)

このモジュールでは、クラス `NNTP` を定義しています。このクラスは NNTP プロトコルのクライアント側を実装しています。このモジュールを使えば、ニュースリーダーや記事投稿プログラム、または自動的にニュース記事を処理するプログラムを実装することができます。NNTP (Network News Transfer Protocol、ネットニュース転送プロトコル) の詳細については、インターネット [RFC 977](#) を参照してください。

以下にこのモジュールの使い方の小さな例を二つ示します。ニュースグループに関する統計情報を列挙し、最新 10 件の記事を出力するには以下のようにします:

```
>>> s = NNTP('news.gmane.org')
>>> resp, count, first, last, name = s.group('gmane.comp.python.committers')
>>> print 'Group', name, 'has', count, 'articles, range', first, 'to', last
```

(次のページに続く)

(前のページからの続き)

```
Group gmane.comp.python.committers has 1071 articles, range 1 to 1071
>>> resp, subs = s.xhdr('subject', first + '-' + last)
>>> for id, sub in subs[-10:]: print id, sub
...
1062 Re: Mercurial Status?
1063 Re: [python-committers] (Windows) buildbots on 3.x
1064 Re: Mercurial Status?
1065 Re: Mercurial Status?
1066 Python 2.6.6 status
1067 Commit Privileges for Ask Solem
1068 Re: Commit Privileges for Ask Solem
1069 Re: Commit Privileges for Ask Solem
1070 Re: Commit Privileges for Ask Solem
1071 2.6.6 rc 2
>>> s.quit()
'205 Bye!'
```

ファイルから記事を投稿するには、以下のようにします。(この例では記事番号は有効な番号を指定していて、あなたがそのニュースグループに投稿する 権限を持っていると仮定しています)

```
>>> s = NNTP('news.gmane.org')
>>> f = open('articlefile')
>>> s.post(f)
'240 Article posted successfully.'
>>> s.quit()
'205 Bye!'
```

このモジュール自体では以下の内容を定義しています:

class `nntplib.NNTP` (`host` [, `port` [, `user` [, `password` [, `readermode`] [, `usenetr`]]])

ホスト `host` 上で動作し、ポート番号 `port` で要求待ちをしている NNTP サーバとの接続を表現する新たな `NNTP` クラスのインスタンスを返します。標準の `port` は 119 です。オプションの `user` および `password` が与えられているか、または `/.netrc` に適切な認証情報が指定されていて `usenetr` が真 (デフォルト) の場合、`AUTHINFO USER` および `AUTHINFO PASS` 命令を使ってサーバに対して身元証明および認証を行います。オプションのフラグ `readermode` が真の場合、認証の実行に先立って `mode reader` 命令が送信されます。reader モードは、ローカルマシン上の NNTP サーバに接続していて、`group` のような reader 特有の命令を呼び出したい場合に便利ことがあります。予期せず `NNTPPermanentError` に遭遇したなら、`readermode` を設定する必要があるかもしれません。`readermode` のデフォルト値は `None` です。`usenetr` のデフォルト値は `True` です。

バージョン 2.4 で変更: `usenetr` 引数を追加しました。

exception `nntplib.NNTPError`

標準の例外 `Exception` から派生しており、`nntplib` モジュールが送出する全ての例外の基底クラスです。

exception `nntplib.NNTPReplyError`

期待はずれの応答がサーバから返された場合に送出される例外です。以前のバージョンとの互換性のために、`error_reply` はこのクラスと等価になっています。

exception nntplib.NNTPTemporaryError

エラーコードの範囲が 400-499 のエラーを受信した場合に送出される例外です。以前のバージョンとの互換性のために、`error_temp` はこのクラスと等価になっています。

exception nntplib.NNTPPermanentError

エラーコードの範囲が 500-599 のエラーを受信した場合に送出される例外です。以前のバージョンとの互換性のために、`error_perm` はこのクラスと等価になっています。

exception nntplib.NNTPProtocolError

サーバから返される応答が 1-5 の範囲の数字で始まっていない場合に送出される例外です。以前のバージョンとの互換性のために、`error_proto` はこのクラスと等価になっています。

exception nntplib.NNTPDataError

応答データ中に何らかのエラーが存在する場合に送出される例外です。以前のバージョンとの互換性のために、`error_data` はこのクラスと等価になっています。

20.11.1 NNTP オブジェクト

NNTP インスタンスは以下のメソッドを持っています。全てのメソッドにおける戻り値のタプルで最初の要素となる *response* は、サーバの応答です: この文字列は 3 桁の数字からなるコードで始まります。サーバの応答がエラーを示す場合、上記のいずれかの例外が送出されます。

NNTP.getwelcome()

サーバに最初に接続した際に送信される応答中のウェルカムメッセージを返します。(このメッセージには時に、ユーザにとって重要な免責事項や ヘルプ情報が入っています。)

NNTP.set_debuglevel(*level*)

インスタンスのデバッグレベルを設定します。このメソッドは印字されるデバッグ出力の量を制御します。標準では 0 に設定されていて、これはデバッグ出力を全く印字しません。1 はそこそこの量、一般に NNTP 要求や応答あたり 1 行のデバッグ出力を生成します。値が 2 やそれ以上の場合、(メッセージテキストを含めて) NNTP 接続上で送受信された全ての内容を一行ごとにログ出力する、最大限のデバッグ出力を生成します。

NNTP.newgroups(*date*, *time*[, *file*])

NEWSGROUPS 命令を送信します。*date* 引数は 'yyymmdd' の形式を取り、日付を表します。*time* 引数は 'hhmmss' の形式をとり、時刻を表します。与えられた日付および時刻以後新たに出現したニュースグループ名のリストを *groups* として、(*response*, *groups*) を返します。*file* 引数が指定されている場合、NEWSGROUPS コマンドの出力結果はファイルに格納されます。*file* が文字列の場合、この文字列をファイル名としてファイルをオープンし、書き込み後にクローズします。*file* がファイルオブジェクトの場合、オブジェクトの `write()` メソッドを呼び出して出力結果を格納します。*file* が指定されている場合は戻り値として空のリストを返します。

NNTP.newnews(*group*, *date*, *time*[, *file*])

NEWNEWS 命令を送信します。ここで、*group* はグループ名または '*' で、*date* および *time* は `newsgroups()` における引数と同じ意味を持ちます。(*response*, *articles*) からなるペアを返し、*articles* はメッセージ ID のリストです。*file* 引数が指定されている場合、NEWNEWS コマンドの

出力結果は ファイルに格納されます。 *file* が文字列の場合、この文字列をファイル名としてファイルをオープンし、書き込み後にクローズします。 *file* がファイルオブジェクトの場合、オブジェクトの `write()` メソッドを呼び出して出力結果を格納します。 *file* が指定されている場合は戻り値として空のリストを返します。

`NNTP.list([file])`

`LIST` 命令を送信します。 `(response, list)` からなるペアを返します。 *list* はタプルからなるリストです。各タプルは `(group, last, first, flag)` の形式をとり、*group* がグループ名、*last* および *first* はそれぞれ最新および最初の記事の記事番号 (を表す文字列)、そして *flag* は投稿が可能な場合には `'y'`、そうでない場合には `'n'`、グループがモデレート (moderated) されている場合には `'m'` となります。(順番に注意してください: *last, first* の順です。) *file* 引数が指定されている場合、`LIST` コマンドの出力結果はファイルに格納されます。 *file* が文字列の場合、この文字列をファイル名としてファイルをオープンし、書き込み後にクローズします。 *file* がファイルオブジェクトの場合、オブジェクトの `write()` メソッドを呼び出して出力結果を格納します。 *file* が指定されている場合は戻り値として空のリストを返します。

`NNTP.descriptions(grouppattern)`

`LIST NEWSGROUPS` 命令を送信します。 *grouppattern* は RFC2980 の定義に従う wildmat 文字列です (実際には、DOS や UNIX のシェルワイルドカード文字列と同じです)。 `(response, list)` からなるペアを返し、*list* はタプル `(name, title)` リストになります。

バージョン 2.4 で追加.

`NNTP.description(group)`

単一のグループ *group* から説明文字列を取り出します。 (`'group'` が実際には wildmat 文字列で) 複数のグループがマッチした場合、最初にマッチしたものを返します。何もマッチしなければ空文字列を返します。

このメソッドはサーバからの応答コードを省略します。応答コードが必要なら、`descriptions()` を使ってください。

バージョン 2.4 で追加.

`NNTP.group(name)`

`GROUP` 命令を送信します。 *name* はグループ名です。タプル `(response, count, first, last, name)` を返します。 *count* はグループ中の記事数 (の推定値) で、*first* はグループ中の最初の記事番号、*last* はグループ中の 最新の記事番号、*name* はグループ名です。記事番号は文字列で返されます。

`NNTP.help([file])`

`HELP` 命令を送信します。 `(response, list)` からなるペアを返します。 *list* はヘルプ文字列からなるリストです。 *file* 引数が指定されている場合、`HELP` コマンドの出力結果はファイルに格納されます。 *file* が文字列の場合、この文字列をファイル名としてファイルをオープンし、書き込み後にクローズします。 *file* がファイルオブジェクトの場合、オブジェクトの `write()` メソッドを呼び出して出力結果を格納します。 *file* が指定されている場合は戻り値として空のリストを返します。

`NNTP.stat(id)`

`STAT` 命令を送信します。 *id* は (`'<'` と `'>'` に囲まれた形式の) メッセージ ID か、(文字列の) 記事

番号です。三つ組み (*response*, *number*, *id*) を返します。 *number* は (文字列の) 記事番号で、 *id* は ('<' と '>' に囲まれた形式の) メッセージ ID です。

`NNTP.next()`

NEXT 命令を送信します。 `stat()` のような応答を返します。

`NNTP.last()`

LAST 命令を送信します。 `stat()` のような応答を返します。

`NNTP.head(id)`

HEAD 命令を送信します。 *id* は `stat()` におけるのと同じ意味を持ちます。 (*response*, *number*, *id*, *list*) からなるタプルを返します。最初の 3 要素は `stat()` と同じもので、 *list* は記事のヘッダからなるリスト (まだ解析されておらず、末尾の改行が取り去られたヘッダ行のリスト) です。

`NNTP.body(id[, file])`

BODY 命令を送信します。 *id* は `stat()` におけるのと同じ意味を持ちます。 *file* 引数が与えられている場合、記事本体 (body) はファイルに保存されます。 *file* が文字列の場合、このメソッドはその名前を持つファイルオブジェクトを開き、記事を書き込んで閉じます。 *file* がファイルオブジェクトの場合、 `write()` を呼び出して記事本体を記録します。 `head()` のような戻り値を返します。 *file* が与えられていた場合、返される *list* は空のリストになります。

`NNTP.article(id)`

ARTICLE 命令を送信します。 *id* は `stat()` におけるのと同じ意味を持ちます。 `head()` のような戻り値を返します。

`NNTP.slave()`

SLAVE 命令を送信します。サーバの *response* を返します。

`NNTP.xhdr(header, string[, file])`

XHDR 命令を送信します、この命令は RFC には定義されていませんが、一般に広まっている拡張です。 *header* 引数は、例えば 'subject' といったヘッダキーワードです。 *string* 引数は 'first-last' の形式でなければならず、ここで *first* および *last* は検索の対象とする記事範囲の最初と最後の記事番号です。 (*response*, *list*) のペアを返します。 *list* は (*id*, *text*) のペアからなるリストで、 *id* が (文字列で表した) 記事番号、 *text* がその記事のヘッダテキストです。 *file* 引数が指定されている場合、XHDR コマンドの出力結果はファイルに格納されます。 *file* が文字列の場合、この文字列をファイル名としてファイルをオープンし、書き込み後にクローズします。 *file* がファイルオブジェクトの場合、オブジェクトの `write()` メソッドを呼び出して出力結果を格納します。 *file* が指定されている場合は戻り値として空のリストを返します。

`NNTP.post(file)`

POST 命令を使って記事をポストします。 *file* 引数は開かれているファイルオブジェクトで、その内容は `readline()` メソッドを使って EOF まで読み出されます。内容は必要なヘッダを含め、正しい形式のニュース記事でなければなりません。 `post()` メソッドは . で始まる行を自動的にエスケープします。

`NNTP.ihave(id, file)`

IHAVE 命令を送信します。 *id* は ('<' と '>' に囲まれた) メッセージ ID です。 応答がエラーでない

場合、*file* を *post()* と全く同じように扱います。

NNTP.date()

タプル (response, date, time) を返します。このタプルには *newnews()* および *newgroups()* メソッドに合った形式の、現在の日付および時刻が入っています。これはオプションの NNTP 拡張なので、全てのサーバでサポートされているとは限りません。

NNTP.xgtitle(name[, file])

XGTITLE 命令を処理し、(response, list) からなるペアを返します。list は (name, title) を含むタプルのリストです。file 引数が指定されている場合、XHDR コマンドの出力結果はファイルに格納されます。file が文字列の場合、この文字列をファイル名としてファイルをオープンし、書き込み後にクローズします。file がファイルオブジェクトの場合、オブジェクトの write() メソッドを呼び出して出力結果を格納します。file が指定されている場合は戻り値として空のリストを返します。これはオプションの NNTP 拡張なので、全てのサーバでサポートされているとは限りません。

RFC2980 では、“この拡張は撤廃すべきである”と主張しています。descriptions() または description() を使うようにしてください。

NNTP.xover(start, end[, file])

(resp, list) からなるペアを返します。list はタプルからなるリストで、各タプルは記事番号 start および end の間に区切られた記事です。各タプルは (article number, subject, poster, date, id, references, size, lines) の形式をとります。file 引数が指定されている場合、XHDR コマンドの出力結果はファイルに格納されます。file が文字列の場合、この文字列をファイル名としてファイルをオープンし、書き込み後にクローズします。file がファイルオブジェクトの場合、オブジェクトの write() メソッドを呼び出して出力結果を格納します。file が指定されている場合は戻り値として空のリストを返します。これはオプションの NNTP 拡張なので、全てのサーバでサポートされているとは限りません。

NNTP.xpath(id)

(resp, path) からなるペアを返します。path はメッセージ ID が id である記事のディレクトリパスです。これはオプションの NNTP 拡張なので、全てのサーバでサポートされているとは限りません。

NNTP.quit()

QUIT 命令を送信し、接続を閉じます。このメソッドを呼び出した後は、NNTP オブジェクトの他のいかなるメソッドも呼び出してはいけません。

20.12 smtplib — SMTP プロトコルクライアント

ソースコード: [Lib/smtplib.py](#)

smtplib モジュールは、SMTP または ESMTP のリスナーデーモンを備えた任意のインターネット上のホストにメールを送るために使用することができる SMTP クライアント・セッション・オブジェクトを定義します。SMTP および ESMTP オペレーションの詳細は、[RFC 821](#) (Simple Mail Transfer Protocol) や [RFC 1869](#) (SMTP Service Extensions) を調べてください。

```
class smtplib.SMTP ([host[, port[, local_hostname[, timeout ]]])
```

SMTP インスタンスは SMTP コネクションをカプセル化し、SMTP と ESMTP の命令をサポートをします。オプションである *host* と *port* を与えた場合は、SMTP クラスのインスタンスが作成されると同時に、*connect()* メソッドを呼び出し初期化されます。*connect()* 呼び出しが成功コード以外を返した場合、*SMTPConnectError* 例外が送出されます。*local_hostname* を指定する場合、これは HELO/EHLO コマンドでのローカルホストの FQDN として使われます。指定しない場合はローカルホスト名は *socket.getfqdn()* が使われます。オプションの *timeout* 引数を与える場合、コネクションの接続時などのブロックする操作におけるタイムアウト時間を秒数で設定します。(指定されなかった場合は、グローバルのデフォルトタイムアウト設定が利用されます)。タイムアウトに達すると *socket.timeout* 例外が送出されます。

For normal use, you should only require the initialization/connect, *sendmail()*, and *SMTP.quit()* methods. An example is included below.

バージョン 2.6 で変更: *timeout* が追加されました

```
class smtplib.SMTP_SSL ([host[, port[, local_hostname[, keyfile[, certfile[, timeout ]]])
```

SMTP_SSL のインスタンスは *SMTP* のインスタンスと全く同じように動作します。*SMTP_SSL* は、接続の最初の段階から SSL が要求され、*starttls()* では対応できない場合にのみ利用されるべきです。*host* が指定されなかった場合は、localhost が利用されます。*port* が指定されなかった場合は、標準の SMTP-over-SSL ポート (465) が利用されます。*local_hostname* は *SMTP* クラスのものと同じです。*keyfile* と *certfile* もオプションで、SSL 接続のための、PEM フォーマットのプライベートキーと、証明パス (certificate chain) ファイルを指定することができます。オプションの *timeout* 引数を与える場合、コネクションの接続時などのブロックする操作におけるタイムアウト時間を秒数で設定します (指定されなかった場合は、グローバルのデフォルトタイムアウト設定が利用されます)。タイムアウトに達すると *socket.timeout* 例外が送出されます。

バージョン 2.6 で追加.

```
class smtplib.LMTP ([host[, port[, local_hostname ]]])
```

ESMTP に非常に似ている LMTP プロトコルは、SMTP クライアントに基づいています。LMTP にはよく Unix ソケットが利用されるので、*connect()* メソッドは通常の *host:port* サーバーと同じように Unix ソケットもサポートしています。*local_hostname* は *SMTP* クラスのものと同じです。Unix ソケットを指定するには、*host* 引数に、*'/'* で始まる絶対パスを指定します。

認証は、通常の SMTP 機構を利用してサポートされています。Unix ソケットを利用する場合、LMTP は通常認証をサポートしたり要求したりはしません。しかし、あなたが必要であれば、利用することができます。

バージョン 2.6 で追加.

このモジュールの例外には次のものがあります:

```
exception smtplib.SMTPException
```

このモジュールで提供される全ての例外クラスのベースクラスです。

```
exception smtplib.SMTPServerDisconnected
```

この例外はサーバが突然コネクションを切断するか、もしくは *SMTP* インスタンスを生成する前にコネクションを張ろうとした場合に上げられます。

exception `smtpplib.SMTPResponseException`

SMTP のエラーコードを含んだ例外のクラスです。これらの例外は SMTP サーバがエラーコードを返すときに生成されます。エラーコードは `smtp_code` 属性に格納されます。また、`smtp_error` 属性にはエラーメッセージが格納されます。

exception `smtpplib.SMTPSenderRefused`

送信者のアドレスが弾かれたときに上げられる例外です。全ての `SMTPResponseException` 例外に、SMTP サーバが弾いた 'sender' アドレスの文字列がセットされます。

exception `smtpplib.SMTPRecipientsRefused`

全ての受取人アドレスが弾かれたときに上げられる例外です。各受取人のエラーは属性 `recipients` によってアクセス可能で、`SMTP.sendmail()` が返す辞書と同じ並びの辞書になっています。

exception `smtpplib.SMTPDataError`

SMTP サーバが、メッセージのデータを受け入れることを拒絶した時に上げられる例外です。

exception `smtpplib.SMTPConnectError`

サーバへの接続時にエラーが発生した時に上げられる例外です。

exception `smtpplib.SMTPHeloError`

サーバーが HELO メッセージを弾いた時に上げられる例外です。

exception `smtpplib.SMTPAuthenticationError`

SMTP 認証が失敗しました。最もあり得る可能性は、サーバーがユーザ名/パスワードのペアを受付なかった事です。

参考:

RFC 821 - Simple Mail Transfer Protocol SMTP のプロトコル定義です。このドキュメントでは SMTP のモデル、操作手順、プロトコルの詳細についてカバーしています。

RFC 1869 - SMTP Service Extensions SMTP に対する ESMTP 拡張の定義です。このドキュメントでは、新たな命令による SMTP の拡張、サーバによって提供される命令を動的に発見する機能のサポート、およびいくつかの追加命令定義について記述しています。

20.12.1 SMTP オブジェクト

`SMTP` クラスインスタンスは次のメソッドを提供します:

SMTP.set_debuglevel (*level*)

コネクション間でやりとりされるメッセージ出力のレベルをセットします。メッセージの冗長さは *level* に応じて決まります。

SMTP.docmd (*cmd* [, *argstring*])

サーバへコマンド *cmd* を送信します。オプション引数 *argstring* はスペース文字でコマンドに連結します。

戻り値は、整数値のレスポンスコードと、サーバからの応答の値をタプルで返します。(サーバからの応答が数行に渡る場合でも一つの大きな文字列で返します。)

通常、この命令を明示的に使う必要はありませんが、自分で拡張する時に使用するときに役立つかもしれません。

応答待ちのときに、サーバへのコネクションが失われると、`SMTPServerDisconnected` が上がります。

`SMTP.connect([host[, port]])`

ホスト名とポート番号をもとに接続します。デフォルトは `localhost` の標準的な SMTP ポート (25 番) に接続します。もしホスト名の末尾がコロン (':') で、後に番号がついている場合は、「ホスト名:ポート番号」として扱われます。このメソッドはコンストラクタにホスト名及びポート番号が指定されている場合、自動的に呼び出されます。戻り値は、この接続の応答内でサーバによって送信された応答コードとメッセージの 2 要素タプルです。

`SMTP.helo([hostname])`

SMTP サーバに HELO コマンドで身元を示します。デフォルトでは `hostname` 引数はローカルホストを指します。サーバが返したメッセージは、オブジェクトの `helo_resp` 属性に格納されます。

通常は `sendmail()` が呼び出すため、これを明示的に呼び出す必要はありません。

`SMTP.ehlo([hostname])`

EHLO を利用し、ESMTP サーバに身元を明かします。デフォルトでは `hostname` 引数はローカルホストの FQDN です。また、ESMTP オプションのために応答を調べたものは、`has_extn()` に備えて保存されます。また、幾つかの情報を属性に保存します: サーバが返したメッセージは `ehlo_resp` 属性に、`does_esmtp` 属性はサーバが ESMTP をサポートしているかどうかによって `true` か `false` に、`esmtp_features` 属性は辞書で、サーバが対応している SMTP サービス拡張の名前と、もしあればそのパラメータを格納します。

`has_extn()` をメールを送信する前に使わない限り、明示的にこのメソッドを呼び出す必要があるべきではなく、`sendmail()` が必要とした場合に呼ばれます。

`SMTP.ehlo_or_helo_if_needed()`

このメソッドは、現在のセッションでまだ EHLO か HELO コマンドが実行されていない場合、`ehlo()` and/or `helo()` メソッドを呼び出します。このメソッドは先に ESMTP EHLO を試します。

`SMTPHelloError` サーバが HELO に正しく返事しなかった。

バージョン 2.6 で追加。

`SMTP.has_extn(name)`

`name` が拡張 SMTP サービスセットに含まれている場合には `True` を返し、そうでなければ `False` を返します。大小文字は区別されません。

`SMTP.verify(address)`

VRFY を利用して SMTP サーバにアドレスの妥当性をチェックします。妥当である場合はコード 250 と完全な RFC 822 アドレス (人名) のタプルを返します。それ以外の場合は、400 以上のエラーコードとエラー文字列を返します。

注釈: ほとんどのサイトはスパマーの裏をかくために SMTP の VRFY は使用不可になっています。

`SMTP.login(user, password)`

認証が必要な SMTP サーバにログインします。認証に使用する引数はユーザ名とパスワードです。まだセッションが無い場合は、`EHLO` または `HELO` コマンドでセッションを作ります。ESMTP の場合は `EHLO` が先に試されます。認証が成功した場合は通常このメソッドは戻りますが、例外が起こった場合は以下の例外が上がります:

`SMTPHeloError` サーバーが `HELO` に正しく返事しなかった。

`SMTPAuthenticationError` サーバがユーザ名/パスワードでの認証に失敗した。

`SMTPException` どんな認証方法も見付からなかった。

`SMTP.starttls([keyfile[, certfile]])`

TLS(Transport Layer Security) モードで SMTP コネクションを出し、全ての SMTP コマンドは暗号化されます。これは `ehlo()` をもう一度呼び出すときにすべきです。

`keyfile` と `certfile` が提供された場合に、`socket` モジュールの `ssl()` 関数が通るようになります。

もしまだ `EHLO` か `HELO` コマンドが実行されていない場合、このメソッドは ESMTP `EHLO` を先に試します。

バージョン 2.6 で変更。

`SMTPHeloError` サーバーが `HELO` に正しく返事しなかった。

`SMTPException` サーバーが STARTTLS 拡張に対応していない。

バージョン 2.6 で変更。

`RuntimeError` 実行中の Python インタプリタで、SSL/TLS サポートが利用できない。

`SMTP.sendmail(from_addr, to_addrs, msg[, mail_options, rcpt_options])`

メールを送信します。必要な引数は **RFC 822** の `from` アドレス文字列、**RFC 822** の `to` アドレス文字列またはアドレス文字列のリスト、メッセージ文字列です。送信側は `MAIL FROM` コマンドで使われる `mail_options` の ESMTP オプション (8bitmime のような) のリストを得るかもしれません。全ての `RCPT` コマンドで使われるべき ESMTP オプション (例えば `DSN` コマンド) は、`rcpt_options` を通して利用することができます。(もし送信先別に ESMTP オプションを使う必要があれば、メッセージを送るために `mail()`、`rcpt()`、`data()` といった下位レベルのメソッドを使う必要があります。)

注釈: 配送エージェントは `from_addr`、`to_addrs` 引数を使い、メッセージのエンベロープを構成します。`SMTP` はメッセージヘッダを修正しません。

まだセッションが無い場合は、`EHLO` または `HELO` コマンドでセッションを作ります。ESMTP の場合は `EHLO` が先に試されます。また、サーバが ESMTP 対応ならば、メッセージサイズとそれぞれ指定されたオプションも渡します。(feature オプションがあればサーバの広告をセットします) `EHLO` が失敗した場合は、ESMTP オプションの無い `HELO` が試されます。

このメソッドは最低でも 1 つの受信者にメールが受け入れられたときは普通に返りますが、そうでない場合は例外を投げます。このメソッドが例外を投げられなければ、誰かが送信したメールを得るべきで

す。また、例外を投げられなかった場合は、拒絶された受取人ごとへの 1 つのエントリーと共に、辞書を返します。各エントリーは、サーバーによって送られた SMTP エラーコードおよびエラーメッセージのタプルを含んでいます。

このメソッドは次の例外を上げることがあります:

SMTPRecipientsRefused 全ての受信を拒否され、誰にもメールが届けられませんでした。例外オブジェクトの `recipients` 属性は、受信拒否についての情報の入った辞書オブジェクトです。(辞書は少なくとも一つは受信されたときに似ています)。

SMTPHeloError サーバーが HELO に正しく返事しなかった。

SMTPSenderRefused サーバが `from_addr` を受理しませんでした。

SMTPDataError サーバが予期しないエラーコードを返しました (受信拒否以外)。

また、この他の注意として、例外が上がった後もコネクションは開いたままになっています。

`SMTP.quit()`

SMTP セッションを終了し、コネクションを閉じます。SMTP QUIT コマンドの結果を返します。

バージョン 2.6 で変更: 結果を返すようになりました

下位レベルのメソッドは標準 SMTP/ESMTP コマンド HELP、RSET、NOOP、MAIL、RCPT、DATA に対応しています。通常これらは直接呼ぶ必要はなく、また、ドキュメントもありません。詳細はモジュールのコードを調べてください。

20.12.2 SMTP 使用例

次の例は最低限必要なメールアドレス ('To' と 'From') を含んだメッセージを送信するものです。この例では RFC 822 ヘッダの加工もしていません。メッセージに含まれるヘッダは、メッセージに含まれる必要があり、特に、明確な 'To'、と 'From' アドレスはメッセージヘッダに含まれている必要があります。

```
import smtplib

def prompt(prompt):
    return raw_input(prompt).strip()

fromaddr = prompt("From: ")
toaddrs = prompt("To: ").split()
print "Enter message, end with ^D (Unix) or ^Z (Windows):"

# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\n\r\n"
       % (fromaddr, ", ".join(toaddrs)))
while 1:
    try:
        line = raw_input()
    except EOFError:
        break
```

(次のページに続く)

(前のページからの続き)

```
if not line:
    break
msg = msg + line

print "Message length is " + repr(len(msg))

server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

注釈: 多くの場合、*email* パッケージの機能を使って email メッセージを構築し、それを文字列に変換して、`sendmail()` で送信する、という手順を用います。 *email: 使用例* を参照してください。

20.13 smtpd — SMTP サーバー

Source code: [Lib/smtpd.py](#)

このモジュールでは、SMTP サーバを実装するためのクラスをいくつか提供しています。一つは何も行わない、オーバーライドできる汎用のサーバで、その他の二つでは特定のメール送信ストラテジを提供しています。

20.13.1 SMTPServer オブジェクト

class `smtpd.SMTPServer` (*localaddr*, *remoteaddr*)

新たな *SMTPServer* オブジェクトを作成し、それをローカルのアドレス *localaddr* に bind します。このオブジェクトは *remoteaddr* を上流の SMTP リレー先とします。 *localaddr* と *remoteaddr* のどちらも (*host*, *port*) タプルである必要があります。このクラスは *asyncore.dispatcher* を継承しており、インスタンス化時に自身を *asyncore* のイベントループに登録します。

process_message (*peer*, *mailfrom*, *rcpttos*, *data*)

このクラスでは *NotImplementedError* 例外を送出します。受信したメッセージを使って何か意味のある処理をしたい場合にはこのメソッドを オーバライドしてください。コンストラクタの *remoteaddr* に渡した値は `_remoteaddr` 属性で参照できます。 *peer* はリモートホストのアドレスで、 *mailfrom* はメッセージエンベロープの発信元 (envelope originator)、 *rcpttos* はメッセージエンベロープの受信対象、そして *data* は電子メールの内容が入った (**RFC 2822** 形式の) 文字列です。

20.13.2 DebuggingServer オブジェクト

class `smtpd.DebuggingServer` (*localaddr*, *remoteaddr*)

新たなデバッグ用サーバを生成します。引数は *SMTPServer* と同じです。メッセージが届いても無視し、標準出力に出力します。

20.13.3 PureProxy オブジェクト

class `smtpd.PureProxy` (*localaddr*, *remoteaddr*)

新たな単純プロキシ (pure proxy) サーバを生成します。引数は *SMTPServer* と同じです。全てのメッセージを *remoteaddr* にリレーします。このオブジェクトを動作させるとオープンリレーを作成してしまう可能性が多分にあります。注意してください。

20.13.4 MailmanProxy Objects

class `smtpd.MailmanProxy` (*localaddr*, *remoteaddr*)

新たな単純プロキシサーバを生成します。引数は *SMTPServer* と同じです。全てのメッセージを *remoteaddr* にリレーしますが、ローカルの mailman の設定に *remoteaddr* がある場合には mailman を使って処理します。このオブジェクトを動作させるとオープンリレーを作成してしまう可能性が多分にあります。注意してください。

20.14 telnetlib — Telnet クライアント

ソースコード: [Lib/telnetlib.py](#)

telnetlib モジュールでは、Telnet プロトコルを実装している *Telnet* クラスを提供します。Telnet プロトコルについての詳細は [RFC 854](#) を参照してください。加えて、このモジュールでは Telnet プロトコルにおける制御文字 (下を参照してください) と、telnet オプションに対するシンボル定数を提供しています。telnet オプションに対するシンボル名は `arpa/telnet.h` の `TELOPT_` がない状態での定義に従います。伝統的に `arpa/telnet.h` に含まれていない telnet オプションのシンボル名については、このモジュールのソースコード自体を参照してください。

telnet コマンドのシンボル定数は、IAC、DONT、DO、WONT、WILL、SE (サブネゴシエーション終了)、NOP (何もしない)、DM (データマーク)、BRK (ブレイク)、IP (プロセス割り込み)、AO (出力中断)、AYT (応答確認)、EC (文字削除)、EL (行削除)、GA (進め)、SB (サブネゴシエーション開始) です。

class `telnetlib.Telnet` (*[host[, port[, timeout]]]*)

Telnet represents a connection to a Telnet server. The instance is initially not connected by default; the *open()* method must be used to establish a connection. Alternatively, the host name and optional port number can be passed to the constructor, to, in which case the connection to the server will be established before the constructor returns. The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).

すでに接続の開かれているインスタンスを再度開いてはいけません。

このクラスは多くの *read_*()* メソッドを持っています。これらのメソッドのいくつかは、接続の終端を示す文字を読み込んだ場合に *EOFError* を送出するので注意してください。例外を送出するのは、これらの関数が終端に到達しなくても空の文字列を返す可能性があるからです。詳しくは下記の個々の説明を参照してください。

バージョン 2.6 で変更: *timeout* が追加されました。

参考:

RFC 854 - Telnet プロトコル仕様 Telnet プロトコルの定義。

20.14.1 Telnet オブジェクト

Telnet インスタンスは以下のメソッドを持っています:

`Telnet.read_until(expected[, timeout])`

expected で指定された文字列を読み込むか、*timeout* で指定された秒数が経過するまで読み込みます。

与えられた文字列に一致する部分が見つからなかった場合、読み込むことができたものを返します。これは空の文字列になる可能性があります。接続が閉じられ、転送処理済みのデータが得られない場合には *EOFError* が送出されます。

`Telnet.read_all()`

EOF に到達するまでの全てのデータを読み込みます; 接続が閉じられるまでブロックします。

`Telnet.read_some()`

EOF に到達しない限り、少なくとも 1 バイトの転送処理済みデータを読み込みます。EOF に到達した場合は `''` を返します。すぐに読み出せるデータが存在しない場合にはブロックします。

`Telnet.read_very_eager()`

I/O によるブロックを起こさずに読み出せる全てのデータを読み込みます (eager モード)。

接続が閉じられており、転送処理済みのデータとして読み出せるものがない場合には *EOFError* が送出されます。それ以外の場合で、単に読み出せるデータがない場合には `''` を返します。IAC シーケンス操作中でないかぎりブロックしません。

`Telnet.read_eager()`

現在すぐに読み出せるデータを読み出します。

接続が閉じられており、転送処理済みのデータとして読み出せるものがない場合には *EOFError* が送出されます。それ以外の場合で、単に読み出せるデータがない場合には `''` を返します。IAC シーケンス操作中でないかぎりブロックしません。

`Telnet.read_lazy()`

すでにキューに入っているデータを処理して返します (lazy モード)。

接続が閉じられており、読み出せるデータがない場合には *EOFError* を送出します。それ以外の場合で、転送処理済みのデータで読み出せるものがない場合には `''` を返します。IAC シーケンス操作中でないかぎりブロックしません。

`Telnet.read_very_lazy()`

すでに処理済みキューに入っているデータを処理して返します (very lazy モード)。

接続が閉じられており、読み出せるデータがない場合には *EOFError* を送出します。それ以外の場合で、転送処理済みのデータで読み出せるものがない場合には `''` を返します。このメソッドは決してブ

ロックしません。

`Telnet.read_sb_data()`

SB/SE ペア (サブオプション開始 / 終了) の間に収集されたデータを返します。SE コマンドによって起動されたコールバック関数はこれらのデータにアクセスしなければなりません。このメソッドは決してブロックしません。

バージョン 2.3 で追加。

`Telnet.open(host[, port[, timeout]])`

サーバホストに接続します。第二引数はオプションで、ポート番号を指定します。標準の値は通常の Telnet ポート番号 (23) です。オプション引数の *timeout* が渡された場合、コネクション接続時などのブロックする操作のタイムアウト時間を秒数で指定します (指定されなかった場合は、グローバルのデフォルトタイムアウト設定が利用されます)。

すでに接続しているインスタンスで再接続を試みてはいけません。

バージョン 2.6 で変更: *timeout* が追加されました。

`Telnet.msg(msg[, *args])`

デバッグレベルが > 0 のとき、デバッグ用のメッセージを出力します。追加の引数が存在する場合、標準の文字列書式化演算子 `%` を使って *msg* 中の書式指定子に代入されます。

`Telnet.set_debuglevel(debuglevel)`

デバッグレベルを設定します。 *debuglevel* が大きくなるほど、(`sys.stdout` に) デバッグメッセージがたくさん出力されます。

`Telnet.close()`

コネクションをクローズします。

`Telnet.get_socket()`

内部的に使われているソケットオブジェクトです。

`Telnet.fileno()`

内部的に使われているソケットオブジェクトのファイル記述子です。

`Telnet.write(buffer)`

ソケットに文字列を書き込みます。このとき IAC 文字については 2 度送信します。接続がブロックした場合、書き込みがブロックする可能性があります。接続が閉じられた場合、`socket.error` が送出されるかもしれません。

`Telnet.interact()`

非常に低機能の telnet クライアントをエミュレートする対話関数です。

`Telnet.mt_interact()`

`interact()` のマルチスレッド版です。

`Telnet.expect(list[, timeout])`

正規表現のリストのうちどれか一つにマッチするまでデータを読みます。

第一引数は正規表現のリストです。コンパイルされたもの (`regex objects`) でも、コンパイルされていないもの (文字列) でもかまいません。オプションの第二引数はタイムアウトで、単位は秒です; 標準の値は無期限に設定されています。

3 つの要素からなるタプル: 最初にマッチした正規表現のインデクス; 返されたマッチオブジェクト; マッチ部分を含む、マッチするまでに読み込まれたテキストデータ、を返します。

ファイル終了子が見つかり、かつ何もテキストデータが読み込まれなかった場合、`EOFError` が送出されます。そうでない場合で何もマッチしなかった場合には `(-1, None, text)` が返されます。ここで `text` はこれまで受信したテキストデータです (タイムアウトが発生した場合には空の文字列になる場合もあります)。

正規表現の末尾が `(.*` のような) 貪欲マッチングになっている場合や、入力に対して 1 つ以上の正規表現がマッチする場合には、その結果は決定不能で、I/O のタイミングに依存するでしょう。

`Telnet.set_option_negotiation_callback (callback)`

telnet オプションが入力フローから読み込まれるたびに、`callback` が (設定されていれば) 以下の引数形式: `callback(telnet socket, command (DO/DONT/WILL/WONT), option)` で呼び出されます。その後 telnet オプションに対しては `telnetlib` は何も行いません。

20.14.2 Telnet Example

典型的な使用例のサンプルコード:

```
import getpass
import sys
import telnetlib

HOST = "localhost"
user = raw_input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)

tn.read_until("login: ")
tn.write(user + "\n")
if password:
    tn.read_until("Password: ")
    tn.write(password + "\n")

tn.write("ls\n")
tn.write("exit\n")

print tn.read_all()
```

20.15 uuid — RFC 4122 に準拠した UUID オブジェクト

バージョン 2.5 で追加.

このモジュールでは immutable (変更不能) な `UUID` オブジェクト (`UUID` クラス) と [RFC 4122](#) の定めるバージョン 1、3、4、5 の UUID を生成するための `uuid1()`, `uuid2()`, `uuid3()`, `uuid4()`, `uuid5()`, が提供されています。

もしユニークな ID が必要なだけであれば、おそらく `uuid1()` か `uuid4()` をコールすれば良いでしょう。`uuid1()` はコンピュータのネットワークアドレスを含む UUID を生成するためにプライバシーを侵害するかもしれない点に注意してください。 `uuid4()` はランダムな UUID を生成します。

class `uuid.UUID` ([`hex` [, `bytes` [, `bytes_le` [, `fields` [, `int` [, `version`]]]]]])

Create a UUID from either a string of 32 hexadecimal digits, a string of 16 bytes in big-endian order as the `bytes` argument, a string of 16 bytes in little-endian order as the `bytes_le` argument, a tuple of six integers (32-bit `time_low`, 16-bit `time_mid`, 16-bit `time_hi_version`, 8-bit `clock_seq_hi_variant`, 8-bit `clock_seq_low`, 48-bit `node`) as the `fields` argument, or a single 128-bit integer as the `int` argument. When a string of hex digits is given, curly braces, hyphens, and a URN prefix are all optional. For example, these expressions all yield the same UUID:

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes='\x12\x34\x56\x78'*4)
UUID(bytes_le='\x78\x56\x34\x12\x34\x12\x78\x56' +
              '\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x567812345678))
UUID(int=0x12345678123456781234567812345678)
```

`hex`, `bytes`, `bytes_le`, `fields`, または `int` のうち、どれかただ一つだけが与えられなければいけません。`version` 引数はオプションです; 与えられた場合、結果の UUID は与えられた `hex`, `bytes`, `bytes_le`, `fields`, または `int` をオーバーライドして、RFC 4122 に準拠した variant と version ナンバーのセットを持つこととなります。`bytes_le`, `fields`, or `int`.

`UUID` インスタンスは以下の読み取り専用属性を持ちます:

`UUID.bytes`

16 バイト文字列 (バイトオーダーがビッグエンディアンの 6 つの整数フィールドを持つ) の UUID。

`UUID.bytes_le`

16 バイト文字列 (`time_low`, `time_mid`, `time_hi_version` をリトルエンディアンで持つ) の UUID。

`UUID.fields`

UUID の 6 つの整数フィールドを持つタプルで、これは 6 つの個別の属性と 2 つの派生した属性としても取得可能です:

フィールド	意味
<code>time_low</code>	UUID の最初の 32 ビット
<code>time_mid</code>	UUID の次の 16 ビット
<code>time_hi_version</code>	UUID の次の 16 ビット
<code>clock_seq_hi_variant</code>	UUID の次の 8 ビット
<code>clock_seq_low</code>	UUID の次の 8 ビット
<code>node</code>	UUID の最後の 48 ビット
<i>time</i>	60 ビットのタイムスタンプ
<code>clock_seq</code>	14 ビットのシーケンス番号

UUID.hex

32 文字の 16 進数文字列での UUID。

UUID.int

128 ビット整数での UUID。

UUID.urn

RFC 4122 で規定される URN での UUID。

UUID.variant

UUID の内部レイアウトを決定する UUID の variant。これは定数 *RESERVED_NCS*, *RFC_4122*, *RESERVED_MICROSOFT*, *RESERVED_FUTURE* のいずれかです。

UUID.version

UUID の version 番号 (1 から 5、variant が *RFC_4122* である場合だけ意味があります)。

The *uuid* モジュールには以下の関数があります:

uuid.getnode()

48 ビットの正の整数としてハードウェアアドレスを取得します。最初にこれを起動すると、別個のプログラムが立ち上がって非常に遅くなることがあります。もしハードウェアを取得する試みが全て失敗すると、ランダムな 48 ビットに RFC 4122 で推奨されているように 8 番目のビットを 1 に設定した数を使います。”ハードウェアアドレス”とはネットワークインターフェースの MAC アドレスを指し、複数のネットワークインターフェースを持つマシンの場合、それらのどれか一つの MAC アドレスが返るでしょう。

uuid.uuid1([node[, clock_seq]])

UUID をホスト ID、シーケンス番号、現在時刻から生成します。 *node* が与えられなければ、*getnode()* がハードウェアアドレス取得のために使われます。 *clock_seq* が与えられると、これはシーケンス番号として使われます; さもなくば 14 ビットのランダムなシーケンス番号が選ばれます。

uuid.uuid3(namespace, name)

UUID を名前空間識別子 (これは UUID です) と名前 (文字列です) の MD5 ハッシュから生成します。

uuid.uuid4()

ランダムな UUID を生成します。

`uuid.uuid5(namespace, name)`

名前空間識別子（これは UUID です）と名前（文字列です）の SHA-1 ハッシュから生成します。

`uuid` モジュールは `uuid3()` または `uuid5()` で利用するために次の名前空間識別子を定義しています。

`uuid.NAMESPACE_DNS`

この名前空間が指定された場合、`name` 文字列は完全修飾ドメイン名です。

`uuid.NAMESPACE_URL`

この名前空間が指定された場合、`name` 文字列は URL です。

`uuid.NAMESPACE_OID`

この名前空間が指定された場合、`name` 文字列は ISO OID です。

`uuid.NAMESPACE_X500`

この名前空間が指定された場合、`name` 文字列は X.500 DN の DER またはテキスト出力形式です。

The `uuid` モジュールは以下の定数を `variant` 属性が取りうる値として定義しています:

`uuid.RESERVED_NCS`

NCS 互換性のために予約されています。

`uuid.RFC_4122`

RFC 4122 で与えられた UUID レイアウトを指定します。

`uuid.RESERVED_MICROSOFT`

Microsoft の互換性のために予約されています。

`uuid.RESERVED_FUTURE`

将来のために予約されています。

参考:

RFC 4122 - Universally Unique Identifier (UUID) の URN 名前空間 この仕様は UUID のための Uniform Resource Name 名前空間、UUID の内部フォーマットと UUID の生成方法を定義しています。

20.15.1 例

典型的な `uuid` モジュールの利用方法を示します:

```
>>> import uuid

>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
```

(次のページに続く)

(前のページからの続き)

```
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')

>>> # make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
>>> x.bytes
'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')
```

20.16 urlparse — URL を解析して構成要素にする

注釈: `urlparse` モジュールは、Python 3 では `urllib.parse` にリネームされました。 *2to3* ツールが自動的にソースコードの `import` を修正します。

ソースコード: [Lib/urlparse.py](#)

このモジュールでは URL (Uniform Resource Locator) 文字列をその構成要素 (アドレススキーム、ネットワーク上の位置、パスその他) に分解したり、構成要素を URL に組みなおしたり、“相対 URL (relative URL)” を指定した “基底 URL (base URL)” に基づいて絶対 URL に変換するための標準的なインタフェースを定義しています。

このモジュールは Relative Uniform Resource Locators (相対 URL) に関する RFC に適合するように設計されており、次のスキームをサポートしています: `file`, `ftp`, `gopher`, `hdl`, `http`, `https`, `imap`, `mailto`, `mms`, `news`, `nnntp`, `prospero`, `rsync`, `rtsp`, `rtspu`, `sftp`, `shttp`, `sip`, `sips`, `snews`, `svn`, `svn+ssh`, `telnet`, `wais`.

バージョン 2.5 で追加: `sftp` および `sips` スキームのサポートが追加されました。

`urlparse` モジュールは以下の関数を定義しています:

```
urlparse.urlparse(urlstring[, scheme[, allow_fragments]])
```

URL を解析して 6 つの構成要素にし、6 要素のタプルを返します。このタプルは URL の一般的な構

造: `scheme://netloc/path;parameters?query#fragment` に対応しています。各タプル要素は文字列で、空の場合もあります。構成要素がさらに小さい要素に分解されることはありません (例えばネットワーク上の位置は単一の文字列になります)。また `%` によるエスケープは展開されません。上で示された区切り文字がタプルの各要素の一部分として含まれることはありませんが、`path` 要素の先頭のスラッシュがある場合には例外です。たとえば以下ようになります:

```
>>> from urlparse import urlparse
>>> o = urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
>>> o
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            ↪ params='', query='', fragment='')
>>> o.scheme
'http'
>>> o.port
80
>>> o.geturl()
'http://www.cwi.nl:80/%7Eguido/Python.html'
```

RFC 1808 にある文法仕様に基づき、`urlparse` は `'/'` で始まる場合にのみ `netloc` を認識します。それ以外の場合は、入力は相対 URL であると推定され、`path` 部分で始まることになります。

```
>>> from urlparse import urlparse
>>> urlparse('://www.cwi.nl:80/%7Eguido/Python.html')
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('www.cwi.nl/%7Eguido/Python.html')
ParseResult(scheme='', netloc='', path='www.cwi.nl/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('help/Python.html')
ParseResult(scheme='', netloc='', path='help/Python.html', params='',
            query='', fragment='')
```

`scheme` 引数が指定されている場合、標準のアドレススキームを表し、アドレススキームを指定していない URL に対してのみ使用されます。この引数の標準の値は空文字列です。

引数 `allow_fragments` が偽の場合、URL のアドレススキームがフラグメント指定をサポートしていても、フラグメント識別子は URL のそれ以前の部分に続く構成要素として認識、解析されません。この引数のデフォルトは `True` です。

戻り値は実際には `tuple` のサブクラスのインスタンスです。このクラスには以下の読み出し専用の便利な属性が追加されています:

属性	インデックス	値	指定されなかった場合の値
scheme	0	URL スキーム	<i>scheme</i> パラメータ
netloc	1	ネットワーク上の位置	空文字列
path	2	階層的パス	空文字列
params	3	最後のパス要素に対するパラメータ	空文字列
query	4	クエリ要素	空文字列
fragment	5	フラグメント識別子	空文字列
username		ユーザ名	<i>None</i>
password		パスワード	<i>None</i>
hostname		ホスト名 (小文字)	<i>None</i>
port		ポート番号を表わす整数 (もしあれば)	<i>None</i>

結果オブジェクトのより詳しい情報は `urlparse()` および `urlsplit()` の結果 節を参照してください。

Characters in the `netloc` attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a *ValueError*. If the URL is decomposed before parsing, or is not a Unicode string, no error will be raised.

バージョン 2.5 で変更: 戻り値に属性が追加されました。

バージョン 2.7 で変更: IPv6 URL の解析も行えるようになりました。

バージョン 2.7.17 で変更: Characters that affect netloc parsing under NFKC normalization will now raise *ValueError*.

`urlparse.parse_qs(qs[, keep_blank_values[, strict_parsing[, max_num_fields]])`

文字列引数として渡されたクエリ文字列 (`application/x-www-form-urlencoded` 型のデータ) を解析します。解析されたデータを辞書として返します。辞書のキーは一意的なクエリ変数名で、値は各変数名に対する値からなるリストです。

任意の引数 `keep_blank_values` は、パーセントエンコードされたクエリの中の値が入っていないクエリの値を空白文字列と見なすかどうかを示すフラグです。値が真であれば、値の入っていないフィールドは空文字列のままになります。標準では偽で、値の入っていないフィールドを無視し、そのフィールドはクエリに含まれていないものとして扱います。

任意の引数 `strict_parsing` はパース時のエラーをどう扱うかを定めるフラグです。値が偽なら (デフォルトの設定です)、エラーは暗黙のうちに無視します。値が真なら *ValueError* 例外を送出します。

The optional argument `max_num_fields` is the maximum number of fields to read. If set, then throws a *ValueError* if there are more than `max_num_fields` fields read.

辞書等をクエリ文字列に変換する場合は `urllib.urlencode()` 関数を使用してください。

バージョン 2.6 で追加: `cgi` モジュールからコピーされてきました。

バージョン 2.7.16 で変更: Added `max_num_fields` parameter.

`urlparse.parse_qs1(qs[, keep_blank_values[, strict_parsing[, max_num_fields]])`

文字列引数として渡されたクエリ文字列 (*application/x-www-form-urlencoded* 型のデータ) を解析します。解析されたデータは名前と値のペアからなるリストです。

任意の引数 *keep_blank_values* は、パーセントエンコードされたクエリの中の値が入っていないクエリの値を空白文字列と見なすかどうかを示すフラグです。値が真であれば、値の入っていないフィールドは空文字列のままになります。標準では偽で、値の入っていないフィールドを無視し、そのフィールドはクエリに含まれていないものとして扱います。

任意の引数 *strict_parsing* はパース時のエラーをどう扱うかを定めるフラグです。値が偽なら (デフォルトの設定です)、エラーは暗黙のうちに無視します。値が真なら *ValueError* 例外を送出します。

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a *ValueError* if there are more than *max_num_fields* fields read.

ペアのリストからクエリ文字列を生成する場合には *urllib.urlencode()* 関数を使用します。

バージョン 2.6 で追加: *cgi* モジュールからコピーされてきました。

バージョン 2.7.16 で変更: Added *max_num_fields* parameter.

urlparse.urlunparse(parts)

urlparse() が返すような形式のタプルから URL を構築します。 *parts* 引数は任意の 6 要素イテラブルです。解析された元の URL が、不要な区切り文字を持っていた場合には、多少違いはあるが等価な URL になるかもしれません (例えばクエリ内容が空の ? のようなもので、RFC はこれらを等価だと述べています)。

urlparse.urlsplit(urlstring[, scheme[, allow_fragments]])

urlparse() に似ていますが、URL から params を切り離しません。このメソッドは通常、URL の *path* 部分において、各セグメントにパラメーター指定をできるようにした最近の URL 構文 ([RFC 2396](#) 参照) が必要な場合に、 *urlparse()* の代わりに使われます。パスセグメントとパラメーターを分割するためには分割用の関数が必要です。この関数は 5 要素のタプル: (アドレススキーム、ネットワーク上の位置、パス、クエリ、フラグメント識別子) を返します。

戻り値は実際には *tuple* のサブクラスのインスタンスです。このクラスには以下の読み出し専用の便利な属性が追加されています:

属性	インデックス	値	指定されなかった場合の値
<i>scheme</i>	0	URL スキーム	<i>scheme</i> パラメータ
<i>netloc</i>	1	ネットワーク上の位置	空文字列
<i>path</i>	2	階層的パス	空文字列
<i>query</i>	3	クエリ要素	空文字列
<i>fragment</i>	4	フラグメント識別子	空文字列
<i>username</i>		ユーザ名	<i>None</i>
<i>password</i>		パスワード	<i>None</i>
<i>hostname</i>		ホスト名 (小文字)	<i>None</i>
<i>port</i>		ポート番号を表わす整数 (もしあれば)	<i>None</i>

結果オブジェクトのより詳しい情報は *urlparse()* および *urlsplit()* の結果 節を参照してください。

Characters in the `netloc` attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a `ValueError`. If the URL is decomposed before parsing, or is not a Unicode string, no error will be raised.

バージョン 2.2 で追加.

バージョン 2.5 で変更: 戻り値に属性が追加されました。

バージョン 2.7.17 で変更: Characters that affect netloc parsing under NFKC normalization will now raise `ValueError`.

`urlparse.urlunsplit(parts)`

`urlsplit()` が返すような形式のタプル中のエレメントを組み合わせて、文字列の完全な URL にします。 `parts` 引数は任意の 5 要素イテラブルです。解析された元の URL が、不要な区切り文字を持っていた場合には、多少違いはあるが等価な URL になるかもしれません (例えばクエリ内容が空の ? のようなもので、RFC はこれらを等価だと述べています)。

バージョン 2.2 で追加.

`urlparse.urljoin(base, url[, allow_fragments])`

”基底 URL”(`base`) と別の URL (`url`) を組み合わせて、完全な URL (”絶対 URL”) を構成します。ぶっちゃけ、この関数は基底 URL の要素、特にアドレススキーム、ネットワーク上の位置、およびパス (の一部) を使って、相対 URL にない要素を提供します。以下の例のようになります:

```
>>> from urlparse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

`allow_fragments` 引数は `urlparse()` における引数と同じ意味とデフォルトを持ちます。

注釈: `url` が (`//` か `scheme://` で始まっている) 絶対 URL であれば、その `url` のホスト名と / もしくは `scheme` は結果に反映されます。例えば:

```
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         '//www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

もしこの動作が望みのものでない場合は、 `url` を `urlsplit()` と `urlunsplit()` で先に処理して、 `scheme` と `netloc` を削除してください。

`urlparse.urldefrag(url)`

`url` がフラグメント識別子を含む場合、フラグメント識別子を持たないバージョンに修正された `url` と、別の文字列に分割されたフラグメント識別子を返します。 `url` 中にフラグメント識別子がない場合、そのままの `url` と空文字列を返します。

参考:

RFC 3986 - Uniform Resource Identifiers これが現在の標準規格 (STD66) です。 `urlparse` モジュールに対

するすべての変更はこの規格に準拠していなければなりません、若干の逸脱はありえます。これは主には後方互換性のため、また主要なブラウザで一般的に見られる、URL を解析する上でのいくつかの事実上の要件を満たすためです。

RFC 2732 - Format for Literal IPv6 Addresses in URL's. この規格は IPv6 の URL を解析するときの要求事項を記述しています。

RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax この RFC では Uniform Resource Name (URN) と Uniform Resource Locator (URL) の両方に対する一般的な文法的要求事項を記述しています。

RFC 2368 - The mailto URL scheme. mailto URL スキームに対する文法的要求事項です。

RFC 1808 - Relative Uniform Resource Locators この RFC には絶対 URL と相対 URL を結合するための規則がボーダケースの取扱い方を決定する "異常な例" つきで収められています。

RFC 1738 - Uniform Resource Locators (URL) この RFC では絶対 URL の形式的な文法と意味付けを仕様化しています。

20.16.1 `urlparse()` および `urlsplit()` の結果

`urlparse()` および `urlsplit()` から得られる結果オブジェクトはそれぞれ *tuple* 型のサブクラスです。これらのクラスはそれぞれの関数の説明の中で述べたような属性とともに、追加のメソッドを一つ提供しています。

`ParseResult.geturl()`

再結合された形で元の URL の文字列を返します。この文字列は元の URL とは次のような点で異なるかもしれません。スキームは常に小文字に正規化されます。また空の要素は省略されます。特に、空のパラメータ、クエリ、フラグメント識別子は取り除かれます。

加えた解析関数を逆に行えばこのメソッドの戻り値は元の URL になります:

```
>>> import urlparse
>>> url = 'HTTP://www.Python.org/doc/#'
```

```
>>> r1 = urlparse.urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
```

```
>>> r2 = urlparse.urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

バージョン 2.5 で追加.

以下のクラスが解析結果の実装を提供します:

class `urlparse.ParseResult` (*scheme, netloc, path, params, query, fragment*)
`urlparse()` の戻り値の具象クラス。

class `urlparse.SplitResult` (*scheme, netloc, path, query, fragment*)
`urlsplit()` の戻り値の具象クラス。

20.17 SocketServer — ネットワークサーバ構築のためのフレームワーク

注釈: `SocketServer` モジュールは、Python 3 では `socketserver` にリネームされました。 `2to3` ツールが自動的にソースコードの `import` を修正します。

ソースコード: `Lib/SocketServer.py`

`SocketServer` モジュールはネットワークサーバを実装するタスクを単純化します。

基本的な具象サーバクラスが 4 つあります:

class `SocketServer.TCPServer` (*server_address*, *RequestHandlerClass*,
bind_and_activate=True)
This uses the Internet TCP protocol, which provides for continuous streams of data between the client and server. If *bind_and_activate* is true, the constructor automatically attempts to invoke *server_bind()* and *server_activate()*. The other parameters are passed to the `BaseServer` base class.

class `SocketServer.UDPServer` (*server_address*, *RequestHandlerClass*,
bind_and_activate=True)
This uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The parameters are the same as for `TCPServer`.

class `SocketServer.UnixStreamServer` (*server_address*, *RequestHandlerClass*,
bind_and_activate=True)

class `SocketServer.UnixDatagramServer` (*server_address*, *RequestHandlerClass*,
bind_and_activate=True)
These more infrequently used classes are similar to the TCP and UDP classes, but use Unix domain sockets; they're not available on non-Unix platforms. The parameters are the same as for `TCPServer`.

これらの 4 つのクラスは要求を同期的に (*synchronously*) 処理します; 各要求は次の要求を開始する前に完結していなければなりません。同期的な処理は、サーバで大量の計算を必要とする、あるいはクライアントが処理するには時間がかかりすぎるような大量のデータを返す、といった理由によってリクエストに長い時間がかかる状況には向いていません。こうした状況の解決方法は別のプロセスを生成するか、個々の要求を扱うスレッドを生成することです; `ForkingMixIn` および `ThreadingMixIn` 配合クラス (mix-in classes) を使えば、非同期的な動作をサポートできます。

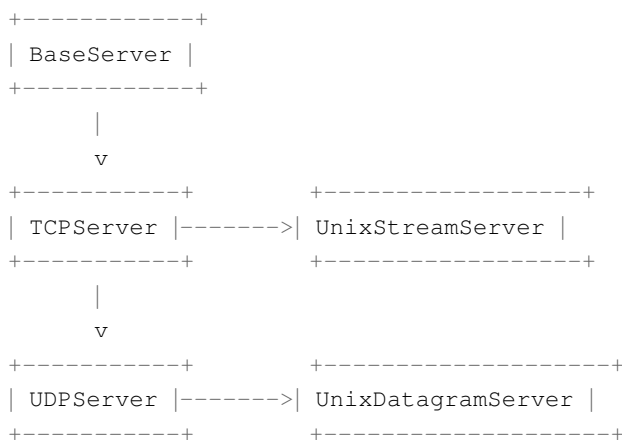
Creating a server requires several steps. First, you must create a request handler class by subclassing the `BaseRequestHandler` class and overriding its *handle()* method; this method will process incoming requests. Second, you must instantiate one of the server classes, passing it the server's address and the request handler class. Then call the *handle_request()* or *serve_forever()* method of the server object to process one or many requests. Finally, call *server_close()* to close the socket.

`ThreadingMixIn` から継承してスレッドを利用した接続を行う場合、突発的な通信切断時の処理を明示的に指定する必要があります。 `ThreadingMixIn` クラスには `daemon_threads` 属性があり、サーバがスレッドの終了を待ち合わせるかどうかを指定する事ができます。スレッドが独自の処理を行う場合は、このフラグを明示的に指定します。デフォルトは `False` で、Python は `ThreadingMixIn` クラスが起動した全てのスレッドが終了するまで実行し続けます。

サーバクラス群は使用するネットワークプロトコルに関わらず、同じ外部メソッドおよび属性を持ちます。

20.17.1 サーバ生成に関するノート

継承図にある五つのクラスのうち四つは四種類の同期サーバを表わしています:



`UnixDatagramServer` は `UDPServer` から派生していて、 `UnixStreamServer` からではないことに注意してください — IP と Unix ストリームサーバの唯一の違いはアドレスファミリーでそれは両方の Unix サーバクラスで単純に繰り返されています。

```
class SocketServer.ForkingMixIn
```

```
class SocketServer.ThreadingMixIn
```

Forking and threading versions of each type of server can be created using these mix-in classes. For instance, `ThreadingUDPServer` is created as follows:

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

The mix-in class comes first, since it overrides a method defined in `UDPServer`. Setting the various attributes also changes the behavior of the underlying server mechanism.

`ForkingMixIn` and the Forking classes mentioned below are only available on POSIX platforms that support `fork()`.

```
class SocketServer.ForkingTCPServer
```

```
class SocketServer.ForkingUDPServer
```

```
class SocketServer.ThreadingTCPServer
```

class `SocketServer.ThreadingUDPServer`

These classes are pre-defined using the mix-in classes.

To implement a service, you must derive a class from `BaseRequestHandler` and redefine its `handle()` method. You can then run various versions of the service by combining one of the server classes with your request handler class. The request handler class must be different for datagram or stream services. This can be hidden by using the handler subclasses `StreamRequestHandler` or `DatagramRequestHandler`.

もちろん、まだ頭を使わなければなりません! たとえば、サービスがリクエストによっては書き換えられるようなメモリ上の状態を使うならば、フォークするサーバを使うのは馬鹿げています。というのも子プロセスでの書き換えは親プロセスで保存されている初期状態にも親プロセスから分配される各子プロセスの状態にも届かないからです。この場合、スレッド実行するサーバを使うことはできますが、共有データの一貫性を保つためにロックを使わなければならなくなるでしょう。

一方、全てのデータが外部に (たとえばファイルシステムに) 保存される HTTP サーバを作っているのだとすると、同期クラスではどうしても一つの要求が処理されている間サービスが「耳の聞こえない」状態を呈することになります — この状態はもしクライアントが要求した全てのデータをゆっくり受け取るととても長い時間続きかねません。こういう場合にはサーバをスレッド実行したりフォークすることが適切です。

In some cases, it may be appropriate to process part of a request synchronously, but to finish processing in a forked child depending on the request data. This can be implemented by using a synchronous server and doing an explicit fork in the request handler class `handle()` method.

スレッドも `fork()` もサポートされない環境で (もしくはサービスにとってそれらがあまりに高価についたり不適切な場合に) 多数の同時要求を捌くもう一つのアプローチは、部分的に処理し終えた要求のテーブルを自分で管理し、次にどの要求に対処するか (または新しく入ってきた要求を扱うかどうか) を決めるのに `select()` を使う方法です。これは (もしスレッドやサブプロセスが使えなければ) 特にストリームサービスに対して重要で、そのようなサービスでは各クライアントが潜在的に長く接続し続けます。この問題を管理する別の方法について、`asyncore` モジュールを参照してください。

20.17.2 Server オブジェクト

class `SocketServer.BaseServer` (*server_address*, *RequestHandlerClass*)

This is the superclass of all Server objects in the module. It defines the interface, given below, but does not implement most of the methods, which is done in subclasses. The two parameters are stored in the respective *server_address* and *RequestHandlerClass* attributes.

fileno()

サーバが要求待ちを行っているソケットのファイル記述子を整数で返します。この関数は一般的に、同じプロセス中の複数のサーバを監視できるようにするために、`select.select()` に渡されます。

handle_request()

Process a single request. This function calls the following methods in order: `get_request()`, `verify_request()`, and `process_request()`. If the user-provided `handle()` method of the handler class raises an exception, the server's `handle_error()` method will be called.

If no request is received within `timeout` seconds, `handle_timeout()` will be called and `handle_request()` will return.

`serve_forever` (`poll_interval=0.5`)

Handle requests until an explicit `shutdown()` request. Poll for shutdown every `poll_interval` seconds. Ignores the `timeout` attribute. If you need to do periodic tasks, do them in another thread.

`shutdown()`

`serve_forever()` ループに停止するように指示し、停止されるまで待ちます。

バージョン 2.6 で追加.

`server.close()`

サーバをクリーンアップします。上書き出来ます。

バージョン 2.6 で追加.

`address_family`

サーバのソケットが属しているプロトコルファミリです。一般的な値は `socket.AF_INET` および `socket.AF_UNIX` です。

`RequestHandlerClass`

ユーザが提供する要求処理クラスです; 要求ごとにこのクラスのインスタンスが生成されます。

`server.address`

The address on which the server is listening. The format of addresses varies depending on the protocol family; see the documentation for the `socket` module for details. For Internet protocols, this is a tuple containing a string giving the address, and an integer port number: `('127.0.0.1', 80)`, for example.

`socket`

サーバが入力の要求待ちを行うためのソケットオブジェクトです。

サーバクラスは以下のクラス変数をサポートします:

`allow_reuse_address`

サーバがアドレスの再使用を許すかどうかを示す値です。この値は標準で `False` で、サブクラスで再使用ポリシーを変更するために設定することができます。

`request_queue_size`

要求待ち行列 (queue) のサイズです。単一の要求を処理するのに長時間かかる場合には、サーバが処理中に届いた要求は最大 `request_queue_size` 個まで待ち行列に置かれます。待ち行列が一杯になると、それ以降のクライアントからの要求は "接続拒否 (Connection denied)" エラーになります。標準の値は通常 5 ですが、この値はサブクラスで上書きすることができます。

`socket.type`

サーバが使うソケットの型です; 一般的な 2 つの値は、`socket.SOCK_STREAM` と `socket.SOCK_DGRAM` です。

`timeout`

タイムアウト時間 (秒)、もしくは、タイムアウトを望まない場合に `None`。 `handle_request()`

がこの時間内にリクエストを受信しない場合、`handle.timeout()` メソッドが呼ばれます。

`TCPServer` のような基底クラスのサブクラスで上書きできるサーバメソッドは多数あります; これらのメソッドはサーバオブジェクトの外部のユーザにとっては役に立たないものです。

`finish_request` (*request*, *client_address*)

Actually processes the request by instantiating `RequestHandlerClass` and calling its `handle()` method.

`get_request` ()

ソケットから要求を受理して、クライアントとの通信に使われる 新しいソケットオブジェクト、およびクライアントのアドレスからなる、2 要素のタプルを返します。

`handle_error` (*request*, *client_address*)

This function is called if the `handle()` method of a `RequestHandlerClass` instance raises an exception. The default action is to print the traceback to standard output and continue handling further requests.

`handle_timeout` ()

この関数は `timeout` 属性が `None` 以外に設定されて、リクエストがないままタイムアウト秒数が過ぎたときに呼ばれます。fork 型サーバーでのデフォルトの動作は、終了した子プロセスの情報を集めるようになっています。スレッド型サーバーではこのメソッドは何もしません。

`process_request` (*request*, *client_address*)

`finish_request()` を呼び出して、`RequestHandlerClass` のインスタンスを生成します。必要なら、この関数から新たなプロセスかスレッドを生成して要求を処理することができます; その処理は `ForkingMixIn` または `ThreadingMixIn` クラスが行います。

`server.activate` ()

Called by the server's constructor to activate the server. The default behavior for a TCP server just invokes `listen()` on the server's socket. May be overridden.

`server.bind` ()

サーバのコンストラクタによって呼び出され、適切なアドレスにソケットをバインドします。このメソッドは上書きできます。

`verify_request` (*request*, *client_address*)

ブール値を返さなければなりません; 値が `True` の場合には要求が処理され、`False` の場合には要求は拒否されます。サーバへのアクセス制御を実装するためにこの関数を上書きすることができます。標準の実装では常に `True` を返します。

20.17.3 Request Handler Objects

`class SocketServer.BaseRequestHandler`

This is the superclass of all request handler objects. It defines the interface, given below. A concrete request handler subclass must define a new `handle()` method, and can override any of the other methods. A new instance of the subclass is created for each request.

setup()

handle() メソッドより前に呼び出され、何らかの必要な初期化処理を行います。標準の実装では何も行いません。

handle()

この関数では、クライアントからの要求を実現するために必要な全ての作業を行わなければなりません。デフォルト実装では何もしません。この作業の上で、いくつかのインスタンス属性を利用することができます; クライアントからの要求は `self.request` です; クライアントのアドレスは `self.client_address` です; そしてサーバごとの情報にアクセスする場合には、サーバインスタンスを `self.server` で取得できます。

The type of `self.request` is different for datagram or stream services. For stream services, `self.request` is a socket object; for datagram services, `self.request` is a pair of string and socket.

finish()

handle() メソッドより後に呼び出され、何らかの必要なクリーンアップ処理を行います。標準の実装では何も行いません。 *setup()* メソッドが例外を送出した場合、このメソッドは呼び出されません。

class `SocketServer.StreamRequestHandler`

class `SocketServer.DatagramRequestHandler`

These *BaseRequestHandler* subclasses override the *setup()* and *finish()* methods, and provide `self.rfile` and `self.wfile` attributes. The `self.rfile` and `self.wfile` attributes can be read or written, respectively, to get the request data or return data to the client.

20.17.4 例

`SocketServer.TCPServer` の例

サーバーサイドの例です:

```
import SocketServer

class MyTCPHandler(SocketServer.BaseRequestHandler):
    """
    The request handler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print "{} wrote:".format(self.client_address[0])
        print self.data
        # just send back the same data, but upper-cased
```

(次のページに続く)

(前のページからの続き)

```
self.request.sendall(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    server = SocketServer.TCPServer((HOST, PORT), MyTCPHandler)

    # Activate the server; this will keep running until you
    # interrupt the program with Ctrl-C
    server.serve_forever()
```

別の、ストリーム (標準のファイル型のインタフェースを利用して通信をシンプルにしたファイルライクオブジェクト) を使うリクエストハンドラクラスの例です:

```
class MyTCPHandler(SocketServer.StreamRequestHandler):

    def handle(self):
        # self.rfile is a file-like object created by the handler;
        # we can now use e.g. readline() instead of raw recv() calls
        self.data = self.rfile.readline().strip()
        print "{} wrote:".format(self.client_address[0])
        print self.data
        # Likewise, self.wfile is a file-like object used to write back
        # to the client
        self.wfile.write(self.data.upper())
```

先ほどの違いは、`readline()` の呼び出しが、改行を受け取るまで `recv()` を複数回呼び出すことです。1 回の `recv()` の呼び出しは、クライアント側から 1 回の `sendall()` 呼び出しで送信された分しか受け取りません。

クライアントサイドの例:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(data + "\n")

    # Receive data from the server and shut down
    received = sock.recv(1024)
finally:
```

(次のページに続く)

(前のページからの続き)

```

    sock.close()

print "Sent:      {}".format(data)
print "Received: {}".format(received)

```

この例の出力は次のようになります:

Server:

```

$ python TCPServer.py
127.0.0.1 wrote:
hello world with TCP
127.0.0.1 wrote:
python is nice

```

Client:

```

$ python TCPClient.py hello world with TCP
Sent:      hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent:      python is nice
Received: PYTHON IS NICE

```

SocketServer.UDPServer の例

サーバーサイドの例です:

```

import SocketServer

class MyUDPHandler(SocketServer.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and since
    there is no connection the client address must be given explicitly
    when sending data back via sendto().
    """

    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print "{} wrote:".format(self.client_address[0])
        print data
        socket.sendto(data.upper(), self.client_address)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    server = SocketServer.UDPServer((HOST, PORT), MyUDPHandler)
    server.serve_forever()

```


クライアントサイドの例:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no connections.
# Instead, data is directly sent to the recipient via sendto().
sock.sendto(data + "\n", (HOST, PORT))
received = sock.recv(1024)

print "Sent: {}".format(data)
print "Received: {}".format(received)
```

この例の出力は、TCP サーバーの例と全く同じようになります。

非同期処理の **Mix-in**

複数の接続を非同期に処理するハンドラを作るには、*ThreadingMixIn* か *ForkingMixIn* クラスを利用します。

ThreadingMixIn クラスの利用例:

```
import socket
import threading
import SocketServer

class ThreadedTCPRequestHandler(SocketServer.BaseRequestHandler):

    def handle(self):
        data = self.request.recv(1024)
        cur_thread = threading.current_thread()
        response = "{}: {}".format(cur_thread.name, data)
        self.request.sendall(response)

class ThreadedTCPServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer):
    pass

def client(ip, port, message):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((ip, port))
    try:
        sock.sendall(message)
        response = sock.recv(1024)
        print "Received: {}".format(response)
    finally:
```

(次のページに続く)

(前のページからの続き)

```

        sock.close()

if __name__ == "__main__":
    # Port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    ip, port = server.server_address

    # Start a thread with the server -- that thread will then start one
    # more thread for each request
    server_thread = threading.Thread(target=server.serve_forever)
    # Exit the server thread when the main thread terminates
    server_thread.daemon = True
    server_thread.start()
    print "Server loop running in thread:", server_thread.name

    client(ip, port, "Hello World 1")
    client(ip, port, "Hello World 2")
    client(ip, port, "Hello World 3")

    server.shutdown()
    server.server_close()

```

この例の出力は次のようになります:

```

$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3

```

The *ForkingMixIn* class is used in the same way, except that the server will spawn a new process for each request. Available only on POSIX platforms that support *fork()*.

20.18 BaseHTTPServer — 基本的な機能を持つ HTTP サーバ

注釈: *BaseHTTPServer* モジュールは、Python 3 では *http.server* モジュールに統合されました。2to3 ツールが自動的にソースコード内の *import* を修正します。

ソースコード: [Lib/BaseHTTPServer.py](#)

このモジュールでは、HTTP サーバ (Web サーバ) を実装するための二つのクラスを定義しています。通常、このモジュールが直接使用されることはなく、特定の機能を持つ Web サーバを構築するために使われます。*SimpleHTTPServer* および *CGIHTTPServer* モジュールを参照してください。

最初のクラス `HTTPServer` は `SocketServer.TCPServer` のサブクラスで、従って `SocketServer.BaseServer` インタフェースを実装しています。 `HTTPServer` は HTTP ソケットを生成してリクエスト待ち (listen) を行い、リクエストをハンドラに渡します。サーバを作成して動作させるためのコードは以下のようになります:

```
def run(server_class=BaseHTTPServer.HTTPServer,
        handler_class=BaseHTTPServer.BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

class `BaseHTTPServer.HTTPServer` (*server_address, RequestHandlerClass*)

このクラスは `TCPServer` 型のクラスの上に構築されており、サーバのアドレスをインスタンス変数 `server_name` および `server_port` に記憶します。サーバはハンドラからアクセス可能で、通常 `server` インスタンス変数でアクセスします。

class `BaseHTTPServer.BaseHTTPRequestHandler` (*request, client_address, server*)

このクラスはサーバに到着したリクエストを処理します。このメソッド自体では、実際のリクエストに応答することはできません; (GET や POST のような) 各リクエストメソッドを処理するためにはサブクラス化しなければなりません。 `BaseHTTPRequestHandler` では、サブクラスで使うためのクラスやインスタンス変数、メソッド群を数多く提供しています。

このハンドラはリクエストを解釈し、次いでリクエスト形式ごとに固有のメソッドを呼び出します。メソッド名はリクエストの名称から構成されます。例えば、リクエストメソッド `SPAM` に対しては、`do_SPAM()` メソッドが引数なしで呼び出されます。リクエストに関連する情報は全て、ハンドラのインスタンス変数に記憶されています。サブクラスでは `__init__()` メソッドを上書きしたり拡張したりする必要はありません。

`BaseHTTPRequestHandler` は以下のインスタンス変数を持っています:

client_address

HTTP クライアントのアドレスを参照している、(host, port) の形式をとるタプルが入っています。

server

`server` インスタンスが入っています。

command

HTTP 命令 (リクエスト形式) が入っています。例えば 'GET' です。

path

リクエストされたパスが入っています。

request_version

リクエストのバージョン文字列が入っています。例えば 'HTTP/1.0' です。

headers

`MessageClass` クラス変数で指定されたクラスのインスタンスを保持しています。このインスタンスは HTTP リクエストのヘッダを解釈し、管理しています。

rfile

入力ストリームが入っており、そのファイルポインタはオプション入力データ部の先頭を指しています。

wfile

クライアントに返送する応答を書き込むための出力ストリームが入っています。このストリームに書き込む際には、HTTP プロトコルに従った形式をとらなければなりません。

`BaseHTTPRequestHandler` は以下のクラス変数を持っています:

server.version

サーバのソフトウェアバージョンを指定します。この値は上書きする必要が生じるかもしれません。書式は複数の文字列を空白で分割したもので、各文字列はソフトウェア名 [/バージョン] の形式をとります。例えば、'BaseHTTP/0.2' です。

sys.version

Python 処理系のバージョンが、`version.string` メソッドや `server.version` クラス変数で利用可能な形式で入っています。例えば 'Python/1.4' です。

error_message_format

クライアントに返すエラー応答を構築するための書式化文字列を指定します。この文字列は丸括弧で囲ったキー文字列で指定する形式を使うので、書式化の対象となる値は辞書でなければなりません。キー `code` は整数で、HTTP エラーコードを特定する数値です。 `message` は文字列で、何が発生したかを表す (詳細な) エラーメッセージが入ります。 `explain` はエラーコード番号の説明です。 `message` および `explain` の標準の値は `response` クラス変数で見つけることができます。

error_content_type

エラーレスポンスをクライアントに送信する時に使う Content-Type HTTP ヘッダを指定します。デフォルトでは 'text/html' です。

バージョン 2.6 で追加: 以前は、Content-Type は常に 'text/html' でした。

protocol.version

この値には応答に使われる HTTP プロトコルのバージョンを指定します。 'HTTP/1.1' に設定されると、サーバは持続的 HTTP 接続を許可します; しかしその場合、サーバは全てのクライアントに対する応答に、正確な値を持つ Content-Length ヘッダを (`send_header()` を使って) 含めなければなりません。以前のバージョンとの互換性を保つため、標準の設定値は 'HTTP/1.0' です。

MessageClass

HTTP ヘッダを解釈するための `rfc822.Message` 類似のクラスを指定します。通常この値が上書きされることはなく、標準の値 `mimetypes.Message` になっています。

responses

この変数はエラーコードを表す整数を二つの要素をもつタプルに対応付けます。タプルには短いメッセージと長いメッセージが入っています。例えば、`{code: (shortmessage, longmessage)}` といったようになります。 `shortmessage` は通常、エラー応答における `message` キーの値として使われ、 `longmessage` は `explain` キーの値として使われます (`error_message_format` クラス変数を参照してください)。

`BaseHTTPRequestHandler` インスタンスは以下のメソッドを持っています:

handle()

`handle_one_request()` を一度だけ (持続的接続が有効になっている場合には複数回) 呼び出して、HTTP リクエストを処理します。このメソッドを上書きする必要はまったくありません; そうする代わりに適切な `do_*()` を実装してください。

handle_one_request()

このメソッドはリクエストを解釈し、適切な `do_*()` メソッドに転送します。このメソッドを上書きする必要はまったくありません。

send_error(*code* [, *message*])

完全なエラー応答をクライアントに送信し、ログ記録します。*code* は数値型で、HTTP エラーコードを指定します。*message* はオプションで、より詳細なメッセージテキストです。完全なヘッダのセットが送信された後、`error_message_format` クラス変数を使って組み立てられたテキストが送られます。メソッドが HEAD もしくはレスポンスコードが次のどれかの場合はボディは空です: 1xx, 204 No Content, 205 Reset Content, 304 Not Modified。

send_response(*code* [, *message*])

応答ヘッダを送信し、受理したリクエストをログ記録します。HTTP 応答行が送られた後、`Server` および `Date` ヘッダが送られます。これら二つのヘッダはそれぞれ `version_string()` および `date_time_string()` メソッドで取り出します。

send_header(*keyword*, *value*)

出力ストリームに特定の HTTP ヘッダを書き込みます。*keyword* はヘッダのキーワードを指定し、*value* にはその値を指定します。

end_headers()

応答中の HTTP ヘッダの終了を示す空行を送信します。

log_request([*code* [, *size*]])

受理された (成功した) リクエストをログに記録します。*code* にはこの応答に関連付けられた HTTP コード番号を指定します。応答メッセージの大きさを知ることができる場合、*size* パラメタに渡すとよいでしょう。

log_error(...)

リクエストを遂行できなかった際に、エラーをログに記録します。標準では、メッセージを `log_message()` に渡します。従って同じ引数 (*format* と追加の値) を取ります。

log_message(*format*, ...)

任意のメッセージを `sys.stderr` にログ記録します。このメソッドは通常、カスタムのエラーログ記録機構を作成するために上書きされます。*format* 引数は標準の `printf` 形式の書式化文字列で、`log_message()` に渡された追加の引数は書式化の入力として適用されます。ログ記録される全てのメッセージには、クライアントの IP アドレスおよび現在の日付、時刻が先頭に付けられます。

version_string()

サーバソフトウェアのバージョン文字列を返します。この文字列はクラス変数 `server_version` および `sys.version` を組み合わせたものです。

date.time_string([*timestamp*])

メッセージヘッダ向けに書式化された、*timestamp* (*time.time()* のフォーマットである必要があります) で与えられた日時を返します。もし *timestamp* が省略された場合には、現在の日時が使われます。

出力は 'Sun, 06 Nov 1994 08:49:37 GMT' のようになります。

バージョン 2.5 で追加: *timestamp* パラメータ。

log.date_time_string()

ログ記録向けに書式化された、現在の日付および時刻を返します。

address_string()

ログ記録向けに書式化された、クライアントのアドレスを返します。このときクライアントの IP アドレスに対する名前解決を行います。

20.18.1 他の例

永遠ではなく、何かの条件が満たされるまでの間実行するサーバーを作るには:

```
def run_while_true(server_class=BaseHTTPServer.HTTPServer,
                   handler_class=BaseHTTPServer.BaseHTTPRequestHandler):
    """
    This assumes that keep_running() is a function of no arguments which
    is tested initially and after each request.  If its return value
    is true, the server continues.
    """
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    while keep_running():
        httpd.handle_request()
```

参考:

CGIHTTPServer モジュール CGI スクリプトをサポートするように拡張されたリクエストハンドラ。

SimpleHTTPServer モジュール ドキュメントルートの下にあるファイルに対する要求への応答のみに制限した基本リクエストハンドラ。

20.19 SimpleHTTPServer — 簡潔な HTTP リクエストハンドラ

注釈: *SimpleHTTPServer* モジュールは、Python 3 では `http.server` モジュールに統合されました。*2to3* ツールが自動的にソースコード内の `import` を修正します。

警告: `SimpleHTTPServer` is not recommended for production. It only implements basic security checks.

`SimpleHTTPServer` モジュールは、`SimpleHTTPRequestHandler` クラス 1 つを提供しています。このクラスは、`BaseHTTPServer.BaseHTTPRequestHandler` に対して互換性のあるインタフェースを持っています。

`SimpleHTTPServer` モジュールは以下のクラスを定義します:

class `SimpleHTTPServer.SimpleHTTPRequestHandler` (*request, client_address, server*)

このクラスは、現在のディレクトリ以下にあるファイルを、HTTP リクエストにおけるディレクトリ構造に直接対応付けて提供します。

リクエストの解釈のような、多くの作業は基底クラス `BaseHTTPServer.BaseHTTPRequestHandler` で行われます。このクラスは関数 `do_GET()` および `do_HEAD()` を実装しています。

`SimpleHTTPRequestHandler` では以下のメンバ変数を定義しています:

server.version

この値は `"SimpleHTTP/" + __version__` になります。`__version__` はこのモジュールで定義されている値です。

extensions.map

拡張子を MIME 型指定子に対応付ける辞書です。標準の型指定は空文字列で表され、この値は `application/octet-stream` と見なされます。対応付けは大小文字の区別をするので、小文字のキーのみを入れるべきです。

`SimpleHTTPRequestHandler` では以下のメソッドを定義しています:

do_HEAD()

このメソッドは 'HEAD' 型のリクエスト処理を実行します: すなわち、GET リクエストの時に送信されるものと同じヘッダを送信します。送信される可能性のあるヘッダについての完全な説明は `do_GET()` メソッドを参照してください。

do_GET()

リクエストを現在の作業ディレクトリからの相対的なパスとして解釈することで、リクエストをローカルシステム上のファイルと対応付けます。

リクエストがディレクトリに対応付けられた場合、`index.html` または `index.htm` を (この順序で) チェックします。もしファイルを発見できればその内容を、そうでなければディレクトリー覧を `list_directory()` メソッドで生成して、返します。このメソッドは `os.listdir()` をディレクトリのスキャンに用いており、`listdir()` が失敗した場合には 404 応答が返されます。

リクエストがファイルに対応付けられた場合、そのファイルを開いて内容を返します。要求されたファイルを開く際に何らかの `IOError` 例外が送出された場合、リクエストは 404、'File not found' エラーに対応づけられます。そうでない場合、`extensions_map` 変数を用いて `guess_type()` メソッドにより `content-type` が推測されます。

出力は 'Content-type:' と推測されたコンテンツタイプで、その後にファイルサイズを示す 'Content-Length;' ヘッダと、ファイルの更新日時を示す 'Last-Modified:' ヘッダが続きます。

そしてヘッダの終了を示す空白行が続き、さらにその後にファイルの内容が続きます。このファイルはコンテンツタイプが `text/` で始まっている場合はテキストモードで、そうでなければバイナリモードで開かれます。

`SimpleHTTPServer` モジュールの `test()` 関数は `SimpleHTTPRequestHandler` をハンドラとして使うサーバを作る例になっています。

バージョン 2.5 で追加: 'Last-Modified' ヘッダ。

`SimpleHTTPServer` モジュールを使って現在のディレクトリ以下にあるファイルにアクセスできるだけの、非常に初歩的な Web サーバを立ち上げる方法は以下の通りです。

```
import SimpleHTTPServer
import SocketServer

PORT = 8000

Handler = SimpleHTTPServer.SimpleHTTPRequestHandler

httpd = SocketServer.TCPServer(("", PORT), Handler)

print "serving at port", PORT
httpd.serve_forever()
```

インタプリタの `-m` スイッチで `SimpleHTTPServer` モジュールとポート番号を指定して直接実行することもできます。上の例と同じように、ここで立ち上がったサーバは現在のディレクトリ以下のファイルへのアクセスを提供します。

```
python -m SimpleHTTPServer 8000
```

参考:

`BaseHTTPServer` モジュール Web サーバとリクエスト処理機構を実装した基底クラスです。

20.20 CGIHTTPServer — CGI 実行機能付き HTTP リクエスト処理機構

注釈: `BaseHTTPServer` モジュールは Python 3 では `http.server` に統合されました。ソースコードを 3 用に変換する時は、`2to3` ツールが自動的に `import` を修正します。

`CGIHTTPServer` モジュールでは、`BaseHTTPServer.BaseHTTPRequestHandler` 互換のインタフェースを持ち、`SimpleHTTPServer.SimpleHTTPRequestHandler` の動作を継承していますが CGI スクリプトを動作することもできる、HTTP リクエストハンドラクラスを定義しています。

注釈: このモジュールは CGI スクリプトを Unix および Windows システム上で実行させることができます。

注釈: `CGIHTTPRequestHandler` クラスで実行される CGI スクリプトは HTTP コード 200 (スクリプトの出力が後に続く) を実行に先立って出力される (これがステータスコードになります) ため、リダイレクト (コード 302) を行なうことができません。

`CGIHTTPServer` モジュールでは、以下のクラスを定義しています:

class `CGIHTTPServer.CGIHTTPRequestHandler` (*request, client_address, server*)

このクラスは、現在のディレクトリかその下のディレクトリにおいて、ファイルか CGI スクリプト出力を提供するために使われます。HTTP 階層構造からローカルなディレクトリ構造への対応付けは `SimpleHTTPServer.SimpleHTTPRequestHandler` と全く同じなので注意してください。

このクラスでは、ファイルが CGI スクリプトであると推測された場合、これをファイルとして提供する代わりにスクリプトを実行します。— 他の一般的なサーバ設定は特殊な拡張子を使って CGI スクリプトであることを示すのに対し、ディレクトリベースの CGI だけが使われます。

`do_GET()` および `do_HEAD()` 関数は、HTTP 要求が `cgi_directories` パス以下のどこかを指している場合、ファイルを提供するのではなく、CGI スクリプトを実行してその出力を提供するように変更されています。

`CGIHTTPRequestHandler` では以下のデータメンバを定義しています:

cgi_directories

この値は標準で `['/cgi-bin', '/htbin']` であり、CGI スクリプトを含んでいることを示すディレクトリを記述します。

`CGIHTTPRequestHandler` では以下のメソッドを定義しています:

do_POST()

このメソッドは、CGI スクリプトでのみ許されている 'POST' 型の HTTP 要求に対するサービスを行います。CGI でない url に対して POST を試みた場合、出力は `Error 501, "Can only POST to CGI scripts"` になります。

セキュリティ上の理由から、CGI スクリプトはユーザ nobody の UID で動作するので注意してください。CGI スクリプトが原因で発生した問題は、Error 403 に変換されます。

使用例については、`test()` 関数の実装を参照してください。

参考:

`BaseHTTPServer` モジュール Web サーバとリクエスト処理機構を実装した基底クラスです。

20.21 cookielib — HTTP クライアント用の Cookie 処理

注釈: `cookielib` モジュールは、Python 3 では `http.cookiejar` にリネームされました。 `2to3` ツールが自動的にソースコードの `import` を修正します。

バージョン 2.4 で追加.

ソースコード: [Lib/cookiejar.py](#)

`cookielib` モジュールは HTTP クッキーの自動処理をおこなうクラスを定義します。これは小さなデータの断片 – クッキー – を要求する web サイトにアクセスする際に有用です。クッキーとは web サーバの HTTP レスポンスによってクライアントのマシンに設定され、のちの HTTP リクエストをおこなうさいにサーバに返されるものです。

標準的な Netscape クッキープロトコルおよび [RFC 2965](#) で定義されているプロトコルの両方を処理できます。RFC 2965 の処理はデフォルトではオフになっています。 [RFC 2109](#) のクッキーは Netscape クッキーとして解析され、のちに有効な 'ポリシー' に従って Netscape または RFC 2965 クッキーとして処理されます。但し、インターネット上の大多数のクッキーは Netscape クッキーです。 `cookielib` はデファクトスタンダードの Netscape クッキープロトコル (これは元々 Netscape が策定した仕様とはかなり異なっています) に従うようになっており、RFC 2109 で導入された `max-age` や `port` などのクッキー属性にも注意を払います。

注釈: `Set-Cookie` や `Set-Cookie2` ヘッダに現れる多種多様なパラメータの名前 (`domain` や `expires` など) は便宜上 属性 と呼ばれますが、ここでは Python の属性と区別するため、かわりに クッキー属性 と呼ぶことにします。

このモジュールは以下の例外を定義しています:

exception `cookielib.LoadError`

この例外は `FileCookieJar` インスタンスがファイルからクッキーを読み込むのに失敗した場合に発生します。

注釈: (`IOError` を発行する) Python 2.4 との後方互換性のために、 `LoadError` は `IOError` のサブクラスです。

以下のクラスが提供されています:

class `cookielib.CookieJar` (*policy=None*)

policy は `CookiePolicy` インターフェイスを実装するオブジェクトです。

`CookieJar` クラスには HTTP クッキーを保管します。これは HTTP リクエストに応じてクッキーを取り出し、それを HTTP レスポンスの中で返します。必要に応じて、 `CookieJar` インスタンスは保

管されているクッキーを自動的に破棄します。このサブクラスは、クッキーをファイルやデータベースに格納したり取り出したりする操作をおこなう役割を負っています。

class `cookielib.FileCookieJar` (*filename*, *delayload=None*, *policy=None*)

policy は `CookiePolicy` インターフェイスを実装するオブジェクトです。これ以外の引数については、該当する属性の説明を参照してください。

`FileCookieJar` はディスク上のファイルからのクッキーの読み込み、もしくは書き込みをサポートします。実際には、`load()` または `revert()` のどちらかのメソッドが呼ばれるまでクッキーは指定されたファイルからはロードされません。このクラスのサブクラスは `FileCookieJar` のサブクラスと `web ブラウザとの連携` 節で説明します。

class `cookielib.CookiePolicy`

このクラスは、あるクッキーをサーバから受け入れるべきか、そしてサーバに返すべきかを決定する役割を負っています。

class `cookielib.DefaultCookiePolicy` (*blocked_domains=None*, *allowed_domains=None*, *netscape=True*, *rfc2965=False*, *rfc2109_as_netscape=None*, *hide_cookie2=False*, *strict_domain=False*, *strict_rfc2965_unverifiable=True*, *strict_ns_unverifiable=False*, *strict_ns_domain=DefaultCookiePolicy.DomainLiberal*, *strict_ns_set_initial_dollar=False*, *strict_ns_set_path=False*)

コンストラクタはキーワード引数しか取りません。`blocked_domains` はドメイン名からなるシーケンスで、ここからは決してクッキーを受けとらないし、このドメインにクッキーを返すこともありません。`allowed_domains` が `None` でない場合、これはこのドメインのみからクッキーを受けとり、返すという指定になります。これ以外の引数については `CookiePolicy` および `DefaultCookiePolicy` オブジェクトの説明をごらんください。

`DefaultCookiePolicy` は Netscape および RFC 2965 クッキーの標準的な許可 / 拒絶のルールを実装しています。デフォルトでは、RFC 2109 のクッキー (`Set-Cookie` の `version` クッキー属性が 1 で受けとられるもの) は RFC 2965 のルールで扱われます。しかし、RFC 2965 処理が無効に設定されているか `rfc2109_as_netscape` が `True` の場合、RFC 2109 クッキーは `CookieJar` インスタンスによって `Cookie` のインスタンスの `version` 属性を 0 に設定する事で Netscape クッキーに「ダウングレード」されます。また `DefaultCookiePolicy` にはいくつかの細かいポリシー設定をおこなうパラメータが用意されています。

class `cookielib.Cookie`

このクラスは Netscape クッキー、RFC 2109 のクッキー、および RFC 2965 のクッキーを表現します。`cookielib` のユーザが自分で `Cookie` インスタンスを作成することは想定されていません。かわりに、必要に応じて `CookieJar` インスタンスの `make_cookies()` を呼ぶことになっています。

参考:

`urllib2` モジュール クッキーの自動処理をおこない URL を開くモジュールです。

`Cookie` モジュール HTTP のクッキークラスで、基本的にはサーバサイドのコードで有用です。
`cookielib` および `Cookie` モジュールは互いに依存してはいません。

https://curl.haxx.se/rfc/cookie_spec.html 元祖 Netscape のクッキープロトコルの仕様です。今でもこれが主流のプロトコルですが、現在のメジャーなブラウザ(と `cookielib`) が実装している「Netscape クッキープロトコル」は `cookie_spec.html` で述べられているものとおおまかにしか似ていません。

RFC 2109 - HTTP State Management Mechanism RFC 2965 によって過去の遺物になりました。
`Set-Cookie` の `version=1` で使います。

RFC 2965 - HTTP State Management Mechanism Netscape プロトコルのバグを修正したものです。
`Set-Cookie` のかわりに `Set-Cookie2` を使いますが、普及してはいません。

<http://kristol.org/cookie/errata.html> RFC 2965 に対する未完の正誤表です。

RFC 2964 - Use of HTTP State Management

20.21.1 CookieJar および FileCookieJar オブジェクト

`CookieJar` オブジェクトは保管されている `Cookie` オブジェクトをひとつずつ取り出すための、イテレータ (*iterator*) ・プロトコルをサポートしています。

`CookieJar` は以下のようなメソッドを持っています:

`CookieJar.add_cookie_header(request)`

`request` に正しい `Cookie` ヘッダを追加します。

ポリシーが許すのであれば (`CookieJar` の `CookiePolicy` インスタンスにある属性のうち、`rfc2965` および `hide_cookie2` がそれぞれ真と偽であるような場合)、必要に応じて `Cookie2` ヘッダも追加されます。

`request` オブジェクト (通常は `urllib2.Request` インスタンス) は、`urllib2` のドキュメントに記されているように、`get_full_url()`, `get_host()`, `get_type()`, `unverifiable()`, `get_origin_req_host()`, `has_header()`, `get_header()`, `header_items()` および `add_unredirected_header()` の各メソッドをサポートしている必要があります。

`CookieJar.extract_cookies(response, request)`

HTTP `response` からクッキーを取り出し、ポリシーによって許可されていればこれを `CookieJar` 内に保管します。

`CookieJar` は `response` 引数の中から許可されている `Set-Cookie` および `Set-Cookie2` ヘッダを探しだし、適切に (`CookiePolicy.set_ok()` メソッドの承認におうじて) クッキーを保管します。

`response` オブジェクト (通常は `urllib2.urlopen()` あるいはそれに類似する呼び出しによって得られます) は `info()` メソッドをサポートしている必要があります。これは `getallmatchingheaders()` メソッドのあるオブジェクト (通常は `mimetools.Message` インスタンス) を返すものです。

`request` オブジェクト (通常は `urllib2.Request` インスタンス) は `urllib2` のドキュメ

ントに記されているように、`get_full_url()`、`get_host()`、`unverifiable()` および `get_origin_req_host()` の各メソッドをサポートしている必要があります。この `request` はそのクッキーの保存が許可されているかを確認するとともに、クッキー属性のデフォルト値を設定するのに使われます。

`CookieJar.set_policy(policy)`

使用する `CookiePolicy` インスタンスを指定します。

`CookieJar.make_cookies(response, request)`

`response` オブジェクトから得られた `Cookie` オブジェクトからなるシーケンスを返します。

`response` および `request` 引数で要求されるインスタンスについては、`extract_cookies()` の説明を参照してください。

`CookieJar.set_cookie_if_ok(cookie, request)`

ポリシーが許すのであれば、与えられた `Cookie` を設定します。

`CookieJar.set_cookie(cookie)`

与えられた `Cookie` を、それが設定されるべきかどうかのポリシーのチェックを行わずに設定します。

`CookieJar.clear([domain[, path[, name]])]`

いくつかのクッキーを消去します。

引数なしで呼ばれた場合は、すべてのクッキーを消去します。引数がひとつ与えられた場合、その `domain` に属するクッキーのみを消去します。ふたつの引数が与えられた場合、指定された `domain` と URL `path` に属するクッキーのみを消去します。引数が3つ与えられた場合、`domain`、`path` および `name` で指定されるクッキーが消去されます。

与えられた条件に一致するクッキーがない場合は `KeyError` を発生させます。

`CookieJar.clear_session_cookies()`

すべてのセッションクッキーを消去します。

保存されているクッキーのうち、`discard` 属性が真になっているものすべてを消去します (通常これは `max-age` または `expires` のどちらのクッキー属性もないか、あるいは明示的に `discard` クッキー属性が指定されているものです)。対話的なブラウザの場合、セッションの終了はふつうブラウザのウィンドウを閉じることに相当します。

注意: `ignore_discard` 引数に真を指定しないかぎり、`save()` メソッドはセッションクッキーは保存しません。

さらに `FileCookieJar` は以下のようなメソッドを実装しています:

`FileCookieJar.save(filename=None, ignore_discard=False, ignore_expires=False)`

クッキーをファイルに保存します。

この基底クラスは `NotImplementedError` を発生させます。サブクラスはこのメソッドを実装しないままにしておいてもかまいません。

`filename` はクッキーを保存するファイルの名前です。`filename` が指定されない場合、`self.filename` が使用されます (このデフォルト値は、それが存在する場合は、コンストラクタに渡されています)。

`self.filename` も `None` の場合は `ValueError` が発生します。

`ignore_discard` : 破棄されるよう指示されていたクッキーでも保存します。`ignore_expires` : 期限の切れたクッキーでも保存します。

ここで指定されたファイルがもしすでに存在する場合は上書きされるため、以前にあったクッキーはすべて消去されます。保存したクッキーはあとで `load()` または `revert()` メソッドを使って復元することができます。

`FileCookieJar.load(filename=None, ignore_discard=False, ignore_expires=False)`

ファイルからクッキーを読み込みます。

それまでのクッキーは新しいものに上書きされない限り残ります。

ここでの引数の値は `save()` と同じです。

名前のついたファイルはこのクラスがわかるやり方で指定する必要があります。さもないと `LoadError` が発生します。さらに、例えばファイルが存在しないような時に `IOError` が発生する場合があります。

注釈: (`IOError` を発行する) Python 2.4 との後方互換性のために、`LoadError` は `IOError` のサブクラスです。

`FileCookieJar.revert(filename=None, ignore_discard=False, ignore_expires=False)`

すべてのクッキーを破棄し、保存されているファイルから読み込み直します。

`revert()` は `load()` と同じ例外を発生させる事ができます。失敗した場合、オブジェクトの状態は変更されません。

`FileCookieJar` インスタンスは以下のような公開の属性をもっています:

`FileCookieJar.filename`

クッキーを保存するデフォルトのファイル名を指定します。この属性には代入することができます。

`FileCookieJar.delayload`

真であれば、クッキーを読み込むさいにディスクから遅延読み込みします。この属性には代入することができません。この情報は単なるヒントであり、(ディスク上のクッキーが変わらない限りは) インスタンスのふるまいには影響を与えず、パフォーマンスのみに影響します。`CookieJar` オブジェクトはこの値を無視することもあります。標準ライブラリに含まれている `FileCookieJar` クラスで遅延読み込みをおこなうものではありません。

20.21.2 FileCookieJar のサブクラスと web ブラウザとの連携

クッキーの読み書きのために、以下の `CookieJar` サブクラスが提供されています。

`class cookielib.MozillaCookieJar(filename, delayload=None, policy=None)`

Mozilla の `cookies.txt` ファイル形式 (この形式はまた Lynx と Netscape ブラウザによっても使わ

れています) でディスクにクッキーを読み書きするための `FileCookieJar` です。

注釈: Firefox 3 ウェブブラウザは、もうクッキーを `cookies.txt` ファイルフォーマットでは保存しなくなっています。

注釈: このクラスは RFC 2965 クッキーに関する情報を失います。また、より新しいか、標準でない `port` などのクッキー属性についての情報も失います。

警告: もしクッキーの損失や欠損が望ましくない場合は、クッキーを保存する前にバックアップを取っておくようにしてください (ファイルへの読み込み / 保存をくり返すと微妙な変化が生じる場合があります)。

また、Mozilla の起動中にクッキーを保存すると、Mozilla によって内容が破壊されてしまうことにも注意してください。

class `cookielib.LWPCookieJar` (*filename, delayload=None, policy=None*)

`libwww-perl` のライブラリである `Set-Cookie3` ファイル形式でディスクにクッキーを読み書きするための `FileCookieJar` です。これはクッキーを人間に可読な形式で保存するのに向いています。

20.21.3 CookiePolicy オブジェクト

`CookiePolicy` インターフェイスを実装するオブジェクトは以下のようなメソッドを持っています:

`CookiePolicy.set_ok(cookie, request)`

クッキーがサーバから受け入れられるべきかどうかを表わす `boolean` 値を返します。

`cookie` は `cookielib.Cookie` インスタンスです。 `request` は `CookieJar.extract_cookies()` の説明で定義されているインターフェイスを実装するオブジェクトです。

`CookiePolicy.return_ok(cookie, request)`

クッキーがサーバに返されるべきかどうかを表わす `boolean` 値を返します。

`cookie` は `cookielib.Cookie` インスタンスです。 `request` は `CookieJar.add_cookie_header()` の説明で定義されているインターフェイスを実装するオブジェクトです。

`CookiePolicy.domain_return_ok(domain, request)`

与えられたクッキーのドメインに対して、そこにクッキーを返すべきでない場合には `false` を返します。

このメソッドは高速化のためのものです。これにより、すべてのクッキーをある特定のドメインに対してチェックする (これには多数のファイル読みこみを伴う場合があります) 必要がなくなります。

`domain.return_ok()` および `path.return_ok()` の両方から `true` が返された場合、すべての決定は `return_ok()` に委ねられます。

もし、このクッキードメインに対して `domain.return_ok()` が `true` を返すと、つぎにそのクッキーのパス名に対して `path.return_ok()` が呼ばれます。そうでない場合、そのクッキードメインに対する `path.return_ok()` および `return_ok()` は決して呼ばれることはありません。`path.return_ok()` が `true` を返すと、`return_ok()` がその `Cookie` オブジェクト自身の全チェックのために呼ばれます。そうでない場合、そのクッキーパス名に対する `return_ok()` は決して呼ばれることはありません。

注意: `domain.return_ok()` は `request` ドメインだけではなく、すべての `cookie` ドメインに対して呼ばれます。たとえば `request` ドメインが `"www.example.com"` だった場合、この関数は `".example.com"` および `"www.example.com"` の両方に対して呼ばれることがあります。同じことは `path.return_ok()` にもいえます。

`request` 引数は `return_ok()` で説明されているとおりです。

`CookiePolicy.path_return_ok(path, request)`

与えられたクッキーのパス名に対して、そこにクッキーを返すべきでない場合には `false` を返します。

`domain.return_ok()` の説明を参照してください。

上のメソッドの実装にくわえて、`CookiePolicy` インターフェイスの実装では以下の属性を設定する必要があります。これはどのプロトコルがどのように使われるべきかを示すもので、これらの属性にはすべて代入することが許されています。

`CookiePolicy.netscape`

Netscape プロトコルを実装していることを示します。

`CookiePolicy.rfc2965`

RFC 2965 プロトコルを実装していることを示します。

`CookiePolicy.hide_cookie2`

`Cookie2` ヘッダをリクエストに含めないようにします (このヘッダが存在する場合、私たちは RFC 2965 クッキーを理解するということをサーバに示すことになります)。

もっとも有用な方法は、`DefaultCookiePolicy` をサブクラス化した `CookiePolicy` クラスを定義して、いくつか (あるいはすべて) のメソッドをオーバーライドすることでしょう。`CookiePolicy` 自体はどのようなクッキーも受け入れて設定を許可する「ポリシー無し」ポリシーとして使うこともできます (これが役に立つことはあまりありません)。

20.21.4 DefaultCookiePolicy オブジェクト

クッキーを受けつけ、またそれを返す際の標準的なルールを実装します。

RFC 2965 クッキーと Netscape クッキーの両方に対応しています。デフォルトでは、RFC 2965 の処理はオフになっています。

自分のポリシーを提供するいちばん簡単な方法は、このクラスを継承して、自分用の追加チェックの前にオーバーライドした元のメソッドを呼び出すことです:

```
import cookielib
class MyCookiePolicy(cockielib.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not cookielib.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

`CookiePolicy` インターフェイスを実装するのに必要な機能に加えて、このクラスではクッキーを受けとったり設定したりするドメインを許可したり拒絶したりできるようになっています。ほかにも、Netscape プロトコルのかなり緩い規則をややきつくするために、いくつかの厳密性のスイッチがついています(いくつかの良性クッキーをブロックする危険性もありますが)。

ドメインのブラックリスト機能やホワイトリスト機能も提供されています(デフォルトではオフになっています)。ブラックリストがなく、(ホワイトリスト機能を使用している場合は) ホワイトリストにあるドメインのみがクッキーを設定したり返したりすることを許可されます。コンストラクタの引数 `blocked_domains`、および `blocked_domains()` と `set_blocked_domains()` メソッドを使ってください(`allowed_domains` に関しても同様の対応する引数とメソッドがあります)。ホワイトリストを設定した場合は、それを `None` にすることでホワイトリスト機能をオフにすることができます。

ブラックリストあるいはホワイトリスト中にあるドメインのうち、ドット (.) で始まっていないものは、正確にそれと一致するドメインのクッキーにしか適用されません。たとえばブラックリスト中のエントリ `"example.com"` は、`"example.com"` にはマッチしますが、`"www.example.com"` にはマッチしません。一方ドット (.) で始まっているドメインは、より特化されたドメインともマッチします。たとえば、`".example.com"` は、`"www.example.com"` と `"www.coyote.example.com"` の両方にマッチします(が、`"example.com"` 自身にはマッチしません)。IP アドレスは例外で、つねに正確に一致する必要があります。たとえば、かりに `blocked_domains` が `"192.168.1.2"` と `".168.1.2"` を含んでいたとして、`192.168.1.2` はブロックされますが、`193.168.1.2` はブロックされません。

`DefaultCookiePolicy` は以下のような追加メソッドを実装しています:

`DefaultCookiePolicy.blocked_domains()`

ブロックしているドメインのシーケンスを (タプルとして) 返します。

`DefaultCookiePolicy.set_blocked_domains(blocked_domains)`

ブロックするドメインを設定します。

`DefaultCookiePolicy.is_blocked(domain)`

`domain` がクッキーを授受しないブラックリストに載っているかどうかを返します。

`DefaultCookiePolicy.allowed_domains()`

`None` あるいは明示的に許可されているドメインを (タプルとして) 返します。

`DefaultCookiePolicy.set_allowed_domains(allowed_domains)`

許可するドメイン、あるいは `None` を設定します。

`DefaultCookiePolicy.is_not_allowed(domain)`

`domain` がクッキーを授受するホワइटリストに載っているかどうかを返します。

`DefaultCookiePolicy` インスタンスは以下の属性をもっています。これらはすべてコンストラクタから同じ名前の引数をつかって初期化することができ、代入してもかまいません。

`DefaultCookiePolicy.rfc2109_as_netscape`

True の場合、`CookieJar` のインスタンスに RFC 2109 クッキー (即ち `Set-Cookie` ヘッダの `Version` 属性の値が 1 のクッキー) を Netscape クッキーへ、`Cookie` インスタンスの `version` 属性を 0 に設定する事でダウングレードするように要求します。デフォルトの値は `None` であり、この場合 RFC 2109 クッキーは RFC 2965 処理が無効に設定されている場合に限りダウングレードされます。それ故に RFC 2109 クッキーはデフォルトではダウングレードされます。

バージョン 2.5 で追加.

一般的な厳密性のスイッチ:

`DefaultCookiePolicy.strict_domain`

サイトに、国別コードとトップレベルドメインだけからなるドメイン名 (`.co.uk`, `.gov.uk`, `.co.nz` など) を設定させないようにします。これは完璧からはほど遠い実装であり、いつもうまくいくとは限りません!

RFC 2965 プロトコルの厳密性に関するスイッチ:

`DefaultCookiePolicy.strict_rfc2965.unverifiable`

検証不可能なトランザクション (通常これはリダイレクトか、別のサイトがホスティングしているイメージの読み込み要求です) に関する RFC 2965 の規則に従います。この値が偽の場合、検証可能性を基準にしてクッキーがブロックされることは決してありません

Netscape プロトコルの厳密性に関するスイッチ:

`DefaultCookiePolicy.strict_ns.unverifiable`

検証不可能なトランザクションに関する RFC 2965 の規則を Netscape クッキーに対しても適用します。

`DefaultCookiePolicy.strict_ns.domain`

Netscape クッキーに対するドメインマッチングの規則をどの程度厳しくするかを指示するフラグです。とりうる値については下の説明を見てください。

`DefaultCookiePolicy.strict_ns.set_initial_dollar`

Set-Cookie: ヘッダで、'\$' で始まる名前のクッキーを無視します。

`DefaultCookiePolicy.strict_ns.set_path`

要求した URI にパスがマッチしないクッキーの設定を禁止します。

`strict_ns.domain` はいくつかのフラグの集合です。これはいくつかの値を or することで構成します (たとえば `DomainStrictNoDots|DomainStrictNonDomain` は両方のフラグが設定されていることになります)。

`DefaultCookiePolicy.DomainStrictNoDots`

クッキーを設定するさい、ホスト名のプレフィクスにドットが含まれるのを禁止します (例: `www.foo.`

`bar.com` は `.bar.com` のクッキーを設定することはできません、なぜなら `www.foo` はドットを含んでいるからです。

`DefaultCookiePolicy.DomainStrictNonDomain`

`domain` クッキー属性を明示的に指定していないクッキーは、そのクッキーを設定したドメインと同一のドメインだけに返されます (例: `example.com` からのクッキーに `domain` クッキー属性がない場合、そのクッキーが `spam.example.com` に返されることはありません)。

`DefaultCookiePolicy.DomainRFC2965Match`

クッキーを設定するさい、RFC 2965 の完全ドメインマッチングを要求します。

以下の属性は上記のフラグのうちもっともよく使われる組み合わせで、便宜をはかるために提供されています:

`DefaultCookiePolicy.DomainLiberal`

0 と同じです (つまり、上述の Netscape のドメイン厳密性フラグがすべてオフにされます)。

`DefaultCookiePolicy.DomainStrict`

`DomainStrictNoDots`|`DomainStrictNonDomain` と同じです。

20.21.5 Cookie オブジェクト

`Cookie` インスタンスは、さまざまなクッキーの標準で規定されている標準的なクッキー属性とおおまかに対応する Python 属性をもっています。しかしデフォルト値を決める複雑なやり方が存在しており、また `max-age` および `expires` クッキー属性は同じ値をもつことになっているので、また RFC 2109 クッキーは `cookieclib` によって version 1 から version 0 (Netscape) クッキーへ 'ダウングレード' される場合があるため、この対応は 1 対 1 ではありません。

`CookiePolicy` メソッド内でのごくわずかな例外を除けば、これらの属性に代入する必要はないはずです。このクラスは内部の一貫性を保つようにはしていないため、代入するのは自分のやっていることを理解している場合のみにしてください。

`Cookie.version`

整数または `None`。Netscape クッキーはバージョン 0 であり、RFC 2965 および RFC 2109 クッキーはバージョン 1 です。しかし、`cookieclib` は RFC 2109 クッキーを Netscape クッキー (`version` が 0) に 'ダウングレード' する場合がある事に注意して下さい。

`Cookie.name`

クッキーの名前 (文字列)。

`Cookie.value`

クッキーの値 (文字列)、あるいは `None`。

`Cookie.port`

ポートあるいはポートの集合をあらわす文字列 (例: `'80'` または `'80,8080'`)、あるいは `None`。

`Cookie.path`

クッキーのパス名 (文字列、例: `'/acme/rocket_launchers'`)。

Cookie.secure

そのクッキーを返せるのがセキュアな接続のみならば `True` を返します。

Cookie.expires

クッキーの期限が切れる日時をあわらす整数 (エポックから経過した秒数)、あるいは `None`。
`is_expired()` も参照してください。

Cookie.discard

これがセッションクッキーであれば `True` を返します。

Cookie.comment

このクッキーの働きを説明する、サーバからのコメント文字列、あるいは `None`。

Cookie.comment_url

このクッキーの働きを説明する、サーバからのコメントのリンク URL、あるいは `None`。

Cookie.rfc2109

RFC 2109 クッキー (即ち *Set-Cookie* ヘッダにあり、かつクッキー属性 `Version` の値が 1 のクッキー) の場合、`True` を返します。`cookielib` が RFC 2109 クッキーを Netscape クッキー (`version` が 0) に 'ダウングレード' する場合があるので、この属性が提供されています。

バージョン 2.5 で追加。

Cookie.port_specified

サーバがポート、あるいはポートの集合を (*Set-Cookie* / *Set-Cookie2* ヘッダ内で) 明示的に指定していれば `True` を返します。

Cookie.domain_specified

サーバにより明示的にドメインが指定されていれば `True` を返します。

Cookie.domain_initial_dot

サーバが明示的に指定したドメインがドット ('.') で始まっていれば `True` を返します。

クッキーは、オプションとして標準的でないクッキー属性を持つこともできます。これらは以下のメソッドでアクセスできます:

Cookie.has_nonstandard_attr (*name*)

そのクッキーが指定された名前のクッキー属性をもっている場合には真を返します。

Cookie.get_nonstandard_attr (*name*, *default=None*)

クッキーが指定された名前のクッキー属性をもっていれば、その値を返します。そうでない場合は *default* を返します。

Cookie.set_nonstandard_attr (*name*, *value*)

指定された名前のクッキー属性を設定します。

`Cookie` クラスは以下のメソッドも定義しています:

Cookie.is_expired (*[now=None]*)

サーバが要求するクッキーの有効期限を過ぎていれば `True` を返します。*now* が (エポックからの経過秒で) 指定されているときは、特定の時刻で期限切れかどうかを判定します。

20.21.6 例

はじめに、もっとも一般的な *cookielib* の使用例をあげます:

```
import cookielib, urllib2
cj = cookielib.CookieJar()
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

以下の例では、URL を開く際に Netscape や Mozilla または Lynx のクッキーを使う方法を示しています (クッキーファイルの位置は Unix/Netscape の慣例にしたがうものと仮定しています):

```
import os, cookielib, urllib2
cj = cookielib.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".netscape", "cookies.txt"))
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

つぎの例は *DefaultCookiePolicy* の使用例です。RFC 2965 クッキーをオンにし、Netscape クッキーを設定したり返したりするドメインに対してより厳密な規則を適用します。そしていくつかのドメインからクッキーを設定あるいは返還するのをブロックしています:

```
import urllib2
from cookielib import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=DefaultCookiePolicy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

20.22 Cookie — HTTP の状態管理

注釈: Python 3 では *Cookie* モジュールは `http.cookies` にリネームされました。ソースコードを 3 用に変換する時は、*2to3* ツールが自動的に `import` を修正します。

ソースコード: [Lib/Cookie.py](#)

Cookie モジュールは HTTP の状態管理機能である *cookie* の概念を抽象化、定義しているクラスです。単純な文字列のみで構成される *cookie* のほか、シリアル化可能なあらゆるデータ型でクッキーの値を保持するための機能も備えています。

このモジュールは元々 **RFC 2109** と **RFC 2068** に定義されている構文解析の規則を厳密に守っていました。しかし、MSIE 3.0x がこれらの RFC で定義された文字の規則に従っていないことが判明し、また、現代の多

くのブラウザとサーバも Cookie の処理に緩い解析をしており、結局、やや厳密さを欠く構文解析規則にせざるを得ませんでした。

`string.ascii_letters`、`string.digits`、`!#$%&'*+-.^_`|~` が、このモジュールが Cookie 名 (*key*) での利用に合法である認めている文字集合です。

注釈: 正しくない cookie に遭遇した場合、`CookieError` 例外を送出します。なので、ブラウザから持ってきた cookie データを parse するときには常に `CookieError` 例外を catch して不正な cookie に備えるべきです。

exception `Cookie.CookieError`

属性や `Set-Cookie` ヘッダが正しくないなど、**RFC 2109** に合致していないときに発生する例外です。

class `Cookie.BaseCookie` (`[input]`)

このクラスはキーが文字列、値が `Morsel` インスタンスで構成される辞書風オブジェクトです。値に対するキーを設定するときは、値がキーと値を含む `Morsel` に変換されることに注意してください。

`input` が与えられたときは、そのまま `load()` メソッドへ渡されます。

class `Cookie.SimpleCookie` (`[input]`)

このクラスは `BaseCookie` の派生クラスで、`value_decode()` は与えられた値の正当性を確認するように、`value_encode()` は `str()` で文字列化するようにそれぞれオーバーライドします。

class `Cookie.SerialCookie` (`[input]`)

このクラスは `BaseCookie` の派生クラスで、`value_decode()` と `value_encode()` をそれぞれ `pickle.loads()` と `pickle.dumps()` を実行するようにオーバーライドします。

バージョン 2.3 で非推奨: このクラスを使ってはいけません! 信頼できない cookie のデータから pickle 化された値を読み込むことは、あなたのサーバ上で任意のコードを実行するために pickle 化した文字列の作成が可能であることを意味し、重大なセキュリティホールとなります。

class `Cookie.SmartCookie` (`[input]`)

このクラスは `BaseCookie` の派生クラスで、`value_decode()` を、値が pickle 化されたデータとして正当なときは `pickle.loads()` を実行、そうでないときはその値自体を返すようにオーバーライドします。また `value_encode()` を、値が文字列以外のときは `pickle.dumps()` を実行、文字列のときはその値自体を返すようにオーバーライドします。

バージョン 2.3 で非推奨: `SerialCookie` と同じセキュリティ上の注意が当てはまります。

関連して、さらなるセキュリティ上の注意があります。後方互換性のため、`Cookie` モジュールは `Cookie` というクラス名を `SmartCookie` のエイリアスとしてエクスポートしています。これはほぼ確実に誤った措置であり、将来のバージョンでは削除することが適当と思われます。アプリケーションにおいて `SerialCookie` クラスを使うべきでないのと同じ理由で `Cookie` クラスを使うべきではありません。

参考:

`cookiecrlib` モジュール Web クライアント 向けの HTTP クッキー処理です。 `cookiecrlib` と `Cookie` は互いに独立しています。

RFC 2109 - HTTP State Management Mechanism このモジュールが実装している HTTP の状態管理に関する規格です。

20.22.1 Cookie オブジェクト

`BaseCookie.value_decode(val)`

文字列表現を値にデコードして返します。戻り値の型はどのようなものでも許されます。このメソッドは `BaseCookie` において何も実行せず、オーバーライドされるためにだけ存在します。

`BaseCookie.value_encode(val)`

エンコードした値を返します。元の値はどのような型でもかまいませんが、戻り値は必ず文字列となります。このメソッドは `BaseCookie` において何も実行せず、オーバーライドされるためにだけ存在します。

通常 `value_encode()` と `value_decode()` はともに `value_decode` の処理内容から逆算した範囲に収まっていなければなりません。

`BaseCookie.output([attrs[, header[, sep]]])`

HTTP ヘッダ形式の文字列表現を返します。 `attrs` と `header` はそれぞれ `Morsel` の `output()` メソッドに送られます。 `sep` はヘッダの連結に用いられる文字で、デフォルトは `'\r\n'` (CRLF) となっています。

バージョン 2.5 で変更: デフォルトのセパレータを `'\n'` から、クッキーの使用にあわせた。

`BaseCookie.js_output([attrs])`

ブラウザが JavaScript をサポートしている場合、HTTP ヘッダを送信した場合と同様に動作する埋め込み可能な JavaScript snippet を返します。

`attrs` の意味は `output()` と同じです。

`BaseCookie.load(rawdata)`

`rawdata` が文字列であれば、HTTP_COOKIE として処理し、その値を `Morsel` として追加します。辞書の場合は次と同様の処理をおこないます。

```
for k, v in rawdata.items():
    cookie[k] = v
```

20.22.2 Morsel オブジェクト

`class Cookie.Morsel`

RFC 2109 の属性をキーと値で保持する abstract クラスです。

`Morsel` は辞書風のオブジェクトで、キーは次のような **RFC 2109** 準拠の定数となっています。

- `expires`

- `path`
- `comment`
- `domain`
- `max-age`
- `secure`
- `version`
- `httponly`

`httponly` 属性は、cookie が HTTP リクエストでのみ送信されて、JavaScript からのアクセスできない事を示します。これはいくつかのクロスサイトスクリプティングの脅威を和らげることを意図しています。

キーの大小文字は区別されません。

バージョン 2.6 で追加: `httponly` 属性が追加されました。

`Morsel.value`

クッキーの値。

`Morsel.coded_value`

実際に送信する形式にエンコードされた cookie の値。

`Morsel.key`

cookie の名前。

`Morsel.set` (*key*, *value*, *coded_value*)

属性 *key*、*value*、*coded_value* に値をセットします。

`Morsel.isReservedKey` (*K*)

K が *Morsel* のキーであるかどうかを判定します。

`Morsel.output` ([*attrs*[, *header*]])

Mosel を HTTP ヘッダ形式の文字列表現にして返します。 *attrs* を指定しない場合、デフォルトですべての属性を含めます。 *attrs* を指定する場合、属性をリストで渡さなければなりません。 *header* のデフォルトは "Set-Cookie:" です。

`Morsel.js_output` ([*attrs*])

ブラウザが JavaScript をサポートしている場合、HTTP ヘッダを送信した場合と同様に動作する埋め込み可能な JavaScript snippet を返します。

attrs の意味は *output* () と同じです。

`Morsel.OutputString` ([*attrs*])

Mosel の文字列表現を HTTP や JavaScript で困まずに出力します。

attrs の意味は *output* () と同じです。

20.22.3 例

次の例は *Cookie* の使い方を示したものです。

```
>>> import Cookie
>>> C = Cookie.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print C # generate HTTP headers
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print C.output() # same thing
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> C = Cookie.SimpleCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print C.output(header="Cookie:")
Cookie: rocky=road; Path=/cookie
>>> print C.output(attrs=[], header="Cookie:")
Cookie: rocky=road
>>> C = Cookie.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger") # load from a string (HTTP header)
>>> print C
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = Cookie.SimpleCookie()
>>> C.load('keebler="E=everybody; L=\\\"Loves\\\"; fudge=\\\"012;\";')
>>> print C
Set-Cookie: keebler="E=everybody; L=\\\"Loves\\\"; fudge=\\\"012;\"
>>> C = Cookie.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print C
Set-Cookie: oreo=doublestuff; Path=/
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
>>> C = Cookie.SimpleCookie()
>>> C["number"] = 7 # equivalent to C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print C
Set-Cookie: number=7
Set-Cookie: string=seven
>>> # SerialCookie and SmartCookie are deprecated
>>> # using it can cause security loopholes in your code.
>>> C = Cookie.SerialCookie()
>>> C["number"] = 7
```

(次のページに続く)

(前のページからの続き)

```

>>> C["string"] = "seven"
>>> C["number"].value
7
>>> C["string"].value
'seven'
>>> print C
Set-Cookie: number="I7\012."
Set-Cookie: string="S'seven'\012p1\012."
>>> C = Cookie.SmartCookie()
>>> C["number"] = 7
>>> C["string"] = "seven"
>>> C["number"].value
7
>>> C["string"].value
'seven'
>>> print C
Set-Cookie: number="I7\012."
Set-Cookie: string=seven

```

20.23 xmlrpclib — XML-RPC クライアントアクセス

注釈: `xmlrpclib` モジュールは、Python 3 では `xmlrpc.client` にリネームされました。 [2to3](#) ツールは、自動的にソースコードの `import` を Python 3 用に修正します。

バージョン 2.2 で追加.

Source code: [Lib/xmlrpclib.py](#)

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP(S) as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data. This module supports writing XML-RPC client code; it handles all the details of translating between conformable Python objects and XML on the wire.

警告: `xmlrpclib` モジュールは悪意を持って構築されたデータに対して安全ではありません。信頼できないデータや認証されていないデータを解析する必要があるばあいは [XML の脆弱性](#) を参照してください。

バージョン 2.7.9 で変更: For HTTPS URIs, `xmlrpclib` now performs all the necessary certificate and hostname checks by default.

```

class xmlrpclib.ServerProxy(uri[, transport[, encoding[, verbose[, allow_none[,
                                use_datetime[, context]]]]]])

```

A `ServerProxy` instance is an object that manages communication with a remote XML-RPC server. The

required first argument is a URI (Uniform Resource Indicator), and will normally be the URL of the server. The optional second argument is a transport factory instance; by default it is an internal `SafeTransport` instance for https: URLs and an internal `HTTP Transport` instance otherwise. The optional third argument is an encoding, by default UTF-8. The optional fourth argument is a debugging flag.

The following parameters govern the use of the returned proxy instance. If `allow_none` is true, the Python constant `None` will be translated into XML; the default behaviour is for `None` to raise a `TypeError`. This is a commonly-used extension to the XML-RPC specification, but isn't supported by all clients and servers; see <http://ontosys.com/xml-rpc/extensions.php> for a description. The `use_datetime` flag can be used to cause date/time values to be presented as `datetime.datetime` objects; this is false by default. `datetime.datetime` objects may be passed to calls.

Both the HTTP and HTTPS transports support the URL syntax extension for HTTP Basic Authentication: `http://user:pass@host:port/path`. The `user:pass` portion will be base64-encoded as an HTTP 'Authorization' header, and sent to the remote server as part of the connection process when invoking an XML-RPC method. You only need to use this if the remote server requires a Basic Authentication user and password. If an HTTPS URL is provided, `context` may be `ssl.SSLContext` and configures the SSL settings of the underlying HTTPS connection.

生成されるインスタンスはリモートサーバへのプロキシオブジェクトで、RPC 呼び出しを行う為のメソッドを持ちます。リモートサーバがイントロスペクション API をサポートしている場合は、リモートサーバのサポートするメソッドを検索 (サービス検索) やサーバのメタデータの取得なども行えます。

Types that are conformable (e.g. that can be marshalled through XML), include the following (and except where noted, they are unmarshalled as the same Python type):

XML-RPC の型	Python の型
boolean	<code>bool</code>
int または i4	<code>int</code> or <code>long</code> in range from -2147483648 to 2147483647.
double	<code>float</code>
string	<code>str</code> or <code>unicode</code>
array	適合する要素を持つ <code>list</code> または <code>tuple</code> 。array は <code>list</code> として返します。
struct	<code>dict</code> . Keys must be strings, values may be any conformable type. Objects of user-defined classes can be passed in; only their <code>__dict__</code> attribute is transmitted.
<code>dateTime.iso8601</code>	<code>DateTime</code> or <code>datetime.datetime</code> . Returned type depends on values of the <code>use_datetime</code> flags.
base64	<code>Binary</code>
nil	<code>None</code> 定数。 <code>allow_none</code> が真の場合にのみ渡すことが出来ます。

上記の XML-RPC でサポートする全データ型を使用することができます。メソッド呼び出し時、XML-RPC サーバエラーが発生すると `Fault` インスタンスを送出し、HTTP/HTTPS トランスポート層でエラーが発生した場合には `ProtocolError` を送냅니다。 `Error` をベースとする `Fault` と `ProtocolError` の両方が発生します。Python 2.2 以降では組み込み型のサブクラスを作成する事が

できますが、現在のところ `xmlrpclib` ではそのようなサブクラスのインスタンスをマーシャルすることはできません。

文字列を渡す場合、`<`, `>`, `&` などの XML で特殊な意味を持つ文字は自動的にエスケープされます。しかし、ASCII 値 0~31 の制御文字 (もちろん、タブ 'TAB', 改行 'LF', リターン 'CR' は除く) などの XML で使用することのできない文字を使用することはできず、使用するとその XML-RPC リクエストは well-formed な XML とはなりません。そのような文字列を渡す必要がある場合は、後述の *Binary* ラッパクラスを使用してください。

`Server` は、後方互換性の為に *`ServerProxy`* の別名として残されています。新しいコードでは *`ServerProxy`* を使用してください。

バージョン 2.5 で変更: `use_datetime` フラグが追加されました

バージョン 2.6 で変更: ニュースタイルクラス (*`new-style class`*) も、`__dict__` 属性を持っていて、特別な方法でマーシャルされている親クラスを持っていなければ、渡すことができます。

バージョン 2.7.9 で変更: `context` 引数が追加されました。

参考:

XML-RPC HOWTO 数種類のプログラミング言語で記述された XML-RPC の操作とクライアントソフトウェアの素晴らしい説明が掲載されています。XML-RPC クライアントの開発者が知っておくべきことがほとんど全て記載されています。

XML-RPC Introspection インストロペクションをサポートする、XML-RPC プロトコルの拡張を解説しています。

XML-RPC Specification 公式の仕様

XML-RPC 非公式正誤表 Fredrik Lundh による "unofficial errata, intended to clarify certain details in the XML-RPC specification, as well as hint at 'best practices' to use when designing your own XML-RPC implementations."

20.23.1 ServerProxy オブジェクト

`ServerProxy` インスタンスの各メソッドはそれぞれ XML-RPC サーバの遠隔手続き呼び出しに対応しており、メソッドが呼び出されると名前と引数をシグネチャとして RPC を実行します (同じ名前のメソッドでも、異なる引数シグネチャによってオーバーロードされます)。RPC 実行後、変換された値を返すか、または *`Fault`* オブジェクトもしくは *`ProtocolError`* オブジェクトでエラーを通知します。

Servers that support the XML introspection API support some common methods grouped under the reserved `system` attribute:

`ServerProxy.system.listMethods()`

XML-RPC サーバがサポートするメソッド名 (`system` 以外) を格納する文字列のリストを返します。

`ServerProxy.system.methodSignature(name)`

XML-RPC サーバで実装されているメソッドの名前を指定し、利用可能なシグネチャの配列を取得しま

す。シグネチャは型のリストで、先頭の型は戻り値の型を示し、以降はパラメータの型を示します。

XML-RPC では複数のシグネチャ (オーバーロード) を使用することができるので、単独のシグネチャではなく、シグネチャのリストを返します。

シグネチャは、メソッドが使用する最上位のパラメータにのみ適用されます。例えばあるメソッドのパラメータが構造体の配列で戻り値が文字列の場合、シグネチャは単に”string, array” となります。パラメータが三つの整数で戻り値が文字列の場合は”string, int, int, int” となります。

メソッドにシグネチャが定義されていない場合、配列以外の値が返ります。Python では、この値は list 以外の値となります。

`ServerProxy.system.methodHelp (name)`

XML-RPC サーバで実装されているメソッドの名前を指定し、そのメソッドを解説する文書文字列を取得します。文書文字列を取得できない場合は空文字列を返します。文書文字列には HTML マークアップが含まれます。

20.23.2 Boolean オブジェクト

このクラスは全ての Python の値で初期化することができ、生成されるインスタンスは指定した値の真偽値によってのみ決まります。Boolean という名前から想像される通りに各種の Python 演算子を実装しており、`--cmp--()`、`--repr--()`、`--int--()`、`--nonzero--()` で定義される演算子を使用することができます。

以下のメソッドは、主に内部的にアンマーシャル時に使用されます:

`Boolean.encode (out)`

出力ストリームオブジェクト `out` に、XML-RPC エンコーディングの Boolean 値を出力します。

動作する例です。サーバー側:

```
import xmlrpclib
from SimpleXMLRPCServer import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_function(is_even, "is_even")
server.serve_forever()
```

上記のサーバーに対するクライアント側:

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
print "3 is even: %s" % str(proxy.is_even(3))
print "100 is even: %s" % str(proxy.is_even(100))
```

20.23.3 DateTime オブジェクト

class xmlrpclib.DateTime

このクラスは、エポックからの秒数、タプルで表現された時刻、ISO 8601 形式の時間/日付文字列、`datetime.datetime`、のインスタンスのいずれかで初期化することができます。このクラスには以下のメソッドがあり、主にコードをマーシャル/アンマーシャルするための内部処理を行います:

decode (*string*)

文字列をインスタンスの新しい時間を示す値として指定します。

encode (*out*)

出力ストリームオブジェクト *out* に、XML-RPC エンコーディングの *DateTime* 値を出力します。

It also supports certain of Python's built-in operators through `--cmp--()` and `--repr--()` methods.

動作する例です。サーバー側:

```
import datetime
from SimpleXMLRPCServer import SimpleXMLRPCServer
import xmlrpclib

def today():
    today = datetime.datetime.today()
    return xmlrpclib.DateTime(today)

server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_function(today, "today")
server.serve_forever()
```

上記のサーバーに対するクライアント側:

```
import xmlrpclib
import datetime

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")

today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print "Today: %s" % converted.strftime("%d.%m.%Y, %H:%M")
```

20.23.4 Binary オブジェクト

class xmlrpclib.Binary

このクラスは、文字列 (NUL を含む) で初期化することができます。 *Binary* の内容は、属性で参照します。

data

Binary インスタンスがカプセル化しているバイナリデータ。このデータは 8bit クリーンです。

Binary オブジェクトは以下のメソッドを持ち、主に内部的にマーシャル/アンマーシャル時に使用されます:

decode (*string*)

指定された base64 文字列をデコードし、インスタンスのデータとします。

encode (*out*)

Write the XML-RPC base 64 encoding of this binary item to the *out* stream object.

The encoded data will have newlines every 76 characters as per [RFC 2045 section 6.8](#), which was the de facto standard base64 specification when the XML-RPC spec was written.

また、`--cmp--()` で定義される演算子を使用することができます。

バイナリオブジェクトの使用例です。XML-RPC ごしに画像を転送します。

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
import xmlrpclib

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpclib.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_function(python_logo, 'python_logo')

server.serve_forever()
```

クライアント側は画像を取得して、ファイルに保存します。

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
```

20.23.5 Fault オブジェクト

class `xmlrpclib.Fault`

Fault オブジェクトは、XML-RPC の *fault* タグの内容をカプセル化しており、以下の属性を持ちます:

faultCode

失敗のタイプを示す文字列。

faultString

失敗の診断メッセージを含む文字列。

以下のサンプルでは、複素数型のオブジェクトを返そうとして、故意に *Fault* を起こしています。

```

from SimpleXMLRPCServer import SimpleXMLRPCServer

# A marshallng error is going to occur because we're returning a
# complex number
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_function(add, 'add')

server.serve_forever()

```

上記のサーバーに対するクライアント側:

```

import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
try:
    proxy.add(2, 5)
except xmlrpclib.Fault as err:
    print "A fault occurred"
    print "Fault code: %d" % err.faultCode
    print "Fault string: %s" % err.faultString

```

20.23.6 ProtocolError オブジェクト

class xmlrpclib.ProtocolError

ProtocolError オブジェクトはトランスポート層で発生したエラー (URI で指定したサーバが見つからなかった場合に発生する 404 'not found' など) の内容を示し、以下の属性を持ちます:

url

エラーの原因となった URI または URL。

errcode

エラーコード。

errmsg

エラーメッセージまたは診断文字列。

headers

エラーの原因となった HTTP/HTTPS リクエストを含む文字列。

In the following example we're going to intentionally cause a *ProtocolError* by providing a URI that doesn't point to an XMLRPC server:

```

import xmlrpclib

# create a ServerProxy with a URI that doesn't respond to XMLRPC requests

```

(次のページに続く)

(前のページからの続き)

```

proxy = xmlrpclib.ServerProxy("http://www.google.com/")

try:
    proxy.some_method()
except xmlrpclib.ProtocolError as err:
    print "A protocol error occurred"
    print "URL: %s" % err.url
    print "HTTP/HTTPS headers: %s" % err.headers
    print "Error code: %d" % err.errcode
    print "Error message: %s" % err.errmsg

```

20.23.7 MultiCall オブジェクト

バージョン 2.4 で追加.

遠隔のサーバに対する複数の呼び出しをひとつのリクエストにカプセル化する方法は、<http://www.xmlrpc.com/discuss/msgReader%241208> で示されています。

```
class xmlrpclib.MultiCall(server)
```

巨大な (boxcar) メソッド呼び出しに使えるオブジェクトを作成します。 *server* には最終的に呼び出しを行う対象を指定します。作成した MultiCall オブジェクトを使って呼び出しを行うと、即座に None を返し、呼び出したい手続き名とパラメタを *MultiCall* オブジェクトに保存するだけに留まります。オブジェクト自体を呼び出すと、それまでに保存しておいたすべての呼び出しを単一の system.multicall リクエストの形で伝送します。呼び出し結果はジェネレータ (*generator*) になります。このジェネレータにわたってイテレーションを行うと、個々の呼び出し結果を返します。

以下にこのクラスの使い方を示します。サーバー側のコード:

```

from SimpleXMLRPCServer import SimpleXMLRPCServer

def add(x, y):
    return x+y

def subtract(x, y):
    return x-y

def multiply(x, y):
    return x*y

def divide(x, y):
    return x/y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')

```

(次のページに続く)

(前のページからの続き)

```
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

上記のサーバーに対するクライアント側:

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
multicall = xmlrpclib.MultiCall(proxy)
multicall.add(7,3)
multicall.subtract(7,3)
multicall.multiply(7,3)
multicall.divide(7,3)
result = multicall()

print "7+3=%d, 7-3=%d, 7*3=%d, 7/3=%d" % tuple(result)
```

20.23.8 補助関数

`xmlrpclib.boolean(value)`

Python の値を、XML-RPC の Boolean 定数 `True` または `False` に変換します。

`xmlrpclib.dumps(params[, methodname[, methodresponse[, encoding[, allow_none]]])`

params を XML-RPC リクエストの形式に変換します。 *methodresponse* が真の場合、XML-RPC レスポンスの形式に変換します。 *params* に指定できるのは、引数からなるタプルか `Fault` 例外クラスのインスタンスです。 *methodresponse* が真の場合、単一の値だけを返します。従って、*params* の長さも 1 でなければなりません。 *encoding* を指定した場合、生成される XML のエンコード方式になります。デフォルトは UTF-8 です。Python の `None` は標準の XML-RPC には利用できません。 `None` を使えるようにするには、*allow_none* を真にして、拡張機能つきにしてください。

`xmlrpclib.loads(data[, use_datetime])`

XML-RPC リクエストまたはレスポンスを (*params*, *methodname*) の形式をとる Python オブジェクトにします。 *params* は引数のタプルです。 *methodname* は文字列で、パケット中にメソッド名がない場合には `None` になります。例外条件を示す XML-RPC パケットの場合には、`Fault` 例外を送出します。 *use_datetime* フラグは `datetime.datetime` のオブジェクトとして日付/時刻を表現する時に使用し、デフォルトでは `false` に設定されています。

バージョン 2.5 で変更: *use_datetime* フラグが追加されました

20.23.9 クライアントのサンプル

```
# simple test program (from the XML-RPC specification)
from xmlrpclib import ServerProxy, Error
```

(次のページに続く)

(前のページからの続き)

```
# server = ServerProxy("http://localhost:8000") # local server
server = ServerProxy("http://betty.userland.com")

print server

try:
    print server.examples.getStateName(41)
except Error as v:
    print "ERROR", v
```

XML-RPC サーバに HTTP プロキシを経由して接続する場合、カスタムトランスポートを定義する必要があります。以下に例を示します:

```
import xmlrpclib, httplib

class ProxiedTransport(xmlrpclib.Transport):
    def set_proxy(self, proxy):
        self.proxy = proxy

    def make_connection(self, host):
        self.realhost = host
        h = httplib.HTTPConnection(self.proxy)
        return h

    def send_request(self, connection, handler, request_body):
        connection.putrequest("POST", 'http://%s%s' % (self.realhost, handler))

    def send_host(self, connection, host):
        connection.putheader('Host', self.realhost)

p = ProxiedTransport()
p.set_proxy('proxy-server:8080')
server = xmlrpclib.ServerProxy('http://time.xmlrpc.com/RPC2', transport=p)
print server.currentTime.getCurrentTime()
```

20.23.10 クライアントとサーバーの利用例

SimpleXMLRPCServer の例 を参照してください。

脚注

20.24 SimpleXMLRPCServer — 基本的な XML-RPC サーバー

注釈: *SimpleXMLRPCServer* モジュールは、Python 3 では `xmlrpc.server` モジュールに統合されました。 *2to3* ツールが自動的にソースコード内の `import` を修正します。

バージョン 2.2 で追加.

Source code: [Lib/SimpleXMLRPCServer.py](#)

The *SimpleXMLRPCServer* module provides a basic server framework for XML-RPC servers written in Python. Servers can either be free standing, using *SimpleXMLRPCServer*, or embedded in a CGI environment, using *CGIXMLRPCRequestHandler*.

```
class SimpleXMLRPCServer.SimpleXMLRPCServer (addr[, requestHandler[, logRe-
                                         quests[, allow_none[, encoding[,
                                         bind_and_activate]]]])
```

新しくサーバーインスタンスを作成します。このクラスは XML-RPC プロトコルで呼ばれる関数の登録のためのメソッドを提供します。引数 *requestHandler* には、リクエストハンドラーインスタンスのファクトリーを設定します。デフォルトは *SimpleXMLRPCRequestHandler* です。引数 *addr* と *requestHandler* は *SocketServer.TCPServer* のコンストラクターに引き渡されます。もし引数 *logRequests* が真 (true) であれば、(それがデフォルトですが、) リクエストはログに記録されます。偽 (false) である場合にはログは記録されません。引数 *allow_none* と *encoding* は *xmlrpclib* に引き継がれ、サーバーから返される XML-RPC レスポンスを制御します。 *bind_and_activate* 引数は、コンストラクタの呼び出し直後に *server_bind()* と *server_activate()* を呼ぶかどうかを指定します。デフォルトでは True です。この引数に False を指定することで、バインドする前に、*allow_reuse_address* クラス変数を操作することができます。(訳注: 同じ名前のインスタンス変数を追加することで、クラス変数をオーバーライドすることができます。)

バージョン 2.5 で変更: 引数 *allow_none* と *encoding* が追加されました。

バージョン 2.6 で変更: *bind_and_activate* 引数が追加されました。

```
class SimpleXMLRPCServer.CGIXMLRPCRequestHandler ([allow_none[, encoding]])
```

CGI 環境における XML-RPC リクエストハンドラーを、新たに作成します。引数 *allow_none* と *encoding* は *xmlrpclib* に引き継がれ、サーバーから返される XML-RPC レスポンスを制御します。

バージョン 2.3 で追加.

バージョン 2.5 で変更: 引数 *allow_none* と *encoding* が追加されました。

```
class SimpleXMLRPCServer.SimpleXMLRPCRequestHandler
```

Create a new request handler instance. This request handler supports POST requests and modifies logging so that the *logRequests* parameter to the *SimpleXMLRPCServer* constructor parameter is honored.

20.24.1 SimpleXMLRPCServer オブジェクト

The *SimpleXMLRPCServer* class is based on *SocketServer.TCPServer* and provides a means of creating simple, stand alone XML-RPC servers.

```
SimpleXMLRPCServer.register_function (function[, name])
```

XML-RPC リクエストに応じる関数を登録します。引数 *name* が与えられている場合はその値が、関数 *function* に関連付けられます。これが与えられない場合は *function.__name__* の値が用いられます。

引数 *name* は通常の文字列でもユニコード文字列でも良く、Python で識別子として正しくない文字 (".", "ピリオドなど) を含んでいても。

`SimpleXMLRPCServer.register_instance(instance[, allow_dotted_names])`

オブジェクトを登録し、そのオブジェクトの `register_function()` で登録されていないメソッドを公開します。もし、*instance* がメソッド `_dispatch()` を定義していれば、`_dispatch()` が、リクエストされたメソッド名とパラメータの組を引数として呼び出されます。そして、`_dispatch()` の戻り値が結果としてクライアントに返されます。その API は `def _dispatch(self, method, params)` (注意: *params* は可変引数リストではありません) です。仕事をするために下位の関数を呼ぶ時には、その関数は `func(*params)` のように呼ばれます。`_dispatch()` の戻り値はクライアントへ結果として返されます。もし、*instance* がメソッド `_dispatch()` を定義していなければ、リクエストされたメソッド名がそのインスタンスに定義されているメソッド名から探されます。

もしオプション引数 *allow_dotted_names* が真 (true) で、インスタンスがメソッド `_dispatch()` を定義していないとき、リクエストされたメソッド名がピリオドを含む場合は、(訳注: 通常の Python でのピリオドの解釈と同様に) 階層的にオブジェクトを探索します。そして、そこで見つかったオブジェクトをリクエストから渡された引数で呼び出し、その戻り値をクライアントに返します。

警告: *allow_dotted_names* オプションを有効にすると、侵入者にあなたのモジュールのグローバル変数にアクセスすることを許し、あなたのコンピュータで任意のコードを実行することを許すことがあります。このオプションは安全な閉じたネットワークでのみお使い下さい。

バージョン 2.3.5, で変更: 2.4.1 *allow_dotted_names* はセキュリティホールを塞ぐために追加されました。以前のバージョンは安全ではありません。

`SimpleXMLRPCServer.register_introspection_functions()`

XML-RPC のイントロスペクション関数、`system.listMethods`、`system.methodHelp`、`system.methodSignature` を登録します。

バージョン 2.3 で追加。

`SimpleXMLRPCServer.register_multicall_functions()`

XML-RPC における複数の要求を処理する関数 `system.multicall` を登録します。

`SimpleXMLRPCRequestHandler.rpc_paths`

この属性値は XML-RPC リクエストを受け付ける URL の正当なパス部分をリストするタプルでなければなりません。これ以外のパスへのリクエストは 404 「そのようなページはありません」 HTTP エラーになります。このタプルが空の場合は全てのパスが正当であると見なされます。デフォルト値は `('/', '/RPC2')` です。

バージョン 2.5 で追加。

`SimpleXMLRPCRequestHandler.encode_threshold`

この属性が `None` でない場合、クライアントが受け付けるのであれば、この値よりも大きいサイズのレスポンスを *gzip transfer encoding* を利用して圧縮されます、デフォルト値は、だいたい TCP の 1 パケットに収まるように 1400 です。

バージョン 2.7 で追加.

SimpleXMLRPCServer の例

サーバーのコード:

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
from SimpleXMLRPCServer import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
server = SimpleXMLRPCServer(("localhost", 8000),
                            requestHandler=RequestHandler)
server.register_introspection_functions()

# Register pow() function; this will use the value of
# pow.__name__ as the name, which is just 'pow'.
server.register_function(pow)

# Register a function under a different name
def adder_function(x, y):
    return x + y
server.register_function(adder_function, 'add')

# Register an instance; all the methods of the instance are
# published as XML-RPC methods (in this case, just 'div').
class MyFuncs:
    def div(self, x, y):
        return x // y

server.register_instance(MyFuncs())

# Run the server's main loop
server.serve_forever()
```

以下のクライアントコードは上のサーバーで使えるようになったメソッドを呼び出します:

```
import xmlrpclib

s = xmlrpclib.ServerProxy('http://localhost:8000')
print s.pow(2,3) # Returns 2**3 = 8
print s.add(2,3) # Returns 5
print s.div(5,2) # Returns 5//2 = 2

# Print list of available methods
print s.system.listMethods()
```

The following *SimpleXMLRPCServer* example is included in the module *Lib/SimpleXMLRPCServer.py*:

```
server = SimpleXMLRPCServer(("localhost", 8000))
server.register_function(pow)
server.register_function(lambda x,y: x+y, 'add')
server.register_multicall_functions()
server.serve_forever()
```

このデモサーバはコマンドラインから起動することができます。

```
python -m SimpleXMLRPCServer
```

上記サーバとお喋りをするクライアントのコード例は *Lib/xmlrpclib.py* に含まれています:

```
server = ServerProxy("http://localhost:8000")
print server
multi = MultiCall(server)
multi.pow(2, 9)
multi.add(5, 1)
multi.add(24, 11)
try:
    for response in multi():
        print response
except Error, v:
    print "ERROR", v
```

このクライアントは以下コマンドを使って直接起動することも出来ます:

```
python -m xmlrpclib
```

20.24.2 CGIXMLRPCRequestHandler

CGIXMLRPCRequestHandler クラスは、Python の CGI スクリプトに送られた XML-RPC リクエストを処理するときに使用できます。

CGIXMLRPCRequestHandler.register_function (*function* [, *name*])

XML-RPC リクエストに応じる関数を登録します。引数 *name* が与えられている場合はその値が、関数 *function* に関連付けられます。これが与えられない場合は *function.__name__* の値が用いられます。引数 *name* は通常の文字列でもユニコード文字列でも良く、Python で識別子として正しくない文字 (".", "ピリオドなど) を含んでもかまいません。

CGIXMLRPCRequestHandler.register_instance (*instance*)

オブジェクトを登録し、そのオブジェクトの *register_function()* で登録されていないメソッドを公開します。もし、*instance* がメソッド *_dispatch()* を定義していれば、*_dispatch()* が、リクエストされたメソッド名とパラメータの組を引数として呼び出されます。そして、*_dispatch()* の戻り値が結果としてクライアントに返されます。もし、*instance* がメソッド *_dispatch()* を定義していなければ、リクエストされたメソッド名がそのインスタンスに定義されているメソッド名から探されます。リクエストされたメソッド名がピリオドを含む場合は、(訳注: 通常の Python でのピリオドの解釈と同様に) 階層的にオブジェクトを探索します。そして、そこで見つかったオブジェクトをリク

エストから渡された引数で呼び出し、その返り値をクライアントに返します。

`CGIXMLRPCRequestHandler.register_introspection_functions()`

XML-RPC のイントロスペクション関数、`system.listMethods`、`system.methodHelp`、`system.methodSignature` を登録します。

`CGIXMLRPCRequestHandler.register_multicall_functions()`

XML-RPC マルチコール関数 `system.multicall` を登録します。

`CGIXMLRPCRequestHandler.handle_request([request_text = None])`

XML-RPC リクエストを処理します。与えられた場合、`request_text` は HTTP サーバが提供する POST データでなければなりません。そうでない場合、標準入力の内容が使われます。

例:

```
class MyFuncs:
    def div(self, x, y): return x // y

handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```

20.25 DocXMLRPCServer — セルフドキュメンティング XML-RPC サーバ

注釈: `DocXMLRPCServer` モジュールは、Python 3 では `xmlrpc.server` モジュールに統合されました。`2to3` ツールは、ソースコード内の `import` を自動的に Python 3 用に修正します。

バージョン 2.3 で追加。

The `DocXMLRPCServer` module extends the classes found in `SimpleXMLRPCServer` to serve HTML documentation in response to HTTP GET requests. Servers can either be free standing, using `DocXMLRPCServer`, or embedded in a CGI environment, using `DocCGIXMLRPCRequestHandler`.

```
class DocXMLRPCServer.DocXMLRPCServer(addr[, requestHandler[, logRequests[, allow_none[, encoding[, bind_and_activate]]]])
```

当たなサーバ・インスタンスを生成します。各パラメータの内容は `SimpleXMLRPCServer.SimpleXMLRPCServer` のものと同じですが、`requestHandler` のデフォルトは `DocXMLRPCRequestHandler` になっています。

```
class DocXMLRPCServer.DocCGIXMLRPCRequestHandler
```

CGI 環境に XML-RPC リクエスト・ハンドラの新たなインスタンスを生成します。

class `DocXMLRPCServer.DocXMLRPCRequestHandler`

Create a new request handler instance. This request handler supports XML-RPC POST requests, documentation GET requests, and modifies logging so that the *logRequests* parameter to the *DocXMLRPCServer* constructor parameter is honored.

20.25.1 DocXMLRPCServer オブジェクト

The *DocXMLRPCServer* class is derived from *SimpleXMLRPCServer.SimpleXMLRPCServer* and provides a means of creating self-documenting, stand alone XML-RPC servers. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

`DocXMLRPCServer.set_server_title(server_title)`

生成する HTML ドキュメントのタイトルをセットします。このタイトルは HTML の title 要素として使われます。

`DocXMLRPCServer.set_server_name(server_name)`

生成する HTML ドキュメントの名前をセットします。この名前は HTML 冒頭の h1 要素に使われます。

`DocXMLRPCServer.set_server_documentation(server_documentation)`

生成する HTML ドキュメントの本文をセットします。この本文はドキュメント中の名前の下にパラグラフとして出力されます。

20.25.2 DocCGIXMLRPCRequestHandler

DocCGIXMLRPCRequestHandler は *SimpleXMLRPCServer.CGIXMLRPCRequestHandler* の派生クラスで、セルフ-ドキュメンティングの手段と XML-RPC CGI スクリプト機能を提供します。HTTP POST リクエストは XML-RPC メソッドの呼び出しとして扱われます。HTTP GET リクエストは pydoc スタイルの HTML ドキュメント生成のリクエストとして扱われます。これはサーバが自分自身のドキュメントを Web ベースで提供可能であることを意味します。

`DocCGIXMLRPCRequestHandler.set_server_title(server_title)`

生成する HTML ドキュメントのタイトルをセットします。このタイトルは HTML の title 要素として使われます。

`DocCGIXMLRPCRequestHandler.set_server_name(server_name)`

生成する HTML ドキュメントの名前をセットします。この名前は HTML 冒頭の h1 要素に使われます。

`DocCGIXMLRPCRequestHandler.set_server_documentation(server_documentation)`

生成する HTML ドキュメントの本文をセットします。この本文はドキュメント中の名前の下にパラグラフとして出力されます。

第 21 章

マルチメディアサービス

この章で記述されているモジュールは、主にマルチメディアアプリケーションに役立つさまざまなアルゴリズムまたはインターフェイスを実装しています。これらのモジュールはインストール時の自由裁量に応じて利用できます:

21.1 audioop — 生の音声データを操作する

audioop モジュールは音声データの便利な操作を含んでいます。このモジュールは、Python 文字列型に保存された 8, 16, 32 ビット幅の符号付き整数でできた音声データを操作します。これは *al* および *sunaudiodev* で使われているのと同じ形式です。特に指定の無いかぎり、スカラ量を表す要素はすべて整数型になっています。

このモジュールは a-LAW、u-LAW そして Intel/DVI ADPCM エンコードをサポートしています。

複雑な操作のうちいくつかはサンプル幅が 16 ビットのデータに対してのみ働きますが、それ以外は常にサンプル幅を操作のパラメタとして (バイト単位で) 渡します。

このモジュールでは以下の変数と関数を定義しています:

exception `audioop.error`

この例外は、未知のサンプル当たりのバイト数を指定した時など、全般的なエラーに対して送出されます。

`audioop.add(fragment1, fragment2, width)`

パラメタとして渡された 2 つのサンプルの和のデータを返します。 *width* はバイト単位のサンプル幅で、1, 2, “4” のいずれかです。両方のデータは同じ長さでなければなりません。オーバーフローした場合は、切り捨てられます。

`audioop.adpcm2lin(adpcmfragment, width, state)`

Intel/DVI ADPCM 形式のデータをリニア (linear) 形式にデコードします。ADPCM 符号化方式の詳細については `lin2adpcm()` の説明を参照して下さい。(sample, newstate) からなるタプルを返し、サンプルは *width* に指定した幅になります。

`audioop.alaw2lin(fragment, width)`

a-LAW 形式のデータをリニア (linear) 形式に変換します。a-LAW 形式は常に 8 ビットのサンプルを使

用するので、ここでは *width* は単に出力データのサンプル幅となります。

バージョン 2.5 で追加。

`audioop.avg(fragment, width)`

データ中の全サンプルの平均値を返します。

`audioop.avgpp(fragment, width)`

データ中の全サンプルの平均 peak-peak 振幅を返します。フィルタリングを行っていない場合、このルーチンの有用性は疑問です。

`audioop.bias(fragment, width, bias)`

元の音声データの各サンプルにバイアスを加算した音声データを返します。オーバーフローした場合はラップアラウンドされます。

`audioop.cross(fragment, width)`

引数に渡したデータ中のゼロ交差回数を返します。

`audioop.findfactor(fragment, reference)`

`rms(add(fragment, mul(reference, -F)))` を最小にするような係数 *F*、すなわち、*reference* に乗算したときにもっとも *fragment* に近くなるような値を返します。*fragment* と *reference* のサンプル幅はいずれも 2 バイトでなければなりません。

このルーチンの実行に要する時間は `len(fragment)` に比例します。

`audioop.findfit(fragment, reference)`

reference を可能な限り *fragment* に一致させようとしします (*fragment* は *reference* より長くなければなりません)。この処理は (概念的には) *fragment* からスライスをいくつか取り出し、それぞれについて `findfactor()` を使って最良な一致を計算し、誤差が最小の結果を選ぶことで実現します。*fragment* と *reference* のサンプル幅は両方とも 2 バイトでなければなりません。(offset, factor) からなるタプルを返します。offset は最適な一致箇所が始まる *fragment* のオフセット値 (整数) で、factor は `findfactor()` の返す係数 (浮動小数点数) です。

`audioop.findmax(fragment, length)`

fragment から、長さが *length* サンプル (バイトではありません!) で最大のエネルギーを持つスライス、すなわち、`rms(fragment[i*2:(i+length)*2])` を最大にするようなスライスを探し、*i* を返します。データのはサンプル幅は 2 バイトでなければなりません。

このルーチンの実行に要する時間は `len(fragment)` に比例します。

`audioop.getsample(fragment, width, index)`

データ中の *index* サンプル目の値を返します。

`audioop.lin2adpcm(fragment, width, state)`

データを 4 ビットの Intel/DVI ADPCM 符号化方式に変換します。ADPCM 符号化方式とは適応符号化方式の一つで、あるサンプルと (可変の) ステップだけ離れたその次のサンプルとの差を 4 ビットの整数で表現する方式です。Intel/DVI ADPCM アルゴリズムは IMA (国際 MIDI 協会) に採用されているので、おそらく標準になるはずです。

state はエンコーダの内部状態が入ったタプルです。エンコーダは (adpcmfrag, newstate) のタ

ブルを返し、次に `lin2adpcm()` を呼び出す時に `newstate` を渡さねばなりません。最初に呼び出す時には `state` に `None` を渡してもかまいません。 `adpcmfrag` は ADPCM で符号化されたデータで、バイト当たり 2 つの 4 ビット値がバックされています。

`audioop.lin2alaw(fragment, width)`

音声データのサンプルを a-LAW エンコーディングに変換し、Python 文字列として返します。a-LAW とは 13 ビットのダイナミックレンジを 8bit だけで表現できる音声エンコーディングです。Sun の音声ハードウェアなどで使われています。

バージョン 2.5 で追加。

`audioop.lin2lin(fragment, width, newwidth)`

サンプル幅を 1、2、4 バイト形式の間で変換します。

注釈: .WAV のような幾つかのオーディオフォーマットでは、16bit と 32bit のサンプルは符号付きですが、8bit のサンプルは符号なしです。そのため、そのようなフォーマットで 8bit に変換する場合は、変換結果に 128 を足さなければなりません:

```
new_frames = audioop.lin2lin(frames, old_width, 1)
new_frames = audioop.bias(new_frames, 1, 128)
```

逆に、8bit から 16bit や 32bit に変換する場合も、同じことが言えます。

`audioop.lin2ulaw(fragment, width)`

音声データのサンプルを u-LAW エンコーディングに変換し、Python 文字列として返します。u-LAW とは 14 ビットのダイナミックレンジを 8bit だけで表現できる音声エンコーディングです。Sun の音声ハードウェアなどで使われています。

`audioop.max(fragment, width)`

音声データ全サンプルの絶対値の最大値を返します。

`audioop.maxpp(fragment, width)`

音声データの最大 peak-peak 振幅を返します。

`audioop.minmax(fragment, width)`

音声データ全サンプル中における最小値と最大値からなるタプルを返します。

`audioop.mul(fragment, width, factor)`

元の音声データの各サンプルに浮動小数点数 `factor` を乗算した音声データを返します。オーバーフローした場合は切り捨てられます。

`audioop.ratecv(fragment, width, nchannels, inrate, outrate, state[, weightA[, weightB]])`

入力したデータのフレームレートを変換します。

`state` は変換ルーチンの内部状態を入れたタプルです。変換ルーチンは (`newfragment`, `newstate`) を返し、次に `ratecv()` を呼び出す時には `newstate` を渡さねばなりません。最初の呼び出しでは `None` を渡します。

引数 *weightA* と *weightB* は単純なデジタルフィルタのパラメタで、デフォルト値はそれぞれ 1 と 0 です。

`audioop.reverse(fragment, width)`

データ内のサンプルの順序を逆転し、変更されたデータを返します。

`audioop.rms(fragment, width)`

データの自乗平均根 (root-mean-square)、すなわち $\sqrt{\sum (S_i^2) / n}$ を返します。

これはオーディオ信号の強度 (power) を測る一つの目安です。

`audioop.tomono(fragment, width, lfactor, rfactor)`

ステレオ音声データをモノラル音声データに変換します。左チャンネルのデータに *lfactor*、右チャンネルのデータに *rfactor* を掛けた後、二つのチャンネルの値を加算して単一チャンネルの信号を生成します。

`audioop.tostereo(fragment, width, lfactor, rfactor)`

モノラル音声データをステレオ音声データに変換します。ステレオ音声データの各サンプル対は、モノラル音声データの各サンプルをそれぞれ左チャンネルは *lfactor* 倍、右チャンネルは *rfactor* 倍して生成します。

`audioop.ulaw2lin(fragment, width)`

u-LAW で符号化されている音声データを線形に符号化された音声データに変換します。u-LAW 符号化は常にサンプル当たり 8 ビットを使うため、*width* は出力音声データのサンプル幅にしか使われません。

`mul()` や `max()` といった操作はモノラルとステレオを区別しない、すなわち全てのデータを平等に扱うということに注意してください。この仕様が問題になるようなら、あらかじめステレオ音声データを二つのモノラル音声データに分割しておき、操作後に再度統合してください。そのような例を以下に示します：

```
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
    rsample = audioop.tomono(sample, width, 0, 1)
    lsample = audioop.mul(lsample, width, lfactor)
    rsample = audioop.mul(rsample, width, rfactor)
    lsample = audioop.tostereo(lsample, width, 1, 0)
    rsample = audioop.tostereo(rsample, width, 0, 1)
    return audioop.add(lsample, rsample, width)
```

もし ADPCM 符号をネットワークパケットの構築に使うて独自のプロトコルをステートレスにしたい場合 (つまり、パケットロスを許容したい場合) は、データだけを送信して、ステートを送信すべきではありません。デコーダに従って *initial* ステート (`lin2adpcm()` に渡される値) を送るべきで、最終状態 (符号化器が返す値) を送るべきではないことに注意してください。もし、`struct.Struct` をバイナリでの状態保存に使用したい場合は、最初の要素 (予測値) を 16bit で符号化し、2 番目の要素 (デルタインデックス) を 8bit で符号化できます。

このモジュールの ADPCM 符号のテストは自分自身に対してのみ行っており、他の ADPCM 符号との間では行っていません。作者が仕様を誤解している部分もあるかもしれませんが、それぞれの標準との間で相互運用できない場合もあり得ます。

`find*()` ルーチンは一見滑稽に見えるかもしれませんが。これらの関数の主な目的はエコー除去 (echo cancellation) にあります。エコー除去を十分高速に行うには、出力サンプル中から最も大きなエネルギーを

持った部分を取り出し、この部分が入力サンプル中のどこにあるかを調べ、入力サンプルから出力サンプル自体を減算します:

```
def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 800)      # one tenth second
    out_test = outputdata[pos*2:]
    in_test = inputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test)
    # Optional (for better cancellation):
    # factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)],
    #                             out_test)
    prefill = '\0'*(pos+ipos)*2
    postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
    outputdata = prefill + audioop.mul(outputdata, 2, -factor) + postfill
    return audioop.add(inputdata, outputdata, 2)
```

21.2 imageop — 生の画像データを操作する

バージョン 2.6 で非推奨: *imageop* モジュールは Python 3 で削除されました。

imageop モジュールは画像に関する便利な演算が含まれています。Python 文字列に保存されている 8 または 32 ビットのピクセルから構成される画像を操作します。これは `gl.lrectwrite()` と *imgfile* モジュールが使用しているものと同じフォーマットです。

このモジュールでは以下の変数と関数を定義しています:

exception *imageop.error*

この例外はピクセル当りの未知のビット数などのすべてのエラーで発生させられます。

imageop.crop (*image*, *psize*, *width*, *height*, *x0*, *y0*, *x1*, *y1*)

image の選択された部分を返します。 *image* は *width* × *height* の大きさで、 *psize* バイトのピクセルから構成されなければなりません。 *x0*, *y0*, *x1* および *y1* は `gl.lrectread()` パラメータと同様です。すなわち、境界は新画像に含まれます。新しい境界は画像の内部である必要はありません。旧画像の外側になるピクセルは値をゼロに設定されます。 *x0* が *x1* より大きければ、新画像は鏡像反転されます。 *y* 軸についても同じことが適用されます。

imageop.scale (*image*, *psize*, *width*, *height*, *newwidth*, *newheight*)

image を大きさ *newwidth* × *newheight* に伸縮させて返します。補間も行われません。ばかばかしいほど単純なピクセルの複製と間引きを行い伸縮させます。そのため、コンピュータで作った画像やディザ処理された画像は伸縮した後見た目が良くありません。

imageop.tovideo (*image*, *psize*, *width*, *height*)

垂直ローパスフィルタ処理を画像全体に行います。それぞれの目標ピクセルを垂直に並んだ二つの元ピクセルの平均を計算することで行います。このルーチンの主な用途としては、画像がインターレース走査のビデオ装置に表示された場合に極端なちらつきを抑えるために用います。そのため、この名前があります。

imageop.grey2mono (*image*, *width*, *height*, *threshold*)

全ピクセルを二値化することによって、深さ 8 ビットのグレースケール画像を深さ 1 ビットの画像へ変換します。処理後の画像は隙間なく詰め込まれ、おそらく `mono2grey()` の引数としてしか使い道がないでしょう。

`imageop.dither2mono (image, width, height)`

(ばかばかしいほど単純な) ディザ処理アルゴリズムを用いて、8 ビットグレースケール画像を 1 ビットのモノクロ画像に変換します。

`imageop.mono2grey (image, width, height, p0, p1)`

1 ビットモノクロ画像を 8 ビットのグレースケールまたはカラー画像に変換します。入力で値ゼロの全てのピクセルは出力では値 `p0` を取り、値 1 の入力ピクセルは出力では値 `p1` を取ります。白黒のモノクロ画像をグレースケールへ変換するためには、値 0 と 255 をそれぞれ渡してください。

`imageop.grey2grey4 (image, width, height)`

ディザ処理を行わずに、8 ビットグレースケール画像を 4 ビットグレースケール画像へ変換します。

`imageop.grey2grey2 (image, width, height)`

ディザ処理を行わずに、8 ビットグレースケール画像を 2 ビットグレースケール画像に変換します。

`imageop.dither2grey2 (image, width, height)`

ディザ処理を行い、8 ビットグレースケール画像を 2 ビットグレースケール画像へ変換します。`dither2mono()` については、ディザ処理アルゴリズムは現在とても単純です。

`imageop.grey42grey (image, width, height)`

4 ビットグレースケール画像を 8 ビットグレースケール画像へ変換します。

`imageop.grey22grey (image, width, height)`

2 ビットグレースケール画像を 8 ビットグレースケール画像へ変換します。

`imageop.backward_compatible`

0 にセットすると、このモジュールの関数は、リトルエンディアンのシステムで以前のバージョンと互換性のない方法でマルチバイトピクセル値を表現するようになります。このモジュールはもともと SGI 向けに書かれたのですが、SGI はビッグエンディアンのシステムであり、この変数を設定しても何の影響もありません。とはいえ、このコードはもともとどこでも動作するように考えて作られたわけではないので、バイトオーダーに関する仮定が相互利用向けではありませんでした。この変数を 0 にすると、リトルエンディアンのシステムではバイトオーダーを反転して、ビッグエンディアンと同じにします。

21.3 aifc — AIFF および AIFC ファイルの読み書き

ソースコード: [Lib/aifc.py](#)

このモジュールは AIFF と AIFF-C ファイルの読み書きをサポートします。AIFF (Audio Interchange File Format) はデジタルオーディオサンプルをファイルに保存するためのフォーマットです。AIFF-C は AIFF の新しいバージョンで、オーディオデータの圧縮に対応しています。

注釈: 操作のいくつかは IRIX 上でのみ動作します; そういう操作では IRIX でのみ利用できる `cl` モジュールをインポートしようとして、`ImportError` を発生します。

オーディオファイルには、オーディオデータについて記述したパラメータがたくさん含まれています。サンプリングレートあるいはフレームレートは、1 秒あたりのオーディオサンプル数です。チャンネル数は、モノラル、ステレオ、4 チャンネルかどうかを示します。フレームはそれぞれ、チャンネルごとに一つのサンプルからなります。サンプルサイズは、一つのサンプルの大きさをバイト数で示したものです。したがって、一つのフレームは $nchannels * samplesize$ バイト からなり、1 秒間では $nchannels * samplesize * framerate$ バイト で構成されます。

例えば、CD 品質のオーディオは 2 バイト (16 ビット) のサンプルサイズを持っていて、2 チャンネル (ステレオ) であり、44,100 フレーム / 秒のフレームレートを持っています。そのため、フレームサイズは 4 バイト ($2 * 2$) で、1 秒間では $2 * 2 * 44100$ バイト (176,400 バイト) になります。

`aifc` モジュールは以下の関数を定義しています:

`aifc.open(file[, mode])`

AIFF あるいは AIFF-C ファイルを開き、後述するメソッドを持つインスタンスを返します。引数 *file* はファイルを示す文字列か、[ファイルオブジェクト](#) のいずれかです。mode は、読み込み用に開くときには 'r' か 'rb' のどちらかで、書き込み用に開くときには 'w' か 'wb' のどちらかでなければなりません。もし省略されたら、`file.mode` が存在すればそれが使用され、なければ 'rb' が使われます。書き込み用にこのメソッドを使用するときには、これから全部でどれだけのサンプル数を書き込むのか分からなかったり、`writeframesraw()` と `setnframes()` を使わないなら、ファイルオブジェクトはシーク可能でなければなりません。

ファイルが `open()` によって読み込み用に開かれたときに返されるオブジェクトには、以下のメソッドがあります:

`aifc.getnchannels()`

オーディオチャンネル数 (モノラルなら 1、ステレオなら 2) を返します。

`aifc.getsampwidth()`

サンプルサイズをバイト数で返します。

`aifc.getframerate()`

サンプリングレート (1 秒あたりのオーディオフレーム数) を返します。

`aifc.getnframes()`

ファイルの中のオーディオフレーム数を返します。

`aifc.getcomptype()`

オーディオファイルで使用されている圧縮形式を示す 4 バイトのバイト文字列を返します。AIFF ファイルでは 'NONE' が返されます。

`aifc.getcompname()`

オーディオファイルの圧縮形式を人に判読可能な記述を返します。AIFF ファイルでは 'not compressed' が返されます。

`aifc.getparams()`

以上の全ての値を上の順に並べたタプルを返します。

`aifc.getmarkers()`

オーディオファイルのマーカのリストを返します。一つのマーカは三つの要素のタプルです。要素の 1 番目はマーク ID (整数)、2 番目はマーク位置のフレーム数をデータの始めから数えた値 (整数)、3 番目はマークの名称 (文字列) です。

`aifc.getmark(id)`

与えられた `id` のマークの要素を `getmarkers()` で述べたタプルで返します。

`aifc.readframes(nframes)`

オーディオファイルの次の `nframes` 個のフレームを読み込んで返します。返されるデータは、全チャンネルの圧縮されていないサンプルをフレームごとに文字列にしたものです。

`aifc.rewind()`

読み込むポインタをデータの始めに巻き戻します。次に `readframes()` を使用すると、データの始めから読み込みます。

`aifc.setpos(pos)`

指定したフレーム数の位置にポインタを設定します。

`aifc.tell()`

現在のポインタのフレーム位置を返します。

`aifc.close()`

AIFF ファイルを閉じます。このメソッドを呼び出したあとでは、オブジェクトはもう使用できません。

ファイルが `open()` によって書き込み用に開かれたときに返されるオブジェクトには、`readframes()` と `setpos()` を除く上述の全てのメソッドがあります。さらに以下のメソッドが定義されています。 `get*()` メソッドは、対応する `set*()` を呼び出したあとでのみ呼び出し可能です。最初に `writeframes()` あるいは `writeframesraw()` を呼び出す前に、フレーム数を除く全てのパラメータが設定されていなければなりません。

`aifc.aiff()`

AIFF ファイルを作ります。デフォルトでは AIFF-C ファイルが作られますが、ファイル名が `'.aiff'` で終わっていれば AIFF ファイルが作られます。

`aifc.aifc()`

AIFF-C ファイルを作ります。デフォルトでは AIFF-C ファイルが作られますが、ファイル名が `'.aiff'` で終わっていれば AIFF ファイルが作られます。

`aifc.setnchannels(nchannels)`

オーディオファイルのチャンネル数を設定します。

`aifc.setsampwidth(width)`

オーディオのサンプルサイズをバイト数で設定します。

`aifc.setframerate(rate)`

サンプリングレートを 1 秒あたりのフレーム数で設定します。

`aifc.setnframes(nframes)`

オーディオファイルに書き込まれるフレーム数を設定します。もしこのパラメータが設定されていなかったり正しくなかったら、ファイルはシークに対応していなければなりません。

`aifc.setcomptype(type, name)`

圧縮形式を設定します。もし設定しなければ、オーディオデータは圧縮されません。AIFF ファイルは圧縮できません。変数 `name` は圧縮形式を人に判読可能にしたもので、変数 `type` は 4 文字の文字列でなければなりません。現在のところ、以下の圧縮形式がサポートされています：NONE, ULAW, ALAW, G722。

`aifc.setparams(nchannels, sampwidth, framerate, comptype, compname)`

上の全パラメータを一度に設定します。引数はそれぞれのパラメータからなるタプルです。つまり、`setparams()` の引数として、`getparams()` を呼び出した結果を使うことができます。

`aifc.setmark(id, pos, name)`

指定した ID (1 以上)、位置、名称でマークを加えます。このメソッドは、`close()` の前ならいつでも呼び出すことができます。

`aifc.tell()`

出力ファイルの現在の書き込み位置を返します。`setmark()` との組み合わせで使うと便利です。

`aifc.writeframes(data)`

出力ファイルにデータを書き込みます。このメソッドは、オーディオファイルのパラメータを設定したあとでのみ呼び出し可能です。

`aifc.writeframesraw(data)`

オーディオファイルのヘッダ情報が更新されないことを除いて、`writeframes()` と同じです。

`aifc.close()`

AIFF ファイルを閉じます。ファイルのヘッダ情報は、オーディオデータの実際のサイズを反映して更新されます。このメソッドを呼び出したあとでは、オブジェクトはもう使用できません。

21.4 sunau — Sun AU ファイルの読み書き

ソースコード: `Lib/sunau.py`

`sunau` モジュールは、Sun AU サウンドフォーマットへの便利なインターフェースを提供します。このモジュールは、`aifc` モジュールや `wave` モジュールと互換性のあるインターフェースを備えています。

オーディオファイルはヘッダとそれに続くデータから構成されます。ヘッダのフィールドは以下の通りです：

フィールド	内容
magic word	4 バイト文字列 <code>.snd</code> 。
header size	<code>info</code> を含むヘッダのサイズをバイト数で示したもの。
data size	データの物理サイズをバイト数で示したもの。
encoding	オーディオサンプルのエンコード形式。
sample rate	サンプリングレート。
# of channels	サンプルのチャンネル数。
info	オーディオファイルについての説明を ASCII 文字列で示したもの（null バイトで埋められます）。

`info` フィールド以外の全てのヘッダフィールドは 4 バイトの大きさです。ヘッダフィールドは big-endian でエンコードされた、計 32 ビットの符合なし整数です。

`sunau` モジュールは以下の関数を定義しています:

`sunau.open (file, mode)`

`file` が文字列ならその名前のファイルを開き、そうでないならファイルのようにシーク可能なオブジェクトとして扱います。`mode` は以下のうちのいずれかです

'r' 読み込みのみのモード。

'w' 書き込みのみのモード。

読み込み / 書き込み両方のモードで開くことはできないことに注意して下さい。

'r' の `mode` は `AU_read` オブジェクトを返し、'w' と 'wb' の `mode` は `AU_write` オブジェクトを返します。

`sunau.openfp (file, mode)`

`open()` と同義。後方互換性のために残されています。

`sunau` モジュールは以下の例外を定義しています:

exception `sunau.Error`

Sun AU の仕様や実装に対する不適切な操作により何か実行不可能となった時に発生するエラー。

`sunau` モジュールは以下のデータアイテムを定義しています:

`sunau.AUDIO_FILE_MAGIC`

big-endian で保存された正規の Sun AU ファイルは全てこの整数で始まります。これは文字列 `.snd` を整数に変換したものです。

`sunau.AUDIO_FILE_ENCODING_MULAW_8`

`sunau.AUDIO_FILE_ENCODING_LINEAR_8`

`sunau.AUDIO_FILE_ENCODING_LINEAR_16`

`sunau.AUDIO_FILE_ENCODING_LINEAR_24`

`sunau.AUDIO_FILE_ENCODING_LINEAR_32`

`sunau.AUDIO_FILE_ENCODING_ALAW_8`

AU ヘッダの encoding フィールドの値で、このモジュールでサポートしているものです。

`sunau.AUDIO_FILE_ENCODING_FLOAT`

`sunau.AUDIO_FILE_ENCODING_DOUBLE`

`sunau.AUDIO_FILE_ENCODING_ADPCM_G721`

`sunau.AUDIO_FILE_ENCODING_ADPCM_G722`

`sunau.AUDIO_FILE_ENCODING_ADPCM_G723_3`

`sunau.AUDIO_FILE_ENCODING_ADPCM_G723_5`

AU ヘッダの encoding フィールドの値のうち既知のものとして追加されているものですが、このモジュールではサポートされていません。

21.4.1 AU_read オブジェクト

上述の `open()` によって返される AU_read オブジェクトには、以下のメソッドがあります:

`AU_read.close()`

ストリームを閉じ、このオブジェクトのインスタンスを使用できなくします。(これはオブジェクトのガベージコレクション時に自動的に呼び出されます。)

`AU_read.getnchannels()`

オーディオチャンネル数 (モノラルなら 1、ステレオなら 2) を返します。

`AU_read.getsampwidth()`

サンプルサイズをバイト数で返します。

`AU_read.getframerate()`

サンプリングレートを返します。

`AU_read.getnframes()`

オーディオフレーム数を返します。

`AU_read.getcomptype()`

圧縮形式を返します。'ULAW', 'ALAW', 'NONE' がサポートされている形式です。

`AU_read.getcompname()`

`getcomptype()` を人に判読可能な形にしたものです。上述の形式に対して、それぞれ 'CCITT G.711 u-law', 'CCITT G.711 A-law', 'not compressed' がサポートされています。

`AU_read.getparams()`

`get*()` メソッドが返すのと同じ (`nchannels`, `sampwidth`, `framerate`, `nframes`, `comptype`, `compname`) のタプルを返します。

`AU_read.readframes(n)`

n 個のオーディオフレームの値を読み込んで、バイトごとに文字に変換した文字列を返します。データは linear 形式で返されます。もし元のデータが u-LAW 形式なら、変換されます。

`AU_read.rewind()`

ファイルのポインタをオーディオストリームの先頭に戻します。

以下の 2 つのメソッドは共通の”位置”を定義しています。”位置”は他の関数とは独立して実装されています。

`AU_read.setpos(pos)`

ファイルのポインタを指定した位置に設定します。 `tell()` で返される値を `pos` として使用しなければなりません。

`AU_read.tell()`

ファイルの現在のポインタ位置を返します。返される値はファイルの実際の位置に対して何も操作はしません。

以下の 2 つのメソッドは `aifc` モジュールとの互換性のために定義されていますが、何も面白いことはしません。

`AU_read.getmarkers()`

`None` を返します。

`AU_read.getmark(id)`

エラーを発生します。

21.4.2 AU_write オブジェクト

上述の `open()` によって返される `Wave_write` オブジェクトには、以下のメソッドがあります:

`AU_write.setnchannels(n)`

チャンネル数を設定します。

`AU_write.setsampwidth(n)`

サンプルサイズを (バイト数で) 設定します。

`AU_write.setframerate(n)`

フレームレートを設定します。

`AU_write.setnframes(n)`

フレーム数を設定します。あとからフレームが書き込まれるとフレーム数は変更されます。

`AU_write.setcomptype(type, name)`

圧縮形式とその記述を設定します。'NONE' と 'ULAW' だけが、出力時にサポートされている形式です。

`AU_write.setparams(tuple)`

`tuple` は `(nchannels, sampwidth, framerate, nframes, comptype, compname)` で、それぞれ `set*()` のメソッドの値にふさわしいものでなければなりません。全ての変数を設定します。

`AU_write.tell()`

ファイルの中の現在位置を返します。 `AU_read.tell()` と `AU_read.setpos()` メソッドでお断りしたことがこのメソッドにも当てはまります。

`AU_write.writeframesraw(data)`

`nframes` の修正なしにオーディオフレームを書き込みます。

`AU_write.writeframes(data)`

オーディオフレームを書き込んで `nframes` を修正します。

`AU_write.close()`

`nframes` が正しいか確認して、ファイルを閉じます。

このメソッドはオブジェクトの削除時に呼び出されます。

`writeframes()` や `writeframesraw()` メソッドを呼び出したあとで、どんなパラメータを設定しようとしても不正となることに注意して下さい。

21.5 wave — WAV ファイルの読み書き

ソースコード: [Lib/wave.py](#)

`wave` モジュールは、WAV サウンドフォーマットへの便利なインターフェイスを提供するモジュールです。このモジュールは圧縮 / 展開をサポートしていませんが、モノラル / ステレオには対応しています。

`wave` モジュールは、以下の関数と例外を定義しています:

`wave.open(file[, mode])`

`file` が文字列ならその名前のファイルを開き、そうでないならシーク可能な file like オブジェクトとして扱います。 `mode` は以下のうちのいずれかです。

'r', 'rb' 読み込みのみのモード。

'w', 'wb' 書き込みのみのモード。

WAV ファイルに対して読み込み / 書き込み両方のモードで開くことはできないことに注意して下さい。

'r' と 'rb' の `mode` は `Wave_read` オブジェクトを返し、'w' と 'wb' の `mode` は `Wave_write` オブジェクトを返します。 `mode` が省略されていて、ファイルのようなオブジェクトが `file` として渡されると、 `file.mode` が `mode` のデフォルト値として使われます (必要であれば、さらにフラグ 'b' が付け加えられます)。

file like オブジェクトを渡した場合、 `wave` オブジェクトの `close()` メソッドを呼び出してもその file like オブジェクトを `close` しません。 file like オブジェクトの `close` は呼び出し側の責任になります。

`wave.openfp(file, mode)`

`open()` と同義。後方互換性のために残されています。

exception `wave.Error`

WAV の仕様を犯したり、実装の欠陥に遭遇して何か実行不可能となった時に発生するエラー。

21.5.1 Wave.read オブジェクト

`open()` によって返される `Wave.read` オブジェクトには、以下のメソッドがあります:

`Wave.read.close()`

`wave` によって開かれていた場合はストリームを閉じ、このオブジェクトのインスタンスを使用できなくします。これはオブジェクトのガベージコレクション時に自動的に呼び出されます。

`Wave.read.getnchannels()`

オーディオチャンネル数 (モノラルなら 1、ステレオなら 2) を返します。

`Wave.read.getsampwidth()`

サンプルサイズをバイト数で返します。

`Wave.read.getframerate()`

サンプリングレートを返します。

`Wave.read.getnframes()`

オーディオフレーム数を返します。

`Wave.read.getcomptype()`

圧縮形式を返します ('NONE' だけがサポートされている形式です)。

`Wave.read.getcompname()`

`getcomptype()` を人に判読可能な形にしたものです。通常、'NONE' に対して 'not compressed' が返されます。

`Wave.read.getparams()`

`get*()` メソッドが返すのと同じ (`nchannels`, `sampwidth`, `framerate`, `nframes`, `comptype`, `compname`) のタプルを返します。

`Wave.read.readframes(n)`

現在のポインタから `n` 個のオーディオフレームの値を読み込んで、バイトごとに文字に変換して文字列を返します。

`Wave.read.rewind()`

ファイルのポインタをオーディオストリームの先頭に戻します。

以下の 2 つのメソッドは `aifc` モジュールとの互換性のために定義されており、何も面白いことはしません。

`Wave.read.getmarkers()`

`None` を返します。

`Wave.read.getmark(id)`

エラーを発生します。

以下の 2 つのメソッドは共通の”位置”を定義しています。”位置”は他の関数とは独立して実装されています。

`Wave.read.setpos(pos)`

ファイルのポインタを指定した位置に設定します。

`Wave_read.tell()`

ファイルの現在のポインタ位置を返します。

21.5.2 Wave_write オブジェクト

`open()` によって返される `Wave_write` オブジェクトには、以下のメソッドがあります:

`Wave_write.close()`

`nframes` が正しいか確認して、ファイルが `wave` によって開かれていた場合は閉じます。このメソッドはオブジェクトがガベージコレクションされるときに呼び出されます。

`Wave_write.setnchannels(n)`

チャンネル数を設定します。

`Wave_write.setsampwidth(n)`

サンプルサイズを `n` バイトに設定します。

`Wave_write.setframerate(n)`

サンプリングレートを `n` に設定します。

`Wave_write.setnframes(n)`

フレーム数を `n` に設定します。あとからフレームが書き込まれるとフレーム数は変更されます。

`Wave_write.setcomptype(type, name)`

圧縮形式とその記述を設定します。現在のところ、非圧縮を示す圧縮形式 `NONE` だけがサポートされています。

`Wave_write.setparams(tuple)`

この `tuple` は (`nchannels`, `sampwidth`, `framerate`, `nframes`, `comptype`, `compname`) でなければならず、その値はそれぞれの `set*()` メソッドで有効でなければなりません。すべてのパラメータを設定します。

`Wave_write.tell()`

ファイルの中の現在位置を返します。 `Wave_read.tell()` と `Wave_read.setpos()` メソッドでお断りしたことがこのメソッドにも当てはまります。

`Wave_write.writeframesraw(data)`

`nframes` の修正なしにオーディオフレームを書き込みます。

`Wave_write.writeframes(data)`

オーディオフレームを書き込んで `nframes` を修正します。

`writeframes()` や `writeframesraw()` メソッドを呼び出したあとで、どんなパラメータを設定しようとしても不正となることに注意して下さい。そうすると `wave.Error` を発生します。

21.6 chunk — IFF チャンクデータの読み込み

このモジュールは EA IFF 85 チャンクを使用しているファイルの読み込みのためのインターフェースを提供します。^{*1} このフォーマットは少なくとも、Audio Interchange File Format (AIFF/AIFF-C) と Real Media File Format (RMFF) で使われています。WAVE オーディオファイルフォーマットも厳密に対応しているので、このモジュールで読み込みできます。

チャンクは以下の構造を持っています:

Offset 値	長さ	内容
0	4	チャンク ID
4	4	big- endian で示したチャンクのサイズで、ヘッダは含みません
8	n	バイトデータで、 n はこれより先のフィールドのサイズ
$8 + n$	0 or 1	n が奇数ならチャンクの整頓のために埋められるバイト

ID はチャンクの種類を識別する 4 バイトの文字列です。

サイズフィールド (big-endian でエンコードされた 32 ビット値) は、8 バイトのヘッダを含まないチャンクデータのサイズを示します。

普通、IFF タイプのファイルは 1 個かそれ以上のチャンクからなります。このモジュールで定義される `Chunk` クラスの使い方として提案しているのは、それぞれのチャンクの始めにインスタンスを作り、終わりに達するまでそのインスタンスから読み取り、その後で新しいインスタンスを作るということです。ファイルの終わりで新しいインスタンスを作ろうとすると、`EOFError` の例外が発生して失敗します。

class `chunk.Chunk` (`file` [, `align`, `bigendian`, `inclheader`])

チャンクを表わすクラス。 `file` 引数はファイル風オブジェクトであると期待されます。このクラスのインスタンスは特別に許可されます。唯一の必要なメソッドは `read()` です。メソッド `seek()` および `tell()` が存在し、例外を上げない場合、それらも使用されます。これらのメソッドが存在し、例外を上げる場合、それらのメソッドはオブジェクトを変更しないことが想定されます。オプションの引数 `align` が `true` の場合、チャンクは 2 バイト境界上で整列されていると仮定されます。 `align` が `false` の場合、整列は仮定されません。デフォルト値は `true` です。オプションの引数 `bigendian` が `false` の場合、チャンクサイズはリトルエンディアン順になっていると仮定されます。これは WAVE オーディオファイルに必要とされます。デフォルト値は `true` です。オプションの引数 `inclheader` が `true` の場合、チャンクヘッダ中で与えられたサイズはヘッダのサイズを含んでいます。デフォルト値は `false` です。

`Chunk` オブジェクトには以下のメソッドが定義されています:

getname()

チャンクの名前 (ID) を返します。これはチャンクの始めの 4 バイトです。

getsize()

チャンクのサイズを返します。

^{*1} "EA IFF 85" Standard for Interchange Format Files, Jerry Morrison, Electronic Arts, January 1985.

close()

オブジェクトを閉じて、チャンクの終わりまで飛びます。これは元のファイル自体は閉じません。

close() メソッドが呼ばれた後で他のメソッドを呼ぶと *IOError* が送出されます。

isatty()

`False` を返します。

seek(*pos* [, *whence*])

チャンクの現在位置を設定します。引数 *whence* は省略可能で、デフォルト値は 0 (ファイルの絶対位置) です; 他に 1 (現在位置から相対的にシークします) と 2 (ファイルの末尾から相対的にシークします) の値を取ります。何も値は返しません。もし元のファイルがシークに対応していなければ、前方へのシークのみが可能です。

tell()

チャンク内の現在位置を返します。

read([*size*])

チャンクから最大で *size* バイト (*size* バイトを読み込むまで、少なくともチャンクの最後に行き着くまで) 読み込みます。もし引数 *size* が負か省略されたら、チャンクの最後まで全てのデータを読み込みます。バイト値は文字列のオブジェクトとして返されます。チャンクの最後に行き着いたら、空文字列を返します。

skip()

チャンクの最後まで飛びます。さらにチャンクの *read()* を呼び出すと、`''` が返されます。もしチャンクの内容に興味がないなら、このメソッドを呼び出してファイルポインタを次のチャンクの始めに設定します。

21.7 colorsys — 色体系間の変換

ソースコード: [Lib/colors.py](#)

colorsys モジュールは、計算機のディスプレイモニタで使われている RGB (Red Green Blue) 色空間で表された色と、他の 3 種類の色座標系: YIQ, HLS (Hue Lightness Saturation: 色相、彩度、飽和) および HSV (Hue Saturation Value: 色相、彩度、明度) との間の双方向の色値変換を定義します。これらの色空間における色座標系は全て浮動小数点数で表されます。YIQ 空間では、Y 軸は 0 から 1 ですが、I および Q 軸は正の値も負の値もとります。他の色空間では、各軸は全て 0 から 1 の値をとります。

参考:

More information about color spaces can be found at <https://www.poynton.com/ColorFAQ.html> and <https://www.cambridgeincolour.com/tutorials/color-spaces.htm>.

colorsys モジュールでは、以下の関数が定義されています:

`colorsys.rgb_to_yiq(r, g, b)`

RGB から YIQ に変換します。

`colorsys.yiq_to_rgb(y, i, q)`
YIQ から RGB に変換します。

`colorsys.rgb_to_hls(r, g, b)`
RGB から HLS に変換します。

`colorsys.hls_to_rgb(h, l, s)`
HLS から RGB に変換します。

`colorsys.rgb_to_hsv(r, g, b)`
RGB から HSV に変換します。

`colorsys.hsv_to_rgb(h, s, v)`
HSV から RGB に変換します。

例:

```
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

21.8 imghdr — 画像の形式を決定する

ソースコード: [Lib/imghdr.py](#)

`imghdr` モジュールはファイルやバイトストリームに含まれる画像の形式を決定します。

`imghdr` モジュールは次の関数を定義しています:

`imghdr.what(filename[, h])`

`filename` という名前のファイル内の画像データをテストし、画像形式を表す文字列を返します。オプションの `h` が与えられた場合は、`filename` は無視され、テストするバイトストリームを含んでいると `h` は仮定されます。

以下に `what()` からの戻り値とともにリストするように、次の画像形式が認識されます:

値	Image format
'rgb'	SGI ImgLib Files
'gif'	GIF 87a and 89a Files
'pbm'	Portable Bitmap Files
'pgm'	Portable Graymap Files
'ppm'	Portable Pixmap Files
'tiff'	TIFF Files
'rast'	Sun Raster Files
'xbm'	X Bitmap Files
'jpeg'	JPEG data in JFIF or Exif formats
'bmp'	BMP files
'png'	Portable Network Graphics

バージョン 2.5 で追加: Exif detection.

この変数に追加することで、あなたは `imghdr` が認識できるファイル形式のリストを拡張できます:

`imghdr.tests`

個別のテストを行う関数のリスト。それぞれの関数は二つの引数をとります: バイトストリームとオープンされたファイルのようにふるまうオブジェクト。 `what()` がバイトストリームとともに呼び出されたときは、ファイルのようにふるまうオブジェクトは `None` でしょう。

テストが成功した場合は、テスト関数は画像形式を表す文字列を返すべきです。あるいは、失敗した場合は `None` を返すべきです。

例:

```
>>> import imghdr
>>> imghdr.what('bass.gif')
'gif'
```

21.9 `sndhdr` — サウンドファイルの識別

ソースコード: `Lib/sndhdr.py`

`sndhdr` モジュールには、ファイルに保存されたサウンドデータの形式を識別するのに便利な関数が定義されています。どんな形式のサウンドデータがファイルに保存されているのか識別可能な場合、これらの関数は `(type, sampling_rate, channels, frames, bits_per_sample)` のタプルを返します。 `type` はデータの形式を示す文字列で、 `'aifc'`, `'aiff'`, `'au'`, `'hcom'`, `'sndr'`, `'sndt'`, `'voc'`, `'wav'`, `'8svx'`, `'sb'`, `'ub'`, `'ul'` のうちの一つです。 `sampling_rate` は実際のサンプリングレート値で、未知の場合や読み取ることが出来なかった場合は `0` です。同様に、 `channels` はチャンネル数で、識別できない場合や読み取ることが出来なかった場合は `0` です。 `frames` はフレーム数で、識別できない場合は `-1` です。タプルの最後の要素 `bits_per_sample` はサンプルサイズを示すビット数ですが、A-LAW なら `'A'`, u-LAW なら `'U'` です。

`sndhdr.what(filename)`

`whathdr()` を使って、ファイル `filename` に保存されたサウンドデータの形式を識別します。識別可能なら上記のタプルを返し、識別できない場合は `None` を返します。

`sndhdr.whathdr(filename)`

ファイルのヘッダ情報をもとに、保存されたサウンドデータの形式を識別します。ファイル名は `filename` で渡されます。識別可能なら上記のタプルを返し、識別できない場合は `None` を返します。

21.10 ossaudiodev — OSS 互換オーディオデバイスへのアクセス

バージョン 2.3 で追加。

このモジュールを使うと OSS (Open Sound System) オーディオインターフェースにアクセスできます。OSS はオープンソースあるいは商用の Unix で広く利用でき、Linux (カーネル 2.4 まで) と FreeBSD で標準のオーディオインターフェースです。

参考:

[Open Sound System Programmer's Guide](#) OSS C API の公式ドキュメント

このモジュールでは、OSS デバイスドライバが提供している大量の定数が定義されています; 一覧は Linux や FreeBSD 上の `<sys/soundcard.h>` を参照してください。

`ossaudiodev` では以下の変数と関数を定義しています:

exception `ossaudiodev.OSSAudioError`

何らかのエラーのときに送出される例外です。引数は何が誤っているかを示す文字列です。

(`ossaudiodev` が `open()`、`write()`、`ioctl()` などのシステムコールからエラーを受け取った場合、`IOError` を送出します。 `ossaudiodev` によって直接検知されたエラーでは `OSSAudioError` になります。)

(以前のバージョンとの互換性のため、この例外クラスは `ossaudiodev.error` としても利用できます。)

`ossaudiodev.open(mode)`

`ossaudiodev.open(device, mode)`

オーディオデバイスを開き、OSS オーディオデバイスオブジェクトを返します。このオブジェクトは `read()`、`write()`、`fileno()` といったファイル類似オブジェクトのメソッドを数多くサポートしています。(とはいえ、伝統的な Unix の `read/write` における意味づけと OSS デバイスの `read/write` との間には微妙な違いがあります)。また、オーディオ特有の多くのメソッドがあります; メソッドの完全なリストについては下記を参照してください。

`device` は使用するオーディオデバイスファイルネームです。もしこれが指定されないなら、このモジュールは使うデバイスとして最初に環境変数 `AUDIODEV` を参照します。見つからなければ `/dev/dsp` を参照します。

`mode` は読み出し専用アクセスの場合には `'r'`、書き込み専用 (ブレイバック) アクセスの場合には `'w'`、読み書きアクセスの場合には `'rw'` にします。多くのサウンドカードは一つのプロセスが一度に

レコーダとプレーヤのどちらかしか開けないようにしているため、必要な操作に応じたデバイスだけを開くようにするのがよいでしょう。また、サウンドカードには半二重 (half- duplex) 方式のものがあります: こうしたカードでは、デバイスを読み出しまたは書き込み用に開くことはできますが、両方同時には開けません。

呼び出しの文法が普通と異なることに注意してください: 最初の引数は省略可能で、2 番目が必須です。これは `ossaudiodev` にとってかわられた古い `linuxaudiodev` との互換性のためという歴史的な産物です。

`ossaudiodev.openmixer([device])`

ミキサデバイスを開き、OSS ミキサデバイスオブジェクトを返します。 `device` は使用するミキサデバイスのファイル名です。 `device` を指定しない場合、モジュールはまず環境変数 `MIXERDEV` を参照して使用するデバイスを探します。見つからなければ、 `/dev/mixer` を参照します。

21.10.1 オーディオデバイスオブジェクト

オーディオデバイスに読み書きできるようになるには、まず 3 つのメソッドを正しい順序で呼び出さねばなりません:

1. `setfmt()` で出力形式を設定し
2. `channels()` でチャンネル数を設定し
3. `speed()` でサンプリングレートを設定します

この代わりに `setparameters()` メソッドを呼び出せば、三つのオーディオパラメタを一度で設定できます。 `setparameters()` は便利ですが、多くの状況で柔軟性に欠けるでしょう。

`open()` の返すオーディオデバイスオブジェクトには以下のメソッドおよび (読み出し専用の) 属性があります:

`oss_audio_device.close()`

オーディオデバイスを明示的に閉じます。オーディオデバイスは、読み出しや書き込みが終了したら必ず閉じねばなりません。閉じたオブジェクトを再度開くことはできません。

`oss_audio_device.fileno()`

デバイスに関連付けられているファイル記述子を返します。

`oss_audio_device.read(size)`

オーディオ入力から `size` バイトを読みだし、Python 文字列型にして返します。多くの Unix デバイスドライバと違い、ブロックデバイスモード (デフォルト) の OSS オーディオデバイスでは、要求した量のデータ全体を取り込むまで `read()` がブロックします。

`oss_audio_device.write(data)`

Python 文字列 `data` の内容をオーディオデバイスに書き込み、書き込まれたバイト数を返します。オーディオデバイスがブロックモード (デフォルト) の場合、常に文字列データ全体を書き込みます (前述のように、これは通常の Unix デバイスの振舞いとは異なります)。デバイスが非ブロックモードの場合、データの一部が書き込まれないことがあります — `writeall()` を参照してください。

`oss_audio_device.writeall(data)`

Python 文字列の *data* 全体をオーディオデバイスに書き込みます。オーディオデバイスがデータを受け取れるようになるまで待機し、書き込めるだけのデータを書き込むという操作を、*data* を全て書き込み終わるまで繰り返します。デバイスがブロックモード (デフォルト) の場合には、このメソッドは `write()` と同じです。 `writeall()` が有用なのは非ブロックモードだけです。実際に書き込まれたデータの量と渡したデータの量は必ず同じになるので、戻り値はありません。

以下の各メソッドは、厳密に 1 つの `ioctl()` システムコールに対応しています。対応関係は明らかです: 例えば、`setfmt()` は `ioctl` の `SNDCTL_DSP_SETFMT` に対応し、`sync()` は `SNDCTL_DSP_SYNC` に対応します (これは OSS のドキュメントを調べるときに便利です)。中で呼んでいる `ioctl()` が失敗した場合は、`IOError` が送出されます。

`oss_audio_device.nonblock()`

デバイスを非ブロックモードにします。いったん非ブロックモードにしたら、ブロックモードは戻せません。

`oss_audio_device.getfmts()`

サウンドカードがサポートしているオーディオ出力形式をビットマスクで返します。以下は OSS でサポートされているフォーマットの一部です:

フォーマット	説明
AFMT_MU_LAW	対数符号化 (Sun の .au 形式や /dev/audio で使われている形式)
AFMT_A_LAW	対数符号化
AFMT_IMA_ADPCM	Interactive Multimedia Association で定義されている 4:1 圧縮形式
AFMT_U8	符号なし 8 ビットオーディオ
AFMT_S16_LE	符号つき 16 ビットオーディオ、リトルエンディアンバイトオーダー (Intel プロセッサで使われている形式)
AFMT_S16_BE	符号つき 16 ビットオーディオ、ビッグエンディアンバイトオーダー (68k、PowerPC、Sparc で使われている形式)
AFMT_S8	符号つき 8 ビットオーディオ
AFMT_U16_LE	符号なし 16 ビットリトルエンディアンオーディオ
AFMT_U16_BE	符号なし 16 ビットビッグエンディアンオーディオ

オーディオ形式の完全なリストは OSS の文書をひもといてください。ただ、ほとんどのシステムは、こうした形式のサブセットしかサポートしていません。古めのデバイスの中には `AFMT_U8` だけしかサポートしていないものがあります。現在使われている最も一般的な形式は `AFMT_S16_LE` です。

`oss_audio_device.setfmt(format)`

現在のオーディオ形式を *format* に設定しようと試みます — *format* については `getfmts()` のリストを参照してください。実際にデバイスに設定されたオーディオ形式を返します。要求通りの形式でないこともあります。 `AFMT_QUERY` を渡すと現在デバイスに設定されているオーディオ形式を返します。

`oss_audio_device.channels(nchannels)`

出力チャンネル数を *nchannels* に設定します。1 はモノラル、2 はステレオです。いくつかのデバイスでは 2 つより多いチャンネルを持つものもありますし、ハイレンドなデバイスではモノラルをサポートしないものもあります。デバイスに設定されたチャンネル数を返します。

`oss_audio_device.speed(samplerate)`

サンプリングレートを 1 秒あたり *samplerate* に設定しようと試み、実際に設定されたレートを返します。たいていのサウンドデバイスでは任意のサンプリングレートをサポートしていません。一般的なレートは以下の通りです:

レート	説明
8000	/dev/audio のデフォルト
11025	会話音声の録音に使われるレート
22050	
44100	(サンプルあたり 16 ビットで 2 チャンネルの場合) CD 品質のオーディオ
96000	(サンプルあたり 24 ビットの場合) DVD 品質のオーディオ

`oss_audio_device.sync()`

サウンドデバイスがバッファ内の全てのデータを再生し終わるまで待機します。(デバイスを閉じると暗黙のうちに *sync()* が起こります) OSS のドキュメント上では、*sync()* を使うよりデバイスを一度閉じて開き直すよう勧めています。

`oss_audio_device.reset()`

再生あるいは録音を即座に中止して、デバイスをコマンドを受け取れる状態に戻します。OSS のドキュメントでは、*reset()* を呼び出した後に一度デバイスを閉じ、開き直すよう勧めています。

`oss_audio_device.post()`

ドライバに出力の一時停止 (pause) が起きそうであることを伝え、ドライバが一時停止をより賢く扱えるようにします。短いサウンドエフェクトを再生した直後やユーザ入力待ちの前、またディスク I/O 前などに使うことになるでしょう。

以下のメソッドは、複数の *ioctl()* を組み合わせたり、*ioctl()* と単純な計算を組み合わせたりした便宜用メソッドです。

`oss_audio_device.setparameters(format, nchannels, samplerate[, strict=False])`

主要なオーディオパラメタ、サンプル形式、チャンネル数、サンプルレートを一つのメソッド呼び出しで設定します。*format*、*nchannels* および *samplerate* には、それぞれ *setfmt()*、*channels()* および *speed()* と同じやり方で値を設定します。*strict* の値が真の場合、*setparameters()* は値が実際に要求通りにデバイスに設定されたかどうか調べ、違っていれば *OSSAudioError* を送出します。実際にデバイスドライバが設定したパラメタ値を表す (*format*, *nchannels*, *samplerate*) からなるタプルを返します (*setfmt()*、*channels()* および *speed()* の返す値と同じです)。

例えば、

```
(fmt, channels, rate) = dsp.setparameters(fmt, channels, rate)
```

は以下と同等です

```
fmt = dsp.setfmt(fmt)
channels = dsp.channels(channels)
rate = dsp.rate(rate)
```

`oss_audio_device.bufsize()`

ハードウェアのバッファサイズをサンプル数で返します。

`oss_audio_device.obufcount()`

ハードウェアバッファ上に残っていてまだ再生されていないサンプル数を返します。

`oss_audio_device.obuffree()`

ブロックを起こさずにハードウェアの再生キューに書き込めるサンプル数を返します。

オーディオデバイスオブジェクトは読み出し専用の属性もサポートしています:

`oss_audio_device.closed`

デバイスが閉じられたかどうかを示す真偽値です。

`oss_audio_device.name`

デバイスファイルの名前を含む文字列です。

`oss_audio_device.mode`

ファイルの I/O モードで、`"r"`、`"rw"`、`"w"` のどれかです。

21.10.2 ミキサデバイスオブジェクト

ミキサオブジェクトには、2 つのファイル類似メソッドがあります:

`oss_mixer_device.close()`

このメソッドは開いているミキサデバイスオブジェクトを閉じます。このファイルを閉じた後もミキサを使おうとすると、`IOError` が送出されます。

`oss_mixer_device.fileno()`

開かれているミキサデバイスファイルのファイルハンドルナンバを返します。

以下はオーディオミキシング固有のメソッドです:

`oss_mixer_device.controls()`

このメソッドは、利用可能なミキサコントロール (`SOUND_MIXER_PCM` や `SOUND_MIXER_SYNTH` のように、ミキシングを行えるチャンネル) を指定するビットマスクを返します。このビットマスクは利用可能な全てのミキサコントロールのサブセットです — 定数 `SOUND_MIXER_*` はモジュールレベルで定義されています。例えば、もし現在のミキサオブジェクトが PCM ミキサをサポートしているか調べるには、以下の Python コードを実行します:

```
mixer=ossaudiodev.openmixer()
if mixer.controls() & (1 << ossaudiodev.SOUND_MIXER_PCM):
    # PCM is supported
    ... code ...
```

ほとんどの用途には、`SOUND_MIXER_VOLUME` (マスタボリューム) と `SOUND_MIXER_PCM` コントロールがあれば十分でしょう — とはいえ、ミキサを使うコードを書くときには、コントロールを選ぶ時に柔軟性を持たせるべきです。例えば Gravis Ultrasound には `SOUND_MIXER_VOLUME` がありません。

```
oss_mixer_device.stereocontrols()
```

ステレオミキサコントロールを示すビットマスクを返します。ビットが立っているコントロールはステレオであることを示し、立っていないコントロールはモノラルか、ミキサがサポートしていないコントロールである (どちらの理由かは `controls()` と組み合わせて使うことで判別できます) ことを示します。

ビットマスクから情報を得る例は関数 `controls()` のコード例を参照してください。

```
oss_mixer_device.recontrols()
```

録音に使用できるミキサコントロールを特定するビットマスクを返します。ビットマスクから情報を得る例は関数 `controls()` のコード例を参照してください。

```
oss_mixer_device.get(control)
```

指定したミキサコントロールのボリュームを返します。2 要素のタプル (`left_volume`, `right_volume`) を返します。ボリュームの値は 0 (無音) から 100 (最大) で示されます。コントロールがモノラルでも 2 要素のタプルが返されますが、2 つの要素の値は同じになります。

不正なコントロールを指定した場合は `OSSAudioError` を、サポートされていないコントロールを指定した場合は `IOError` を送出します。

```
oss_mixer_device.set(control, (left, right))
```

指定したミキサコントロールのボリュームを (`left`, `right`) に設定します。`left` と `right` は整数で、0 (無音) から 100 (最大) の間で指定せねばなりません。呼び出しに成功すると新しいボリューム値を 2 要素のタプルで返します。サウンドカードによっては、ミキサの分解能上の制限から、指定したボリュームと厳密に同じにはならない場合があります。

不正なコントロールを指定した場合や、指定したボリューム値が範囲外であった場合、`OSSAudioError` を送出します。

```
oss_mixer_device.get_recsrc()
```

現在録音のソースに使われているコントロールを示すビットマスクを返します。

```
oss_mixer_device.set_recsrc(bitmask)
```

録音のソースを指定するために、この関数を呼び出します。成功した場合は、新しい録音のソース (1 つもしくは複数) を示すビットマスクを返します; 不正なソースを指定した場合は、`IOError` を送出します。現在の録音のソースをマイク入力に設定するには以下のように呼び出します:

```
mixer.setrecsrc (1 << ossaudiodev.SOUND_MIXER_MIC)
```


第 22 章

国際化

この章で解説されるモジュールは、プログラムのメッセージで使用される言語を選択したり、または出力を地域の習慣に従って変更するメカニズムを提供して、言語や地域に依存しないソフトの開発を手助けします。

この章で解説されるモジュールのリスト:

22.1 gettext — 多言語対応に関する国際化サービス

ソースコード: [Lib/gettext.py](#)

`gettext` モジュールは、Python によるモジュールやアプリケーションの国際化 (I18N, I-nternationalizatio-N) および地域化 (L10N, L-ocalizatio-N) サービスを提供します。このモジュールは GNU `gettext` メッセージカタログへの API と、より高レベルで Python ファイルに適しているクラスに基づいた API の両方をサポートしています。以下で述べるインタフェースを使うことで、モジュールやアプリケーションのメッセージをある自然言語で記述しておき、翻訳されたメッセージのカタログを与えて他の異なる自然言語の環境下で動作させることができます。

ここでは Python のモジュールやアプリケーションを地域化するためのいくつかのヒントも提供しています。

22.1.1 GNU `gettext` API

`gettext` モジュールでは、以下の GNU `gettext` API に非常に良く似た API を提供しています。この API を使う場合、メッセージ翻訳の影響はアプリケーション全体に及ぼすことになります。アプリケーションが単一の言語しか扱わず、各言語に依存する部分をユーザのロケール情報によって選ぶのなら、ほとんどの場合この方法でやりたいことを実現できます。Python モジュールを地域化していたり、アプリケーションの実行中に言語を切り替えたい場合、おそらくクラスに基づいた API を使いたくなるでしょう。

```
gettext.bindtextdomain(domain[, localedir])
```

`domain` をロケール辞書 `localedir` に結び付け (bind) ます。具体的には、`gettext` は与えられたドメインに対するバイナリ形式の `.mo` ファイルを、(Unix では) `localedir/language/LC_MESSAGES/domain.mo` から探します。ここで `languages` はそれぞれ環境変数 `LANGUAGE`、`LC_ALL`、`LC_MESSAGES`、および `LANG` の中から検索されます。

`localedir` が省略されるか `None` の場合、現在 `domain` に結び付けられている内容が返されます。^{*1}

`gettext.bind_textdomain_codeset (domain[, codeset])`

`domain` を `codeset` に結び付けて、`gettext()` ファミリの関数が返す文字列のエンコード方式を変更します。`codeset` を省略すると、現在結び付けられているコードセットを返します。

バージョン 2.4 で追加.

`gettext.textdomain ([domain])`

現在のグローバルドメインを調べたり変更したりします。`domain` が `None` の場合、現在のグローバルドメインが返されます。それ以外の場合にはグローバルドメインは `domain` に設定され、設定されたグローバルドメインを返します。

`gettext.gettext (message)`

現在のグローバルドメイン、言語、およびロケール辞書に基づいて、`message` の特定地域向けの翻訳を返します。通常、ローカルな名前空間ではこの関数に `_()` という別名をつけます (下の例を参照してください)。

`gettext.lgettext (message)`

`gettext()` と同じですが、`bind_textdomain_codeset()` で特にエンコードを指定しない限り、翻訳結果を優先システムエンコーディング (preferred system encoding) で返します。

バージョン 2.4 で追加.

`gettext.dgettext (domain, message)`

`gettext()` と同様ですが、指定された `domain` からメッセージを探します。

`gettext.ldgettext (domain, message)`

`dgettext()` と同じですが、`bind_textdomain_codeset()` で特にエンコードを指定しない限り、翻訳結果を優先システムエンコーディング (preferred system encoding) で返します。

バージョン 2.4 で追加.

`gettext.ngettext (singular, plural, n)`

`gettext()` と同様ですが、複数形の場合を考慮しています。翻訳文字列が見つかった場合、`n` の様式を適用し、その結果得られたメッセージを返します (言語によっては二つ以上の複数形があります)。翻訳文字列が見つからなかった場合、`n` が 1 なら `singular` を返します; そうでない場合 `plural` を返します。

複数形の様式はカタログのヘッダから取り出されます。様式は C または Python の式で、自由な変数 `n` を持ちます; 式の評価値はカタログ中の複数形のインデクスとなります。 `.po` ファイルで用いられる詳細な文法と、様々な言語における様式については、GNU gettext ドキュメントを参照してください。

バージョン 2.3 で追加.

^{*1} 標準でロケールが収められているディレクトリはシステム依存です; 例えば、RedHat Linux では `/usr/share/locale` ですが、Solaris では `/usr/lib/locale` です。 `gettext` モジュールはこうしたシステム依存の標準設定をサポートしません; その代わりに `sys.prefix/share/locale` を標準の設定とします。この理由から、常にアプリケーションの開始時に絶対パスで明示的に指定して `bind_textdomain()` を呼び出すのが最良のやり方ということになります。

`gettext.lgettext(singular, plural, n)`

`nggettext()` と同じですが、`bind_textdomain_codeset()` で特にエンコードを指定しない限り、翻訳結果を優先システムエンコーディング (preferred system encoding) で返します。

バージョン 2.4 で追加。

`gettext.dgettext(domain, singular, plural, n)`

`nggettext()` と同様ですが、指定された `domain` からメッセージを探します。

バージョン 2.3 で追加。

`gettext.ldgettext(domain, singular, plural, n)`

`dngettext()` と同じですが、`bind_textdomain_codeset()` で特にエンコードを指定しない限り、翻訳結果を優先システムエンコーディング (preferred system encoding) で返します。

バージョン 2.4 で追加。

GNU **gettext** では `dcgettext()` も定義していますが、このメソッドはあまり有用ではないと思われるので、現在のところ実装されていません。

以下にこの API の典型的な使用法を示します:

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print _('This is a translatable string.')
```

22.1.2 クラスに基づいた API

The class-based API of the `gettext` module gives you more flexibility and greater convenience than the GNU **gettext** API. It is the recommended way of localizing your Python applications and modules. `gettext` defines a "translations" class which implements the parsing of GNU .mo format files, and has methods for returning either standard 8-bit strings or Unicode strings. Instances of this "translations" class can also install themselves in the built-in namespace as the function `_()`.

`gettext.find(domain[, loaledir[, languages[, all]]])`

この関数は標準的な .mo ファイル検索アルゴリズムを実装しています。 `textdomain()` と同じく、`domain` を引数にとります。オプションの `loaledir` は `bindtextdomain()` と同じです。またオプションの `languages` は文字列を列挙したリストで、各文字列は言語コードを表します。

`loaledir` が与えられていない場合、標準のシステムロケールディレクトリが使われます。^{*2} `languages` が与えられなかった場合、以下の環境変数: `LANGUAGE`、`LC_ALL`、`LC_MESSAGES`、および `LANG` が検索されます。空でない値を返した最初の候補が `languages` 変数として使われます。この環境変数は言語名をコロンで分かち書きしたリストを含んでいなければなりません。 `find()` はこの文字列をコロンで分割し、言語コードの候補リストを生成します。

^{*2} 上の `bindtextdomain()` に関する脚注を参照してください。

`find()` は次に言語コードを展開および正規化し、リストの各要素について、以下のパス構成:

```
localedir/language/LC_MESSAGES/domain.mo
```

からなる実在するファイルの探索を反復的に行います。 `find()` は上記のような実在するファイルで最初に見つかったものを返します。該当するファイルが見つからなかった場合、 `None` が返されます。 `all` が与えられていれば、全ファイル名のリストが言語リストまたは環境変数で指定されている順番に並べられたものを返します。

```
gettext.translation(domain[, localedir[, languages[, class[, fallback[, codeset]]]])
```

`Translations` インスタンスを `domain`、`localedir`、および `languages` に基づいて生成して返します。 `domain`、`localedir`、および `languages` はまず関連付けられている `.mo` ファイルパスのリストを取得するために `find()` に渡されます。同じ `.mo` ファイル名を持つインスタンスはキャッシュされます。実際にインスタンス化されるクラスは `class_` が与えられていればそのクラスが、そうでない時には `GNUTranslations` です。クラスのコンストラクタは単一の引数としてファイルオブジェクトを取らなくてはなりません。 `codeset` を指定した場合、翻訳文字列のエンコードに使う文字セットを変更します。

複数のファイルが発見された場合、後で見つかったファイルは前に見つかったファイルの代替で見なされ、後で見つかった方が利用されます。代替の設定を可能にするには、 `copy.copy()` を使ってキャッシュから翻訳オブジェクトを複製します; こうすることで、実際のインスタンスデータはキャッシュのものと共有されます。

`.mo` ファイルが見つからなかった場合、 `fallback` が偽 (標準の設定です) ならこの関数は `IOError` を送出し、 `fallback` が真なら `NullTranslations` インスタンスが返されます。

バージョン 2.4 で変更: `codeset` パラメータを追加しました。

```
gettext.install(domain[, localedir[, unicode[, codeset[, names]]])
```

`translation()` に `domain`、`localedir`、および `codeset` を渡してできる関数 `_()` を Python の組み込み名前空間に組み込みます。 `unicode` フラグは `translation()` の返す翻訳オブジェクトの `install()` メソッドに渡されます。

`names` パラメータについては、翻訳オブジェクトの `install()` メソッドの説明を参照ください。

以下に示すように、通常はアプリケーション中の文字列を関数 `_()` の呼び出して包み込んで翻訳対象候補であることを示します:

```
print _('This string will be translated.')
```

利便性を高めるためには、 `_()` 関数を Python の組み込み名前空間に組み入れる必要があります。こうすることで、アプリケーション内の全てのモジュールからアクセスできるようになります。

バージョン 2.4 で変更: `codeset` パラメータを追加しました。

バージョン 2.5 で変更: `names` パラメータを追加しました。

NullTranslations クラス

翻訳クラスは、元のソースファイル中のメッセージ文字列から翻訳されたメッセージ文字列への変換処理が実際に実装されているクラスです。全ての翻訳クラスで基底クラスとして使われているクラスが `NullTranslations` です; このクラスは、独自の翻訳クラスを実装するのに使える基本的なインタフェースを提供しています。以下に `NullTranslations` のメソッドを示します:

class `gettext.NullTranslations` (`[fp]`)

オプションのファイルオブジェクト `fp` を取ります。この引数は基底クラスでは無視されます。このメソッドは "保護された (protected)" インスタンス変数 `_info` および `_charset` を初期化します。これらの変数の値は派生クラスで設定することができます。同様に `_fallback` も初期化しますが、この値は `add_fallback()` で設定されます。その後、`fp` が `None` でない場合 `self._parse(fp)` を呼び出します。

_parse (`fp`)

基底クラスでは何もしない (no-op) になっています。このメソッドの役割はファイルオブジェクト `fp` を引数に取り、ファイルからデータを読み出し、メッセージカタログを初期化することです。サポートされていないメッセージカタログ形式を使っている場合、その形式を解釈するためにはこのメソッドを上書きしなくてはなりません。

add_fallback (`fallback`)

`fallback` を現在の翻訳オブジェクトの代替オブジェクトとして追加します。翻訳オブジェクトが与えられたメッセージに対して翻訳メッセージを提供できない場合、この代替オブジェクトに問い合わせることになります。

gettext (`message`)

If a fallback has been set, forward `gettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

lgettext (`message`)

If a fallback has been set, forward `lgettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

バージョン 2.4 で追加.

ugettext (`message`)

If a fallback has been set, forward `ugettext()` to the fallback. Otherwise, return the translated message as a Unicode string. Overridden in derived classes.

ngettext (`singular, plural, n`)

If a fallback has been set, forward `ngettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

バージョン 2.3 で追加.

lngettext (`singular, plural, n`)

If a fallback has been set, forward `lngettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

バージョン 2.4 で追加.

ungettext (*singular, plural, n*)

If a fallback has been set, forward `ungettext()` to the fallback. Otherwise, return the translated message as a Unicode string. Overridden in derived classes.

バージョン 2.3 で追加.

info ()

”protected” の `_info` 変数を返します。

charset ()

”protected” の `_charset` 変数を返します。

output_charset ()

翻訳メッセージとして返す文字列のエンコードを決める、”protected” の `_output_charset` 変数を返します。

バージョン 2.4 で追加.

set_output_charset (*charset*)

翻訳メッセージとして返す文字列のエンコードを決める、”protected” の変数 `_output_charset` を変更します。

バージョン 2.4 で追加.

install ([*unicode*, *names*])

unicode フラグが偽の場合、このメソッドは `self.gettext()` を組み込み名前空間に組み入れ、`_` と結び付けます。 *unicode* が真の場合、 `self.gettext()` の代わりに `self.ugettext()` を結び付けます。標準では *unicode* は偽です。

names パラメータには、 `_()` 以外に組み込みの名前空間にインストールしたい関数名のシーケンスを指定します。サポートしている名前は 'gettext' (*unicode* フラグの設定に応じて `self.gettext()` あるいは `self.ugettext()` のいずれかに対応します)、 'ngettext' (*unicode* フラグの設定に応じて `self.ngettext()` あるいは `self.ungettext()` のいずれかに対応します)、 'lgettext' および 'lngettext' です。

この方法はアプリケーションで `_()` 関数を利用できるようにするための最も便利な方法ですが、唯一の手段でもあるので注意してください。この関数はアプリケーション全体、とりわけ組み込み名前空間に影響するので、地域化されたモジュールで `_()` を組み入れることができないのです。その代わりに、以下のコードを使って `_()` を使えるようにしなければなりません。:

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

この操作は `_()` をモジュール内だけのグローバル名前空間に組み入れるので、モジュール内の `_()` の呼び出しだけに影響します。

バージョン 2.5 で変更: *names* パラメータを追加しました.

GNUTranslations クラス

`gettext` モジュールでは `NullTranslations` から派生したもう一つのクラス: `GNUTranslations` を提供しています。このクラスはビッグエンディアン、およびリトルエンディアン両方のバイナリ形式の GNU `gettext` .mo ファイルを読み出せるように `_parse()` を上書きしています。また、このクラスはメッセージ id とメッセージ文字列の両方を Unicode に型強制します。

このクラスではまた、翻訳カタログ以外に、オプションのメタデータを読み込んで解釈します。GNU `gettext` では、空の文字列に対する変換先としてメタデータを取り込むことが慣習になっています。このメタデータは **RFC 822** 形式の `key: value` のペアになっており、`Project-Id-Version` キーを含んでいなければなりません。キー `Content-Type` があった場合、`charset` の特性値 (property) は "保護された" `_charset` インスタンス変数を初期化するために用いられます。値がない場合には、デフォルトとして `None` が使われます。エンコードに用いられる文字セットが指定されている場合、カタログから読み出された全てのメッセージ id とメッセージ文字列は、指定されたエンコードを用いて Unicode に変換されます。`ugettext()` は常に Unicode を返し、`gettext()` はエンコードされた 8 ビット文字列を返します。どちらのメソッドにおける引数 id の場合も、Unicode 文字列か US-ASCII 文字のみを含む 8 ビット文字列だけが受理可能です。国際化された Python プログラムでは、メソッドの Unicode 版 (すなわち `ugettext()` や `ungettext()`) の利用が推奨されています。

key/value ペアの集合全体は辞書型データ中に配置され、"保護された" `_info` インスタンス変数に設定されます。

.mo ファイルのマジックナンバーが不正な場合、あるいはその他の問題がファイルの読み出し中に発生した場合、`GNUTranslations` クラスのインスタンス化で `IOError` が送出されることがあります。

以下のメソッドは基底クラスの実装からオーバーライドされています:

`GNUTranslations.gettext(message)`

カタログから `message` id を検索して、対応するメッセージ文字列を、カタログの文字セットが既知のエンコードの場合、エンコードされた 8 ビット文字列として返します。`message` id に対するエントリがカタログに存在せず、フォールバックが設定されている場合、フォールバック検索はオブジェクトの `gettext()` メソッドに転送されます。そうでない場合、`message` id 自体が返されます。

`GNUTranslations.lgettext(message)`

`gettext()` と同じですが、翻訳結果は `set_output_charset()` で特にエンコーディングが指定されていなければ、優先システムエンコーディングで返します。

バージョン 2.4 で追加.

`GNUTranslations.ugettext(message)`

カタログから `message` id を検索して、対応するメッセージ文字列を、Unicode でエンコードして返します。`message` id に対するエントリがカタログに存在せず、フォールバックが設定されている場合、フォールバック検索はオブジェクトの `ugettext()` メソッドに転送されます。そうでない場合、`message` id 自体が返されます。

`GNUTranslations.ngettext(singular, plural, n)`

メッセージ id に対する複数形を検索します。カタログに対する検索では `singular` がメッセージ id として用いられ、`n` にはどの複数形を用いるかを指定します。返されるメッセージ文字列は 8 ビットの文字列で、カタログの文字セットが既知の場合にはその文字列セットでエンコードされています。

メッセージ id がカタログ中に見つからず、フォールバックオブジェクトが指定されている場合、メッセージ検索要求はフォールバックオブジェクトの `gettext()` メソッドに転送されます。そうでない場合、`n` が 1 ならば *singular* が返され、それ以外に対しては *plural* が返されます。

バージョン 2.3 で追加。

`GNUTranslations.lgettext(singular, plural, n)`

`gettext()` と同じですが、翻訳結果は `set_output_charset()` で特にエンコーディングが指定されていないければ、優先システムエンコーディングで返します。

バージョン 2.4 で追加。

`GNUTranslations.ungettext(singular, plural, n)`

メッセージ id に対する複数形を検索します。カタログに対する検索では *singular* がメッセージ id として用いられ、`n` にはどの複数形を用いるかを指定します。返されるメッセージ文字列は Unicode 文字列です。

メッセージ id がカタログ中に見つからず、フォールバックオブジェクトが指定されている場合、メッセージ検索要求はフォールバックオブジェクトの `ungettext()` メソッドに転送されます。そうでない場合、`n` が 1 ならば *singular* が返され、それ以外に対しては *plural* が返されます。

以下に例を示します。:

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ungettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

バージョン 2.3 で追加。

Solaris メッセージカタログ機構のサポート

Solaris オペレーティングシステムでは、独自の `.mo` バイナリファイル形式を定義していますが、この形式に関するドキュメントが手に入らないため、現時点ではサポートされていません。

Catalog コンストラクタ

GNOME では、James Henstridge によるあるバージョンの `gettext` モジュールを使っていますが、このバージョンは少し異なった API を持っています。ドキュメントに書かれている利用法は:

```
import gettext
cat = gettext.Catalog(domain, localedir)
_ = cat.gettext
print _('hello world')
```

となっています。過去のモジュールとの互換性のために、`Catalog()` は前述の `translation()` 関数の別名になっています。

このモジュールと Henstridge のバージョンとの間には一つ相違点があります: 彼のカタログオブジェクトはマップ型の API を介したアクセスがサポートされていましたが、この API は使われていないらしく、現在はサポートされていません。

22.1.3 プログラムやモジュールを国際化する

国際化 (I18N, I-nternationalizatio-N) とは、プログラムを複数の言語に対応させる操作を指します。地域化 (L10N, L-ocalizatio-N) とは、すでに国際化されているプログラムを特定地域の言語や文化的な事情に対応させることを指します。Python プログラムに多言語メッセージ機能を追加するには、以下の手順を踏む必要があります:

1. プログラムやモジュールで翻訳対象とする文字列に特殊なマークをつけて準備します
2. マークづけをしたファイルに一連のツールを走らせ、生のメッセージカタログを生成します
3. 特定の言語へのメッセージカタログの翻訳を作成します
4. メッセージ文字列を適切に変換するために `gettext` モジュールを使います

ソースコードを I18N 化する準備として、ファイル内の全ての文字列を探す必要があります。翻訳を行う必要のある文字列はどれも `_('...')` — すなわち関数 `_()` の呼び出しで包むことでマーク付けしなくてはなりません。例えば以下のようにします:

```
filename = 'mylog.txt'
message = _('writing a log message')
fp = open(filename, 'w')
fp.write(message)
fp.close()
```

この例では、文字列 `'writing a log message'` が翻訳対象候補としてマーク付けされており、文字列 `'mylog.txt'` および `'w'` はされていません。

Python の配布物には、ソースコードに準備作業を行った後でメッセージカタログの生成を助ける 2 つのツールが付属します。これらはバイナリ配布の場合には付属していたりしなかったりしますが、ソースコード配布には入っており、`Tools/i18n` ディレクトリにあります。

pygettext プログラム^{*3} は全ての Python ソースコードを走査し、予め翻訳対象としてマークした文字列を探し出します。このツールは GNU **gettext** プログラムと同様ですが、Python ソースコードの機微について熟知している反面、C 言語や C++ 言語のソースコードについては全く知りません。(C 言語による拡張モジュールのように) C 言語のコードも翻訳対象にしたいのでない限り、GNU **gettext** は必要ありません。

pygettext は、テキスト形式 Uniform スタイルによる人間が判読可能なメッセージカタログ `.pot` ファイル群を生成します。このファイル群はソースコード中でマークされた全ての文字列と、それに対応する翻訳文字列のためのプレースホルダを含むファイルで構成されています。**pygettext** はコマンドライン形式のスクリプトで、**xgettext** と同様のコマンドラインインタフェースをサポートします; 使用法についての詳細を見るには以下を起動してください。:

^{*3} 同様の作業を行う **xpot** と呼ばれるプログラムを Francois Pinard が書いています。このプログラムは彼の `po-utils` パッケージの一部です。


```
pygettext.py --help
```

これら `.pot` ファイルのコピーは次に、サポート対象の各自然言語について、言語ごとのバージョンを作成する個々の人間の翻訳者に頒布されます。翻訳者たちはプレースホルダ部分を埋めて言語ごとのバージョンをつくり、`.po` ファイルとして返します。(Tools/i18n ディレクトリ内の) `msgfmt.py`^{*4} プログラムを使い、翻訳者から返された `.po` ファイルから機械可読な `.mo` バイナリカタログファイルを生成します。`.mo` ファイルは、`gettext` モジュールが実行時に実際の翻訳処理を行うために使われます。

`gettext` モジュールをソースコード中でどのように使うかは単一のモジュールを国際化するのか、それともアプリケーション全体を国際化するのにかによります。次のふたつのセクションで、それぞれについて説明します。

モジュールを地域化する

モジュールを地域化する場合、グローバルな変更、例えば組み込み名前空間への変更を行わないように注意しなければなりません。GNU `gettext` API ではなく、クラススペースの API を使うべきです。

仮に対象のモジュール名を `"spam"` とし、モジュールの各言語における翻訳が収められた `.mo` ファイルが `/usr/share/locale` に GNU `gettext` 形式で置かれているとします。この場合、モジュールの最初で以下のようにします:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.lgettext
```

翻訳オブジェクトが `.po` ファイル中の Unicode 文字列を返すようになっているのなら、上の代わりに以下のようにします:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.ugettext
```

アプリケーションを地域化する

アプリケーションを地域化するのなら、関数 `_()` をグローバルな組み込み名前空間に組み入れなければならず、これは通常アプリケーションの主ドライバ (main driver) ファイルで行います。この操作によって、アプリケーション独自のファイルは明示的に各ファイルで `_()` の組み入れを行わなくても単に `_('...')` を使うだけで済むようになります。

単純な場合では、単に以下の短いコードをアプリケーションの主ドライバファイルに追加するだけです:

```
import gettext
gettext.install('myapplication')
```

^{*4} `msgfmt.py` は GNU `msgfmt` とバイナリ互換ですが、より単純で、Python だけを使った実装がされています。このプログラムと `pygettext.py` があれば、通常 Python プログラムを国際化するために GNU `gettext` パッケージをインストールする必要はありません。

ロケールディレクトリや *unicode* フラグを設定する必要がある場合、それらの値を *install()* 関数に渡すことができます:

```
import gettext
gettext.install('myapplication', '/usr/share/locale', unicode=1)
```

動作中 (on the fly) に言語を切り替える

多くの言語を同時にサポートする必要がある場合、複数の翻訳インスタンスを生成して、例えば以下のコードのように、インスタンスを明示的に切り替えてもかまいません。:

```
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

翻訳処理の遅延解決

コードを書く上では、ほとんどの状況で文字列はコードされた場所で翻訳されます。しかし場合によっては、翻訳対象として文字列をマークはするが、その後実際に翻訳が行われるように遅延させる必要が生じます。古典的な例は以下のようなコードです:

```
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python', ]

# ...
for a in animals:
    print a
```

ここで、リスト *animals* 内の文字列は翻訳対象としてマークはしたいが、文字列が出力されるまで実際に翻訳を行うのは避けたいとします。

こうした状況进行处理する一つの方法を以下に示します:

```
def _(message): return message

animals = [_('mollusk'),
```

(次のページに続く)

(前のページからの続き)

```

    _('albatross'),
    _('rat'),
    _('penguin'),
    _('python'), ]

del _

# ...
for a in animals:
    print _(a)

```

ダミーの `_()` 定義が単に文字列をそのまま返すようになっているので、上のコードはうまく動作します。かつ、このダミーの定義は、組み込み名前空間に置かれた `_()` の定義で (`del` 命令を実行するまで) 一時的に上書きすることができます。もしそれまでに `_()` をローカルな名前空間に持っていたら注意してください。

二つ目の例における `_()` の使い方では、“a” は文字列リテラルではないので、**pygettext** プログラムが翻訳可能な対象として識別しません。

もう一つの処理法は、以下の例のようなやり方です:

```

def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print _(a)

```

この例の場合では、翻訳可能な文字列を関数 `N_()` でマーク付けしており^{*5}、`_()` の定義とは全く衝突しません。しかしメッセージ展開プログラムには翻訳対象の文字列が `N_()` でマークされていることを教える必要が出てくるでしょう。**pygettext** および **xpot** は両方とも、コマンドライン上のスイッチでこの機能をサポートしています。

gettext () VS. lgettext ()

Python 2.4 からは、`lgettext()` ファミリが導入されました。この関数の目的は、現行の GNU gettext 実装によりよく準拠した別の関数を提供することにあります。翻訳メッセージファイル中で使われているのと同じコードセットを使って文字列をエンコードして返す `gettext()` と違い、これらの関数は `locale.getpreferredencoding()` の返す優先システムエンコーディングを使って翻訳メッセージ文字列をエンコードして返します。また、Python 2.4 では、翻訳メッセージ文字列で使われているコードセットを明示的に選べるようにする関数が新たに導入されていることにも注意してください。コードセットを明示的に設定すると、`lgettext()` でさえ、指定したコードセットで翻訳メッセージ文字列を返します。これは GNU gettext

^{*5} この `N_()` をどうするかは全くの自由です; `MarkThisStringForTranslation()` などとしてもかまいません。

実装が期待している仕様と同じです。

22.1.4 謝辞

以下の人々が、このモジュールのコード、フィードバック、設計に関する助言、過去の実装、そして有益な経験談による貢献をしてくれました:

- Peter Funk
- James Henstridge
- Juan David Ibez Palomar
- Marc-Andr Lemburg
- Martin von Lwis
- Franois Pinard
- Barry Warsaw
- Gustavo Niemeyer

脚注

22.2 locale — 国際化サービス

`locale` モジュールは POSIX ロケールデータベースおよびロケール関連機能へのアクセスを提供します。POSIX ロケール機構を使うことで、プログラマはソフトウェアが実行される各国における詳細を知らなくても、アプリケーション上で特定の地域文化に関係する部分を扱うことができます。

`locale` モジュールは、`_locale` を被うように実装されており、ANSI C ロケール実装を使っている `_locale` が利用可能なら、こちらを先に使うようになっています。

`locale` モジュールでは以下の例外と関数を定義しています:

exception `locale.Error`

`setlocale()` に渡されたロケールが認識されない場合例外が送出されます。

`locale.setlocale(category[, locale])`

`locale` が渡され `None` でない場合、`setlocale()` は `category` のロケール設定を変更します。利用可能なカテゴリーは下記の表を参照してください。 `locale` は文字列か 2 つの文字列の iterable (言語コードと文字コード) です。iterable の場合 locale aliasing engine を用いてロケール名に変換されます。空の文字列はユーザのデフォルトの設定を指定します。ロケールの変更に失敗した場合 `Error` が送出されます。成功した場合新しいロケールの設定が返されます。

`locale` が省略されたり `None` の場合、`category` の現在の設定が返されます。

`setlocale()` はほとんどのシステムでスレッド安全ではありません。アプリケーションを書くとき、大抵は以下のコード

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

から書き始めます。これは全てのカテゴリをユーザの環境における標準設定 (大抵は環境変数 `LANG` で指定されています) に設定します。その後複数スレッドを使ってロケールを変更したりしない限り、問題は起こらないはずです。

バージョン 2.0 で変更: 引数 *locale* の値にイテラブルを渡せるようになりました。

`locale.localeconv()`

地域的な慣行のデータベースを辞書として返します。辞書は以下の文字列をキーとして持っています:

カテゴリ	キー	意味
<i>LC_NUMERIC</i>	'decimal_point'	小数点を表す文字です。
	'grouping'	'thousands_sep' が来るかもしれない場所を相対的に表した数からなる配列です。配列が <i>CHAR_MAX</i> で終端されている場合、それ以上の桁では桁数字のグループ化を行いません。配列が 0 で終端されている場合、最後に指定したグループが反復的に使われます。
	'thousands_sep'	桁グループ間を区切るために使われる文字です。
<i>LC_MONETARY</i>	'int_curr_symbol'	国際通貨を表現する記号です。
	'currency_symbol'	地域的な通貨を表現する記号です。
	'p_cs_precedes/n_cs_precedes'	通貨記号が値の前につくかどうかです (それぞれ正の値、負の値を表します)。
	'p_sep_by_space/n_sep_by_space'	通貨記号と値との間にスペースを入れるかどうかです (それぞれ正の値、負の値を表します)。
	'mon_decimal_point'	金額表示の際に使われる小数点です。
	'frac_digits'	金額を地域的な方法で表現する際の小数点以下の桁数です。
	'int_frac_digits'	金額を国際的な方法で表現する際の小数点以下の桁数です。
	'mon_thousands_sep'	金額表示の際に桁区切り記号です。
	'mon_grouping'	'grouping' と同じで、金額表示の際に使われます。
	'positive_sign'	正の値の金額表示に使われる記号です。
	'negative_sign'	負の値の金額表示に使われる記号です。
	'p_sign_posn/n_sign_posn'	符号の位置です (それぞれ正の値と負の値を表します)。以下を参照ください。

数値形式の値に *CHAR_MAX* を設定すると、そのロケールでは値が指定されていないことを表します。

'p_sign_posn' および 'n_sign_posn' の取り得る値は以下の通りです。

値	説明
0	通貨記号および値は丸括弧で囲われます。
1	符号は値と通貨記号より前に来ます。
2	符号は値と通貨記号の後に続きます。
3	符号は値の直前に来ます。
4	符号は値の直後に来ます。
CHAR_MAX	このロケールでは特に指定しません。

`locale.nl_langinfo(option)`

ロケール特有の情報を文字列として返します。この関数は全てのシステムで利用可能なわけではなく、指定できる *option* もプラットフォーム間で大きく異なります。引数として使えるのは、`locale` モジュールで利用可能なシンボル定数を表す数字です。

関数 `nl_langinfo()` は以下のキーのうち一つを受理します。ほとんどの記述は GNU C ライブラリ中の対応する説明から引用されています。

`locale.CODESET`

選択されたロケールで用いられている文字エンコーディングの名前を文字列で取得します。

`locale.D_T_FMT`

日付と時刻をロケール特有の方法で表現するために、`time.strftime()` の書式化文字列として用いることのできる文字列を取得します。

`locale.D_FMT`

日付をロケール特有の方法で表現するために、`time.strftime()` の書式化文字列として用いることのできる文字列を取得します。

`locale.T_FMT`

時刻をロケール特有の方法で表現するために、`time.strftime()` の書式化文字列として用いることのできる文字列を取得します。

`locale.T_FMT_AMPM`

時刻を午前 / 午後の書式で表現するために、`time.strftime()` の書式化文字列として用いることのできる文字列を取得します。

`DAY_1 ... DAY_7`

1 週間中の *n* 番目の曜日名を取得します。

注釈: ロケール US における、`DAY_1` を日曜日とする慣行に従っています。国際的な (ISO 8601) 月曜日を週の初めとする慣行ではありません。

`ABDAY_1 ... ABDAY_7`

1 週間中の *n* 番目の曜日名を略式表記で取得します。

`MON_1 ... MON_12`

n 番目の月の名前を取得します。

ABMON.1 ... ABMON.12

n 番目の月の名前を略式表記で取得します。

locale.RADIXCHAR

基数点 (小数点ドット、あるいは小数点コンマ、等) を取得します。

locale.THOUSEP

1000 単位桁区切り (3 桁ごとのグループ化) の区切り文字を取得します。

locale.YESEXPR

肯定 / 否定で答える質問に対する肯定回答を正規表現関数で認識するために利用できる正規表現を取得します。

注釈: 表現は C ライブラリの `regex()` 関数に合ったものでなければならず、これは *re* で使われている構文とは異なるかもしれません。

locale.NOEXPR

肯定 / 否定で答える質問に対する否定回答を正規表現関数で認識するために利用できる正規表現を取得します。

locale.CRNCYSTR

通貨シンボルを取得します。シンボルを値の前に表示させる場合には “-”、値の後ろに表示させる場合には “+”、シンボルを基数点と置き換える場合には “.” を前につけます。

locale.ERA

現在のロケールで使われている年代を表現する値を取得します。

ほとんどのロケールではこの値を定義していません。この値を設定しているロケールの例は Japanese です。日本には日付の伝統的な表示法として、時の天皇に対応する元号名があります。

通常この値を直接指定する必要はありません。E を書式化文字列に指定することで、関数 `time.strftime()` がこの情報を使うようになります。返される文字列の様式は決められていないので、異なるシステム間で様式に関する同じ知識が使えると期待してはいけません。

locale.ERA_D_T_FMT

日付および時間をロケール固有の年代に基づいた方法で表現するために、`time.strftime()` の書式化文字列として用いることのできる文字列を取得します。

locale.ERA_D_FMT

日付をロケール固有の年代に基づいた方法で表現するために、`time.strftime()` の書式化文字列として用いることのできる文字列を取得します。

locale.ERA_T_FMT

時刻をロケール固有の年代に基づいた方法で表現するために、`time.strftime()` の書式化文字列として用いることのできる文字列を取得します。

locale.ALT_DIGITS

返される値は 0 から 99 までの 100 個の値の表現です。

`locale.getdefaultlocale([envvars])`

標準のロケール設定を取得しようと試み、結果をタプル (language code, encoding) の形式で返します。

POSIX によると、`setlocale(LC_ALL, '')` を呼ばなかったプログラムは、移植可能な 'C' ロケール設定を使います。`setlocale(LC_ALL, '')` を呼ぶことで、LANG 変数で定義された標準のロケール設定を使うようになります。Python では現在のロケール設定に干渉したくないので、上で述べたような方法でその挙動をエミュレーションしています。

他のプラットフォームとの互換性を維持するために、環境変数 LANG だけでなく、引数 *envvars* で指定された環境変数のリストも調べられます。最初に見つかった定義が使われます。*envvars* は標準では GNU gettext で使われている検索順になります; これは常に変数名 LANG を探します。GNU gettext では 'LANGUAGE'、'LC_ALL'、'LC_CTYPE'、および 'LANG' の順に調べられます。

'C' の場合を除き、言語コードは [RFC 1766](#) に対応します。*language code* および *encoding* が決定できなかった場合、None になるかもしれません。

バージョン 2.0 で追加。

`locale.getlocale([category])`

与えられたロケールカテゴリに対する現在の設定を、*language code*、*encoding* を含むシーケンスで返します。*category* として `LC_ALL` 以外の `LC_*` の値の一つを指定できます。標準の設定は `LC_CTYPE` です。

'C' の場合を除き、言語コードは [RFC 1766](#) に対応します。*language code* および *encoding* が決定できなかった場合、None になるかもしれません。

バージョン 2.0 で追加。

`locale.getpreferredencoding([do_setlocale])`

テキストデータをエンコードする方法を、ユーザの設定に基づいて返します。ユーザの設定は異なるシステム間では異なった方法で表現され、システムによってはプログラミング的に得ることができないこともあるので、この関数が返すのはただの推測です。

システムによっては、ユーザの設定を取得するために `setlocale()` を呼び出す必要があるため、この関数はスレッド安全ではありません。`setlocale()` を呼び出す必要がない、または呼び出したくない場合、*do_setlocale* を False に設定する必要があります。

バージョン 2.3 で追加。

`locale.normalize(localename)`

与えたロケール名を規格化したロケールコードを返します。返されるロケールコードは `setlocale()` で使うために書式化されています。規格化が失敗した場合、もとの名前がそのまま返されます。

与えたエンコードがシステムにとって未知の場合、標準の設定では、この関数は `setlocale()` と同様に、エンコーディングをロケールコードにおける標準のエンコーディングに設定します。

バージョン 2.0 で追加。

`locale.resetlocale([category])`

category のロケールを標準設定にします。

標準設定は `getdefaultlocale()` を呼ぶことで決定されます。 `category` は標準で `LC_ALL` になっています。

バージョン 2.0 で追加.

`locale.strcoll(string1, string2)`

現在の `LC_COLLATE` 設定に従って二つの文字列を比較します。他の比較を行う関数と同じように、`string1` が `string2` に対して前に来るか、後に来るか、あるいは二つが等しいかによって、それぞれ負の値、正の値、あるいは 0 を返します。

`locale.strxfrm(string)`

文字列を組み込み関数 `cmp()` で使える形式に変換し、かつロケールに則した結果を返します。この関数は同じ文字列が何度も比較される場合、例えば文字列からなるシーケンスを順序付けて並べる際に使うことができます。

`locale.format(format, val[, grouping[, monetary]])`

数値 `val` を現在の `LC_NUMERIC` の設定に基づいて書式化します。書式は % 演算子の慣行に従います。浮動小数点数については、必要に応じて浮動小数点が変更されます。 `grouping` が真なら、ロケールに配慮した桁数の区切りが行われます。

`monetary` が真なら、桁区切り記号やグループ化文字列を用いて変換を行います。

この関数や、1 文字の指定子でしか動作しないことに注意しましょう。フォーマット文字列を使う場合は `format.string()` を使用します。

バージョン 2.5 で変更: `monetary` パラメータが追加されました。

`locale.format_string(format, val[, grouping])`

`format % val` 形式のフォーマット指定子を、現在のロケール設定を考慮したうえで処理します。

バージョン 2.5 で追加.

`locale.currency(val[, symbol[, grouping[, international]]])`

数値 `val` を、現在の `LC_MONETARY` の設定にあわせてフォーマットします。

`symbol` が真の場合は、返される文字列に通貨記号が含まれるようになります。これはデフォルトの設定です。 `grouping` が真の場合 (これはデフォルトではありません) は、値をグループ化します。 `international` が真の場合 (これはデフォルトではありません) は、国際的な通貨記号を使用します。

この関数は 'C' ロケールでは動作しないことに注意しましょう。まず最初に `setlocale()` でロケールを設定する必要があります。

バージョン 2.5 で追加.

`locale.str(float)`

浮動小数点数を `str(float)` と同じように書式化しますが、ロケールに配慮した小数点が使われます。

`locale.atof(string)`

文字列を `LC_NUMERIC` で設定された慣行に従って浮動小数点に変換します。

`locale.atoi(string)`

文字列を `LC_NUMERIC` で設定された慣行に従って整数に変換します。

locale.LC_CTYPE

文字タイプ関連の関数のためのロケールカテゴリです。このカテゴリの設定に従って、モジュール *string* における関数の振る舞いが変わります。

locale.LC_COLLATE

文字列を並べ替えるためのロケールカテゴリです。 *locale* モジュールの関数 *strcoll()* および *strxfrm()* が影響を受けます。

locale.LC_TIME

時刻を書式化するためのロケールカテゴリです。 *time.strftime()* はこのカテゴリに設定されている慣行に従います。

locale.LC_MONETARY

金額に関係する値を書式化するためのロケールカテゴリです。設定可能なオプションは関数 *localeconv()* で得ることができます。

locale.LC_MESSAGES

メッセージ表示のためのロケールカテゴリです。現在 Python はアプリケーション毎にロケールに対応したメッセージを出力する機能はサポートしていません。 *os.strerror()* が返すような、オペレーティングシステムによって表示されるメッセージはこのカテゴリによって影響を受けます。

locale.LC_NUMERIC

数字を書式化するためのロケールカテゴリです。関数 *format()*、*atoi()*、*atof()* および *locale* モジュールの *str()* が影響を受けます。他の数値書式化操作は影響を受けません。

locale.LC_ALL

全てのロケール設定を総合したものです。ロケールを変更する際にこのフラグが使われた場合、そのロケールにおける全てのカテゴリを設定しようと試みます。一つでも失敗したカテゴリがあった場合、全てのカテゴリにおいて設定変更を行いません。このフラグを使ってロケールを取得した場合、全てのカテゴリにおける設定を示す文字列が返されます。この文字列は、後に設定を元に戻すために使うことができます。

locale.CHAR_MAX

localeconv() の返す特別な値のためのシンボル定数です。

例:

```
>>> import locale
>>> loc = locale.getlocale() # get current locale
# use German locale; name might vary with platform
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
>>> locale.strcoll('f\xxe4n', 'foo') # compare a string containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '') # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C') # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc) # restore saved locale
```

22.2.1 ロケールの背景、詳細、ヒント、助言および補足説明

C 標準では、ロケールはプログラム全体にわたる特性であり、その変更は高価な処理であるとしています。加えて、頻繁にロケールを変更するようなひどい実装はコアダンプを引き起こすこともあります。このことがロケールを正しく利用する上で苦痛となっています。

そもそも、プログラムが起動した際、ロケールはユーザの希望するロケールにかかわらず C です。プログラムは `setlocale(LC_ALL, '')` を呼び出して、明示的にユーザの希望するロケール設定を行わなければなりません。

`setlocale()` をライブラリルーチン内で呼ぶことは、それがプログラム全体に及ぼす副作用の面から、一般的によくはない考えです。ロケールを保存したり復帰したりするのもよくありません: 高価な処理であり、ロケールの設定が復帰する以前に起動してしまった他のスレッドに悪影響を及ぼすからです。

もし、汎用を目的としたモジュールを作っていて、ロケールによって影響をうけるような操作 (例えば `string.lower()` や `time.strftime()` の書式の一部) のロケール独立のバージョンが必要ということになれば、標準ライブラリルーチンを使わずに何とかしなければなりません。よりましな方法は、ロケール設定が正しく利用できているか確かめることです。最後の手段は、あなたのモジュールが C ロケール以外の設定には互換性がないとドキュメントに書くことです。

`string` モジュールの大小文字の変換を行う関数はロケール設定によって影響を受けます。 `setlocale()` 関数を呼んで `LC_CTYPE` 設定を変更した場合、変数 `string.lowercase`、`string.uppercase` および `string.letters` は計算しなおされます。例えば `from string import letters` のように、`'from ... import ...'` を使ってこれらの変数を使っている場合には、それ以降の `setlocale()` の影響を受けないので注意してください。

ロケールに従って数値操作を行うための唯一の方法はこのモジュールで特別に定義されている関数: `atof()`、`atoi()`、`format()`、`str()` を使うことです。

22.2.2 Python 拡張の作者と、Python を埋め込むようなプログラムに関して

拡張モジュールは、現在のロケールを調べる以外は、決して `setlocale()` を呼び出してはなりません。しかし、返される値もロケールの復帰のために使えるだけなので、さほど便利とはいえません (例外はおそらくロケールが C かどうか調べることでしょう)。

ロケールを変更するために Python コードで `locale` モジュールを使った場合、Python を埋め込んでいるアプリケーションにも影響を及ぼします。Python を埋め込んでいるアプリケーションに影響が及ぶことを望まない場合、`config.c` ファイル内の組み込みモジュールのテーブルから `_locale` 拡張モジュール (ここで全てを行っています) を削除し、共有ライブラリから `_locale` モジュールにアクセスできないようにしてください。

22.2.3 メッセージカタログへのアクセス

```
locale.gettext(msg)
```

```
locale.dgettext(domain, msg)
```

`locale.dgettext (domain, msg, category)`

`locale.textdomain (domain)`

`locale.bindtextdomain (domain, dir)`

C ライブラリの `gettext` インタフェースが提供されているシステムでは、`locale` モジュールでそのインタフェースを公開しています。このインタフェースは関数 `gettext()`、`dgettext()`、`dcgettext()`、`textdomain()`、`bindtextdomain()`、`bindtextdomain_codeset()` からなります。これらは `gettext` モジュールの同名の関数に似ていますが、メッセージカタログとして C ライブラリのバイナリフォーマットを使い、メッセージカタログを探すために C ライブラリの検索アルゴリズムを使います。

Python アプリケーションでは、通常これらの関数を呼び出す必要はないはずで、代わりに `gettext` を呼び出すべきです。例外として知られているのは、内部で `gettext()` または `dcgettext()` を呼び出すような C ライブラリにリンクするアプリケーションです。こうしたアプリケーションでは、ライブラリが正しいメッセージカタログを探せるようにテキストドメイン名を指定する必要があります。

第 23 章

プログラムのフレームワーク

この章で解説されるモジュールはあなたのプログラムの大枠を規定するフレームワークです。現状では、ここで解説されるモジュールは全てコマンドラインインタフェースを書くためのものです。

この章で解説されるモジュールの完全な一覧は:

23.1 cmd — 行指向のコマンドインタプリタのサポート

ソースコード: [Lib/cmd.py](#)

`Cmd` クラスでは、行指向のコマンドインタプリタを書くための簡単なフレームワークを提供します。テストハネスや管理ツール、そして、後により洗練されたインタフェースでラップするプロトタイプとして、こうしたインタプリタはよく役に立ちます。

```
class cmd.Cmd([completekey[, stdin[, stdout]]])
```

`Cmd` インスタンス、あるいはサブクラスのインスタンスは、行指向のインタプリタ・フレームワークです。 `Cmd` 自身をインスタンス化することはありません。むしろ、 `Cmd` のメソッドを継承したり、アクションメソッドをカプセル化するために、あなたが自分で定義するインタプリタクラスのスーパークラスとしての便利です。

オプション引数 `completekey` は、補完キーの `readline` 名です。デフォルトは `Tab` です。 `completekey` が `None` でなく、 `readline` が利用できるならば、コマンド補完は自動的に行われます。

オプション引数 `stdin` と `stdout` には、 `Cmd` またはそのサブクラスのインスタンスが入出力に使用するファイルオブジェクトを指定します。省略時には `sys.stdin` と `sys.stdout` が使用されます。

引数に渡した `stdin` を使いたい場合は、インスタンスの `use_rawinput` 属性を `False` にセットしてください。そうしないと `stdin` は無視されます。

バージョン 2.3 で変更: パラメータ `stdin` と `stdout` が追加されました。

23.1.1 Cmd オブジェクト

`Cmd` インスタンスは、次のメソッドを持ちます:

Cmd.cmdloop([intro])

プロンプトを繰り返し出力し、入力を受け取り、受け取った入力から取り去った先頭の語を解析し、その行の残りを引数としてアクションメソッドへディスパッチします。

オプションの引数は、最初のプロンプトの前に表示されるバナーあるいはイントロ用の文字列です (これはクラス属性 `intro` をオーバーライドします)。

`readline` モジュールがロードされているなら、入力は自動的に `bash` のような履歴リスト編集機能を受け継ぎます (例えば、`Control-P` は直前のコマンドへのスクロールバック、`Control-N` は次のものへ進む、`Control-F` はカーソルを右へ非破壊的に進める、`Control-B` はカーソルを非破壊的に左へ移動させる等)。

入力のファイル終端は、文字列 `'EOF'` として渡されます。

メソッド `do_foo()` を持っている場合に限り、インタプリタのインスタンスはコマンド名 `foo` を認識します。特別な場合として、文字 `'?'` で始まる行はメソッド `do_help()` へディスパッチします。他の特別な場合として、文字 `'!'` で始まる行はメソッド `do_shell()` へディスパッチします (このようなメソッドが定義されている場合)。

このメソッドは `postcmd()` メソッドが真を返したときに `return` します。 `postcmd()` に対する `stop` 引数は、このコマンドが対応する `do_*` () メソッドからの戻り値です。

補完が有効になっているなら、コマンドの補完が自動的に行われます。また、コマンド引数の補完は、引数 `text`, `line`, `begidx`, および `endidx` と共に `complete_foo()` を呼び出すことによって行われます。 `text` は、マッチしようとしている文字列の先頭の語です。返されるマッチは全てそれで始まっていなければなりません。 `line` は始めの空白を除いた現在の入力行です。 `begidx` と `endidx` は先頭のテキストの始まりと終わりのインデックスで、引数の位置に依存した異なる補完を提供するのに使えます。

`Cmd` のすべてのサブクラスは、定義済みの `do_help()` を継承します。このメソッドは、(引数 `'bar'` と共に呼ばれたとすると) 対応するメソッド `help_bar()` を呼び出します。そのメソッドが存在しない場合、`do_bar()` の `docstring` があればそれを表示します。引数がないければ、`do_help()` は、すべての利用可能なヘルプ見出し (すなわち、対応する `help_*` () メソッドを持つすべてのコマンドまたは `docstring` を持つコマンド) をリストアップします。また、文書化されていないコマンドでも、すべてリストアップします。

Cmd.onecmd(str)

プロンプトに答えてタイプしたかのように引数を解釈実行します。これをオーバーライドすることがあるかもしれませんが、通常は必要ないでしょう。便利な実行フックについては、`precmd()` と `postcmd()` メソッドを参照してください。戻り値は、インタプリタによるコマンドの解釈実行をやめるかどうかを示すフラグです。コマンド `str` に対応する `do_*` () メソッドがある場合、そのメソッドの戻り値が返されます。そうでない場合は `default()` メソッドからの戻り値が返されます。

Cmd.emptyline()

プロンプトに空行が入力されたときに呼び出されるメソッド。このメソッドがオーバーライドされていないなら、最後に入力された空行でないコマンドが繰り返されます。

Cmd.default(line)

コマンドの先頭の語が認識されないときに、入力行に対して呼び出されます。このメソッドがオーバーライドされていないなら、エラーメッセージを表示して戻ります。

`Cmd.completedefault` (*text, line, begidx, endidx*)

利用可能なコマンド固有の `complete_*` () が存在しないときに、入力行を補完するために呼び出されるメソッド。デフォルトでは、空行を返します。

`Cmd.precmd` (*line*)

コマンド行 *line* が解釈実行される直前、しかし入力プロンプトが作られ表示された後に実行されるフックメソッド。このメソッドは `Cmd` 内のスタブであって、サブクラスでオーバーライドされるために存在します。戻り値は `onecmd()` メソッドが実行するコマンドとして使われます。 `precmd()` の実装では、コマンドを書き換えるかもしれないし、あるいは単に変更していない *line* を返すかもしれません。

`Cmd.postcmd` (*stop, line*)

コマンドディスパッチが終わった直後に実行されるフックメソッド。このメソッドは `Cmd` 内のスタブで、サブクラスでオーバーライドされるために存在します。 *line* は実行されたコマンド行で、 *stop* は `postcmd()` の呼び出しの後に実行を停止するかどうかを示すフラグです。これは `onecmd()` メソッドの戻り値です。このメソッドの戻り値は、 *stop* に対応する内部フラグの新しい値として使われます。偽を返すと、実行を続けます。

`Cmd.preloop` ()

`cmdloop()` が呼び出されたときに一度だけ実行されるフックメソッド。このメソッドは `Cmd` 内のスタブであって、サブクラスでオーバーライドされるために存在します。

`Cmd.postloop` ()

`cmdloop()` が戻る直前に一度だけ実行されるフックメソッド。このメソッドは `Cmd` 内のスタブであって、サブクラスでオーバーライドされるために存在します。

`Cmd` のサブクラスのインスタンスは、公開されたインスタンス変数をいくつか持っています:

`Cmd.prompt`

入力を求めるために表示されるプロンプト。

`Cmd.identchars`

コマンドの先頭の語として受け入れられる文字の文字列。

`Cmd.lastcmd`

最後の空でないコマンドプリフィックス。

`Cmd.cmdqueue`

キューに入れられた入力行のリスト。 `cmdqueue` リストは新たな入力が必要な際に `cmdloop()` 内でチェックされます; これが空でない場合、その要素は、あたかもプロンプトから入力されたかのように順に処理されます。

`Cmd.intro`

イントロあるいはバナーとして表示される文字列。 `cmdloop()` メソッドに引数を与えるために、オーバーライドされるかもしれません。

`Cmd.doc_header`

ヘルプ出力に文書化されたコマンドのセクションがある場合に表示するヘッダ。

Cmd.misc_header

ヘルプの出力にその他のヘルプ見出しがある (すなわち、`do_*()` メソッドに対応していない `help_*()` メソッドが存在する) 場合に表示するヘッダ。

Cmd.undoc_header

ヘルプ出力に文書化されていないコマンドのセクションがある (すなわち、対応する `help_*()` メソッドを持たない `do_*()` メソッドが存在する) 場合に表示するヘッダ。

Cmd.ruler

ヘルプメッセージのヘッダの下に、区切り行を表示するために使われる文字。空のときは、ルーラ行が表示されません。デフォルトでは、`'='` です。

Cmd.use_rawinput

フラグで、デフォルトでは真です。真ならば、`cmdloop()` はプロンプトを表示して次のコマンド読み込むために `raw_input()` を使います。偽ならば、`sys.stdout.write()` と `sys.stdin.readline()` が使われます。(これが意味するのは、`readline` を `import` することによって、それをサポートするシステム上では、インタプリタが自動的に **Emacs** 形式の行編集とコマンド履歴のキーストロークをサポートするということです。)

23.2 shlex — 単純な字句解析

バージョン 1.5.2 で追加。

ソースコード: [Lib/shlex.py](#)

`shlex` クラスは Unix シェルを思わせる単純な構文に対する字句解析器を簡単に書けるようにします。このクラスはしばしば、Python アプリケーションのための実行制御ファイルのような小規模言語を書く上で便利です。

Python 2.7.3 以前はこのモジュールは今のところ Unicode 入力をサポートしていませんでした。

`shlex` モジュールは以下の関数を定義します。

`shlex.split(s[, comments[, posix]])`

シェル類似の文法を使って、文字列 `s` を分割します。`comments` が `False` (デフォルト値) の場合、受理した文字列内のコメントを解析しません (`shlex` インスタンスの `commenters` メンバの値を空文字列にします)。この関数はデフォルトでは POSIX モードで動作し、`posix` 引数が `false` の場合は non-POSIX モードで動作します。

バージョン 2.3 で追加。

バージョン 2.6 で変更: `posix` パラメータを追加。

注釈: `split()` 関数は `shlex` クラスのインスタンスを利用するので、`s` に `None` を渡すと標準入力から分割する文字列を読み込みます。

`shlex` モジュールは以下のクラスを定義します。

```
class shlex.shlex([instream[, infile[, posix]]])
```

`shlex` クラスとサブクラスのインスタンスは、字句解析器オブジェクトです。初期化引数を与えると、どこから文字を読み込むかを指定できます。指定先は `read()` メソッドと `readline()` メソッドを持つファイル/ストリーム類似オブジェクトか、文字列でなくてはなりません（文字列が受理されるようになったのは Python 2.3 以降）。引数を与えられなければ、`sys.stdin` から入力を受け付けます。第 2 引数は、ファイル名を表す文字列で、`infile` メンバの値の初期値を決定します。`instream` 引数が省略された場合や、この値が `sys.stdin` である場合、第 2 引数のデフォルト値は `"stdin"` になります。`posix` 引数は Python 2.3 で導入されました。これは動作モードを定義します。`posix` が真でない場合（デフォルト）`shlex` インスタンスは互換モードで動作します。POSIX モードで動作中、`shlex` は、できる限り POSIX シェルの解析規則に似せようとします。

参考:

`ConfigParser` モジュール Windows `.ini` ファイルに似た設定ファイルのパパー。

23.2.1 shlex オブジェクト

`shlex` インスタンスは以下のメソッドを持っています:

```
shlex.get_token()
```

トークンを一つ返します。トークンが `push_token()` で使ってスタックに積まれていた場合、トークンをスタックからポップします。そうでない場合、トークンを一つ入力ストリームから読み出します。読み出し即時にファイル終了子に遭遇した場合、`eof` (非 POSIX モードでは空文字列 (' '))、POSIX モードでは `None`) が返されます。

```
shlex.push_token(str)
```

トークンスタックに引数文字列をスタックします。

```
shlex.read_token()
```

生 (raw) のトークンを読み出します。プッシュバックスタックを無視し、かつソースリクエストを解釈しません (通常これは便利なエントリポイントではありません。完全性のためにここで記述されています)。

```
shlex.sourcehook(filename)
```

`shlex` がソースリクエスト (下の `source` を参照してください) を検出した際、このメソッドはその後続くトークンを引数として渡され、ファイル名と開かれたファイル類似オブジェクトからなるタプルを返すとされています。

通常、このメソッドはまず引数から何らかのクオートを剥ぎ取ります。処理後の引数が絶対パス名であった場合か、以前に有効になったソースリクエストが存在しない場合か、以前のソースが (`sys.stdin` のような) ストリームであった場合、この結果はそのままにされます。そうでない場合で、処理後の引数が相対パス名の場合、ソースインクルードスタックにある直前のファイル名からディレクトリ部分が取り出され、相対パスの前の部分に追加されます (この動作は C 言語プリプロセッサにおける `#include "file.h"` の扱いと同様です)。

これらの操作の結果はファイル名として扱われ、タプルの最初の要素として返されます。同時にこのファイル名で `open()` を呼び出した結果が二つ目の要素になります (注意: インスタンス初期化のときとは引数の並びが逆になっています！)

このフックはディレクトリサーチパスや、ファイル拡張子の追加、その他の名前空間に関するハックを実装できるようにするために公開されています。'close' フックに対応するものではありませんが、shlex インスタンスはソースリクエストされている入力ストリームが EOF を返した時には `close()` を呼び出します。

ソーススタックをより明示的に操作するには、`push_source()` および `pop_source()` メソッドを使ってください。

`shlex.push_source(stream[, filename])`

入力ソースストリームを入力スタックにプッシュします。ファイル名引数が指定された場合、以後のエラーメッセージ中で利用することができます。 `sourcehook()` メソッドが内部で使用しているのと同じメソッドです。

バージョン 2.1 で追加。

`shlex.pop_source()`

最後にプッシュされた入力ソースを入力スタックからポップします。字句解析器がスタック上の入力ストリームの EOF に到達した際に利用するメソッドと同じです。

バージョン 2.1 で追加。

`shlex.error_leader([file[, line]])`

このメソッドはエラーメッセージの論述部分を Unix C コンパイラエラーラベルの形式で生成します; この書式は `'"%s", line %d: '` で、`%s` は現在のソースファイル名で置き換えられ、`%d` は現在の入力行番号で置き換えられます (オプションの引数を使ってこれらを上書きすることもできます)。

このやり方は、`shlex` のユーザに対して、Emacs やその他の Unix ツール群が解釈できる一般的な書式でのメッセージを生成することを推奨するために提供されています。

`shlex` サブクラスのインスタンスは、字句解析を制御したり、デバッグに使えるような public なインスタンス変数を持っています:

`shlex.commenters`

コメントの開始として認識される文字列です。コメントの開始から行末までのすべてのキャラクタ文字は無視されます。標準では単に `'#'` が入っています。

`shlex.wordchars`

複数文字からなるトークンを構成するためにバッファに蓄積していくような文字からなる文字列です。標準では、全ての ASCII 英数字およびアンダースコアが入っています。

`shlex.whitespace`

空白と見なされ、読み飛ばされる文字群です。空白はトークンの境界を作ります。標準では、スペース、タブ、改行 (linefeed) および復帰 (carriage-return) が入っています。

`shlex.escape`

エスケープ文字と見なされる文字群です。これは POSIX モードでのみ使われ、デフォルトでは `'\'` だ

けが入っています。

バージョン 2.3 で追加.

`shlex.quotes`

文字列引用符と見なされる文字群です。トークンを構成する際、同じクオートが再び出現するまで文字をバッファに蓄積します (すなわち、異なるクオート形式はシェル中で互いに保護し合う関係にあります)。標準では、ASCII 単引用符および二重引用符が入っています。

`shlex.escapedquotes`

`quotes` のうち、`escape` で定義されたエスケープ文字を解釈する文字群です。これは POSIX モードでのみ使われ、デフォルトでは `'` だけが入っています。

バージョン 2.3 で追加.

`shlex.whitespace_split`

この値が `True` であれば、トークンは空白文字でのみで分割されます。たとえば `shlex` がシェル引数と同じ方法で、コマンドラインを解析するのに便利です。

バージョン 2.3 で追加.

`shlex.infile`

現在の入力ファイル名です。クラスのインスタンス化時に初期設定されるか、その後のソースリクエストでスタックされます。エラーメッセージを構成する際にこの値を調べると便利ことがあります。

`shlex.instream`

`shlex` インスタンスが文字を読み出している入力ストリームです。

`shlex.source`

このメンバ変数は標準で `None` を取ります。この値に文字列を代入すると、その文字列は多くのシェルにおける `source` キーワードに似た、字句解析レベルでのインクルード要求として認識されます。すなわち、その直後に現れるトークンをファイル名として新たなストリームを開き、そのストリームを入力として、EOF に到達するまで読み込まれます。新たなストリームの EOF に到達した時点で `close()` が呼び出され、入力元の入力ストリームに戻されます。ソースリクエストは任意のレベルの深さまでスタックしてかまいません。

`shlex.debug`

このメンバ変数が数値で、かつ 1 またはそれ以上の値の場合、`shlex` インスタンスは動作に関する冗長な進捗報告を出力します。この出力を使いたいなら、モジュールのソースコードを読めば詳細を学ぶことができます。

`shlex.lineno`

ソース行番号 (遭遇した改行の数に 1 を加えたもの) です。

`shlex.token`

トークンバッファです。例外を捕捉した際にこの値を調べると便利ことがあります。

`shlex.eof`

ファイルの終端を決定するのに使われるトークンです。非 POSIX モードでは空文字列 (`'`)、POSIX モードでは `None` が入ります。

バージョン 2.3 で追加.

23.2.2 解析規則

非 POSIX モードで動作中の *shlex* は以下の規則に従おうとします。

- ワード内の引用符を認識しない (`Do"Not"Separate` は単一ワード `Do"Not"Separate` として解析されます)
- エスケープ文字を認識しない
- 引用符で囲まれた文字列は、引用符内の全ての文字リテラルを保持する
- 閉じ引用符でワードを区切る (`"Do"Separate` は、`"Do"` と `Separate` であると解析されます)
- *whitespace.split* が `False` の場合、*wordchar*、*whitespace* または *quote* として宣言されていない全ての文字を、単一の文字トークンとして返す。 `True` の場合、*shlex* は空白文字でのみ単語を区切る。
- 空文字列 (`' '`) で EOF を送出する
- 引用符に囲んであっても、空文字列を解析しない

POSIX モードで動作中の *shlex* は以下の解析規則に従おうとします。

- 引用符を取り除き、引用符で単語を分解しない (`"Do"Not"Separate"` は単一ワード `DoNotSeparate` として解析されます)
- 引用符で囲まれないエスケープ文字群 (`'\'` など) は直後に続く文字のリテラル値を保持する
- *escapedquotes* でない引用符文字 (`"'"` など) で囲まれている全ての文字のリテラル値を保持する
- 引用符に囲まれた *escapedquotes* に含まれる文字 (`'"'` など) は、*escape* に含まれる文字を除き、全ての文字のリテラル値を保持する。エスケープ文字群は使用中の引用符、または、そのエスケープ文字自身が直後にある場合のみ、特殊な機能を保持する。他の場合にはエスケープ文字は普通の文字とみなされる。
- *None* で EOF を送出する
- 引用符に囲まれた空文字列 (`' '`) を許す

第 24 章

Tk を用いたグラフィカルユーザインターフェイス

Tk/Tcl は長きにわたり Python の不可欠な一部でありつづけています。Tk/Tcl は頑健でプラットフォームに依存しないウィンドウ構築ツールキットであり、Python プログラマは *Tkinter* モジュールやその拡張の *Tix*, *ttk* モジュールを使って利用できます。

Tkinter モジュールは、Tcl/Tk 上に作られた軽量なオブジェクト指向のレイヤです。*Tkinter* を使うために Tcl コードを書く必要はありませんが、Tk のドキュメントや、場合によっては Tcl のドキュメントを調べる必要があるでしょう。*Tkinter* は Tk のウィジェットを Python のクラスとして実装しているラッパをまとめたものです。加えて、内部モジュール `_tkinter` では、Python と Tcl がやり取りできるようなスレッド安全なメカニズムを提供しています。

Tkinter の一番素晴らしい点は速く、そして普通に Python に付属してくることです。標準ドキュメントが頼りないものだとしても、代わりになるものが入手可能です: リファレンス、チュートリアル、書籍その他です。*Tkinter* は古臭いルックアンドフィールでも有名ですが、その点は Tk 8.5 で幅広く改善されました。とはいえ、興味を引きそうな GUI ライブラリは他にも多数あります。そういったものについてはもっと知りたい人は [他のグラフィカルユーザインタフェースパッケージ](#) 節を参照してください。

24.1 Tkinter — Tcl/Tk への Python インタフェース

Tkinter モジュール (“Tk インタフェース”) は、Tk GUI ツールキットに対する標準の Python インタフェースです。Tk と *Tkinter* はほとんどの Unix プラットフォームの他、Windows システム上でも利用できます。(Tk 自体は Python の一部ではありません。Tk は ActiveState で保守されています。)

Running `python -m Tkinter` from the command line should open a window demonstrating a simple Tk interface, letting you know that *Tkinter* is properly installed on your system, and also showing what version of Tcl/Tk is installed, so you can read the Tcl/Tk documentation specific to that version.

注釈: *Tkinter* モジュールは、Python 3 では `tkinter` にリネームされました。*2to3* ツールが自動的にソースコードの `import` を修正します。

参考:

Tkinter ドキュメント:

[Python Tkinter Resources](#) Python Tkinter Topic Guide では、Tk を Python から利用する上での情報と、その他の Tk にまつわる情報源を数多く提供しています。

[TKDocs](#) 豊富なチュートリアルといくつかのウィジェットについてのよりフレンドリなページ。

[Tkinter reference: a GUI for Python](#) オンラインリファレンス資料です。

[Tkinter docs from effbot](#) effbot.org が提供している tkinter のオンラインリファレンスです。

[Programming Python](#) マーク・ルッツによる書籍で、Tkinter についても広く取り上げています。

[Modern Tkinter for Busy Python Developers](#) Mark Roseman による書籍で、Python と Tkinter を使用した魅力的でモダンなグラフィカルユーザーインターフェースの構築方法を説明しています。

[Python and Tkinter Programming](#) John Grayson による解説書 (ISBN 1-884777-81-3) です。

Tcl/Tk ドキュメント:

[Tk commands](#) Most commands are available as `Tkinter` or `Tkinter.ttk` classes. Change '8.6' to match the version of your Tcl/Tk installation.

[Tcl/Tk の最近の man pages](#) www.tcl.tk の最近の Tcl/Tk マニュアルです。

[ActiveState Tcl ホームページ](#) Tk/Tcl の開発は ActiveState で大々的に行われています。

[Tcl and the Tk Toolkit](#) Tcl の発明者である John Ousterhout による本です。

[Practical Programming in Tcl and Tk](#) Brent Welch の百科事典のような本です。

24.1.1 Tkinter モジュール

ほとんどの場合、本当に必要となるのは `Tkinter` モジュールだけですが、他にもいくつかの追加モジュールを利用できます。Tk インタフェース自体は `_tkinter` という名前のバイナリモジュール内にあります。このモジュールに入っているのは Tk への低水準のインタフェースであり、アプリケーションプログラマが直接使ってはなりません。`_tkinter` は通常共有ライブラリ (や DLL) になっていますが、Python インタプリタに静的にリンクされていることもあります。

Tk インタフェースモジュールの他にも、`Tkinter` には Python モジュールが数多く入っています。最も重要なモジュールは、`Tkinter` 自体と `Tkconstants` と呼ばれるモジュールの二つです。前者は自動的に後者を `import` するので、以下のように一方のモジュールを `import` するだけで Tkinter を使えるようになります:

```
import Tkinter
```

あるいは、よく使うやり方で以下のようにします:

```
from Tkinter import *
```

class `Tkinter.Tk` (*screenName=None, baseName=None, className='Tk', useTk=1*)

`Tk` クラスは引数なしでインスタンス化します。これは Tk のトップレベルウィジェットを生成します。通常、トップレベルウィジェットはアプリケーションのメインウィンドウになります。それぞれのインスタンスごとに固有の Tcl インタプリタが関連づけられます。

バージョン 2.4 で変更: `useTk` パラメタが追加されました。

`Tkinter.Tcl` (*screenName=None, baseName=None, className='Tk', useTk=0*)

`Tcl()` はファクトリ関数で、`Tk` クラスで生成するオブジェクトとよく似たオブジェクトを生成します。ただし Tk サブシステムを初期化しません。この関数は、余分なトップレベルウィンドウを作る必要がなかったり、(X サーバを持たない Unix/Linux システムなどのように) 作成できない環境において Tcl インタプリタを駆動したい場合に便利です。`Tcl()` で生成したオブジェクトに対して `loadtk()` メソッドを呼び出せば、トップレベルウィンドウを作成 (して、Tk サブシステムを初期化) します。

バージョン 2.4 で追加。

Tk をサポートしているモジュールには、他にも以下のようなモジュールがあります:

`ScrolledText` 垂直スクロールバー付きのテキストウィジェットです。

`tkColorChooser` ユーザに色を選択させるためのダイアログです。

`tkCommonDialog` このリストの他のモジュールが定義しているダイアログの基底クラスです。

`tkFileDialog` ユーザが開きたいファイルや保存したいファイルを指定できるようにする共通のダイアログです。

`tkFont` フォントの扱いを補助するためのユーティリティです。

`tkMessageBox` 標準的な Tk のダイアログボックスにアクセスします。

`tkSimpleDialog` 基本的なダイアログと便宜関数 (convenience function) です。

`Tkdnd` `Tkinter` 用のドラッグアンドドロップのサポートです。実験的なサポートで、Tk DND に置き替わった時点で撤廃されるはずです。

`turtle` Tk ウィンドウ上でタートルグラフィックスを実現します。

これらも Python 3 で改名されました。新たな `tkinter` パッケージのサブモジュールになったのです。

24.1.2 Tkinter お助け手帳

この節は、Tk や Tkinter を全て網羅したチュートリアルを目指しているわけではありません。むしろ、Tkinter のシステムを学ぶ上での指針を示すための、その場しのぎ的なマニュアルです。

謝辞:

- Tkinter は Steen Lumholt と Guido van Rossum が作成しました。
- Tk は John Ousterhout が Berkeley の在籍中に作成しました。
- この Life Preserver は Virginia 大学の Matt Conway 他が執筆しました。

- html へのレンダリングやたくさんの編集は、Ken Manheimer が FrameMaker 版から行いました。
- Fredrik Lundh はクラスインタフェース詳細な説明を書いたり内容を改訂したりして、現行の Tk 4.2 に合うようにしました。
- Mike Clarkson はドキュメントを LaTeX 形式に変換し、リファレンスマニュアルのユーザインタフェースの章をコンパイルしました。

この節の使い方

この節は二つの部分で構成されています: 前半では、背景となることばを (大雑把に) 網羅しています。後半は、キーボードの横に置けるような手軽なリファレンスになっています。

When trying to answer questions of the form "how do I do blah", it is often best to find out how to do "blah" in straight Tk, and then convert this back into the corresponding *Tkinter* call. Python programmers can often guess at the correct Python command by looking at the Tk documentation. This means that in order to use Tkinter, you will have to know a little bit about Tk. This document can't fulfill that role, so the best we can do is point you to the best documentation that exists. Here are some hints:

- Tk の man マニュアルのコピーを手に入れるよう強く勧めます。とりわけ最も役立つのは `man3` ディレクトリ内にあるマニュアルです。man3 のマニュアルページは Tk ライブラリに対する C インタフェースについての説明なので、スクリプト書きにとって取り立てて役に立つ内容ではありません。
- Addison-Wesley は John Ousterhout の書いた Tcl and the Tk Toolkit (ISBN 0-201-63337-X) という名前の本を出版しています。この本は初心者向けの Tcl と Tk の良い入門書です。内容は網羅的ではなく、詳細の多くは man ページ任せにしています。
- たいいていの場合、`Tkinter.py` は参照先としては最後の地 (last resort) ですが、それ以外の手段で調べても分からない場合には救いの地 (good place) になるかもしれません。

簡単な Hello World プログラム

```
from Tkinter import *

class Application(Frame):
    def say_hi(self):
        print "hi there, everyone!"

    def createWidgets(self):
        self.QUIT = Button(self)
        self.QUIT["text"] = "QUIT"
        self.QUIT["fg"] = "red"
        self.QUIT["command"] = self.quit

        self.QUIT.pack({"side": "left"})

        self.hi_there = Button(self)
        self.hi_there["text"] = "Hello",
        self.hi_there["command"] = self.say_hi
```

(次のページに続く)

(前のページからの続き)

```

        self.hi_there.pack({"side": "left"})

    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

root = Tk()
app = Application(master=root)
app.mainloop()
root.destroy()

```

24.1.3 Tcl/Tk を (本当に少しだけ) 見渡してみる

クラス階層は複雑に見えますが、実際にプログラムを書く際には、アプリケーションプログラマはほとんど常にクラス階層の最底辺にあるクラスしか参照しません。

注釈:

- クラスのいくつかは、特定の関数を一つの名前空間下にまとめるために提供されています。こうしたクラスは個別にインスタンス化するためのものではありません。
- `Tk` クラスはアプリケーション内で一度だけインスタンス化するようになっています。アプリケーションプログラマが明示的にインスタンス化する必要はなく、他のクラスがインスタンス化されると常にシステムが作成します。
- `Widget` クラスもまた、インスタンス化して使うようにはなっていません。このクラスはサブクラス化して「実際の」ウィジェットを作成するためのものです。(C++ で言うところの、' 抽象クラス (abstract class) ' です)。

このリファレンス資料を活用するには、Tk の短いプログラムを読んだり、Tk コマンドの様々な側面を知っておく必要がままあるでしょう。(下の説明の *Tkinter* 版は、[基本的な Tk プログラムと Tkinter との対応関係](#) 節を参照してください。)

Tk スクリプトは Tcl プログラムです。全ての Tcl プログラムに同じく、Tk スクリプトはトークンをスペースで区切って並べます。Tk ウィジェットとは、ウィジェットのクラス、ウィジェットの設定を行う オプション、そしてウィジェットに役立つことをさせる アクション を組み合わせたものに過ぎません。

Tk でウィジェットを作るには、常に次のような形式のコマンドを使います:

```
classCommand newPathname options
```

classCommand どの種類のウィジェット (ボタン、ラベル、メニュー、...) を作るかを表します。

newPathname 作成するウィジェットにつける新たな名前です。Tk 内の全ての名前は一意になっていなければなりません。一意性を持たせる助けとして、Tk 内のウィジェットは、ファイルシステムにおけるファイルと同様、パス名 (*pathname*) を使って名づけられます。トップレベルのウィジェット、すなわ

ちルートは `.` (ピリオド) という名前になり、その子ウィジェット階層もピリオドで区切ってゆきます。ウィジェットのの名前は、例えば `.myApp.controlPanel.okButton` のようになります。

options ウィジェットの見た目を設定します。場合によってはウィジェットの挙動も設定します。オプションはフラグと値がリストになった形式をとります。Unix のシェルコマンドのフラグと同じように、フラグの前には `'-'` がつき、複数の単語からなる値はクオートで囲まれます。

例えば:

```
button      .fred      -fg red -text "hi there"
  ^          ^          |
  |          |          |
class      new          options
command widget (-opt val -opt val ...)
```

ウィジェットを作成すると、ウィジェットへのパス名は新しいコマンドになります。この新たな *widget command* は、プログラマが新たに作成したウィジェットに *action* を実行させる際のハンドル (handle) になります。C では `someAction(fred, someOptions)` と表し、C++ では `fred.someAction(someOptions)` と表すでしょう。Tk では:

```
.fred someAction someOptions
```

のようにします。オブジェクト名 `.fred` はドットから始まっている点に注意してください。

読者の想像の通り、*someAction* に指定できる値はウィジェットのクラスに依存しています: `fred` がボタンなら `.fred disable` はうまくいきます (`fred` はグレーになります) が、`fred` がラベルならうまくいきません (Tk ではラベルの無効化をサポートしていないからです)。

someOptions に指定できる値はアクションの内容に依存しています。 `disable` のようなアクションは引数を必要としませんが、テキストエントリボックスの `delete` コマンドのようなアクションにはテキストを削除する範囲を指定するための引数が必要になります。

24.1.4 基本的な Tk プログラムと Tkinter との対応関係

Tk のクラスコマンドは、Tkinter のクラスコンストラクタに対応しています。

```
button .fred                                =====> fred = Button()
```

オブジェクトの親 (master) は、オブジェクトの作成時に指定した新たな名前から非明示的に決定されます。Tkinter では親を明示的に指定します。

```
button .panel.fred                          =====> fred = Button(panel)
```

Tk の設定オプションは、ハイフンをつけたタグと値の組からなるリストで指定します。Tkinter では、オプションはキーワード引数にしてインスタンスのコンストラクタに指定したり、`config()` にキーワード引数を指定して呼び出したり、インデクス指定を使ってインスタンスに代入したりして設定します。オプションの設定については [オプションの設定](#) 節を参照してください。

```
button .fred -fg red          =====> fred = Button(panel, fg = "red")
.fred configure -fg red      =====> fred["fg"] = red
                                OR ==> fred.config(fg = "red")
```

Tk でウィジェットにアクションを実行させるには、ウィジェット名をコマンドにして、その後にアクション名を続け、必要に応じて引数 (オプション) を続けます。Tkinter では、クラスインスタンスのメソッドを呼び出して、ウィジェットのアクションを呼び出します。あるウィジェットがどんなアクション (メソッド) を実行できるかは、Tkinter.py モジュール内にリストされています。

```
.fred invoke                  =====> fred.invoke()
```

Tk でウィジェットを packer (ジオメトリマネージャ) に渡すには、pack コマンドをオプション引数付きで呼び出します。Tkinter では Pack クラスがこの機能すべてを握っていて、様々な pack の形式がメソッドとして実装されています。Tkinter のウィジェットは全て Packer からサブクラス化されているため、pack 操作にまつわる全てのメソッドを継承しています。Form ジオメトリマネージャに関する詳しい情報については *Tix* モジュールのドキュメントを参照してください。

```
pack .fred -side left        =====> fred.pack(side = "left")
```

24.1.5 Tk と Tkinter はどのように関わっているのか

上から下に、呼び出しの階層構造を説明してゆきます:

あなたのアプリケーション (Python) まず、Python アプリケーションが *Tkinter* を呼び出します。

Tkinter (Python モジュール) 上記の呼び出し (例えば、ボタンウィジェットの作成) は、Tkinter モジュール内で実現されており、Python で書かれています。この Python で書かれた関数は、コマンドと引数を解析して変換し、あたかもコマンドが Python スクリプトではなく Tk スクリプトから来たようにみせかけます。

tkinter (C) 上記のコマンドと引数は *tkinter* (小文字です。注意してください) 拡張モジュール内の C 関数に渡されます。

Tk ウィジェット (C および Tcl) 上記の C 関数は、Tk ライブラリを構成する C 関数の入った別の C モジュールへの呼び出しを行えるようになっています。Tk は C と Tcl を少し使って実装されています。Tk ウィジェットの Tcl 部分は、ウィジェットのデフォルト動作をバインドするために使われ、Python で書かれた *Tkinter* モジュールが import される時点で一度だけ実行されます。(ユーザがこの過程を目にすることはありません。)

Tk (C) Tk ウィジェットの Tk 部分で実装されている最終的な対応付け操作によって...

Xlib (C) Xlib ライブラリがスクリーン上にグラフィックスを描きます。

24.1.6 簡単なリファレンス

オプションの設定

オプションは、色やウィジェットの境界線幅などを制御します。オプションの設定には三通りの方法があります:

オブジェクト生成時、キーワード引数を使用する

```
fred = Button(self, fg = "red", bg = "blue")
```

オブジェクト生成後、オプション名を辞書インデックスのように扱う

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

オブジェクト生成後に、`config()` メソッドを使って複数の属性を更新する

```
fred.config(fg = "red", bg = "blue")
```

オプションとその振る舞いに関する詳細な説明は、該当するウィジェットの Tk の man ページを参照してください。

man ページには、各ウィジェットの "STANDARD OPTIONS (標準オプション)" と "WIDGET SPECIFIC OPTIONS (ウィジェット固有のオプション)" がリストされていることに注意してください。前者は多くのウィジェットに共通のオプションのリストで、後者は特定のウィジェットに特有のオプションです。標準オプションの説明は man ページの *options(3)* にあります。

このドキュメントでは、標準オプションとウィジェット固有のオプションを区別していません。オプションによっては、ある種のウィジェットに適用できません。あるウィジェットがあるオプションに対応しているかどうかは、ウィジェットのクラスによります。例えばボタンには `command` オプションがありますが、ラベルにはありません。

あるウィジェットがどんなオプションをサポートしているかは、ウィジェットの man ページにリストされています。また、実行時にウィジェットの `config()` メソッドを引数なしで呼び出したり、`keys()` メソッドを呼び出したりして問い合わせることもできます。メソッド呼び出しを行うと辞書型の値を返します。この辞書は、オプションの名前がキー (例えば `'relief'`) になっていて、値が 5 要素のタプルになっています。

`bg` のように、いくつかのオプションはより長い名前を持つ共通のオプションに対する同義語になっています (`bg` は "background" を短縮したものです)。短縮形のオプション名を `config()` に渡すと、5 要素ではなく 2 要素のタプルを返します。このタプルには、同義語の名前と「本当の」オプション名が入っています (例えば `('bg', 'background')`)。

インデックス	意味	例
0	オプション名	<code>'relief'</code>
1	データベース検索用のオプション名	<code>'relief'</code>
2	データベース検索用のオプションクラス	<code>'Relief'</code>
3	デフォルト値	<code>'raised'</code>
4	現在の値	<code>'groove'</code>

例:

```
>>> print fred.config()
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

もちろん、実際に出力される辞書には利用可能なオプションが全て表示されます。上の表示例は単なる例にすぎません。

Packer

packer は Tk のジオメトリ管理メカニズムの一つです。ジオメトリマネージャは、複数のウィジェットの位置を、それぞれのウィジェットを含むコンテナ - 共通の マスタ (*master*) からの相対で指定するために使います。やや扱いにくい *placer* (あまり使われないのでここでは取り上げません) と違い、packer は定性的な関係を表す指定子 - 上 (*above*)、~ の左 (*to the left of*)、引き延ばし (*filling*) など - を受け取り、厳密な配置座標の決定を全て行ってくれます。

どんな マスタ ウィジェットでも、大きさは内部の "スレイブ (slave) ウィジェット" の大きさで決まります。packer は、スレイブウィジェットを pack 先のマスタウィジェット中のどこに配置するかを制御するために使われます。望みのレイアウトを達成するには、ウィジェットをフレームにパックし、そのフレームをまた別のフレームにパックできます。さらに、一度パックを行うと、それ以後の設定変更に合わせて動的に並べ方を調整します。

ジオメトリマネージャがウィジェットのジオメトリを確定するまで、ウィジェットは表示されないので注意してください。初心者のころにはよくジオメトリの確定を忘れてしまい、ウィジェットを生成したのに何も表示されず驚くことになります。ウィジェットは、(例えば packer の `pack()` メソッドを適用して) ジオメトリを確定した後で初めて表示されます。

`pack()` メソッドは、キーワード引数つきで呼び出せます。キーワード引数は、ウィジェットをコンテナ内のどこに表示するか、メインのアプリケーションウィンドウをリサイズしたときにウィジェットがどう振舞うかを制御します。以下に例を示します:

```
fred.pack()                                # defaults to side = "top"
fred.pack(side = "left")
fred.pack(expand = 1)
```

Packer のオプション

packer と packer の取りえるオプションについての詳細は、man ページや John Ousterhout の本の 183 ページを参照してください。

anchor アンカーの型です。packer が区画内に各スレイブを配置する位置を示します。

expand ブール値で、0 または 1 になります。

fill 指定できる値は 'x'、'y'、'both'、'none' です。

ipadx および **ipady** スレイブウィジェットの各側面の内側に行うパディング幅を表す長さを指定します。

padx および **pady** スレイブウィジェットの各側面の外側に行うパディング幅を表す長さを指定します。

side 指定できる値は 'left', 'right', 'top', 'bottom' です。

ウィジェット変数を関連付ける

ウィジェットによっては、(テキスト入力ウィジェットのように) 特殊なオプションを使って、現在設定されている値をアプリケーション内の変数に直接関連付けできます。このようなオプションには `variable`, `textvariable`, `onvalue`, `offvalue` および `value` があります。この関連付けは双方向に働きます: 変数の値が何らかの理由で変更されると、関連付けされているウィジェットも更新され、新しい値を反映します。

残念ながら、現在の *Tkinter* の実装では、`variable` や `textvariable` オプションでは任意の Python の値をウィジェットに渡せません。この関連付け機能がうまく働くのは、*Tkinter* モジュール内で `Variable` というクラスからサブクラス化されている変数によるオプションだけです。

`Variable` には、`StringVar`, `IntVar`, `DoubleVar`, `BooleanVar` といった便利なサブクラスがすでにすでに数多く定義されています。こうした変数の現在の値を読み出したければ、`get()` メソッドを呼び出します。また、値を変更したければ `set()` メソッドを呼び出します。このプロトコルに従っている限り、それ以上なにも手を加えなくてもウィジェットは常に現在値に追従します。

例えば:

```
class App(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()

        self.entrythingy = Entry()
        self.entrythingy.pack()

        # here is the application variable
        self.contents = StringVar()
        # set it to some value
        self.contents.set("this is a variable")
        # tell the entry widget to watch this variable
        self.entrythingy["textvariable"] = self.contents

        # and here we get a callback when the user hits return.
        # we will have the program print out the value of the
        # application variable when the user hits return
        self.entrythingy.bind('<Key-Return>',
                               self.print_contents)

    def print_contents(self, event):
        print "hi. contents of entry is now ---->", \
              self.contents.get()
```

ウィンドウマネージャ

Tk には、ウィンドウマネージャとやり取りするための `wm` というユーティリティコマンドがあります。 `wm` コマンドにオプションを指定すると、タイトルや配置、アイコンビットマップなどを操作できます。 *Tkinter*

では、こうしたコマンドは `Wm` クラスのメソッドとして実装されています。トップレベルウィジェットは `Wm` クラスからサブクラス化されているので、`Wm` のメソッドを直接呼び出せます。

あるウィジェットの入っているトップレベルウィンドウを取得したい場合、大抵は単にウィジェットのマスタを参照するだけですみます。とはいえ、ウィジェットがフレーム内にパックされている場合、マスタはトップレベルウィンドウではありません。任意のウィジェットの入っているトップレベルウィンドウを知りたければ `_root()` メソッドを呼び出してください。このメソッドはアンダースコアがついていますが、これはこの関数が `Tkinter` の実装の一部であり、`Tk` の機能に対するインタフェースではないことを示しています。

以下に典型的な使い方の例をいくつか挙げます:

```
from Tkinter import *
class App(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()

# create the application
myapp = App()

#
# here are method calls to the window manager class
#
myapp.master.title("My Do-Nothing Application")
myapp.master.maxsize(1000, 400)

# start the program
myapp.mainloop()
```

Tk オプションデータ型

anchor 指定できる値はコンパスの方位です: "n"、"ne"、"e"、"se"、"s"、"sw"、"w"、"nw"、および "center"。

bitmap 八つの組み込み、名前付きビットマップ: 'error'、'gray25'、'gray50'、'hourglass'、'info'、'questhead'、'question'、'warning'。X ビットマップファイル名を指定するために、"@/usr/contrib/bitmap/gumby.bit" のような @ を先頭に付けたファイルへの完全なパスを与えてください。

boolean 整数 0 または 1、あるいは、文字列 "yes" または "no" を渡すことができます。

callback これは引数を取らない Python 関数ならどれでも構いません。例えば:

```
def print_it():
    print "hi there"
fred["command"] = print_it
```

color 色は `rgb.txt` ファイルの X カラーの名前か、または RGB 値を表す文字列として与えられます。RGB 値を表す文字列は、4 ビット: "#RGB", 8 bit: "#RRGGBB", 12 bit: "#RRRGGBBBB", あるいは、16 bit

"#RRRRGGGGBBBB" の範囲を取ります。ここでは、R,G,B は適切な十六進数ならどんなものでも表します。詳細は、Ousterhout の本の 160 ページを参照してください。

cursor `cursorfont.h` の標準 X カーソル名を、接頭語 `XC_` 無しで使うことができます。例えば、hand カーソル (`XC_hand2`) を得るには、文字列 "hand2" を使ってください。あなた自身のビットマップとマスクファイルを指定することもできます。Ousterhout の本の 179 ページを参照してください。

distance スクリーン上の距離をピクセルか絶対距離のどちらかで指定できます。ピクセルは数値として与えられ、絶対距離は文字列として与えられます。絶対距離を表す文字列は、単位を表す終了文字 (センチメートルには `c`、インチには `i`、ミリメートルには `m`、プリンタのポイントには `p`) を伴います。例えば、3.5 インチは "3.5i" と表現します。

font Tk は {courier 10 bold} のようなリストフォント名形式を使います。正の数のフォントサイズはポイント単位で表され、負の数のサイズはピクセル単位で表されます。

geometry これは `widthxheight` 形式の文字列です。ここでは、ほとんどのウィジェットに対して幅と高さピクセル単位で (テキストを表示するウィジェットに対しては文字単位で) 表されます。例えば:

```
fred["geometry"] = "200x100".
```

justify 指定できる値は文字列です: "left"、"center"、"right"、そして "fill"。

region これは空白で区切られた四つの要素をもつ文字列です。各要素は指定可能な距離です (以下を参照)。例えば: "2 3 4 5" と "3i 2i 4.5i 2i" と "3c 2c 4c 10.43c" は、すべて指定可能な範囲です。

relief ウィジェットのボーダのスタイルが何かを決めます。指定できる値は: "raised"、"sunken"、"flat"、"groove"、と "ridge"。

scrollcommand これはほとんど常にスクロールバー・ウィジェットの `set()` メソッドですが、一引数を取るどんなウィジェットメソッドでもあり得ます。例えば、Python ソース配布の `Demo/tkinter/matt/canvas-with-scrollbars.py` ファイルを参照してください。

wrap: 次の中の一つでなければなりません: "none"、"char"、あるいは "word"。

バインドとイベント

ウィジェットコマンドからの `bind` メソッドによって、あるイベントを待つことと、そのイベント型が起きたときにコールバック関数を呼び出すことができるようになります。bind メソッドの形式は:

```
def bind(self, sequence, func, add='')
```

ここでは:

sequence は対象とするイベントの型を示す文字列です (詳細については、bind の man ページと John Ousterhout の本の 201 ページを参照してください)。

func は一引数を取り、イベントが起きるときに呼び出される Python 関数です。イベント・インスタンスが引数として渡されます。(このように実施される関数は、一般に *callbacks* として知られています。)

`add` はオプションで、`' '` か `'+'` のどちらかです。空文字列を渡すことは、このイベントが関係する他のどんなバインドをもこのバインドが置き換えることを意味します。`'+'` を使う仕方は、この関数がこのイベント型にバインドされる関数のリストに追加されることを意味しています。

例えば:

```
def turnRed(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turnRed)
```

イベントのウィジェットフィールドが `turnRed()` コールバック内でどのようにアクセスされているかに注意してください。このフィールドは X イベントを捕らえるウィジェットを含んでいます。以下の表はあなたがアクセスできる他のイベントフィールドとそれらの Tk での表現方法の一覧です。Tk man ページを参照するときに役に立つでしょう。

Tk	Tkinter Event Field	Tk	Tkinter Event Field
--	-----	--	-----
%f	focus	%A	char
%h	height	%E	send_event
%k	keycode	%K	keysym
%s	state	%N	keysym_num
%t	time	%T	type
%w	width	%W	widget
%x	x	%X	x_root
%y	y	%Y	y_root

index パラメータ

たくさんのウィジェットが渡される `"index"` パラメータを必要とします。これらはテキストウィジェットでの特定の場所や、エントリウィジェットでの特定の文字、あるいは、メニューウィジェットでの特定のメニュー項目を指定するために使われます。

エントリウィジェットのインデックス (インデックス、ビューインデックスなど) エントリウィジェットは表示されているテキスト内の文字位置を参照するオプションを持っています。テキストウィジェットにおけるこれらの特別な位置にアクセスするために、これらの *Tkinter* 関数を使うことができます:

`AtEnd()` テキストの最後の位置を参照します

`AtInsert()` テキストカーソルの位置を参照します

`AtSelFirst()` 選択されたテキストの先頭の位置を指します

`AtSelLast()` 選択されているテキストおよび最終的に選択されたテキストの末尾の位置を示します。

`At(x[, y])` ピクセル位置 `x, y` (テキストを一行だけ含むテキストエントリウィジェットの場合には `y` は使われない) の文字を参照します。

テキストウィジェットのインデックス テキストウィジェットに対するインデックス記法はとても機能が豊富で、Tk man ページでよく説明されています。

メニューのインデックス (`menu.invoke()`、`menu.entryconfig()` など) メニューに対するいくつかのオプションとメソッドは特定のメニュー項目を操作します。メニューインデックスはオプションまたはパラメータのために必要とされるときはいつでも、以下のものを渡すことができます:

- ウィジェット内の数字の先頭からの位置を指す整数。先頭は 0。
- 文字列 `'active'`。現在カーソルがあるメニューの位置を指します。
- 文字列 `"last"`。最後のメニューを指します。
- `@6` のような `@` が前に来る整数。ここでは、整数がメニューの座標系における y ピクセル座標として解釈されます。
- 文字列 `"none"`。どんなメニューエントリもまったく指しておらず、ほとんどの場合、すべてのエントリの動作を停止させるために `menu.activate()` と一緒に使われます。そして、最後に、
- メニューの先頭から一番下までスキャンしたときに、メニューエントリのラベルに一致したパターンであるテキスト文字列。このインデックス型は他すべての後に考慮されることに注意してください。その代わりに、それは `last`、`active` または `none` とラベル付けされたメニュー項目への一致は上のリテラルとして解釈されることを意味します。

画像

Images of different formats can be created through the corresponding subclass of `Tkinter.Image`:

- XBM 形式の画像のための `BitmapImage`。
- PGM, PPM, GIF, PNG 形式の画像のための `PhotoImage`。最後のは Tk 8.6 からサポートされるようになりました。

画像のどちらの型でも `file` または `data` オプションを使って作られます (その上、他のオプションも利用できます)。

`image` オプションがウィジェットにサポートされる場所ならどこでも、画像オブジェクトを使うことができます (例えば、ラベル、ボタン、メニュー)。これらの場合では、Tk は画像への参照を保持しないでしょう。画像オブジェクトへの最後の Python の参照が削除されたときに、画像データも削除されます。そして、どこで画像が使われていようと、Tk は空の箱を表示します。

参考:

[Pillow](#) パッケージにより、BMP, JPEG, TIFF, WebP などの形式のサポートが追加されました。

24.1.7 ファイルハンドラ

Tk を使うとコールバック関数の登録や解除ができ、ファイルディスクリプタに対する入出力が可能になるときに、Tk のメインループからその関数が呼ばれます。ファイルディスクリプタ 1 つにつき、1 つだけハンドラは登録されます。コード例です:

```
import Tkinter
widget = Tkinter.Tk()
mask = Tkinter.READABLE | Tkinter.WRITABLE
widget.tk.createfilehandler(file, mask, callback)
...
widget.tk.deletefilehandler(file)
```

これらの機能は Windows では利用できません。

読み込みに使えるバイト数は分からないので、*BufferedIOBase* クラスや *TextIOBase* クラスの *read()* メソッドおよび *readline()* メソッドを使おうとしないでください。これらは読み込みの際に、あらかじめ決められたバイト数を要求するのです。ソケットには、*recv()* や *recvfrom()* メソッドを使うといいです。その他のファイルには、raw 読み込みか *os.read(file.fileno(), maxbytecount)* を使ってください。

Widget.tk.createfilehandler (*file*, *mask*, *func*)

ファイルハンドラであるコールバック関数 *func* を登録します。 *file* 引数は、(ファイルやソケットオブジェクトのような) *fileno()* メソッドを持つオブジェクトか、整数のファイルディスクリプタとなります。 *mask* 引数は、以下にある 3 つの定数の組み合わせの OR を取ったものです。コールバックは次のように呼ばれます:

```
callback(file, mask)
```

Widget.tk.deletefilehandler (*file*)

ファイルハンドラの登録を解除します。

Tkinter.READABLE

Tkinter.WRITABLE

Tkinter.EXCEPTION

mask 引数で使う定数です。

24.2 ttk — Tk のテーマ付きウィジェット

ttk モジュールは Tk 8.5 で導入された Tk のテーマ付きウィジェットへのアクセスを提供します。 Tk 8.5 が無い環境で Python がコンパイルされていた場合でも、 Tile がインストールされていればこのモジュールはそれを使おうとします。しかし、 X11 上のフォントのアンチエイリアスや透過ウィンドウ (X11 ではコンポジションウィンドウマネージャが必要です) などの新しい Tk が提供している機能は使えません。

ttk の基本的なアイディアは、拡張可能性のためにウィジェットの動作を実装するコードと見た目を記述するコードを分離することです。

参考:

[Tk Widget Styling Support](#) Tk のテーマサポートの立ち上げのドキュメント

24.2.1 Ttk を使う

Ttk を使い始めるために、モジュールをインポートします:

```
import ttk
```

しかしこのようなコードでは

```
from Tkinter import *
```

このように使いたいことがあるかもしれません

```
from Tkinter import *
from ttk import *
```

このように書くと、いくつかの *ttk* ウィジェット (Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale, Scrollbar) は自動的に Tk ウィジェットを置き換えます。

これにはプラットフォームをまたいでより良い見た目を得られるという、直接的な利益がありますが、ウィジェットは完全な互換性を持っているわけではないことに注意してください。一番の違いは "fg" や "bg" やその他のスタイルに関するウィジェットのオプションが Tk ウィジェットから無くなっていることです。同じ (もしくはより良い) 見た目にするためには *ttk.Style* を使ってください。

参考:

[Converting existing applications to use the Tile widgets](#) Tcl において、アプリケーションを新しいウィジェットに移行するときに出てくる典型的な差異について書かれているテキスト

24.2.2 Ttk ウィジェット

Ttk には 17 のウィジェットがあり、そのうち 11 は Tkinter に既にあるものです: Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale, Scrollbar。新しい 6 つのウィジェットクラスは次のものです: *Combobox*, *Notebook*, *Progressbar*, *Separator*, *Sizegrip*, *Treeview*。これらのクラスは全て *Widget* の subclasses です。

上にも書いた通り、スタイルの記述コードと同様に見た目も変わっていることに気付くでしょう。それを見せるために、非常に簡単な例を以下に示します。

Tk のコード

```
l1 = Tkinter.Label(text="Test", fg="black", bg="white")
l2 = Tkinter.Label(text="Test", fg="black", bg="white")
```

それに相当する Ttk のコード

```
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")
```

(次のページに続く)

(前のページからの続き)

```
l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

TtkStyling についての情報は *Style* クラスの文書を読んでください。

24.2.3 ウィジェット

ttk.Widget は Tk のテーマ付きウィジェットがサポートしている標準のオプションやメソッドを定義するもので、これを直接インスタンス化するものではありません。

標準オプション

全ての *ttk* ウィジェットは以下のオプションを受け付けます:

オプション	説明
class	ウィンドウクラスを指定します。このクラスはオプションデータベースにウィンドウの他のオプションについて問い合わせを行うときに使われ、これによりウィンドウのデフォルトのバインドタグを決定したり、ウィジェットのデフォルトのレイアウトやスタイルを選択します。これは読み取り専用のオプションでウィンドウが作られるときにのみ指定できます。
cursor	このウィジェットで使うマウスカーソルを指定します。空文字列 (デフォルト) が設定されている場合は、カーソルは親ウィジェットのものを引き継ぎます。
takefocus	キーボードによる移動のときにウィンドウがフォーカスを受け入れるかを決定します。0、1、空文字列のいずれかを返します。0 の場合、キーボードによる移動でそのウィンドウは常にスキップされます。1 の場合、そのウィンドウが表示されているときに限り入力フォーカスを受け入れます。空文字列は、移動スクリプトによってウィンドウにフォーカスを当てるかどうかが決まることを意味します。
style	独自のウィジェットスタイルを指定するのに使われます。

スクロール可能ウィジェットのオプション

以下のオプションはスクロールバーで操作されるウィジェットが持っているオプションです。

option	description
xscrollcommand	水平方向のスクロールバーとのやり取りに使われます。 ウィジェットのウィンドウが再描画されたとき、ウィジェットは <code>scrollcommand</code> に基いて Tcl コマンドを生成します。 通常このオプションにはあるスクロールバーの <code>Scrollbar.set()</code> メソッドが設定されます。こうすると、ウィンドウの見た目が変わったときにスクロールバーの状態も更新されます。
yscrollcommand	垂直方向のスクロールバーとのやり取りに使われます。詳しいことは、上記を参照してください。

ラベルオプション

以下のオプションはラベルやボタンやボタンに類似したウィジェットが持っているオプションです。

option	description
text	ウィジェットに表示される文字列を指定します。
textvariable	text オプションの代わりに使う値の変数名を指定します。
underline	このオプションを設定すると、文字列の中で下線を引く文字のインデックス (0 基点) を指定します。下線が引かれた文字はショートカットとして使われます。
image	表示する画像を指定します。これは 1 つ以上の要素を持つリストです。先頭の要素はデフォルトの画像名です。残りの要素は <code>Style.map()</code> で定義されているような状態名と値のペアの並びで、ウィジェットがある状態、もしくはある状態の組み合わせにいたときに使用する別の画像を指定します。このリストにある全ての画像は同じサイズでなければなりません。
compound	text オプションと image オプションが両方とも指定されていた場合に、テキストに対して画像をどう配置するかを指定します。 <ul style="list-style-type: none"> text: テキストのみ表示する image: 画像のみ表示する top, bottom, left, right: それぞれ画像をテキストの上、下、左、右に配置する。 none: デフォルト。もしあれば画像を表示し、そうでなければテキストを表示する。
width	0 より大きい場合、テキストラベルを作成するのにどれくらいのスペースを使うかを文字の幅で指定します。0 より小さい場合、最小の幅が指定されます。0 もしくは無指定の場合、テキストラベルに対して自然な幅が使われます。

互換性オプション

option	description
state	”normal” か ”disabled” に設定され、”disabled” 状態のビットをコントロールします。これは書き込み専用のオプションです: これを設定するとウィジェットの状態を変更できますが、 <code>Widget.state()</code> メソッドはこのオプションに影響を及ぼしません。

ウィジェットの状態

ウィジェットの状態は独立した状態フラグのビットマップです。

フラグ	description
active	マウスカーソルがウィジェットの上にあり、マウスのボタンをクリックすることで何らかの動作をさせられます。
disabled	プログラムによってウィジェットは無効化されています。
focus	ウィジェットにキーボードフォーカスがあります。
pressed	ウィジェットは押されています。
selected	チェックボタンやラジオボタンのようなウィジェットでの ”オン” や ”チェック有” や ”選択中” に当たります。
background	Windows と Mac には ”アクティブな” もしくは最前面のウィンドウという概念があります。背面のウィンドウにあるウィジェットには <i>background</i> 状態が設定され、最前面のウィンドウにあるウィジェットでは解除されます。
read-only	ウィジェットはユーザからの変更を受け付けません。
alternate	ウィジェット特有の切り替え表示になっています。
invalid	ウィジェットの値が不正です。

状態仕様は状態名の並びになっていて、状態名の先頭にはビットがオフになっていることを示す感嘆符が付くことがあります。

ttk.Widget

以下に書かれているメソッドに加えて、`ttk.Widget` クラスは `Tkinter.Widget.cget()` メソッドと `Tkinter.Widget.configure()` メソッドをサポートしています。

```
class ttk.Widget
```

```
    identify(x, y)
```

`x y` の位置にある要素の名前、もしくはその位置に要素が無ければ空文字列を返します。

`x` と `y` はウィジェットに対するピクセル単位の座標です。

```
    instate(statespec, callback=None, *args, **kw)
```

ウィジェットの状態をチェックします。コールバックが指定されていない場合、ウィジェットの状態が `statespec` に一致していれば `True`、そうでなければ `False` を返します。コールバックが指定されていて、ウィジェットの状態が `statespec` に一致している場合、引数に `args` を指定してそのコールバックを呼び出します。

```
    state([statespec=None])
```

ウィジェットの状態を変更したり、取得したりします。 `statespec` が指定されている場合、それに応じてウィジェットの状態を設定し、どのフラグが変更されたかを示す新しい `statespec` を返します。 `statespec` が指定されていない場合、現在の状態フラグを返します。

通常 `statespec` はリストもしくはタプルです。

24.2.4 コンボボックス

`ttk.Combobox` ウィジェットはテキストフィールドと値のポップダウンリストを結び付けます。このウィジェットは `Entry` の subclasses です。

`Widget` から継承したメソッド (`Widget.cget()`, `Widget.configure()`, `Widget.identify()`, `Widget.instate()`, `Widget.state()`) と `Entry` から継承したメソッド (`Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.selection()`, `Entry.xview()`) に加え、このクラスには `ttk.Combobox` で説明するメソッドがあります。

オプション

このウィジェットは以下のオプションを受け付けます:

option	description
export-selection	真偽値を取る。設定されている場合、ウィジェットの選択はウィンドウマネージャの選択とリンクしています。(例えば、 <code>Misc.selection_get()</code> を実行することで得られます。)
justify	ウィジェットの中でテキストをどう配置するかを指定します。 "left", "center", "right" のうちのどれか 1 つです。
height	ポップダウンリストの高さを行数で指定します。
post-command	コンボボックスの値を表示する直前に呼び出される、(<code>Misc.register()</code> など登録した) スクリプトです。どの値を表示するかについても指定できます。
state	"normal", "readonly", "disabled" のどれか 1 つです。 "readonly" 状態では、直接入力値を編集することはできず、ユーザはドロップダウンリストから値を 1 つ選ぶことしかできません。 "normal" 状態では、テキストフィールドは直接編集できます。 "disabled" 状態では、コンボボックスは一切反応しません。
textvariable	コンボボックスの値とリンクさせる変数名を指定します。その変数の値が変更されたとき、ウィジェットの値は更新されます。ウィジェットの値が更新されたときも同様です。 <code>Tkinter.StringVar</code> を参照してください。
values	ドロップダウンリストに表示する値のリストを指定します。
width	入力ウィンドウに必要な幅をウィジェットのフォントの平均的なサイズの文字で測った、文字数を指定します。

仮想イベント

コンボボックスウィジェットは、ユーザが値のリストから 1 つ選んだときに仮想イベント `<<ComboboxSelected>>` を生成します。

ttk.Combobox

```
class ttk.Combobox
```

```
current ([newindex=None])
```

`newindex` が指定されている場合、コンボボックスの値がドロップダウンリストの `newindex` の位置にある値に設定されます。そうでない場合、現在の値のインデックスを、もしくは現在の値がリストに含まれていないなら -1 を返します。

```
get ()
```

コンボボックスの現在の値を返します。

```
set (value)
```

コンボボックスの値を *value* に設定します。

24.2.5 ノートブック

Tk ノートブックウィジェットは複数のウィンドウを管理し、同時に 1 つのウィンドウを表示します。それぞれの子ウィンドウはタブの関連付けられていて、ユーザはそれを選択して表示されているウィンドウを切り替えます。

オプション

このウィジェットは以下の固有のオプションを受け付けます:

option	description
height	0 より大きな値が設定されている場合、(内部のパディングやタブを含まない) ペイン領域に必要な高さを指定します。設定されていない場合、全てのペインの高さの最大値が使われます。
padding	ノートブックの外周に付け足す追加の領域の量を指定します。パディングは最大 4 個の長さ指定のリストです: 左、上、右、下の順で指定します。4 個より少ない場合、デフォルトで下は上と、右は左と、上は左と同じ値が、それぞれ使われます。
width	0 より大きな値が指定されている場合、(内部のパディングを含まない) ペイン領域に必要な幅を指定します。設定されていない場合、全てのペインの幅の最大値が使われます。

タブオプション

タブ用のオプションもあります:

option	description
state	"normal", "disabled", "hidden" のうちどれか 1 つです。"disabled" の場合、タブは選択することができません。"hidden" の場合、タブは表示されません。
sticky	ペイン領域の中に子ウィンドウがどう置かれるかを指定します。指定する値は "n", "s", "e", "w" からなる 0 文字以上の文字列です。配置マネージャの <code>grid()</code> と同様に、それぞれの文字は子ウィンドウが (北、南、東、西の) どの辺に対して追従するかに対応しています。
padding	ノートブックとこのペインの間に付け足す追加の領域の量を指定します。文法はこのウィジェットの <code>padding</code> オプションと同じです。
text	タブに表示するテキストを指定します。
image	タブに表示する画像を指定します。 <i>Widget</i> のオプション <code>image</code> の説明を参照してください。
compound	<code>text</code> オプションと <code>image</code> オプションが両方指定されているときにテキストに対して画像をどう表示するかを指定します。指定する値については ラベルオプション を参照してください。
underline	テキスト中の下線を引く文字のインデックス (0 基点) を指定します。 <i>Notebook</i> . <code>enable_traversal()</code> が呼ばれていた場合、下線が引かれた文字はショートカットとして使われます。

タブ識別子

`ttk.Notebook` のいくつかのメソッドにある `tab_id` は以下の形式を取ります:

- 0 からタブの数の間の整数。
- 子ウィンドウの名前。
- タブを指し示す "@x,y" という形式の位置指定。
- 現在選択されているタブを指し示すリテラル文字列 "current"。
- タブ数を返すリテラル文字列 "end" (*Notebook.index()* でのみ有効)。

仮想イベント

このウィジェットは新しいタブが選択された後に仮想イベント `<<NotebookTabChanged>>` を生成します。

ttk.Notebook

```
class ttk.Notebook
```

```
add(child, **kw)
```

ノートブックに新しいタブを追加します。

ウィンドウが現在ノートブックによって管理されているが隠れている場合、以前の位置に復元します。

利用可能なオプションのリストについては *Tab Options* を参照してください。

forget (*tab_id*)

tab_id で指定されたタブを削除します。関連付けられていたウィンドウは切り離され、管理対象でなくなります。

hide (*tab_id*)

tab_id で指定されたタブを隠します。

タブは表示されませんが、関連付いているウィンドウはノートブックによって保持されていて、その設定も記憶されています。隠れたタブは *add()* コマンドで復元できます。

identify (*x*, *y*)

x y の位置にあるタブの名前を、そこにタブが無ければ空文字列を返します。

index (*tab_id*)

tab_id で指定されたタブのインデックスを、*tab_id* が文字列の "end" だった場合はタブの総数を返します。

insert (*pos*, *child*, ***kw*)

指定された位置にペインを挿入します。

pos は文字列の "end" か整数のインデックスか管理されている子ウィンドウの名前です。 *child* が既にノートブックの管理対象だった場合、指定された場所に移動させます。

利用可能なオプションのリストについては *Tab Options* を参照してください。

select ([*tab_id*])

指定された *tab_id* を選択します。

関連付いている子ウィンドウは表示され、直前に選択されていたウィンドウは (もし異なれば) 表示されなくなります。 *tab_id* が指定されていない場合は、現在選択されているペインのウィジェット名を返します。

tab (*tab_id*, *option=None*, ***kw*)

指定された *tab_id* のオプションを問い合わせたり、変更したりします。

kw が与えられなかった場合、タブのオプション値の辞書を返します。 *option* が指定されていた場合、その *option* の値を返します。それ以外の場合は、オプションに対応する値が設定されます。

tabs ()

ノートブックに管理されているウィンドウのリストを返します。

enable_traversal ()

このノートブックを含む最上位にあるウィンドウでのキーボード移動を可能にします。

これによりノートブックを含んだ最上位にあるウィンドウに対し、以下のキーバインディングが追加されます:

- Control-Tab: 現在選択されているタブの 1 つ次のタブを選択します。
- Shift-Control-Tab: 現在選択されているタブの 1 つ前のタブを選択します。

- Alt-K: *K* があるタブの (下線が引かれた) ショートカットキーだとして、そのタブを選択します。

ネストしたノートブックも含め、1つのウィンドウの最上位にある複数のノートブックのキーボード移動が可能になることもあります。しかしノートブック上の移動は、全てのペインが同じノートブックを親としているときのみ正しく動作します。

24.2.6 プログレスバー

`ttk.Progressbar` ウィジェットは長く走る処理の状態を表示します。このウィジェットは2つのモードで動作します: 決定的モードでは、全ての処理の総量のうち完了した量を表示します。非決定的モードでは、今何か処理が行われていることをユーザに示します。

オプション

このウィジェットは以下の固有のオプションを受け付けます:

option	description
orient	"horizontal" もしくは "vertical" のいずれかです。プログレスバーの方向を指定します。
length	プログレスバーの長さを指定します。(水平方向の場合は幅、垂直方向の場合は高さです)
mode	"determinate" か "indeterminate" のいずれかです。
maximum	最大値を数値で指定します。デフォルトは 100 です。
value	プログレスバーの現在値です。決定的 ("determinate") モードでは、完了した処理の量を表示します。非決定的 ("indeterminate") モードでは、 <i>maximum</i> を法として解釈され、値が <i>maximum</i> に達したときにプログレスバーは 1 "サイクル" を完了したことになります。
variable	value オプションとリンクさせる変数名です。指定されている場合、変数の値が変更されるとプログレスバーの値は自動的にその値に設定されます。
phase	読み取り専用のオプションです。このウィジェットの値が 0 より大きく、かつ決定的モードでは最大値より小さいときに、ウィジェットが定期的にこのオプションの値を増加させます。このオプションは現在の画面テーマが追加のアニメーション効果を出すのに使います。

ttk.Progressbar

`class` `ttk.Progressbar`

`start` (`[interval]`)

自動増加モードを開始します: *interval* ミリ秒ごとに `Progressbar.step()` を繰り返し呼び出

すタイマーイベントを設定します。引数で指定しない場合は、*interval* はデフォルトで 50 ミリ秒になります。

step(*[amount]*)

プログレスバーの値を *amount* だけ増加させます。

引数で指定しない場合は、*amount* はデフォルトで 1.0 になります。

stop()

自動増加モードを停止します: このプログレスバーの *Progressbar.start()* で開始された繰り返しのタイマーイベントを全てキャンセルします。

24.2.7 セパレータ

`ttk.Separator` ウィジェットは水平もしくは垂直のセパレータを表示します。

`ttk.Widget` から継承したメソッド以外にメソッドを持ちません。

オプション

このウィジェットは以下の固有のオプションを受け付けます:

option	description
orient	"horizontal" か "vertical" のいずれかです。セパレータの方向を指定します。

24.2.8 サイズグリップ

(グローボックスとしても知られる) `ttk.Sizegrip` ウィジェットは、押してつまみ部分をドラッグすることで最上位のウィンドウのサイズを変更できます。

このウィジェットは `ttk.Widget` から継承したもの以外のオプションとメソッドを持ちません。

プラットフォーム固有のメモ

- Mac OS X では、最上位のウィンドウにはデフォルトで組み込みのサイズグリップが含まれています。組み込みのグリップが `Sizegrip` を隠してしまうので、`Sizegrip` を追加するのは無害です。

バグ

- 最上位のウィンドウの位置がスクリーンに対して右や下に指定されている場合 (などなど....)、`Sizegrip` ウィジェットはウィンドウのサイズ変更をしません。
- このウィジェットは "南東" 方向のサイズ変更しかサポートしていません。

24.2.9 ツリービュー

`ttk.Treeview` ウィジェットは階層のある要素 (アイテム) の集まりを表示します。それぞれの要素はテキストラベル、オプションの画像、オプションのデータのリストを持っています。データはラベルの後に続くカラムに表示されます。

データが表示される順序はウィジェットの `displaycolumns` オプションで制御されます。ツリーウィジェットはカラムヘッダを表示することもできます。カラムには数字もしくはウィジェットの `columns` オプションにある名前でアクセスできます。 *Column Identifiers* を参照してください。

それぞれの要素は一意的な名前で識別されます。要素の作成時に識別子が与えられなかった場合、ウィジェットが要素の識別子を生成します。このウィジェットには `{}` という名前の特別なルート要素があります。ルート要素自身は表示されません; その子要素たちが階層の最上位に現れます。

それぞれの要素はタグのリストも持っていて、イベントバインディングと個別の要素を関連付け、要素の見た目を管理するのに使えます。

ツリービューウィジェットは水平方向と垂直方向のスクロールをサポートしていて、 *Scrollable Widget Options* に記述してあるオプションと `Treeview.xview()` メソッドおよび `Treeview.yview()` メソッドが使えます。

オプション

このウィジェットは以下の固有のオプションを受け付けます:

option	description
columns	カラム数とその名前を指定するカラム識別子のリストです。
displaycolumns	どのデータカラムをどの順序で表示するかを指定する、(名前もしくは整数のインデックスの) カラム識別子のリストか、文字列 ”#all” です。
height	表示する行数を指定します。メモ: 表示に必要な幅はカラム幅の合計から決定されます。
padding	ウィジェットの内部のパディングのサイズを指定します。パディングは最大 4 個の長さ指定のリストです。
selectmode	組み込みのクラスバインディングが選択状態をどう管理するかを指定します。設定する値は ”extended”, ”browse”, ”none” のどれか 1 つです。 ”extended” に設定した場合 (デフォルト)、複数の要素が選択できます。 ”browse” に設定した場合、同時に 1 つの要素しか選択できません。 ”none” に設定した場合、選択を変更することはできません。 このオプションの値によらず、アプリケーションのコードと関連付けられているタグからは好きなように選択状態を設定できます。
show	ツリーのどの要素を表示するかを指定する、以下にある値を 0 個以上含むリストです。 <ul style="list-style-type: none"> • tree: カラム #0 にツリーのラベルを表示します。 • headings: ヘッダ行を表示します。 デフォルトは ”tree headings”、つまり全ての要素を表示します。 メモ: show= ”tree” が指定されていない場合でも、カラム #0 は常にツリーカラムを参照します。

要素オプション

以下の要素オプションは、ウィジェットの insert コマンドと item コマンドで要素に対して指定できます。

option	description
text	アイテムに表示するテキストラベルです。
image	ラベルの左に表示される Tk 画像です。
values	要素に関連付けられている値のリストです。 それぞれの要素はウィジェットの columns オプションと同じ数の値を持たなければいけません。 columns オプションより少ない場合、残りの値は空として扱われます。 columns オプションより多い場合、余計な値は無視されます。
open	要素の子供を表示するか隠すかを指示する真偽値です。
tags	この要素に関連付けられているタグのリストです。

タグオプション

以下のオプションはタグに対して設定できます:

option	description
foreground	テキストの色を指定します。
background	セルや要素の背景色を指定します。
font	テキストを描画するときに使うフォントを指定します。
image	要素の image オプションが空だった場合に使用する画像を指定します。

カラム識別子

カラム識別子は以下のいずれかの形式を取ります:

- columns オプションのリストにある名前。
- n 番目のデータカラムを指し示す整数 n。
- n を整数として n 番目の表示されているカラムを指し示す #n という形式の文字列。

注釈:

- 要素のオプション値は実際に格納されている順序とは違った順序で表示されることがあります。
- show="tree" が指定されていない場合でも、カラム #0 は常にツリーカラムを指しています。

データカラムを指す数字は、要素の values オプションのリストのインデックスです; 表示カラムを指す数字は、値が表示されているツリーのカラム番号です。ツリーラベルはカラム #0 に表示されます。displaycolumns オプションが設定されていない場合は、n 番目のデータカラムはカラム #n+1 に表示されます。再度言っておくと、カラム #0 は常にツリーカラムを指します。

仮想イベント

ツリービューは以下の仮想イベントを生成します。

イベント	description
<<TreeviewSelect>>	選択状態が変更されたときに生成されます。
<<TreeviewOpen>>	フォーカスが当たっている要素に open=True が設定される直前に生成されます。
<<TreeviewClose>>	フォーカスが当たっている要素に open=False が設定された直後に生成されます。

`Treeview.focus()` メソッドと `Treeview.selection()` メソッドは変更を受けた要素を判別するのに使えます。

ttk.Treeview**class** `ttk.Treeview`**bbox** (*item*, *column=None*)

(ツリービューウィジェットのウィンドウを基準として) 指定された *item* のバウンディングボックス情報を (x 座標, y 座標, 幅, 高さ) の形式で返します。

column が指定されている場合は、セルのバウンディングボックスを返します。(例えば、閉じた状態の要素の子供であったり、枠外にスクロールされていて) *item* が見えなくなっている場合は、空文字列が返されます。

get_children ([*item*])

item の子要素のリストを返します。

item が指定されていなかった場合は、ルート要素の子供が返されます。

set_children (*item*, **newchildren*)

item の子要素を *newchildren* で置き換えます。

item にいる子供のうち *newchildren* にないものはツリーから切り離されます。 *newchildren* にあるどの要素も *item* の祖先であってはいけません。 *newchildren* を指定しなかった場合は、 *item* の子要素が全て切り離されることに注意してください。

column (*column*, *option=None*, ***kw*)

指定した *column* のオプションを問い合わせたり、変更したりします。

kw が与えられなかった場合は、カラムのオプション値の辞書が返されます。 *option* が指定されていた場合は、その *option* の値が返されます。それ以外の場合は、オプションに値を設定します。

設定できるオプションとその値は次の通りです:

- **id** カラム名を返します。これは読み取り専用のオプションです。
- **anchor**: 標準の Tk anchor の値 このカラムでセルに対してテキストをどう配置するかを指定します。
- **minwidth**: 幅 カラムの最小幅をピクセル単位で表したものです。ツリービューウィジェットは、ウィジェットのサイズが変更されたりカラムをユーザがドラッグして移動させたりしたときに、このオプションで指定した幅より狭くすることはありません。
- **stretch**: True もしくは False ウィジェットがサイズ変更されたとき、カラムの幅をそれに合わせるかどうかを指定します。
- **width**: 幅 カラムの幅をピクセル単位で表したものです。

ツリーカラムの設定を行うには、 `column = "#0"` を付けてこのメソッドを呼び出してください。

delete (**items*)

指定された *items* とその子孫たち全てを削除します。

ルート要素は削除されません。

detach (*items)

指定された *items* を全てツリーから切り離します。

その要素と子孫たちは依然として存在していて、ツリーの別の場所に再度挿入することができますが、隠された状態になり表示はされません。

ルート要素は切り離されません。

exists (item)

指定された *item* がツリーの中にあれば `True` を返します。

focus ([item=None])

item が指定されていた場合は、*item* にフォーカスを当てます。そうでない場合は、現在フォーカスが当たっている要素が、どの要素にもフォーカスが当たっていない場合は `None` が返されます。

heading (column, option=None, **kw)

指定された *column* の heading のオプションを問い合わせたり、変更したりします。

kw が与えられなかった場合は、見出しのオプション値の辞書が返されます。*option* が指定されている場合は、*option* の値が返されます。それ以外の場合は、オプションに値を設定します。

設定できるオプションとその値は次の通りです:

- **text**: テキスト カラムの見出しに表示するテキスト。
- **image**: 画像名 カラムの見出しの右に表示する画像を指定します。
- **anchor**: anchor 見出しのテキストをどう配置するかを指定します。標準の Tk anchor の値です。
- **command**: コールバック 見出しラベルがクリックされたときに実行されるコールバックです。

ツリーカラムの見出しの設定を行うには、`column = "#0"` を付けてこのメソッドを呼び出してください。

identify (component, x, y)

x y で与えられた場所にある指定された *component* の説明を返します。その場所に指定された *component* が無い場合は空文字列を返します。(訳注: *component* には "region", "item", "column", "row", "element" が指定でき、それぞれ "cell", "heading" などの場所の名前、要素の識別子、*#n* という形式のカラム名、その行にある要素の識別子、"text", "padding" などの画面構成要素の名前を返します。)

identify_row (y)

y 座標が *y* の位置にある要素の識別子を返します。

identify_column (x)

x 座標が *x* の位置にあるセルのデータカラムの識別子を返します。

ツリーカラムは `#0` という識別子を持ちます。

identify_region (*x*, *y*)

以下のうち 1 つを返します:

region	意味
heading	ツリーの見出し領域
separator	2 つのカラム見出しの間のスペース
tree	ツリーの領域
cell	データセル

使用可能バージョン: Tk 8.6

identify_element (*x*, *y*)

x y の位置にある画面構成要素の名前を返します。

使用可能バージョン: Tk 8.6

index (*item*)

親要素の子要素リストの中での *item* のインデックスを返します。

insert (*parent*, *index*, *iid=None*, ***kw*)

新しい要素を作り、その要素の識別子を返します。

parent は親となる要素の識別子で、空文字列にすると新しい要素を最上位に作成します。 *index* は整数もしくは "end" という値で、それによって親要素の子要素リストのどこに新しい要素を挿入するかを指定します。 *index* が 0 以下だった場合は、新しい要素は先頭に挿入されます; *index* が現在の子要素の数以上だった場合は末尾に挿入されます。 *iid* が指定された場合は、要素の識別子として使われます; *iid* はまだツリーに存在していないものに限りです。それ以外の場合は、一意な識別子が生成されます。

使用できるオプションのリストについては [Item Options](#) を参照してください。

item (*item* [, *option* [, ***kw*]])

指定された *item* のオプションを問い合わせたり、変更したりします。

オプションが与えられなかった場合は、要素のオプションと値が辞書の形で返されます。 *option* が指定された場合は、そのオプションの値が返されます。それ以外の場合は、*kw* で与えられたようにオプションに値が設定されます。

move (*item*, *parent*, *index*)

item を *parent* の子要素リストの *index* の位置に移動します。

要素を自身の子孫の下に移動させるのは許されていません。 *index* が 0 以下の場合、*item* は先頭に移動されます; 子要素の数以上だった場合、末尾に移動されます。 *item* が切り離された状態の場合は、再度取り付けられます。

next (*item*)

item の 1 つ下の兄弟の識別子を、*item* が親にとって一番下の子供だった場合 " " を返します。

parent (*item*)

item の親の識別子を、*item* が階層の最上位にいた場合 ” を返します。

prev (*item*)

item の 1 つ上の兄弟の識別子を、*item* が親にとって一番上の子供だった場合 ” を返します。

reattach (*item*, *parent*, *index*)

`Treeview.move()` のエイリアスです。

see (*item*)

item を見える状態にします。

item の全ての子孫の open オプションを `True` にし、必要であれば *item* がツリーの見える範囲に来るようにウィジェットをスクロールさせます。

selection ([*selop*=None[, *items*=None]])

selop が指定されなかった場合は、選択されている要素を返します。そうでなければ、以下の選択メソッドに従って動作します。(訳注: *selop* には "set", "add", "remove", "toggle" のうち 1 つが指定できます。 *items* にはそれぞれのメソッドの引数を指定します。)

selection.set (*items*)

新しく選択状態の要素が *items* になります。

selection.add (*items*)

選択状態の要素として *items* を追加します。

selection.remove (*items*)

選択状態の要素から *items* を取り除きます。

selection.toggle (*items*)

items のそれぞれの要素の選択状態を入れ替えます。

set (*item*, *column*=None, *value*=None)

1 引数で呼び出された場合、指定された *item* のカラムと値のペアからなる辞書を返します。2 引数で呼び出された場合、指定された *column* の現在の値を返します。3 引数で呼び出された場合、与えられた *item* の *column* を指定された値 *value* に設定します。

tag.bind (*tagname*, *sequence*=None, *callback*=None)

与えられたイベント *sequence* 用のコールバックをタグ *tagname* にバインドします。イベントが要素に渡ってきたときに、要素の tags オプションのそれぞれのコールバックが呼び出されます。

tag.configure (*tagname*, *option*=None, ***kw*)

指定された *tagname* のオプションを問い合わせたり、変更したりします。

kw が与えられなかった場合、*tagname* のオプション設定を辞書の形で返します。*option* が指定された場合、指定された *tagname* の *option* の値を返します。それ以外の場合、与えられた *tagname* のオプションに値を設定します。

tag.has (*tagname* [, *item*])

item が指定されていた場合、指定された *item* が与えられた *tagname* を持っているかどうかに従っ

て 1 または 0 が返されます。そうでない場合、指定されたタグを持つ全ての要素のリストを返します。

使用可能バージョン: Tk 8.6

xview (*args)

ツリービューの水平方向の位置を問い合わせたり、変更したりします。

yview (*args)

ツリービューの垂直方向の位置を問い合わせたり、変更したりします。

24.2.10 Ttk スタイル

`ttk` のそれぞれのウィジェットにはスタイルが関連付けられていて、それと動的もしくはデフォルトで設定される要素のオプションによってウィジェットを構成する要素とその配置を指定します。デフォルトではスタイル名はウィジェットのクラス名と同じですが、ウィジェットの `style` オプションで上書きすることができます。ウィジェットのクラス名が分からない場合は、`Misc.winfo_class()` (`somewidget.winfo_class()`) メソッドを使ってください。

参考:

[Tcl'2004 conference のプレゼンテーション](#) この文書ではテーマエンジンがどう動くかを説明しています

class `ttk.Style`

このクラスはスタイルデータベースを操作するために使われます。

configure (style, query_opt=None, **kw)

style の指定されたオプションのデフォルト値を問い合わせたり、設定したりします。

kw のそれぞれのキーはオプション名で値はそのオプションの値の文字列です。

例えば、全てのデフォルトのボタンをパディングのある平らな見た目にし背景の色を変更するには以下のようにします

```
import ttk
import Tkinter

root = Tkinter.Tk()

ttk.Style().configure("TButton", padding=6, relief="flat",
    background="#ccc")

btn = ttk.Button(text="Sample")
btn.pack()

root.mainloop()
```

map (style, query_opt=None, **kw)

style の指定されたオプションの動的な値を問い合わせたり、設定したりします。

kw のそれぞれのキーはオプション名で、値はタプルやリストや何か他の好きなものでグループ化された状態仕様 (*statespec*) を要素とするリストやタプルです。状態仕様は 1 つもしくは複数の状態と値の組み合わせです。

以下に例を示します:

```
import Tkinter
import ttk

root = Tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=[('pressed', 'red'), ('active', 'blue')],
    background=[('pressed', '!disabled', 'black'), ('active', 'white')]
)

colored_btn = ttk.Button(text="Test", style="C.TButton").pack()

root.mainloop()
```

あるオプションに対する状態と値の組 (*states*, *value*) の並び順はスタイルに影響を与えることに注意してください。例えば、*foreground* オプションの順序を `[('active', 'blue'), ('pressed', 'red')]` に変更した場合、ウィジェットがアクティブもしくは押された状態のとき前面が青くなります。

lookup (*style*, *option*, *state=None*, *default=None*)

style の指定された *option* の値を返します。

state を指定する場合は、1 つ以上の状態名の並びである必要があります。 *default* 引数が指定されていた場合は、オプション指定が見付からなかったときに代わりに返される値として使われます。

デフォルトでボタンがどのフォントを使うかを調べるには、以下のように実行します

```
import ttk

print ttk.Style().lookup("TButton", "font")
```

layout (*style*, *layoutspect=None*)

与えられた *style* でのウィジェットのレイアウトを定義します。 *layoutspect* が省略されていた場合は、与えられたスタイルのレイアウト仕様を返します。

layoutspect を指定する場合は、リストもしくは (文字列を除いた) 何か他のシーケンス型である必要があります。それぞれの要素はタプルで、レイアウト名を 1 番目の要素とし、2 番目の要素は [Layouts](#) で説明されているフォーマットである必要があります。

フォーマットを理解するために以下の例を見てください (何かを使い易くするための例ではありません):

```
import ttk
import Tkinter

root = Tkinter.Tk()

style = ttk.Style()
style.layout("TMenubutton", [
    ("Menubutton.background", None),
    ("Menubutton.button", {"children":
        [ ("Menubutton.focus", {"children":
            [ ("Menubutton.padding", {"children":
                [ ("Menubutton.label", {"side": "left", "expand": 1})]
            })]
        })]
    })],
])

mbtn = ttk.Menubutton(text='Text')
mbtn.pack()
root.mainloop()
```

element.create (*elementname*, *etype*, **args*, ***kw*)

与えられた *etype* ("image", "from", "vsapi" のいずれか) の現在のテーマに新しい要素を作成します。最後の "vsapi" は Windows XP と Vista の Tk 8.6a のみで使用可能でここでは説明しません。

"image" が使われた場合、*args* はデフォルトの画像名の後ろに状態仕様と値のペア (これが画像仕様です) を並べたものである必要があります。 *kw* には以下のオプションが指定できます:

- **border=padding** padding は 4 個以下の整数のリストで、それぞれ左、上、右、下の縁の幅を指定します。
- **height=height** 要素の最小の高さを指定します。0 より小さい場合は、画像の高さをデフォルトとして使用します。
- **padding=padding** 要素の内部のパディングを指定します。指定されない場合は、border の値がデフォルトとして使われます。
- **sticky=spec** 1 つ外側の枠に対し画像をどう配置するかを指定します。spec は "n", "s", "w", "e" の文字を 0 個以上含みます。
- **width=width** 要素の最小の幅を指定します。0 より小さい場合は、画像の幅をデフォルトとして使用します。

etype の値として "from" が使われた場合は、`element.create()` が現在の要素を複製します。*args* は要素の複製元のテーマの名前と、オプションで複製する要素を含んでいる必要があります。複製元の要素が指定されていなかった場合、空要素が使用され、*kw* は破棄されます。

element.names ()

現在のテーマに定義されている要素のリストを返します。

element.options (*elementname*)

elementname のオプションのリストを返します。

theme.create (*themename*, *parent=None*, *settings=None*)

新しいテーマを作成します。

themename が既に存在していた場合はエラーになります。 *parent* が指定されていた場合は、新しいテーマは親テーマからスタイルや要素やレイアウトを継承します。 *settings* が指定された場合は、 *theme.settings()* で使われるのと同じ形式である必要があります。

theme.settings (*themename*, *settings*)

一時的に現在のテーマを *themename* に設定し、指定された *settings* を適用した後、元のテーマを復元します。

settings のそれぞれのキーはスタイル名で値はさらに 'configure', 'map', 'layout', 'element create' をキーとして持ち、その値はそれぞれ *Style.configure()*, *Style.map()*, *Style.layout()*, *Style.element_create()* メソッドで指定するのと同じ形式である必要があります。

例として、コンボボックスの default テーマを少し変更してみましょう

```
import ttk
import Tkinter

root = Tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
    "TCombobox": {
        "configure": {"padding": 5},
        "map": {
            "background": [("active", "green2"),
                           ("!disabled", "green4")],
            "fieldbackground": [("!disabled", "green3")],
            "foreground": [("focus", "OliveDrab1"),
                           ("!disabled", "OliveDrab2")]
        }
    }
})

combo = ttk.Combobox().pack()

root.mainloop()
```

theme.names ()

全ての既存のテーマのリストを返します。

theme.use ([*themename*])

themename が与えられなかった場合は、現在使用中のテーマ名を返します。そうでない場合は、現在のテーマを *themename* に設定し、全てのウィジェットを再描画し、 <<ThemeChanged>> イベントを発生させます。

レイアウト

レイアウトはオプションを取らない場合はただの `None` にできますし、そうでない場合は要素をどう配置するかを指定するオプションの辞書になります。レイアウト機構は単純化したジオメトリマネージャを使っています: 最初に空間が与えられ、それぞれの要素に分割された空間が配分されます。設定できるオプションと値は次の通りです:

- `side`: 辺の名前 要素を空間のどちら側に配置するかを指定します; `top`, `right`, `bottom`, `left` のどれか 1 つです。省略された場合は、要素は空間全体を占めます。
- `sticky`: `n`, `s`, `w`, `e` から 0 個以上 配分された空間の内部に要素をどう配置するかを指定します。
- `unit`: 0 か 1 1 に設定されると、`Widget.identify()` などには要素とその子で単一の要素として扱われます。これは、グリップのついたスクロールバーサムのようなものに使われます。
- `children`: [内部レイアウト...] 要素の内部に配置する要素のリストを指定します。リストのそれぞれの要素はタプル (もしくは他のシーケンス型) で、その 1 番目の要素はレイアウト名でそれ以降は `Layout` です。

24.3 Tix — Tk の拡張ウィジェット

`Tix` (Tk Interface Extension) モジュールは豊富な追加ウィジェットを提供します。標準 Tk ライブラリには多くの有用なウィジェットがありますが、完全では決してありません。 `Tix` ライブラリは標準 Tk に欠けている一般的に必要とされるウィジェットの大部分を提供します: `HList`、`ComboBox`、`Control` (別名 `SpinBox`) および各種のスクロール可能なウィジェット。 `Tix` には、一般的に幅広い用途に役に立つたくさんのウィジェットも含まれています: `NoteBook`、`FileEntry`、`PanedWindow` など。それらは 40 以上あります。

これら全ての新しいウィジェットと使うと、より便利でより直感的なユーザインタフェース作成し、あなたは新しい相互作用テクニックをアプリケーションに導入することができます。アプリケーションとユーザに特有の要求に合うように、大部分のアプリケーションウィジェットを選ぶことによって、アプリケーションを設計できます。

注釈: `Tix` モジュールは、Python 3 では `tkinter.tix` にリネームされました。 `2to3` ツールが自動的にソースコードの `import` を修正します。

参考:

[Tix Homepage](#) `Tix` のホームページ。ここには追加ドキュメントとダウンロードへのリンクがあります。

[Tix Man Pages](#) `man` ページと参考資料のオンライン版。

[Tix Programming Guide](#) プログラマ用参考資料のオンライン版。

[Tix Development Applications](#) `Tix` と `Tkinter` プログラムの開発のための `Tix` アプリケーション。 `Tide` アプリケーションは Tk または `Tkinter` に基づいて動作します。また、リモートで `Tix/Tk/Tkinter` アプリケーションを変更やデバッグするためのインスペクタ `TixInspect` が含まれます。

24.3.1 Tix を使う

class `Tix.Tix` (`screenName`[, `baseName`[, `className`]])

たいていはアプリケーションのメインウィンドウを表す `Tix` のトップレベルウィジェット。それには `Tcl` インタープリタが付随します。

`Tix` モジュールのクラスは `Tkinter` モジュールのクラスをサブクラス化します。前者は後者をインポートします。だから、`Tkinter` と一緒に `Tix` を使うためにやらなければならないのは、モジュールを一つインポートすることだけです。一般的に、`Tix` をインポートし、トップレベルでの `Tkinter.Tk` の呼び出しを `Tix.Tk` に置き換えるだけでよいのです：

```
import Tix
from Tkconstants import *
root = Tix.Tk()
```

`Tix` を使うためには、通常 `Tk` ウィジェットのインストールと平行して、`Tix` ウィジェットをインストールしなければなりません。インストールをテストするために、次のことを試してください：

```
import Tix
root = Tix.Tk()
root.tk.eval('package require Tix')
```

これが失敗した場合は、先に進む前に解決しなければならない問題が `Tk` のインストールにあることとなります。インストールされた `Tix` ライブラリを指定するためには環境変数 `TIX_LIBRARY` を使ってください。`Tk` 動的オブジェクトライブラリ (`tk8183.dll` または `libtk8183.so`) を含むディレクトリと同じディレクトリに、動的オブジェクトライブラリ (`tix8183.dll` または `libtix8183.so`) があるかどうかを確かめてください。動的オブジェクトライブラリのあるディレクトリには、`pkgIndex.tcl` (大文字、小文字を区別します) という名前のファイルも含まれているべきで、それには次の一行が含まれます：

```
package ifneeded Tix 8.1 [list load "[file join $dir tix8183.dll]" Tix]
```

24.3.2 Tix ウィジェット

`Tix` は 40 個以上のウィジェットクラスを `Tkinter` のレパートリーに導入します。標準配布の `Demo/tix` ディレクトリには、`Tix` ウィジェットのデモがあります。

基本ウィジェット

class `Tix.Balloon`

ヘルプを提示するためにウィジェット上にポップアップする `Balloon`。ユーザがカーソルを `Balloon` ウィジェットが束縛されているウィジェット内部へ移動させたとき、説明のメッセージが付いた小さなポップアップウィンドウがスクリーン上に表示されます。

class `Tix.ButtonBox`

`ButtonBox` ウィジェットは、`Ok` `Cancel` のためだけに普通は使われるようなボタンボックスを作成します。

class `Tix.ComboBox`

`ComboBox` ウィジェットは MS Windows のコンボボックスコントロールに似ています。ユーザはエントリ・サブウィジェットでタイプするか、リストボックス・サブウィジェットから選択するかのどちらかで選択肢を選びます。

class `Tix.Control`

`Control` ウィジェットは `SpinBox` ウィジェットとしても知られています。ユーザは二つの矢印ボタンを押すか、またはエントリに直接値を入力して値を調整します。新しい値をユーザが定義した上限と下限に対してチェックします。

class `Tix.LabelEntry`

`LabelEntry` ウィジェットはエントリウィジェットとラベルを一つのメガウィジェットにまとめたものです。”記入形式”型のインタフェースの作成を簡単に行うために使うことができます。

class `Tix.LabelFrame`

`LabelFrame` ウィジェットはフレームウィジェットとラベルを一つのメガウィジェットにまとめたものです。`LabelFrame` ウィジェット内部にウィジェットを作成するためには、`frame` サブウィジェットに対して新しいウィジェットを作成し、それらを `frame` サブウィジェット内部で取り扱います。

class `Tix.Meter`

`Meter` ウィジェットは実行に時間のかかるバックグラウンド・ジョブの進み具合を表示するために使用できます。

class `Tix.OptionMenu`

`OptionMenu` はオプションのメニューボタンを作成します。

class `Tix.PopupMenu`

`PopupMenu` ウィジェットは `tk_popup` コマンドの代替品として使用できます。*`Tix.PopupMenu`* ウィジェットの利点は、操作するためにより少ないアプリケーション・コードしか必要としないことです。

class `Tix.Select`

`Select` ウィジェットはボタン・サブウィジェットのコンテナです。ユーザに対する選択オプションのラジオボックスまたはチェックボックス形式を提供するために利用することができます。

class `Tix.StdButtonBox`

`StdButtonBox` ウィジェットは、Motif に似たダイアログボックスのための標準的なボタンのグループです。

ファイルセクタ

class `Tix.DirList`

`DirList` ウィジェットは、ディレクトリの一覧ビュー (その前のディレクトリとサブディレクトリ) を表示します。ユーザはリスト内の表示されたディレクトリの一つを選択したり、あるいは他のディレクトリへ変更したりできます。

class `Tix.DirTree`

`DirTree` ウィジェットはディレクトリのツリービュー (その前のディレクトリとそのサブディレクトリ)

を表示します。ユーザはリスト内に表示されたディレクトリの一つを選択したり、あるいは他のディレクトリに変更したりできます。

class `Tix.DirSelectDialog`

`DirSelectDialog` ウィジェットは、ダイアログウィンドウにファイルシステム内のディレクトリを提示します。望みのディレクトリを選択するために、ユーザはファイルシステムを介して操作するこのダイアログウィンドウを利用できます。

class `Tix.DirSelectBox`

`DirSelectBox` は標準 Motif(TM) ディレクトリ選択ボックスに似ています。ユーザがディレクトリを選択するために一般的に使われます。DirSelectBox は主に最近 `ComboBox` ウィジェットに選択されたディレクトリを保存し、すばやく再選択できるようにします。

class `Tix.ExFileSelectBox`

`ExFileSelectBox` ウィジェットは、たいてい `tixExFileSelectDialog` ウィジェット内に組み込まれます。ユーザがファイルを選択するのに便利なメソッドを提供します。`ExFileSelectBox` ウィジェットのスタイルは、MS Windows 3.1 の標準ファイルダイアログにとてもよく似ています。

class `Tix.FileSelectBox`

`FileSelectBox` は標準的な Motif(TM) ファイル選択ボックスに似ています。ユーザがファイルを選択するために一般的に使われます。FileSelectBox は主に最近 `ComboBox` ウィジェットに選択されたファイルを保存し、素早く再選択できるようにします。

class `Tix.FileEntry`

`FileEntry` ウィジェットはファイル名を入力するために使うことができます。ユーザは手でファイル名をタイプできます。その代わりに、ユーザはエントリの横に並んでいるボタンウィジェットを押すことができます。それはファイル選択ダイアログを表示します。

ハイアラキカルリストボックス

class `Tix.HList`

`HList` ウィジェットは階層構造をもつどんなデータ (例えば、ファイルシステムディレクトリツリー) でも表示するために使用できます。リストエントリは字下げされ、階層のそれぞれの場所に応じて分岐線で接続されます。

class `Tix.CheckList`

`CheckList` ウィジェットは、ユーザが選ぶ項目のリストを表示します。CheckList は Tk のチェックリストやラジオボタンより多くの項目を扱うことができることを除いて、チェックボタンあるいはラジオボタンウィジェットと同じように動作します。

class `Tix.Tree`

`Tree` ウィジェットは階層的なデータをツリー形式で表示するために使うことができます。ユーザはツリーの一部を開いたり閉じたりすることによって、ツリーの見えを調整できます。

タビュラーリストボックス

class `Tix.TList`

`TList` ウィジェットは、表形式でデータを表示するために使うことができます。 `TList` ウィジェットのリスト・エントリは、Tk のリストボックス・ウィジェットのエントリに似ています。主な差は、(1) `TList` ウィジェットはリスト・エントリを二次元形式で表示でき、(2) リスト・エントリに対して複数の色やフォントだけでなく画像も使うことができるということです。

管理ウィジェット

`class Tix.PanedWindow`

`PanedWindow` ウィジェットは、ユーザがいくつかのペインのサイズを対話的に操作できるようにします。ペインは垂直または水平のどちらかに配置されます。ユーザは二つのペインの間でリサイズ・ハンドルをドラッグしてペインの大きさを変更します。

`class Tix.ListNoteBook`

`ListNoteBook` ウィジェットは、`TixNoteBook` ウィジェットにとてもよく似ています。ノートのメタファを使って限られた空間をに多くのウィンドウを表示するために使われます。ノートはたくさんのページ(ウィンドウ)に分けられています。ある時には、これらのページの一つしか表示できません。ユーザは `hlist` サブウィジェットの中の望みのページの名前を選択することによって、これらのページを切り替えることができます。

`class Tix.NoteBook`

`NoteBook` ウィジェットは、ノートのメタファを多くのウィンドウを表示することができます。ノートはたくさんのページに分けられています。ある時には、これらのページの一つしか表示できません。ユーザは `NoteBook` ウィジェットの一番上にある目に見える”タブ”を選択することで、これらのページを切り替えることができます。

画像タイプ

`Tix` モジュールは次のものを追加します:

- 全ての `Tix` と `Tkinter` ウィジェットに対して XPM ファイルからカラー画像を作成する `pixmap` 機能。
- `Compound` 画像タイプは複数の水平方向の線から構成される画像を作成するために使うことができます。それぞれの線は左から右に並べられた一連のアイテム(テキスト、ビットマップ、画像あるいは空白)から作られます。例えば、Tk の `Button` ウィジェットの中にビットマップとテキスト文字列を同時に表示するために `compound` 画像は使われます。

その他のウィジェット

`class Tix.InputOnly`

`InputOnly` ウィジェットは、ユーザから入力を受け付けます。それは、`bind` コマンドを使って行われます (Unix のみ)。

ジオメトリマネージャを作る

加えて、`Tix` は次のものを提供することで `Tkinter` を補強します:

class `Tix.Form`

Tk ウィジェットに対する接続ルールに基づいたジオメトリマネージャを 作成 (Form) します。

24.3.3 Tix コマンド

class `Tix.tixCommand`

`tix` コマンドは `Tix` の内部状態と `Tix` アプリケーション・コンテキストのいろいろな要素へのアクセスを提供します。これらのメソッドによって操作される情報の大部分は、特定のウィンドウというよりもむしろアプリケーション全体がスクリーンあるいはディスプレイに関するものです。

現在の設定を見るための一般的な方法は:

```
import Tix
root = Tix.Tk()
print root.tix_configure()
```

`tixCommand.tix_configure` (*cnf=None **kw*)

`Tix` アプリケーション・コンテキストの設定オプションを問い合わせたり、変更したりします。オプションが指定されなければ、利用可能なオプションすべてのディクショナリを返します。オプションが値なしで指定された場合は、メソッドは指定されたオプションを説明するリストを返します (このリストはオプションが指定されていない場合に返される値に含まれている、指定されたオプションに対応するサブリストと同一です)。一つ以上のオプション-値のペアが指定された場合は、メソッドは与えられたオプションが与えられた値を持つように変更します。この場合は、メソッドは空文字列を返します。オプションは設定オプションのどれでも構いません。

`tixCommand.tix_cget` (*option*)

option によって与えられた設定オプションの現在の値を返します。オプションは設定オプションのどれでも構いません。

`tixCommand.tix_getbitmap` (*name*)

ビットマップディレクトリの一つの中の `name.xpm` または `name` という名前のビットマップファイルの場所を見つけ出します (`tix.addbitmapdir()` メソッドを参照してください)。 `tix_getbitmap()` を使うことで、アプリケーションにビットマップファイルのパス名をハードコーディングすることを選避けることができます。成功すれば、文字 `@` を先頭に付けたビットマップファイルの完全なパス名を返します。戻り値を Tk と `Tix` ウィジェットの `bitmap` オプションを設定するために使うことができます。

`tixCommand.tix_addbitmapdir` (*directory*)

`Tix` は `tix_getimage()` と `tix_getbitmap()` メソッドが画像ファイルを検索するディレクトリのリストを保持しています。標準ビットマップディレクトリは `$TIX_LIBRARY/bitmaps` です。 `tix_addbitmapdir()` メソッドは *directory* をこのリストに追加します。そのメソッドを使うことによって、アプリケーションの画像ファイルを `tix_getimage()` または `tix_getbitmap()` メソッドを使って見つけることができます。

`tixCommand.tix_filedialog` (*[dlgclass]*)

このアプリケーションからの異なる呼び出しの間で共有される可能性があるファイル選択ダイアログ

を返します。最初に呼ばれた時に、このメソッドはファイル選択ダイアログ・ウィジェットを作成します。このダイアログはその後のすべての `tix.filedialog()` への呼び出しで返されます。オプションの `dlgclass` パラメータは、要求されているファイル選択ダイアログ・ウィジェットの型を指定するために文字列として渡されます。指定可能なオプションは `tix`、`FileSelectDialog` あるいは `tixExFileSelectDialog` です。

`tixCommand.tix_getimage(self, name)`

ビットマップディレクトリの一つの中の `name.xpm`、`name.xbm` または `name.ppm` という名前の画像ファイルの場所を見つけ出します (上の `tix.addbitmapdir()` メソッドを参照してください)。同じ名前 (だが異なる拡張子) のファイルが一つ以上ある場合は、画像のタイプが X ディスプレイの深さに応じて選択されます。xbm 画像はモノクロディスプレイの場合に選択され、カラー画像はカラーディスプレイの場合に選択されます。 `tix_getimage()` を使うことによって、アプリケーションに画像ファイルのパス名をハードコーディングすることを避けられます。成功すれば、このメソッドは新たに作成した画像の名前を返し、Tk と Tix ウィジェットの `image` オプションを設定するためにそれを使うことができます。

`tixCommand.tix_option.get(name)`

Tix のスキーム・メカニズムによって保持されているオプションを得ます。

`tixCommand.tix_resetoptions(newScheme, newFontSet[, newScmPrio])`

Tix アプリケーションのスキームとフォントセットを `newScheme` と `newFontSet` それぞれへと再設定します。これはこの呼び出し後に作成されたそれらのウィジェットだけに影響します。そのため、Tix アプリケーションのどんなウィジェットを作成する前に `resetoptions` メソッドを呼び出すのが最も良いのです。

オプション・パラメータ `newScmPrio` を、Tix スキームによって設定される Tk オプションの優先度レベルを再設定するために与えることができます。

Tk が X オプションデータベースを扱う方法のため、Tix がインポートされ初期化された後に、カラースキームとフォントセットを `tix.config()` メソッドを使って再設定することができません。その代わりに、 `tix_resetoptions()` メソッドを使わなければならないのです。

24.4 ScrolledText — スクロールするテキストウィジェット

`ScrolledText` モジュールは “正しい動作” をするように設定された垂直スクロールバーをもつ基本的なテキストウィジェットを実装する同じ名前のクラスを提供します。 `ScrolledText` クラスを使うことは、テキストウィジェットとスクロールバーを直接設定するより簡単です。コンストラクタは `Tkinter.Text` クラスのものを同じです。

注釈: ”`ScrolledText` は Python 3 で `tkinter.scrolledtext` に改名されました。 `2to3` ツールが自動的にソースコードの `import` を修正します。

テキストウィジェットとスクロールバーは `Frame` の中に一緒に `pack` され、 `Grid` と `Pack` ジオメトリマネージャのメソッドは `Frame` オブジェクトから得られます。これによって、もっとも標準的なジオメトリマネ

ジャの振る舞いにするために、直接 `ScrolledText` ウィジェットを使えるようになります。

特定の制御が必要ならば、以下の属性が利用できます:

`ScrolledText.frame`

テキストとスクロールバーウィジェットを取り囲むフレーム。

`ScrolledText.vbar`

スクロールバーウィジェット。

24.5 turtle — Tk のためのタートルグラフィックス

24.5.1 はじめに

タートルグラフィックスは子供にプログラミングを紹介するのによく使われます。タートルグラフィックスは Wally Feurzig と Seymore Papert が 1966 年に開発した Logo プログラミング言語の一部でした。

x-y 平面の (0, 0) から動き出すロボット亀を想像してみてください。 `turtle.forward(15)` という命令を出すと、その亀が (スクリーン上で!) 15 ピクセル分顔を向けている方向に動き、動きに沿って線を引きます。 `turtle.left(25)` という命令を出すと、今度はその場で 25 度反時計回りに回ります。

これらの命令と他の同様な命令を組み合わせることで、複雑な形や絵が簡単に描けます。

`turtle` モジュールは同じ名前を持った Python 2.5 までのモジュールの拡張された再実装です。

再実装に際しては古い `turtle` モジュールのメリットをそのままに、(ほぼ) 100% 互換性を保つようにしました。すなわち、まず第一に、学習中のプログラマがモジュールを `-n` スイッチを付けて走らせている IDLE の中から全てのコマンド、クラス、メソッドを対話的に使えるようにしました。

`turtle` モジュールはオブジェクト指向と手続き指向の両方の方法でタートルグラフィックス・プリミティブを提供します。グラフィックスの基礎として `Tkinter` を使っているために、Tk をサポートした Python のバージョンが必要です。

オブジェクト指向インターフェイスでは、本質的に 2+2 のクラスを使います:

1. `TurtleScreen` クラスはタートルが絵を描きながら走り回る画面を定義します。そのコンストラクタには `Tkinter.Canvas` または `ScrolledCanvas` を渡す必要があります。 `turtle` をアプリケーションの一部として用いたい場合にはこれを使うべきです。

`Screen()` 関数は `TurtleScreen` のサブクラスのシングルトンオブジェクトを返します。 `turtle` をグラフィックスを使う一つの独立したツールとして使う場合には、この関数を呼び出すべきです。シングルトンなので、そのクラスからの継承はできません。

`TurtleScreen/Screen` の全てのメソッドは関数としても、すなわち、手続き指向インターフェイスの一部としても存在しています。

2. `RawTurtle` (別名: `RawPen`) は `TurtleScreen` 上に絵を描く Turtle オブジェクトを定義します。コンストラクタには `Canvas`, `ScrolledCanvas`, `TurtleScreen` のいずれかを引数として渡して `RawTurtle` オブジェクトがどこに絵を描くかを教えます。

RawTurtle の派生はサブクラス *Turtle* (別名: Pen) で、”唯一の” *Screen* (既に与えられているのでなければ自動的に作られたインスタンス) に絵を描きます。

RawTurtle/Turtle の全てのメソッドは関数としても、すなわち、手続き指向インターフェイスの一部としても存在しています。

手続き型インターフェイスでは *Screen* および *Turtle* クラスのメソッドを元にした関数を提供しています。その名前は対応するメソッドと一緒です。Screen のメソッドを元にした関数が呼び出されるといつでも screen オブジェクトが自動的に作られます。Turtle のメソッドを元にした関数が呼び出されるといつでも (名無しの) turtle オブジェクトが自動的に作られます。

複数のタートルを一つのスクリーン上で使いたい場合、オブジェクト指向インターフェイスを使わなければなりません。

注釈: 以下の文書では関数に対する引数リストが与えられています。メソッドでは、勿論、ここでは省略されている *self* が第一引数になります。

24.5.2 Turtle および Screen のメソッド概観

Turtle のメソッド

Turtle の動き

移動および描画

```
forward() | fd()
backward() | bk() | back()
right() | rt()
left() | lt()
goto() | setpos() | setposition()
setx()
sety()
setheading() | seth()
home()
circle()
dot()
stamp()
clearstamp()
clearstamps()
undo()
speed()
```

Turtle の状態を知る

```
position() | pos()
towards()
xcor()
ycor()
heading()
distance()
```

設定と計測

```
degrees()
radians()
```

Pen の制御

描画状態

```
pendown() | pd() | down()
penup() | pu() | up()
pensize() | width()
pen()
isdown()
```

色の制御

```
color()
pencolor()
fillcolor()
```

塗りつぶし

```
fill()
begin_fill()
end_fill()
```

さらなる描画の制御

```
reset()
clear()
write()
```

タートルの状態

可視性

```
showturtle() | st()
hideturtle() | ht()
isvisible()
```

見たい目

`shape()`
`resizemode()`
`shapeseize()` | `turtlesize()`
`settiltangle()`
`tiltangle()`
`tilt()`

イベントを利用する

`onclick()`
`onrelease()`
`ondrag()`
`mainloop()` | `done()`

特別な Turtle のメソッド

`begin_poly()`
`end_poly()`
`get_poly()`
`clone()`
`getturtle()` | `getpen()`
`getscreen()`
`setundobuffer()`
`undobufferentries()`
`tracer()`
`window_width()`
`window_height()`

TurtleScreen/Screen のメソッド

ウィンドウの制御

`bgcolor()`
`bgpic()`
`clear()` | `clearscreen()`
`reset()` | `resetscreen()`
`screensize()`
`setworldcoordinates()`

アニメーションの制御

`delay()`

```
tracer()  
update()
```

スクリーンイベントを利用する

```
listen()  
onkey()  
onclick() | onscreenclick()  
ontimer()
```

設定と特殊なメソッド

```
mode()  
colormode()  
getcanvas()  
getshapes()  
register_shape() | addshape()  
turtles()  
window_height()  
window_width()
```

Screen 独自のメソッド

```
bye()  
exitonclick()  
setup()  
title()
```

24.5.3 RawTurtle/Turtle のメソッドと対応する関数

この節のほとんどの例では `turtle` という名前の Turtle インスタンスを使います。

Turtle の動き

```
turtle.forward(distance)  
turtle.fd(distance)
```

パラメータ **distance** – 数 (整数または浮動小数点数)

タートルが頭を向けている方へ、タートルを距離 *distance* だけ前進させます。

```
>>> turtle.position()  
(0.00,0.00)  
>>> turtle.forward(25)  
>>> turtle.position()
```

(次のページに続く)

(前のページからの続き)

```
(25.00, 0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00, 0.00)
```

`turtle.back(distance)``turtle.bk(distance)``turtle.backward(distance)`パラメータ **distance** – 数

タートルが頭を向けている方と反対方向へ、タートルを距離 *distance* だけ後退させます。タートルの向きは変えません。

```
>>> turtle.position()
(0.00, 0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00, 0.00)
```

`turtle.right(angle)``turtle.rt(angle)`パラメータ **angle** – 数 (整数または浮動小数点数)

タートルを *angle* 単位だけ右に回します。(単位のデフォルトは度ですが、`degrees()` と `radians()` 関数を使って設定できます。) 角度の向きはタートルのモードによって意味が変わります。 `mode()` を参照してください。

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

`turtle.left(angle)``turtle.lt(angle)`パラメータ **angle** – 数 (整数または浮動小数点数)

タートルを *angle* 単位だけ左に回します。(単位のデフォルトは度ですが、`degrees()` と `radians()` 関数を使って設定できます。) 角度の向きはタートルのモードによって意味が変わります。 `mode()` を参照してください。

```
>>> turtle.heading()
22.0
>>> turtle.left(45)
>>> turtle.heading()
67.0
```



```

turtle.goto(x, y=None)
turtle.setpos(x, y=None)
turtle.setposition(x, y=None)

```

パラメータ

- **x** – 数または数のペア/ベクトル
- **y** – 数または None

y が None の場合、**x** は座標のペアかまたは *Vec2D* (たとえば *pos()* で返されます) でなければなりません。

タートルを指定された絶対位置に移動します。ペンが下りていれば線を引きます。タートルの向きは変わりません。

```

>>> tp = turtle.pos()
>>> tp
(0.00, 0.00)
>>> turtle.setpos(60, 30)
>>> turtle.pos()
(60.00, 30.00)
>>> turtle.setpos((20, 80))
>>> turtle.pos()
(20.00, 80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00, 0.00)

```

```

turtle.setx(x)

```

パラメータ **x** – 数 (整数または浮動小数点数)

タートルの第一座標を **x** にします。第二座標は変わりません。

```

>>> turtle.position()
(0.00, 240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00, 240.00)

```

```

turtle.sety(y)

```

パラメータ **y** – 数 (整数または浮動小数点数)

タートルの第二座標を **y** にします。第一座標は変わりません。

```

>>> turtle.position()
(0.00, 40.00)
>>> turtle.sety(-10)
>>> turtle.position()
(0.00, -10.00)

```

```
turtle.setheading(to_angle)
```

```
turtle.seth(to_angle)
```

パラメータ **to_angle** – 数 (整数または浮動小数点数)

タートルの向きを *to_angle* に設定します。以下はよく使われる方向を度で表わしたものです:

標準モード	logo モード
0 - 東	0 - 北
90 - 北	90 - 東
180 - 西	180 - 南
270 - 南	270 - 西

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

```
turtle.home()
```

タートルを原点 – 座標 (0, 0) – に移動し、向きを開始方向に設定します (開始方向はモードによって違います。 [mode\(\)](#) を参照してください)。

```
>>> turtle.heading()
90.0
>>> turtle.position()
(0.00, -10.00)
>>> turtle.home()
>>> turtle.position()
(0.00, 0.00)
>>> turtle.heading()
0.0
```

```
turtle.circle(radius, extent=None, steps=None)
```

パラメータ

- **radius** – 数
- **extent** – 数 (または None)
- **steps** – 整数 (または None)

半径 *radius* の円を描きます。中心はタートルの左 *radius* ユニットの点です。 *extent* – 角度です – は円のどの部分を描くかを決定します。 *extent* が与えられなければ、デフォルトで完全な円になります。 *extent* が完全な円でない場合は、弧の一つの端点は、現在のペンの位置です。 *radius* が正の場合、弧は反時計回りに描かれます。そうでなければ、時計回りです。最後にタートルの向きが *extent* 分だけ変わります。

円は内接する正多角形で近似されます。 *steps* でそのために使うステップ数を決定します。この値は与えられなければ自動的に計算されます。また、これを正多角形の描画に利用することもできます。

```

>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(50)
>>> turtle.position()
(-0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(120, 180) # draw a semicircle
>>> turtle.position()
(0.00,240.00)
>>> turtle.heading()
180.0

```

`turtle.dot (size=None, *color)`

パラメータ

- **size** – 1 以上の整数 (与えられる場合には)
- **color** – 色を表わす文字列またはタプル

直径 *size* の丸い点を *color* で指定された色で描きます。 *size* が与えられなかった場合、`pensize+4` と `2*pensize` の大きい方が使われます。

```

>>> turtle.home()
>>> turtle.dot()
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
>>> turtle.position()
(100.00,-0.00)
>>> turtle.heading()
0.0

```

`turtle.stamp()`

キャンバス上の現在タートルがいる位置にタートルの姿のハンコを押します。そのハンコに対して `stamp.id` が返されますが、これを使うと後で `clearstamp(stamp_id)` のように呼び出して消すことができます。

```

>>> turtle.color("blue")
>>> turtle.stamp()
11
>>> turtle.fd(50)

```

`turtle.clearstamp (stampid)`

パラメータ **stampid** – 整数で、先立つ `stamp()` 呼出して返された値でなければなりません
stampid に対応するハンコを消します。

```
>>> turtle.position()
(150.00,-0.00)
>>> turtle.color("blue")
>>> astamp = turtle.stamp()
>>> turtle.fd(50)
>>> turtle.position()
(200.00,-0.00)
>>> turtle.clearstamp(astamp)
>>> turtle.position()
(200.00,-0.00)
```

`turtle.clearstamps(n=None)`

パラメータ **n** – 整数 (または None)

全ての、または最初の/最後の *n* 個のハンコを消します。 *n* が None の場合、全てのハンコを消します。
n が正の場合には最初の *n* 個、 *n* が負の場合には最後の *n* 個を消します。

```
>>> for i in range(8):
...     turtle.stamp(); turtle.fd(30)
13
14
15
16
17
18
19
20
>>> turtle.clearstamps(2)
>>> turtle.clearstamps(-2)
>>> turtle.clearstamps()
```

`turtle.undo()`

最後の (繰り返すことにより複数の) タートルの動きを取り消します。取り消しできる動きの最大数は `undobuffer` のサイズによって決まります。

```
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
...     turtle.undo()
```

`turtle.speed(speed=None)`

パラメータ **speed** – 0 から 10 までの整数またはスピードを表わす文字列 (以下の説明を参照)

タートルのスピードを 0 から 10 までの範囲の整数に設定します。引数を与えられない場合は現在のスピードを返します。

与えられた数字が 10 より大きかったり 0.5 より小さかったりした場合は、スピードは 0 になります。
スピードを表わす文字列は次のように数字に変換されます:

- "fastest": 0
- "fast": 10
- "normal": 6
- "slow": 3
- "slowest": 1

1 から 10 までのスピードを上げていくにつれて線を描いたりタートルが回ったりするアニメーションがだんだん速くなります。

注意: *speed* = 0 はアニメーションを無くします。forward/backward ではタートルがジャンプし、left/right では瞬時に方向を変えます。

```
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
```

Turtle の状態を知る

`turtle.position()`

`turtle.pos()`

タートルの現在位置を (*Vec2D* のベクトルとして) 返します。

```
>>> turtle.pos()
(440.00, -0.00)
```

`turtle.towards(x, y=None)`

パラメータ

- **x** – 数または数のペア/ベクトルまたはタートルのインスタンス
- **y** – *x* が数ならば数、そうでなければ None

タートルの位置から指定された (x,y) への直線の角度を返します。この値はタートルの開始方向にそして開始方向はモード ("standard"/"world" または "logo") に依存します。

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0, 0)
225.0
```

`turtle.xcor()`

タートルの x 座標を返します。

```
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28, 76.60)
>>> print turtle.xcor()
64.2787609687
```

`turtle.ycor()`

タートルの y 座標を返します。

```
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print turtle.pos()
(50.00, 86.60)
>>> print turtle.ycor()
86.6025403784
```

`turtle.heading()`

タートルの現在の向きを返します (返される値はタートルのモードに依存します。 `mode()` を参照してください)。

```
>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0
```

`turtle.distance(x, y=None)`

パラメータ

- **x** – 数または数のペア/ベクトルまたはタートルのインスタンス
- **y** – **x** が数ならば数、そうでなければ `None`

タートルから与えられた (x,y) あるいはベクトルあるいは渡されたタートルへの距離を、タートルのステップを単位として測った値を返します。

```
>>> turtle.home()
>>> turtle.distance(30, 40)
50.0
>>> turtle.distance((30, 40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

設定と計測

`turtle.degrees (fullcircle=360.0)`

パラメータ **fullcircle** – 数

角度を計る単位「度」を、円周を何等分するかという値に指定します。デフォルトは 360 等分で通常の意味での度です。

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0

Change angle measurement unit to grad (also known as gon,
grade, or gradian and equals 1/100-th of the right angle.)
>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
>>> turtle.degrees(360)
>>> turtle.heading()
90.0
```

`turtle.radians()`

角度を計る単位をラジアンにします。 `degrees(2*math.pi)` と同じ意味です。

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966
```

Pen の制御

描画状態

`turtle.pendown()`

`turtle.pd()`

`turtle.down()`

ペンを下ろします – 動くと線が引かれます。

`turtle.penup()`

`turtle.pu()`

`turtle.up()`

ペンを上げます – 動いても線は引かれません。

`turtle.pensize (width=None)`

`turtle.width (width=None)`

パラメータ **width** – 正の数

線の太さを *width* にするか、または現在の太さを返します。resizemode が "auto" でタートルの形が多角形の場合、その多角形も同じ太さで描画されます。引数が渡されなければ、現在の pensize が返されます。

```
>>> turtle.pensize()
1
>>> turtle.pensize(10)    # from here on lines of width 10 are drawn
```

`turtle.pen` (*pen=None, **pendict*)

パラメータ

- **pen** – 以下にリストされたキーをもった辞書
- **pendict** – 以下にリストされたキーをキーワードとするキーワード引数

ペンの属性を "pen-dictionary" に以下のキー/値ペアで設定するかまたは返します:

- "shown": True/False
- "pendown": True/False
- "pencolor": 色文字列または色タプル
- "fillcolor": 色文字列または色タプル
- "pensize": 正の数
- "speed": 0 から 10 までの整数
- "resizemode": "auto" または "user" または "noresize"
- "stretchfactor": (正の数, 正の数)
- "outline": 正の数
- "tilt": 数

この辞書を以降の *pen()* 呼出しに渡して以前のペンの状態に復旧することができます。さらに一つ以上の属性をキーワード引数として渡すこともできます。一つの文で幾つものペンの属性を設定するのに使えます。

```
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
 ('shown', True), ('speed', 9), ('stretchfactor', (1, 1)), ('tilt', 0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", "")
>>> turtle.penup()
>>> sorted(turtle.pen().items())
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow'),
```

(次のページに続く)

(前のページからの続き)

```
( 'pendown', False), ('pensize', 10), ('resizemode', 'noresize'),
('shown', True), ('speed', 9), ('stretchfactor', (1, 1)), ('tilt', 0)]
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red'),
('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
('shown', True), ('speed', 9), ('stretchfactor', (1, 1)), ('tilt', 0)]
```

`turtle.isdown()`

もしペンが下りていれば `True` を、上がってれば `False` を返します。

```
>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
>>> turtle.isdown()
True
```

色の制御

`turtle.pencolor(*args)`

ペンの色 (`pencolor`) を設定するかまたは返します。

4 種類の入力形式が受け入れ可能です:

`pencolor()` 現在のペンの色を色指定文字列またはタプルで返します (例を見て下さい)。次の `color/pencolor/fillcolor` の呼び出しへの入力に使うこともあるでしょう。

`pencolor(colorstring)` ペンの色を *colorstring* に設定します。その値は Tk の色指定文字列で、`"red"`, `"yellow"`, `"#33cc8c"` のような文字列です。

`pencolor((r, g, b))` ペンの色を *r, g, b* のタプルで表された RGB の色に設定します。各 *r, g, b* は 0 から `colormode` の間の値でなければなりません。ここで `colormode` は 1.0 か 255 のどちらかです (`colormode()` を参照)。

`pencolor(r, g, b)`

ペンの色を *r, g, b* で表された RGB の色に設定します。各 *r, g, b* は 0 から `colormode` の間の値でなければなりません。

タートルの形 (`turtleshape`) が多角形の場合、多角形の外側が新しく設定された色で描かれます。

```
>>> colormode()
1.0
>>> turtle.pencolor()
'red'
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
'brown'
```

(次のページに続く)

(前のページからの続き)

```
>>> tup = (0.2, 0.8, 0.55)
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)
>>> colormode(255)
>>> turtle.pencolor()
(51, 204, 140)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
(50, 193, 143)
```

`turtle.fillcolor(*args)`

塗りつぶしの色 (fillcolor) を設定するかまたは返します。

4 種類の入力形式が受け入れ可能です:

`fillcolor()` 現在の塗りつぶしの色を色指定文字列またはタプルで返します (例を見て下さい)。次の `color/pencolor/fillcolor` の呼び出しへの入力に使うこともあるでしょう。

`fillcolor(colorstring)` 塗りつぶしの色を *colorstring* に設定します。その値は Tk の色指定文字列で、"red", "yellow", "#33cc8c" のような文字列です。

`fillcolor((r, g, b))` 塗りつぶしの色を *r, g, b* のタプルで表された RGB の色に設定します。各 *r, g, b* は 0 から `colormode` の間の値でなければなりません。ここで `colormode` は 1.0 か 255 のどちらかです (`colormode()` を参照)。

`fillcolor(r, g, b)`

塗りつぶしの色を *r, g, b* で表された RGB の色に設定します。各 *r, g, b* は 0 から `colormode` の間の値でなければなりません。

タートルの形 (`turtleshape`) が多角形の場合、多角形の内側が新しく設定された色で描かれます。

```
>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
'violet'
>>> col = turtle.pencolor()
>>> col
(50, 193, 143)
>>> turtle.fillcolor(col)
>>> turtle.fillcolor()
(50, 193, 143)
>>> turtle.fillcolor('#ffffff')
>>> turtle.fillcolor()
(255, 255, 255)
```

`turtle.color(*args)`

ペンの色 (`pencolor`) と塗りつぶしの色 (`fillcolor`) を設定するかまたは返します。

いくつかの入力形式が受け入れ可能です。形式ごとに 0 から 3 個の引数を以下のように使います:

`color()` 現在のペンの色と塗りつぶしの色を `pencolor()` および `fillcolor()` で返される色指定文字列またはタプルのペアで返します。

`color(colorstring), color((r,g,b)), color(r,g,b)` `pencolor()` の入力と同じですが、塗りつぶしの色とペンの色、両方を与えられた値に設定します。

`color(colorstring1, colorstring2), color((r1,g1,b1), (r2,g2,b2))`

`pencolor(colorstring1)` および `fillcolor(colorstring2)` を呼び出すのと等価です。もう一つの入力形式についても同様です。

タートルの形 (`turtleshape`) が多角形の場合、多角形の内側も外側も新しく設定された色で描かれます。

```
>>> turtle.color("red", "green")
>>> turtle.color()
('red', 'green')
>>> color("#285078", "#a0c8f0")
>>> color()
((40, 80, 120), (160, 200, 240))
```

こちらも参照: スクリーンのメソッド `colormode()`。

塗りつぶし

`turtle.fill(flag)`

パラメータ **flag** – True/False (またはそれぞれ 1/0)

塗りつぶしたい形を描く前に `fill(True)` を呼び出し、それが終わったら `fill(False)` を呼び出します。引数なしで呼び出されたときは、塗りつぶしの状態 (`fillstate`) の値 (True なら塗りつぶす、False なら塗りつぶさない) を返します。

```
>>> turtle.fill(True)
>>> for _ in range(3):
...     turtle.forward(100)
...     turtle.left(120)
...
>>> turtle.fill(False)
```

`turtle.begin_fill()`

塗りつぶしたい図形を描く直前に呼び出します。 `fill(True)` と等価です。

`turtle.end_fill()`

最後に呼び出された `begin_fill()` の後に描かれた図形を塗りつぶします。 `fill(False)` と等価です。

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
```

(次のページに続く)

(前のページからの続き)

```
>>> turtle.circle(80)
>>> turtle.end_fill()
```

さらなる描画の制御

`turtle.reset()`

タートルの描いたものをスクリーンから消し、タートルを中心に戻して、全ての変数をデフォルト値に設定し直します。

```
>>> turtle.goto(0,-22)
>>> turtle.left(100)
>>> turtle.position()
(0.00,-22.00)
>>> turtle.heading()
100.0
>>> turtle.reset()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

`turtle.clear()`

タートルの描いたものをスクリーンから消します。タートルは動かしません。タートルの状態と位置、それに他のタートルたちの描いたものは影響を受けません。

`turtle.write(arg, move=False, align="left", font=("Arial", 8, "normal"))`

パラメータ

- **arg** – TurtleScreen に書かれるオブジェクト
- **move** – True/False
- **align** – 文字列 "left", "center", "right" のどれか
- **font** – 三つ組み (fontname, fontsize, fonttype)

文字を書きます `arg` の文字列表現を、現在のタートルの位置に、`align` ("left", "center", "right" のどれか) に従って、与えられたフォントで。もし `move` が真ならば、ペンは書いた文の右下隅に移動します。デフォルトでは、`move` は False です。

```
>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)
```

タートルの状態

可視性

```
turtle.hideturtle()
```

```
turtle.ht()
```

タートルを見えなくします。複雑な図を描いている途中、タートルが見えないようにするのは良い考えです。というのもタートルを隠すことで描画が目に見えて速くなるからです。

```
>>> turtle.hideturtle()
```

```
turtle.showturtle()
```

```
turtle.st()
```

タートルが見えるようにします。

```
>>> turtle.showturtle()
```

```
turtle.isvisible()
```

タートルが見えている状態ならば `True` を、隠されていれば `False` を返します。

```
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

見た目

```
turtle.shape(name=None)
```

パラメータ **name** – 形の名前 (shapename) として正しい文字列

タートルの形を与えられた名前 (*name*) の形に設定するか、もしくは名前が与えられなければ現在の形の名前を返します。 *name* という名前の形は `TurtleScreen` の形の辞書に載っていなければなりません。最初は次の多角形が載っています: "arrow", "turtle", "circle", "square", "triangle", "classic"。形についての扱いを学ぶには `Screen` のメソッド `register_shape()` を参照して下さい。

```
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```

```
turtle.resizemode(rmode=None)
```

パラメータ **rmode** – 文字列 "auto", "user", "noresize" のどれか

サイズ変更のモード (resizemode) を "auto", "user", "noresize" のどれかに設定します。もし *rmode* が与えられなければ、現在のサイズ変更モードを返します。それぞれのサイズ変更モードは以下の効果を持ちます:

- "auto": ペンのサイズに対応してタートルの見た目を調整します。
- "user": 伸長係数 (stretchfactor) およびアウトライン幅 (outlinewidth) の値に対応してタートルの見た目を調整します。これらの値は `shapesize()` で設定します。
- "noresize": タートルの見た目を調整しません。

`resizemode("user")` は `shapesize()` に引数を渡したときに呼び出されます。

```
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

`turtle.shapesize(stretch_wid=None, stretch_len=None, outline=None)`

`turtle.turtlesize(stretch_wid=None, stretch_len=None, outline=None)`

パラメータ

- **stretch_wid** – 正の数
- **stretch_len** – 正の数
- **outline** – 正の数

ペンの属性 x/y-伸長係数および/またはアウトラインを返すかまたは設定します。サイズ変更のモードは "user" に設定されます。サイズ変更のモードが "user" に設定されたときかつそのときに限り、タートルは伸長係数 (stretchfactor) に従って伸長されて表示されます。 `stretch_wid` は進行方向に直交する向きの伸長係数で、 `stretch_len` は進行方向に沿ったの伸長係数、 `outline` はアウトラインの幅を決めるものです。

```
>>> turtle.shapesize()
(1, 1, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

`turtle.tilt(angle)`

パラメータ **angle** – 数

タートルの形 (turtleshape) を現在の傾斜角から角度 (*angle*) だけ回転します。このときタートルの進む方向は変わりません。

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
```

(次のページに続く)

(前のページからの続き)

```
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

`turtle.settiltangle(angle)`

パラメータ **angle** – 数

タートルの形 (`turtleshape`) を現在の傾斜角に関わらず、指定された角度 (*angle*) の向きに回転します。タートルの進む方向は 変わりません。

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.settiltangle(45)
>>> turtle.fd(50)
>>> turtle.settiltangle(-45)
>>> turtle.fd(50)
```

`turtle.tiltangle()`

現在の傾斜角を返します。すなわち、タートルの形が向いている角度と進んでいく方向との間の角度を返します。

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

イベントを利用する

`turtle.onclick(fun, btn=1, add=None)`

パラメータ

- **fun** – 2 引数の関数でキャンパスのクリックされた点の座標を引数として呼び出されるものです
- **btn** – マウスボタンの番号、デフォルトは 1 (左マウスボタン)
- **add** – `True` または `False` – `True` ならば、新しい束縛が追加されますが、そうでなければ、以前の束縛を置き換えます。

fun をタートルのマウスクリック (mouse-click) イベントに束縛します。*fun* が `None` ならば、既存の束縛が取り除かれます。無名タートル、つまり手続き的なやり方の例です:

```
>>> def turn(x, y):
...     left(180)
...
>>> onclick(turn)    # Now clicking into the turtle will turn it.
>>> onclick(None)    # event-binding will be removed
```

`turtle.onrelease` (*fun*, *btn=1*, *add=None*)

パラメータ

- **fun** – 2 引数の関数でキャンバスのクリックされた点の座標を引数として呼び出されるものです
- **btn** – マウスボタンの番号、デフォルトは 1 (左マウスボタン)
- **add** – True または False – True ならば、新しい束縛が追加されますが、そうでなければ、以前の束縛を置き換えます。

fun をタートルのマウスボタンリリース (mouse-button-release) イベントに束縛します。 *fun* が None ならば、既存の束縛が取り除かれます。

```
>>> class MyTurtle(Turtle):
...     def glow(self, x, y):
...         self.fillcolor("red")
...     def unglow(self, x, y):
...         self.fillcolor("")
...
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)    # clicking on turtle turns fillcolor red,
>>> turtle.onrelease(turtle.unglow) # releasing turns it to transparent.
```

`turtle.ondrag` (*fun*, *btn=1*, *add=None*)

パラメータ

- **fun** – 2 引数の関数でキャンバスのクリックされた点の座標を引数として呼び出されるものです
- **btn** – マウスボタンの番号、デフォルトは 1 (左マウスボタン)
- **add** – True または False – True ならば、新しい束縛が追加されますが、そうでなければ、以前の束縛を置き換えます。

fun をタートルのマウスムーブ (mouse-move) イベントに束縛します。 *fun* が None ならば、既存の束縛が取り除かれます。

注意: 全てのマウスムーブイベントのシーケンスに先立ってマウスクリックイベントが起こります。

```
>>> turtle.ondrag(turtle.goto)
```

この後、タートルをクリックしてドラッグするとタートルはスクリーン上を動きそれによって (ペンが下りていれば) 手書きの線ができあがります。

```
turtle.mainloop()
```

```
turtle.done()
```

イベントループの開始 - Tkinter の mainloop 関数の呼び出し。タートルグラフィックスプログラムの最後の実行文でなければなりません。

```
>>> turtle.mainloop()
```

特別な Turtle のメソッド

```
turtle.begin_poly()
```

多角形の頂点の記録を開始します。現在のタートル位置が最初の頂点です。

```
turtle.end_poly()
```

多角形の頂点の記録を停止します。現在のタートル位置が最後の頂点です。この頂点が最初の頂点と結ばれます。

```
turtle.get_poly()
```

最後に記録された多角形を返します。

```
>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
>>> turtle.left(20)
>>> turtle.fd(30)
>>> turtle.left(60)
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)
```

```
turtle.clone()
```

位置、向きその他のプロパティがそっくり同じタートルのクローンを作って返します。

```
>>> mick = Turtle()
>>> joe = mick.clone()
```

```
turtle.getturtle()
```

```
turtle.getpen()
```

Turtle オブジェクトそのものを返します。唯一の意味のある使い方: 無名タートルを返す関数として使う:

```
>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
```

```
turtle.getscreen()
```

タートルが描画中の *TurtleScreen* オブジェクトを返します。TurtleScreen のメソッドをそのオブジェクトに対して呼び出すことができます。

```
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

`turtle.setundobuffer(size)`

パラメータ **size** – 整数または None

アンドゥバッファを設定または無効化します。 *size* が整数ならばそのサイズの空のアンドゥバッファを用意します。 *size* の値はタートルのアクションを何度 `undo()` メソッド/関数で取り消せるかの最大数を与えます。 *size* が None ならば、アンドゥバッファは無効化されます。

```
>>> turtle.setundobuffer(42)
```

`turtle.undobufferentries()`

アンドゥバッファのエントリー数を返します。

```
>>> while undobufferentries():
...     undo()
```

`turtle.tracer(flag=None, delay=None)`

対応する TurtleScreen のメソッドの複製です。

バージョン 2.6 で非推奨.

`turtle.window_width()`

`turtle.window_height()`

どちらも対応する TurtleScreen のメソッドの複製です。

バージョン 2.6 で非推奨.

合成形の使用に関する補遺

合成されたタートルの形、つまり幾つかの色の違う多角形から成るような形を使うには、以下のように補助クラス *Shape* を直接使わなければなりません:

1. タイプ "compound" の空の Shape オブジェクトを作ります。
2. `addcomponent()` メソッドを使って、好きなだけここにコンポーネントを追加します。

例えば:

```
>>> s = Shape("compound")
>>> poly1 = ((0,0), (10,-5), (0,10), (-10,-5))
>>> s.addcomponent(poly1, "red", "blue")
>>> poly2 = ((0,0), (10,-5), (-10,-5))
>>> s.addcomponent(poly2, "blue", "red")
```

3. こうして作った Shape を Screen の形のリスト (shapelist) に追加して使います:

```
>>> register_shape("myshape", s)
>>> shape("myshape")
```

注釈: *Shape* クラスは *register_shape()* の内部では違った使われ方をします。アプリケーションを書く人が *Shape* クラスを扱わなければならないのは、上で示したように合成された形を使うとき だけ です！

24.5.4 TurtleScreen/Screen のメソッドと対応する関数

この節のほとんどの例では `screen` という名前の *TurtleScreen* インスタンスを使います。

ウィンドウの制御

`turtle.bgcolor(*args)`

パラメータ **args** – 色文字列または 0 から `colormode` の範囲の数 3 つ、またはそれを三つ組みにしたもの

TurtleScreen の背景色を設定するかまたは返します。

```
>>> screen.bgcolor("orange")
>>> screen.bgcolor()
'orange'
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()
(128, 0, 128)
```

`turtle.bgpic(picname=None)`

パラメータ **picname** – 文字列で gif ファイルの名前 `"nopic"`、または `None`

背景の画像を設定するかまたは現在の背景画像 (`backgroundimage`) の名前を返します。 *picname* がファイル名ならば、その画像を背景に設定します。 *picname* が `"nopic"` ならば、(もしあれば) 背景画像を削除します。 *picname* が `None` ならば、現在の背景画像のファイル名を返します。

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif"
```

`turtle.clear()`

`turtle.clearscreen()`

全ての図形と全てのタートルを *TurtleScreen* から削除します。そして空になった *TurtleScreen* をリセットして初期状態に戻します: 白い背景、背景画像もイベント束縛もなく、トレーシングはオンです。

注釈: この TurtleScreen メソッドはグローバル関数としては `clearscreen` という名前だけで使えます。グローバル関数 `clear` は Turtle メソッドの `clear` から派生した別ものです。

`turtle.reset()`

`turtle.resetScreen()`

スクリーン上の全てのタートルをリセットしその初期状態に戻します。

注釈: この TurtleScreen メソッドはグローバル関数としては `resetScreen` という名前だけで使えます。グローバル関数 `reset` は Turtle メソッドの `reset` から派生した別ものです。

`turtle.screensize` (*canvwidth=None, canvheight=None, bg=None*)

パラメータ

- **canvwidth** – 正の整数でピクセル単位の新しいキャンパス幅 (canvaswidth)
- **canvheight** – 正の整数でピクセル単位の新しいキャンパス高さ (canvasheight)
- **bg** – 色文字列または色タプルで新しい背景色

引数が渡されなければ、現在の (キャンパス幅, キャンパス高さ) を返します。そうでなければタートルが描画するキャンパスのサイズを変更します。描画ウィンドウには影響しません。キャンパスの隠れた部分を見るためにはスクロールバーを使って下さい。このメソッドを使うと、以前はキャンパスの外にあったそうした図形の一部を見えるようにすることができます。

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000,1500)
>>> screen.screensize()
(2000, 1500)
```

逃げ出してしまったタートルを探すためとかね ;-)

`turtle.setworldcoordinates` (*llx, lly, urx, ury*)

パラメータ

- **llx** – 数でキャンパスの左下隅の x-座標
- **lly** – 数でキャンパスの左下隅の y-座標
- **urx** – 数でキャンパスの右上隅の x-座標
- **ury** – 数でキャンパスの右上隅の y-座標

ユーザー定義座標系を準備し必要ならばモードを "world" に切り替えます。この動作は `screen.reset()` を伴います。すでに "world" モードになっていた場合、全ての図形は新しい座標に従って再描画されます。

重要なお知らせ: ユーザー定義座標系では角度が歪むかもしれません。

```
>>> screen.reset()
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...     left(10)
...
>>> for _ in range(8):
...     left(45); fd(2)    # a regular octagon
```

アニメーションの制御

`turtle.delay(delay=None)`

パラメータ **delay** – 正の整数

描画の遅延 (*delay*) をミリ秒単位で設定するかまたはその値を返します。(これは概ね引き続くキャンバス更新の時間間隔です。) 遅延が大きくなると、アニメーションは遅くなります。

オプション引数:

```
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

`turtle.tracer(n=None, delay=None)`

パラメータ

- **n** – 非負整数
- **delay** – 非負整数

タートルのアニメーションをオン・オフし、描画更新の遅延を設定します。 *n* が与えられた場合、通常のスクリーン更新のうち $1/n$ しか実際に実行されません。(複雑なグラフィックスの描画を加速するのに使えます。) 二つ目の引数は遅延の値を設定します (*delay()* も参照)。

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

`turtle.update()`

TurtleScreen の更新を実行します。トレーサーがオフの時に使われます。

RawTurtle/Turtle のメソッド *speed()* も参照して下さい。

スクリーンイベントを利用する

`turtle.listen (xdummy=None, ydummy=None)`

TurtleScreen に (キー・イベントを収集するために) フォーカスします。ダミー引数は `listen()` を `onclick` メソッドに渡せるようにするためのものです。

`turtle.onkey (fun, key)`

パラメータ

- **fun** – 引数なしの関数または `None`
- **key** – 文字列: キー (例 "a") またはキー・シンボル (例 "space")

`fun` を指定されたキーのキーリリース (key-release) イベントに束縛します。 `fun` が `None` ならばイベント束縛は除かれます。注意: キー・イベントを登録できるようにするためには TurtleScreen はフォーカスを持っていないとなりません (`listen()` を参照)。

```
>>> def f():
...     fd(50)
...     lt(60)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

`turtle.onclick (fun, btn=1, add=None)``turtle.onscreenclick (fun, btn=1, add=None)`

パラメータ

- **fun** – 2 引数の関数でキャンバスのクリックされた点の座標を引数として呼び出されるものです
- **btn** – マウスボタンの番号、デフォルトは 1 (左マウスボタン)
- **add** – `True` または `False` – `True` ならば、新しい束縛が追加されますが、そうでなければ、以前の束縛を置き換えます。

`fun` をタートルのマウスクリック (mouse-click) イベントに束縛します。 `fun` が `None` ならば、既存の束縛が取り除かれます。

Example for a `screen` という名の TurtleScreen インスタンスと `turtle` という名前の Turtle インスタンスの例:

```
>>> screen.onclick(turtle.goto) # Subsequently clicking into the TurtleScreen_
↪will
>>>                                     # make the turtle move to the clicked point.
>>> screen.onclick(None)             # remove event binding again
```

注釈: この TurtleScreen メソッドはグローバル関数としては `onscreenclick` という名前だけで使え

ます。グローバル関数 `onclick` は Turtle メソッドの `onclick` から派生した別ものです。

`turtle.ontimer (fun, t=0)`

パラメータ

- **fun** – 引数なし関数
- **t** – 数 ≥ 0

t ミリ秒後に `fun` を呼び出すタイマーを仕掛けます。

```
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...         screen.ontimer(f, 250)
>>> f()    ### makes the turtle march around
>>> running = False
```

設定と特殊なメソッド

`turtle.mode (mode=None)`

パラメータ **mode** – 文字列 "standard", "logo", "world" のいずれか

タートルのモード ("standard", "logo", "world" のいずれか) を設定してリセットします。モードが渡されなければ現在のモードが返されます。

モード "standard" は古い `turtle` 互換です。モード "logo" は Logo タートルグラフィックスとほぼ互換です。モード "world" はユーザーの定義した「世界座標 (world coordinates)」を使います。重要なお知らせ: このモードでは x/y 比が 1 でないと角度が歪むかもしれません。

モード	タートルの向きの初期値	正の角度
"standard"	右 (東) 向き	反時計回り
"logo"	上 (北) 向き	時計回り

```
>>> mode("logo")    # resets turtle heading to north
>>> mode()
'logo'
```

`turtle.colormode (cmode=None)`

パラメータ **cmode** – 1.0 か 255 のどちらかの値

色モード (colormode) を返すか、または 1.0 か 255 のどちらかの値に設定します。設定した後は、色トリプルの r, g, b 値は 0 から `cmode` の範囲になければなりません。

```
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
...
TurtleGraphicsError: bad color sequence: (240, 160, 80)
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

`turtle.getcanvas()`

この TurtleScreen の Canvas を返します。Tkinter の Canvas を使って何をするか知っている人には有用です。

```
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas instance at 0x...>
```

`turtle.getshapes()`

現在使うことのできる全てのタートルの形のリストを返します。

```
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

`turtle.register_shape(name, shape=None)`

`turtle.addshape(name, shape=None)`

この関数を呼び出す三つの異なる方法があります:

- (1) *name* が gif ファイルの名前で *shape* が None: 対応する画像の形を取り込みます。

```
>>> screen.register_shape("turtle.gif")
```

注釈: 画像の形はタートルが向きを変えても回転しませんので、タートルがどちらを向いているか見ても判りません!

- (2) *name* が任意の文字列で *shape* が座標ペアのタプル: 対応する多角形を取り込みます。

```
>>> screen.register_shape("triangle", ((5,-3), (0,5), (-5,-3)))
```

- (3) *name* が任意の文字列で *shape* が (合成形の) *Shape* オブジェクト: 対応する合成形を取り込みます。

タートルの形を TurtleScreen の形リスト (shapelist) に加えます。このように登録された形だけが `shape(shapename)` コマンドに使えます。

`turtle.turtles()`

スクリーン上のタートルのリストを返します。

```
>>> for turtle in screen.turtles():
...     turtle.color("red")
```

`turtle.window_height()`

タートルウィンドウの高さを返します。

```
>>> screen.window_height()
480
```

`turtle.window_width()`

タートルウィンドウの幅を返します。

```
>>> screen.window_width()
640
```

Screen 独自のメソッド、TurtleScreen から継承したもの以外

`turtle.bye()`

タートルグラフィックス (turtlegraphics) のウィンドウを閉じます。

`turtle.exitonclick()`

スクリーン上のマウスクリックに `bye()` メソッドを束縛します。

設定辞書中の `"using_IDLE"` の値が `False` (デフォルトです) の場合、さらにメインループ (mainloop) に入ります。注意: もし IDLE が `-n` スイッチ (サブプロセスなし) 付きで使われているときは、この値は `turtle.cfg` の中で `True` とされているべきです。この場合、IDLE のメインループもクライアントスクリプトから見てアクティブです。

`turtle.setup(width=_CFG["width"], height=_CFG["height"], startx=_CFG["leftright"], starty=_CFG["topbottom"])`

メインウィンドウのサイズとポジションを設定します。引数のデフォルト値は設定辞書に収められており、`turtle.cfg` ファイルを通じて変更できます。

パラメータ

- **width** – 整数ならばピクセル単位のサイズ、浮動小数点数ならばスクリーンに対する割合 (スクリーンの 50% がデフォルト)
- **height** – 整数ならばピクセル単位の高さ、浮動小数点数ならばスクリーンに対する割合 (スクリーンの 75% がデフォルト)
- **startx** – 正の数ならばスクリーンの左端からピクセル単位で測った開始位置、負の数ならば右端から、`None` ならば水平方向に真ん中
- **starty** – 正の数ならばスクリーンの上端からピクセル単位で測った開始位置、負の数ならば下端から、`None` ならば垂直方向に真ん中

```
>>> screen.setup (width=200, height=200, startx=0, starty=0)
>>>             # sets window to 200x200 pixels, in upper left of screen
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
>>>             # sets window to 75% of screen by 50% of screen and centers
```

`turtle.title (titlestring)`

パラメータ **titlestring** – タートルグラフィックスウィンドウのタイトルバーに表示される文字列

ウインドウのタイトルを *titlestring* に設定します。

```
>>> screen.title("Welcome to the turtle zoo!")
```

24.5.5 turtle モジュールのパブリッククラス

class `turtle.RawTurtle (canvas)`

class `turtle.RawPen (canvas)`

パラメータ **canvas** – `Tkinter.Canvas`, *ScrolledCanvas*, *TurtleScreen* のいずれか

タートルを作ります。タートルには上の「Turtle/RawTurtle のメソッド」で説明した全てのメソッドがあります。

class `turtle.Turtle`

`RawTurtle` のサブクラスで同じインターフェイスを持ちますが、最初に必要になったとき自動的に作られる *Screen* オブジェクトに描画します。

class `turtle.TurtleScreen (cv)`

パラメータ **cv** – `Tkinter.Canvas`

上で説明した `setbg()` のようなスクリーン向けのメソッドを提供します。

class `turtle.Screen`

`TurtleScreen` のサブクラスで 4 つのメソッドが加わっています。

class `turtle.ScrolledCanvas (master)`

パラメータ **master** – この `ScrolledCanvas` すなわちスクロールバーの付いた `Tkinter canvas` を収める `Tkinter` ウィジェット

タートルたちが遊び回る場所として自動的に `ScrolledCanvas` を提供する `Screen` クラスによって使われます。

class `turtle.Shape (type_, data)`

パラメータ **type_** – 文字列 "polygon", "image", "compound" のいずれか

形をモデル化するデータ構造。ペア (`type_`, `data`) は以下の仕様に従わなければなりません:

<i>type_</i>	<i>data</i>
"polygon"	多角形タプル、すなわち座標ペアのタプル
"image"	画像 (この形式は内部的にのみ使用されます!)
"compound"	None (合成形は <code>addcomponent()</code> メソッドを使って作らなければなりません)

addcomponent (*poly*, *fill*, *outline=None*)

パラメータ

- **poly** – 多角形、すなわち数のペアのタプル
- **fill** – *poly* を塗りつぶす色
- **outline** – *poly* のアウトラインの色 (与えられた場合)

例:

```
>>> poly = ((0,0), (10,-5), (0,10), (-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... add more components and then use register_shape()
```

合成形の使用に関する補遺 を参照。

class `turtle.Vec2D` (*x*, *y*)

2次元ベクトルのクラスで、タートルグラフィックスを実装するための補助クラス。タートルグラフィックスを使ったプログラムでも有用でしょう。タプルから派生しているので、ベクターはタプルです!

以下の演算が使えます (*a*, *b* はベクトル、*k* は数):

- *a* + *b* ベクトル和
- *a* - *b* ベクトル差
- *a* * *b* 内積
- *k* * *a* および *a* * *k* スカラー倍
- `abs(a)` *a* の絶対値
- `a.rotate(angle)` 回転

24.5.6 ヘルプと設定

ヘルプの使い方

`Screen` と `Turtle` クラスのパブリックメソッドはドキュメント文字列で網羅的に文書化されていますので、Python のヘルプ機能を通じてオンラインヘルプとして利用できます:

- IDLE を使っているときは、打ち込んだ関数/メソッド呼び出しのシグニチャとドキュメント文字列の一行目がツールチップとして表示されます。
- `help()` をメソッドや関数に対して呼び出すとドキュメント文字列が表示されます:

```
>>> help(Screen.bgcolor)
Help on method bgcolor in module turtle:

bgcolor(self, *args) unbound turtle.Screen method
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.

    >>> screen.bgcolor("orange")
    >>> screen.bgcolor()
    "orange"
    >>> screen.bgcolor(0.5,0,0.5)
    >>> screen.bgcolor()
    "#800080"

>>> help(Turtle.penup)
Help on method penup in module turtle:

penup(self) unbound turtle.Turtle method
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    >>> turtle.penup()
```

- メソッドに由来する関数のドキュメント文字列は変更された形をとります:

```
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.

    Example::

    >>> bgcolor("orange")
    >>> bgcolor()
    "orange"
    >>> bgcolor(0.5,0,0.5)
    >>> bgcolor()
```

(次のページに続く)

(前のページからの続き)

```

"#800080"

>>> help(penup)
Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    Example:
    >>> penup()

```

これらの変更されたドキュメント文字列はインポート時にメソッドから導出される関数定義と一緒に自動的に作られます。

ドキュメント文字列の翻訳

Screen と Turtle クラスのパブリックメソッドについて、キーがメソッド名で値がドキュメント文字列である辞書を作るユーティリティがあります。

```
turtle.write_docstringdict(filename="turtle_docstringdict")
```

パラメータ **filename** – ファイル名として使われる文字列

ドキュメント文字列辞書 (docstring-dictionary) を作って与えられたファイル名の Python スクリプトに書き込みます。この関数はわざわざ呼び出さなければなりません (タートルグラフィックスのクラスから使われることはありません)。ドキュメント文字列辞書は *filename.py* という Python スクリプトに書き込まれます。ドキュメント文字列の異なった言語への翻訳に対するテンプレートとして使われることを意図したものです。

もしあなたが (またはあなたの生徒さんが) *turtle* を自国語のオンラインヘルプ付きで使いたいならば、ドキュメント文字列を翻訳してできあがったファイルをたとえば *turtle.docstringdict-german.py* という名前で保存しなければなりません。

さらに *turtle.cfg* ファイルで適切な設定をしておけば、このファイルがインポート時に読み込まれて元の英語のドキュメント文字列を置き換えます。

この文書を書いている時点ではドイツ語とイタリア語のドキュメント文字列辞書が存在します。 (glingl@aon.at にリクエストして下さい。)

Screen および Turtle の設定方法

初期デフォルト設定では古い turtle の見た目と振る舞いを真似るようにして、互換性を最大限に保つようにしています。

このモジュールの特性を反映した、あるいは個々人の必要性 (たとえばクラスルームでの使用) に合致した、異なった設定を使いたい場合、設定ファイル `turtle.cfg` を用意してインポート時に読み込ませその設定に従わせることができます。

初期設定は以下の `turtle.cfg` に対応します:

```
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

いくつかピックアップしたエントリーの短い説明:

- 最初の 4 行は `Screen.setup()` メソッドの引数に当たります。
- 5 行目 6 行目は `Screen.screensize()` メソッドの引数に当たります。
- `shape` は最初から用意されている形ならどれでも使えます (`arrow`, `turtle` など)。詳しくは `help(shape)` をお試しください。
- 塗りつぶしの色 (`fillcolor`) を使いたくない (つまりタートルを透明にしたい) 場合、`fillcolor = ""` と書かなければなりません (しかし全ての空でない文字列は `cfg` ファイル中で引用符を付けてはいけません)。
- タートルにその状態を反映させるためには `resizemode = auto` とします。
- たとえば `language = italian` とするとドキュメント文字列辞書 (`docstringdict`) として `turtle.docstringdict-italian.py` がインポート時に読み込まれます (もしそれがインポートパス、たとえば `turtle` と同じディレクトリにあれば)。
- `exampleturtle` および `examplescreen` はこれらのオブジェクトのドキュメント文字列内での呼び名を決めます。メソッドのドキュメント文字列から関数のドキュメント文字列に変換する際に、これらの名前は取り除かれます。
- `using_IDLE`: IDLE とその `-n` スイッチ (サブプロセスなし) を常用するならば、この値を `True` に設定して下さい。これにより `exitonclick()` がメインループ (`mainloop`) に入るのを阻止します。

`turtle.cfg` ファイルは `turtle` の保存されているディレクトリと現在の作業ディレクトリに追加的に存在し得ます。後者が前者の設定をオーバーライドします。

Demo/turtle ディレクトリにも `turtle.cfg` ファイルがあります。デモを実際に (できればデモビューワからでなく) 実行してそこに書かれたものとその効果を学びましょう。

24.5.7 デモスクリプト

ソース配布物の Demo/turtle ディレクトリにデモスクリプト一式があります。

内容は以下の通りです:

- 新しい `turtle` モジュールの 15 の異なった特徴を示すデモスクリプト一式
- ソースコードを眺めつつスクリプトを実行できるデモビューワ `turtleDemo.py`。14 個が Examples メニューからアクセスできます。もちろんそれらを独立して実行することもできます。
- `turtledemo_two_canvases.py` は同時に二つのキャンバスを使用するデモです。これはビューワからは実行できません。
- `turtle.cfg` ファイルも同じディレクトリにあり、設定ファイルの書き方の例としても参考にできます。

デモスクリプトは以下の通りです:

名前	説明	特徴
bytedesign	複雑な古典的タートルグラフィックスパターン	<code>tracer()</code> , <code>delay</code> , <code>update()</code>
chaos	verhust 力学系のグラフ化, コンピュータの計算が常識的な予想に反する場合があることを示します。	世界座標系
clock	コンピュータの時間を示すアナログ時計	タートルが時計の針, <code>ontimer</code>
colormixer	r, g, b の実験	<code>ondrag()</code>
fractalcurves	Hilbert & Koch 曲線	再帰
lindenmayer	民俗的数学 (インド kolams)	L-システム
minimal_hanoi	ハノイの塔	ハノイ盤として正方形のタートル (<code>shape</code> , <code>shapsize</code>)
paint	超極小主義的描画プログラム	<code>onclick()</code>
peace	初歩的	<code>turtle</code> : 見た目とアニメーション
penrose	風と矢による非周期的タイリング	<code>stamp()</code>
planet_and_moon	重力系のシミュレーション	合成形, <code>Vec2D</code>
tree	(図形的) 幅優先木 (ジェネレータを使って)	<code>clone()</code>
wikipedia	タートルグラフィックスについての wikipedia の記事の例	<code>clone()</code> , <code>undo()</code>
yinyang	もう一つの初歩的な例	<code>circle()</code>

楽しんでね!

24.6 IDLE

IDLE は Python の統合開発環境で、学習用環境です。

IDLE は次のような特徴があります:

- `tkinter` GUI ツールキットを使って、100% ピュア Python でコーディングされています
- クロスプラットフォーム: Windows, Unix, Mac OS X で動作します
- コード入力、出力、エラーメッセージの色付け機能を持った Python shell (対話的インタプリタ) ウィンドウ
- 多段 Undo、Python 対応の色づけ、自動的な字下げ、呼び出し情報の表示、自動補完、他たくさんの機能をもつマルチウィンドウ・テキストエディタ
- 任意のウィンドウ内での検索、エディタウィンドウ内での置換、複数ファイルを跨いだ検索 (`grep`)
- 永続的なブレイクポイント、ステップ実行、グローバルとローカル名前空間の視覚化機能を持ったデバッガ
- 設定、ブラウザ群、ほかダイアログ群

24.6.1 メニュー

IDLE には 2 種類のメインウィンドウのタイプがあります。Shell ウィンドウ、Editor ウィンドウです。Editor ウィンドウは同時に複数起動できます。ファイル内の編集 (Edit) / 検索 (Find) で使われるような出力ウィンドウ (Output ウィンドウ) は edit ウィンドウのサブタイプで、Editor ウィンドウと同じトップメニューを持ちますが、デフォルトタイトルとコンテキストメニューが違います。

IDLE のメニューはどちらのウィンドウが現在選択されているかによって動的に変化します。このドキュメントでは、各々のメニューがどちらのウィンドウのタイプに関係するのかわかるようにしています。(訳注: *IDLE* は地域化されておらずメニューは全て英語のため、以降メニュー項目説明の見出しでは、*"item [訳語]"* の様式とします。)

File メニュー (Shell ウィンドウ、Editor ウィンドウ)

New File [新規ファイル] 新しいファイル編集ウィンドウを作成します。

Open... [開く...] Open ダイアログを使って既存のファイルをオープンします。

Recent Files [最近使ったファイル] 最近使ったファイルのリストを開きます。リストを一つクリックするとそれを開きます。

Open Module... [モジュールを開く...] 既存のモジュールをオープンします (`sys.path` を検索します)。

Class Browser [クラスブラウザ] 現在 Editor が開いているファイル内にあるクラス、関数、メソッドを木構造で可視化します。Shell からの場合は、先にモジュール選択のダイアログが開きます。

Path Browser [パスブラウザ] sys.path ディレクトリ、モジュール、クラスおよびメソッドを木構造で可視化します。

Save [保存] 現在のウィンドウに対応するファイルがあればそこに保存します。開いてから、または最後に保存したのちに編集があった場合のウィンドウには、ウィンドウタイトルの前後に * が付けられます。対応するファイルがなければ代わりに Save As が実行されます。

Save As... [名前を付けて保存...] Save As ダイアログを使って現在のウィンドウをセーブします。セーブされたファイルがウィンドウに新しく対応するファイルになります。

Save Copy As... [コピーとして保存...] 現在のウィンドウに対応するファイルを変えずに異なるファイルにセーブします。

Print Window [ウィンドウを印刷] 現在のウィンドウをデフォルトプリンタで印刷します。

Close [閉じる] 現在のウィンドウを閉じます (未保存の場合は保存するか質問します)。

Exit [終了] すべてのウィンドウを閉じて IDLE を終了します (未セーブの場合はセーブするか質問します)。

Edit メニュー (Shell ウィンドウ、Editor ウィンドウ)

Undo [元に戻す] 現在のウィンドウに対する最後の変更を Undo (取り消し) します。最大で 1000 個の変更が Undo できます。

Redo [やり直し] 現在のウィンドウに対する最後に undo された変更を Redo (再実行) します。

Cut [切り取り] システムのクリップボードへ選択された部分をコピーします。それから選択された部分を削除します。

Copy [コピー] 選択された部分をシステムのクリップボードへコピーします。

Paste [貼り付け] システムのクリップボードの内容をカレントウィンドウへ挿入します。

クリップボードの機能はコンテキストメニューからも使えます。

Select All [全て選択] カレントウィンドウの内容全体を選択します。

Find... [検索...] たくさんのオプションをもつ検索ダイアログボックスを開きます。

Find Again [再検索] 直前の検索があれば、それを繰り返します。

Find Selection [現在の選択を検索] 現在選択された文字列があれば、それを検索します。

Find in Files... [ファイルから検索...] ファイル検索ダイアログを開きます。結果を新しい出力ウィンドウに出力します。

Replace... [置換...] 検索と置換ダイアログを開きます。

Go to Line [指定行へジャンプ] カーソルを要求された行番号に移動し、その行が見えるようにします。

Show Completions [補完候補の一覧] キーワードや属性選択のリストを開きます。下の方に補完についての説明があります。

Expand Word [語の展開] 先頭だけタイプしたものを、同一ウィンドウ内の完全な語と合致するものに展開します。異なる展開を得るためには繰り返します。

Show call tip [呼び出し方ヒントの表示] 関数の開き括弧の後ろで、関数パラメータについてのヒントを表示する小さなウィンドウを開きます。

Show surrounding parens [囲んでいる括弧の強調] 囲んでいる括弧をハイライトします。

Format メニュー (Shell ウィンドウ、Editor ウィンドウ)

Indent Region [領域をインデント] 選択された行を右へインデント幅分シフトします (デフォルトは空白 4 個)。

Dedent Region [領域をインデント解除] 選択された行を左へインデント幅分シフトします (デフォルトは空白 4 個)。

Comment Out Region [領域をコメントアウト] 選択された行の先頭に ## を挿入します。

Uncomment Region [領域のコメントを解除] 選択された行から先頭の # あるいは ## を取り除きます。

Tabify Region [領域のタブ化] 先頭の一続きの空白をタブに置き換えます (注意: Python コードのインデントには 4 つの空白を使うことをお勧めします)。

Untabify Region [領域の非タブ化] すべてのタブを適切な数の空白に置き換えます。

Toggle Tabs [タブの切り替え] 字下げのために空白を使うかタブを使うかを切り替えるダイアログを開きます。

New Indent Width [新しいインデント幅] インデント幅を変更するダイアログを開きます。Python コミュニティによって受け容れられているデフォルトは空白 4 個です。

Format Paragraph [パラグラフのフォーマット] コメント内、マルチライン文字列リテラル内、あるいは選択行の現在位置の (空行区切り) パラグラフの再整形。パラグラフ内の全ての行は N カラム (デフォルトは 72) 以内で再整形されます。

Strip trailing whitespace [末尾の空白を取り除く] 行の末尾の最後の非空白文字以降の、任意の空白文字を削除します。

Run メニュー (Editor ウィンドウのみ)

Python Shell [Python シェル] Python Shell ウィンドウを開くか、起こします。

Check Module [モジュールのチェック] Editor ウィンドウでいま開いているモジュールを構文チェックします。モジュールが未保存の場合、*Options -> Configure IDLE -> General* の "Autosave Preferences" の設定にもとづき、確認を求められるか自動的に保存します。構文エラーが見つかったら Editor ウィンドウでそのおおよその位置に移動します。

Run Module [モジュールの実行] まず Check Module (上述) を実行し、エラーがなければ Shell の環境をクリアにして再スタートした上で、モジュールを実行します。出力は Shell ウィンドウに表示されます。この出力は `print` や `write` しない限り、ありません。モジュール実行が完了すると Shell はフォークスを戻してプロンプトを表示します。これにより対話的に実行結果を調べることが出来るでしょう。この機能はコマンドラインからファイルを `python -i file` で実行することに相当します。

Shell メニュー (Shell ウィンドウのみ)

View Last Restart [最後の再スタートを見る] 最後に Shell 再スタートした場所まで Shell ウィンドウをスクロールします。

Restart Shell [Shell の再スタート] shell を再スタートして環境を綺麗にします。

Interrupt Execution [実行の中断] プログラムの実行を停止します。

Debug メニュー (Shell ウィンドウのみ)

Go to File/Line [ファイル/行へ移動] カーソルのある現在行の上にファイル名と行番号が見つければ、(開かれていなければ) そのファイルを開いてその行に飛びます。例外のトレースバックが参照しているソース行を見るのにこれを使いましょう。そのようなファイル名・行番号表示をしている行を見つけるのには Find を使えます。この機能は Shell ウィンドウと Output ウィンドウのコンテキストメニューからも使えます。

Debugger [デバッガ] (トグル切り替え) アクティブにすると、Shell または Editor に入力したコードをデバッガのもとで実行します。Editor ではコンテキストメニューからブレイクポイントをセット出来ます。この機能はまだ不完全で、少々実験的です。

Stack Viewer [スタックビューア] 最後の例外のスタックトレースと locals 辞書、globals 辞書をツリーウィジットで表示します。

Auto-open Stack Viewer [スタックビューアの自動オープン] 未捕捉の例外時にスタックビューアを自動的に開くかどうかを切り替えます。

Options メニュー (Shell ウィンドウ、Editor ウィンドウ)

Configure IDLE [IDLE の設定] 設定ダイアログ (configuration dialog) を開き、好みの設定 (preference) を変更出来ます: fonts [フォント], indentation [字下げ/インデント], keybindings [キーバインド設定], text color themes [テキスト色テーマ], startup windows and size [IDLE 開始時についての設定 (Shell から開始するか Editor から開始するか、サイズ)], additional help sources [ヘルプの参照先追加], extensions [拡張の設定] (下記参照)。OS X ではアプリケーションメニューより Preferences を選択して設定ダイアログを開きます。古い IDLE で新しい組み込みの色テーマ (IDLE Dark) を使うには、それを新しいカスタムテーマとして保存してください。

デフォルトとは異なるユーザ設定は、ユーザのホームディレクトリの `.idlerc/` ディレクトリに保存されます。間違ったユーザ設定が問題になったらこの中のファイルを編集するなり削除するなりで直せます。

Code Context [コードコンテキスト] (トグル切り替え) (Editor ウィンドウのみ) Editor ウィンドウの一番上にペインが開きます。ここには、ウィンドウの一番上に見えているコードが属しているブロックコンテキストが表示されます。(訳注: 原文からは離れた訳をしていますが、いずれにしても文章だけでは伝わらないので実際に動かしてみてください。)

Window メニュー (Shell ウィンドウ、Editor ウィンドウ)

Zoom Height [ウィンドウの高さの切り替え] ウィンドウの高さを標準か最大かで切り替えます。 *Options -> Configure IDLE -> General* で設定を変えていない場合のデフォルトの初期状態のサイズは 80 カラム、40 行です。

このメニューの残りはすべての開いたウィンドウの名前の一覧になっています。一つを選ぶとそれを最前面に持ってきます (アイコン化されていれば元に戻します)。

Help メニュー (Shell ウィンドウ、Editor ウィンドウ)

About IDLE [IDLE について] バージョン、コピーライト、ライセンス、クレジット、その他を表示します。

IDLE Help [IDLE ヘルプ] IDLE のメニューオプション、基本的な編集やナビゲーションその他ヒントを詳しく書いた help ファイルを表示します。(訳注: 英語。内容は今ご覧になっているこのドキュメントと大差ないです。)

Python Docs [Python ドキュメント] Python ドキュメントがローカルにインストールされていればそれを開きます。もしくはウェブブラウザで <https://docs.python.org> 最新の Python ドキュメントを開きます。(訳注: いずれも何もしていなければ英語。下記参照。)

Turtle Demo [Turtle デモ] `turtledemo` モジュールを実行します。 `turtle` グラフィックスのサンプルをソースコードとともに見ることが出来ます。(訳注: これが使えるかどうかはインストール依存だと思えます。Windows の Python 2.7 標準インストールでは使えません。(3.4 では使える。)) なお turtle デモはソース配布物の `Demo/turtle` に含まれています。IDLE から起動するのはその中のメインモジュール `turtleDemo.py`。)

追加のヘルプソースを *Options -> Configure IDLE -> General -> Additional Help Source* で追加出来ます。(訳注: 日本語翻訳プロジェクトのもの <https://docs.python.jp/> を追加しておくといいでしょう。これをする *Help* メニューに翻訳ドキュメントに飛ぶアイテムが追加されます。)

コンテキストメニュー

ウィンドウ内で右クリック (OS X では Control-クリック) でコンテキストメニューが開きます。コンテキストメニューには Edit メニューにもある標準的なクリップボード機能が含まれています。

Cut [切り取り] システムのクリップボードへ選択された部分をコピーします。それから選択された部分を削除します。

Copy [コピー] 選択された部分をシステムのクリップボードへコピーします。

Paste [貼り付け] システムのクリップボードの内容をカレントウィンドウへ挿入します。

Editor ウィンドウではさらにブレイクポイント機能が使えます。ブレイクポイントがセットされた行は目立つように印がつきます。ブレイクポイントはデバッガのもとでの実行にだけ影響します。ファイルに付けたブレイクポイントはユーザの `.idlerc/` ディレクトリに保存されます。

Set Breakpoint [ブレイクポイントのセット] 現在行にブレイクポイントをセットします。

Clear Breakpoint [ブレイクポイントのクリア] その行のブレイクポイントをクリアします。

Shell、Output ウィンドウには以下があります。

Go to file/line [ファイル/行へ移動] Debug メニューと同じものです。

24.6.2 編集とナビゲーション

ここでの説明で 'C' は、Windows と Unix の場合は Control キー、Mac OSX では Command キーを示します。

- Backspace は左側を削除し、Del は右側を削除します。
- C-Backspace は語単位で左側を削除、C-Del は語単位で右側を削除します。
- 矢印キーと Page Up/Page Down はそれぞれその通りに移動します。
- C-LeftArrow と C-RightArrow は語単位で移動します。
- Home/End は行の始め/終わりへ移動します。
- C-Home/C-End はファイルの始め/終わりへ移動します。
- いくつかの有用な Emacs バインディングが Tcl/Tk から継承されています:
 - C-a で行頭へ移動。
 - C-e で行末へ移動。
 - C-k で行を削除 (ただしクリップボードには入りません)。
 - C-l で挿入ポイントをウィンドウの中心にする。
 - C-b go backward one character without deleting (usually you can also use the cursor key for this)
 - C-f で一文字分文字削除なしで進む (普通どおりカーソルキーでも出来ます)。
 - C-p で一行上へ移動 (普通どおりカーソルキーでも出来ます)。
 - C-d で次の文字を削除。

標準的なキーバインディング (C-c がコピーで C-v がペーストである、のような) は動くかもしれません。キーバインディングは Configure IDLE ダイアログで選択します。

自動的な字下げ

After a block-opening statement, the next line is indented by 4 spaces (in the Python Shell window by one tab). After certain keywords (break, return etc.) the next line is dedented. In leading indentation, Backspace deletes up to 4 spaces if they are there. Tab inserts spaces (in the Python Shell window one tab), number depends on Indent width. Currently, tabs are restricted to four spaces due to Tcl/Tk limitations.

edit メニューの indent/dedent region コマンドも参照してください。

補完 (Completions)

補完は、関数、クラス、クラスの属性について、組み込み型でもユーザ定義でも効きます。ファイル名でも使えます。

`.` や (文字列内で) `os.sep` 文字をタイプすると、定義済の遅延 (デフォルトでは 2 秒) の後 AutoCompleteWindow (ACW) が開きます。それらの文字をタイプ (あるいはそれに続けて数文字タイプ) して Tab キーを押すと、入力に続けられる候補が見つければ ACW はすぐに開きます。

入力についての補完候補がただ一つであれば、Tab は ACW を開かず補完します。

'Show Completions' will force open a completions window, by default the C-space will open a completions window. In an empty string, this will contain the files in the current directory. On a blank line, it will contain the built-in and user-defined functions and classes in the current namespaces, plus any modules imported. If some characters have been entered, the ACW will attempt to be more specific.

文字列がタイプされると ACW の選択が、それら入力に一番近いものにジャンプします。Editor ウィンドウや Shell ウィンドウでは、tab をタイプすれば、曖昧でない最長マッチするものを拾ってきます。ACW 内の選択状態ではリターンキーやダブルクリックで戻ってもいいですが tab を 2 回タイプしても良いです。ACW 内ではカーソルキーや PgUp/PgDn、マウス操作やスクロールバー操作も出来ます。

”隠し” 属性には `.` のあとで例えば `.` のような隠し属性名の開始文字を入力すればアクセス出来ます。これによって、モジュールの `__all__` やクラスプライベートな属性にアクセス出来ます。

Completion と 'Expand Word' 機能を活用して猛烈タイピングから卒業しましょう!

補完は現在のところ、名前空間内のものに制限されています。 `__main__` を介さない Editor ウィンドウ内や `sys.modules` 内の名前は見つけれられません。あなたのインポートとともにモジュールを一度実行すれば、この状況は正せます。なお、IDLE 自身は `sys.modules` 内のモジュールのかなりの数を使う (re モジュールなど) ため、デフォルトでかなりの数が見つかります (訳注: 動かしてみるとこの表現がわかりにくかったりします。 import とタイプしたときに候補モジュールに re が出てくるわけではないので)。

ACW の招かれざるポップアップがイヤだと仰るならば、設定で delay を長くしたり拡張自体を無効にすればよろしいです。

呼び出しヒント (Calltips)

calltip は、アクセス可能な 関数名に続けて (をタイプすると表示されます。名前の式ではドットや添え字を含んでいても構いません。calltip は、それをクリックするか、カーソルが引数のエリアから外れるか、) を

タイプするまでは表示されたままとなります。カーソルが引数部分にあれば、メニューやショートカットで calltip を表示できます。

calltip では関数シグニチャとドキュメンテーション文字列の最初の行を表示します。アクセスできるシグニチャがない組み込み関数の場合は、ドキュメンテーション文字列から、5 行目までの全行か最初の空行までの全行を表示します。いまのところ。

アクセス可能な 関数の集合は IDLE 自身がインポートしたものも含め、最後に IDLE を再スタートしてから既にユーザプロセス内で何がインポートされたのか、どの定義が実行されたのかに拠ります。

たとえば Shell を再スタートして (訳注: `itertools` をインポートせずに) `itertools.count()` とタイプしてみてください。calltip が呼ばれて飛び出ると思いますが、これは IDLE が `itertools` を自身の使用のためにユーザプロセス内にインポートするからです。(バージョンによりますよ、無論)。`turtle.write()` では何も出ないでしょう。IDLE は `turtle` をインポートしませんから。メニューやショートカットも、ウンともスンとも言いません。`import turtle` してからなら `turtle.write()` でも calltip が出ます。

Editor 内では `import` 文そのものだけでは、一度でもそのファイルを実行しない限り calltip に影響しません。ファイルの上の方でインポート文を書いたら実行するとか、あるいは既存のものは編集前に実行してしまえ、というのも良いかもしれませんね。

Python Shell ウィンドウ

- `C-c` で実行中のコマンドを中断します。
- `C-d` でファイル終端 (end-of-file) を送り、`>>>` プロンプトでタイプしていた場合はウィンドウを閉じます。
- `Alt-/` (語を展開します) もタイピングを減らすのに便利です。

コマンド履歴

- `Alt-p` は、以前のコマンドから検索します。OS X では `C-p` を使ってください。
- `Alt-n` は、次を取り出します。OS X では `C-n` を使ってください。
- `Return` は、以前のコマンドを取り出しているときは、そのコマンドを取り出します。

テキストの色

IDLE はデフォルトで白背景に黒字に、特別な意味を持った色付きテキストを使います。Shell では shell 出力、shell エラー、ユーザエラー。Python コードについては Shell プロンプト内や Editor でのキーワード、組み込みクラスや組み込み関数の名前、`class` や `def` に続く名前、文字列、そしてコメント。また一般のテキストウィンドウではカーソル (あれば)、検索で合致したテキスト (あれば)、そして選択されているテキストです。

このテキストの色付けはバックグラウンドで行われます。ので、たまーに色が付いてない状態が見えてしまいます。カラースキームを変えるには Configure IDLE [IDLE の設定] ダイアログの Highlighting タブで出来ます。

24.6.3 スタートアップとコードの実行

`-s` オプションとともに起動すると、IDLE は環境変数 `IDLESTARTUP` か `PYTHONSTARTUP` で参照されているファイルを実行します。IDLE はまず `IDLESTARTUP` をチェックし、あれば参照しているファイルを実行します。`IDLESTARTUP` が無ければ、IDLE は `PYTHONSTARTUP` をチェックします。これらの環境変数で参照されているファイルは、IDLE シェルでよく使う関数を置いたり、一般的なモジュールの `import` 文を実行するのに便利です。

加えて、Tk もスタートアップファイルがあればそれをロードします。その Tk のファイルは無条件にロードされることに注意してください。このファイルは `.Idle.py` で、ユーザーのホームディレクトリから探されます。このファイルの中の文は Tk の名前空間で実行されるので、IDLE の Python シェルで使う関数を `import` するのには便利ではありません。

コマンドラインの使い方

```
idle.py [-c command] [-d] [-e] [-h] [-i] [-r file] [-s] [-t title] [-] [arg] ...

-c command    run command in the shell window
-d            enable debugger and open shell window
-e            open editor window
-h            print help message with legal combinations and exit
-i            open shell window
-r file       run file in shell window
-s            run $IDLESTARTUP or $PYTHONSTARTUP first, in shell window
-t title      set title of shell window
-            run stdin in shell (- must be last option before args)
```

引数がある場合 (訳注: 以下の説明、たぶん実情に反してますが一応訳しています):

- `-`, `-c`, `-r` のどれかを使う場合、全ての引数は `sys.argv[1:...]` に入り、`sys.argv[0]` には `''`, `'-c'`, `'-r'` の、与えたものが入ります。Options ダイアログでデフォルトだったとしても Editor ウィンドウが開くことはありません。
- これ以外の場合は引数は編集対象のファイルとして開かれて、`sys.argv` には IDLE そのものに渡された引数が反映されます。

IDLE とコンソールの違い

As much as possible, the result of executing Python code with IDLE is the same as executing the same code in a console window. However, the different interface and operation occasionally affect visible results. For instance, `sys.modules` starts with more entries.

IDLE はまた、`sys.stdin`, `sys.stdout`, `sys.stderr` を、シェルウィンドウから入力を受け取り、また出力をシェルウィンドウに送るオブジェクトと置き換えます。このウィンドウは、フォーカスを持っている間キーボードと画面をコントロールします。これは通常透過的ですが、キーボードと画面に直接アクセスする機能は動きません。もし `sys` が `reload(sys)` によってリセットされたら、IDLE の変更が失われて、`input`, `raw_input`, `print` のようなものは正しく動きません。

IDLE の Shell では、ユーザーは完全な文を入力し、編集し、呼び戻すことができます。一度に一つきりの物理行でしか動作しないコンソールもありますが、IDLE は、`exec` を使用してそれぞれの文 (statement) を実行します。その結果、それぞれの文に対して毎回 `'__builtins__'` が定義されます。

サブプロセスを起こさずに起動する

デフォルトでは、IDLE はユーザコードを分離されたサブプロセスで、ソケット経由で実行します。これは内部的なループバックインターフェイスを使います。この接続は外部からは見えませんし、インターネットにデータを送信したり受信したりといったことはしません。ファイアウォールソフトウェアが何か文句を言ってくるても、放っておいて大丈夫です。

If the attempt to make the socket connection fails, Idle will notify you. Such failures are sometimes transient, but if persistent, the problem may be either a firewall blocking the connection or misconfiguration of a particular system. Until the problem is fixed, one can run Idle with the `-n` command line switch.

IDLE を `-n` で開始すれば、IDLE は単独プロセス内で動作し、RPC Python 実行サーバを走らせるサブプロセスを作りません。これは、あなたのプラットフォームで Python がサブプロセスや RPC ソケットインターフェイスを作れないなら有用かもしれないです。ですけれども、このモードはユーザコードが IDLE 自身から隔離されていませんし、Run/Run Module (F5) 選択時に環境がまっさらでやり直しにもなりません。コードを変更したら影響するモジュールを `reload()` してやらないといけませんし、ある種のもの (`from foo import baz` など) は再インポートしてやらないといけません。これらの理由から、可能なら常にデフォルトのサブプロセスを起こすモードで IDLE を起動するのが吉です。

バージョン 3.4 で非推奨.

24.6.4 ヘルプとお好み設定

Additional help sources [ヘルプ参照先の追加]

IDLE には "Python Docs" なるメニューエントリがあります。これはチュートリアルを含む大掛かりなヘルプを開きます。これは <https://docs.python.org> から入手出来ます。Configure IDLE ダイアログでいつでもヘルプメニューにヘルプ参照先の URL を追加したり削除したり出来ます。詳しくは *Help* メニュー (*Shell* ウィンドウ、*Editor* ウィンドウ) をご覧ください。

Setting preferences [お好み設定]

The font preferences, highlighting, keys, and general preferences can be changed via Configure IDLE on the Option menu. Keys can be user defined; IDLE ships with four built-in key sets. In addition, a user can create a custom key set in the Configure IDLE dialog under the keys tab.

Extensions [拡張]

IDLE contains an extension facility. Preferences for extensions can be changed with Configure Extensions. See the beginning of `config-extensions.def` in the `idlelib` directory for further information. The default extensions are currently:

- `FormatParagraph`
- `AutoExpand`
- `ZoomHeight`
- `ScriptBinding`
- `CallTips`
- `ParenMatch`
- `AutoComplete`
- `CodeContext`
- `RstripExtension`

24.7 他のグラフィカルユーザインタフェースパッケージ

主要なクロスプラットフォーム (Windows, Mac OS X, Unix 系) GUI ツールキットで Python でも使えるものは:

参考:

PyGTK は **GTK** ウィジェットセットのための一連のバインディングです。C のものより少しだけ高レベルなオブジェクト指向インタフェースを提供します。Tkinter が提供するよりも沢山のウィジェットがあり、Python に特化した参考資料も良いものがあります。GNOME に対しても、バインディングがあります。オンラインチュートリアルが手に入ります。

PyQt PyQt は **sip** でラップされた Qt ツールキットへのバインディングです。Qt は Unix、Windows および Mac OS X で利用できる大規模な C++ GUI ツールキットです。sip は Python クラスとして C++ ライブラリに対するバインディングを生成するためのツールキットで、特に Python 用に設計されています。PyQt3 バインディング向けの書籍に Boudewijn Rempt 著 *GUI Programming with Python: QT Edition* があります。PyQt4 向けにも Mark Summerfield 著 *Rapid GUI Programming with Python and Qt* があります。

wxPython wxPython はクロスプラットフォームの Python 用 GUI ツールキットで、人気のある wxWidgets (旧名 wxWindows) C++ ツールキットに基づいて作られています。このツールキットは Windows, Mac OS X および Unix システムのアプリケーションに、それぞれのプラットフォームのネイティブなウィジェットを可能な限りに利用して (Unix 系のシステムでは GTK+)、ネイティブなルック&フィールを提供します。多彩なウィジェットの他に、オンラインドキュメントや場面に応じたヘルプ、印刷、HTML 表示、低級デバイスコンテキスト描画、ドラッグ&ドロップ、システムクリップボードへのアクセス、XML に基づいたリソースフォーマット、さらにユーザ寄贈のモジュールからなる成長し続けているライブラリ等々を wxPython は提供しています。wxPython を扱った書籍として Noel Rappin、Robin Dunn 著 *wxPython in Action* があります。

PyGTK、PyQt および wxPython は全て現代的なルック&フィールを具え Tkinter より豊富なウィジェットがあります。これらに加えて、他にも Python 用 GUI ツールキットが、クロスプラットフォームのもの、ブラッ

トフォーム固有のものを含め、沢山あります。Python Wiki の [GUI Programming](#) ページも参照してください。もっとずっと完全なリストや、GUI ツールキット同士の比較をしたドキュメントへのリンクがあります。

第 25 章

開発ツール

この章で紹介されるモジュールはソフトウェアを書くことを支援します。たとえば、`pydoc` モジュールはモジュールの内容からドキュメントを生成します。`doctest` と `unittest` モジュールでは、自動的に実行して予想通りの出力が生成されるか確認するユニットテストを書くことができます。`2to3` は Python2.x 用のソースコードを正当な Python 3.x コードに翻訳できます。

この章で解説されるモジュールのリスト:

25.1 `pydoc` — ドキュメント生成とオンラインヘルプシステム

バージョン 2.1 で追加.

ソースコード: [Lib/pydoc.py](#)

`pydoc` モジュールは、Python モジュールから自動的にドキュメントを生成します。生成されたドキュメントをテキスト形式でコンソールに表示したり、Web ブラウザにサーバとして提供したり、HTML ファイルとして保存したりできます。

モジュール、クラス、関数、メソッドについての表示されるドキュメンテーションは、オブジェクトの docstring (つまり、`__doc__` 属性) に基き、また、そのドキュメント可能なメンバが再帰的に続きます。docstring がない場合、`pydoc` は、クラス、関数、メソッドについてはその定義の直前の、モジュールについてはファイル先頭の、ソースファイルのコメント行のブロックから記述を取得しようと試みます (`inspect.getcomments()` を参照してください)。

組み込み関数の `help()` を使うことで、対話型のインタプリタからオンラインヘルプを起動することができます。コンソール用のテキスト形式のドキュメントをつくるのにオンラインヘルプでは `pydoc` を使っています。`pydoc` を Python インタプリタからではなく、オペレーティングシステムのコマンドプロンプトから起動した場合でも、同じテキスト形式のドキュメントを見ることができます。例えば、以下の実行を

```
pydoc sys
```

shell から行くと `sys` モジュールのドキュメントを、Unix の `man` コマンドのような形式で表示させることができます。`pydoc` の引数として与えることができるのは、関数名・モジュール名・パッケージ名、また、

モジュールやパッケージ内のモジュールに含まれるクラス・メソッド・関数へのドット”.”形式での参照です。`pydoc` への引数がパスと解釈されるような場合で (オペレーティングシステムのパス区切り記号を含む場合です。例えば Unix ならば `"/` (スラッシュ) 含む場合になります)、さらに、そのパスが Python のソースファイルを指しているなら、そのファイルに対するドキュメントが生成されます。

注釈: オブジェクトとそのドキュメントを探すために、`pydoc` はドキュメント対象のモジュールを `import` します。そのため、モジュールレベルのコードはそのときに実行されます。`if __name__ == '__main__':` ガードを使って、ファイルがスクリプトとして実行したときのみコードを実行し、`import` されたときには実行されないようにして下さい。

コンソールへの出力時には、`pydoc` は読みやすさのために出力をページ化しようと試みます。環境変数 `PAGER` がセットされていれば `pydoc` はその値で設定されたページ化プログラムを使います。

引数の前に `-w` フラグを指定すると、コンソールにテキストを表示させるかわりにカレントディレクトリに HTML ドキュメントを生成します。

引数の前に `-k` フラグを指定すると、引数をキーワードとして利用可能な全てのモジュールの概要を検索します。検索のやりかたは、Unix の `man` コマンドと同様です。モジュールの概要というのは、モジュールのドキュメントの一行目のことです。

また、`pydoc` を使うことでローカルマシンにウェブブラウザから閲覧可能なドキュメントを提供する HTTP サーバーを起動することもできます。`pydoc -p 1234` とすると、HTTP サーバーをポート 1234 で起動します。これで、好きなウェブブラウザを使って `http://localhost:1234/` からドキュメントを見ることができます。`pydoc -g` はサーバーを起動したうえで、検索用に小さい `Tkinter` ベースの GUI を表示します。

`pydoc` でドキュメントを生成する場合、その時点での環境とパス情報に基づいてモジュールがどこにあるのが決定されます。そのため、`pydoc spam` を実行した場合につくられるドキュメントは、Python インタプリタを起動して `import spam` と入力したときに読み込まれるモジュールに対するドキュメントになります。

コアモジュールのドキュメントは <https://docs.python.org/library/> にあると仮定されています。これは、ライブラリリファレンスマニュアルを置いている異なる URL かローカルディレクトリを環境変数 `PYTHONDOSCS` に設定することでオーバーライドすることができます。

25.2 doctest — 対話的な実行例をテストする

`doctest` モジュールは、対話的 Python セッションのように見えるテキストを探し出し、セッションの内容を実行して、そこに書かれている通りに振舞うかを調べます。`doctest` は以下のような用途によく使われています:

- モジュールの `docstring` (ドキュメンテーション文字列) 中にある対話実行例のすべてが書かれている通りに動作するか検証することで、`docstring` の内容が最新かどうかチェックする。
- テストファイルやテストオブジェクト中の対話実行例が期待通りに動作するかを検証することで、回帰テストを実現します。

- 入出力例を豊富に使ったパッケージのチュートリアルドキュメントが書けます。入出力例と解説文のどちらに注目するかによって、ドキュメントは「読めるテスト」にも「実行できるドキュメント」にもなります。

以下に完全かつ短い実行例を示します:

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    If the result is small enough to fit in an int, return an int.
    Else return a long.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> [factorial(long(n)) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    2652528598121910586363084800000000L
    >>> factorial(30L)
    2652528598121910586363084800000000L
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    2652528598121910586363084800000000L

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

import math
if not n >= 0:
    raise ValueError("n must be >= 0")
```

(次のページに続く)

(前のページからの続き)

```
if math.floor(n) != n:
    raise ValueError("n must be exact integer")
if n+1 == n: # catch a value like 1e300
    raise OverflowError("n too large")
result = 1
factor = 2
while factor <= n:
    result *= factor
    factor += 1
return result

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

example.py をコマンドラインから直接実行すると、`doctest` はその魔法を働かせます:

```
$ python example.py
$
```

出力は何もありません！しかしこれが正常で、すべての実行例が正しく動作することを意味しています。スク립トに `-v` を与えると、`doctest` は何を行おうとしているのかを記録した詳細なログを出力し、最後にまとめを出力します:

```
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
Trying:
    [factorial(long(n)) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

といった具合で、最後には:

```
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
      ...
    OverflowError: n too large
```

(次のページに続く)

(前のページからの続き)

```
ok
2 items passed all tests:
  1 tests in __main__
  8 tests in __main__.factorial
9 tests in 2 items.
9 passed and 0 failed.
Test passed.
$
```

doctest の生産的な利用を始めるために知る必要があるのはこれだけです！さあやってみましょう。詳細な事柄は後続の各節ですべて説明しています。doctest の例は、標準の Python テストスイートやライブラリ中に沢山あります。標準のテストファイル `Lib/test/test_doctest.py` には、特に役に立つ例があります。

25.2.1 簡単な利用法: docstring 中の実行例をチェックする

doctest を試す簡単な方法 (とはいえ、いつもそうする必要はないのですが) は、各モジュール *M* の最後を、以下のようにして締めくくることです:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

こうすると、*doctest* は *M* 中の docstring を検査します。

モジュールをスクリプトとして実行すると、docstring 中の実行例が実行され、検証されます:

```
python M.py
```

docstring に書かれた実行例の実行が失敗しない限り、何も表示されません。失敗すると、失敗した実行例と、その原因が (場合によっては複数) 標準出力に印字され、最後に `***Test Failed*** N failures.` という行を出力します。ここで、*N* は失敗した実行例の数です。

一方、`-v` スイッチをつけて走らせると:

```
python M.py -v
```

実行を試みたすべての実行例について詳細に報告し、最後に各種まとめを行った内容が標準出力に印字されます。

`verbose=True` を *testmod()* に渡せば、詳細報告 (verbose) モードを強制できます。また、`verbose=False` にすれば禁止できます。どちらの場合にも、*testmod()* は `sys.argv` 上のスイッチを調べません。(したがって、`-v` をつけても効果はありません)。

Python 2.6 からは、*testmod()* を実行するコマンドラインショートカットもあります。Python インタプリタに doctest モジュールを標準ライブラリから直接実行して、テストするモジュール名をコマンドライン引数に与えます:


```
python -m doctest -v example.py
```

こうすると `example.py` を単体モジュールとしてインポートして、それに対して `testmod()` を実行します。このファイルがパッケージの一部で他のサブモジュールをそのパッケージからインポートしている場合はうまく動かないことに注意してください。

`testmod()` の詳しい情報は [基本 API](#) 節を参照してください。

25.2.2 簡単な利用法: テキストファイル中の実行例をチェックする

doctest のもう一つの簡単な用途は、テキストファイル中にある対話実行例に対するテストです。これには `testfile()` 関数を使います:

```
import doctest
doctest.testfile("example.txt")
```

この短いスクリプトは、`example.txt` というファイルの中に入っている対話モードの Python 操作例すべてを実行して、その内容を検証します。ファイルの内容は一つの巨大な docstring であるかのように扱われます; ファイルが Python プログラムである必要はありません! 例えば、`example.txt` には以下のような内容が入っているとします:

```
The ``example`` module
=====

Using ``factorial``
-----

This is an example text file in reStructuredText format.  First import
``factorial`` from the ``example`` module:

    >>> from example import factorial

Now use it:

    >>> factorial(6)
    120
```

`doctest.testfile("example.txt")` を実行すると、このドキュメント内のエラーを見つけ出します:

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

`testmod()` と同じく、`testfile()` は実行例が失敗しない限り何も表示しません。実行例が失敗すると、失敗した実行例とその原因が (場合によっては複数) `testmod()` と同じ書式で標準出力に書き出されます。

デフォルトでは、`testfile()` は自分自身を呼び出したモジュールのあるディレクトリを探します。その他の場所にあるファイルを見に行くように `testfile()` に指示するためのオプション引数についての説明は [基本 API](#) 節を参照してください。

`testmod()` と同様に `testfile()` の冗長性 (verbosity) はコマンドラインスイッチ `-v` またはオプションのキーワード引数 `verbose` によって指定できます。

Python 2.6 からは、`testfile()` を実行するコマンドラインショートカットもあります。Python インタプリタに `doctest` モジュールを標準ライブラリから直接実行して、テストするモジュール名をコマンドライン引数に与えます:

```
python -m doctest -v example.txt
```

ファイル名が `.py` で終わっていないので、`doctest` は `testmod()` ではなく `testfile()` を使って実行するのだと判断します。

`testfile()` の詳細は [基本 API](#) 節を参照してください。

25.2.3 doctest のからくり

この節では、doctest のからくり: どの docstring を見に行くのか、どのように対話実行例を見つけ出すのか、どんな実行コンテキストを使うのか、例外をどう扱うか、上記の振る舞いを制御するためにどのようなオプションフラグを使うか、について詳しく吟味します。こうした情報は、doctest に対応した実行例を書くために必要な知識です; 書いた実行例に対して実際に doctest を実行する上で必要な情報については後続の節を参照してください。

どの docstring が検証されるのか?

モジュールの docstring と、すべての関数、クラスおよびメソッドの docstring が検索されます。モジュールに `import` されたオブジェクトは検索されません。

加えて、`M.__test__` が存在し、“真の値を持つ” 場合、この値は辞書でなければならず、辞書の各エントリは (文字列の) 名前を関数オブジェクト、クラスオブジェクト、または文字列へとマップします。`M.__test__` から得られた関数およびクラスオブジェクトの docstring は、その名前がプライベートなものでも検索され、文字列の場合にはそれが docstring であるかのように扱われます。出力においては、`M.__test__` におけるキー `K` は、以下の名前で表示されます

```
<name of M>.__test__.K
```

検索中に見つかったクラスも同様に再帰的に検索が行われ、クラスに含まれているメソッドおよびネストされたクラスについて docstring のテストが行われます。

バージョン 2.4 で変更: “プライベート名” の概念は廃止されたため、ドキュメント化されなくなりました。

docstring 内の実行例をどのように認識するのか？

ほとんどの場合、対話コンソールセッション上でのコピー / ペーストはうまく動作します。とはいえ、*doctest* は特定の Python シェルの振る舞いを正確にエミュレーションしようとするわけではありません。

```
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print "yes"
... else:
...     print "no"
...     print "NO"
...     print "NO!!!"
...
no
NO
NO!!!
>>>
```

コードを含む最後の '`>>>`' または '`...`' 行の直下に期待する出力結果が置かれます。(出力結果がもしあれば) それは次の '`>>>`' 行か、すべて空白文字の行まで続きます。

詳細事項:

- 期待する出力結果には、空白だけの行が入ってはいけません。そのような行は期待する出力結果の終了を表すと見なされるからです。もし期待する出力結果の内容に空白行が入っている場合には、空白行が入るべき場所すべてに `<BLANKLINE>` を入れてください。

バージョン 2.4 で追加: `<BLANKLINE>` が追加されました; 以前のバージョンでは、空白行を含む出力結果を扱う方法がありませんでした。

- 全てのタブ文字は 8 カラムのタブ位置でスペースに展開されます。テスト対象のコードが作成した出力にあるタブは変更されません。出力例にあるどんなタブも展開されるので、コードの出力がハードタブを含んでいた場合、*doctest* が通るには `NORMALIZE_WHITESPACE` オプションか *directive* を有効にするしかありません。そうする代わりに、テストが出力を読み取りテストの一部として期待される値と正しく比較するように、テストを書き直すこともできます。このソース中のタブの扱いは試行錯誤の結果であり、タブの扱いの中で最も問題の起きにくいものだということが分かっています。タブの扱いについては、独自の `DocTestParser` クラスを実装して、別のアルゴリズムを使うこともできます。

バージョン 2.4 で変更: 新たにタブをスペースに展開するようになりました; 以前のバージョンはハードタブを保存しようとしていたので、混乱させるようなテスト結果になってしまっていました。

- 標準出力への出力は取り込まれますが、標準エラーは取り込まれません (例外発生時のトレースバックは別の方法で取り込まれます)。
- 対話セッションにおいて、バックスラッシュを用いて次の行に続ける場合や、その他の理由でバックスラッシュを用いる場合、raw docstring を使ってバックスラッシュを入力どおりに扱わせるようにしなければなりません:

```
>>> def f(x):
...     r'''Backslashes in a raw docstring: m\n'''
>>> print f.__doc__
Backslashes in a raw docstring: m\n
```

そうしない場合は、バックスラッシュは文字列の一部として解釈されます。例えば、上の `\n` は改行文字として解釈されてしまいます。それ以外の方法では、`doctest` にあるバックスラッシュを二重にする (そして raw string を使わない) という方法もあります:

```
>>> def f(x):
...     '''Backslashes in a raw docstring: m\\n'''
>>> print f.__doc__
Backslashes in a raw docstring: m\n
```

- 開始カラムはどこでもかまいません:

```
>>> assert "Easy!"
>>> import math
>>> math.floor(1.9)
1.0
```

期待する出力結果の先頭部にある空白文字列は、実行例の開始部分にあたる `'>>> '` 行の先頭にある空白文字列と同じだけ取り除かれます。

実行コンテキストとは何か?

デフォルトでは、`doctest` はテストを行うべき docstring を見つけるたびに `M` のグローバル名前空間の浅いコピーを使い、テストの実行によってモジュールの実際のグローバル名前空間を変更しないようにし、かつ `M` 内で行ったテストが痕跡を残して偶発的に別のテストを誤って動作させないようにしています。したがって、実行例中では `M` 内のトップレベルで定義されたすべての名前と、docstring が動作する以前に定義された名前を自由に使えます。個々の実行例は他の docstring 中で定義された名前を参照できません。

`testmod()` や `testfile()` に `globs=your_dict` を渡し、自前の辞書を実行コンテキストとして使うこともできます。

例外はどう扱えばよいか?

トレースバックが実行例によって生成される唯一の出力なら問題ありません。単にトレースバックを貼り付けてください。^{*1} トレースバックには、頻繁に変更されがちな情報 (例えばファイルパスや行番号など) が入っているものなので、これは受け入れるテスト結果に柔軟性を持たせようと `doctest` が苦勞している部分の一つです。

簡単な例を示しましょう:

^{*1} 期待する出力結果と例外の両方を含んだ例はサポートされていません。一方の終わりと他方の始まりを見分けようとするのはエラーの元になりがちですし、解りにくいテストになってしまいます。

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

この doctest は、`ValueError` が送出され、その詳細情報が `list.remove(x): x not in list` である場合に成功します。

例外が発生したときの期待する出力はトレースバックヘッダから始まっていなければなりません。トレースバックの形式は以下の二通りの行のいずれかで、実行例の最初の行と同じインデントでなければなりません:

```
Traceback (most recent call last):
Traceback (innermost last):
```

トレースバックヘッダの後ろにトレースバックスタックが続いてもかまいませんが、doctest はその内容を無視します。普通はトレースバックスタックを省略するか、対話セッションからそのままコピーしてきます。

トレースバックスタックの後ろにはもっとも有意義な部分、例外の型と詳細情報の入った行があります。これは通常トレースバックの最後の行ですが、例外が複数行の詳細情報を持っている場合、複数の行にわたることもあります:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: multi
    line
detail
```

上の例では、最後の 3 行 (`ValueError` から始まる行) における例外の型と詳細情報だけが比較され、それ以外の部分は無視されます。

バージョン 2.4 で変更: 以前のバージョンでは、複数行にわたる例外の詳細を扱えませんでした。

例外を扱うコツは、実行例をドキュメントとして読む上で明らかに価値のある情報でない限り、トレースバックスタックは省略する、ということです。したがって、先ほどの例は以下のように書くべきでしょう:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
...
ValueError: multi
    line
detail
```

トレースバックの扱いは非常に特殊なので注意してください。特に、上の書き直した実行例では、`...` の扱いは doctest の *ELLIPSIS* オプションとは独立しています。この例での省略記号は何かの省略を表しているかもしれませんが、コンマや数字が 3 個 (または 300 個) かもしれませんが、Monty Python のスキットをインデントして書き写したものかもしれません。

以下の詳細はずっと覚えておく必要はないのですが、一度目を通しておいってください:

- doctest は期待する出力の出所が print 文なのか例外なのかを推測できません。したがって、例えば期待する出力が `ValueError: 42 is prime` であるような実行例は、`ValueError` が実際に送出された場合と、万が一期待する出力と同じ文字列を print した場合の両方で成功してしまいます。現実的には、通常の出力がトレースバックヘッダから始まることはないので、実際に問題になることはないでしょう。
- トレースバックスタック (がある場合) の各行は、実行例の最初の行よりも深くインデントされているか、または 英数文字以外で始まっていなければなりません。トレースバックヘッダ以後に現れる行のうち、インデントが等しく英数文字で始まる最初の行は例外の詳細情報が書かれた行とみなされるからです。もちろん、本物のトレースバックでは正しく動作します。
- doctest のオプション `IGNORE_EXCEPTION_DETAIL` を指定した場合、最も左端のコロン以後の全ての内容と、例外名の中の全てのモジュール情報が無視されます。
- 対話シェルでは、`SyntaxError` の場合にトレースバックヘッダが省略されることがあります。しかし doctest にとっては、例外を例外でないものと区別するためにトレースバックヘッダが必要です。そこで、トレースバックヘッダを省略するような `SyntaxError` をテストする必要があるというごく稀なケースでは、実行例にトレースバックヘッダを手作業で追加する必要があるでしょう。
- `SyntaxError` の場合、Python は構文エラーの起きた場所を ^ マーカで表示します:

```
>>> 1 1
      File "<stdin>", line 1
        1 1
         ^
SyntaxError: invalid syntax
```

例外の型と詳細情報の前にエラー位置を示す行がくるため、doctest はこの行を調べません。例えば、以下の例では、間違った場所に ^ マーカを入れても成功してしまいます:

```
>>> 1 1
      File "<stdin>", line 1
        1 1
         ^
SyntaxError: invalid syntax
```

オプションフラグ

doctest の振る舞いは、数多くのオプションフラグによって様々な側面から制御されています。フラグのシンボル名はモジュールの定数として提供され、bit ごとの OR で合成して様々な関数に渡せます。シンボル名は *doctest directives* でも使用できます。

最初に説明するオプション群は、テストのセマンティクスを決めます。すなわち、実際にテストを実行したときの出力と実行例中の期待する出力とが一致しているかどうかを doctest がどのように判断するかを制御します:

`doctest.DONT_ACCEPT_TRUE_FOR_1`

デフォルトでは、期待する出力ブロックに単に 1 だけが入っており、実際の出力ブロックに 1 または

True だけが入っていた場合、これらの出力は一致しているとみなされます。0 と False の場合も同様です。 `DONT_ACCEPT_TRUE_FOR_1` を指定すると、こうした値の読み替えを行いません。デフォルトの挙動で読み替えを行うのは、最近の Python で多くの関数の戻り値型が整数型からブール型に変更されたことに対応するためです; 読み替えを行う場合、“通常の整数”の出力を期待する出力とするような doctest も動作します。このオプションはそのうちなくなるでしょうが、ここ数年はそのままでしょう。

`doctest.DONT_ACCEPT_BLANKLINE`

デフォルトでは、期待する出力ブロックに `<BLANKLINE>` だけの入った行がある場合、その行は実際の出力における空行に一致するようになります。完全な空行を入れてしまうと期待する出力がそこで終わっているとみなされてしまうため、期待する出力に空行を入れたい場合にはこの方法を使わなければなりません。 `DONT_ACCEPT_BLANKLINE` を指定すると、`<BLANKLINE>` の読み替えを行わなくなります。

`doctest.NORMALIZE_WHITESPACE`

このフラグを指定すると、連続する空白 (空白と改行文字) は互いに等価であるとみなします。期待する出力における任意の空白列は実際の出力における任意の空白と一致します。デフォルトでは、空白は厳密に一致しなければなりません。 `NORMALIZE_WHITESPACE` は、期待する出力の内容が非常に長いために、ソースコード中でその内容を複数行に折り返して書きたい場合に特に便利です。

`doctest.ELLIPSIS`

このフラグを指定すると、期待する出力中の省略記号マーカ (`...`) が実際の出力中の任意の部分文字列と一致するようになります。部分文字列は行境界にわたるものや空文字列を含みます。したがって、このフラグを使うのは単純な内容を対象にする場合にとどめましょう。複雑な使い方をすると、正規表現に `.*` を使ったときのように “しまった、マッチしすぎた! (match too much!)” と驚くことになりかねません。

`doctest.IGNORE_EXCEPTION_DETAIL`

このフラグを指定すると、期待する実行結果に例外が入るような実行例で、期待通りの型の例外が送出された場合に、例外の詳細情報が一致していなくてもテストが成功します。例えば、期待する出力が `ValueError: 42` であるような実行例は、実際に送出された例外が `ValueError: 3*14` でも成功しますが、`TypeError` が送出されるといった場合には成功しません。

また Python 3 の doctest レポートにあるモジュール名も無視します。従ってこのフラグを指定すると、テストを Python 2.7 と Python 3.2 (もしくはそれ以降) のどちらで行ったかに関係無く、どちらに対しても正しく動作します:

```
>>> raise CustomError('message')
Traceback (most recent call last):
CustomError: message

>>> raise CustomError('message')
Traceback (most recent call last):
my_module.CustomError: message
```

なお、`ELLIPSIS` を使っても例外メッセージの詳細を無視することができますが、モジュールの詳細が例外名の一部として表示されるかどうかには依存するようなテストは、やはり失敗します。また、`IGNORE_EXCEPTION_DETAIL` と Python 2.3 の詳細情報を使うことが、例外の詳細に影響されず、な

おかつ Python 2.3 以前の Python (これらのリリースは *doctest* ディレクティブをサポートせず、これらと無関係なコメントとして無視します) で成功する doctest を書くための、唯一の明確な方法です。例えば、例外の詳細情報は 2.4 で変更され、`"doesn't"` の代わりに `"does not"` と書くようになりましたが:

```
>>> (1, 2)[3] = 'moo'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object doesn't support item assignment
```

とすると、Python 2.3 や Python 2.4 以降の Python バージョンでテストを成功させることができます。

バージョン 2.7 で変更: *doctest* モジュールの *IGNORE_EXCEPTION_DETAIL* フラグが、テストされている例外を含むモジュールの名前を無視するようになりました。

`doctest.SKIP`

このフラグを指定すると、実行例は一切実行されません。こうした機能は *doctest* の実行例がドキュメントとテストを兼ねていて、ドキュメントのためには含めておかなければならないけれどチェックされなくても良い、というような文脈で役に立ちます。例えば、実行例の出力がランダムであるとか、テストドライバーには利用できないリソースに依存している場合などです。

SKIP フラグは一時的に実行例を”コメントアウト”するのにも使えます。

バージョン 2.5 で追加.

`doctest.COMPARISON_FLAGS`

上記の比較フラグすべての論理和をとったビットマスクです。

二つ目のオプション群は、テストの失敗を報告する方法を制御します:

`doctest.REPORT_UDIFF`

このオプションを指定すると、期待する出力および実際の出力が複数行になるときにテストの失敗結果を unified diff 形式を使って表示します。

`doctest.REPORT_CDIFF`

このオプションを指定すると、期待する出力および実際の出力が複数行になるときにテストの失敗結果を context diff 形式を使って表示します。

`doctest.REPORT_NDIFF`

このオプションを指定すると、期待する出力と実際の出力との間の差分を `difflib.Differ` を使って算出します。使われているアルゴリズムは有名な `ndiff.py` ユーティリティと同じです。これは、行単位の差分と同じように行内の差分にマーカーをつけられるようにする唯一の手段です。例えば、期待する出力のある行に数字の 1 が入っていて、実際の出力には 1 が入っている場合、不一致の起きているカラム位置を示すキャレットの入った行が一行挿入されます。

`doctest.REPORT_ONLY_FIRST_FAILURE`

このオプションを指定すると、各 *doctest* で最初にエラーの起きた実行例だけを表示し、それ以後の実行例の出力を抑制します。これにより、正しく書かれた実行例が、それ以前の実行例の失敗によっておかしくなってしまった場合に、*doctest* がそれを報告しないようになります。とはいえ、最初に失敗を引き起こした実行例とは関係なく誤って書かれた実行例の報告も抑制してしまいます。

`REPORT_ONLY_FIRST_FAILURE` を指定した場合、実行例がどこかで失敗しても、それ以後の実行例を続けて実行し、失敗したテストの総数を報告します; 出力が抑制されるだけです。

`doctest.REPORTING_FLAGS`

上記のエラー報告に関するフラグすべての論理和をとったビットマスクです。

バージョン 2.4 で追加: 定数 `DONT_ACCEPT_BLANKLINE`, `NORMALIZE_WHITESPACE`, `ELLIPSIS`, `IGNORE_EXCEPTION_DETAIL`, `REPORT_UDIFF`, `REPORT_CDIFF`, `REPORT_NDIFF`, `REPORT_ONLY_FIRST_FAILURE`, `COMPARISON_FLAGS`, および `REPORTING_FLAGS` が追加されました。

サブクラス化で `doctest` 内部を拡張するつもりがなければ役に立ちませんが、別の新しいオプションフラグ名を登録する方法もあります:

`doctest.register_optionflag(name)`

名前 `name` の新たなオプションフラグを作成し、作成されたフラグの整数値を返します。`register_optionflag()` は `OutputChecker` や `DocTestRunner` をサブクラス化して、その中で新たに作成したオプションをサポートさせる際に使います。`register_optionflag()` は以下のような定形文で呼び出さなければなりません:

```
MY_FLAG = register_optionflag('MY_FLAG')
```

バージョン 2.4 で追加.

ディレクティブ (Directives)

`doctest` ディレクティブは個々の実行例の **オプションフラグ** を操作するために使われます。`doctest` ティレクティブは後のソースコード例にあるような特殊な Python コメントです:

```
directive          ::=  "#" "doctest:" directive_options
directive_options  ::=  directive_option ("," directive_option)\*
directive_option    ::=  on_or_off directive_option_name
on_or_off           ::=  "+" \| "-"
directive_option_name ::=  "DONT_ACCEPT_BLANKLINE" \| "NORMALIZE_WHITESPACE" \| ...
```

+ や - とディレクティブオプション名の間に空白を入れてはなりません。ディレクティブオプション名は上で説明したオプションフラグ名のいずれかです。

ある実行例の `doctest` ディレクティブは、その実行例だけの `doctest` の振る舞いを変えます。ある特定の挙動を有効にしたければ + を、無効にしたければ - を使います。

例えば、以下のテストは成功します:

```
>>> print range(20)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

ディレクティブがない場合、実際の出力には一桁の数字の間に二つスペースが入っていないこと、実際の出力は1行になることから、テストは成功しないはずです。別のディレクティブを使って、このテストを成功させることもできます:

```
>>> print range(20)
[0, 1, ..., 18, 19]
```

複数のディレクティブは、一つの物理行の中にコンマで区切って指定できます:

```
>>> print range(20)
[0,      1, ...,      18,      19]
```

一つの実行例中で複数のディレクティブコメントを使った場合、それらは組み合わせられます:

```
>>> print range(20)
...
[0,      1, ...,      18,      19]
```

この実行例で分かるように、実行例にはディレクティブだけを含ま... 行を追加することができます。この書きかたは、実行例が長すぎるためにディレクティブを同じ行に入れると収まりが悪い場合に便利です:

```
>>> print range(5) + range(10,20) + range(30,40) + range(50,60)
...
[0, ..., 4, 10, ..., 19, 30, ..., 39, 50, ..., 59]
```

フォルトではすべてのオプションが無効になっており、ディレクティブは特定の実行例だけに影響を及ぼすので、通常意味があるのは有効にするためのオプション (+ のついたディレクティブ) だけです。とはいえ、doctest を実行する関数はオプションフラグを指定してデフォルトとは異なった挙動を実現できるので、そのような場合には - を使った無効化オプションも意味を持ちます。

バージョン 2.4 で追加: doctest ディレクティブが追加されました。

警告

doctest では、期待する出力に対する完全一致を厳格に求めます。一致しない文字が一文字でもあると、テストは失敗してしまいます。このため、Python が出力に関して何を保証していて、何を保証していないかを正確に知っていないと度々混乱させられることでしょう。例えば、辞書を出力する際、Python はキーと値のペアが常に特定の順番で並ぶよう保証してはいません。したがって、以下のようなテスト

```
>>> foo()
{"Hermione": "hippogryph", "Harry": "broomstick"}
```

は失敗するかもしれないのです! 回避するには

```
>>> foo() == {"Hermione": "hippogryph", "Harry": "broomstick"}
True
```

とするのが一つのやり方です。別のやり方は

```
>>> d = foo().items()
>>> d.sort()
>>> d
[('Harry', 'broomstick'), ('Hermione', 'hippogryph')]
```

他のやり方もありますが、あとは自分で考えてみてください。

以下のように、オブジェクトアドレスを埋め込むような結果を `print` するのもよくありません

```
>>> id(1.0) # certain to fail some of the time
7948648
>>> class C: pass
>>> C() # the default repr() for instances embeds an address
<__main__.C instance at 0x00AC18F0>
```

ELLIPSIS ディレクティブを使うと、上のような例をうまく解決できます:

```
>>> C()
<__main__.C instance at 0x...>
```

浮動小数点数もまた、プラットフォーム間での微妙な出力の違いの原因となります。というのも、Python は浮動小数点の書式化をプラットフォームの C ライブラリに委ねており、この点では、C ライブラリはプラットフォーム間で非常に大きく異なっているからです。

```
>>> 1./7 # risky
0.14285714285714285
>>> print 1./7 # safer
0.142857142857
>>> print round(1./7, 6) # much safer
0.142857
```

`I/2.**J` の形式になる数値はどのプラットフォームでもうまく動作するので、私はこの形式の数値を生成するように `doctest` の実行例を工夫しています:

```
>>> 3./4 # utterly safe
0.75
```

単純な分数は人間にとっても理解しやすく、良いドキュメントを書くために役に立ちます。

25.2.4 基本 API

数 `testmod()` と `testfile()` は、ほとんどの基本的な用途に十分な `doctest` インタフェースを提供しています。これら二つの関数についてあまり形式的でない入門が読みたければ、[簡単な利用法: `docstring` 中の実行例をチェックする](#) 節や [簡単な利用法: テキストファイル中の実行例をチェックする](#) 節を参照してください。

`doctest.testfile(filename[, module_relative][, name][, package][, globs][, verbose][, report][, options][, extraglobs][, raise_on_error][, parser][, encoding])`
`filename` 以外の引数はすべてオプションで、キーワード引数形式で指定しなければなりません。

filename に指定したファイル内にある実行例をテストします。(failure_count, test_count) を返します。

オプション引数の *module_relative* は、ファイル名をどのように解釈するかを指定します:

- *module_relative* が `True` (デフォルト) の場合、*filename* は OS に依存しないモジュールの相対パスになります。デフォルトでは、このパスは関数 `testfile()` を呼び出しているモジュールからの相対パスになります; ただし、*package* 引数を指定した場合には、パッケージからの相対になります。OS への依存性を除くため、*filename* ではパスを分割する文字に `/` を使わなければならない、絶対パスにしてはなりません (パス文字列を `/` で始めてはなりません)。
- *module_relative* が `False` の場合、*filename* は OS 依存のパスを示します。パスは絶対パスでも相対パスでもかまいません; 相対パスにした場合、現在の作業ディレクトリを基準に解決します。

オプション引数 *name* には、テストの名前を指定します; デフォルトの場合や `None` を指定した場合、`os.path.basename(filename)` になります。

オプション引数 *package* には、Python パッケージを指定するか、モジュール相対のファイル名の場合には相対の基準ディレクトリとなる Python パッケージの名前を指定します。パッケージを指定しない場合、関数を呼び出しているモジュールのディレクトリを相対の基準ディレクトリとして使います。*module_relative* を `False` に指定している場合、*package* を指定するとエラーになります。

オプション引数 *globs* には辞書を指定します。この辞書は、実行例を実行する際のグローバル変数として用いられます。doctest はこの辞書の浅いコピーを生成するので、実行例は白紙の状態からスタートします。デフォルトの場合や `None` を指定した場合、新たな空の辞書になります。

オプション引数 *extraglobs* には辞書を指定します。この辞書は、実行例を実行する際にグローバル変数にマージされます。マージは `dict.update()` のように振舞います: *globs* と *extraglobs* との間に同じキー値がある場合、両者を合わせた辞書中には *extraglobs* の方の値が入ります。デフォルト、または `None` であるとき、追加のグローバル変数は使われません。この仕様は、パラメータ付きで doctest を実行するという、やや進んだ機能です。例えば、一般的な名前を使って基底クラス向けに doctest を書いておき、その後で辞書で一般的な名前からテストしたいサブクラスへの対応付けを行う辞書を *extraglobs* に渡して、様々なサブクラスをテストできます。

オプション引数 *verbose* が真の場合、様々な情報を出力します。偽の場合にはテストの失敗だけを報告します。デフォルトの場合や `None` を指定した場合、`sys.argv` に `'-v'` を指定しない限りこの値は真になりません。

オプション引数 *report* が真の場合、テストの最後にサマリを出力します。それ以外の場合には何も出力しません。verbose モードの場合、サマリには詳細な情報を出力しますが、そうでない場合にはサマリはとても簡潔になります (実際には、すべてのテストが成功した場合には何も出力しません)。

オプション引数 *optionflags* は、各オプションフラグの論理和をとった値を指定します。 [オプションフラグ](#) 節を参照してください。

オプション引数 *raise_on_error* の値はデフォルトでは偽です。真にすると、最初のテスト失敗や予期しない例外が起きたときに例外を送出します。このオプションを使うと、失敗の原因を検死デバッグ (post-mortem debug) できます。デフォルトの動作では、実行例の実行を継続します。

オプション引数 `parser` には、`DocTestParser` (またはそのサブクラス) を指定します。このクラスはファイルから実行例を抽出するために使われます。デフォルトでは通常のパーザ (`DocTestParser()`) です。

オプション引数 `encoding` にはファイルをユニコードに変換する際に使われるエンコーディングを指定します。

バージョン 2.4 で追加。

バージョン 2.5 で変更: `encoding` パラメータが追加されました。

```
doctest.testmod([m][, name][, globs][, verbose][, report][, optionflags][, extraglobs][,
                 raise_on_error][, exclude_empty])
```

引数はすべてオプションで、`m` 以外の引数はキーワード引数として指定しなければなりません。

モジュール `m` (`m` を指定しないか `None` にした場合には `__main__`) から到達可能な関数およびクラスの `docstring` 内にある実行例をテストします。 `m.__doc__` 内の実行例からテストを開始します。

また、辞書 `m.__test__` が存在し、`None` でない場合、この辞書から到達できる実行例もテストします。 `m.__test__` は、(文字列の) 名前から関数、クラスおよび文字列への対応付けを行っています。関数およびクラスの場合には、その `docstring` 内から実行例を検索します。文字列の場合には、`docstring` と同じようにして実行例の検索を直接実行します。

モジュール `m` に属するオブジェクトにつけられた `docstring` のみを検索します。

(`failure_count`, `test_count`) を返します。

オプション引数 `name` には、モジュールの名前を指定します。デフォルトの場合や `None` を指定した場合には、`m.__name__` を使います。

オプション引数 `exclude_empty` はデフォルトでは偽になっています。この値を真にすると、`doctest` を持たないオブジェクトを考慮から外します。デフォルトの設定は依存のバージョンとの互換性を考えたハックであり、`doctest.master.summarize()` と `testmod()` を合わせて利用しているようなコードでも、テスト実行例を持たないオブジェクトから出力を得るようにしています。新たに追加された `DocTestFinder` のコンストラクタの `exclude_empty` はデフォルトで真になります。

オプション引数 `extraglobs`, `verbose`, `report`, `optionflags`, `raise_on_error`, および `globs` は上で説明した `testfile()` の引数と同じです。ただし、`globs` のデフォルト値は `m.__dict__` になります。

バージョン 2.3 で変更: `optionflags` パラメータが追加されました。

バージョン 2.4 で変更: パラメータ `extraglobs`, `raise_on_error`, `exclude_empty` が追加されました。

バージョン 2.5 で変更: オプション引数 `isprivate` は、2.4 では非推奨でしたが、廃止されました。

```
doctest.run_docstring_examples(f, globs[, verbose][, name][, compileflags][, optionflags])
```

オブジェクト `*f*` に関連付けられた実行例をテストします。 `*f*` は文字列、モジュール、関数、またはクラスオブジェクトです。

引数 `globs` に辞書を指定すると、その浅いコピーを実行コンテキストに使います。

オプション引数 `name` はテスト失敗時のメッセージに使われます。デフォルトの値は `'NoName'` です。

オプション引数 `verbose` の値を真にすると、テストが失敗しなくても出力を生成します。デフォルトでは、実行例のテストに失敗したときのみ出力を生成します。

オプション引数 `compileflags` には、実行例を実行するときに Python バイトコードコンパイラが使うフラグを指定します。デフォルトの場合や `None` を指定した場合、フラグは `globs` 内にある `future` 機能セットに対応したものになります。

オプション引数 `optionflags` は、上で述べた `testfile()` と同様の働きをします。

25.2.5 単体テスト API

doctest 化したモジュールのコレクションが増えるにつれ、すべての doctest をシステマティックに実行したいと思うようになるはずです。Python 2.4 以前の `doctest` には `Tester` というほとんどドキュメント化されていないクラスがあり、複数のモジュールの doctest を統合する初歩的な手段を提供していました。Tester は非力であり、実際のところ、もっときちんとした Python のテストフレームワークが `unittest` モジュールで構築されており、複数のソースコードからのテストを統合する柔軟な方法を提供しています。そこで Python 2.4 では `doctest` の `Tester` クラスを廃止し、モジュールや doctest の入ったテキストファイルから `unittest` テストスイートを作成できるような二つの関数を `doctest` 側で提供するようにしました。`unittest` のテストディスカバリと統合するには、テストモジュールに `load_tests()` 関数を書いておいてください:

```
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
    return tests
```

doctest の入ったテキストファイルやモジュールから `unittest.TestSuite` インスタンスを生成するための主な関数は二つあります:

`doctest.DocFileSuite(*paths, [module_relative][, package][, setUp][, tearDown][, globs][, optionflags][, parser][, encoding])`

単一または複数のテキストファイルに入っている doctest 形式のテストを、`unittest.TestSuite` インスタンスに変換します。

この関数の返す `unittest.TestSuite` インスタンスは、`unittest` フレームワークで動作させ、各ファイルの実行例を対話的に実行するためのものです。ファイル内の何らかの実行例の実行に失敗すると、この関数で生成した単体テストは失敗し、該当するテストの入っているファイルの名前と、(場合によりだいたい) 行番号の入った `failureException` 例外を送出します。

関数には、テストを行いたい一つまたは複数のファイルへのパスを (文字列で) 渡します。

`DocFileSuite()` には、キーワード引数でオプションを指定できます:

オプション引数 `module_relative` は `paths` に指定したファイル名をどのように解釈するかを指定します:

- `module_relative` が `True` (デフォルト) の場合、`paths` 中のファイル名は OS に依存しないモジュール

ルの相対パスになります。デフォルトでは、このパスは呼び出し元のモジュールのディレクトリからの相対パスになります; ただし、`package` 引数を指定した場合には、パッケージからの相対になります。OS への依存性を除くため、各ファイル名はパスを分割するのに / 文字を使わなければならない、絶対パスにしてはなりません (パス文字列を / で始めてはなりません)。

- `module_relative` が `False` の場合、`filename` は OS 依存のパスを示します。パスは絶対パスでも相対パスでもかまいません; 相対パスにした場合、現在の作業ディレクトリを基準に解決します。

オプション引数 `package` には、Python パッケージを指定するか、モジュール相対のファイル名の場合には相対の基準ディレクトリとなる Python パッケージの名前を指定します。パッケージを指定しない場合、関数を呼び出しているモジュールのディレクトリを相対の基準ディレクトリとして使います。`module_relative` を `False` に指定している場合、`package` を指定するとエラーになります。

オプション引数 `setUp` には、テストスイートのセットアップに使う関数を指定します。この関数は、各ファイルのテストを実行する前に呼び出されます。`setUp` 関数は `DocTest` オブジェクトに引き渡されます。`setUp` は `globals` 属性を介してテストのグローバル変数にアクセスできます。

オプション引数 `tearDown` には、テストを解体 (tear-down) するための関数を指定します。この関数は、各ファイルのテストの実行を終了するたびに呼び出されます。`tearDown` 関数は `DocTest` オブジェクトに引き渡されます。`tearDown` は `globals` 属性を介してテストのグローバル変数にアクセスできます。

オプション引数 `globals` は辞書で、テストのグローバル変数の初期値が入ります。この辞書は各テストごとに新たにコピーして使われます。デフォルトでは `globals` は空の新たな辞書です。

オプション引数 `optionflags` には、テストを実行する際にデフォルトで適用される `doctest` オプションを OR で結合して指定します。 [オプションフラグ](#) 節を参照してください。結果レポートに関するオプションを指定するより適切な方法は下記の `set_unittest_reportflags()` の説明を参照してください。

オプション引数 `parser` には、 `DocTestParser` (またはそのサブクラス) を指定します。このクラスはファイルから実行例を抽出するために使われます。デフォルトでは通常のパーザ (`DocTestParser()`) です。

オプション引数 `encoding` にはファイルをユニコードに変換する際に使われるエンコーディングを指定します。

バージョン 2.4 で追加。

バージョン 2.5 で変更: `DocFileSuite()` を使用してテキストファイルからロードされた doctests に提供される `globals` に、グローバル変数 `__file__` が追加されます。

バージョン 2.5 で変更: `encoding` パラメータが追加されました。

注釈: `testmod()` や `DocTestFinder` とは違い、`module` が docstring を含んでいない場合この関数は `ValueError` を送出します。 `exclude_empty` キーワード引数を `False` にした `DocTestFinder` インスタンスを、 `test_finder` 引数に渡すことでこのエラーの発生を防ぐことができます:

```
>>> finder = doctest.DocTestFinder(exclude_empty=False)
>>> suite = doctest.DocTestSuite(test_finder=finder)
```

```
doctest.DocTestSuite([module][, globs][, extraglobs][, test_finder][, setUp][, tearDown][,
                      checker])
```

doctest のテストを `unittest.TestSuite` に変換します。

この関数の返す `unittest.TestSuite` インスタンスは、`unittest` フレームワークで動作させ、モジュール内の各 `doctest` を実行するためのものです。何らかの `doctest` の実行に失敗すると、この関数で生成した単体テストは失敗し、該当するテストの入っているファイルの名前と、(場合によりだいたい) 行番号の入った `failureException` 例外を送出します。

オプション引数 `module` には、テストしたいモジュールの名前を指定します。`module` にはモジュールオブジェクトまたは (ドット表記の) モジュール名を指定できます。`module` を指定しない場合、この関数呼び出ししているモジュールになります。

オプション引数 `globs` は辞書で、テストのグローバル変数の初期値が入ります。この辞書は各テストごとに新たにコピーして使われます。デフォルトでは `globs` は空の新たな辞書です。

オプション引数 `extraglobs` には追加のグローバル変数セットを指定します。この変数セットは `globs` に統合されます。デフォルトでは、追加のグローバル変数はありません。

オプション引数 `test_finder` は、モジュールから `doctest` を抽出するための `DocTestFinder` オブジェクト (またはその代替となるオブジェクト) です。

オプション引数 `setUp`、`tearDown`、および `optionflags` は上の `DocFileSuite()` と同じです。

バージョン 2.3 で追加。

バージョン 2.4 で変更: `globs`, `extraglobs`, `test_finder`, `setUp`, `tearDown`, および `optionflags` パラメータが追加されました。また、この関数は `doctest` の検索に `testmod()` と同じテクニックを使うようになりました。

裏側では `DocTestSuite()` は `doctest.DocTestCase` インスタンスから `unittest.TestSuite` を作成しており、`DocTestCase` は `unittest.TestCase` のサブクラスになっています。`DocTestCase` についてはここでは説明しません (これは内部実装上の詳細だからです) が、そのコードを調べてみれば、`unittest` の組み込みの詳細に関する疑問を解決できるはずです。

同様に、`DocFileSuite()` は `doctest.DocFileCase` インスタンスから `unittest.TestSuite` を作成し、`DocFileCase` は `DocTestCase` のサブクラスになっています。

そのため、`unittest.TestSuite` クラスを生成するどちらの方法も `DocTestCase` のインスタンスを実行します。これは次のような微妙な理由で重要です: `doctest` 関数を自分で実行する場合、オプションフラグを `doctest` 関数に渡すことで、`doctest` のオプションを直接操作できます。しかしながら、`unittest` フレームワークを書いている場合には、いつどのようにテストを動作させるかを `unittest` が完全に制御してしまいます。フレームワークの作者はたいてい、`doctest` のレポートオプションを (コマンドラインオプションで指定するなどして) 操作したいと考えますが、`unittest` を介して `doctest` のテストランナーにオプションを渡す方法は存在しないのです。

このため、`doctest` では、以下の関数を使って、`unittest` サポートに特化したレポートフラグ表記方法もサポートしています:

`doctest.set_unittest_reportflags(flags)`

`doctest` のレポートフラグをセットします。

引数 `flags` にはオプションフラグを OR で結合して渡します。 [オプションフラグ](#) 節を参照してください。「レポートフラグ」しか使えません。

この関数で設定した内容はモジュール全体にわたるものであり、関数呼び出し以後に `unittest` モジュールから実行されるすべての `doctest` に影響します: `DocTestCase` の `runTest()` メソッドは、`DocTestCase` インスタンスが作成された際に、現在のテストケースに指定されたオプションフラグを見に行きます。レポートフラグが指定されていない場合 (通常の場合で、望ましいケースです)、`doctest` の `unittest` レポートフラグが bit ごとの OR で結合され、`doctest` を実行するために作成される `DocTestRunner` インスタンスに渡されます。`DocTestCase` インスタンスを構築する際に何らかのレポートフラグが指定されていた場合、`doctest` の `unittest` レポートフラグは無視されます。

この関数は、関数を呼び出す前に有効になっていた `unittest` レポートフラグの値を返します。

バージョン 2.4 で追加。

25.2.6 拡張 API

基本 API は、`doctest` を使いやすくするための簡単なラップであり、柔軟性があるほとんどのユーザの必要を満たしています; とはいえ、もっとテストをきめ細かに制御したい場合や、`doctest` の機能を拡張したい場合、拡張 API (advanced API) を使わなければなりません。

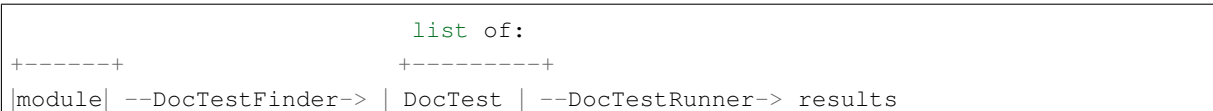
拡張 API は、`doctest` ケースから抽出した対話モードでの実行例を記憶するための二つのコンテナクラスを中心に構成されています:

- *Example*: 1 つの Python 文 (*statement*) と、その期待する出力をペアにしたもの。
- *DocTest*: *Example* の集まり。通常一つの docstring やテキストファイルから抽出されます。

その他に、`doctest` の実行例を検索、構文解析、実行、チェックするための処理クラスが以下のように定義されています:

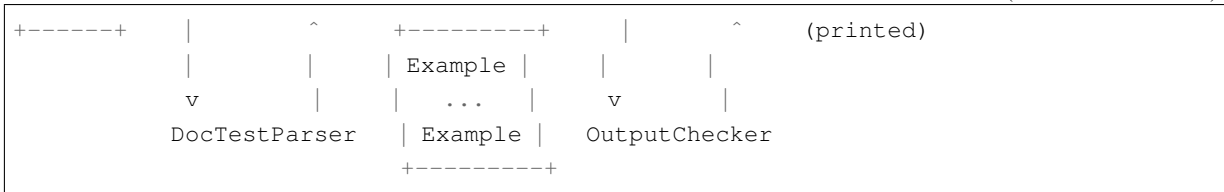
- *DocTestFinder*: 与えられたモジュールからすべての docstring を検索し、*DocTestParser* を使って対話モードでの実行例が入ったすべての docstring から *DocTest* を生成します。
- *DocTestParser*: (オブジェクトの docstring 等の) 文字列から *DocTest* オブジェクトを生成します。
- *DocTestRunner*: *DocTest* 内の実行例を実行し、*OutputChecker* を使って出力を検証します。
- *OutputChecker*: `doctest` 実行例から実際に出力された結果を期待する出力と比較し、両者が一致するか判別します。

これらの処理クラスの間関係を図にまとめると、以下のようになります:



(次のページに続く)

(前のページからの続き)



DocTest オブジェクト

class `doctest.DocTest` (*examples, globs, name, filename, lineno, docstring*)

単一の名前空間内で実行される `doctest` 実行例の集まりです。コンストラクタの引数は `DocTest` インスタンス中の同名の属性の初期化に使われます。

バージョン 2.4 で追加。

`DocTest` では、以下の属性を定義しています。これらの変数はコンストラクタで初期化されます。直接変更してはなりません。

examples

対話モードにおける実行例それぞれをエンコードしていて、テストで実行される、`Example` オブジェクトからなるリストです。

globs

実行例を実行する名前空間 (いわゆるグローバル変数) です。このメンバは、名前から値への対応付けを行っている辞書です。実行例が名前空間に対して (新たな変数を束縛するなど) 何らかの変更を行った場合、`globs` への反映はテストの実行後に起こります。

name

`DocTest` を識別する名前の文字列です。通常、この値はテストを取り出したオブジェクトかファイルの名前になります。

filename

`DocTest` を取り出したファイルの名前です; ファイル名が未知の場合や `DocTest` をファイルから取り出したのでない場合には `None` になります。

lineno

`filename` 中で `DocTest` のテスト実行例が始まっている行の行番号で、行番号が利用できなければ `None` です。行番号は、ファイルの先頭を 0 として数えます。

docstring

テストを取り出した `docstring` 自体を現す文字列です。docstring 文字列を得られない場合や、文字列からテスト実行例を取り出したのでない場合には `None` になります。

Example オブジェクト

class `doctest.Example` (*source, want[, exc_msg][, lineno][, indent][, options]*)

ひとつの Python 文と、それに対する期待する出力からなる、単一の対話的モードの実行例です。コンストラクタの引数は `Example` インスタンス中の同名の属性の初期化に使われます。

バージョン 2.4 で追加.

Example では、以下の属性を定義しています。これらの変数はコンストラクタで初期化されます。直接変更してはなりません。

source

実行例のソースコードが入った文字列です。ソースコードは単一の Python で、末尾は常に改行です。コンストラクタは必要に応じて改行を追加します。

want

実行例のソースコードを実行した際の期待する出力 (標準出力と、例外が生じた場合にはトレースバック) です。 *want* の末尾は、期待する出力がまったくない場合を除いて常に改行になります。期待する出力がない場合には空文字列になります。コンストラクタは必要に応じて改行を追加します。

exc_msg

実行例が例外を生成すると期待される場合の例外メッセージです。例外を送出しない場合には `None` です。この例外メッセージは、 `traceback.format_exception_only()` の戻り値と比較されます。値が `None` でない限り、 *exc_msg* は改行で終わっていなければなりません; コンストラクタは必要に応じて改行を追加します。

lineno

この実行例を含む文字列における実行例が始まる行番号です。行番号は文字列の先頭を 0 として数えます。

indent

実行例の入っている文字列のインデント、すなわち実行例の最初のプロンプトより前にある空白文字の数です。

options

オプションフラグを `True` または `False` に対応付けている辞書です。実行例に対するデフォルトオプションを上書きするために用いられます。この辞書に入っていないオプションフラグはデフォルトの状態 (*DocTestRunner* の `optionflags` の内容) のままになります。

DocTestFinder オブジェクト

```
class doctest.DocTestFinder ([verbose][, parser][, recurse][, exclude_empty])
```

与えられたオブジェクトについて、そのオブジェクト自身の docstring か、そのオブジェクトに含まれるオブジェクトの docstring から *DocTest* を抽出する処理クラスです。現在のところ、モジュール、関数、クラス、メソッド、静的メソッド、クラスメソッド、プロパティから *DocTest* を抽出できます。

オプション引数 *verbose* を使うと、抽出処理の対象となるオブジェクトを表示できます。デフォルトは `False` (出力を行わない) です。

オプション引数 *parser* には、docstring から *DocTest* を抽出するのに使う *DocTestParser* オブジェクト (またはその代替となるオブジェクト) を指定します。

オプション引数 *recurse* が偽の場合、 *DocTestFinder.find()* は与えられたオブジェクトだけを調べ、そのオブジェクトに含まれる他のオブジェクトを調べません。

オプション引数 `exclude_empty` が偽の場合、`DocTestFinder.find()` は空の docstring を持つオブジェクトもテスト対象に含めます。

バージョン 2.4 で追加.

`DocTestFinder` では以下のメソッドを定義しています:

find(*obj*[, *name*][, *module*][, *globs*][, *extraglobs*])

obj または *obj* 内に入っているオブジェクトの docstring 中で定義されている `DocTest` のリストを返します。

オプション引数 *name* には、オブジェクトの名前を指定します。この名前は、関数が返す `DocTest` の名前になります。 *name* を指定しない場合、`obj.__name__` を使います。

オプションのパラメータ *module* は、指定したオブジェクトを収めているモジュールを指定します。 *module* を指定しないか、`None` を指定した場合には、正しいモジュールを自動的に決定しようと試みます。オブジェクトのモジュールは以下のような役割を果たします:

- *globs* を指定していない場合、オブジェクトのモジュールはデフォルトの名前空間になります。
- 他のモジュールから import されたオブジェクトに対して `DocTestFinder` が `DocTest` を抽出するのを避けるために使います。 (*module* 由来でないオブジェクトを無視します。)
- オブジェクトの入っているファイル名を調べるために使います。
- オブジェクトがファイル内の何行目にあるかを調べる手助けにします。

module が `False` の場合には、モジュールの検索を試みません。これは正確さを欠くような使い方で、通常 doctest 自体のテストにしか使いません。 *module* が `False` の場合、または *module* が `None` で自動的に的確なモジュールを見つけ出せない場合には、すべてのオブジェクトは (non-existent) モジュールに属するとみなされ、そのオブジェクト内のすべてのオブジェクトに対して (再帰的に) doctest の検索を行います。

各 `DocTest` のグローバル変数は、*globs* と *extraglobs* を合わせたもの (*extraglobs* 内の束縛が *globs* 内の束縛を上書きする) になります。各々の `DocTest` に対して、グローバル変数を表す辞書の新たな浅いコピーを生成します。 *globs* を指定しない場合に使われるのデフォルト値は、モジュールを指定していればそのモジュールの `__dict__` になり、指定していなければ `{}` になります。 *extraglobs* を指定しない場合、デフォルトの値は `{}` になります。

DocTestParser オブジェクト

class `doctest.DocTestParser`

対話モードの実行例を文字列から抽出し、それを使って `DocTest` オブジェクトを生成するために使われる処理クラスです。

バージョン 2.4 で追加.

`DocTestParser` では以下のメソッドを定義しています:

get_doctest(*string*, *globs*, *name*, *filename*, *lineno*)

指定した文字列からすべての doctest 実行例を抽出し、`DocTest` オブジェクト内に集めます。

`globs`, `name`, `filename`, および `lineno` は新たに作成される `DocTest` オブジェクトの属性になります。詳しくは `DocTest` のドキュメントを参照してください。

`get_examples (string[, name])`

指定した文字列からすべての `doctest` 実行例を抽出し、`Example` オブジェクトからなるリストにして返します。各 `Example` の行番号は 0 から数えます。オプション引数 `name` はこの文字列につける名前で、エラーメッセージにしか使われません。

`parse (string[, name])`

指定した文字列を、実行例とその間のテキストに分割し、実行例を `Example` オブジェクトに変換し、`Example` と文字列からなるリストにして返します。各 `Example` の行番号は 0 から数えます。オプション引数 `name` はこの文字列につける名前で、エラーメッセージにしか使われません。

DocTestRunner オブジェクト

`class doctest.DocTestRunner ([checker][, verbose][, optionflags])`

`DocTest` 内の対話モード実行例を実行し、検証する際に用いられる処理クラスです。

期待する出力と実際の出力との比較は `OutputChecker` で行います。比較は様々なオプションフラグを使ってカスタマイズできます; 詳しくは [オプションフラグ](#) を参照してください。オプションフラグでは不十分な場合、コンストラクタに `OutputChecker` のサブクラスを渡して比較方法をカスタマイズできます。

テストランナーの表示出力の制御には二つの方法があります。一つ目は、`TestRunner.run()` に出力用の関数を渡すというものです。この関数は、表示すべき文字列を引数にして呼び出されます。デフォルトは `sys.stdout.write` です。出力を取り込んで処理するだけでは不十分な場合、`DocTestRunner` をサブクラス化し、`report_start()`, `report_success()`, `report_unexpected_exception()`, および `report_failure()` をオーバーライドすればカスタマイズできます。

オプションのキーワード引数 `checker` には、`OutputChecker` オブジェクト (またはその代替となるオブジェクト) を指定します。このオブジェクトは `doctest` 実行例の期待する出力と実際の出力との比較を行う際に使われます。

オプションのキーワード引数 `verbose` は、`DocTestRunner` の出すメッセージの冗長性を制御します。`verbose` が `True` の場合、各実行例を実行する都度、その実行例についての情報を出力します。`verbose` が `False` の場合、テストの失敗だけを出力します。`verbose` を指定しない場合や `None` を指定した場合、コマンドラインスイッチ `-v` を使った場合にのみ `verbose` 出力を適用します。

オプションのキーワード引数 `optionflags` を使うと、テストランナーが期待される出力と実際の出力を比較する方法や、テストの失敗を表示する方法を制御できます。詳しくは [オプションフラグ](#) 節を参照してください。

バージョン 2.4 で追加。

`DocTestParser` では以下のメソッドを定義しています:

`report_start (out, test, example)`

テストランナーが実行例を処理しようとしているときにレポートを出力します。

`DocTestRunner` の出力をサブクラスでカスタマイズできるようにするためのメソッドです。直接呼び出してはなりません。

`example` は処理する実行例です。 `test` は `example` の入っているテストです。 `out` は出力用の関数で、 `DocTestRunner.run()` に渡されます。

report_success (`out`, `test`, `example`, `got`)

与えられた実行例が正しく動作したことを報告します。このメソッドは `DocTestRunner` のサブクラスで出力をカスタマイズできるようにするために提供されています; 直接呼び出してはなりません。

`example` は処理する実行例です。 `got` は実行例から実際に得られた出力です。 `test` は `example` の入っているテストです。 `out` は出力用の関数で、 `DocTestRunner.run()` に渡されます。

report_failure (`out`, `test`, `example`, `got`)

与えられた実行例が正しく動作しなかったことを報告します。このメソッドは `DocTestRunner` のサブクラスで出力をカスタマイズできるようにするために提供されています; 直接呼び出してはなりません。

`example` は処理する実行例です。 `got` は実行例から実際に得られた出力です。 `test` は `example` の入っているテストです。 `out` は出力用の関数で、 `DocTestRunner.run()` に渡されます。

report_unexpected_exception (`out`, `test`, `example`, `exc_info`)

与えられた実行例が期待とは違う例外を送出したことを報告します。このメソッドは `DocTestRunner` のサブクラスで出力をカスタマイズできるようにするために提供されています; 直接呼び出してはなりません。

`example` は処理する実行例です。 `exc_info` には予期せず送出された例外の情報を入れたタプル (`sys.exc_info()` の返す内容) になります。 `test` は `example` の入っているテストです。 `out` は出力用の関数で、 `DocTestRunner.run()` に渡されます。

run (`test`[, `compileflags`][, `out`][, `clear_globs`])

`test` 内の実行例 (`DocTest` オブジェクト) を実行し、その結果を出力用の関数 `out` を使って表示します。

実行例は名前空間 `test.globs` の下で実行されます。 `clear_globs` が真 (デフォルト) の場合、名前空間はテストの実行後に消去され、ガベージコレクションを促します。テストの実行完了後にその内容を調べたければ、 `clear_globs=False` としてください。

`compileflags` には、実行例を実行する際に Python コンパイラに適用するフラグセットを指定します。 `compileflags` を指定しない場合、デフォルト値は `globs` で適用されている `future-import` フラグセットになります。

各実行例の出力は `DocTestRunner` の出力チェッカで検査され、その結果は `DocTestRunner.report_*()` メソッドで書式化されます。

summarize ([`verbose`])

この `DocTestRunner` が実行したすべてのテストケースのサマリを出力し、名前付きタプル (`named tuple`) `TestResults(failed, attempted)` を返します。

オプションの `verbose` 引数を使うと、どのくらいサマリを詳しくするかを制御できます。冗長度を指定しない場合、`DocTestRunner` 自体の冗長度を使います。

バージョン 2.6 で変更: 名前付きタプル (named tuple) を使うようになりました。

OutputChecker オブジェクト

`class doctest.OutputChecker`

`doctest` 実行例を実際に実行したときの出力が期待する出力と一致するかどうかをチェックするために使われるクラスです。`OutputChecker` では、与えられた二つの出力を比較して、一致する場合には真を返す `check_output()` と、二つの出力間の違いを説明する文字列を返す `output_difference()` の、二つのメソッドがあります。

バージョン 2.4 で追加.

`OutputChecker` では以下のメソッドを定義しています:

`check_output(want, got, optionflags)`

実行例から実際に得られた出力 (`got`) と、期待する出力 (`want`) が一致する場合にのみ `True` を返します。二つの文字列がまったく同一の場合には常に一致するとみなしますが、テストランナーの使っているオプションフラグにより、厳密には同じ内容になっていなくても一致するとみなす場合もあります。オプションフラグについての詳しい情報は [オプションフラグ](#) 節を参照してください。

`output_difference(example, got, optionflags)`

与えられた実行例 (`example`) の期待する出力と、実際に得られた出力 (`got`) の間の差異を解説している文字列を返します。`optionflags` は `want` と `got` を比較する際に使われるオプションフラグのセットです。

25.2.7 デバッグ

`doctest` では、`doctest` 実行例をデバッグするメカニズムをいくつか提供しています:

- `doctest` を実行可能な Python プログラムに変換し、Python デバッガ `pdb` で実行できるようにするための関数がいくつかあります。
- `DocTestRunner` のサブクラス `DebugRunner` クラスがあります。このクラスは、最初に失敗した実行例に対して例外を送出します。例外には実行例に関する情報が入っています。この情報は実行例の検死デバッグに利用できます。
- `DocTestSuite()` の生成する `unittest` テストケースは、`debug()` メソッドをサポートしています。`debug()` は `unittest.TestCase` で定義されています。
- `pdb.set_trace()` を `doctest` 実行例の中で呼び出しておけば、その行が実行されたときに Python デバッガが組み込まれます。デバッガを組み込んだあとは、変数の現在の値などを調べられます。たとえば、以下のようなモジュールレベルの docstring の入ったファイル `a.py` があるとします:

```
"""
>>> def f(x):
...     g(x*2)
>>> def g(x):
...     print x+3
...     import pdb; pdb.set_trace()
>>> f(3)
9
"""
```

対話セッションは以下になるでしょう:

```
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
1      def g(x):
2          print x+3
3  ->      import pdb; pdb.set_trace()
[EOF]
(Pdb) print x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
-> g(x*2)
(Pdb) list
1      def f(x):
2  ->      g(x*2)
[EOF]
(Pdb) print x
3
(Pdb) step
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>>
```

バージョン 2.4 で変更: `pdb.set_trace()` を `doctest` の中で有効に使えるようになりました。

以下は、`doctest` を Python コードに変換して、できたコードをデバッガ下で実行できるようにするための関数です:

`doctest.script_from_examples(s)`

実行例の入ったテキストをスクリプトに変換します。

引数 *s* は `doctest` 実行例の入った文字列です。この文字列は Python スクリプトに変換され、その中では *s* の `doctest` 実行例が通常のコードに、それ以外は Python のコメント文になります。生成したスクリプ

トを文字列で返します。例えば、

```
import doctest
print doctest.script_from_examples(r"""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print x+y
    3
""")
```

は:

```
# Set x and y to 1 and 2.
x, y = 1, 2
#
# Print their sum:
print x+y
# Expected:
## 3
```

この関数は内部的に他の関数から使われていますが(下記参照)、対話セッションを Python スクリプトに変換したいような場合にも便利でしょう。

バージョン 2.4 で追加.

`doctest.testsource(module, name)`

あるオブジェクトの doctest をスクリプトに変換します。

引数 *module* はモジュールオブジェクトか、対象の doctest を持つオブジェクトの入ったモジュールのドット表記名です。引数 *name* は対象の doctest を持つオブジェクトの(モジュール内の)名前です。対象オブジェクトの docstring を上記の `script_from_examples()` で説明した方法で Python スクリプトに変換してできた文字列を返します。例えば、`a.py` モジュールのトップレベルに関数 `f()` がある場合、以下のコード

```
import a, doctest
print doctest.testsource(a, "a.f")
```

を実行すると、`f()` の docstring から doctest をコードに変換し、それ以外をコメントにしたスクリプトを出力します。

バージョン 2.3 で追加.

`doctest.debug(module, name[, pm])`

オブジェクトの持つ doctest をデバッグします。

module および *name* 引数は上の `testsource()` と同じです。指定したオブジェクトの docstring から合成された Python スクリプトは一時ファイルに書き出され、その後 Python デバッガ `pdb` の制御下で実行されます。

ローカルおよびグローバルの実行コンテキストには、`module.__dict__` の浅いコピーが使われます。

オプション引数 `pm` は、検死デバッグを行うかどうかを指定します。`pm` が真の場合、スクリプトファイルは直接実行され、スクリプトが送出した例外が処理されないまま終了した場合にのみデバッグが立ち入ります。その場合、`pdb.post_mortem()` によって検死デバッグを起動し、処理されなかった例外から得られたトレースバックオブジェクトを渡します。`pm` を指定しないか値を偽にした場合、`pdb.run()` に適切な `execfile()` 呼び出しを渡して、最初からデバッグの下でスクリプトを実行します。

バージョン 2.3 で追加。

バージョン 2.4 で変更: `pm` 引数が追加されました。

```
doctest.debug_src(src[, pm][, globs])
```

文字列中の doctest をデバッグします。

上の `debug()` に似ていますが、doctest の入った文字列は `src` 引数で直接指定します。

オプション引数 `pm` は上の `debug()` と同じ意味です。

オプション引数 `globs` には、ローカルおよびグローバルな実行コンテキストの両方に使われる辞書を指定します。`globs` を指定しない場合や `None` にした場合、空の辞書を使います。辞書を指定した場合、実際の実行コンテキストには浅いコピーが使われます。

バージョン 2.4 で追加。

`DebugRunner` クラス自体や `DebugRunner` クラスが送出する特殊な例外は、テストフレームワークの作者にとって非常に興味のあるところですが、ここでは概要しか述べられません。詳しくはソースコード、とりわけ `DebugRunner` の docstring (それ自体 doctest です!) を参照してください:

```
class doctest.DebugRunner ([checker][, verbose][, optionflags])
```

テストの失敗に遭遇するとすぐに例外を送出するようになっている `DocTestRunner` のサブクラスです。予期しない例外が生じると、`UnexpectedException` 例外を送出します。この例外には、テスト、実行例、もともと送出された例外が入っています。期待する出力と実際の出力が一致しないために失敗した場合には、`DocTestFailure` 例外を送出します。この例外には、テスト、実行例、実際の出力が入っています。

コンストラクタのパラメータやメソッドについては、[拡張 API](#) 節の `DocTestRunner` のドキュメントを参照してください。

`DebugRunner` インスタンスの送出する例外には以下の二つがあります:

```
exception doctest.DocTestFailure (test, example, got)
```

doctest 実行例の実際の出力が期待する出力と一致しなかったことを示すために `DocTestRunner` が送出する例外です。コンストラクタの引数は、インスタンスの同名の属性を初期化するために使われます。

`DocTestFailure` では以下の属性を定義しています:

```
DocTestFailure.test
```

実行例が失敗した時に実行されていた `DocTest` オブジェクトです。

`DocTestFailure.example`

失敗した *Example* オブジェクトです。

`DocTestFailure.got`

実行例の実際の出力です。

exception `doctest.UnexpectedException (test, example, exc_info)`

`doctest` 実行例が予期しない例外を送出したことを示すために *DocTestRunner* が送出する例外です。コンストラクタの引数は、インスタンスの同名の属性を初期化するために使われます。

UnexpectedException では以下の属性を定義しています:

`UnexpectedException.test`

実行例が失敗した時に実行されていた *DocTest* オブジェクトです。

`UnexpectedException.example`

失敗した *Example* オブジェクトです。

`UnexpectedException.exc_info`

予期しない例外についての情報の入ったタプルで、`sys.exc_info()` が返すのと同じものです。

25.2.8 アドバイス

冒頭でも触れたように、*doctest* は、以下の三つの主な用途を持つようになりました:

1. docstring 内の実行例をチェックする。
2. 回帰テストを行う。
3. 実行可能なドキュメント/読めるテストの実現。

これらの用途にはそれぞれ違った要求があるので、区別して考えるのが重要です。特に、docstring を曖昧なテストケースに埋もれさせてしまうとドキュメントとしては最悪です。

docstring の例は注意深く作成してください。doctest の作成にはコツがあり、きちんと学ぶ必要があります — 最初はすんなりできないでしょう。実行例はドキュメントに本物の価値を与えます。良い例は、たくさんの言葉と同じ価値を持つことがしばしばあります。注意深くやれば、例はユーザにとっては非常に有益であり、時を経るにつれて、あるいは状況が変わった際に、何度も修正するのにかかる時間を節約するという形で、きっと見返りを得るでしょう。私は今でも、自分の *doctest* 実行例が “無害な” 変更を行った際にうまく動作しなくなることに驚いています。

説明テキストの作成をけちらなければ、*doctest* は回帰テストの優れたツールにもなり得ます。説明文と実行例を交互に記述していけば、実際に何をどうしてテストしているのかもっと簡単に把握できるようになるでしょう。もちろん、コードベースのテストに詳しくコメントを入れるのも手ですが、そんなことをするプログラマはほとんどいません。多くの人々が、*doctest* のアプローチをとった方がきれいにテストを書けると気づいています。おそらく、これは単にコード中にコメントを書くのが少し面倒だからという理由でしょう。私はもう少し穿った見方もしています。doctest ベースのテストを書くときの自然な態度は、自分のソフトウェアのよい点を説明しようとして、実行例を使って説明しようとするときの態度そのものだからだ、という理由です。それゆえに、テストファイルは自然と単純な機能の解説から始め、論理的により複雑で境界条件的な

ケースに進むような形になります。結果的に、一見ランダムに見えるような個別の機能をテストしている個別の関数の集まりではなく、首尾一貫した説明ができるようになるのです。 *doctest* によるテストの作成はまったく別の取り組み方であり、テストと説明の区別をなくして、まったく違う結果を生み出すのです。

回帰テストは特定のオブジェクトやファイルにまとめておくのがよいでしょう。回帰テストの組み方にはいくつか選択肢があります:

- テストケースを対話モードの実行例にして入れたテキストファイルを書き、 *testfile()* や *DocFileSuite()* を使ってそのファイルをテストします。この方法をお勧めします。最初から *doctest* を使うようにしている新たなプロジェクトでは、この方法が一番簡単です。
- *_regtest_topic* という名前の関数を定義します。この関数には、あるトピックに対応するテストケースの入った docstring が一つだけ入っています。この関数はモジュールと同じファイルの中にも置けますし、別のテストファイルに分けてもかまいません。
- 回帰テストのトピックをテストケースの入った docstring に対応付けた辞書 *__test__* 辞書を定義します。

テストコードをモジュールに組み込むことで、そのモジュールは、モジュール自身がテストランナー (test runner) になることができます。テストが失敗した場合には、問題箇所をデバッグする際に、失敗した *doctest* だけを再度実行することで、作成したテストランナーを修正することができます。これがこのようなテストランナーの最小例となります。

```
if __name__ == '__main__':
    import doctest
    flags = doctest.REPORT_NDIFF|doctest.REPORT_ONLY_FIRST_FAILURE
    if len(sys.argv) > 1:
        name = sys.argv[1]
        if name in globals():
            obj = globals()[name]
        else:
            obj = __test__[name]
        doctest.run_docstring_examples(obj, globals(), name=name,
                                       optionflags=flags)
    else:
        fail, total = doctest.testmod(optionflags=flags)
        print("{} failures out of {} tests".format(fail, total))
```

25.3 unittest — ユニットテストフレームワーク

バージョン 2.1 で追加.

(読者の方がすでにテストの基本概念についてなじみがあるようでしたら、この部分をとばして *the list of assert methods* に進むと良いでしょう。)

この Python ユニットテストフレームワークは時に "PyUnit" と呼ばれ、Kent Beck と Erich Gamma による JUnit の Python 版です。JUnit はまた Kent の Smalltalk 用テストフレームワークの Java 版で、どちらもそれぞれの言語で業界標準のユニットテストフレームワークとなっています。

`unittest` では、テストの自動化・初期設定と終了処理の共有・テストの分類・テスト実行と結果レポートの分離などの機能を提供しており、`unittest` のクラスを使って簡単にたくさんのテストを開発できるようになっています。

これを実現するために、`unittest` はいくつかの重要な概念をサポートしています：

テストフィクスチャ (test fixture) *test fixture* とは、テスト実行のために必要な準備や終了処理を指します。
例: テスト用データベースの作成・ディレクトリ・サーバプロセスの起動など。

テストケース (test case) *test case* はテストの最小単位で、各入力に対する結果をチェックします。テストケースを作成する場合は、`unittest` が提供する `TestCase` クラスを基底クラスとして利用することができます。

テストスイート (test suite) *test suite* はテストケースとテストスイートの集まりで、同時に実行しなければならないテストをまとめる場合に使用します。

テストランナー (test runner) *test runner* はテストの実行と結果表示を管理するコンポーネントです。ランナーはグラフィカルインターフェースでもテキストインターフェースでも良いですし、何も表示せずにテスト結果を示す値を返すだけの場合もあります。

`unittest` では、テストケースとテストフィクスチャーを、`TestCase` クラスと `FunctionTestCase` クラスで提供しています。`TestCase` クラスは新規にテストを作成する場合に使用し、`FunctionTestCase` は既存のテストを `unittest` に組み込む場合に使用します。テストフィクスチャーの初期化処理と終了処理は、`TestCase` では `setUp()` メソッドと `tearDown()` をオーバーライドして記述し、`FunctionTestCase` では初期化・終了処理を行う既存の関数をコンストラクタで指定します。テスト実行時、まずテストフィクスチャーの初期設定が最初に実行されます。初期化処理が正常終了した場合、テスト実行後にはテスト結果に関わらず終了処理が実行されます。`TestCase` の各インスタンスが実行するテストは一つだけで、テストフィクスチャーは各テストごとに新しく作成されます。

テストスイートは `TestSuite` クラスで実装されており、複数のテストとテストスイートをまとめる事ができます。テストスイートを実行すると、スイートと子スイートに追加されている全てのテストが実行されます。

テストランナーは `run()` メソッドを持つオブジェクトです。このメソッドは引数として `TestCase` か `TestSuite` オブジェクトを受け取り、テスト結果を `TestResult` オブジェクトで戻します。`unittest` ではデフォルトでテスト結果を標準エラーに出力する `TextTestRunner` をサンプルとして実装しています。これ以外のランナー (グラフィックインターフェース用など) を実装する場合でも、特別なクラスから派生させて実装する必要はありません。

参考:

`doctest` モジュール もうひとつのテストをサポートするモジュールで、本モジュールと趣きが異なります。

`unittest2: A backport of new unittest features for Python 2.4-2.6` Python 2.7 になり多くの機能が `unittest` に追加されました。特に、テストディスカバリが追加されました。`unittest2` を導入する事で以前のバージョンの Python でもこれらの機能を使えます。

`Simple Smalltalk Testing: With Patterns` Kent Beck のテストイングフレームワークに関する原論文で、ここに記載されたパターンを `unittest` が使用しています。

Nose and pytest サードパーティのユニットテストフレームワークで軽量な文法でテストを書くことができます。例えば、`assert func(10) == 42` のように書きます。

The Python Testing Tools Taxonomy 多くの Python のテストツールが一覧で紹介されています。ファンクショナルテストのフレームワークやモックライブラリも掲載されています。

Testing in Python **メーリングリスト** Python でテストやテストツールについての議論に特化したグループです。

25.3.1 基礎的な例

`unittest` モジュールには、テストの開発や実行の為の優れたツールが用意されており、この節では、その一部を紹介します。ほとんどのユーザにとっては、ここで紹介するツールだけで十分でしょう。

以下は、三つの文字列メソッドをテストするスクリプトです:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

テストケースは、`unittest.TestCase` のサブクラスとして作成します。メソッド名が `test` で始まる三つのメソッドがテストです。テストランナーはこの命名規約によってテストを行うメソッドを検索します。

これらのテスト内では、予定の結果が得られていることを確かめるために `assertEqual()` を、条件のチェックに `assertTrue()` や `assertFalse()` を、例外が発生する事を確認するために `assertRaises()` をそれぞれ呼び出しています。 `assert` 文の代わりにこれらのメソッドを使用すると、テストランナーでテスト結果を集計してレポートを作成する事ができます。

`setUp()` および `tearDown()` メソッドによって各テストメソッドの前後に実行する命令を実装することが出来ます。詳細は [テストの構成](#) を参照してください。

サンプルの末尾が、簡単なテストの実行方法です。 `unittest.main()` は、テストスクリプトのコマンドライン用インターフェースです。コマンドラインから起動された場合、上記のスクリプトから以下のような結果

が出力されます:

```
...
-----
Ran 3 tests in 0.000s

OK
```

簡略化した結果を出力したり、コマンドライン以外からも起動する等のより細かい制御が必要であれば、`unittest.main()` を使用せずに別の方法でテストを実行します。コマンドラインから起動する必要もありません。例えば、上記サンプルの最後の 2 行は以下のように書くことができます:

```
suite = unittest.TestLoader().loadTestsFromTestCase(TestStringMethods)
unittest.TextTestRunner(verbosity=2).run(suite)
```

変更後のスクリプトをインタプリタや別のスクリプトから実行すると、以下の出力が得られます:

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

-----
Ran 3 tests in 0.001s

OK
```

以上が `unittest` モジュールでよく使われる機能で、ほとんどのテストではこれだけでも十分です。基礎となる概念や全ての機能については以降の章を参照してください。

25.3.2 コマンドラインインターフェイス

ユニットテストモジュールはコマンドラインから使うこともできます。モジュール、クラス、もしくは、特定のテストメソッドで定義されたテストを実行します:

```
python -m unittest test_module1test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

引数として渡す事ができるのは、テストが定義されたモジュール名、もしくはクラス、メソッドのフルパス名です。

テスト実行時に (より冗長な) 詳細を表示するには `-v` フラグを渡します:

```
python -m unittest -v test_module
```

コマンドラインオプションの一覧を表示するには以下のコマンドを実行します:

```
python -m unittest -h
```

バージョン 2.7 で変更: 以前のバージョンでは、個々のテストメソッドしか実行することができず、モジュール単位やクラス単位で実行することは不可能でした。

コマンドラインオプション

unittest には以下のコマンドラインオプションがあります:

-b, --buffer

標準出力と標準エラーのストリームをテスト実行の間バッファリングします。テストが成功している間は結果の出力は破棄されます。テストが失敗、もしくはエラーが発生した場合には、結果にエラーメッセージが追加されたうえで通常通り出力されます。

-c, --catch

Control-C を実行中のテストが終了するまで遅延させ、そこまでの結果を出力します。二回目の Control-C は、通常通り *KeyboardInterrupt* の例外を発生させます。

この機能の仕組みについては、[シグナルハンドリング](#) を参照してください。

-f, --failfast

初回のエラーもしくは失敗の時にテストを停止します。

バージョン 2.7 で追加: コマンドラインオプションの **-b**、**-c**、**-f** が追加されました。

このコマンドラインは、プロジェクト内の全テストを実行したり、サブセットのみを実行したりといった、テストディスカバリを使用することもできます。

25.3.3 テストディスカバリ

バージョン 2.7 で追加.

unittest はシンプルなテストディスカバリをサポートします。このテストディスカバリに対応するために、テストが定義された全ファイルは modules もしくは packages としてプロジェクトの最上位のディスカバリでインポート可能である必要があります (つまり、これらのファイルは identifiers として有効である必要があるということです)。

テストディスカバリは *TestLoader.discover()* で実装されています。しかし、コマンドラインからも使うことができます。コマンドラインからは以下のように使用します:

```
cd project_directory
python -m unittest discover
```

discover サブコマンドには以下のオプションがあります:

-v, --verbose

詳細な出力

-s, --start-directory directory

ディスカバリを開始するディレクトリ (デフォルトは .)

-p, --pattern pattern

テストファイル名を識別するパターン (デフォルトは `test*.py`)

-t, --top-level-directory directory

プロジェクトの最上位のディスカバリのディレクトリ (デフォルトは開始のディレクトリ)

`-s`、`-p`、および `-t` の各オプションは、この順番で指定すれば位置固定の引数として指定する事ができます。以下の二つのコマンドは同じ結果になります:

```
python -m unittest discover -s project_directory -p "*_test.py"
python -m unittest discover project_directory "*_test.py"
```

パスを渡すのはもちろんのこと、例えば `myproject.subpackage.test` のように、パッケージ名をスタートディレクトリとして渡すことができます。指定したパッケージがインポートされ、そのパッケージのファイルシステム上のパスがスタートディレクトリになります。

ご用心: テストディスカバリはテストをインポートすることで読み込みます。テストディスカバリは一度、指定した開始ディレクトリから全テストファイルを探索し、そのファイルのパスをパッケージ名に変換してインポートします。例えば、`foo/bar/baz.py` は `foo.bar.baz` としてインポートされます。

もしパッケージをグローバルにインストールしていて、インストールしたのとは異なるパッケージのコピーをディスカバリしようとする、間違った場所からインポートしてしまうかもしれません。このような状態になるとテストディスカバリは警告を出し、停止します。

スタートディレクトリとしてディレクトリのパスではなくパッケージ名を指定した場合は、いずれかの場所からインポートされます。この場合は警告が表示されません。

テストモジュールとテストパッケージは、テストのロードとディスカバリをカスタマイズすることができます。そのために [load_tests](#) プロトコル を使用します。

25.3.4 テストの構成

ユニットテストの基礎となる構築要素は、*test case* — セットアップを行い、正しさのチェックを行う、独立したシナリオ — です。 `unittest` では、テストケースは `unittest` モジュールの `TestCase` クラスのインスタンスで表現します。テストケースを作成するには `TestCase` のサブクラスを記述するか、または `FunctionTestCase` を使用します。

`TestCase` から派生したクラスのインスタンスは、このオブジェクトだけで単独のテストメソッドをお膳立て (set-up) と後片付け (tidy-up / clean-up) を伴って実行します。

`TestCase` インスタンスは外部から完全に独立し、単独で実行する事も、他の任意のテストと一緒に実行する事もできなければなりません。

以下のように、`TestCase` のサブクラスは `runTest()` をオーバーライドし、必要なテスト処理を記述するだけで簡単に書くことができます:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def runTest(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50), 'incorrect default size')
```

何らかのテストを行う場合、ベースクラス `TestCase` の `assert*`() 系メソッドのどれかを使用してください。テストが失敗すると例外が送出され、`unittest` はテスト結果を *failure* とします。その他の例外は *error* となります。これによりどこに問題があるかが判ります。*failure* は間違った結果 (6 になるはずが 5 だった) で発生します。*error* は間違ったコード (たとえば間違った関数呼び出しによる *TypeError*) で発生します。

テストの実行方法については後述とし、まずはテストケースインスタンスの作成方法を示します。テストケースインスタンスは、以下のように引数なしでコンストラクタを呼び出して作成します。:

```
testCase = DefaultWidgetSizeTestCase()
```

さて、そのようなテストケースが夥しい数になってくると、セットアップ処理は繰り返しになりがちです。例えば上記のような `Widget` のテストが 100 種類も必要な場合、それぞれのサブクラスで `Widget` オブジェクトを生成する処理を記述するのは好ましくありません。

ラッキーですね、私たちは初期化処理を `setUp()` メソッドに追い出すことが出来ます。これでテスト実行時にテストフレームワークが、私たちのためにそれを自動的に実行してくれます:

```
import unittest

class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

class DefaultWidgetSizeTestCase(SimpleWidgetTestCase):
    def runTest(self):
        self.assertEqual(self.widget.size(), (50,50),
                          'incorrect default size')

class WidgetResizeTestCase(SimpleWidgetTestCase):
    def runTest(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                          'wrong size after resize')
```

テスト中に `setUp()` メソッドで例外が発生した場合、テストフレームワークはテストを実行することができないとみなし、`runTest()` を実行しません。

同様に、終了処理を `tearDown()` メソッドに記述すると、`runTest()` メソッド終了後に実行されます:

```
import unittest

class SimpleWidgetTestCase(unittest.TestCase):
```

(次のページに続く)

(前のページからの続き)

```
def setUp(self):
    self.widget = Widget('The widget')

def tearDown(self):
    self.widget.dispose()
    self.widget = None
```

`setUp()` が正常終了した場合、`runTest()` が成功したかどうかに関わらず `tearDown()` が実行されます。

このような、テストを実行する環境を *fixture* と呼びます。

多数の小さなテストケース群が同じフィクスチャを使うことは良くあります。今の場合だと私たちは結局 `SimpleWidgetTestCase` を派生して、`DefaultWidgetSizeTestCase` のような小さくて一つだけメソッドを持つクラスをたくさん作る必要があるところでした。これは退屈で時間のかかるものです。ですから JUnit と同じようなやり方で、`unittest` ではより簡単なメカニズムを用意しています:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
        self.widget = None

    def test_default_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                          'incorrect default size')

    def test_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                          'wrong size after resize')
```

この例では `runTest()` がありませんが、二つのテストメソッドを定義しています。このクラスのインスタンスは `test_*`() メソッドのどちらか一方の実行と、`self.widget` の生成・解放を行います。この場合、テストケースインスタンス生成時に、コンストラクタの引数として実行するメソッド名を指定します:

```
defaultSizeTestCase = WidgetTestCase('test_default_size')
resizeTestCase = WidgetTestCase('test_resize')
```

`unittest` では `test suite` によってテストケースインスタンスをテスト対象の機能によってグループ化することができます。 *test suite* は、`unittest` の `TestSuite` クラスで作成します。:

```
widgetTestSuite = unittest.TestSuite()
widgetTestSuite.addTest(WidgetTestCase('test_default_size'))
widgetTestSuite.addTest(WidgetTestCase('test_resize'))
```


各テストモジュールで、テストケースを組み込んだテストスイートオブジェクトを作成する呼び出し可能オブジェクトを用意しておくと、テストの実行や参照が容易になります:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_size'))
    suite.addTest(WidgetTestCase('test_resize'))
    return suite
```

または:

```
def suite():
    tests = ['test_default_size', 'test_resize']

    return unittest.TestSuite(map(WidgetTestCase, tests))
```

良く似た名前のテスト関数をたくさん定義した *TestCase* のサブクラスを作るのはよくあるパターンなので、*unittest* ではテストスイートの作成と個々のテストをそれを追加するのを自動化するのに使える、*TestLoader* を用意しています。たとえば、:

```
suite = unittest.TestLoader().loadTestsFromTestCase(WidgetTestCase)
```

は *WidgetTestCase.test_default_size()* と *WidgetTestCase.test_resize* を走らせるテストスイートを作成します。 *TestLoader* は自動的にテストメソッドを識別するのに 'test' というメソッド名の接頭辞を使います。

テストケースはテスト関数名を組み込みの文字列順に並べ替えた順序で実行されることに注意してください。

システム全体のテストを一度に行う場合など、テストスイートをさらにグループ化したい場合がよくあります。これは簡単です。 *TestSuite* インスタンスには *TestCase* インスタンスだけでなく *TestSuite* も追加出来るからです:

```
suite1 = module1.TheTestSuite()
suite2 = module2.TheTestSuite()
alltests = unittest.TestSuite([suite1, suite2])
```

テストケースやテストスイートは (*widget.py* のような) テスト対象のモジュール内にも記述できますが、テストは (*test_widget.py* のような) 独立したモジュールに置いた方が以下のような点で有利です:

- テストモジュールだけをコマンドラインから実行することができる。
- テストコードと出荷するコードを分離する事ができる。
- テストコードを、テスト対象のコードに合わせて修正する誘惑に駆られにくい。
- テストコードは、テスト対象コードほど頻繁に更新されない。
- テストコードをより簡単にリファクタリングすることができる。
- C で書いたモジュールのテストはどっちにしろ独立したモジュールとなるのだから、同様にしない理由もない

- テスト戦略を変更した場合でも、ソースコードを変更する必要がない。

25.3.5 既存テストコードの再利用

既存のテストコードが有るとき、このテストを `unittest` で実行しようとするために古いテスト関数をいちいち `TestCase` クラスのサブクラスに変換するのは大変です。

このような場合は、`unittest` では `TestCase` のサブクラスである `FunctionTestCase` クラスを使い、既存のテスト関数をラップします。初期設定と終了処理も行なえます。

以下のテストコードがあった場合:

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

等価なテストケースインスタンスは次のように作成できます:

```
testcase = unittest.FunctionTestCase(testSomething)
```

テストケースの操作の部品として呼び出されるべき追加的なお膳立てと後片付けメソッドを手持ちなのであれば、このようにして提供出来ます:

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

既存のテストスイートからの移行を容易にするため、`unittest` は `AssertionError` の送出でテストの失敗を示すような書き方もサポートしています。しかしながら、`TestCase.fail*()` および `TestCase.assert*()` メソッドを使って明確に書くことが推奨されています。`unittest` の将来のバージョンでは、`AssertionError` は別の目的に使用される可能性があります。

注釈: `FunctionTestCase` を使って既存のテストを `unittest` ベースのテスト体系に変換することができますが、この方法は推奨されません。時間を掛けて `TestCase` のサブクラスに書き直した方が将来的なテストのリファクタリングが限りなく易くなります。

既存のテストが `doctest` を使って書かれている場合もあるでしょう。その場合、`doctest` は `DocTestSuite` クラスを提供します。このクラスは、既存の `doctest` ベースのテストから、自動的に `unittest.TestSuite` のインスタンスを作成します。

25.3.6 テストのスキップと意図的な失敗

バージョン 2.7 で追加.

unittest は特定のテストメソッドやテストクラス全体をスキップする仕組みを備えています。さらに、この機能はテスト結果を「意図的な失敗 (expected failure)」とすることができ、テストが失敗しても *TestResult* の失敗数にはカウントされなくなります。

テストをスキップするには、単に *skip()* デコレータ (*decorator*) を使用するか、条件を表現するための *skip()* に類するデコレータを使用します。

スキップは以下のようになります:

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass
```

このサンプルを詳細モードで実行すると以下のように出力されます:

```
test_format (__main__.MyTestCase) ... skipped 'not supported in this library.
↪version'
test_nothing (__main__.MyTestCase) ... skipped 'demonstrating skipping'
test_windows_support (__main__.MyTestCase) ... skipped 'requires Windows'

-----
Ran 3 tests in 0.005s

OK (skipped=3)
```

テストクラスは以下のようにメソッドをスキップすることができます:

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

TestCase.setUp() もスキップすることができます。この機能はセットアップの対象のリソースが使用不可能な状態の時に便利です。

意図的な失敗の機能を使用するには、*expectedFailure()* デコレータを使います。

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

独自のスキップ用のデコレータも簡単に作成することができます。そのためには、独自のデコレータのスキップしたい時点で `skip()` を呼び出します。以下のデコレータはオブジェクトに指定した属性が無い場合にテストをスキップします:

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj, attr))
```

以下のデコレータはテストのスキップと意図的な失敗を実装しています:

`unittest.skip(reason)`

デコレートしたテストを無条件でスキップします。*reason* にはテストをスキップした理由を記載します。

`unittest.skipIf(condition, reason)`

condition が真の場合に、デコレートしたテストをスキップします。

`unittest.skipUnless(condition, reason)`

condition が偽の場合に、デコレートしたテストをスキップします。

`unittest.expectedFailure()`

テストの失敗が意図的であることを表します。該当のテストが失敗しても、そのテストは失敗にカウントされません。

exception `unittest.SkipTest(reason)`

この例外はテストをスキップするために送出されます。

ふつうはこれを直接送出する代わりに `TestCase.skipTest()` やスキッピングデコレータの一つを使用出来ます。

スキップしたテストの前後では、`setUp()` および `tearDown()` は実行されません。同様に、スキップしたテストクラスの前後では、`setUpClass()` および `tearDownClass()` は実行されません。

25.3.7 クラスと関数

この節では、`unittest` モジュールの API の詳細について説明します。

テストクラス

class `unittest.TestCase(methodName='runTest')`

`TestCase` クラスのインスタンスは、`unittest` の世界におけるテストの最小実行単位を示します。このクラスをベースクラスとして使用し、必要なテストを具象サブクラスに実装します。 `TestCase`

クラスでは、テストランナーがテストを実行するためのインターフェースと、各種のチェックやテスト失敗をレポートするためのメソッドを実装しています。

それぞれの `TestCase` クラスのインスタンスはただ一つのテストメソッド、`methodName` という名のメソッドを実行します。既に次のような例を扱ったことを憶えているでしょうか。:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_size'))
    suite.addTest(WidgetTestCase('test_resize'))
    return suite
```

ここでは、それぞれが一つずつのテストを実行するような `WidgetTestCase` の二つのインスタンスを作成しています。

`methodName` のデフォルトは `runTest()` です。

`TestCase` のインスタンスのメソッドは 3 種類のグループに分けられます。1 つ目のグループのメソッドはテストの実行で使用します。2 つ目のグループのメソッドは条件の確認および失敗のレポートといったテストの実装で使用されます。3 つ目のグループである問い合わせ用のメソッドはテスト自身の情報を収集するために使用します。

はじめのグループ (テスト実行) に含まれるメソッドは以下の通りです:

`setUp()`

テストフィクスチャの準備のために呼び出されるメソッドです。テストメソッドの直前に呼び出されます。このメソッドを実行中に `AssertionError` や `SkipTest` 以外の例外が発生した場合、テストの失敗ではなくエラーとされます。デフォルトの実装では何も行いません。

`tearDown()`

テストメソッドが実行され、結果が記録された直後に呼び出されるメソッドです。このメソッドはテストメソッドで例外が投げられても呼び出されます。そのため、サブクラスでこのメソッドを実装する場合は、内部状態を確認することが必要になるでしょう。このメソッドで `AssertionError` や `SkipTest` 以外の例外が発生した場合、テストの失敗とは別のエラーとみなされます (従って報告されるエラーの総数は増えます)。このメソッドは、テストの結果に関わらず `setUp()` が成功した場合にのみ呼ばれます。デフォルトの実装では何も行いません。

`setUpClass()`

A class method called before tests in an individual class are run. `setUpClass` is called with the class as the only argument and must be decorated as a `classmethod()`:

```
@classmethod
def setUpClass(cls):
    ...
```

詳しくは [クラスとモジュールのフィクスチャ](#) を参照してください。

バージョン 2.7 で追加.

`tearDownClass()`

クラス内に定義されたテストが実行された後に呼び出されるクラスメソッドです。`tearDownClass` はクラスを唯一の引数として取り、`classmethod()` でデコレートされている必要があります:

```
@classmethod
def tearDownClass(cls):
    ...
```

詳しくは [クラスとモジュールのフィクスチャ](#) を参照してください。

バージョン 2.7 で追加.

run (*result=None*)

テストを実行し、テスト結果を *result* に指定されたテスト結果オブジェクトに渡します。 *result* 省略されるか `None` が渡された場合、一時的な結果オブジェクトを (`defaultTestCase()` メソッドを呼んで) 生成して、それを使用します。結果オブジェクトは `run()` の呼び出し元には返されません。

このメソッドは、単に `TestCase` インスタンスを呼び出した場合と同様に振る舞います。

skipTest (*reason*)

現在のテストでテストクラスもしくは `setUp()` をスキップする場合に呼ばれます。詳細については、[テストのスキップと意図的な失敗](#) を参照してください。

バージョン 2.7 で追加.

debug ()

テスト結果を収集せずにテストを実行します。例外が呼び出し元に通知されます。また、テストをデバッガで実行することができます。

`TestCase` クラスは失敗の検査と報告を行う多くのメソッドを提供しています。以下の表は最も一般的に使われるメソッドを列挙しています (より多くのアサートメソッドについては表の下を見てください):

Method	確認事項	初出
<code>assertEqual(a, b)</code>	<code>a == b</code>	
<code>assertNotEqual(a, b)</code>	<code>a != b</code>	
<code>assertTrue(x)</code>	<code>bool(x) is True</code>	
<code>assertFalse(x)</code>	<code>bool(x) is False</code>	
<code>assertIs(a, b)</code>	<code>a is b</code>	2.7
<code>assertIsNot(a, b)</code>	<code>a is not b</code>	2.7
<code>assertIsNone(x)</code>	<code>x is None</code>	2.7
<code>assertIsNotNone(x)</code>	<code>x is not None</code>	2.7
<code>assertIn(a, b)</code>	<code>a in b</code>	2.7
<code>assertNotIn(a, b)</code>	<code>a not in b</code>	2.7
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>	2.7
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>	2.7

(`assertRaises()` と `assertRaisesRegexp()` を除く) すべてのアサートメソッドには `msg` 引数を指定することができ、テストの失敗時のエラーメッセージで使用されます (`longMessage` も参照してください)。

assertEqual (*first, second, msg=None*)

first と *second* が等しいことをテストします。両者が比較出来ない場合は、テストが失敗します。

さらに、*first* と *second* が厳密に同じ型であり、その型が、list, tuple, dict, set, frozenset もしくは unicode のいずれか、または `addTypeEqualityFunc()` で比較関数が登録されている型の場合には、より有益なデフォルトのエラーメッセージを生成するために、その型特有の比較関数が呼ばれます (*list of type-specific methods* も参照してください)。

バージョン 2.7 で変更: 自動で型特有の比較関数が呼ばれるようになりました。

assertNotEqual (*first, second, msg=None*)

first と *second* が等しくないことをテストします。両者が比較出来ない場合は、テストが失敗します。

assertTrue (*expr, msg=None*)

assertFalse (*expr, msg=None*)

expr が真 (偽) であることをテストします。

このメソッドは、`bool(expr) is True` と等価であり、`expr is True` と等価ではないことに注意が必要です (後者のためには、`assertIs(expr, True)` が用意されています)。また、専用のメソッドが使用できる場合には、そちらを使用してください (例えば `assertTrue(a == b)` の代わりに `assertEqual(a, b)` を使用してください)。そうすることにより、テスト失敗時のエラーメッセージを詳細に表示することができます。

assertIs (*first, second, msg=None*)

assertIsNot (*first, second, msg=None*)

first と *second* が同じオブジェクトであること (そうでないこと) をテストします。

バージョン 2.7 で追加.

assertIsNone (*expr, msg=None*)

assertIsNotNone (*expr, msg=None*)

expr が `None` であること (および、そうでないこと) をテストします。

バージョン 2.7 で追加.

assertIn (*first, second, msg=None*)

assertNotIn (*first, second, msg=None*)

first が *second* に含まれること (そうでないこと) をテストします。

バージョン 2.7 で追加.

assertIsInstance (*obj, cls, msg=None*)

assertNotIsInstance (*obj, cls, msg=None*)

obj が *cls* のインスタンスであること (あるいはそうでないこと) をテストします (この *cls* は、

`isinstance()` が扱うことのできる、クラスもしくはクラスのタプルである必要があります)。
正確な型をチェックするためには、`assertIs(type(obj), cls)` を使用してください。

バージョン 2.7 で追加。

例外と例外発生時の警告を確認するために以下のメソッドを使用することができます:

Method	確認事項	初出
<code>assertRaises(exc, fun, *args, **kwargs)</code>	<code>fun(*args, **kwargs)</code> raises <code>exc</code>	
<code>assertRaisesRegexp(exc, r, fun, *args, **kwargs)</code>	<code>fun(*args, **kwargs)</code> が <code>exc</code> を raise してメッセージが正規表現 <code>r</code> とマッチすること	2.7

assertRaises (*exception*, *callable*, **args*, ***kwargs*)

assertRaises (*exception*)

callable を呼び出した時に例外が発生することをテストします。 `assertRaises()` で指定した位置パラメータとキーワードパラメータを該当メソッドに渡します。 *exception* が投げられた場合にテストが成功します。また、他の例外が投げられた場合はエラー、例外が投げられなかった場合は失敗になります。複数の例外をキャッチする場合には、例外クラスのタプルを *exception* に指定してください。

exception 引数のみが渡された場合には、コンテキストマネージャが返されます。これにより関数名を渡す形式ではなく、インラインでテスト対象のコードを書くことができます。

```
with self.assertRaises(SomeException):
    do_something()
```

このコンテキストマネージャは *exception* で指定されたオブジェクトを格納します。これにより、例外発生時の詳細な確認をおこなうことができます:

```
with self.assertRaises(SomeException) as cm:
    do_something()

the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

バージョン 2.7 で変更: `assertRaises()` がコンテキストマネージャとして使えるようになりました。

assertRaisesRegexp (*exception*, *regexp*, *callable*, **args*, ***kwargs*)

assertRaisesRegexp (*exception*, *regexp*)

`assertRaises()` と同等ですが、例外の文字列表現が *regexp* にマッチすることもテストします。*regexp* は正規表現オブジェクトか、`re.search()` が扱える正規表現が書かれた文字列である必要があります。例えば以下ようになります:

```
self.assertRaisesRegexp(ValueError, "invalid literal for.*XYZ'",
                        int, 'XYZ')
```

もしくは:

```
with self.assertRaisesRegexp(ValueError, 'literal'):
    int('XYZ')
```

バージョン 2.7 で追加.

さらに特有の確認を行うために以下のメソッドが用意されています:

Method	確認事項	初出
<code>assertAlmostEqual(a, b)</code>	<code>round(a-b, 7) == 0</code>	
<code>assertNotAlmostEqual(a, b)</code>	<code>round(a-b, 7) != 0</code>	
<code>assertGreater(a, b)</code>	<code>a > b</code>	2.7
<code>assertGreaterEqual(a, b)</code>	<code>a >= b</code>	2.7
<code>assertLess(a, b)</code>	<code>a < b</code>	2.7
<code>assertLessEqual(a, b)</code>	<code>a <= b</code>	2.7
<code>assertRegexpMatches(s, r)</code>	<code>r.search(s)</code>	2.7
<code>assertNotRegexpMatches(s, r)</code>	<code>not r.search(s)</code>	2.7
<code>assertItemsEqual(a, b)</code>	意味的に <code>sorted(a) == sorted(b)</code> ですが、ハッシュ不能オブジェクトでも動作	2.7
<code>assertDictContainsSubset(a, b)</code>	<code>a</code> の全てのキー/値ペアが <code>b</code> に存在すること	2.7

assertAlmostEqual (*first*, *second*, *places*=7, *msg*=None, *delta*=None)

assertNotAlmostEqual (*first*, *second*, *places*=7, *msg*=None, *delta*=None)

first と *second* が近似的に等しい (等しくない) ことをテストします。この比較は、*places* (デフォルト 7) で指定した小数位で丸めた差分をゼロと比べることでおこないます。これらのメソッドは、(`round()` と同様に) 小数位 を指定するのであって、有効桁数 を指定するのではないことに注意してください。

places の代わりに *delta* が渡された場合には、*first* と *second* の差分が *delta* 以下 (以上) であることをテストします。

delta と *places* の両方が指定された場合は `TypeError` が投げられます。

バージョン 2.7 で変更: `assertAlmostEqual()` は、オブジェクトが等しい場合には自動で近似的に等しいとみなすようになりました。 `assertNotAlmostEqual()` は、オブジェクトが等しい場合には自動的に失敗するようになりました。 `delta` 引数が追加されました。

assertGreater (*first, second, msg=None*)

assertGreaterEqual (*first, second, msg=None*)

assertLess (*first, second, msg=None*)

assertLessEqual (*first, second, msg=None*)

first が *second* と比べて、メソッド名に対応して `>`, `>=`, `<` もしくは `<=` であることをテストします。そうでない場合はテストが失敗します:

```
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or equal to "4"
```

バージョン 2.7 で追加.

assertRegexpMatches (*text, regexp, msg=None*)

regexp の検索が *text* とマッチすることをテストします。テスト失敗時には、エラーメッセージにパターンと *text* が表示されます (もしくは、パターンと意図しないかたちでマッチした *text* の一部が表示されます)。 *regexp* は正規表現オブジェクトか、 `re.search()` が扱える正規表現が書かれた文字列である必要があります。

バージョン 2.7 で追加.

assertNotRegexpMatches (*text, regexp, msg=None*)

regexp の検索が *text* とマッチしないことをテストします。テスト失敗時には、エラーメッセージにマッチしたパターンと *text* が表示されます。 *regexp* は正規表現オブジェクトか、 `re.search()` が扱える正規表現が書かれた文字列である必要があります。

バージョン 2.7 で追加.

assertItemsEqual (*actual, expected, msg=None*)

シーケンス *expected* が *actual* と同じ要素を含んでいることをテストします。要素の順序はテスト結果に影響しません。要素が含まれていない場合には、シーケンスの差分がエラーメッセージとして表示されます。

actual と *expected* の比較では、重複した要素は無視 されません 。両者に同じ数の要素が含まれていることを検証します。このメソッドは `assertEqual(sorted(expected), sorted(actual))` と同等に振る舞うことに加えて、ハッシュ化できないオブジェクトのシーケンスでも動作します。

Python 3 ではこのメソッドは `assertCountEqual` と名付けられています。

バージョン 2.7 で追加.

assertDictContainsSubset (*expected, actual, msg=None*)

辞書 *actual* のキー/値ペアが *expected* のスーパーセットになっているかどうかをテストします。そうっていない場合には、欠けたキーと不一致の値を一覧するエラーメッセージが生成されます。

バージョン 2.7 で追加.

バージョン 3.2 で非推奨.

`assertEqual()` メソッドは、同じ型のオブジェクトの等価性確認のために、型ごとに特有のメソッドにディスパッチします。これらのメソッドは、ほとんどの組み込み型用のメソッドは既に実装されています。さらに、`addTypeEqualityFunc()` を使う事で新たなメソッドを登録することができます:

addTypeEqualityFunc (*typeobj*, *function*)

`assertEqual()` で呼び出される型特有のメソッドを登録します。登録するメソッドは、比較する 2 つのオブジェクトの型が厳密に *typeobj* と同じ (サブクラスでもいけません) の場合に等価性を確認します。*function* は `assertEqual()` と同様に、2 つの位置固定引数と、3 番目に `msg=None` のキーワード引数を取れる必要があります。このメソッドは、始めの 2 つに指定したパラメータ間の差分を検出した時に `self.failureException(msg)` の例外を投げる必要があります。この例外を投げる際は、出来る限り、エラーの内容が分かる有用な情報と差分の詳細をエラーメッセージに含めてください。

バージョン 2.7 で追加.

`assertEqual()` が自動的に呼び出す型特有のメソッドの概要を以下の表示に記載しています。これらのメソッドは通常は直接呼び出す必要がないことに注意が必要です。

Method	比較の対象	初出
<code>assertMultiLineEqual(a, b)</code>	文字列	2.7
<code>assertSequenceEqual(a, b)</code>	シーケンス	2.7
<code>assertListEqual(a, b)</code>	リスト	2.7
<code>assertTupleEqual(a, b)</code>	タプル	2.7
<code>assertSetEqual(a, b)</code>	set または frozenset	2.7
<code>assertDictEqual(a, b)</code>	辞書	2.7

assertMultiLineEqual (*first*, *second*, *msg=None*)

Test that the multiline string *first* is equal to the string *second*. When not equal a diff of the two strings highlighting the differences will be included in the error message. This method is used by default when comparing Unicode strings with `assertEqual()`.

バージョン 2.7 で追加.

assertSequenceEqual (*seq1*, *seq2*, *msg=None*, *seq_type=None*)

2 つのシーケンスが等しいことをテストします。*seq_type* が指定された場合、*seq1* と *seq2* が *seq_type* のインスタンスで無い場合にはテストが失敗します。シーケンスどうしが異なる場合には、両者の差分がエラーメッセージに表示されます。

このメソッドは直接 `assertEqual()` からは呼ばれませんが、`assertListEqual()` と `assertTupleEqual()` の実装で使われています。

バージョン 2.7 で追加.

assertListEqual (*list1*, *list2*, *msg=None*)

assertTupleEqual (*tuple1, tuple2, msg=None*)

2つのリストまたはタプルが等しいかどうかをテストします。等しくない場合には、両者の差分を表示します。2つのパラメータの型が異なる場合にはテストがエラーになります。このメソッドは、デフォルトで、`assertEqual()` が list または tuple を比較するときに自動的に使用します。

バージョン 2.7 で追加.

assertSetEqual (*set1, set2, msg=None*)

2つのセットが等しいかどうかをテストします。等しくない場合には、両者の差分を表示します。このメソッドは、デフォルトで、`assertEqual()` が set もしくは frozenset を比較するときに自動的に使用します。

set1 or *set2* のいずれかに `set.difference()` メソッドが無い場合にはテストは失敗します。

バージョン 2.7 で追加.

assertDictEqual (*expected, actual, msg=None*)

2つの辞書が等しいかどうかをテストします。等しくない場合には、両者の差分を表示します。このメソッドは、デフォルトで、`assertEqual()` が dict を比較するときに自動的に使用します。

バージョン 2.7 で追加.

最後に、`TestCase` の残りのメソッドと属性を紹介します:

fail (*msg=None*)

無条件にテストを失敗させます。エラーメッセージの表示に、*msg* または `None` が使われます。

failureException

`test()` メソッドが送出する例外を指定するクラス属性です。例えばテストフレームワークで追加情報を付した特殊な例外が必要になる場合、この例外のサブクラスとして作成します。この属性の初期値は `AssertionError` です。

longMessage

この属性に `True` が設定された場合、`assert methods` で指定したすべての明示的な失敗メッセージが、通常の失敗メッセージに追加されます。通常の失敗メッセージには、オブジェクトに関する有用な情報が含まれています。例えば、`assertEqual` は異なるオブジェクトの `repr` を表示します。この属性を `True` にすることで、カスタマイズしたエラーメッセージを通常のメッセージに追加することができます。

この属性はデフォルトで `False` になっていて、カスタムメッセージが渡されても表示しないようになっています。

アサートメソッドを呼び出す前に、インスタンス属性として `True` または `False` を指定することで、この設定をオーバーライドすることができます。

バージョン 2.7 で追加.

maxDiff

この属性は、アサーションメソッドが失敗をレポートする時に表示する差分の長さをコントロールします。デフォルトは 80*8 文字です。この属性が影響するメソッドは、

`assertSequenceEqual()` (およびこのメソッドに委譲するシーケンス比較メソッド)、`assertDictEqual()` と `assertMultiLineEqual()` です。

`maxDiff` を `None` に設定すると差分表示の上限がなくなります。

バージョン 2.7 で追加.

テストフレームワークは、テスト情報を収集するために以下のメソッドを使用します:

`countTestCases()`

テストオブジェクトに含まれるテストの数を返します。 `TestCase` インスタンスは常に 1 を返します。

`defaultTestResult()`

このテストケースクラスで使われるテスト結果クラスのインスタンスを (もし `run()` メソッドに他の結果インスタンスが提供されないならば) 返します。

`TestCase` インスタンスに対しては、いつも `TestResult` のインスタンスですので、`TestCase` のサブクラスでは必要に応じてこのメソッドをオーバーライドしてください。

`id()`

テストケースを特定する文字列を返します。通常、`id` はモジュール名・クラス名を含む、テストメソッドのフルネームを指定します。

`shortDescription()`

テストの説明を一行分、または説明がない場合には `None` を返します。デフォルトでは、テストメソッドの docstring の先頭の一行、または `None` を返します。

`addCleanup(function, *args, **kwargs)`

`tearDown()` の後に呼び出される関数を追加します。この関数はリソースのクリーンアップのために使用します。追加された関数は、追加された順と逆の順番で呼び出されます (LIFO)。 `addCleanup()` に渡された引数とキーワード引数が追加された関数にも渡されます。

`setUp()` が失敗した場合、つまり `tearDown()` が呼ばれなかった場合でも、追加されたクリーンアップ関数は呼び出されます。

バージョン 2.7 で追加.

`doCleanups()`

このメソッドは、`tearDown()` の後、もしくは、`setUp()` が例外を投げた場合は `setUp()` の後に、無条件で呼ばれます。

このメソッドは、`addCleanup()` で追加された関数を呼び出す責務を担います。もし、クリーンアップ関数を `tearDown()` より前に呼び出す必要がある場合には、`doCleanups()` を明示的に呼び出してください。

`doCleanups()` は、どこで呼び出されても、クリーンアップ関数をスタックから削除して実行します。

バージョン 2.7 で追加.

```
class unittest.FunctionTestCase (testFunc, setUp=None, tearDown=None, description=None)
```

このクラスでは `TestCase` インターフェースの内、テストランナーがテストを実行するためのインターフェースだけを実装しており、テスト結果のチェックやレポートに関するメソッドは実装していません。既存のテストコードを `unittest` によるテストフレームワークに組み込むために使用します。

廃止予定のエイリアス

歴史的な経緯で、`TestCase` のいくつかのエイリアスは廃止予定となりました。以下の表に、廃止予定のエイリアスをまとめます:

メソッド名	廃止予定のエイリアス
<code>assertEqual()</code>	<code>failUnlessEqual</code> , <code>assertEquals</code>
<code>assertNotEqual()</code>	<code>failIfEqual</code>
<code>assertTrue()</code>	<code>failUnless</code> , <code>assert_</code>
<code>assertFalse()</code>	<code>failIf</code>
<code>assertRaises()</code>	<code>failUnlessRaises</code>
<code>assertAlmostEqual()</code>	<code>failUnlessAlmostEqual</code>
<code>assertNotAlmostEqual()</code>	<code>failIfAlmostEqual</code>

バージョン 2.7 で非推奨: この表の「廃止予定」だったものが実際に廃止されました。

テストのグルーピング

```
class unittest.TestSuite (tests=())
```

このクラスは、個々のテストケースやテストスイートの集約を示します。通常のテストケースと同じようにテストランナーで実行するためのインタフェースを備えています。`TestSuite` インスタンスを実行することはスイートの繰り返しを使って個々のテストを実行することと同じです。

引数 `tests` が指定された場合、それはテストケースに亘る繰り返し可能オブジェクトまたは内部でスイートを組み立てるための他のテストスイートでなければなりません。後からテストケースやスイートをコレクションに付け加えるためのメソッドも提供されています。

`TestSuite` は `TestCase` オブジェクトのように振る舞います。違いは、スイートにはテストを実装しない点にあります。代わりに、テストをまとめてグループ化して、同時に実行します。`TestSuite` のインスタンスにテスト追加するためのメソッドが用意されています:

```
addTest (test)
```

`TestCase` 又は `TestSuite` のインスタンスをスイートに追加します。

```
addTests (tests)
```

イテラブル `tests` に含まれる全ての `TestCase` 又は `TestSuite` のインスタンスをスイートに追加します。

このメソッドは `tests` 上のイテレーションをしながらそれぞれの要素に `addTest()` を呼び出すのと等価です。

`TestSuite` クラスは `TestCase` と以下のメソッドを共有します:

run(result)

スイート内のテストを実行し、結果を `result` で指定した結果オブジェクトに収集します。
`TestCase.run()` と異なり、`TestSuite.run()` では必ず結果オブジェクトを指定する必要があります。

debug()

このスイートに関連づけられたテストの結果を収集せずに実行します。これによりテストで送出された例外は呼び出し元に伝わるようになり、デバッガの下でのテスト実行をサポートできるようになります。

countTestCases()

このテストオブジェクトによって表現されるテストの数を返します。これには個別のテストと下位のスイートも含まれます。

__iter__()

`TestSuite` でグループ化されたテストはイテレータでアクセスできます。サブクラスは `__iter__()` をオーバーライドすることで、テストへのアクセスを定義します。1つのメソッド内でこのメソッドは何度も呼ばれる可能性があることに注意してください (例えば、テスト数のカウントと等価性の比較)。そのため、イテレーションを繰り返しても同じテストを返すように実装してください。

バージョン 2.7 で変更: 以前のバージョンでは `TestSuite` はイテレータではなく、直接テストにアクセスしていました。そのため、`__iter__()` をオーバーロードしてもテストにアクセスできませんでした。

通常、`TestSuite` の `run()` メソッドは `TestRunner` が起動するため、ユーザが直接実行する必要はありません。

テストのロードと起動

class unittest.TestLoader

`TestLoader` クラスはクラスとモジュールからテストスイートを生成します。通常、このクラスのインスタンスを明示的に生成する必要はありません。`unittest` モジュールの `unittest.defaultTestLoader` を共用インスタンスとして使用することができます。しかし、このクラスのサブクラスやインスタンスで、属性をカスタマイズすることができます。

`TestLoader` のオブジェクトには以下のメソッドがあります:

loadTestsFromTestCase(testCaseClass)

`TestCase` の派生クラス `testCaseClass` に含まれる全テストケースのスイートを返します。

loadTestsFromModule(module)

指定したモジュールに含まれる全テストケースのスイートを返します。このメソッドは `module` 内の `TestCase` 派生クラスを検索し、見つかったクラスのテストメソッドごとにクラスのインスタンスを作成します。

注釈: `TestCase` クラスを基底クラスとしてクラス階層を構築すると `fixture` や補助的な関数をうまく共用することができますが、基底クラスに直接インスタンス化できないテストメソッドがあると、この `loadTestsFromModule()` を使うことができません。この場合でも、`fixture` が全て別々で定義がサブクラスにある場合は使用することができます。

モジュールが `load_tests` 関数を用意している場合、この関数がテストのロードに使われます。これによりテストのロードをカスタマイズできます。これが `load_tests` プロトコル です。

バージョン 2.7 で変更: `load_tests` のサポートが追加されました。

loadTestsFromName (*name*, *module=None*)

文字列で指定される全テストケースを含むスイートを返します。

name には "ドット修飾名" でモジュールかテストケースクラス、テストケースクラス内のメソッド、`TestSuite` インスタンスまたは `TestCase` か `TestSuite` のインスタンスを返す呼び出し可能オブジェクトを指定します。このチェックはここで挙げた順番に行なわれます。すなわち、候補テストケースクラス内のメソッドは「呼び出し可能オブジェクト」としてではなく「テストケースクラス内のメソッド」として拾い出されます。

例えば `SampleTests` モジュールに `TestCase` から派生した `SampleTestCase` クラスがあり、`SampleTestCase` にはテストメソッド `test_one()`・`test_two()`・`test_three()` があるとします。この場合、*name* に '`SampleTests.SampleTestCase`' と指定すると、`SampleTestCase` の三つのテストメソッドを実行するテストスイートが作成されます。'`SampleTests.SampleTestCase.test_two`' と指定すれば、`test_two()` だけを実行するテストスイートが作成されます。インポートされていないモジュールやパッケージ名を含んだ名前を指定した場合は自動的にインポートされます。

また、*module* を指定した場合、*module* 内の *name* を取得します。

loadTestsFromNames (*names*, *module=None*)

`loadTestsFromName()` と同じですが、名前を一つだけ指定するのではなく、複数の名前のシーケンスを指定する事ができます。戻り値は *names* 中の名前指定されるテスト全てを含むテストスイートです。

getTestCaseNames (*testCaseClass*)

testCaseClass 中の全てのメソッド名を含むソート済みシーケンスを返します。*testCaseClass* は `TestCase` のサブクラスでなければなりません。

discover (*start_dir*, *pattern='test*.py'*, *top_level_dir=None*)

指定された開始ディレクトリからサブディレクトリに再帰することですべてのテストモジュールを検索し、それらを含む `TestSuite` オブジェクトを返します。*pattern* にマッチしたテストファイルだけがロードの対象になります。(シェルスタイルのパターンマッチングが使われます)。その中で、インポート可能なモジュール (つまり Python の識別子として有効であるということです) がロードされます。

すべてのテストモジュールはプロジェクトのトップレベルからインポート可能である必要があります。開始ディレクトリがトップレベルディレクトリでない場合は、トップレベルディレクトリが分

離できなくてはなりません。

例えば、シンタックスエラーなどで、モジュールのインポートに失敗した場合、エラーが記録され、ディスカバリ自体は続けられます。

テストパッケージ名 (`__init__.py` の置かれたディレクトリ名) がパターンにマッチした場合、`load_tests` 関数がチェックされます。この関数が存在している場合、この関数に `loader`, `tests`, `pattern` が渡され呼び出されます。

`load_tests` が存在して、ディスカバリがパッケージ内を再帰的な検索を続けている途中で ない 場合、`load_tests` はそのパッケージ内の全てのテストをロードする責務を担います。

意図的にパターンはローダの属性として保持されないようになっています。それにより、パッケージが自分自身のディスカバリを続ける事ができます。`top_level_dir` は保持されるため、`load_tests` はこの引数を `loader.discover()` に渡す必要はありません。

`start_dir` はドット付のモジュール名でもディレクトリでも構いません。

バージョン 2.7 で追加.

以下の属性は、サブクラス化またはインスタンスの属性値を変更して `TestLoader` をカスタマイズする場合に使用します:

testMethodPrefix

テストメソッドの名前と判断されるメソッド名の接頭語を示す文字列。デフォルト値は `'test'` です。

この値は `getTestCaseNames()` と全ての `loadTestsFrom*` () メソッドに影響を与えます。

sortTestMethodsUsing

`getTestCaseNames()` および全ての `loadTestsFrom*` () メソッドでメソッド名をソートする際に使用する比較関数。デフォルト値は組み込み関数 `cmp()` です。ソートを行なわないようにこの属性に `None` を指定することもできます。

suiteClass

テストのリストからテストスイートを構築する呼び出し可能オブジェクト。メソッドを持つ必要はありません。デフォルト値は `TestSuite` です。

この値は全ての `loadTestsFrom*` () メソッドに影響を与えます。

class unittest.TestResult

このクラスはどのテストが成功しどのテストが失敗したかという情報を収集するのに使います。

`TestResult` は、複数のテスト結果を記録します。`TestCase` クラスと `TestSuite` クラスのテスト結果を正しく記録しますので、テスト開発者が独自にテスト結果を管理する処理を開発する必要はありません。

`unittest` を利用したテストフレームワークでは、`TestRunner.run()` が返す `TestResult` インスタンスを参照し、テスト結果をレポートします。

`TestResult` インスタンスの以下の属性は、テストの実行結果を検査する際に使用することができます:

errors

`TestCase` と例外のトレースバック情報をフォーマットした文字列の 2 要素タプルからなるリスト。それぞれのタプルは予想外の例外を送出したテストに対応します。

バージョン 2.2 で変更: `sys.exc_info()` の結果ではなく、フォーマットしたトレースバックを保存するようにしました。

failures

`TestCase` と例外のトレースバック情報をフォーマットした文字列の 2 要素タプルからなるリスト。それぞれのタプルは `TestCase.assert*()` メソッドを使って見つけ出した失敗に対応します。

バージョン 2.2 で変更: `sys.exc_info()` の結果ではなく、フォーマットしたトレースバックを保存するようにしました。

skipped

`TestCase` インスタンスと理由の文字列の 2 要素タプルからなるリストを保持します。

バージョン 2.7 で追加.

expectedFailures

`TestCase` と例外のトレースバック情報をフォーマットした文字列の 2 要素タプルからなるリスト。それぞれのタプルは意図した失敗に対応します。

unexpectedSuccesses

意図した失敗のマークが付いていながら成功してしまった `TestCase` のインスタンスのリスト。

shouldStop

`True` が設定されると `stop()` によりテストの実行が停止します。

testsRun

これまでに実行したテストの総数です。

buffer

`True` が設定されると、`sys.stdout` と `sys.stderr` は、`startTest()` から `stopTest()` が呼ばれるまでの間バッファリングされます。実際に、結果が `sys.stdout` と `sys.stderr` に出力されるのは、テストが失敗するかエラーが発生した時になります。表示の際には、全ての失敗 / エラーメッセージが表示されます。

バージョン 2.7 で追加.

failfast

`True` が設定されると、`stop()` が最初の失敗もしくはエラーの時に呼び出され、テストの実行が終了します。

バージョン 2.7 で追加.

wasSuccessful()

これまでに実行したテストが全て成功していれば `True` を、それ以外なら `False` を返します。

stop()

このメソッドを呼び出して `TestResult` の `shouldStop` 属性に `True` をセットすることで、実行中のテストは中断しなければならないというシグナルを送ることができます。 `TestRunner` オブジェクトはこのフラグを尊重してそれ以上のテストを実行することなく復帰しなければなりません。

たとえばこの機能は、ユーザのキーボード割り込みを受け取って `TextTestRunner` クラスがテストフレームワークを停止させるのに使えます。 `TestRunner` の実装を提供する対話的なツールでも同じように使用することができます。

`TestResult` クラスの以下のメソッドは内部データ管理用のメソッドですが、対話的にテスト結果をレポートするテストツールを開発する場合などにはサブクラスで拡張することができます。

startTest(test)

`test` を実行する直前に呼び出されます。

stopTest(test)

`test` の実行直後に、テスト結果に関わらず呼び出されます。

startTestRun()

全てのテストが実行される前に一度だけ実行されます。

バージョン 2.7 で追加.

stopTestRun()

全てのテストが実行された後に一度だけ実行されます。

バージョン 2.7 で追加.

addError(test, err)

テスト `test` 実行中に、想定外の例外が発生した場合に呼び出されます。 `err` は `sys.exc_info()` が返すタプル (`type`, `value`, `traceback`) です。

デフォルトの実装では、タプル、(`test`, `formatted_err`) をインスタンスの `errors` 属性に追加します。ここで、`formatted_err` は、`err` から導出される、整形されたトレースバックです。

addFailure(test, err)

テストケース `test` が失敗した場合に呼び出されます。 `err` は `sys.exc_info()` が返すタプル (`type`, `value`, `traceback`) です。

デフォルトの実装では、タプル、(`test`, `formatted_err`) をインスタンスの `failures` 属性に追加します。ここで、`formatted_err` は、`err` から導出される、整形されたトレースバックです。

addSuccess(test)

テストケース `test` が成功した場合に呼び出されます。

デフォルトの実装では何もありません。

addSkip(test, reason)

`test` がスキップされた時に呼び出されます。 `reason` はスキップの際に渡された理由の文字列です。

デフォルトの実装では、`(test, reason)` のタプルをインスタンスの `skipped` 属性に追加します。

addExpectedFailure (*test, err*)

`expectedFailure()` のデコレータでマークされた *test* が失敗した時に呼び出されます。

デフォルトの実装では `(test, formatted_err)` のタプルをインスタンスの `expectedFailures` に追加します。ここで `formatted_err` は *err* から派生した整形されたトレースバックです。

addUnexpectedSuccess (*test*)

`expectedFailure()` のデコレータでマークされた *test* が成功した時に呼び出されます。

デフォルトの実装ではテストをインスタンスの `unexpectedSuccesses` 属性に追加します。

class `unittest.TextTestResult` (*stream, descriptions, verbosity*)

`TextTestRunner` に使用される `TestResult` の具象実装です。

バージョン 2.7 で追加: このクラスは以前 `_TextTestResult` という名前でした。以前の名前はエイリアスとして残っていますが非推奨です。

`unittest.defaultTestLoader`

`TestLoader` のインスタンスで、共用することが目的です。 `TestLoader` をカスタマイズする必要がなければ、新しい `TestLoader` オブジェクトを作らずにこのインスタンスを使用します。

class `unittest.TextTestRunner` (*stream=sys.stderr, descriptions=True, verbosity=1, failfast=False, buffer=False, resultclass=None*)

実行結果を標準エラーに出力する、単純なテストランナー。いくつかの設定項目がありますが、非常に単純です。グラフィカルなテスト実行アプリケーションでは、独自のテストランナーを作成してください。

makeResult ()

このメソッドは `run()` で使われる `TestResult` のインスタンスを返します。このメソッドは明示的に呼び出す必要はありませんが、サブクラスで `TestResult` をカスタマイズすることができます。

`makeResult()` は、`TextTestRunner` のコンストラクタで `resultclass` 引数として渡されたクラスもしくはコーラブルオブジェクトをインスタンス化します。 `resultclass` が指定されていない場合には、デフォルトで `TextTestResult` が使用されます。結果のクラスは以下の引数が渡されインスタンス化されます:

`stream, descriptions, verbosity`

`unittest.main` ([*module* [, *defaultTest* [, *argv* [, *testRunner* [, *testLoader* [, *exit* [, *verbosity* [, *failfast* [, *catchbreak* [, *buffer*]]]]]]]]])

module から複数のテストを読み込んで実行するためのコマンドラインプログラム。この関数を使えば、簡単に実行可能なテストモジュールを作成する事ができます。一番簡単なこの関数の使い方は、以下の行をテストスクリプトの最後に置くことです:

```
if __name__ == '__main__':
    unittest.main()
```

より詳細な情報は `verbosity` 引数を指定して実行すると得られます:

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

`defaultTest` 引数は、`argv` で指定されるテスト名がない場合に実行するテスト名です。これを指定しないか `None` を指定し、また `argv` から与えられない場合は、`module` 内で見つかる全てのテストを実行します。

`argv` 引数には、プログラムに渡されたオプションのリストを、最初の要素がプログラム名のままで渡せます。指定しないか `None` の場合は `sys.argv` が使われます。

引数、`testRunner` は、test runner class、あるいは、そのインスタンスのどちらでも構いません。でフォールトでは `main` はテストが成功したか失敗したかに対応した終了コードと共に `sys.exit()` を呼び出します。

`testLoader` 引数は `TestLoader` インスタンスでなければなりません。デフォルトは `defaultTestLoader` です。

`main` は、`exit=False` を指定する事で対話的なインタプリタから使用することもできます。この引数を指定すると、`sys.exit()` を呼ばずに、結果のみを出力します:

```
>>> from unittest import main
>>> main(module='test_module', exit=False)
```

`failfast`, `catchbreak`, `buffer` は、[コマンドラインオプション](#) にある同名のオプションと同じ効果のあるパラメータです。

`main` を呼び出すと、`TestProgram` のインスタンスが返されます。このインスタンスは、`result` 属性にテスト結果を保持します。

バージョン 2.7 で変更: パラメータ `exit`, `verbosity`, `failfast`, `catchbreak`, `buffer` が追加されました。

load_tests プロトコル

バージョン 2.7 で追加.

モジュールやパッケージには、`load_tests` と呼ばれる関数を実装できます。これにより、通常のテスト実行時やテストディスカバリ時のテストのロードされ方をカスタマイズできます。

テストモジュールが `load_tests` を定義していると、それが `TestLoader.loadTestsFromModule()` から呼ばれます。引数は以下です:

```
load_tests(loader, standard_tests, None)
```

これは `TestSuite` を返すべきです。

`loader` はローディングを行う `TestLoader` のインスタンスです。 `standard_tests` は、そのモジュールからデフォルトでロードされるテストです。これは、テストの標準セットのテストの追加や削除のみを行いたいテストモジュールに一般に使われます。第三引数は、パッケージをテストディスカバリの一部としてロードするときに使われます。

特定の `TestCase` クラスのセットからテストをロードする典型的な `load_tests` 関数は、このようになります:

```
test_cases = (TestCase1, TestCase2, TestCase3)

def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite
```

ディスカバリが開始されると、パッケージ名にマッチするパターンを、コマンドラインまたは `TestLoader.discover()` に与えることで、 `__init__.py` に `load_tests` があるか調べられます。

注釈: デフォルトのパターンは `'test*.py'` です。これは `'test'` で始まるすべての Python ファイルにマッチしますが、テストディレクトリにはマッチしません。

`'test*'` のようなパターンは、モジュールだけでなくテストパッケージにもマッチします。

パッケージ `__init__.py` が `load_tests` を定義していると、それが呼び出され、ディスカバリはそれ以上パッケージ内で続けられません。 `load_tests` が以下の引数で呼び出されます:

```
load_tests(loader, standard_tests, pattern)
```

これはパッケージ内のすべてのテストを表す `TestSuite` を返すべきです。(`standard_tests` には、 `__init__.py` から収集されたテストのみが含まれます。)

パターンは `load_tests` に渡されるので、パッケージは自由にテストディスカバリを継続 (必要なら変更) できます。テストパッケージに ' 何もしない ' `load_tests` 関数は次のようになります:

```
def load_tests(loader, standard_tests, pattern):
    # top level directory cached on loader instance
    this_dir = os.path.dirname(__file__)
    package_tests = loader.discover(start_dir=this_dir, pattern=pattern)
    standard_tests.addTests(package_tests)
    return standard_tests
```

25.3.8 クラスとモジュールのフィクスチャ

クラスレベルとモジュールレベルのフィクスチャが `TestSuite` に実装されました。テストスイートが新しいクラスのテストを始める時、以前のクラス (あれば) の `tearDownClass()` を呼び出し、その後新しい

クラスの `setUpClass()` を呼び出します。

同様に、今回のテストのモジュールが前回のテストとは異なる場合、以前のモジュールの `tearDownModule` を実行し、次に新しいモジュールの `setUpModule` を実行します。

すべてのテストが実行された後、最後の `tearDownClass` と `tearDownModule` が実行されます。

なお、共有フィクスチャは、テストの並列化などの [潜在的な] 機能と同時ににはうまくいかず、テストの分離を壊すので、気をつけて使うべきです。

unittest テストローダによるテスト作成のデフォルトの順序では、同じモジュールやクラスからのテストはすべて同じグループにまとめられます。これにより、`setUpClass` / `setUpModule` (など) は、一つのクラスやモジュールにつき一度だけ呼ばれます。この順序をバラバラにし、異なるモジュールやクラスのテストが並ぶようにすると、共有フィクスチャ関数は、一度のテストで複数回呼ばれるようになります。

共有フィクスチャは標準でない順序で実行されることを意図していません。共有フィクスチャをサポートしたくないフレームワークのために、`BaseTestSuite` がまだ存在しています。

共有フィクスチャ関数のいずれかで例外が発生した場合、そのテストはエラーとして報告されます。そのとき、対応するテストインスタンスが無いので (`TestCase` と同じインタフェースの) `_ErrorHandler` オブジェクトが生成され、エラーを表します。標準 unittest テストランナーを使っている場合はこの詳細は問題になりませんが、あなたがフレームワークの作者である場合は注意してください。

setUpClass と tearDownClass

これらはクラスメソッドとして実装されなければなりません:

```
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

基底クラスの `setUpClass` および `tearDownClass` を使いたいなら、それらを自分で呼び出さなければなりません。 `TestCase` の実装は空です。

`setUpClass` の中で例外が送出されたら、クラス内のテストは実行されず、`tearDownClass` も実行されません。スキップされたクラスは `setUpClass` も `tearDownClass` も実行されません。例外が `SkipTest` 例外であれば、そのクラスはエラーではなくスキップされたものとして報告されます。

setUpModule と tearDownModule

これらは関数として実装されなければなりません:

```
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

`setUpModule` の中で例外が送出されたら、モジュール内のテストは実行されず、`tearDownModule` も実行されません。例外が `SkipTest` 例外であれば、そのモジュールはエラーではなくスキップされたものとして報告されます。

25.3.9 シグナルハンドリング

`unittest` の `-c/--catch` コマンドラインオプションや、`unittest.main()` の `catchbreak` パラメタは、テスト実行中の control-C の処理をよりフレンドリーにします。中断捕捉動作を有効である場合、control-C が押されると、現在実行されているテストまで完了され、そのテストランが終わると今までの結果が報告されます。control-C がもう一度押されると、通常通り `KeyboardInterrupt` が送出されます。

シグナルハンドラを処理する control-c は、独自の `signal.SIGINT` ハンドラをインストールするコードやテストの互換性を保とうとします。`unittest` ハンドラが呼ばれ、それがインストールされた `signal.SIGINT` ハンドラでなければ、すなわちテスト中のシステムに置き換えられて移譲されたなら、それはデフォルトのハンドラを呼び出します。インストールされたハンドラを置き換えて委譲するようなコードは、通常その動作を期待するからです。`unittest` の control-c 処理を無効にしたいような個別のテストには、`removeHandler()` デコレータが使えます。

フレームワークの作者がテストフレームワーク内で control-c 処理を有効にするための、いくつかのユーティリティ関数があります。

`unittest.installHandler()`

control-c ハンドラをインストールします。(主にユーザが control-c を押したことにより) `signal.SIGINT` が受け取られると、登録した結果すべてに `stop()` が呼び出されます。

バージョン 2.7 で追加.

`unittest.registerResult(result)`

control-c 処理のために `TestResult` を登録します。結果を登録するとそれに対する弱参照が格納されるので、結果がガベージコレクトされるのを妨げません。

control-c 処理が有効でなければ、`TestResult` オブジェクトの登録には副作用がありません。ですからテストフレームワークは、処理が有効か無効かにかかわらず、作成する全ての結果を無条件に登録できます。

バージョン 2.7 で追加.

`unittest.removeResult(result)`

登録された結果を削除します。一旦結果が削除されると、control-c が押された際にその結果オブジェクトに対して `stop()` が呼び出されなくなります。

バージョン 2.7 で追加.

`unittest.removeHandler` (*function=None*)

引数なしで呼び出されると、この関数は Ctrl+C のシグナルハンドラを（それがインストールされていた場合）削除します。また、この関数はテストが実行されている間、Ctrl+C のハンドラを一時的に削除するテストデコレーターとしても使用できます。

```
@unittest.removeHandler
def test_signal_handling(self):
    ...
```

バージョン 2.7 で追加。

25.4 2to3 - Python 2 から 3 への自動コード変換

2to3 は、Python 2.x のソースコードを読み込み、一連の変換プログラムを適用して Python 3.x のコードに変換するプログラムです。標準ライブラリはほとんど全てのコードを取り扱うのに十分な変換プログラムを含んでいます。ただし 2to3 を構成している *lib2to3* は柔軟かつ一般的なライブラリなので、2to3 のために自分で変換プログラムを書くこともできます。*lib2to3* は、Python コードを自動編集する必要がある場合にも適用することができます。

25.4.1 2to3 の使用

2to3 は大抵の場合、Python インタープリターと共に、スクリプトとしてインストールされます。場所は、Python のルートディレクトリにある、`Tools/scripts` ディレクトリです。

2to3 に与える基本の引数は、変換対象のファイル、もしくは、ディレクトリのリストです。ディレクトリの場合は、Python ソースコードを再帰的に探索します。

Python 2.x のサンプルコード、`example.py` を示します:

```
def greet(name):
    print "Hello, {0}!".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

It can be converted to Python 3.x code via 2to3 on the command line:

```
$ 2to3 example.py
```

A diff against the original source file is printed. 2to3 can also write the needed modifications right back to the source file. (A backup of the original file is made unless `-n` is also given.) Writing the changes back is enabled with the `-w` flag:

```
$ 2to3 -w example.py
```

変換後、`example.py` は以下ようになります:

```
def greet(name):  
    print("Hello, {0}!".format(name))  
print("What's your name?")  
name = input()  
greet(name)
```

変換処理を通じて、コメントと、インデントは保存されます。

By default, 2to3 runs a set of *predefined fixers*. The `-l` flag lists all available fixers. An explicit set of fixers to run can be given with `-f`. Likewise the `-x` explicitly disables a fixer. The following example runs only the `imports` and `has_key` fixers:

```
$ 2to3 -f imports -f has_key example.py
```

This command runs every fixer except the `apply` fixer:

```
$ 2to3 -x apply example.py
```

Some fixers are *explicit*, meaning they aren't run by default and must be listed on the command line to be run. Here, in addition to the default fixers, the `idioms` fixer is run:

```
$ 2to3 -f all -f idioms example.py
```

ここで、`all` を指定することで、全てのデフォルトの変換プログラムを有効化できることに注意して下さい。

2to3 がソースコードに修正すべき点を見つけても、自動的に修正できない場合もあります。この場合、2to3 はファイルの変更点の下に警告を表示します。3.x に準拠したコードにするために、あなたはこの警告に対処しなくてはなりません。

2to3 can also refactor doctests. To enable this mode, use the `-d` flag. Note that *only* doctests will be refactored. This also doesn't require the module to be valid Python. For example, doctest like examples in a reST document could also be refactored with this option.

The `-v` option enables output of more information on the translation process.

Since some print statements can be parsed as function calls or statements, 2to3 cannot always read files containing the print function. When 2to3 detects the presence of the `from __future__ import print_function` compiler directive, it modifies its internal grammar to interpret `print()` as a function. This change can also be enabled manually with the `-p` flag. Use `-p` to run fixers on code that already has had its print statements converted.

The `-o` or `--output-dir` option allows specification of an alternate directory for processed output files to be written to. The `-n` flag is required when using this as backup files do not make sense when not overwriting the input files.

バージョン 2.7.3 で追加: The `-o` option was added.

The `-W` or `--write-unchanged-files` flag tells 2to3 to always write output files even if no changes were required to the file. This is most useful with `-o` so that an entire Python source tree is copied with translation from

one directory to another. This option implies the `-w` flag as it would not make sense otherwise.

バージョン 2.7.3 で追加: The `-w` flag was added.

The `--add-suffix` option specifies a string to append to all output filenames. The `-n` flag is required when specifying this as backups are not necessary when writing to different filenames. Example:

```
$ 2to3 -n -W --add-suffix=3 example.py
```

とすれば変換後ファイルは `example.py3` として書き出されます。

バージョン 2.7.3 で追加: The `--add-suffix` option was added.

To translate an entire project from one directory tree to another use:

```
$ 2to3 --output-dir=python3-version/mycode -W -n python2-version/mycode
```

25.4.2 変換プログラム

コード変形の各ステップは変換プログラムに隠蔽されます。 `2to3 -l` コマンドは変換プログラムのリストを表示します。 [上記](#) の通り、それぞれの変換プログラムを個別に有効化したり無効化したりすることができません。ここではそれらをさらに詳細に説明します。

apply

`apply()` の使用を削除します。例えば `apply(function, *args, **kwargs)` は `function(*args, **kwargs)` に変換されます。

asserts

廃止になった `unittest` メソッド名を新しい名前に置換します。

対象	変換先
<code>failUnlessEqual(a, b)</code>	<code>assertEqual(a, b)</code>
<code>assertEquals(a, b)</code>	<code>assertEqual(a, b)</code>
<code>failIfEqual(a, b)</code>	<code>assertNotEqual(a, b)</code>
<code>assertNotEquals(a, b)</code>	<code>assertNotEqual(a, b)</code>
<code>failUnless(a)</code>	<code>assertTrue(a)</code>
<code>assert_(a)</code>	<code>assertTrue(a)</code>
<code>failIf(a)</code>	<code>assertFalse(a)</code>
<code>failUnlessRaises(exc, cal)</code>	<code>assertRaises(exc, cal)</code>
<code>failUnlessAlmostEqual(a, b)</code>	<code>assertAlmostEqual(a, b)</code>
<code>assertAlmostEquals(a, b)</code>	<code>assertAlmostEqual(a, b)</code>
<code>failIfAlmostEqual(a, b)</code>	<code>assertNotAlmostEqual(a, b)</code>
<code>assertNotAlmostEquals(a, b)</code>	<code>assertNotAlmostEqual(a, b)</code>

basestring

`basestring` を `str` に変換します。

buffer

`buffer` を `memoryview` に変換します。 `memoryview` API は `buffer` と似ているものの厳密に同じではないので、この変換プログラムはオプションです。

dict

辞書をイテレートするメソッドを修正します。 `dict.iteritems()` は `dict.items()` に、 `dict.iterkeys()` は `dict.keys()` に、 `dict.itervalues()` は `dict.values()` に変換されます。同様に `dict.viewitems()`、`dict.viewkeys()` `dict.viewvalues()` はそれぞれ `dict.items()`、`dict.keys()`、`dict.values()` に変換されます。また、 `list` の呼び出しの中で `dict.items()`、`dict.keys()`、`dict.values()` を使用している場合はそれをラップします。

except

`except X, T` を `except X as T` に変換します。

exec

`exec` 文を `exec()` 関数に変換します。

execfile

`execfile()` の使用を削除します。 `execfile()` への引数は `open()`、`compile()`、`exec()` の呼び出しでラップされます。

exitfunc

`sys.exitfunc` への代入を `atexit` モジュールの使用に変更します。

filter

`list` 呼び出しの中で `filter()` を使用している部分をラップします。

funcattrs

名前が変更された関数の属性を修正します。例えば `my_function.func_closure` は `my_function.__closure__` に変換されます。

future

`from __future__ import new_feature` 文を削除します。

getcwdu

`os.getcwdu()` を `os.getcwd()` に置き換えます。

has_key

`dict.has_key(key)` を `key in dict` に変更します。

idioms

このオプションの変換プログラムは、Python コードをより Python らしい書き方にするいくつかの変形を行います。 `type(x) is SomeClass` や `type(x) == SomeClass` のような型の比較は `isinstance(x, SomeClass)` に変換されます。 `while 1` は `while True` になります。また、適切な場所では `sorted()` が使われるようにします。例えば、このブロックは

```
L = list(some_iterable)
L.sort()
```

次のように変更されます


```
L = sorted(some_iterable)
```

import

暗黙の相対インポート (sibling imports) を検出して、明示的な相対インポート (relative imports) に変換します。

imports

標準ライブラリ中のモジュール名の変更を扱います。

imports2

標準ライブラリ中の別のモジュール名の変更を扱います。単に技術的な制約のために *imports* とは別になっています。

input

`input(prompt)` を `eval(input(prompt))` に変換します。

intern

intern() を `sys.intern()` に変換します。

isinstance

isinstance() の第 2 引数の重複を修正します。例えば `isinstance(x, (int, int))` は `isinstance(x, (int))` に変換されます。

itertools_imports

itertools.ifilter(), *itertools.izip()*, *itertools.imap()* のインポートを削除します。また *itertools.ifilterfalse()* のインポートを `itertools.filterfalse()` に変換します。

itertools

itertools.ifilter(), *itertools.izip()*, *itertools.imap()* を使っている箇所を同等の組み込み関数で置き換えます。 *itertools.ifilterfalse()* は `itertools.filterfalse()` に変換されます。

long

long を *int* に変更します。

map

`list` 呼び出しの中の *map()* をラップします。また、`map(None, x)` を `list(x)` に変換します。
`from future.builtins import map` を使うと、この変換プログラムを無効にできます。

metaclass

古いメタクラス構文 (クラス定義中の `__metaclass__ = Meta`) を、新しい構文 (`class X(metaclass=Meta)`) に変換します。

methodattrs

古いメソッドの属性名を修正します。例えば `meth.im_func` は `meth.__func__` に変換されます。

ne

古い不等号の構文 `<>` を `!=` に変換します。

next

イテレータの `next()` メソッドの使用を `next()` 関数に変換します。また `next()` メソッドを `__next__()` に変更します。

nonzero

`__nonzero__()` を `__bool__()` に変更します。

numliterals

8 進数リテラルを新しい構文に変換します。

paren

リスト内包表記で必要になる括弧を追加します。例えば `[x for x in 1, 2]` は `[x for x in (1, 2)]` になります。

print

`print` 文を `print()` 関数に変換します。

raise

`raise E, V` を `raise E(V)` に、`raise E, V, T` を `raise E(V).with_traceback(T)` に変換します。例外の代わりにタプルを使用することは Python 3 で削除されたので、`E` がタプルならこの変換は不正確になります。

raw_input

`raw_input()` を `input()` に変換します。

reduce

`reduce()` が `functools.reduce()` に移動されたことを扱います。

renames

`sys.maxint` を `sys.maxsize` に変更します。

repr

バッククォートを使った `repr` を `repr()` 関数に置き換えます。

set_literal

`set` コンストラクタの使用を `set` リテラルに置換します。この変換プログラムはオプションです。

standarderror

`StandardError` を `Exception` に変更します。

sys_exc

廃止された `sys.exc_value`, `sys.exc_type`, `sys.exc_traceback` の代わりに `sys.exc_info()` を使うように変更します。

throw

ジェネレータの `throw()` メソッドの API 変更を修正します。

tuple_params

関数定義における暗黙的なタプルパラメータの展開を取り除きます。この変換プログラムによって一時変数が追加されます。

types

`types` モジュールのいくつかのメンバオブジェクトが削除されたことによって壊れたコードを修正します。

unicode

`unicode` を `str` に変更します。

urllib

`urllib` と `urllib2` が `urllib` パッケージに変更されたことを扱います。

ws_comma

コンマ区切りの要素から余計な空白を取り除きます。この変換プログラムはオプションです。

xrange

`xrange()` を `range()` に変更して、既存の `range()` 呼び出しを `list` でラップします。

xreadlines

`for x in file.xreadlines()` を `for x in file` に変更します。

zip

`list` 呼び出しの中で使われている `zip()` をラップします。これは `from future_builtins import zip` が見つかった場合は無効にされます。

25.4.3 lib2to3 - 2to3's library

注釈: `lib2to3` API は安定しておらず、将来、劇的に変更されるかも知れないと考えるべきです。

25.5 test — Python 用回帰テストパッケージ

注釈: `test` パッケージは、Python の内部で使用するためのものです。ドキュメントが書かれたのは Python のコア開発者の利便性を考えてのことです。Python の標準ライブラリ以外でこのパッケージを使用することは、あまりお勧めできません。ここで触られているコードは、Python のリリースの間に予告なく変更されたり削除されたりすることがあります。

`test` パッケージには、Python 用の全ての回帰テストの他に、`test.support` モジュールと `test.regrtest` モジュールが入っています。`test.support` はテストを充実させるために使い、`test.regrtest` はテストスイートを実行するのに使います。

`test` パッケージ内のモジュールのうち、名前が `test_` で始まるものは、特定のモジュールや機能に対するテストスイートです。新しいテストはすべて `unittest` か `doctest` モジュールを使って書くようにしてください。古いテストのいくつかは、`sys.stdout` への出力を比較する「従来の」テスト形式になっていますが、この形式のテストは廃止予定です。

参考:

`unittest` モジュール PyUnit 回帰テストを書く。

`doctest` モジュール ドキュメンテーション文字列に埋め込まれたテスト。

25.5.1 `test` パッケージのためのユニットテストを書く

`unittest` モジュールを使ってテストを書く場合、幾つかのガイドラインに従うことが推奨されます。1 つは、テストモジュールの名前を、`test_` で始め、テスト対象となるモジュール名で終えることです。テストモジュール中のテストメソッドは名前を `test_` で始めて、そのメソッドが何をテストしているかという説明で終わります。これはテスト実行プログラムが、そのメソッドをテストメソッドとして認識するために必要です。また、テストメソッドにはドキュメンテーション文字列を入れるべきではありません。コメント（例えば `# True` あるいは `False` だけを返すテスト関数）を使用して、テストメソッドのドキュメントを記述してください。これは、ドキュメンテーション文字列が存在する場合はその内容が出力されてしまうため、どのテストを実行しているのかをいちいち表示したくないからです。

以下のような決まり文句を使います:

```
import unittest
from test import support

class MyTestCase1(unittest.TestCase):

    # Only use setUp() and tearDown() if necessary

    def setUp(self):
        ... code to execute in preparation for tests ...

    def tearDown(self):
        ... code to execute to clean up after tests ...

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...

    def test_feature_two(self):
        # Test feature two.
        ... testing code ...

    ... more test methods ...

class MyTestCase2(unittest.TestCase):
    ... same structure as MyTestCase1 ...

... more test classes ...

def test_main():
    support.run_unittest(MyTestCase1,
                        MyTestCase2,
```

(次のページに続く)

(前のページからの続き)

```

        ... list other tests ...
    )

if __name__ == '__main__':
    test_main()

```

この定型的なコードによって、テストスイートを `regtest.py` から起動できると同時に、スクリプト自体からも実行できるようになります。

回帰テストの目的はコードを解き明かすことです。そのためには以下のいくつかのガイドラインに従ってください:

- テストスイートから、すべてのクラス、関数および定数を実行するべきです。これには外部に公開される外部 API だけでなく「プライベートな」コードも含まれます。
- ホワイトボックス・テスト（対象のコードの詳細を元にテストを書くこと）を推奨します。ブラックボックス・テスト（公開されるインタフェース仕様だけをテストすること）は、すべての境界条件を確実にテストするには完全ではありません。
- すべての取りうる値を、無効値も含めてテストするようにしてください。そのようなテストを書くことで、全ての有効値が通るだけでなく、不適切な値が正しく処理されることも確認できます。
- コード内のできる限り多くのパスを網羅してください。分岐するように入力を調整したテストを書くことで、コードの多くのパスをたどることができます。
- テスト対象のコードにバグが発見された場合は、明示的にテスト追加するようにしてください。そのようなテストを追加することで、将来コードを変更した際にエラーが再発することを防止できます。
- テストの後始末（例えば一時ファイルをすべて閉じたり削除したりすること）を必ず行ってください。
- テストがオペレーティングシステム特定の状況に依存する場合、テスト開始時に条件を満たしているかを検証してください。
- `import` するモジュールをできるかぎり少なくし、可能な限り早期に `import` を行ってください。そうすることで、テストの外部依存性を最小限にし、モジュールの `import` による副作用から生じる変則的な動作を最小限にできます。
- できる限りテストコードを再利用するようにしましょう。時として、入力の違いだけを記述すれば良くなるくらい、テストコードを小さくすることができます。例えば以下のように、サブクラスで入力を指定することで、コードの重複を最小化することができます:

```

class TestFuncAcceptsSequences(unittest.TestCase):

    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequences):
    arg = [1, 2, 3]

```

(次のページに続く)

(前のページからの続き)

```
class AcceptStrings (TestFuncAcceptsSequences) :  
    arg = 'abc'  
  
class AcceptTuples (TestFuncAcceptsSequences) :  
    arg = (1, 2, 3)
```

参考:

Test Driven Development コードより前にテストを書く方法論に関する Kent Beck の著書。

25.5.2 コマンドラインインタフェースを利用してテストを実行する

`test.regrtest` はスクリプトとして Python の回帰テストスイートを実行できます。 `-m` オプションを利用して、 `python -m test.regrtest` として実行します。スクリプトを実行すると、自動的に `test` パッケージ内のすべての回帰テストを実行し始めます。パッケージ内の名前が `test_` で始まる全モジュールを見つけ、それをインポートし、もしあるなら関数 `test_main()` を実行してテストを行います。実行するテストの名前もスクリプトに渡される可能性があります。単一の回帰テストを指定 (`python -m test.regrtest test_spam`) すると、出力を最小限にします。テストが成功したかあるいは失敗したかだけを出力するので、出力は最小限になります。

直接 `test.regrtest` を実行すると、テストに利用するリソースを設定できます。これを行うには、 `-u` コマンドラインオプションを使います。 `-u` のオプションに `all` を指定すると、すべてのリソースを有効にします: `python -m test.regrtest -uall`。(よくある場合ですが) 何か一つを除く全てが必要な場合、カンマで区切った不要なリソースのリストを `all` の後に並べます。コマンド `python -m test.regrtest -uall,-audio,-largefile` とすると、 `audio` と `largefile` リソースを除く全てのリソースを使って `test.regrtest` を実行します。すべてのリソースのリストと追加のコマンドラインオプションを出力するには、 `python -m test.regrtest -h` を実行してください。

テストを実行しようとするプラットフォームによっては、回帰テストを実行する別の方法があります。 Unix では、Python をビルドしたトップレベルディレクトリで `make test` を実行できます。 Windows 上では、PCBuild ディレクトリから `rt.bat` を実行すると、すべての回帰テストを実行します。

バージョン 2.7.14 で変更: The `test` package can be run as a script: `python -m test`. This works the same as running the `test.regrtest` module.

25.6 test.support — Utility functions for tests

注釈: The `test.test_support` module has been renamed to `test.support` in Python 3.x and 2.7.14. The name `test.test_support` has been retained as an alias in 2.7.

The `test.support` module provides support for Python's regression tests.

このモジュールは次の例外を定義しています:

exception `test.support.TestFailed`

テストが失敗したとき送出される例外です。これは、`unittest` ベースのテストでは廃止予定で、`unittest.TestCase` の `assertXXX` メソッドが推奨されます。

exception `test.support.ResourceDenied`

`unittest.SkipTest` のサブクラスです。(ネットワーク接続のような) リソースが利用できないとき送出されます。`requires()` 関数によって送出されます。

`test.support` モジュールでは、以下の定数を定義しています:

`test.support.verbose`

冗長な出力が有効な場合は `True` です。実行中のテストについてのより詳細な情報が欲しいときにチェックします。`verbose` は `test.regrtest` によって設定されます。

`test.support.have_unicode`

Unicode サポートが利用可能ならば `True` になります。

`test.support.is_jython`

実行中のインタプリタが Jython ならば `True` になります。

`test.support.TESTFN`

テンポラリファイルの名前として安全に利用できる名前に設定されます。作成した一時ファイルは全て閉じ、`unlink` (削除) しなければなりません。

`test.support.TEST_HTTP_URL`

Define the URL of a dedicated HTTP server for the network tests.

`test.support` モジュールでは、以下の関数を定義しています:

`test.support.forget (module_name)`

モジュール名 `module_name` を `sys.modules` から取り除き、モジュールのバイトコンパイル済みファイルを全て削除します。

`test.support.is_resource_enabled (resource)`

`resource` が有効で利用可能ならば `True` を返します。利用可能なリソースのリストは、`test.regrtest` がテストを実行している間のみ設定されます。

`test.support.requires (resource[, msg])`

`resource` が利用できなければ、`ResourceDenied` を送出します。その場合、`msg` は `ResourceDenied` の引数になります。`__name__` が `"__main__"` である関数にから呼び出された場合には常に `True` を返します。テストを `test.regrtest` から実行するときに使われます。

`test.support.findfile (filename)`

`filename` という名前のファイルへのパスを返します。一致するものが見つからなければ、`filename` 自体を返します。`filename` 自体もファイルへのパスでありえるので、`filename` が返っても失敗ではありません。

`test.support.run_unittest (*classes)`

渡された `unittest.TestCase` サブクラスを実行します。この関数は名前が `test_` で始まるメソッド

ドを探して、テストを個別に実行します。

引数に文字列を渡すことも許可されています。その場合、文字列は `sys.module` のキーでなければなりません。指定された各モジュールは、`unittest.TestLoader.loadTestsFromModule()` でスキャンされます。この関数は、よく次のような `test_main()` 関数の形で利用されます。

```
def test_main():
    support.run_unittest(__name__)
```

この関数は、名前指定されたモジュールの中の全ての定義されたテストを実行します。

`test.support.check_warnings(*filters, quiet=True)`

`warning` が正しく発行されているかどうかチェックする、`warnings.catch_warnings()` を使いやすくするラッパーです。これは、`warnings.simplefilter()` を `always` に設定して、記録された結果を自動的に検証するオプションと共に `warnings.catch_warnings(record=True)` を呼ぶのと同様です。

`check_warnings` は `("message regexp", WarningCategory)` の形をした 2 要素タプルをポジション引数として受け取ります。1 つ以上の `filters` が与えられた場合や、オプションのキーワード引数 `quiet` が `False` の場合、警告が期待通りであるかどうかをチェックします。指定された各 `filter` は最低でも 1 回は囲われたコード内で発生した警告とマッチしなければテストが失敗しますし、指定されたどの `filter` ともマッチしない警告が発生してもテストが失敗します。前者のチェックを無効にするには、`quiet` を `True` にします。

引数が 1 つもない場合、デフォルトでは次のようになります：

```
check_warnings(("", Warning), quiet=True)
```

この場合、全ての警告は補足され、エラーは発生しません。

コンテキストマネージャーに入る時、`WarningRecorder` インスタンスが返されます。このレコーダーオブジェクトの `warnings` 属性から、`catch_warnings()` から得られる警告のリストを取得することができます。便利さのために、レコーダーオブジェクトから直接、一番最近に発生した警告を表すオブジェクトの属性にアクセスできます (以下にある例を参照してください)。警告が 1 つも発生しなかった場合、それらの全ての属性は `None` を返します。

レコーダーオブジェクトの `reset()` メソッドは警告リストをクリアします。

コンテキストマネージャーは次のように使います：

```
with check_warnings(("assertion is always true", SyntaxWarning),
                    ("", UserWarning)):
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

この場合、どちらの警告も発生しなかった場合や、それ以外の警告が発生した場合は、`check_warnings()` はエラーを発生させます。

警告が発生したかどうかだけでなく、もっと詳しいチェックが必要な場合は、次のようなコードになります：

```

with check_warnings(quiet=True) as w:
    warnings.warn("foo")
    assert str(w.args[0]) == "foo"
    warnings.warn("bar")
    assert str(w.args[0]) == "bar"
    assert str(w.warnings[0].args[0]) == "foo"
    assert str(w.warnings[1].args[0]) == "bar"
    w.reset()
    assert len(w.warnings) == 0

```

全ての警告をキャプチャし、テストコードがその警告を直接テストします。

バージョン 2.6 で追加。

バージョン 2.7 で変更: 新しいオプション引数 *filters* と *quiet*

`test.support.check_py3k_warnings(*filters, quiet=False)`

`check_warnings()` と似ていますが、Python 3 互換性警告のためのものです。 `sys.py3kwarning == 1` の時、警告が実際に発生していることをチェックします。 `sys.py3kwarning == 0` の時、警告が発生していないことをチェックします。ポジション引数として ("message regexp", `WarningCategory`) の形をした 2 要素タプルを受け取ります。オプション引数 *quiet* が `True` のとき、filter になにもマッチしなくても失敗しません。引数がない場合は次と同じになります:

```
check_py3k_warnings((" ", DeprecationWarning), quiet=False)
```

バージョン 2.7 で追加。

`test.support.captured_stdout()`

これは、with 文の body で `sys.stdout` として `StringIO.StringIO` オブジェクトを利用するコンテキストマネージャーです。このオブジェクトは、with 文の `as` 節で受け取ることができます。

使用例:

```

with captured_stdout() as s:
    print "hello"
assert s.getvalue() == "hello\n"

```

バージョン 2.6 で追加。

`test.support.import_module(name, deprecated=False)`

この関数は *name* で指定されたモジュールを import して返します。通常の import と異なり、この関数はモジュールを import できなかった場合に `unittest.SkipTest` 例外を発生させます。

deprecated が `True` の場合、import 中はモジュールとパッケージの廃止メッセージが抑制されます。

バージョン 2.7 で追加。

`test.support.import_fresh_module(name, fresh=(), blocked=(), deprecated=False)`

この関数は、*name* で指定された Python モジュールを、import 前に `sys.modules` から削除することで新規に import してそのコピーを返します。 `reload()` 関数と違い、もとのモジュールはこの操作によって影響を受けません。

fresh は、同じように `import` 前に `sys.modules` から削除されるモジュール名の iterable です。

blocked もモジュール名の iterable で、`import` 中にモジュールキャッシュ内でその名前を 0 に置き換えることで、そのモジュールを `import` しようとするとき `ImportError` を発生させます。

指定されたモジュールと *fresh* や *blocked* 引数内のモジュール名は `import` 前に保存され、`fresh import` が完了したら `sys.modules` に戻されます。

deprecated が `True` の場合、`import` 中はモジュールとパッケージの廃止メッセージが抑制されます。

この関数はモジュールを `import` できなかった場合に `unittest.SkipTest` 例外を送出します。

使用例:

```
# Get copies of the warnings module for testing without
# affecting the version being used by the rest of the test suite
# One copy uses the C implementation, the other is forced to use
# the pure Python fallback implementation
py_warnings = import_fresh_module('warnings', blocked=['_warnings'])
c_warnings = import_fresh_module('warnings', fresh=['_warnings'])
```

バージョン 2.7 で追加.

`test.support` モジュールでは、以下のクラスを定義しています:

class `test.support.TransientResource` (`exc[, **kwargs]`)

このクラスのインスタンスはコンテキストマネージャーで、指定された型の例外が発生した場合に `ResourceDenied` 例外を発生させます。キーワード引数は全て、`with` 文の中で発生した全ての例外の属性名/属性値と比較されます。全てのキーワード引数が例外の属性に一致した場合に、`ResourceDenied` 例外が発生します。

バージョン 2.6 で追加.

class `test.support.EnvironmentVarGuard`

一時的に環境変数をセット・アンセットするためのクラスです。このクラスのインスタンスはコンテキストマネージャーとして利用されます。また、`os.environ` に対する参照・更新を行う完全な辞書のインタフェースを持ちます。コンテキストマネージャーが終了した時、このインスタンス経由で環境変数へ行った全ての変更はロールバックされます。

バージョン 2.6 で追加.

バージョン 2.7 で変更: 辞書のインタフェースを追加しました。

`EnvironmentVarGuard.set` (`envvar, value`)

一時的に、`envvar` を `value` にセットします。

`EnvironmentVarGuard.unset` (`envvar`)

一時的に `envvar` をアンセットします。

class `test.support.WarningsRecorder`

ユニットテスト時に warning を記録するためのクラスです。上の、`check_warnings()` のドキュメントを参照してください。

バージョン 2.6 で追加.

第 26 章

デバッグとプロファイル

ここに含まれるライブラリは Python での開発を手助けするものです。デバッガはコードのステップ実行や、スタックフレームの解析、ブレークポイントの設定などを可能にします。プロファイラはコードを実行して実行時間の詳細を晒すことでプログラムのボトルネックを見つけることを可能にします。

26.1 bdb — デバッガーフレームワーク

ソースコード: [Lib/bdb.py](#)

`bdb` モジュールは、ブレークポイントを設定したり、デバッガー経由で実行を管理するような、基本的なデバッガー機能を提供します。

以下の例外が定義されています:

exception `bdb.BdbQuit`

`Bdb` クラスが、デバッガーを終了させるために投げる例外。

`bdb` モジュールは 2 つのクラスを定義しています:

class `bdb.Breakpoint` (*self, file, line, temporary=0, cond=None, funcname=None*)

このクラスはテンポラリブレークポイント、無視するカウント、無効化と再有効化、条件付きブレークポイントを実装しています。

ブレークポイントは `bpybnumber` という名前のリストで番号によりインデックスされ、`bplist` により (`file`, `line`) の形でインデックスされます。 `bpybnumber` は `Breakpoint` クラスのインスタンスを指しています。一方 `bplist` は、同じ行に複数のブレークポイントが設定される場合があるので、インスタンスのリストを指しています。

ブレークポイントを作るとき、設定されるファイル名は正規化されていなければなりません。 `funcname` が設定されたとき、ブレークポイントはその関数の最初の行が実行されたときにヒットカウントにカウントされます。条件付ブレークポイントは毎回カウントされます。

`Breakpoint` インスタンスは以下のメソッドを持ちます:

deleteMe()

このブレークポイントをファイル/行に関連付けられたリストから削除します。このブレークポイントがその行に設定された最後のブレークポイントだった場合、そのファイル/行に対するエントリ自体を削除します。

enable()

このブレークポイントを有効にします。

disable()

このブレークポイントを無効にします。

pprint([out])

このブレークポイントに関するすべての情報を表示します。

- ブレークポイント番号。
- テンポラリブレークポイントかどうか。
- ファイル/行の位置。
- ブレークする条件。
- 次の N 回無視されるか。
- ヒットカウント。

class bdb.Bdb (skip=None)

Bdb クラスは一般的な Python デバッガーの基本クラスとして振舞います。

このクラスはトレース機能の詳細を扱います。ユーザーとのインタラクションは、派生クラスが実装すべきです。標準ライブラリのデバッガクラス (*pdb.Pdb*) がその利用例です。

skip 引数は、もし与えられたならグロブ形式のモジュール名パターンの iterable でなければなりません。デバッガはこれらのパターンのどれかにマッチするモジュールで発生したフレームにステップインしなくなります。フレームが特定のモジュールで発生したかどうかは、フレームのグローバル変数の `__name__` によって決定されます。

バージョン 2.7 で追加: *skip* 引数が追加されました。

以下の *Bdb* のメソッドは、通常オーバーライドする必要はありません。

canonic(filename)

標準化されたファイル名を取得するための補助関数。標準化されたファイル名とは、(大文字小文字を区別しないファイルシステムにおいて) 大文字小文字を正規化し、絶対パスにしたものです。ファイル名が "`<`" と "`>`" で囲まれていた場合はそれを取り除いたものです。

reset()

`botframe`, `stopframe`, `returnframe`, `quitting` 属性を、デバッグを始められる状態に設定します。

trace_dispatch(frame, event, arg)

この関数は、デバッグされているフレームのトレース関数としてインストールされます。戻り値は

新しいトレース関数 (殆どの場合はこの関数自身) です。

デフォルトの実装は、実行しようとしている *event* (文字列として渡されます) の種類に基づいてフレームのディスパッチ方法を決定します。 *event* は次のうちのどれかです:

- "line": 新しい行を実行しようとしています。
- "call": 関数が呼び出されているか、別のコードブロックに入ります。
- "return": 関数が別のコードブロックから return しようとしています。
- "exception": 例外が発生しました。
- "c_call": C 関数を呼び出そうとしています。
- "c_return": C 関数から return しました。
- "c_exception": C 関数が例外を発生させました。

Python のイベントに対しては、以下の専用の関数群が呼ばれます。C のイベントに対しては何もしません。

arg 引数は以前のイベントに依存します。

トレース関数についてのより詳しい情報は、 `sys.settrace()` のドキュメントを参照してください。コードとフレームオブジェクトについてのより詳しい情報は、 `types` を参照してください。

`dispatch_line(frame)`

デバッガーが現在の行で止まるべきであれば、 `user_line()` メソッド (サブクラスでオーバーライドされる) を呼び出します。 `Bdb.quitting` フラグ (`user_line()` から設定できます) が設定されていた場合、 `BdbQuit` 例外を発生させます。このスコープのこれからのトレースのために、 `trace_dispatch()` メソッドの参照を返します。

`dispatch_call(frame, arg)`

デバッガーがこの関数呼び出しで止まるべきであれば、 `user_call()` メソッド (サブクラスでオーバーライドされる) を呼び出します。 `Bdb.quitting` フラグ (`user_call()` から設定できます) が設定されていた場合、 `BdbQuit` 例外を発生させます。このスコープのこれからのトレースのために、 `trace_dispatch()` メソッドの参照を返します。

`dispatch_return(frame, arg)`

デバッガーがこの関数からのリターンで止まるべきであれば、 `user_return()` メソッド (サブクラスでオーバーライドされる) を呼び出します。 `Bdb.quitting` フラグ (`user_return()` から設定できます) が設定されていた場合、 `BdbQuit` 例外を発生させます。このスコープのこれからのトレースのために、 `trace_dispatch()` メソッドの参照を返します。

`dispatch_exception(frame, arg)`

デバッガーがこの例外発生で止まるべきであれば、 `user_exception()` メソッド (サブクラスでオーバーライドされる) を呼び出します。 `Bdb.quitting` フラグ (`user_exception()` から設定できます) が設定されていた場合、 `BdbQuit` 例外を発生させます。このスコープのこれからのトレースのために、 `trace_dispatch()` メソッドの参照を返します。

通常、継承クラスは以下のメソッド群をオーバーライドしません。しかし、停止やブレークポイント機能を再定義したい場合には、オーバーライドすることもあります。

stop_here (*frame*)

このメソッドは *frame* がコールスタック中で *botframe* よりも下にあるかチェックします。
botframe はデバッグを開始したフレームです。

break_here (*frame*)

このメソッドは、*frame* に属するファイル名と行に、あるいは、少なくとも現在の関数にブレークポイントがあるかどうかをチェックします。ブレークポイントがテンポラリブレークポイントだった場合、このメソッドはそのブレークポイントを削除します。

break_anywhere (*frame*)

このメソッドは、現在のフレームのファイル名の中にブレークポイントが存在するかどうかをチェックします。

継承クラスはデバッガー操作をするために以下のメソッド群をオーバーライドするべきです。

user_call (*frame*, *argument_list*)

このメソッドは、呼ばれた関数の中でブレークする必要がある可能性がある場合に、*dispatch_call()* から呼び出されます。

user_line (*frame*)

このメソッドは、*stop_here()* か *break_here()* が True を返したときに、*dispatch_line()* から呼び出されます。

user_return (*frame*, *return_value*)

このメソッドは、*stop_here()* が True を返したときに、*dispatch_return()* から呼び出されます。

user_exception (*frame*, *exc_info*)

このメソッドは、*stop_here()* が True を返したときに、*dispatch_exception()* から呼び出されます。

do_clear (*arg*)

ブレークポイントがテンポラリブレークポイントだったときに、それをどう削除するかを決定します。

継承クラスはこのメソッドを実装しなければなりません。

継承クラスとクライアントは、ステップ状態に影響を及ぼすために以下のメソッドを呼び出すことができます。

set_step ()

コードの次の行でストップします。

set_next (*frame*)

与えられたフレームかそれより下 (のフレーム) にある、次の行でストップします。

set_return (*frame*)

指定されたフレームから抜けるときにストップします。

set_until (*frame*)

現在の行番号よりも大きい行番号に到達したとき、あるいは、現在のフレームから戻るときにストップします。

set_trace ([*frame*])

frame からデバッグを開始します。*frame* が指定されなかった場合、デバッグは呼び出し元のフレームから開始します。

set_continue ()

ブレークポイントに到達するか終了したときにストップします。もしブレークポイントが1つも無い場合、システムのトレース関数を `None` に設定します。

set_quit ()

`quitting` 属性を `True` に設定します。これにより、次回の `dispatch_*()` メソッドのどれかの呼び出しで、`BdbQuit` 例外を発生させます。

継承クラスとクライアントは以下のメソッドをブレークポイント操作に利用できます。これらのメソッドは、何か悪いことがあればエラーメッセージを含む文字列を返し、すべてが順調であれば `None` を返します。

set_break (*filename*, *lineno*, *temporary*=0, *cond*=None, *funcname*=None)

新しいブレークポイントを設定します。引数の *lineno* 行が *filename* に存在しない場合、エラーメッセージを返します。*filename* は、`canonic()` メソッドで説明されているような、標準形である必要があります。

clear_break (*filename*, *lineno*)

filename の *lineno* 行にあるブレークポイントを削除します。もしブレークポイントが無かった場合、エラーメッセージを返します。

clear_bpbynumber (*arg*)

`Breakpoint.bpbynumber` の中で *arg* のインデックスを持つブレークポイントを削除します。*arg* が数値でないか範囲外の場合、エラーメッセージを返します。

clear_all_file_breaks (*filename*)

filename に含まれるすべてのブレークポイントを削除します。もしブレークポイントが無い場合、エラーメッセージを返します。

clear_all_breaks ()

すべてのブレークポイントを削除します。

get_break (*filename*, *lineno*)

filename の *lineno* にブレークポイントが存在するかどうかをチェックします。

get_breaks (*filename*, *lineno*)

filename の *lineno* にあるすべてのブレークポイントを返します。ブレークポイントが存在しない場合は空のリストを返します。

get_file_breaks (*filename*)

filename の中のすべてのブレークポイントを返します。ブレークポイントが存在しない場合は空のリストを返します。

get_all_breaks()

セットされているすべてのブレイクポイントを返します。

継承クラスとクライアントは以下のメソッドを呼んでスタックトレースを表現するデータ構造を取得することができます。

get_stack (*f*, *t*)

与えられたフレームおよび上位 (呼び出し側) と下位のすべてのフレームに対するレコードのリストと、上位フレームのサイズを得ます。

format_stack_entry (*frame_lineno* [, *lprefix* = ' '])

(*frame*, *lineno*) で指定されたスタックエントリに関する次のような情報を持つ文字列を返します:

- そのフレームを含むファイル名の標準形。
- 関数名、もしくは "<lambda>"。
- 入力された引数。
- 戻り値。
- (あれば) その行のコード。

以下の 2 つのメソッドは、文字列として渡された文 (*statement*) をデバッグするもので、クライアントから利用されます。

run (*cmd* [, *globals* [, *locals*]])

`exec` 文を利用して文を実行しデバッグします。 *globals* はデフォルトでは `__main__.__dict__` で、 *locals* はデフォルトでは *globals* です。

runeval (*expr* [, *globals* [, *locals*]])

`eval()` 関数を利用して式を実行しデバッグします。 *globals* と *locals* は `run()` と同じ意味です。

runctx (*cmd*, *globals*, *locals*)

後方互換性のためのメソッドです。 `run()` を使ってください。

runcall (*func*, **args*, ***kwds*)

1 つの関数呼び出しをデバッグし、その結果を返します。

最後に、このモジュールは以下の関数を提供しています:

`bdb.checkfuncname` (*b*, *frame*)

この場所でブレイクする必要があるかどうかを、ブレイクポイント *b* が設定された方法に依存する方法でチェックします。

ブレイクポイントが行番号で設定されていた場合、この関数は `b.line` が、同じく引数として与えられた *frame* の中の行に一致するかどうかをチェックします。ブレイクポイントが関数名で設定されていた場合、この関数は *frame* が指定された関数のものであるかどうかと、その関数の最初の行であるかどうかをチェックします。

`bdb.effective (file, line, frame)`

指定されたソースコード中の行に (有効な) ブレークポイントがあるかどうかを判断します。ブレークポイントと、テンポラリブレークポイントを削除して良いかどうかを示すフラグからなるタプルを返します。マッチするブレークポイントが存在しない場合は `(None, None)` を返します。

`bdb.set_trace ()`

`Bdb` クラスのインスタンスを使って、呼び出し元のフレームからデバッグを開始します。

26.2 pdb — Python デバッガ

ソースコード: [Lib/pdb.py](#)

モジュール `pdb` は Python プログラム用の対話型ソースコードデバッガを定義します。(条件付き) ブレークポイントの設定やソース行レベルでのシングルステップ実行、スタックフレームのインスペクション、ソースコードリスティングおよびあらゆるスタックフレームのコンテキストにおける任意の Python コードの評価をサポートしています。事後解析デバッグもサポートし、プログラムの制御下で呼び出すことができます。

デバッガは拡張可能です — 実際にはクラス `Pdb` として定義されています。現在これについてのドキュメントはありませんが、ソースを読めば簡単に理解できます。拡張インターフェースはモジュール `bdb` と `cmd` を使っています。

デバッガのプロンプトは `(Pdb)` です。デバッガに制御された状態でプログラムを実行するための典型的な使い方は以下のようになります:

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

他のスクリプトをデバッグするために、`pdb.py` をスクリプトとして呼び出すこともできます。例えば:

```
python -m pdb myscript.py
```

スクリプトとして `pdb` を起動すると、デバッグ中のプログラムが異常終了した時に `pdb` が自動的に事後デバッグモードに入ります。事後デバッグ後 (またはプログラムの正常終了後) には、`pdb` はプログラムを再起動します。自動再起動を行った場合、`pdb` の状態 (ブレークポイントなど) はそのまま維持されるので、たいていの場合、プログラム終了時にデバッガも終了させるよりも便利なはずで

バージョン 2.4 で追加: 事後デバッグ後の再起動機能が追加されました。

実行するプログラムをデバッガで分析する典型的な使い方は:

```
import pdb; pdb.set_trace()
```

をデバッガで分析したい場所に挿入することです。そうすることで、コードの中の以下に続く文をステップ実行できます、そして、`c` コマンドでデバッガを走らせることなく続けることもできます。

クラッシュしたプログラムを調べるための典型的な使い方は以下のようになります：

```
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./mymodule.py", line 4, in test
    test2()
  File "./mymodule.py", line 3, in test2
    print spam
NameError: spam
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print spam
(Pdb)
```

このモジュールは以下の関数を定義しています。それぞれが少しずつ違った方法でデバッガに入ります：

`pdb.run(statement[, globals[, locals]])`

デバッガに制御された状態で (文字列として与えられた) *statement* を実行します。デバッガプロンプトはあらゆるコードが実行される前に現れます。ブレークポイントを設定し、`continue` とタイプできます。あるいは、文を `step` や `next` を使って一つずつ実行することができます (これらのコマンドはすべて下で説明します)。オプションの *globals* と *locals* 引数はコードを実行する環境を指定します。デフォルトでは、モジュール `__main__` の辞書が使われます。(exec 文または `eval()` 組み込み関数の説明を参照してください。)

`pdb.runeval(expression[, globals[, locals]])`

デバッガの制御もとで (文字列として与えられる) *expression* を評価します。`runeval()` がリターンしたとき、式の値を返します。その他の点では、この関数は `run()` と同様です。

`pdb.runcall(function[, argument, ...])`

function (関数またはメソッドオブジェクト、文字列ではありません) を与えられた引数とともに呼び出します。`runcall()` から復帰するとき、関数呼び出しが返したものはなんでも返します。関数に入るとすぐにデバッガプロンプトが現れます。

`pdb.set_trace()`

スタックフレームを呼び出したところでデバッガに入ります。たとえコードが別の方法でデバッグされている最中でなくても (例えば、アサーションが失敗するとき)、これはプログラムの所定の場所でブレークポイントをハードコードするために役に立ちます。

`pdb.post_mortem([traceback])`

与えられた *traceback* オブジェクトの事後解析デバッグングに入ります。もし *traceback* が与えられなければ、その時点で取り扱っている例外のうちのひとつを使います。(デフォルト動作をさせるには、

例外を取り扱っている最中である必要があります。)

`pdb.pm()`

`sys.last_traceback` のトレースバックの事後解析デバッグングに入ります。

`run*` 関数と `set_trace()` は、`Pdb` クラスをインスタンス化して同名のメソッドを実行することのエイリアス関数です。それ以上の機能を利用したい場合は、インスタンス化を自分で行わなければなりません:

class `pdb.Pdb` (`completekey='tab', stdin=None, stdout=None, skip=None`)

`Pdb` はデバッガクラスです。

`completekey, stdin, stdout` 引数は、基底にある `cmd.Cmd` クラスに渡されます。そちらの解説を参照してください。

`skip` 引数が指定された場合、glob スタイルのモジュール名パターンの iterable (イテレート可能オブジェクト) でなければなりません。デバッガはこのパターンのどれかにマッチするモジュールに属するフレームにはステップインしません。^{*1}

`skip` を使ってトレースする呼び出しの例:

```
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

バージョン 2.7 で追加: `skip` 引数が追加されました。

run (`statement[, globals[, locals]]`)

runeval (`expression[, globals[, locals]]`)

runcall (`function[, argument, ...]`)

set_trace ()

前述のこれら関数のドキュメントを参照してください。

26.3 デバッガコマンド

デバッガは以下のコマンドを認識します。ほとんどのコマンドは一文字または二文字に省略することができます。例えば、`h(elp)` が意味するのは、ヘルプコマンドを入力するために `h` か `help` のどちらか一方を使うことができるということです (が、`he` や `hel` は使えず、また `H` や `Help`、`HELP` も使えません)。コマンドの引数は空白 (スペースまたはタブ) で区切られなければなりません。オプションの引数はコマンド構文の角括弧 (`[]`) の中に入れなければなりません。角括弧をタイプしてはいけません。コマンド構文における選択肢は垂直バー (`|`) で区切られます。

空行を入力すると入力された直前のコマンドを繰り返します。例外: 直前のコマンドが `list` コマンドならば、次の 11 行がリストされます。

デバッガが認識しないコマンドは Python 文とみなして、デバッグしているプログラムのコンテキストにおいて実行されます。先頭に感嘆符 (!) を付けることで Python 文であると明示することもできます。これはデバッグ中のプログラムを調査する強力な方法です。変数を変更したり関数を呼び出したりすることも可能です。このような文で例外が発生した場合には例外名が出力されますが、デバッガの状態は変化しません。

^{*1} フレームが属するモジュールは、そのフレームのグローバルの `__name__` によって決定されます。

複数のコマンドを `;;` で区切って一行で入力することができます。(一つだけの `;` は使われません。なぜなら、Python パーサへ渡される行内の複数のコマンドのための分離記号だからです。) コマンドを分割するために何も知的なことはしていません。たとえ引用文字列の途中であっても、入力は最初の `;;` 対で分割されます。

デバッガはエイリアスをサポートします。エイリアスはパラメータを持つことができ、調査中のコンテキストに対して人がある程度柔軟に対応できます。

ファイル `.pdbrc` はユーザのホームディレクトリか、またはカレントディレクトリにあります。それはまるでデバッガのプロンプトでタイプしたかのように読み込まれて実行されます。これは特にエイリアスのために便利です。両方のファイルが存在する場合、ホームディレクトリのものが最初に読まれ、そこに定義されているエイリアスはローカルファイルにより上書きされることがあります。

`h(elp) [command]` 引数なしでは、利用できるコマンドの一覧をプリントします。引数として `command` がある場合は、そのコマンドについてのヘルプをプリントします。 `help pdb` は完全ドキュメンテーションファイルを表示します。環境変数 `PAGER` が定義されているならば、代わりにファイルはそのコマンドへパイプされます。 `command` 引数が識別子でなければならないので、`!` コマンドについてのヘルプを得るためには `help exec` と入力しなければなりません。

`w(here)` スタックの底にある最も新しいフレームと一緒にスタックトレースをプリントします。矢印はカレントフレームを指し、それがほとんどのコマンドのコンテキストを決定します。

`d(own)` (より新しいフレームに向かって)スタックトレース内でカレントフレームを1レベル下げます。

`u(p)` (より古いフレームに向かって)スタックトレース内でカレントフレームを1レベル上げます。

`b(reak) [[filename:]lineno | function[, condition]]` `lineno` 引数がある場合は、現在のファイルのその場所にブレークポイントを設定します。 `function` 引数がある場合は、その関数の中の最初の実行可能文にブレークポイントを設定します。別のファイル(まだロードされていないかもしれないもの)のブレークポイントを指定するために、行番号はファイル名とコロンをともに先頭に付けられます。ファイルは `sys.path` にそって検索されます。各ブレークポイントは番号を割り当てられ、その番号を他のすべてのブレークポイントコマンドが参照することに注意してください。

第二引数を指定する場合、その値は式で、その評価値が真でなければブレークポイントは有効になりません。

引数なしの場合は、それぞれのブレークポイントに対して、そのブレークポイントに行き当たった回数、現在の通過カウント (ignore count) と、もしあれば関連条件を含めてすべてのブレークポイントをリストします。

`tbreak [[filename:]lineno | function[, condition]]` 一時的なブレークポイントで、最初にそこに達したときに自動的に取り除かれます。引数は `break` と同じです。

`cl(ear) [filename:lineno | bnumber [bnumber ...]]` `filename:lineno` 引数を与えると、その行にある全てのブレークポイントを解除します。スペースで区切られたブレークポイントナンバーのリストを与えると、それらのブレークポイントを解除します。引数なしの場合は、すべてのブレークポイントを解除します (が、はじめに確認します)。

`disable [bnumber [bnumber ...]]` スペースで区切られたブレークポイントナンバーのリストとして与えられるブレークポイントを無効にします。ブレークポイントを無効にすると、プログラムの実行を止め

ることができなくなりますが、ブレークポイントの解除と違いブレークポイントのリストに残ったままになり、(再び)有効にすることができます。

`enable [bpnumber [bpnumber ...]]` 指定したブレークポイントを有効にします。

`ignore bpnumber [count]` 与えられたブレークポイントナンバーに通過カウントを設定します。 `count` が省略されると、通過カウントは 0 に設定されます。通過カウントがゼロになったとき、ブレークポイントが機能する状態になります。ゼロでないときは、そのブレークポイントが無効にされず、どんな関連条件も真に評価されていて、ブレークポイントに来るたびに `count` が減らされます。

`condition bpnumber [condition]` `condition` はブレークポイントが取り上げられる前に真と評価されなければならない式です。 `condition` がない場合は、どんな既存の条件も取り除かれます。すなわち、ブレークポイントは無条件になります。

`commands [bpnumber]` ブレークポイントナンバー `bpnumber` にコマンドのリストを指定します。コマンドそのものはその後の行に続けます。 'end' だけからなる行を入力することでコマンド群の終わりを示します。例を挙げます:

```
(Pdb) commands 1
(com) print some_variable
(com) end
(Pdb)
```

ブレークポイントからコマンドを取り除くには、 `commands` のあとに `end` だけを続けます。つまり、コマンドを一つも指定しないようにします。

`bpnumber` 引数が指定されない場合、最後にセットされたブレークポイントを参照することになります。

ブレークポイントコマンドはプログラムを走らせ直すのに使えます。ただ `continue` コマンドや `step`、その他実行を再開するコマンドを使えば良いのです。

実行を再開するコマンド (現在のところ `continue`, `step`, `next`, `return`, `jump`, `quit` とそれらの省略形) によって、コマンドリストは終了するものと見なされます (コマンドにすぐ `end` が続いているかのように)。というのも実行を再開すれば (それが単純な `next` や `step` であっても) 別のブレークポイントに到達するかもしれないからです。そのブレークポイントにさらにコマンドリストがあれば、どちらのリストを実行すべきか状況が曖昧になります。

コマンドリストの中で 'silent' コマンドを使うと、ブレークポイントで停止したという通常のメッセージはプリントされません。この振る舞いは特定のメッセージを出して実行を続けるようなブレークポイントでは望ましいものでしょう。他のコマンドが何も画面出力をしなければ、そのブレークポイントに到達したというサインを見ないことになります。

バージョン 2.5 で追加.

`s(step)` 現在の行を実行し、最初に実行可能なものがあらわれたときに (呼び出された関数の中か、現在の関数の次の行で) 停止します。

`n(ext)` 現在の関数の次の行に達するか、あるいは関数が返るまで実行を継続します。(`next` と `step` の差は `step` が呼び出された関数の内部で停止するのにに対し、 `next` は呼び出された関数を (ほぼ) 全速力で実行し、現在の関数内の次の行で停止するだけです。)

unt(il) 行番号が現在行より大きくなるまで、もしくは、現在のフレームから戻るまで、実行を続けます。

バージョン 2.6 で追加。

r(eturn) 現在の関数が返るまで実行を継続します。

c(ontinue) ブレークポイントに出会うまで、実行を継続します。

j(ump) *lineno* 次に実行する行を指定します。最も底のフレーム中でのみ実行可能です。前に戻って実行したり、不要な部分をスキップして先の処理を実行する場合に使用します。

ジャンプには制限があり、例えば `for` ループの中には飛び込めませんし、`finally` 節の外にも飛び事ができません。

l(ist) [*first* [, *last*]] 現在のファイルのソースコードをリスト表示します。引数なしの場合は、現在の行の周囲を 11 行リストするか、または前のリストの続きを表示します。引数がある場合は、その行の周囲を 11 行表示します。引数が二つの場合は、与えられた範囲をリスト表示します。第二引数が第一引数より小さいときは、カウントと解釈されます。

a(rgs) 現在の関数の引数リストをプリントします。

p *expression* 現在のコンテキストにおいて *expression* を評価し、その値をプリントします。

注釈: `print` も使うことができますが、デバッガコマンドではありません — これは Python の `print` 文を実行します。

pp *expression* *pprint* モジュールを使って例外の値が整形されることを除いて p コマンドと同様です。

alias [*name* [*command*]] *name* という名前の *command* を実行するエイリアスを作成します。コマンドは引用符で囲まれてはいけません。入れ替え可能なパラメータは `%1`、`%2` などで指し示され、さらに `%*` は全パラメータに置き換えられます。コマンドが与えられなければ、*name* に対する現在のエイリアスを表示します。引数が与えられなければ、すべてのエイリアスがリストされます。

エイリアスは入れ子になってもよく、pdb プロンプトで合法的にタイプできるどんなものでも含めることができます。内部 pdb コマンドをエイリアスによって上書きすることができます。そのとき、このようなコマンドはエイリアスが取り除かれるまで隠されます。エイリアス化はコマンド行の最初の語へ再帰的に適用されます。行の他のすべての語はそのままです。

例として、二つの便利なエイリアスがあります (特に `.pdbrc` ファイルに置かれたときに):

```
#Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print "%1.",k,"=",%1.__dict__[k]
#Print instance variables in self
alias ps pi self
```

unalias *name* 指定したエイリアスを削除します。

[!] *statement* 現在のスタックフレームのコンテキストにおいて (一行の) *statement* を実行します。文の最初の語がデバッガコマンドと共通でない場合は、感嘆符を省略することができます。グローバル変数を設定するために、同じ行に `global` コマンドとともに代入コマンドの前に付けることができます。:

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

`run [args ...]` デバッグ中のプログラムを再実行します。もし引数が与えられると、`"shlex"` で分割され、結果が新しい `sys.argv` として使われます。ヒストリー、ブレークポイント、アクション、そして、デバugg オプションは引き継がれます。`"restart"` は `"run"` の別名です。

バージョン 2.6 で追加.

`q(uit)` デバugg を終了します。実行しているプログラムは中断されます。

脚注

26.4 Python プロファイラ

ソースコード: [Lib/profile.py](#) と [Lib/pstats.py](#)

26.4.1 プロファイラとは

`cProfile` と `profile` は 決定論的プロファイリング (*deterministic profiling*) を行います。プロファイル (*profile*) とは、プログラムの各部分がどれだけ頻繁に呼ばれたか、そして実行にどれだけ時間がかかったかという統計情報です。`pstats` モジュールを使ってこの統計情報をフォーマットし表示することができます。

Python 標準ライブラリは同じインターフェイスを提供するプロファイラの実装を 3 つ提供しています。

1. `cProfile` はほとんどのユーザーに推奨されるモジュールです。C 言語で書かれた拡張モジュールで、オーバーヘッドが少ないため長時間実行されるプログラムのプロファイルに適しています。Brett Rosen と Ted Czotter によって提供された `lsprof` に基づいています。

バージョン 2.5 で追加.

2. `profile` はピュア Python モジュールで、`cProfile` モジュールはこのモジュールのインタフェースを真似ています。対象プログラムに相当のオーバーヘッドが生じます。もしプロファイラに何らかの拡張をしたいのであれば、こちらのモジュールを拡張の方が簡単でしょう。このモジュールはもともと Jim Roskind により設計、実装されました。

バージョン 2.4 で変更: 組み込みの関数やメソッドで消費された時間も報告するようになりました。

3. `hotshot` は、後処理時間の長さと引き換えにプロファイル中のオーバーヘッドを小さくすることに主眼を置いた実験的な C モジュールでした。このモジュールはもう保守されておらず、将来のバージョンの Python から外されるかもしれません。

バージョン 2.5 で変更: 以前より意味のある結果が得られているはずです。かつては時間計測の中核部分に致命的なバグがありました。

`profile` と `cProfile` の両モジュールは同じインタフェースを提供しているので、ほぼ取り替え可能です。`cProfile` はずっと小さなオーバーヘッドで動きますが、まだ新しく、全てのシステムで使えるとは限りないでしょう。`cProfile` は実際には内部モジュール `_lsprof` に被せられた互換性レイヤです。`hotshot` モジュールは特別な使い道のために取っておいてあります。

注釈: 2つのプロファイラモジュールは、与えられたプログラムの実行時プロファイルを取得するために設計されており、ベンチマークのためのものではありません (ベンチマーク用途には `timeit` のほうが正確な計測結果を求められます)。これは Python コードと C で書かれたコードをベンチマークするときに特に大きく影響します。プロファイラは Python コードに対してオーバーヘッドを発生しますが、C 言語レベルの関数に対してはオーバーヘッドを生じません。なので C 言語で書かれたコードが、実際の速度と関係なく、Python で書かれたコードより速く見えるでしょう。

26.4.2 かんたんユーザマニュアル

この節は "マニュアルなんか読みたいくない人" のために書かれています。ここではきわめて簡単な概要説明とアプリケーションのプロファイリングを手取り早く行う方法だけを解説します。

1つの引数を取る関数をプロファイルしたい場合、次のようにできます:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

(お使いのシステムで `cProfile` が使えないときは代わりに `profile` を使って下さい)

上のコードは `re.compile()` を実行して、プロファイル結果を次のように表示します:

```
197 function calls (192 primitive calls) in 0.002 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   1      0.000    0.000    0.001    0.001 <string>:1(<module>)
   1      0.000    0.000    0.001    0.001 re.py:212(compile)
   1      0.000    0.000    0.001    0.001 re.py:268(_compile)
   1      0.000    0.000    0.000    0.000 sre_compile.py:172(_compile_charset)
   1      0.000    0.000    0.000    0.000 sre_compile.py:201(_optimize_charset)
   4      0.000    0.000    0.000    0.000 sre_compile.py:25(_identityfunction)
  3/1      0.000    0.000    0.000    0.000 sre_compile.py:33(_compile)
```

最初の行は 197 回の呼び出しを測定したことを示しています。その中で 192 回はプリミティブです。すなわち呼び出しが再帰によってなされてはいません。次の行は `Ordered by: standard name` は一番右の列の文字列が出力のソートに用いられたことを示します。列の見出しは以下を含みます:

`ncalls` 呼び出し回数、

tottime 与えられた関数に消費された合計時間 (sub-function の呼び出しで消費された時間は除外されます)

percall **tottime** を **ncalls** で割った値

cumtime この関数と全ての subfunction に消費された累積時間 (起動から終了まで)。この数字は再帰関数についても 正確です。

percall **cumtime** をプリミティブな呼び出し回数で割った値

filename:lineno(function) その関数のファイル名、行番号、関数名

最初の行に 2 つの数字がある場合 (たとえば 3/1) は、関数が再帰的に呼び出されたということです。2 つ目の数字はプリミティブな呼び出しの回数で、1 つ目の数字は総呼び出し回数です。関数が再帰的に呼び出されなかった場合、それらは同じ値で数字は 1 つしか表示されないことに留意してください。

run() 関数でファイル名を指定することで、プロファイル実行の終了時に出力を表示する代わりに、ファイルに保存することが出来ます。

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

pstats.Stats クラスはファイルからプロファイルの結果を読み込んで様々な書式に整えます。

ファイル **cProfile** は他のスクリプトをプロファイルするためのスクリプトとして起動することも出来ます。例えば:

```
python -m cProfile [-o output_file] [-s sort_order] myscript.py
```

-o はプロファイルの結果を標準出力の代わりにファイルに書き出します。

-s は出力を **sort_stats()** で出力をソートする値を指定します。**-o** がない場合に有効です。

pstats モジュールの **Stats** クラスにはプロファイル結果のファイルに保存されているデータを処理して出力するための様々なメソッドがあります。

```
import pstats
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

strip_dirs() メソッドによって全モジュール名から無関係なパスが取り除かれました。**sort_stats()** メソッドにより、出力される標準的な モジュール/行/名前 文字列にしたがって全項目がソートされました。**print_stats()** メソッドによって全統計が出力されました。以下のようなソート呼び出しを試すことができます:

```
p.sort_stats('name')
p.print_stats()
```

最初の行ではリストを関数名でソートしています。2 行目で情報を出力しています。さらに次の内容も試してください:

```
p.sort_stats('cumulative').print_stats(10)
```

このようにすると、関数が消費した累計時間でソートして、さらにその上位 10 件だけを表示します。どのアルゴリズムが時間を多く消費しているのか知りたいときは、この方法が役に立つはずです。

ループで多くの時間を消費している関数はどれか調べたいときは、次のようにします:

```
p.sort_stats('time').print_stats(10)
```

上記はそれぞれの関数で消費された時間でソートして、上位 10 件の関数の情報が表示されます。

次の内容も試してください:

```
p.sort_stats('file').print_stats('__init__')
```

このようにするとファイル名でソートされ、そのうちクラスの初期化メソッド (メソッド名 `__init__`) に関する統計情報だけが表示されます:

```
p.sort_stats('time', 'cum').print_stats(.5, 'init')
```

上記は時間 (time) をプライマリー、累計時間 (cumulative time) をセカンダリキーにしてソートした後さらに条件を絞って統計情報を出力します。 `.5` は上位 50% だけを選択することを意味し、さらにその中から文字列 `init` を含むものだけが表示されます。

どの関数がどの関数を呼び出しているのかを知りたいければ、次のようにします (`p` は最後に実行したときの状態でソートされています):

```
p.print_callers(.5, 'init')
```

このようにすると、関数ごとの呼び出し側関数の一覧が得られます。

さらに詳しい機能を知りたいければマニュアルを読むか、次の関数の実行結果から内容を推察してください:

```
p.print_callees()
p.add('restats')
```

スクリプトとして起動した場合、`pstats` モジュールはプロファイルのダンプを読み込み、分析するための統計ブラウザとして動きます。シンプルな行指向のインタフェース (`cmd` を使って実装) とヘルプ機能を備えています。

26.4.3 リファレンスマニュアル – `profile` と `cProfile`

`profile` および `cProfile` モジュールは以下の関数を提供します:

```
profile.run(command, filename=None, sort=-1)
```

この関数は `exec` 文に渡せる一つの引数と、オプション引数としてファイル名を指定できます。全ての場合でこのルーチンは以下を実行します:


```
exec(command, __main__.__dict__, __main__.__dict__)
```

そして実行結果からプロファイル統計情報を収集します。ファイル名が指定されていない場合は `Stats` のインスタンスが自動的に作られ、簡単なレポートが表示されます。 `sort` が指定されている場合これはこの `Stats` インスタンスに渡され、結果をどのように並び替えるかを制御します。

```
profile.runctx(command, globals, locals, filename=None)
```

この関数は `run()` に似ていますが、 `globals`、 `locals` 辞書を `command` のために与えるための追加引数を取ります。このルーチンは以下を実行します:

```
exec(command, globals, locals)
```

そしてプロファイル統計情報を `run()` 同様に収集します。

```
class profile.Profile(timer=None, timeunit=0.0, subcalls=True, builtins=True)
```

このクラスは普通、プロファイリングを `cProfile.run()` 関数が提供するもの以上に正確に制御したい場合にのみ使われます。

`timer` 引数に、コードの実行時間を計測するためのカスタムな時刻取得用関数を渡せます。これは現在時刻を表す単一の数値を返す関数でなければなりません。もしこれが整数を返す関数ならば、 `timeunit` には単位時間当たりの実際の持続時間を指定します。たとえば関数が 1000 分の 1 秒単位で計測した時間を返すとする、 `timeunit` は `.001` でしょう。

`Profile` クラスを直接使うと、プロファイルデータをファイルに書き出すことなしに結果をフォーマット出来ます:

```
import cProfile, pstats, StringIO
pr = cProfile.Profile()
pr.enable()
# ... do something ...
pr.disable()
s = StringIO.StringIO()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print s.getvalue()
```

enable()

プロファイリングデータの収集を開始します。

disable()

プロファイリングデータの収集を停止します。

create_stats()

プロファイリングデータの収集を停止し、現在のプロファイルとして結果を内部で記録します。

print_stats(sort=-1)

現在のプロファイルに基づいて `Stats` オブジェクトを作成し、結果を標準出力に出力します。

dump_stats (*filename*)

現在のプロファイルの結果を *filename* に書き出します。

run (*cmd*)

`exec()` を用いて *cmd* をプロファイルします。

runctx (*cmd, globals, locals*)

`exec()` を用いて指定されたグローバルおよびローカルな環境で *cmd* をプロファイルします。

runcall (*func, *args, **kwargs*)

`func(*args, **kwargs)` をプロファイルします。

26.4.4 Stats クラス

Stats クラスを使用してプロファイラデータを解析します。

class `pstats.Stats` (**filenames or profile, stream=sys.stdout*)

このコンストラクタは *filename* で指定した (単一または複数の) ファイルもしくは `Profile` のインスタンスから “統計情報オブジェクト” のインスタンスを生成します。出力は *stream* で指定したストリームに出力されます。

The file selected by the above constructor must have been created by the corresponding version of *profile* or *cProfile*. To be specific, there is *no* file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers, or the same profiler run on a different operating system. If several files are provided, all the statistics for identical functions will be coalesced, so that an overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing *Stats* object, the *add()* method can be used.

プロファイルデータをファイルから読み込む代わりに、`cProfile.Profile` または *profile.Profile* オブジェクトをプロファイルデータのソースとして使うことができます。

Stats には次のメソッドがあります:

strip_dirs ()

Stats クラスのこのメソッドは、ファイル名の前に付いているすべてのパス情報を取り除くためのものです。出力の幅を 80 文字以内に収めたいときに重宝します。このメソッドはオブジェクトを変更するため、取り除いたパス情報は失われます。パス情報除去の操作後、オブジェクトが保持するデータエントリは、オブジェクトの初期化、ロード直後と同じように “ランダムに” 並んでいます。 *strip_dirs()* を実行した結果、2 つの関数名が区別できない (両者が同じファイルの同じ行番号で同じ関数名となった) 場合、一つのエントリに合算されされます。

add (**filenames*)

Stats クラスのこのメソッドは、既存のプロファイリングオブジェクトに情報を追加します。引数は対応するバージョンの *profile.run()* または `cProfile.run()` によって生成されたファイルの名前でなくてはなりません。関数の名前が区別できない (ファイル名、行番号、関数名が同じ) 場合、一つの関数の統計情報として合算されます。

dump_stats (*filename*)

`Stats` オブジェクトに読み込まれたデータを、ファイル名 `filename` のファイルに保存します。ファイルが存在しない場合は新たに作成され、すでに存在する場合には上書きされます。このメソッドは `profile.Profile` クラスおよび `cProfile.Profile` クラスの同名のメソッドと等価です。

バージョン 2.3 で追加。

`sort_stats (*keys)`

このメソッドは `Stats` オブジェクトを指定した基準に従ってソートします。典型的には引数にソートのキーにしたい項目を示す文字列を指定します (例: `'time'` や `'name'` など)。

2 つ以上のキーが指定された場合、2 つ目以降のキーは、それ以前のキーで等価となったデータエントリの再ソートに使われます。たとえば `sort_stats('name', 'file')` とした場合、まずすべてのエントリが関数名でソートされた後、同じ関数名で複数のエントリがあればファイル名でソートされます。

キー名には他のキーと判別可能である限り綴りを省略して名前を指定できます。現在のバージョンで定義されているキー名は以下の通りです:

有効な引数	意味
<code>'calls'</code>	呼び出し数
<code>'cumulative'</code>	累積時間
<code>'cumtime'</code>	累積時間
<code>'file'</code>	ファイル名
<code>'filename'</code>	ファイル名
<code>'module'</code>	ファイル名
<code>'ncalls'</code>	呼び出し数
<code>'pcalls'</code>	プリミティブな呼び出し回数
<code>'line'</code>	行番号
<code>'name'</code>	関数名
<code>'nfl'</code>	関数名/ファイル名/行番号
<code>'stdname'</code>	標準名
<code>'time'</code>	内部時間
<code>'tottime'</code>	内部時間

すべての統計情報のソート結果は降順 (最も多く時間を消費したものが一番上に来る) となることに注意してください。ただし、関数名、ファイル名、行数に関しては昇順 (アルファベット順) になります。 `'nfl'` と `'stdname'` には微妙な違いがあります。標準名 (standard name) とは表示された名前によるソートで、埋め込まれた行番号のソート順が特殊です。たとえば、(ファイル名が同じで) 3、20、40 という行番号のエントリがあった場合、20、3、40 の順に表示されます。一方 `'nfl'` は行番号を数値として比較します。要するに、`sort_stats('nfl')` は `sort_stats('name', 'file', 'line')` と指定した場合と同じになります。

後方互換性のため、数値を引数に使った `-1`, `0`, `1`, `2` の形式もサポートしています。それぞれ `'stdname'`, `'calls'`, `'time'`, `'cumulative'` として処理されます。引数をこの旧スタイル

で指定した場合、最初のキー（数値キー）だけが使われ、複数のキーを指定しても 2 番目以降は無視されます。

`reverse_order()`

`Stats` クラスのこのメソッドは、オブジェクト内の情報のリストを逆順にソートします。デフォルトでは選択したキーに応じて昇順、降順が適切に選ばれることに注意してください。

`print_stats(*restrictions)`

`Stats` クラスのこのメソッドは、`profile.run()` の項で述べたプロファイルのレポートを出力します。

出力するデータの順序はオブジェクトに対し最後に行った `sort_stats()` による操作に基づきます (`add()` と `strip_dirs()` による制限にも支配されます)。

引数は (もし与えられると) リストを重要なエントリのみ制限するために使われます。初期段階でリストはプロファイルした関数の完全な情報を持っています。制限の指定は、(行数を指定する) 整数、(行のパーセンテージを指定する) 0.0 から 1.0 までの割合を指定する小数、(出力する standard name にマッチする) 正規表現のいずれかを使って行います。複数の制限が指定された場合、指定の順に適用されます。たとえば次のようになります:

```
print_stats(.1, 'foo:')
```

上記の場合まず出力するリストは全体の 10% に制限され、さらにファイル名の一部に文字列 `.*foo:` を持つ関数だけが出力されます:

```
print_stats('foo:', .1)
```

こちらの例の場合、リストはまずファイル名に `.*foo:` を持つ関数だけに制限され、その中の最初の 10% だけが出力されます。

`print_callers(*restrictions)`

`Stats` クラスのこのメソッドは、プロファイルのデータベースの中から何らかの関数呼び出しを行った関数をすべて出力します。出力の順序は `print_stats()` によって与えられるものと同じです。出力を制限する引数も同じです。各呼び出し側関数についてそれぞれ一行ずつ表示されます。フォーマットは統計を作り出したプロファイラごとに微妙に異なります:

- `profile` の場合、呼び出し側関数の後に括弧で囲まれて表示される数値はその呼び出しが何回行われたかを示しています。利便性のため、2 番目の括弧なしで表示される数値によって、関数が消費した累積時間を表しています。
- `cProfile` の場合、各呼び出し側関数の後に 3 つの数字が付きます。呼び出しが何回行われたかと、この呼び出し側関数からの呼び出しによって現在の関数内で消費された合計時間および累積時間です。

`print_callees(*restrictions)`

`Stats` クラスのこのメソッドは、指定した関数から呼び出された関数のリストを出力します。呼び出し側、呼び出される側の方向は逆ですが、引数と出力の順序に関しては `print_callers()` と同じです。

26.4.5 決定論的プロファイリングとは

決定論的プロファイリングとは、すべての関数呼び出し、関数からのリターン、例外発生をモニターし、正確なタイミングを記録することで、イベント間の時間、つまりどの時間にユーザコードが実行されているのかを計測するやり方です。もう一方の統計的プロファイリング(このモジュールでこの方法は採用していません)とは、有効なインストラクションポイントからランダムにサンプリングを行い、プログラムのどこで時間が使われているかを推定する方法です。後者の方法は、オーバーヘッドが少ないものの、プログラムのどこで多くの時間が使われているか、その相対的な示唆に留まります。

Python の場合、実行中は必ずインタプリタが動作しているため、決定論的プロファイリングを行うにあたり、計測用にコードを追加する必要はありません。Python は自動的に各イベントにフック(オプションのコールバック)を提供します。加えて Python のインタプリタという性質によって、実行時に大きなオーバーヘッドを伴う傾向がありますが、それに比べると一般的なアプリケーションでは決定論的プロファイリングで追加される処理のオーバーヘッドは少ない傾向にあります。結果的に、決定論的プロファイリングは少ないコストで Python プログラムの実行時間に関する詳細な統計を得られる方法となっているのです。

呼び出し回数はコード中のバグ発見にも使用できます(とんでもない数の呼び出しが行われている部分)。インライン拡張の対象とすべき部分を見つけるためにも使えます(呼び出し頻度の高い部分)。内部時間の統計は、注意深く最適化すべき”ホットループ”の発見にも役立ちます。累積時間の統計は、アルゴリズム選択に関連した高レベルのエラー検知に役立ちます。なお、このプロファイラは再帰的なアルゴリズム実装の累計時間を計ることが可能で、通常のループを使った実装と直接比較することもできるようになっています。

26.4.6 制限事項

一つの制限はタイミング情報の正確さに関するものです。決定論的プロファイラには正確さに関する根本的問題があります。最も明白な制限は、(一般に)”クロック”は .001 秒の精度しかないということです。これ以上の精度で計測することはできません。仮に十分な精度が得られたとしても、”誤差”が計測の平均値に影響を及ぼすことがあります。この最初の誤差を取り除いたとしても、それがまた別の誤差を引き起こす原因となります。

もう一つの問題として、イベントを検知してからプロファイラがその時刻を実際に取得するまでに”いくらかの時間がかかる”ことです。同様に、イベントハンドラが終了する時にも、時刻を取得して(そしてその値を保存して)から、ユーザコードが処理を再開するまでの間に遅延が発生します。結果的に多く呼び出される関数または多数の関数から呼び出される関数の情報にはこの種の誤差が蓄積する傾向にあります。このようにして蓄積される誤差は、典型的にはクロックの精度を下回ります(1 クロック以下)が、一方でこの時間が累計して非常に大きな値になることもあり得ます。

この問題はオーバーヘッドの小さい `cProfile` よりも `profile` においてより重要です。そのため、`profile` は誤差が確率的に(平均値で)減少するようにプラットフォームごとに補正する機能を備えています。プロファイラに補正を施すと(最小二乗の意味で)正確さが増しますが、ときには数値が負の値になってしまうこともあります(呼び出し回数が極めて少なく、確率の神があなたに意地悪をしたとき :-))。プロファイルの結果に負の値が出力されても驚かないでください。これは補正を行った場合にのみ生じることで、補正を行わない場合に比べて計測結果は実際にはより正確になっているはずだからです。

26.4.7 キャリブレーション (補正)

`profile` のプロファイラは `time` 関数呼び出しおよびその値を保存するためのオーバーヘッドを補正するために、各イベントの処理時間から定数を引きます。デフォルトでこの定数の値は 0 です。以下の手順で、プラットフォームに合った、より適切な定数が得られます ([制限事項](#) 参照)。

```
import profile
pr = profile.Profile()
for i in range(5):
    print pr.calibrate(10000)
```

`calibrate` メソッドは引数として与えられた数だけ Python の呼び出しを行います。直接呼び出す場合と、プロファイラを使って呼び出す場合の両方が実施され、それぞれの時間が計測されます。その結果、プロファイラのイベントに隠されたオーバーヘッドが計算され、その値は浮動小数として返されます。たとえば、1.8 GHz の Intel Core i5 で Mac OS X を使用、Python の `time.clock()` をタイマとして使った場合、値はおよそ $4.04\text{e-}6$ となります。

この手順で使用しているオブジェクトはほぼ一定の結果を返します。非常に早いコンピュータを使う場合、もしくはタイマの性能が貧弱な場合は、一定の結果を得るために引数に 100000 や 1000000 といった大きな値を指定する必要があるかもしれません。

一定の結果が得られたら、それを使う方法には 3 通りあります。^{*1}

```
import profile

# 1. Apply computed bias to all Profile instances created hereafter.
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

選択肢がある場合は、補正値は小さめに設定した方が良いでしょう。プロファイルの結果に負の値が表われる頻度が低く”なるはず”です。

26.4.8 カスタムな時刻取得用関数を使う

プロファイラが時刻を取得する方法を変更したいなら (たとえば、実測時間やプロセスの経過時間を使いたい場合)、時刻取得用の関数を `Profile` クラスのコンストラクタに指定することができます:

```
pr = profile.Profile(your_time_func)
```

^{*1} Python 2.2 より前のバージョンではプロファイラのソースコードに補正値として埋め込まれた定数を直接編集する必要がありました。今でも同じことは可能ですが、その方法は説明しません。なぜなら、もうソースを編集する必要がないからです。

この結果生成されるプロファイラは時刻取得に `your_time_func()` を呼び出すようになります。`profile.Profile` と `cProfile.Profile` のどちらを使っているかにより、`your_time_func` の戻り値は異なって解釈されます:

`profile.Profile` `your_time_func()` は単一の数値、あるいは (`os.times()` と同じように) その合計が累計時間を示すリストを返すようになっていなければなりません。関数が 1 つの数値、あるいは長さ 2 の数値のリストを返すようになっていれば、ディスパッチルーチンには特別な高速化バージョンが使われます。

あなたが選択した時刻取得関数のために、プロファイラのクラスを補正すべきであることを警告しておきます (キャリブレーション (補正) 参照)。ほとんどのマシンでは、プロファイル時のオーバーヘッドの小ささという意味においては、時刻取得関数が長整数値を返すのが最も良い結果を生みます。(`os.times()` は浮動小数点数のタプルを返すのでかなり悪いです。) 綺麗な手段でより良い時刻取得関数に置き換えたいと望むなら、クラスを派生して、ディスパッチメソッドを置き換えてあなたの関数を一番いいように処理するように固定化してしまってください。補正定数の調整もお忘れなく。

`cProfile.Profile` `your_time_func()` は単一の数値を返すようになっていなければなりません。単一の整数を返すのであれば、クラスコンストラクタの第二引数に単位時間当たりの実際の持続時間を渡せます。例えば `your_integer_time_func` が 1000 分の 1 秒単位で計測した時間を返すとすると、`Profile` インスタンスを以下のように構築出来ます:

```
pr = cProfile.Profile(your_integer_time_func, 0.001)
```

`cProfile.Profile` クラスはキャリブレーションができないので、自前のタイマ関数は注意を払って使う必要があります、またそれは可能な限り速くなければなりません。自前のタイマ関数で最高の結果を得るには、`_lsprof` 内部モジュールの C ソースファイルにハードコードする必要があるかもしれません。

26.5 hotshot — ハイパフォーマンス・ロギング・プロファイラ

バージョン 2.2 で追加.

このモジュールは `_hotshot` C モジュールへのより良いインターフェースを提供します。Hotshot は既存の `profile` に置き換わるものです。その大半が C で書かれているため、`profile` に比べパフォーマンス上の影響ははるかに少なく済みます。

注釈: `hotshot` は C モジュールでプロファイル中のオーバーヘッドを極力小さくすることに焦点を絞っており、その代わりに後処理時間の長さというつけを払います。通常の使用法についてはこのモジュールではなく `cProfile` を使うことを推奨します。 `hotshot` は保守されておらず、将来的には標準ライブラリから外されるかもしれません。

バージョン 2.5 で変更: 以前より意味のある結果が得られているはずですが、かつては時間計測の中核部分に致命的なバグがありました。

注釈: `hotshot` プロファイラはまだスレッド環境ではうまく動作しません。測定したいコード上でプロファイラを実行するためにスレッドを使わない版のスクリプトを使う方法が有用です。

class `hotshot.Profile` (*logfile*[, *lineevents*[, *linetimings*]])

プロファイラ・オブジェクト。引数 *logfile* はプロファイル・データのログを保存するファイル名です。引数 *lineevents* はソースコードの 1 行ごとにイベントを発生させるか、関数の呼び出し/リターンのおきだけ発生させるかを指定します。デフォルトの値は 0 (関数の呼び出し/リターンのおきだけログを残す) です。引数 *linetimings* は時間情報を記録するかどうかを指定します。デフォルトの値は 1 (時間情報を記録する) です。

26.5.1 Profile オブジェクト

Profile オブジェクトには以下のメソッドがあります:

`Profile.addinfo` (*key*, *value*)

プロファイル出力の際、任意のラベル名を追加します。

`Profile.close` ()

ログファイルを閉じ、プロファイラを終了します。

`Profile.fileno` ()

プロファイラのログファイルのファイル・デスクリプタを返します。

`Profile.run` (*cmd*)

スクリプト環境で `exec` 互換文字列のプロファイルをおこないます。 `__main__` モジュールのグローバル変数は、スクリプトのグローバル変数、ローカル変数の両方に使われます。

`Profile.runcall` (*func*, **args*, ***keywords*)

単一の呼び出し可能オブジェクトのプロファイルをおこないます。位置依存引数やキーワード引数を追加して呼び出すオブジェクトに渡すこともできます。呼び出しの結果はそのまま返されます。例外が発生したときはプロファイリングが無効になり、例外をそのまま伝えるようになっています。

`Profile.runctx` (*cmd*, *globals*, *locals*)

指定した環境で `exec` 互換文字列の評価をおこないます。文字列のコンパイルはプロファイルを開始する前におこなわれます。

`Profile.start` ()

プロファイラを開始します。

`Profile.stop` ()

プロファイラを停止します。

26.5.2 hotshot データの利用

バージョン 2.2 で追加.

このモジュールは hotshot プロファイル・データを標準の `pstats.Stats` オブジェクトにロードします。

```
hotshot.stats.load(filename)
```

`filename` から hotshot データを読み込み、`pstats.Stats` クラスのインスタンスを返します。

参考:

`profile` モジュール `profile` モジュールの `Stats` クラス

26.5.3 使用例

これは Python の ”ベンチマーク” `pystone` を使った例です。実行にはやや時間がかかり、巨大な出力ファイルを生成するので注意してください。

```
>>> import hotshot, hotshot.stats, test.pystone
>>> prof = hotshot.Profile("stones.prof")
>>> benchtime, stones = prof.runcall(test.pystone.pystones)
>>> prof.close()
>>> stats = hotshot.stats.load("stones.prof")
>>> stats.strip_dirs()
>>> stats.sort_stats('time', 'calls')
>>> stats.print_stats(20)
      850004 function calls in 10.090 CPU seconds

Ordered by: internal time, call count

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    3.295    3.295    10.090    10.090 pystone.py:79(Proc0)
150000    1.315    0.000    1.315    0.000 pystone.py:203(Proc7)
 50000    1.313    0.000    1.463    0.000 pystone.py:229(Func2)
.
.
.
```

26.6 timeit — 小さなコード断片の実行時間計測

バージョン 2.3 で追加.

ソースコード: [Lib/timeit.py](#)

このモジュールは小さい Python コードをの時間を計測するシンプルな手段を提供しています。コマンドラインインターフェイスの他 呼び出しも可能 です。このモジュールは実行時間を計測するときに共通するいくつかの罣を回避します。O'Reilly 出版の *Python Cookbook* にある、Tim Peter による ”Algorithms” 章も参照してください。

26.6.1 基本的な例

次の例は コマンドラインインターフェイス を使って 3 つの異なる式の時間を測定する方法を示しています。

```
$ python -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 40.3 usec per loop
$ python -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 3: 33.4 usec per loop
$ python -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 3: 25.2 usec per loop
```

同じ事を *Python* インターフェイス を使って実現することもできます:

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.8187260627746582
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.7288308143615723
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.5858950614929199
```

ただし、*timeit* はコマンドラインインターフェイスを使った時だけ繰り返し回数を自動で決定する事に注意してください。例節でより高度な例を説明しています。

26.6.2 Python インターフェイス

このモジュールは 3 つの有用な関数と 1 つの公開クラスを持っています:

`timeit.timeit(stmt='pass', setup='pass', timer=<default timer>, number=1000000)`

与えられた命令文、*setup* コードおよび *timer* 関数で *Timer* インスタンスを作成し、その *timeit()* メソッドを *number* 回実行します。

バージョン 2.6 で追加.

`timeit.repeat(stmt='pass', setup='pass', timer=<default timer>, repeat=3, number=1000000)`

与えられた命令文、*setup* コードおよび *timer* 関数で *Timer* インスタンスを作成し、その *repeat()* メソッドを *number* 回実行することを *repeat* 回繰り返します。

バージョン 2.6 で追加.

`timeit.default_timer()`

プラットフォーム依存の方法でデフォルトのタイマを定義します。Windows の場合、*time.clock()* はマイクロ秒の精度がありますが、*time.time()* は 1/60 秒の精度しかありません。一方 Unix の場合、*time.clock()* でも 1/100 秒の精度があり、*time.time()* はもっと正確です。いずれのプラットフォームにおいても、*default_timer()* 関数は CPU 時間ではなく通常の時間 (wall clock time) を返します。つまり、同じコンピュータ上で別のプロセスが動いている場合、測定に干渉する可能性があるということです。

```
class timeit.Timer(stmt='pass', setup='pass', timer=<timer function>)
```

小さなコード片の実行時間を計測するためのクラスです。

コンストラクターは、計測する命令文、セットアップのための追加命令およびタイマー関数をとります。2つの命令文のデフォルトは 'pass' です; タイマー関数はプラットフォーム依存です (モジュールの `doctring` を参照)。`stmt` および `setup` には、複数行の文字列リテラルを含まない限り ; や改行で区切ることで複数の命令文を指定できます。

最初の命令文の実行時間を計測するには、`timeit()` メソッドを使用します。`repeat()` メソッドは `timeit()` を複数回呼び出したい時に使用し、結果をリストで返します。

バージョン 2.6 で変更: `stmt` と `setup` 引数は、引数なしの呼び出し可能オブジェクトも受け取れるようになりました。オブジェクトを与えると、そのオブジェクトへの呼び出しがタイマー関数に埋め込まれ、そしてその関数が `timeit()` によって実行されます。この場合、関数呼び出しが増えるために、オーバーヘッドが少し増えることに注意してください。

```
timeit (number=1000000)
```

メイン文を `number` 回実行した時間を計測します。このメソッドはセットアップ文を 1 回だけ実行し、メイン文を指定回数実行するのにかった秒数を浮動小数で返します。引数はループを何回実行するかで指定で、デフォルト値は 100 万回です。メイン文、セットアップ文、タイマー関数はコンストラクターで指定されたものを使用します。

注釈: デフォルトでは、`timeit()` は計測中、一時的に [ガベージコレクション](#) を停止します。この手法の利点は個々の計測結果がより比較しやすくなることです。欠点は、ガベージコレクションが関数の性能を計測するための重要な要素である場合があることです。それに該当する場合、`setup` 文字列の最初の命令文でガベージコレクションを有効にできます。以下に例を示します:

```
timeit.Timer('for i in xrange(10): oct(i)', 'gc.enable()').timeit()
```

```
repeat (repeat=3, number=1000000)
```

`timeit()` を複数回繰り返します。

これは `timeit()` を繰り返し呼び出したい時に有用で、結果をリストにして返します。最初の引数で何回 `timeit()` を呼ぶか指定します。第 2 引数で `timeit()` の引数 `number` を指定します。

注釈: 結果のベクトルから平均値や標準偏差を計算して出力させたいと思うかもしれませんが、それはあまり意味がありません。多くの場合、最も低い値がそのマシンが与えられたコード断片を実行する場合の下限值です。結果のうち高めの値は、Python のスピードが一定しないために生じたものではなく、その他の計測精度に影響を及ぼすプロセスによるものです。したがって、結果のうち `min()` だけが見るべき値となるでしょう。この点を押さえた上で、統計的な分析よりも常識的な判断で結果を見るようにしてください。

```
print_exc (file=None)
```

計測対象コードのトレースバックを出力するためのヘルパーです。

利用例:

```
t = Timer(...)          # outside the try/except
try:
    t.timeit(...)       # or t.repeat(...)
except:
    t.print_exc()
```

これが標準のトレースバックよりも優れている点は、コンパイルしたテンプレートのソース行が表示されることです。オプションの引数 *file* にはトレースバックの出力先を指定します。デフォルトは `sys.stderr` になります。

26.6.3 コマンドラインインターフェイス

コマンドラインからプログラムとして呼び出す場合は、次の書式を使います:

```
python -m timeit [-n N] [-r N] [-s S] [-t] [-c] [-h] [statement ...]
```

以下のオプションが使用できます:

-n N, --number=N

‘statement’ を実行する回数

-r N, --repeat=N

タイマーを繰り返す回数 (デフォルトは 3)

-s S, --setup=S

最初に 1 回だけ実行する文 (デフォルトは `pass`)

-t, --time

`time.time()` を使用する (Windows を除くすべてのプラットフォームのデフォルト)

-c, --clock

`time.clock()` を使用する (Windows のデフォルト)

-v, --verbose

時間計測の結果をそのまま詳細な数値でくり返し表示する

-h, --help

簡単な使い方を表示して終了する

文は複数行指定することもできます。その場合、各行は独立した文として引数に指定されたものとして処理します。クォートと行頭のスペースを使って、インデントした文を使うことも可能です。この複数行のオプションは `-s` においても同じ形式で指定可能です。

オプション `-n` でループの回数が指定されていない場合、10 回から始めて、所要時間が 0.2 秒になるまで回数を増やすことで適切なループ回数が自動計算されるようになっています。

`default_timer()` の結果は同じコンピュータ上で動作している別のプロセスに影響を受けることがあります。そのため、正確な時間を計測する必要がある場合に最善の方法は、時間の取得を数回くり返してその中の

最短の時間を採用することです。 `-r` オプションはこれをおこなうもので、デフォルトのくり返し回数は 3 回になっています。多くの場合はデフォルトのままで充分でしょう。Unix の場合 `time.clock()` を使って CPU 時間で測定することもできます。

注釈: `pass` 文の実行による基本的なオーバーヘッドが存在することに注意してください。ここにあるコードはこの事実を隠そうとはしていませんが、注意する必要があります。基本的なオーバーヘッドは引数なしでプログラムを起動することにより計測でき、それは Python のバージョンによって異なるでしょう。Python 2.3 とそれ以前の Python の公平な比較をおこなう場合、古い Python では `-O` オプション (最適化を参照) を付けて起動して `SET_LINENO` 命令の実行時間が含まれないようにする必要があります。

26.6.4 例

最初に 1 回だけ実行されるセットアップ文を指定することが可能です:

```
$ python -m timeit -s 'text = "sample string"; char = "g"' 'char in text'
10000000 loops, best of 3: 0.0877 usec per loop
$ python -m timeit -s 'text = "sample string"; char = "g"' 'text.find(char)'
1000000 loops, best of 3: 0.342 usec per loop
```

```
>>> import timeit
>>> timeit.timeit('char in text', setup='text = "sample string"; char = "g"')
0.41440500499993504
>>> timeit.timeit('text.find(char)', setup='text = "sample string"; char = "g"')
1.7246671520006203
```

同じことは `Timer` クラスとそのメソッドを使用して行うこともできます:

```
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample string"; char = "g"')
>>> t.timeit()
0.3955516149999312
>>> t.repeat()
[0.40193588800002544, 0.3960157959998014, 0.39594301399984033]
```

以下の例は、複数行を含んだ式を計測する方法を示しています。ここでは、オブジェクトの存在する属性と存在しない属性に対してテストするために `hasattr()` と `try/except` を使用した場合のコストを比較しています:

```
$ python -m timeit 'try: ' ' str.__nonzero__ 'except AttributeError: ' ' pass'
100000 loops, best of 3: 15.7 usec per loop
$ python -m timeit 'if hasattr(str, "__nonzero__"): pass'
100000 loops, best of 3: 4.26 usec per loop

$ python -m timeit 'try: ' ' int.__nonzero__ 'except AttributeError: ' ' pass'
1000000 loops, best of 3: 1.43 usec per loop
```

(次のページに続く)

(前のページからの続き)

```
$ python -m timeit 'if hasattr(int, "__nonzero__"): pass'
100000 loops, best of 3: 2.23 usec per loop
```

```
>>> import timeit
>>> # attribute is missing
>>> s = """\
... try:
...     str.__nonzero__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.9138244460009446
>>> s = "if hasattr(str, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.5829014980008651
>>>
>>> # attribute is present
>>> s = """\
... try:
...     int.__nonzero__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

定義した関数に `timeit` モジュールがアクセスできるようにするために、`import` 文の入った `setup` パラメーターを渡すことができます:

```
def test():
    """Stupid test function"""
    L = []
    for i in range(100):
        L.append(i)

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

26.7 trace — Python ステートメント実行のトレースと追跡

ソースコード: `Lib/trace.py`

`trace` モジュールはプログラム実行のトレースを可能にし、`generate` ステートメントのカバレッジリストを注釈付きで生成して、呼び出し元/呼び出し先の関連やプログラム実行中に実行された関数のリストを出力します。これは別個のプログラム中またはコマンドラインから利用することができます。

26.7.1 コマンドラインからの使用

`trace` モジュールはコマンドラインから起動することができます。これは次のように単純です

```
python -m trace --count -C . somefile.py ...
```

これで、`somefile.py` の実行中に `import` された Python モジュールの注釈付きリストがカレントディレクトリに生成されます。

--help

使い方を表示して終了します。

--version

モジュールのバージョンを表示して終了します。

主要なオプション

`trace` を実行するときには、以下のオプションの少なくとも 1 つを指定しなければいけません。

--listfuncs オプションは、**--trace** オプション、**--count** オプションと相互排他的です。**--listfuncs** が与えられたとき、**--trace** オプション、**--count** オプションの両方とも受け付けず、他の場合も同様です。

-c, --count

プログラム完了時に、それぞれのステートメントが何回実行されたかを示す注釈付きリストのファイルを生成します。下記の **--coverdir**, **--file**, **--no-report** も参照。

-t, --trace

実行されるままに行を表示します。

-l, --listfuncs

プログラム実行の際に実行された関数を表示します。

-r, --report

--count と **--file** 引数を使った、過去のプログラム実行結果から注釈付きリストのファイルを生成します。コードを実行するわけではありません。

-T, --trackcalls

プログラム実行によって明らかになった呼び出しの関連を表示します。

修飾的オプション

-f, --file=<file>

複数回にわたるトレース実行についてカウント (count) を蓄積するファイルに名前をつけます。

`--count` オプションと一緒に使って下さい。

-C, --coverdir=<dir>

レポートファイルを保存するディレクトリを指定します。 `package.module` についてのカバレッジレポートは `dir/package/module.cover` に書き込まれます。

-m, --missing

注釈付きリストの生成時に、実行されなかった行に `>>>>>` の印を付けます。

-s, --summary

`--count` または `--report` の利用時に、処理されたファイルそれぞれの簡潔なサマリを標準出力 (stdout) に書き出します。

-R, --no-report

注釈付きリストを生成しません。これは `--count` を何度か走らせてから最後に単一の注釈付きリストを生成するような場合に便利です。

-g, --timing

各行の先頭にプログラム開始からの時間を付けます。トレース中にだけ使われます。

フィルターオプション

これらのオプションは複数回指定できます。

--ignore-module=<mod>

指定されたモジュールと (パッケージだった場合は) そのサブモジュールを無視します。引数はカンマ区切りのモジュール名リストです。

--ignore-dir=<dir>

指定されたディレクトリとサブディレクトリ中のモジュールとパッケージを全て無視します。引数は `os.pathsep` で区切られたディレクトリのリストです。

26.7.2 プログラミングインターフェース

```
class trace.Trace([count=1[, trace=1[, countfuncs=0[, countcallers=0[, ignoremods=()[, ignoredirs=()[, infile=None[, outfile=None[, timing=False]]]]]]]]])
```

文 (statement) や式 (expression) の実行をトレースするオブジェクトを作成します。全てのパラメタがオプションです。 `count` は行数を数えます。 `trace` は行実行のトレースを行います。 `countfuncs` は実行中に呼ばれた関数を列挙します。 `countcallers` は呼び出しの関連の追跡を行います。 `ignoremods` は無視するモジュールやパッケージのリストです。 `ignoredirs` は無視するパッケージやモジュールを含むディレクトリのリストです。 `infile` は保存された集計 (count) 情報を読むファイルの名前です。 `outfile` は更新された集計 (count) 情報を書き出すファイルの名前です。 `timing` は、タイムスタンプをトレース開始時点からの相対秒数で表示します。

```
run(cmd)
```

コマンドを実行して、現在のトレースパラメータに基づいてその実行から統計情報を集めます。 `cmd` は、 `exec()` に渡せるような文字列か code オブジェクトです。

runctx (*cmd*, *globals=None*, *locals=None*)

指定された *globals* と *locals* 環境下で、コマンドを実行して、現在のトレースパラメータに基づいてその実行から統計情報を集めます。*cmd* は、`exec()` に渡せるような文字列か `code` オブジェクトです。定義しない場合、*globals* と *locals* はデフォルトで空の辞書となります。

runfunc (*func*, **args*, ***kwds*)

与えられた引数の *func* を、`Trace` オブジェクトのコントロール下で現在のトレースパラメータのもとに呼び出します。

results ()

与えられた `Trace` インスタンスの `run`, `runctx`, `runfunc` の以前の呼び出しについて集計した結果を納めた `CoverageResults` オブジェクトを返します。蓄積されたトレース結果はリセットしません。

class `trace.CoverageResults`

カバレッジ結果のコンテナで、`Trace.results()` で生成されるものです。ユーザーが直接生成するものではありません。

update (*other*)

別の `CoverageResults` オブジェクトのデータを統合します。

write_results ([*show_missing=True*[, *summary=False*[, *coverdir=None*]]])

カバレッジ結果を書き出します。ヒットしなかった行も出力するには *show_missing* を指定します。モジュールごとのサマリーを出力に含めるには *summary* を指定します。*coverdir* に指定するのは結果ファイルを出力するディレクトリです。`None` の場合は各ソースファイルごとの結果がそれぞれのディレクトリに置かれます。

簡単な例でプログラミングインターフェースの使い方を見ましょう:

```
import sys
import trace

# create a Trace object, telling it what to ignore, and whether to
# do tracing or line-counting or both.
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)

# run the new command using the given tracer
tracer.run('main()')

# make a report, placing output in the current directory
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")
```


第 27 章

ソフトウェア・パッケージと配布

以下のライブラリは Python で書かれたソフトウェアを配布、インストールするためのものです。これらは [Python Package Index](#) に対して動作するように設計されていますが、ローカルのインデックスサーバーや、インデックスサーバーなしに使うこともできます。

27.1 `distutils` — Python モジュールの構築とインストール

`distutils` パッケージは、現在インストールされている Python に追加するためのモジュール構築、および実際のインストールを支援します。新規のモジュールは 100%-pure Python でも、C で書かれた拡張モジュールでも、あるいは Python と C 両方のコードが入っているモジュールからなる Python パッケージでもかまいません。

Python ユーザの大半はこのパッケージを直接使いたくはないでしょうが、代わりに Python Packaging Authority が保守しているクロスバージョンツールを使うでしょう。特に、`setuptools` は `distutils` の改良された代替品で、以下を提供しています：

- プロジェクトの依存性の宣言のサポート
- ソースのリリースの際どのファイルを含めるか指定する追加の機構 (バージョン管理システムとの統合のためのプラグインも含む)
- プロジェクトの ”エントリーポイント” を宣言する機能、アプリケーションプラグインシステムとして使うことができます
- インストール時に事前にビルドすることなく、Windows コマンドライン実行ファイルを自動的に生成する機能
- サポートしている Python の全バージョンで一貫性のある挙動

たとえスクリプト自身が `distutils` のみをインポートしていても、推奨される `pip` インストーラは `setuptools` で全 `setup.py` スクリプトを実行します。詳細は [Python Packaging User Guide](#) を参照してください。

現在のパッケージと配布システムへの理解を深めようとしている著者やユーザのために、レガシーな `distutils` に基づくユーザドキュメントと API のリファレンスは利用可能なままになっています。

- [install-index](#)
- [distutils-index](#)

27.2 ensurepip — pip インストーラのブートストラップ

バージョン 2.7.9 で追加.

`ensurepip` パッケージは `pip` インストーラを既にインストールされている Python 環境や仮想環境にブートストラップする助けになります。このブートストラップのアプローチは `pip` が独立したリリースサイクルを持ち、最新の利用可能な安定版が CPython リファレンスインタプリタのメンテナンスリリースや feature リリースにバンドルされていることを反映しています。

ほとんどの場合、Python のエンドユーザーがこのモジュールを直接呼び出す必要はないでしょう (`pip` はデフォルトでブートストラップされるからです)。しかし、もし Python のインストール時に `pip` のインストールをスキップしたり、仮想環境を構築したり、明示的に `pip` をアンインストールした場合、直接呼び出す必要があるかもしれません。

注釈: このモジュールはインターネットに アクセスしません。`pip` のブートストラップに必要な全てはこのパッケージの一部として含まれています。

参考:

`installing-index` エンドユーザーが Python パッケージをインストールする際のガイドです。

PEP 453: Python インストールの際の明示的な `pip` のブートストラッピング このモジュールのももとの論拠と仕様。

PEP 477: Python 2.7 への `ensurepip` (PEP 453) バックポート PEP453 を Python 2.7 にバックポートするための論拠と仕様。

27.2.1 コマンドラインインターフェイス

コマンドラインインターフェイスを起動するには `-m` スイッチをつけてインタプリターを使用します。

最も簡単な起動方法は:

```
python -m ensurepip
```

この起動方法は `pip` をインストールします。既にインストールされていた場合は何もしません。インストールされた `pip` のバージョンを `ensurepip` にバンドルされているもののうち、できるだけ新しいものにするためには、`--upgrade` オプションを追加して:

```
python -m ensurepip --upgrade
```

デフォルトでは、`pip` は現在の仮想環境 (もしアクティブなら) か、システムのサイトパッケージ (もしアクティブな仮想環境がなければ) にインストールされます。インストール先は 2 つの追加コマンドラインオプションで制御できます:

- `--root <dir>`: 現在のアクティブな仮想環境 (もしあれば) の `root` や現在インストールされている Python の `root` ディレクトリに入れる代わりに、与えられたディレクトリを `root` として `pip` をインストールします。
- `--user`: は、現在インストールされている Python にグローバルにインストールされる代わりに、ユーザーの `site packages` ディレクトリに `pip` をインストールします (このオプションはアクティブな仮想環境のもとでは許可されません)。

デフォルトでは `pip`, `pipX`, `pipX.Y` がインストールされます (`X.Y` は `ensurepip` を起動した Python のバージョン)。インストールされるスクリプトは 2 つの追加コマンドラインオプションで制御できます:

- `--altinstall`: alternate インストール。 `X.Y` でバージョン付けされたものだけがインストールされます。
- `--no-default-pip`: non default インストール。 `X`, `X.Y` でバージョン付けされたものだけがインストールされます。

バージョン 2.7.15 で変更: コマンドが失敗した場合は、終了ステータスは非ゼロになります。

27.2.2 モジュール API

`ensurepip` はプログラムから利用出来る 2 つの関数を公開しています:

`ensurepip.version()`

環境にブートストラップする際にインストールされることになる `pip` のバンドルバージョンを示す文字列を返します。

`ensurepip.bootstrap (root=None, upgrade=False, user=False, altinstall=False, default_pip=True, verbosity=0)`

現在の環境あるいは指示された環境へ `pip` をブートストラップします。

`root` で、インストールの `root` ディレクトリを変更します。 `root` が `None` の場合は、インストールは現在の環境でのデフォルトの場所を使います。

`upgrade` で、 `pip` のバンドルのバージョンとして、インストール済みの以前のバージョンをアップグレードするかどうかを指定します。

`user` で、グローバルなインストールではなく `user` スキームを使うかどうかを指定します。

デフォルトでは `pip`, `pipX`, `pipX.Y` がインストールされます (`X.Y` は Python のバージョン)。

`altinstall` が設定されていた場合は `pip`, `pipX` はインストール されません。

`default_pip` が `False` の場合は `pip` はインストール されません。

`altinstall` と `default_pip` の両方を指定すると、 `ValueError` を起こします。

`verbosity` でブートストラップ操作からの `sys.stdout` への出力の冗長レベルをコントロールします。

注釈: ブートストラップ処理は `sys.path`, `os.environ` の両方に対して副作用を持ちます。代わりに、サブプロセスとしてコマンドラインインターフェイスを使うことで、これら副作用を避けることが出来ます。

注釈: ブートストラップ処理は `pip` によって必要とされるモジュールを追加インストールするかもしれませんが、ほかのソフトウェアはそれら依存物がいつもデフォルトで存在していることを仮定すべきではありません (将来のバージョンの `pip` ではその依存はなくなるかもしれませんので)。

第 28 章

Python ランタイムサービス

この章では、Python インタプリタや Python 環境に深く関連する各種の機能を解説します。以下に一覧を示します:

28.1 `sys` — システムパラメータと関数

このモジュールでは、インタプリタで使用・管理している変数や、インタプリタの動作に深く関連する関数を定義しています。このモジュールは常に利用可能です。

`sys.argv`

Python スクリプトに渡されたコマンドライン引数のリスト。 `argv[0]` はスクリプトの名前となりますが、フルパス名かどうかは、OS によって異なります。コマンドライン引数に `-c` を付けて Python を起動した場合、 `argv[0]` は文字列 `'-c'` となります。スクリプト名なしで Python を起動した場合、 `argv[0]` は空文字列になります。

標準入力もしくはコマンドライン引数で指定されたファイルのリストに渡ってループするには、 `fileinput` モジュールを参照してください。

`sys.byteorder`

プラットフォームのバイト順を示します。ビッグエンディアン (最上位バイトが先頭) のプラットフォームでは `'big'`, リトルエンディアン (最下位バイトが先頭) では `'little'` となります。

バージョン 2.0 で追加.

`sys.builtin_module_names`

コンパイル時に Python インタプリタに組み込まれた、全てのモジュール名のタプル (この情報は、他の手段では取得することができません。 `modules.keys()` は、インポートされたモジュールのみのリストを返します。)

`sys.call_tracing(func, args)`

トレーシングが有効な間、 `func(*args)` を呼び出します。トレーシングの状態は保存され、後で復元されます。これは、別のコードをチェックポイントから再帰的にデバッグするために、デバッガから呼び出されることを意図しています。

`sys.copyright`

Python インタプリタの著作権を表示する文字列です。

`sys._clear_type_cache()`

内部の型キャッシュをクリアします。型キャッシュは属性とメソッドの検索を高速化するために利用されます。この関数は、参照リークをデバッグするときに不要な参照を削除するためだけに利用してください。

この関数は、内部的かつ特殊な目的にのみ利用されるべきです。

バージョン 2.6 で追加。

`sys._current_frames()`

各スレッドの識別子を関数が呼ばれた時点のそのスレッドでアクティブになっている一番上のスタックフレームに結びつける辞書を返します。モジュール `traceback` の関数を使えばそのように与えられたフレームのコールスタックを構築できます。

この関数はデッドロックをデバッグするのに非常に有効です。デッドロック状態のスレッドの協調動作を必要としませんし、そういったスレッドのコールスタックはデッドロックである限りフリーズしたままです。デッドロックにないスレッドのフレームについては、そのフレームを調べるコードを呼んだ時にはそのスレッドの現在の実行状況とは関係ないところを指し示しているかもしれません。

この関数は、内部的かつ特殊な目的にのみ利用されるべきです。

バージョン 2.5 で追加。

`sys.dllhandle`

Python DLL のハンドルを示す整数です。利用できる環境: Windows。

`sys.displayhook(value)`

`value` が `None` 以外の場合、`value` を `sys.stdout` に出力して `__builtin__` に保存します。

`sys.displayhook` は、Python の対話セッションで入力された式 (*expression*) が評価されたときに呼び出されます。対話セッションの出力をカスタマイズする場合、`sys.displayhook` に引数の数が一つの関数を指定します。

`sys.dont_write_bytecode`

この値が真の時、Python はソースモジュールをインポートする時に `.pyc` や `.pyo` ファイルを生成しません。この値は `-B` コマンドラインオプションと `PYTHONDONTWRITEBYTECODE` 環境変数の値によって起動時に `True` か `False` に設定されます。しかし、実行時にこの変数を変更して、バイトコード生成を制御することもできます。

バージョン 2.6 で追加。

`sys.excepthook(type, value, traceback)`

指定したトレースバックと例外を `sys.stderr` に出力します。

例外が発生し、その例外が捕捉されない場合、インタプリタは例外クラス・例外インスタンス・トレースバックオブジェクトを引数として `sys.excepthook` を呼び出します。対話セッション中に発生した場合はプロンプトに戻る直前に呼び出され、Python プログラムの実行中に発生した場合はプログラ

ムの終了直前に呼び出されます。このトップレベルでの例外情報出力処理をカスタマイズする場合、`sys.excepthook` に引数の数が三つの関数を指定します。

`sys.__displayhook__`

`sys.__excepthook__`

それぞれ、起動時の `displayhook` と `excepthook` の値を保存しています。この値は、`displayhook` と `excepthook` に不正なオブジェクトが指定された場合に、元の値に復旧するために使用します。

`sys.exc_info()`

この関数は、現在処理中の例外を示す三つの値のタプルを返します。この値は、現在のスレッド・現在のスタックフレームのものです。現在のスタックフレームが例外処理中でない場合、例外処理中のスタックフレームが見つかるまで次々とその呼び出し元スタックフレームを調べます。ここで、“例外処理中”とは“except 節を実行中、または実行した”フレームを指します。どのスタックフレームでも、最後に処理した例外の情報のみを参照することができます。

スタック上で例外が発生していない場合、三つの `None` のタプルを返します。例外が発生している場合、`(type, value, traceback)` を返します。`type` は、処理中の例外の型を示します (クラスオブジェクト)。`value` は、例外パラメータ (例外に関連する値または `raise` の第二引数。`type` がクラスオブジェクトの場合は常にクラスインスタンス) です。`traceback` は、トレースバックオブジェクトで、例外が発生した時点でのコールスタックをカプセル化したオブジェクトです (リファレンスマニュアル参照)。

`exc_clear()` が呼び出されると、現在のスレッドで他の例外が発生するか、又は別の例外を処理中のフレームに実行スタックが復帰するまで、`exc_info()` は三つの `None` を返します。

警告: 例外処理中に戻り値の `traceback` をローカル変数に代入すると循環参照が発生し、関数内のローカル変数やトレースバックが参照している全てのオブジェクトは解放されなくなります。特にトレースバック情報が必要ではなければ `exctype, value = sys.exc_info()[:2]` のように例外型と例外オブジェクトのみを取得するようにして下さい。もしトレースバックが必要な場合には、処理終了後に `delete` して下さい。この `delete` は、`try ... finally ...` で行くと良いでしょう。

注釈: Python 2.2 以降では、ガベージコレクションが有効であればこのような到達不能オブジェクトは自動的に削除されます。しかし、循環参照を作らないようにしたほうが効率的です。

`sys.exc_clear()`

この関数は、現在のスレッドで処理中、又は最後に発生した例外の情報を全てクリアします。この関数を呼び出すと、現在のスレッドで他の例外が発生するか、又は別の例外を処理中のフレームに実行スタックが復帰するまで、`exc_info()` は三つの `None` を返します。

この関数が必要となることは滅多にありません。ロギングやエラー処理などで最後に発生したエラーの報告を行う場合などに使用します。また、リソースを解放してオブジェクトの終了処理を起動するため

に使用することもできますが、オブジェクトが実際にされるかどうかは保障の限りではありません。

バージョン 2.3 で追加。

```
sys.exc_type
sys.exc_value
sys.exc_traceback
```

バージョン 1.5 で非推奨: `exc_info()` を使用してください

これらの変数はグローバル変数なのでスレッド毎の情報を示すことができません。この為、マルチスレッドなプログラムでは安全に参照することはできません。例外処理中でない場合、`exc_type` の値は `None` となり、`exc_value` と `exc_traceback` は未定義となります。

```
sys.exec_prefix
```

Python のプラットフォーム依存なファイルがインストールされているディレクトリ名 (サイト固有)。デフォルトでは、この値は `'/usr/local'` ですが、ビルド時に `configure` の `--exec-prefix` 引数で指定することができます。全ての設定ファイル (`pyconfig.h` など) は `exec_prefix/lib/pythonX.Y/config` に、共有ライブラリは `exec_prefix/lib/pythonX.Y/lib-dynload` にインストールされます。X.Y は Python のバージョン番号で、例えば 2.7 です。

```
sys.executable
```

Python インタプリタの実行ファイルの絶対パスを示す文字列です。このような名前が意味を持つシステムで利用可能です。Python が自身の実行ファイルの実際のパスを取得できない場合、`sys.executable` は空の文字列または `None` になります。

```
sys.exit([arg])
```

Python を終了します。`exit()` は `SystemExit` を送出するので、`try` ステートメントの `finally` 節に終了処理を記述したり、上位レベルで例外を捕捉して `exit` 処理を中断したりすることができます。

オプション引数 `arg` には、終了ステータスとして整数 (デフォルトは 0) や他の型のオブジェクトを指定することができます。整数を指定した場合、シェル等は 0 は "正常終了"、0 以外の整数を "異常終了" として扱います。多くのシステムでは、有効な終了ステータスは 0-127 で、これ以外の値を返した場合の動作は未定義です。システムによっては特定の終了コードに個別の意味を持たせている場合がありますが、このような定義は僅かしかありません。Unix プログラムでは構文エラーの場合には 2 を、それ以外のエラーならば 1 を返します。`arg` に `None` を指定した場合は、数値の 0 を指定した場合と同じです。それ以外の型のオブジェクトを指定すると、そのオブジェクトが `stderr` に出力され、終了コードとして 1 を返します。エラー発生時には `sys.exit("エラーメッセージ")` と書くと、簡単にプログラムを終了することができます。

究極には、`exit()` は例外を送出する "だけ" なので、これがメインスレッドから呼び出されたときは、プロセスを終了するだけで、例外は遮断されません。

```
sys.exitfunc
```

この値はモジュールに存在しませんが、ユーザプログラムでプログラム終了時に呼び出される終了処理関数として、引数の数が 0 の関数を設定することができます。この関数は、インタプリタ終了時に呼び出されます。`exitfunc` に指定することができる終了処理関数は一つだけですので、複数のクリーンアップ処理が必要な場合は `atexit` モジュールを使用してください。

注釈: プログラムがシグナルで kill された場合、Python 内部で致命的なエラーが発生した場合、`os._exit()` が呼び出された場合には、終了処理関数は呼び出されません。

バージョン 2.4 で非推奨: `atexit` を使ってください。

`sys.flags`

属性とシーケンスを利用して、コマンドラインフラグの状態を提供しています。属性は読み込み専用になっています。

属性	フラグ
<code>debug</code>	<code>-d</code>
<code>py3k_warning</code>	<code>-3</code>
<code>division_warning</code>	<code>-Q</code>
<code>division_new</code>	<code>-Qnew</code>
<code>inspect</code>	<code>-i</code>
<code>interactive</code>	<code>-i</code>
<code>optimize</code>	<code>-O</code> または <code>-OO</code>
<code>dont_write_bytecode</code>	<code>-B</code>
<code>no_user_site</code>	<code>-s</code>
<code>no_site</code>	<code>-S</code>
<code>ignore_environment</code>	<code>-E</code>
<code>tabcheck</code>	<code>-t</code> または <code>-tt</code>
<code>verbose</code>	<code>-v</code>
<code>unicode</code>	<code>-U</code>
<code>bytes_warning</code>	<code>-b</code>
<code>hash_randomization</code>	<code>-R</code>

バージョン 2.6 で追加。

バージョン 2.7.3 で追加: `hash_randomization` 属性が追加されました。

`sys.float_info`

属性とシーケンスを利用して、`float` 型に関する情報を提供します。精度と内部表現に関する情報を含みます。プログラミング言語 'C' の標準ヘッダファイル `float.h` に定義された様々な浮動小数点定数に対応する値の詳細については、1999 ISO/IEC C standard [C99] の 4.2.4.2.2 章を参照して下さい。

属性	float.h のマクロ	説明
<code>epsilon</code>	<code>DBL_EPSILON</code>	1 と、その次の表現可能な float 値の差
<code>dig</code>	<code>DBL_DIG</code>	浮動小数点数で正確に表示できる最大の 10 進数桁; 以下参照
<code>mant_dig</code>	<code>DBL_MANT_DIG</code>	浮動小数点精度: 浮動小数点数の主要部の桁 base-radix
<code>max</code>	<code>DBL_MAX</code>	float が表せる最大の (infinite ではない) 値
<code>max_exp</code>	<code>DBL_MAX_EXP</code>	float が $\text{radix}^{**}(\text{e}-1)$ で表現可能な、最大の整数 e
<code>max_10_exp</code>	<code>DBL_MAX_10_EXP</code>	float が $10^{**}e$ で表現可能な、最大の整数 e
<code>min</code>	<code>DBL_MIN</code>	float が表現可能な最小の正の値
<code>min_exp</code>	<code>DBL_MIN_EXP</code>	$\text{radix}^{**}(\text{e}-1)$ が正規化 float であるような最小の整数 e
<code>min_10_exp</code>	<code>DBL_MIN_10_EXP</code>	$10^{**}e$ が正規化 float であるような最小の整数 e
<code>radix</code>	<code>FLT_RADIX</code>	指数部の基数
<code>rounds</code>	<code>FLT_ROUNDS</code>	算術演算で利用される丸めモードを表す整数定数。これはインタプリタ起動時のシステムの <code>FLT_ROUNDS</code> マクロの値を示します。取りうる値とその意味については、C99 標準の 5.2.4.2.2 節を参照してください。

`sys.float_info.dig` に対してはさらに説明が必要です。もし、文字列 `s` が表す 10 進数の有効桁数が多くても `sys.float_info.dig` の時には、`s` を浮動小数点数に変換して戻すと同じ 10 進数を表す文字列に復元されます:

```
>>> import sys
>>> sys.float_info.dig
15
>>> s = '3.14159265358979'      # decimal string with 15 significant digits
>>> format(float(s), '.15g')    # convert to float and back -> same value
'3.14159265358979'
```

ただし、文字列が有効桁数 `sys.float_info.dig` より大きい場合には、常に復元されるとは限りません:

```
>>> s = '9876543211234567'     # 16 significant digits is too many!
>>> format(float(s), '.16g')    # conversion changes value
'9876543211234568'
```

バージョン 2.6 で追加.

`sys.float_repr_style`

`repr()` 関数が浮動小数点数に対してどう振る舞うかを指し示す文字列です。この文字列が値 'short' を持てば、有限の浮動小数点数 `x` に対して、`repr(x)` は `float(repr(x)) == x` を満たす短い文字列を返そうとします。これは、Python 2.7 以降での標準の振る舞いです。そうでなければ、`float_repr_style` は値 'legacy' を持ち、`repr(x)` は 2.7 以前のバージョンの Python と同じように振る舞います。

バージョン 2.7 で追加.

`sys.getcheckinterval()`

インタプリタの”チェックインターバル (check interval)” を返します; `setcheckinterval()` を参照してください。

バージョン 2.3 で追加.

`sys.getdefaultencoding()`

Unicode 実装で使用する現在のデフォルトエンコーディング名を返します。

バージョン 2.0 で追加.

`sys.getdlopenflags()`

`dlopen()` で指定されるフラグを返します。このフラグは `dl` と `DLFCN` で定義されています。

バージョン 2.2 で追加.

`sys.getfilesystemencoding()`

Unicode ファイル名をシステムのファイル名に変換する際に使用するエンコード名を返します。システムのデフォルトエンコーディングを使用する場合には `None` を返します。

- Mac OS X では、エンコーディングは `'utf-8'` となります。
- Unix では、エンコーディングは `nl_langinfo(CODESET)` が返すユーザの設定となります。
`nl_langinfo(CODESET)` が失敗すると `None` を返します。
- Windows NT+ では、Unicode をファイル名として使用できるので変換の必要はありません。
`getfilesystemencoding()` は `'mbcs'` を返しますが、これはある Unicode 文字列をバイト文字列に明示的に変換して、ファイル名として使うと同じファイルを指すようにしたい場合に、アプリケーションが使わなければならないエンコーディングです。
- Windows 9x では、エンコーディングは `'mbcs'` となります。

バージョン 2.3 で追加.

`sys.getrefcount(object)`

`object` の参照数を返します。 `object` は (一時的に) `getrefcount()` から参照されるため、参照数は予想される数よりも 1 多くなります。

`sys.getrecursionlimit()`

現在の最大再帰数を返します。最大再帰数は、Python インタプリタスタックの最大の深さです。この制限は Python プログラムが無限に再帰し、C スタックがオーバーフローしてクラッシュすることを防止するために設けられています。この値は `setrecursionlimit()` で指定することができます。

`sys.getsizeof(object[, default])`

`object` のサイズをバイト数で返します。 `object` は任意の型のオブジェクトです。すべての組み込みオブジェクトは正しい値を返します。サードパーティー製の型については実装依存になります。

オブジェクトがサイズを取得する手段を提供していない時は `default` が返されます。 `default` が指定されていない場合は `TypeError` が送出されます。

`getsizeof()` は `object` の `__sizeof__` メソッドを呼び出し、そのオブジェクトがガベージコレクタに管理されていた場合はガベージコレクタのオーバーヘッドを増やします。

バージョン 2.6 で追加.

`sys._getframe([depth])`

コールスタックからフレームオブジェクトを返します。オプション引数 *depth* を指定すると、スタックのトップから *depth* だけ下のフレームオブジェクトを取得します。 *depth* がコールスタックよりも深ければ、 `ValueError` が発生します。 *depth* のデフォルト値は 0 で、この場合はコールスタックのトップのフレームを返します。

この関数は、内部的かつ特殊な目的にのみ利用されるべきです。全ての Python 実装で存在することが保証されているわけではありません。

`sys.getprofile()`

`setprofile()` 関数で設定したプロファイラ関数を取得します。

バージョン 2.6 で追加.

`sys.gettrace()`

`settrace()` 関数で設定したトレース関数を取得します。

`gettrace()` 関数は、デバッガ、プロファイラ、カバレッジツールなどの実装に使うことのみを想定しています。この関数の振る舞いは言語定義ではなく実装プラットフォームの一部です。そのため、他の Python 実装では利用できないかもしれません。

バージョン 2.6 で追加.

`sys.getwindowsversion()`

実行中の Windows バージョンを示す、名前付きタプルを返します。名前付けされた要素は、 *major*, *minor*, *build*, *platform*, *service_pack*, *service_pack_minor*, *service_pack_major*, *suite_mask*, および *product_type* です。 *service_pack* は、文字列を含み、それ以外は整数です。この構成要素には名前でもアクセスできるので、 `sys.getwindowsversion()[0]` は `sys.getwindowsversion().major` と等価です。先行のバージョンとの互換性のため、最初の 5 要素がインデクシングで得られます。

platform は、以下の値となります:

定数	プラットフォーム
0 (VER_PLATFORM_WIN32s)	Win32s on Windows 3.1
1 (VER_PLATFORM_WIN32_WINDOWS)	Windows 95/98/ME
2 (VER_PLATFORM_WIN32_NT)	Windows NT/2000/XP/x64
3 (VER_PLATFORM_WIN32_CE)	Windows CE

product_type は、以下の値のいずれかになります。:

定数	意味
1 (VER_NT_WORKSTATION)	システムはワークステーションです。
2 (VER_NT_DOMAIN_CONTROLLER)	システムはドメインコントローラです。
3 (VER_NT_SERVER)	システムはサーバですが、ドメインコントローラではありません。

この関数は、Win32 `GetVersionEx()` 関数を呼び出します。これらのフィールドに関する詳細は `OSVERSIONINFOEX()` についてのマイクロソフトのドキュメントを参照してください。

利用出来る環境: Windows.

バージョン 2.3 で追加.

バージョン 2.7 で変更: 名前付きタプルに変更され、`service_pack_minor`, `service_pack_major`, `suite_mask`, および `product_type` が追加されました。

`sys.hexversion`

単精度整数にエンコードされたバージョン番号。この値は新バージョン (正規リリース以外であっても) ごとにならず増加します。例えば、Python 1.5.2 以降でのみ動作するプログラムでは、以下のようなチェックを行います。

```
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

`hexversion` は `hex()` で 16 進数に変換しなければ値の意味がわかりません。より読みやすいバージョン番号が必要な場合には `version_info` を使用してください。

`hexversion` は以下のレイアウトで表される 32-bit 数です:

ビット (ビッグエンディアンオーダー)	意味
1-8	PY_MAJOR_VERSION (2.1.0a3 の 2)
9-16	PY_MINOR_VERSION (2.1.0a3 の 1)
17-24	PY_MICRO_VERSION (2.1.0a3 の 0)
25-28	PY_RELEASE_LEVEL (アルファでは 0xA、ベータでは 0xB、リリース候補では 0xC、そして最終版は 0xF)
29-32	PY_RELEASE_SERIAL (the 3 in 2.1.0a3 の 0、最終リリースでは 0)

従って、2.1.0a3 は `hexversion` で `0x020100a3` です。

バージョン 1.5.2 で追加.

`sys.long_info`

Python における整数の内部表現に関する情報を保持する、構造体のシーケンスです。この属性は読み込み専用です。

属性	説明
<code>bits_per_digit</code>	各桁に保持されるビットの数です。Python の整数は、内部的に <code>2**long_info.bits_per_digit</code> を基数として保存されます。
<code>sizeof_digit</code>	桁を表すために使われる、C 型の大きさ (バイト) です

バージョン 2.7 で追加.

`sys.last_type`

`sys.last_value`

`sys.last_traceback`

通常は定義されておらず、捕捉されない例外が発生してインタプリタがエラーメッセージとトレースバックを出力した場合にのみ設定されます。この値は、対話セッション中にエラーが発生したとき、デバッグモジュールをロード (例: `import pdb; pdb.pm()`) など。詳細は [pdb — Python デバッガ](#) を参照) して発生したエラーを調査する場合に利用します。デバッガをロードすると、プログラムを再実行せずに情報を取得することができます。

変数の意味は、上の `exc_info()` の戻り値と同じです。対話セッションを実行するスレッドは常に 1 つだけなので、`exc_type` のようにスレッドに関する問題は発生しません。

`sys.maxint`

Python の整数型でサポートされる、最大の整数。この値は最低でも $2^{31}-1$ です。最大の負数は `-maxint-1` となります。正負の最大数が非対称ですが、これは 2 の補数計算を行うためです。

`sys.maxsize`

プラットフォームの `Py_ssize_t` 型がサポートしている最大の正の整数。したがって、リスト、文字列、辞書、その他コンテナ型の最大のサイズ。

`sys.maxunicode`

Unicode 文字の最大のコードポイントを示す整数。この値は、オプション設定で Unicode 文字の保存形式として USC-2 と UCS-4 のいずれを指定したかによって異なります。

`sys.meta_path`

[finder](#) オブジェクトのリストです。[finder](#) オブジェクトの `find_module()` メソッドは、インポートするモジュールを探すために呼び出されます。インポートするモジュールがパッケージに含まれる場合、親パッケージの `__path__` 属性が第 2 引数として渡されます。そのメソッドは、モジュールが見つからなかった場合は `None` を、見つかった場合は [loader](#) を返します。

`sys.meta_path` は、デフォルトの暗黙の [finder](#) や、`sys.path` よりも先に検索されます。

オリジナルの仕様については、[PEP 302](#) を参照してください。

`sys.modules`

ロード済みモジュールのモジュール名とモジュールオブジェクトの辞書。強制的にモジュールを再読み込みする場合などに使用します。この辞書からモジュールを削除するのは、`reload()` の呼び出しと等価ではありません。

`sys.path`

モジュールを検索するパスを示す文字列のリスト。`PYTHONPATH` 環境変数と、インストール時に指定したデフォルトパスで初期化されます。

起動時に初期化された後、リストの先頭 (`path[0]`) には Python インタプリタを起動したスクリプトのあるディレクトリが挿入されます。スクリプトのディレクトリがない (インタプリタが対話セッションで起動された時や、スクリプトを標準入力から読み込んだ場合など) 場合、`path[0]` は空文字列と

なり、Python はカレントディレクトリからモジュールの検索を開始します。スクリプトディレクトリは、`PYTHONPATH` で指定したディレクトリの 前 に挿入されますので注意が必要です。

必要に応じて、プログラム内で自由に変更することができます。

バージョン 2.3 で変更: Unicode 文字列が無視されなくなりました。

参考:

`site` モジュールのドキュメントで、`.pth` ファイルを使って `sys.path` を拡張する方法を解説しています。

`sys.path_hooks`

`path` を引数にとって、その `path` に対する *finder* の作成を試みる呼び出し可能オブジェクトのリスト。finder の作成に成功したら、その呼出可能オブジェクトのは finder を返します。失敗した場合は、`ImportError` を発生させます。

オリジナルの仕様は [PEP 302](#) を参照してください。

`sys.path_importer_cache`

finder オブジェクトのキャッシュとなる辞書。キーは `sys.path_hooks` に渡される `path` で、値は見つかった finder オブジェクト。 `path` が有効なファイルシステムパスであり、かつ finder が `sys.path_hooks` から見つからない場合、暗黙のデフォルト finder を利用するという意味で `None` が格納されます。 `path` が既存のパスではない場合、`imp.NullImporter` が格納されます。

オリジナルの仕様は [PEP 302](#) を参照してください。

`sys.platform`

プラットフォームを識別する文字列で、`sys.path` にプラットフォーム固有のサブディレクトリを追加する場合などに利用します。

Unix システムでは、この値は `uname -s` が返す小文字の OS 名を前半に、`uname -r` が返すバージョン名を後半に追加したものになります。例えば、`'sunos5'` や `'linux2'` といった具合です。この値は *Python* をビルドした時のものです。それ以外のシステムでは、次のような値になります。

```
if sys.platform.startswith('freebsd'):
    # FreeBSD-specific code here...
elif sys.platform.startswith('linux'):
    # Linux-specific code here...
```

バージョン 2.7.3 で変更: たくさんのコードが `sys.platform == 'linux2'` をチェックし、そして Linux 2.x と 3.x との間には本質的な違いもないことから、`sys.platform` は Linux 3.x でさえも常に `'linux2'` をセットしています。Python 3.3 とそれ以降ではこの値は常に `'linux'` ですので、`startswith` イディオムを常に使うことをお奨めします。

その他のシステムでは以下の値になります:

システム	<i>platform</i> の値
Linux (2.x と 3.x)	'linux2'
Windows	'win32'
Windows/Cygwin	'cygwin'
Mac OS X	'darwin'
OS/2	'os2'
OS/2 EMX	'os2emx'
RiscOS	'riscos'
AtheOS	'atheos'

参考:

os.name が持つ情報はおおざっぱな括りであり、*os.uname()* はシステムに依存したバージョン情報になります。

platform モジュールはシステムの詳細な識別情報をチェックする機能を提供しています。

sys.prefix

Python のプラットフォームに依存しないファイルがインストールされているディレクトリ名 (サイト固有)。デフォルトでは、この値は '/usr/local' ですが、ビルド時に **configure** の `--prefix` 引数で指定することができます。Python ライブラリの主要なコレクションは *prefix/lib/pythonX.Y* に、プラットフォーム非依存のヘッダファイル (pyconfig.h を除く全て) は *prefix/include/pythonX.Y* にインストールされます。X.Y は Python のバージョン番号で、例えば 2.7 です。

sys.ps1

sys.ps2

インタプリタの一次プロンプト、二次プロンプトを指定する文字列。対話モードで実行中のみ定義され、初期値は '>>>' と '...' です。文字列以外のオブジェクトを指定した場合、インタプリタが対話コマンドを読み込むごとにオブジェクトの *str()* を評価します。この機能は、動的に変化するプロンプトを実装する場合に利用します。

sys.py3kwarning

Python 3 warning flag の状態を格納する Bool 値。Python が -3 オプションを付けて起動された場合は True になります。(この値は定数として扱ってください。この変数を変更しても、Python 3 warning の動作には影響しません)

バージョン 2.6 で追加。

sys.setcheckinterval (*interval*)

インタプリタの "チェック間隔" を示す整数値を指定します。この値はスレッドスイッチやシグナルハンドラのチェックを行う周期を決定します。デフォルト値は 100 で、この場合 100 の Python 仮想命令を実行するとチェックを行います。この値を大きくすればスレッドを利用するプログラムのパフォーマンスが向上します。この値が 0 以下の場合、各仮想命令を実行するたびにチェックを行い、レスポンス速度が最大になりますがオーバーヘッドもまた最大となります。

sys.setdefaultencoding (*name*)

現在の Unicode 処理のデフォルトエンコーディング名を設定します。 *name* に一致するエンコーディ

ングが見つからない場合、`LookupError` が発生します。この関数は、`site` モジュールの実装が、`sitecustomize` モジュールから使用するためだけに定義されています。`site` から呼び出された後、この関数は `sys` から削除されます。

バージョン 2.0 で追加。

`sys.setdlopenflags(n)`

インタプリタが拡張モジュールをロードする時、`dlopen()` で使用するフラグを設定します。`sys.setdlopenflags(0)` とすれば、モジュールインポート時にシンボルの遅延解決を行う事ができます。シンボルを拡張モジュール間で共有する場合には、`sys.setdlopenflags(dl.RTLD_NOW | dl.RTLD_GLOBAL)` と指定します。フラグの定義名は `dl` か `DLFCN` で定義されています。`DLFCN` が存在しない場合、`h2py` スクリプトを使って `/usr/include/dlfcn.h` から生成することができます。

バージョン 2.2 で追加。

`sys.setprofile(profilefunc)`

Set the system's profile function, which allows you to implement a Python source code profiler in Python. See chapter [Python プロファイラ](#) for more information on the Python profiler. The system's profile function is called similarly to the system's trace function (see `settrace()`), but it is called with different events, for example it isn't called for each executed line of code (only on call and return, but the return event is reported even when an exception has been set). The function is thread-specific, but there is no way for the profiler to know about context switches between threads, so it does not make sense to use this in the presence of multiple threads. Also, its return value is not used, so it can simply return `None`.

Profile functions should have three arguments: *frame*, *event*, and *arg*. *frame* is the current stack frame. *event* is a string: 'call', 'return', 'c_call', 'c_return', or 'c_exception'. *arg* depends on the event type.

event には以下の意味があります。

'call' A function is called (or some other code block entered). The profile function is called; *arg* is `None`.

'return' A function (or other code block) is about to return. The profile function is called; *arg* is the value that will be returned, or `None` if the event is caused by an exception being raised.

'c_call' C 関数 (拡張関数かビルトイン関数) が呼ばれようとしている。 *arg* は C 関数オブジェクト。

'c_return' C 関数から戻った。 *arg* は C の関数オブジェクト。

'c_exception' C 関数が例外を発生させた。 *arg* は C の関数オブジェクト。

`sys.setrecursionlimit(limit)`

Python インタプリタの、スタックの最大の深さを *limit* に設定します。この制限は Python プログラムが無限に再帰し、C スタックがオーバーフローしてクラッシュすることを防止するために設けられています。

limit の最大値はプラットフォームによって異なります。深い再帰処理が必要な場合にはプラットフォーム

ムがサポートしている範囲内でより大きな値を指定することができますが、この値が大きすぎればクラッシュするので注意が必要です。

`sys.settrace(tracefunc)`

システムのトレース関数を登録します。トレース関数は Python のソースデバッガを実装するために使用できます。トレース関数はスレッド毎に設定することができるので、デバッグを行うすべてのスレッドで `settrace()` を呼び出し、トレース関数を登録してください。

Trace functions should have three arguments: *frame*, *event*, and *arg*. *frame* is the current stack frame. *event* is a string: 'call', 'line', 'return' or 'exception'. *arg* depends on the event type.

trace 関数は (*event* に 'call' を渡された状態で) 新しいローカルスコープに入るたびに呼ばれます。この場合、そのスコープで利用するローカルの trace 関数への参照か、そのスコープを trace しないのであれば None を返します。

ローカル trace 関数は自身への参照 (もしくはそのスコープの以降の trace を行う別の関数) を返すべきです。もしくは、そのスコープの trace を止めるために None を返します。

event には以下の意味があります。

'call' 関数が呼び出された (もしくは、何かのコードブロックに入った)。グローバルの trace 関数が呼ばれる。 *arg* は None が渡される。戻り値はローカルの trace 関数。

'line' インタプリタがコードの新しい行を実行しようとしている、または、ループの条件で再実行しようとしている。ローカルの trace 関数が呼ばれる。 *arg* は None 。 戻り値は新しいローカルの trace 関数。これがどのように動作するかの詳細な説明は、 `Objects/lnotab.notes.txt` を参照のこと。

'return' 関数 (あるいは別のコードブロック) から戻ろうとしている。ローカルの trace 関数が呼ばれる。 *arg* は返されようとしている値、または、このイベントが例外が送出されることによって起こったなら None 。 trace 関数の戻り値は無視される。

'exception' 例外が発生した。ローカルの trace 関数が呼ばれる。 *arg* は (`exception`, `value`, `traceback`) のタプル。戻り値は新しいローカルの trace 関数。

例外が呼び出しチェーンを辿って伝播していくことに注意してください。 'exception' イベントは各レベルで発生します。

code と frame オブジェクトについては、 `types` を参照してください。

`settrace()` 関数は、デバッガ、プロファイラ、カバレッジツール等で使うためだけのものです。この関数の挙動は言語定義よりも実装プラットフォームの分野の問題で、全ての Python 実装で利用できるとは限りません。

`sys.settsdump(on_flag)`

on_flag が真の場合、Pentium タイムスタンプカウンタを使った VM 計測結果のダンプ出力を有効にします。 *on_flag* をオフにするとダンプ出力を無効化します。この関数は Python を `--with-tsc` つきでコンパイルしたときにのみ利用できます。ダンプの内容を理解したければ、Python ソースコード中の `Python/ceval.c` を読んでください。

バージョン 2.4 で追加。

この関数は CPython の実装の詳細に密接に結びついていますが、そのため他の Python 実装では実装されていないでしょう。

```
sys.stdin
sys.stdout
sys.stderr
```

インタプリタの標準入力・標準出力・標準エラー出力に対応するファイルオブジェクト。stdin はスクリプトの読み込みを除く全ての入力処理で使用され、`input()` や `raw_input()` も stdin から読み込みます。stdout は、`print` や式 (*expression*) の評価結果、`input()`、`raw_input()` のプロンプトの出力先となります。インタプリタのプロンプトは (ほとんど) stderr に出力されます。stdout と stderr は必ずしも組み込みのファイルオブジェクトである必要はなく、`write()` メソッドを持つオブジェクトであれば使用することができます。stdout と stderr を別のオブジェクトに置き換えても、`os.popen()`、`os.system()`、`os` の `exec*`() などから起動されたプロセスが使用する標準 I/O ストリームは変更されません。

```
sys.__stdin__
sys.__stdout__
sys.__stderr__
```

それぞれ起動時の `stdin`、`stderr`、`stdout` の値を保存しています。終了処理時に利用されます。また、`sys.std*` オブジェクトが (訳注:別のファイルライクオブジェクトに) リダイレクトされている場合でも、実際の標準ストリームへの出力に利用できます。

また、標準ストリームが壊れたオブジェクトに置き換えられた場合に、動作する実際のファイルを復元するために利用することもできます。しかし、明示的に置き換え前のストリームを保存しておき、そのオブジェクトを復元する事を推奨します。

```
sys.subversion
```

3 つ組 (repo, branch, version) で Python インタプリタの Subversion 情報を表します。repo はリポジトリの名前で、'CPython'。branch は 'trunk'、'branches/name' または 'tags/name' のいずれかの形式の文字列です。version はもしインタプリタが Subversion のチェックアウトからビルドされたものならば `svnversion` の出力であり、リビジョン番号 (範囲) とローカルでの変更がある場合には最後に 'M' が付きます。ツリーがエクスポートされたもの (または `svnversion` が取得できない) で、branch がタグならば `Include/patchlevel.h` のリビジョンになります。それ以外の場合には `None` です。

バージョン 2.5 で追加。

注釈: Python is now `developed` using Git. In recent Python 2.7 bugfix releases, `subversion` therefore contains placeholder information. It is removed in Python 3.3.

```
sys.tracebacklimit
```

捕捉されない例外が発生した時、出力されるトレースバック情報の最大レベル数を指定する整数値 (デフォルト値は 1000)。0 以下の値が設定された場合、トレースバック情報は出力されず例外型と例外値のみが出力されます。

```
sys.version
```

Python インタプリタのバージョン番号の他、ビルド番号や使用コンパイラなどの情報を示す文字列です。この文字列は Python 対話型インタプリタが起動した時に表示されます。バージョン情報はここから抜き出さずに、`version_info` および `platform` が提供する関数を使って下さい。

`sys.api_version`

使用中のインタプリタの C API バージョン。Python と拡張モジュール間の不整合をデバッグする場合などに利用できます。

バージョン 2.3 で追加。

`sys.version_info`

バージョン番号を示す 5 要素タプル: `major`, `minor`, `micro`, `releaselevel`, `serial`。 `releaselevel` 以外は全て整数です。 `releaselevel` の値は、 `'alpha'`, `'beta'`, `'candidate'`, `'final'` の何れかです。Python 2.0 の `version_info` は、 `(2, 0, 0, 'final', 0)` となります。構成要素には名前でもアクセスできるので、 `sys.version_info[0]` は `sys.version_info.major` と等価、などになります。

バージョン 2.0 で追加。

バージョン 2.7 で変更: 構成する属性に名前をつけました。

`sys.warnoptions`

この値は、`warnings` フレームワーク内部のみ使用され、変更することはできません。詳細は `warnings` を参照してください。

`sys.winver`

Windows プラットフォームで、レジストリのキーとなるバージョン番号。Python DLL の文字列リソース 1000 に設定されています。通常、この値は `version` の先頭三文字となります。この値は参照専用で、別の値を設定しても Python が使用するレジストリキーを変更することはできません。利用できる環境: Windows。

出典

28.2 sysconfig — Python の構成情報にアクセスする

バージョン 2.7 で追加。

ソースコード: [Lib/sysconfig.py](#)

`sysconfig` モジュールは、インストールパスのリストや、現在のプラットフォームに関連した構成などの、Python の構成情報 (configuration information) へのアクセスを提供します。

28.2.1 構成変数

Python の配布物は、Python 自体のバイナリや、`distutils` によってコンパイルされる外部の C 拡張をビルドするために必要な、`Makefile` と `pyconfig.h` ヘッダーファイルを含んでいます。

`sysconfig` はこれらのファイルに含まれるすべての変数を辞書に格納し、`get_config_vars()` や `get_config_var()` でアクセスできるようにします。

Windows では構成変数はだいぶ少なくなります。

`sysconfig.get_config_vars(*args)`

引数がない場合、現在のプラットフォームに関するすべての構成変数の辞書を返します。

引数がある場合、各引数を構成変数辞書から検索した結果の変数のリストを返します。

各引数において、変数が見つからなかった場合は `None` が返されます。

`sysconfig.get_config_var(name)`

1 つの変数 `name` を返します。 `get_config_vars().get(name)` と同じです。

`name` が見つからない場合、`None` を返します。

Example of usage:

```
>>> import sysconfig
>>> sysconfig.get_config_var('Py_ENABLE_SHARED')
0
>>> sysconfig.get_config_var('LIBDIR')
'/usr/local/lib'
>>> sysconfig.get_config_vars('AR', 'CXX')
['ar', 'g++']
```

28.2.2 インストールパス

Python はプラットフォームとインストールオプションによって、異なるインストールスキームを利用します。このスキームは、`os.name` の値に基づいてユニークな識別子で `sysconfig` に格納されます。

`distutils` やそれに基づいたシステムによって新しいコンポーネントをインストールするときは、同じスキームに従ってファイルを正しい場所にコピーします。

Python は現在 7 つのスキームをサポートしています:

- `posix_prefix`: Linux や Mac OS X などの Posix プラットフォーム用のスキームです。これは Python やコンポーネントをインストールするときに使われるデフォルトのスキームです。
- `posix_home`: インストール時に `home` オプションが利用された場合における、Posix プラットフォーム用のスキームです。このスキームはコンポーネントが `Distutils` に特定の `home prefix` を指定してインストールされたときに利用されます。
- `posix_user`: `Distutils` に `user` オプションを指定してコンポーネントをインストールするときに使われる、Posix プラットフォーム用のスキームです。このスキームはユーザーのホームディレクトリ以下に配置されたパスを定義します。
- `nt`: Windows などの NT プラットフォーム用のスキームです。
- `nt_user`: `user` オプションが利用された場合の、NT プラットフォーム用のスキームです。

- *os2*: OS/2 プラットフォーム用のスキームです。
- *os2_home*: scheme for OS/2 platforms, when the *user* option is used.

各スキームは、ユニークな識別子を持ったいくつかのパスの集合から成っています。現在 Python は 8 つのパスを利用します:

- *stdlib*: プラットフォーム非依存の、標準 Python ライブラリファイルを格納するディレクトリ。
- *platstdlib*: プラットフォーム依存の、標準 Python ライブラリファイルを格納するディレクトリ。
- *platlib*: プラットフォーム依存の、site ごとのファイルを格納するディレクトリ。
- *purelib*: プラットフォーム非依存の、site ごとのファイルを格納するディレクトリ。
- *include*: プラットフォーム非依存のヘッダーファイルを格納するディレクトリ。
- *platinclude*: プラットフォーム依存の、ヘッダーファイルを格納するディレクトリ。
- *scripts*: スクリプトファイルのためのディレクトリ。
- *data*: データファイルのためのディレクトリ。

sysconfig はこれらのパスを決定するためのいくつかの関数を提供しています。

`sysconfig.get_scheme_names()`

現在 *sysconfig* でサポートされているすべてのスキームを格納したタプルを返します。

`sysconfig.get_path_names()`

現在 *sysconfig* でサポートされているすべてのパス名を格納したタプルを返します。

`sysconfig.get_path(name[, scheme[, vars[, expand]]])`

scheme で指定されたインストールスキームから、パス *name* に従ってインストールパスを返します。

name は `get_path_names()` が返すリストに含まれる値でなければなりません。

sysconfig はインストールパスを、パス名、プラットフォーム、展開される変数に従って格納します。例えば、*nt* スキームでの *stdlib* パスは `{base}/Lib` になります。

`get_path()` はパスを展開するのに `get_config_vars()` が返す変数を利用します。すべての変数は各プラットフォームにおいてデフォルト値を持っていて、この関数を呼び出したときにデフォルト値を取得する場合があります。

scheme が指定された場合、`get_scheme_names()` が返すリストに含まれる値でなければなりません。指定されなかった場合は、現在のプラットフォームでのデフォルトスキームが利用されます。

vars が指定された場合、`get_config_vars()` が返す辞書をアップデートする変数辞書でなければなりません。

expand が `False` に設定された場合、パスは変数を使って展開されません。

name が見つからない場合、`None` を返します。

```
sysconfig.get_paths([scheme[, vars[, expand]]])
```

インストールスキームに基づいたすべてのインストールパスを格納した辞書を返します。詳しい情報は `get_path()` を参照してください。

`scheme` が指定されない場合、現在のプラットフォームでのデフォルトスキーマが使用されます。

`vars` を指定する場合、パスを展開するために使用される辞書を更新する値の辞書でなければなりません。

`expand` に偽を指定すると、パスは展開されません。

`scheme` が実在するスキームでなかった場合、`get_paths()` は `KeyError` を発生させます。

28.2.3 その他の関数

```
sysconfig.get_python_version()
```

MAJOR.MINOR の型の Python バージョン番号文字列を返します。`sys.version[:3]` に似ています。

```
sysconfig.get_platform()
```

現在のプラットフォームを識別するための文字列を返します。

この関数は主に、プラットフォーム依存のビルドディレクトリやビルド済み配布物を判別するために利用します。通常は OS 名、バージョン、および (`os.uname()` で提供される) アーキテクチャを含みますが、実際の情報は OS 依存です。例えば、IRIX ではアーキテクチャは重要ではない (IRIX は SGI のハードでしか動きません) のに対して、Linux ではカーネルのバージョンが重要な情報ではありません。

返される値の例:

- linux-i586
- linux-alpha (?)
- solaris-2.6-sun4u
- irix-5.3
- irix64-6.2

Windows では以下のどれかを返します:

- win-amd64 (64bit Windows on AMD64, 別名 x86_64, Intel64, EM64T)
- win-ia64 (Itanium 上の 64bit Windows)
- win32 (その他すべて - 具体的には `sys.platform` が返す値)

Mac OS X では以下のどれかを返します:

- macosx-10.6-ppc
- macosx-10.4-ppc64

- macosx-10.3-i386
- macosx-10.4-fat

その他の非 POSIX プラットフォームでは、現在のところ単に `sys.platform` を返します。

`sysconfig.is_python_build()`

現在の Python インストールがソースからビルドされた場合に `True` を返します。

`sysconfig.parse_config_h(fp[, vars])`

`config.h` スタイルのファイルを解析します。

`fp` は `config.h` スタイルのファイルを指すファイルライクオブジェクトです。

`name/value` ペアを格納した辞書を返します。第二引数にオプションの辞書が渡された場合、新しい辞書ではなくその辞書を利用し、ファイルから読み込んだ値で更新します。

`sysconfig.get_config_h_filename()`

`pyconfig.h` のパスを返します。

`sysconfig.get_makefile_filename()`

`Makefile` のパスを返します。

28.3 `__builtin__` — 組み込みオブジェクト

このモジュールは Python の全ての「組み込み」識別子に直接アクセスするためのものです。例えば `__builtin__.open` は組み込み関数 `open()` の完全な名前です。ドキュメントは [組み込み関数](#) と [組み込み定数](#) を参照してください。

通常このモジュールはほとんどのアプリケーションで明示的にアクセスされることはありませんが、組み込みの値と同じ名前のオブジェクトを提供するモジュールが同時にその名前の組み込みオブジェクトも必要とするような場合には有用です。たとえば、組み込みの `open()` をラップした `open()` という関数を実装したいモジュールがあったとすると、このモジュールは次のように直接的に使われます:

```
import __builtin__

def open(path):
    f = __builtin__.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''Wrapper around a file that converts output to upper-case.'''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()

# ...
```


ほとんどのモジュールではグローバル変数の一部として `__builtins__` ('s' に注意) が利用できるようになっていました。 `__builtins__` の内容は通常 `__builtin__` モジュールそのものが、あるいは `__builtin__` モジュールの `__dict__` 属性です。これは実装の詳細部分なので、異なる Python の実装では `__builtins__` は使われていないこともあります。

28.4 `future_builtins` — Python 3 のビルトイン

バージョン 2.6 で追加.

このモジュールは、Python 2.x に存在するけれども、Python 3 では異なった動作をするために、Python 2.x のビルトイン名前空間に追加できない関数を提供します。

代わりに、Python 3 のビルトイン関数と互換性のあるコードを書きたい場合は、次のように、このモジュールからその関数を `import` してください。

```
from future_builtins import map, filter

... code using Python 3-style map and filter ...
```

Python 2 のコードを Python 3 用に変換する *2to3* ツールは、この利用方法を検出し、新しいビルトイン関数をそのまま利用します。

注釈: Python 3 の `print()` 関数は、既にビルトイン関数の中にいます。しかし、`future` 文で指定しない限り、利用することができません

```
from __future__ import print_function
```

利用できるビルトイン関数は、以下の通りです。

`future_builtins.ascii(object)`

`repr()` と同じ値を返します。Python 3 では、`repr()` は表示可能な Unicode 文字をエスケープせずに返し、`ascii()` はその文字列をバックスラッシュでエスケープします。`future_builtins.ascii()` を `repr()` の代わりに利用することで、ASCII 文字列を必要としていることを明示できます。

`future_builtins.filter(function, iterable)`

`itertools.ifilter()` と同じように動作します。

`future_builtins.hex(object)`

ビルトイン `hex()` と同じように動作しますが、`__hex__()` の代わりに、`__index__()` メソッドを利用して整数を取得し、それを 16 進数文字列に変換します。

`future_builtins.map(function, iterable, ...)`

`itertools.imap()` と同じように動作します。

注釈: Python 3 では、`map()` は `None` を引数として許容しません。

`future_builtins.oct(object)`

ビルトイン `oct()` と同じように動作しますが、`__oct__()` の代わりに、`__index__()` メソッドを利用して整数を取得し、それを 16 進数文字列に変換します。

`future_builtins.zip(*iterables)`

`itertools.izip()` と同じように動作します。

28.5 `__main__` — トップレベルのスクリプト環境

このモジュールは Python インタプリタのメインプログラムがコマンドを実行する際の環境をあらわしています。このモジュールを利用することで、通常は無名のこの環境にアクセスすることができます。実行されるコマンドは標準入力、スクリプトファイルあるいは対話環境での入力プロンプトから入力されます。この環境は Python スクリプトをメインプログラムとして実行される際によく使われる”条件付きスクリプト”の一節が実行される環境です。

```
if __name__ == "__main__":
    main()
```

28.6 `warnings` — 警告の制御

バージョン 2.1 で追加.

ソースコード: [Lib/warnings.py](#)

警告メッセージは一般に、ユーザに警告しておいた方がよいような状況下にプログラムが置かれているが、その状況は (通常は) 例外を送出したりそのプログラムを終了させるほどの正当な理由がないといった状況で発されます。例えば、プログラムが古いモジュールを使っている場合には警告を発したくなるかもしれません。

Python プログラマは、このモジュールの `warn()` 関数を使って警告を発することができます。(C 言語のプログラムは `PyErr_WarnEx()` を使います; 詳細は `exceptionhandling` を参照してください)。

警告メッセージは通常 `sys.stderr` に出力されますが、その処理方法は柔軟に変更することができます。すべての警告を無視することも、警告を例外に変更することもできます。警告の処理方法は警告カテゴリ (以下参照)、警告メッセージテキスト、そして警告を発したソースコード上の場所に基づいて変更することができます。ソースコード上の同じ場所に対して特定の警告が繰り返された場合、通常は抑制されます。

警告制御には 2 つの段階 (stage) があります: 第一に、警告が発されるたびに、メッセージを出力すべきかどうかの決定が行われます; 次に、メッセージを出力するなら、メッセージはユーザによって設定が可能なフックを使って書式化され印字されます。

警告メッセージを出力するかどうかの決定は、警告フィルタによって制御されます。警告フィルタは一致規則 (matching rule) と動作からなるシーケンスです。 `filterwarnings()` を呼び出して一致規則をフィルタに追加することができ、 `resetwarnings()` を呼び出してフィルタを標準設定の状態にリセットすることができます。

警告メッセージの印字は `showwarning()` を呼び出して行うことができ、この関数は上書きすることができます; この関数の標準の実装では、 `formatwarning()` を呼び出して警告メッセージを書式化しますが、この関数についても自作の実装を使うことができます。

参考:

`logging.captureWarnings()` を使うことで、標準ロギング基盤ですべての警告を扱うことができます。

28.6.1 警告カテゴリ

警告カテゴリを表現する組み込み例外は数多くあります。このカテゴリ化は警告をグループごとフィルタする上で便利です。現在以下の警告カテゴリクラスが定義されています:

Class	説明
<code>Warning</code>	すべての警告カテゴリクラスの基底クラスです。 <code>Exception</code> のサブクラスです。
<code>UserWarning</code>	<code>warn()</code> の標準のカテゴリです。
<code>DeprecationWarning</code>	その機能が廃止されていることを示す警告カテゴリの基底クラスです (デフォルトでは無視されます)。
<code>SyntaxWarning</code>	その文法機能があいまいであることを示す警告カテゴリの基底クラスです。
<code>RuntimeWarning</code>	そのランタイム機能があいまいであることを示す警告カテゴリの基底クラスです。
<code>FutureWarning</code>	その構文の意味付けが将来変更される予定であることを示す警告カテゴリの基底クラスです。
<code>PendingDeprecationWarning</code>	将来その機能が廃止されることを示す警告カテゴリの基底クラスです (デフォルトでは無視されます)。
<code>ImportWarning</code>	モジュールのインポート処理中に引き起こされる警告カテゴリの基底クラスです (デフォルトでは無視されます)。
<code>UnicodeWarning</code>	Unicode に関連した警告カテゴリの基底クラスです。
<code>BytesWarning</code>	Base category for warnings related to bytes and bytearray.

これらは厳密に言えば組み込み例外ですが、概念的には警告メカニズムに属しているのでここで記述されています。

標準の警告カテゴリをユーザの作成したコード上でサブクラス化することで、さらに別の警告カテゴリを定義することができます。警告カテゴリは常に `Warning` クラスのサブクラスでなければなりません。

バージョン 2.7 で変更: `DeprecationWarning` はデフォルトでは無視されます。

28.6.2 警告フィルタ

警告フィルタは、ある警告を無視すべきか、表示すべきか、あるいは (例外を送出する) エラーにするべきかを制御します。

概念的には、警告フィルタは複数のフィルタ仕様からなる順番リストを維持しています; 何らかの特定の警告が生じると、一致するものが見つかるまでリスト中の各フィルタとの照合が行われます; 一致したフィルタ仕様はその警告の処理方法を決定します。フィルタの各エントリは (*action*, *message*, *category*, *module*, *lineno*) からなるタプルです。ここで:

- *action* は以下の文字列のうちの一つです:

値	処理方法
"error"	一致した警告を例外に変えます
"ignore"	一致した警告を出力しません
"always"	一致した警告を常に出力します
"default"	一致した警告のうち、警告の原因になったソースコード上の場所ごとに、最初の警告のみ出力します
"module"	一致した警告のうち、警告の原因になったモジュールごとに、最初の警告のみ出力します
"once"	一致した警告のうち、警告の原因になった場所にかかわらず最初の警告のみ出力します

- *message* は正規表現を含む文字列で、警告メッセージの先頭はこのパターンに一致しなければなりません。正規表現は常に大小文字の区別をしないようにコンパイルされます。
- *category* はクラス (*Warning* のサブクラス) です。警告クラスはこのクラスのサブクラスに一致しなければなりません。
- *module* は正規表現を含む文字列で、モジュール名はこのパターンに一致しなければなりません。正規表現は常に大小文字の区別をしないようにコンパイルされます。
- *lineno* は整数で、警告が発生した場所の行番号に一致しなければなりません。0 の場合はすべての行と一致します。

Warning クラスは組み込みの *Exception* クラスから派生しているので、警告をエラーに変換するには単に *category* (*message*) を *raise* します。

警告フィルタは Python インタプリタのコマンドラインに渡される *-W* オプションで初期化されます。インタプリタは *-W* オプションに渡されるすべての引数を *sys.warnoptions* に変換せずに保存します; *warnings* モジュールは最初に *import* された際にこれらの引数を解釈します (無効なオプションは *sys.stderr* にメッセージを出力した上で無視されます)。

デフォルトの警告フィルタ

デフォルトで、Python はいくつかの警告フィルタをインストールします。これはコマンドラインオプション *-W* か *filterwarnings()* 関数でオーバーライドできます。

- `DeprecationWarning`, `PendingDeprecationWarning`, `ImportWarning` は無視されます。
- `-b` オプションが 1 度か 2 度指定されない限り、`BytesWarning` は無視されます。 `-b` の時は警告が表示され、`-bb` の時は例外になります。

28.6.3 一時的に警告を抑制する

廃止予定の関数など、警告を発生させることが分かっているコードを利用する場合に、そのような警告を表示したくなければ、`catch_warnings` コンテキストマネージャーを使って警告を抑制することができます：

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
```

このサンプルのコンテキストマネージャーの中では、すべての警告が無視されています。これで、他の廃止予定のコードを含まない(つもりの)部分まで警告を抑止せずに、廃止予定だと分かっているコードだけ警告を表示させないようにすることができます。注意: これが保証できるのはシングルスレッドのアプリケーションだけです。2 つ以上のスレッドが同時に `catch_warnings` コンテキストマネージャーを使用した場合、動作は未定義です。

28.6.4 警告のテスト

コードが警告を発生させることをテストするには、`catch_warnings` コンテキストマネージャーを利用します。このクラスを使うと、一時的に警告フィルタを操作してテストに利用できます。例えば、次のコードでは、発生したすべての警告を取得してチェックしています：

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
    # Cause all warnings to always be triggered.
    warnings.simplefilter("always")
    # Trigger a warning.
    fxn()
    # Verify some things
    assert len(w) == 1
    assert isinstance(w[-1].category, DeprecationWarning)
    assert "deprecated" in str(w[-1].message)
```

`always` の代わりに `error` を利用することで、すべての警告で例外を発生させることができます。1 つ気をつけたいといけないのは、一度 `once/default` ルールによって発生した警告は、フィルタに何をセットして

いるかにかかわらず、警告レジストリをクリアしない限りは 2 度と発生しません。

コンテキストマネージャーが終了したら、警告フィルタはコンテキストマネージャーに入る前のものに戻されます。これは、テスト中に予期しない方法で警告フィルタが変更され、テスト結果が中途半端になる事を予防します。このモジュールの `showwarning()` 関数も元の値に戻されます。注意: これが保証できるのはシングルスレッドのアプリケーションだけです。2 つ以上のスレッドが同時に `catch_warnings` コンテキストマネージャを使用した場合、動作は未定義です。

同じ種類の警告を発生させる複数の操作をテストする場合、各操作が新しい警告を発生させている事を確認するのは大切な事です (例えば、警告を例外として発生させて各操作が例外を発生させることを確認したり、警告リストの長さが各操作で増加していることを確認したり、警告リストを各操作の前に毎回クリアする事ができます)。

28.6.5 コードを新しいバージョンの Python のために更新する

開発者にしか興味の無い警告はデフォルトでは無視されるようになりました。なので、通常、コードをテストするときには無視されている例外を表示するようにする必要があります。これは `-Wd <-W>` コマンドライン引数 (`-W default` の省略形) をインタプリタに指定することで可能です。これはデフォルトでは無視されているものも含めた全ての警告に対して、デフォルトの動作を有効にします。この動作を変更するためには、`-W` への引数を変更します。例えば、`-W error` のようにします。何が可能かについての詳細は、`-W` フラグを参照してください。

プログラムから `-Wd` と同じ事をするには、次のようにします:

```
warnings.simplefilter('default')
```

このコードは可能な限り早く実行してください。そうすることで、どの警告を発生させるかの登録が、将来の警告がどう扱われるかについて思わぬ影響を与えることを避けることができます。

いくつかの警告がデフォルトで無視されているのは、開発者向けの警告をユーザーに見せることを避けるためです。ユーザーがどのインタプリタを使ってコードを実行するのかを必ずしも制御できるわけではないので、自分のコードのリリースサイクルの間に新しいバージョンの Python がリリースされるかもしれません。新しいインタプリタは、古いインタプリタが出さなかった新しい警告を発生させるかもしれません。例えば、利用しているモジュールに対して `DeprecationWarning` が発生されることがあります。開発者としては、廃止予定のモジュールを利用していることに対する通知は有益ですが、一般ユーザーにとってはこの情報はノイズでしか無く、何の役にも立ちません。

28.6.6 利用可能な関数

`warnings.warn(message[, category[, stacklevel]])`

警告を発するか、無視するか、あるいは例外を送出します。 `category` 引数が与えられた場合、警告カテゴリクラスでなければなりません (上記を参照してください); 標準の値は `UserWarning` です。 `message` を `Warning` インスタンスで代用することもできますが、この場合 `category` は無視され、 `message.__class__` が使われ、メッセージ文は `str(message)` になります。発された例外が前述

した警告フィルタによってエラーに変更された場合、この関数は例外を送出します。引数 *stacklevel* は Python でラッパー関数を書く際に利用することができます。例えば:

```
def deprecation(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

こうすることで、警告が参照するソースコード部分を、`deprecation()` 自身ではなく `deprecation()` を呼び出した側にできます (というのも、前者の場合は警告メッセージの目的を台無しにしてしまうからです)。

`warnings.warn_explicit(message, category, filename, lineno[, module[, registry[, module_globals]]])`

`warn()` の機能に対する低レベルのインタフェースで、メッセージ、警告カテゴリ、ファイル名および行番号、そしてオプションのモジュール名およびレジストリ情報 (モジュールの `__warningregistry__` 辞書) を明示的に渡します。モジュール名は標準で `.py` が取り去られたファイル名になります; レジストリが渡されなかった場合、警告が抑制されることはありません。 *message* が文字列のとき、 *category* は `Warning` のサブクラスでなければなりません。また *message* は `Warning` のインスタンスであってもよく、この場合 *category* は無視されます。

module_globals は、もし与えられるならば、警告が発せられるコードが使っているグローバル名前空間でなければなりません (この引数は zipfile やその他の非ファイルシステムのインポート元の中にあるモジュールのソースを表示することをサポートするためのものです)。

バージョン 2.5 で変更: *module_globals* パラメータが追加されました。

`warnings.warnpy3k(message[, category[, stacklevel]])`

Python 3.x で廃止予定についての警告を発生させます。Python が -3 オプション付きで実行されているときのみ警告が表示されます。 `warn()` と同じく、 *message* は文字列で、 *category* は `Warning` のサブクラスである必要があります。 `warnpy3k()` は `DeprecationWarning` をデフォルトの warning クラスとして利用しています。

バージョン 2.6 で追加。

`warnings.showwarning(message, category, filename, lineno[, file[, line]])`

警告をファイルに書き込みます。標準の実装では、 `formatwarning(message, category, filename, lineno, line)` を呼び出し、返された文字列を *file* に書き込みます。 *file* は標準では `sys.stderr` です。この関数は `warnings.showwarning` に別の実装を代入して置き換えることができます。 *line* は警告メッセージに含めるソースコードの 1 行です。 *line* が与えられない場合、 `showwarning()` は *filename* と *lineno* から行を取得することを試みます。

バージョン 2.7 で変更: *line* 引数のサポートが必須になりました。

`warnings.formatwarning(message, category, filename, lineno[, line])`

警告を通常の方法で書式化します。返される文字列内には改行が埋め込まれている可能性があり、かつ文字列は改行で終端されています。 *line* は警告メッセージに含まれるソースコードの 1 行です。 *line* が渡されない場合、 `formatwarning()` は *filename* と *lineno* から行の取得を試みます。

バージョン 2.6 で変更: *line* 引数が追加されました。


```
warnings.filterwarnings(action[, message[, category[, module[, lineno[, append]]]]])
```

警告フィルタ仕様 のリストにエントリを一つ挿入します。標準ではエントリは先頭に挿入されます; *append* が真ならば、末尾に挿入されます。この関数は引数の型をチェックし、*message* および *module* の正規表現をコンパイルしてから、これらをタプルにして警告フィルタのリストに挿入します。二つのエントリが特定の警告に合致した場合、リストの先頭に近い方のエントリが後方にあるエントリに優先します。引数が省略されると、標準ではすべてにマッチする値に設定されます。

```
warnings.simplefilter(action[, category[, lineno[, append]]])
```

単純なエントリを 警告フィルタ仕様 のリストに挿入します。引数の意味は *filterwarnings()* と同じですが、この関数により挿入されるフィルタはカテゴリと行番号が一致していればすべてのモジュールのすべてのメッセージに合致しますので、正規表現は必要ありません。

```
warnings.resetwarnings()
```

警告フィルタをリセットします。これにより、*-W* コマンドラインオプションによるもの *simplefilter()* 呼び出しによるものを含め、*filterwarnings()* の呼び出しによる影響はすべて無効化されます。

28.6.7 利用可能なコンテキストマネージャー

```
class warnings.catch_warnings([*, record=False, module=None])
```

警告フィルタと *showwarning()* 関数をコピーし、終了時に復元するコンテキストマネージャーです。 *record* 引数が *False* (デフォルト値) だった場合、コンテキスト開始時には *None* を返します。もし *record* が *True* だった場合、リストを返します。このリストにはカスタムの *showwarning()* 関数 (この関数は同時に *sys.stdout* への出力を抑制します) によってオブジェクトが継続的に追加されます。リストの中の各オブジェクトは、*showwarning()* 関数の引数と同じ名前の属性を持っています。

module 引数は、保護したいフィルタを持つモジュールを取ります。 *warnings* を *import* して得られるモジュールの代わりに利用されます。この引数は、主に *warnings* モジュール自体をテストする目的で追加されました。

注釈: *catch_warnings* マネージャーは、モジュールの *showwarning()* 関数と内部のフィルタ仕様のリストを置き換え、その後復元することによって動作しています。これは、コンテキストマネージャーがグローバルな状態を変更していることを意味していて、したがってスレッドセーフではありません。

注釈: Python 3 では、*catch_warnings* コンストラクタの引数はキーワード引数のみにになりました。

バージョン 2.6 で追加.

28.7 contextlib — with 文コンテキスト用ユーティリティ

バージョン 2.5 で追加.

ソースコード: [Lib/contextlib.py](#)

このモジュールは with 文に関わる一般的なタスクのためのユーティリティを提供します。詳しい情報は、[コンテキストマネージャ型](#) と [context-managers](#) を参照してください。

提供されている関数:

`contextlib.contextmanager` (*func*)

この関数は with 文コンテキストマネージャのファクトリ関数を定義するために利用できる [デコレータ](#) です。新しいクラスや `__enter__()` と `__exit__()` メソッドを別々に定義しなくても、ファクトリ関数を定義することができます。

While many objects natively support use in with statements, sometimes a resource needs to be managed that isn't a context manager in its own right, and doesn't implement a `close()` method for use with `contextlib.closing`

An abstract example would be the following to ensure correct resource management:

```
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwargs):
    # Code to acquire resource, e.g.:
    resource = acquire_resource(*args, **kwargs)
    try:
        yield resource
    finally:
        # Code to release resource, e.g.:
        release_resource(resource)

>>> with managed_resource(timeout=3600) as resource:
...     # Resource is released at the end of this block,
...     # even if code in the block raises an exception
```

デコレート対象の関数は呼び出されたときに [ジェネレータ](#)-イテレータを返す必要があります。このイテレータは必ず値を 1 つ yield しなければなりません。with 文の `as` 節が存在するなら、その値は `as` 節のターゲットへ束縛されることになります。

ジェネレータが `yield` を実行した箇所で with 文のネストされたブロックが実行されます。ブロックから抜けた後でジェネレータは再開されます。ブロック内で処理されない例外が発生した場合は、ジェネレータ内部の `yield` を実行した箇所で例外が再送出されます。なので、(もしあれば) エラーを捕捉したり、クリーンアップ処理を確実に実行したりするために、`try...except...finally` 構文を使用できます。例外を捕捉する目的が、(完全に例外を抑制してしまうのではなく) 単に例外のログをとるため、もしくはあるアクションを実行するためなら、ジェネレータはその例外を再送出しなければなりません。

例外を再送出しない場合、ジェネレータのコンテキストマネージャは `with` 文に対して例外が処理されたことを示し、`with` 文の直後の文から実行を再開します。

`contextlib.nested(mgr1[, mgr2[, ...]])`

複数のコンテキストマネージャを一つのネストされたコンテキストマネージャへ結合します。

この関数は、`with` 文にマルチマネージャ形式ができたために非推奨になりました。

この関数の `with` 文のマルチマネージャ形式に対する唯一の利点は、引数のアンパックによって、次の例のように可変数個のコンテキストマネージャを扱えることです。

```
from contextlib import nested

with nested(*managers):
    do_something()
```

ネストされたコンテキストマネージャのうちのいずれかの `__exit__()` メソッドが例外を抑制すべきと判断した場合、外側にある残りのすべてのコンテキストマネージャに例外情報が渡されないということに注意してください。同様に、ネストされたコンテキストマネージャのうちのいずれかの `__exit__()` メソッドが例外を送出したならば、それ以前の例外状態は失われ、新しい例外が外側にある残りのすべてのコンテキストマネージャの `__exit__()` メソッドに渡されます。一般的に `__exit__()` メソッドが例外を送出することは避けるべきであり、特に渡された例外を再送出すべきではありません。

この関数が非推奨になった理由に、2つの大きな問題があります。1つ目は、全てのコンテキストマネージャが関数が呼び出される前に構築されることです。内側のコンテキストマネージャの `__new__()` と `__init__()` メソッドは外側のコンテキストマネージャの内側に入っていません。つまり、例えば `nested()` を2つのファイルを開くために利用した場合、2つめのファイルを開くのに失敗すると1つめのファイルが正しく `close` されないというプログラムエラーになります。

2つ目の問題は、内側のコンテキストマネージャの1つの `__enter__()` メソッドが例外を発生させたときに、外側のコンテキストマネージャの `__exit__()` メソッドがその例外を捕まえて抑制させてしまうことで、この場合に `with` の `body` 部分の実行がスキップされるのではなく `RuntimeError` が発生する恐れがあります。

可変数個のコンテキストマネージャのネストをサポートしなければならない場合、`warnings` モジュールを利用してこの関数が発生させる `DeprecationWarning` を抑制するか、この関数を参考にしてアプリケーション独自の実装をすることができます。

バージョン 2.7 で非推奨: `with` 文がこの関数の機能を (この関数と違って奇妙なエラーを発生させることなく) 直接サポートしました。

`contextlib.closing(thing)`

ブロックの完了時に `thing` を `close` するコンテキストマネージャを返します。これは基本的に以下と等価です:

```
from contextlib import contextmanager

@contextmanager
def closing(thing):
```

(次のページに続く)

(前のページからの続き)

```
try:
    yield thing
finally:
    thing.close()
```

そして、明示的に `page` を `close` する必要なしに、次のように書くことができます:

```
from contextlib import closing
import urllib

with closing(urllib.urlopen('http://www.python.org')) as page:
    for line in page:
        print line
```

`page` を明示的に `close` する必要は無く、エラーが発生した場合でも、`with` ブロックを出るときに `page.close()` が呼ばれます。

参考:

PEP 343 - "with" ステートメント 仕様、背景、および、Python `with` 文の例。

28.8 abc — 抽象基底クラス

バージョン 2.6 で追加.

ソースコード: [Lib/abc.py](#)

このモジュールは Python に **PEP 3119** で概要が示された **抽象基底クラス** (ABC) を定義する基盤を提供します。なぜこれが Python に付け加えられたかについてはその PEP を参照してください。(ABC に基づいた数の型階層を扱った **PEP 3141** と `numbers` モジュールも参照してください。)

The `collections` module has some concrete classes that derive from ABCs; these can, of course, be further derived. In addition, the `collections` module has some ABCs that can be used to test whether a class or instance provides a particular interface, for example, if it is hashable or if it is a mapping.

このモジュールは以下のクラスを提供します:

```
class abc.ABCMeta
```

抽象基底クラス (ABC) を定義するためのメタクラス。

ABC を作るときにこのメタクラスを使います。ABC は直接的にサブクラス化することができ、ミックスイン (mix-in) クラスのように振る舞います。また、無関係な具象クラス (組み込み型でも構いません) と無関係な ABC を "仮想的サブクラス" として登録できます – これらとその子孫は組み込み関数 `issubclass()` によって登録した ABC のサブクラスと判定されますが、登録した ABC は MRO (Method Resolution Order, メソッド解決順) には現れませんし、この ABC のメソッド実装が (`super()` を通してだけでなく) 呼び出し可能になるわけでもありません。^{*1}

^{*1} C++ プログラマは Python の仮想的基底クラス概念は C++ のものと同じではないということを銘記すべきです。

メタクラス `ABCMeta` を使って作られたクラスには以下のメソッドがあります:

register(*subclass*)

subclass を ”仮想的サブクラス” としてこの ABC に登録します。たとえば:

```
from abc import ABCMeta

class MyABC:
    __metaclass__ = ABCMeta

MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

また、次のメソッドを抽象基底クラスの中でオーバーライドできます:

__subclasshook__(*subclass*)

(クラスメソッドとして定義しなければなりません。)

subclass がこの ABC のサブクラスと見なせるかどうかチェックします。これによって ABC のサブクラスと見なしたい全てのクラスについて `register()` を呼び出すことなく `issubclass` の振る舞いをさらにカスタマイズできます。(このクラスメソッドは ABC の `__subclasscheck__()` メソッドから呼び出されます。)

このメソッドは `True`, `False` または `NotImplemented` を返さなければなりません。 `True` を返す場合は、*subclass* はこの ABC のサブクラスと見なされます。 `False` を返す場合は、*subclass* はたとえ通常の意味でサブクラスであっても ABC のサブクラスではないと見なされます。 `NotImplemented` の場合、サブクラスチェックは通常のメカニズムに戻ります。

この概念のデモとして、次の ABC 定義の例を見てください:

```
class Foo(object):
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
        return iter(self)

class MyIterable:
    __metaclass__ = ABCMeta

    @abstractmethod
    def __iter__(self):
        while False:
            yield None

    def get_iterator(self):
        return self.__iter__()
```

(次のページに続く)

(前のページからの続き)

```

@classmethod
def __subclasshook__(cls, C):
    if cls is MyIterable:
        if any("__iter__" in B.__dict__ for B in C.__mro__):
            return True
        return NotImplemented

MyIterable.register(Foo)

```

ABC `MyIterable` は標準的なイテラブルのメソッド `__iter__()` を抽象メソッドとして定義します。ここで与えられている抽象クラスの実装は、サブクラスから呼び出すことができます。`get_iterator()` メソッドも `MyIterable` 抽象基底クラスの一部ですが、抽象的でない派生クラスはこれをオーバーライドする必要はありません。

ここで定義されるクラスメソッド `__subclasshook__()` の意味は、`__iter__()` メソッドがクラスの (または `__mro__` でアクセスされる基底クラスの一つの) `__dict__` にある場合にもそのクラスが `MyIterable` だと見なされるということです。

最後に、一番下の行は `Foo` を `__iter__()` メソッドを定義しないにもかかわらず `MyIterable` の仮想的サブクラスにします (`Foo` は古い様式の `__len__()` と `__getitem__()` を用いたイテレータプロトコルを使っています。)。これによって `Foo` のメソッドとして `get_iterator` が手に入るわけではなく、ことに注意してください。それは別に提供されています。

以下のデコレータも提供しています:

`abc.abstractmethod(function)`

抽象メソッドを示すデコレータです。

このデコレータを使うにはクラスのメタクラスが `ABCMeta` であるかまたは派生したものであることが求められます。`ABCMeta` から派生したメタクラスを持つクラスは全ての抽象メソッドおよびプロパティをオーバーライドしない限りインスタンス化できません。抽象メソッドは普通の 'super' 呼び出し機構を使って呼び出すことができます。

クラスに動的に抽象メソッドを追加する、あるいはメソッドやクラスが作られた後から抽象的かどうかの状態を変更しようと試みることは、サポートされません。`abstractmethod()` が影響を与えるのは正規の継承により派生したサブクラスのみで、ABC の `register()` メソッドで登録された "仮想的サブクラス" は影響されません。

使い方:

```

class C:
    __metaclass__ = ABCMeta
    @abstractmethod
    def my_abstract_method(self, ...):
        ...

```

注釈: Java の抽象メソッドと違い、これらの抽象メソッドは実装を持ち得ます。この実装は `super()` メカニズムを通してそれをオーバーライドしたクラスから呼び出すことができます。これは協調的多重

継承を使ったフレームワークにおいて `super` 呼び出しの終点として有効です。

`abc.abstractproperty([fget[, fset[, fdel[, doc]]]])`

組み込みの `property()` のサブクラスで、抽象プロパティであることを示します。

この関数を使うにはクラスのメタクラスが `ABCMeta` であるかまたは派生したものであることが求められます。`ABCMeta` から派生したメタクラスを持つクラスは全ての抽象メソッドおよびプロパティをオーバーライドしない限りインスタンス化できません。抽象プロパティは普通の 'super' 呼び出し機構を使って呼び出すことができます。

使い方:

```
class C:
    __metaclass__ = ABCMeta
    @abstractproperty
    def my_abstract_property(self):
        ...
```

この例は読み取り専用のプロパティを定義しています。プロパティ定義の「長い」形式の宣言を使って、読み書き出来る抽象プロパティを定義することができます:

```
class C:
    __metaclass__ = ABCMeta
    def getx(self): ...
    def setx(self, value): ...
    x = abstractproperty(getx, setx)
```

28.9 atexit — 終了ハンドラ

バージョン 2.0 で追加.

ソースコード: [Lib/atexit.py](#)

`atexit` モジュールは、クリーンアップ関数の登録を行う関数を定義します。登録された関数はインタプリタの通常終了時に自動的に実行されます。`atexit` はそれら関数を登録した順と 逆に 実行します; A、B、C の順に登録した場合、インタプリタ終了時に C、B、A の順に実行されます。

注意: このモジュールを使用して登録された関数は、プログラムが Python が扱わないシグナルによって kill された場合、Python 内部で致命的なエラーが検出された場合、あるいは `os._exit()` が呼び出された場合は実行されません。

このモジュールは、`sys.exitfunc` 変数の提供していた機能の代用となるインタフェースです。

注意: `sys.exitfunc` を設定する他のコードとともに使用した場合には、このモジュールは正しく動作しないでしょう。特に、他のコア Python モジュールでは、プログラマの意図を知らなくても `atexit` を自由に使えます。`sys.exitfunc` を使っている人は、代わりに `atexit` を使うコードに変換してください。

`sys.exitfunc` を設定するコードを変換するには、`atexit` を import し、`sys.exitfunc` へ束縛されていた関数を登録するのが最も簡単です。

```
atexit.register(func[, *args[, **kwargs]])
```

`func` を終了時に実行する関数として登録します。`func` に渡す引数は `register()` の引数として指定しなければなりません。同じ関数を同じ引数で複数回登録できます。

通常のプログラムの終了時、例えば `sys.exit()` が呼び出されるとき、あるいは、メインモジュールの実行が完了したときに、登録された全ての関数を、最後に登録されたものから順に呼び出します。通常、より低レベルのモジュールはより高レベルのモジュールより前に import されるので、後で後始末が行われるという仮定に基づいています。

終了ハンドラの実行中に例外が発生すると、(`SystemExit` 以外の場合は) トレースバックを表示して、例外の情報を保存します。全ての終了ハンドラに動作するチャンスを与えた後に、最後に送出された例外を再送出します。

バージョン 2.6 で変更: この関数は `func` を返すようになったので、これをデコレータとして利用できます。

参考:

`readline` モジュール `readline` ヒストリファイルを読み書きするための `atexit` の有用な例です。

28.9.1 atexit の例

次の簡単な例では、あるモジュールを import した時にカウンタを初期化しておき、プログラムが終了するときにアプリケーションがこのモジュールを明示的に呼び出さなくてもカウンタが更新されるようにする方法を示しています。

```
try:
    _count = int(open("counter").read())
except IOError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    open("counter", "w").write("%d" % _count)

import atexit
atexit.register(savecounter)
```

`register()` に指定した位置引数とキーワード引数は登録した関数を呼び出す際に渡されます:

```
def goodbye(name, adjective):
    print 'Goodbye, %s, it was %s to meet you.' % (name, adjective)
```

(次のページに続く)

(前のページからの続き)

```
import atexit
atexit.register(goodbye, 'Donny', 'nice')

# or:
atexit.register(goodbye, adjective='nice', name='Donny')
```

デコレータとして利用する例:

```
import atexit

@atexit.register
def goodbye():
    print "You are now leaving the Python sector."
```

デコレータとして利用できるのは、その関数が引数なしで呼び出された場合に限られます。

28.10 traceback — スタックトレースの表示または取得

このモジュールは Python プログラムのスタックトレースを抽出し、書式を整え、表示するための標準インターフェースを提供します。モジュールがスタックトレースを表示するとき、Python インタプリタの動作を正確に模倣します。インタプリタの”ラッパー”の場合のように、プログラムの制御の元でスタックトレースを表示したいと思ったときに役に立ちます。

モジュールは traceback オブジェクトを使います — これは変数 `sys.exc_traceback` (非推奨) と `sys.last_traceback` に保存され、`sys.exc_info()` から三番目の項目として返されるオブジェクト型です。

このモジュールでは以下の関数を定義しています:

`traceback.print_tb(tb[, limit[, file]])`

トレースバックオブジェクト `tb` から `limit` までのスタックトレース項目を出力します。 `limit` が省略されるか `None` の場合は、すべての項目が表示されます。 `file` が省略されるか `None` の場合は、`sys.stderr` へ出力されます。それ以外の場合は、出力を受けるためのオープンされたファイルまたはファイル風 (file-like) オブジェクトでなければなりません。

`traceback.print_exception(etype, value, tb[, limit[, file]])`

例外情報とトレースバック `tb` から `limit` までのスタックトレース項目を `file` へ出力します。これは以下のような点で `print_tb()` とは異なります: (1) `tb` が `None` でない場合は、ヘッダ `Traceback (most recent call last):` を出力します。 (2) スタックトレースの後に例外 `etype` と `value` を出力します。 (3) `etype` が `SyntaxError` であり、`value` が適切な形式の場合は、エラーのおおよその位置を示すカレットを付けて構文エラーが起きた行を出力します。

`traceback.print_exc([limit[, file]])`

これは `print_exception(sys.exc_type, sys.exc_value, sys.exc_traceback, limit, file)` の省略表現です。(非推奨の変数を使う代わりにスレッドセーフな方法で同じ情報を引き出すために、実際には `sys.exc_info()` を使います。)

```
traceback.format_exc([limit])
```

これは、`print_exc(limit)` に似ていますが、ファイルに出力する代わりに文字列を返します。

バージョン 2.4 で追加。

```
traceback.print_last([limit[, file]])
```

`print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit, file)` の省略表現です。一般に、例外が対話的なプロンプトに達した後にだけ機能します (`sys.last_type` 参照)。

```
traceback.print_stack([f[, limit[, file]]])
```

This function prints a stack trace from its invocation point. The optional *f* argument can be used to specify an alternate stack frame to start. The optional *limit* and *file* arguments have the same meaning as for `print_exception()`.

```
traceback.extract_tb(tb[, limit])
```

トレースバックオブジェクト *tb* から *limit* まで取り出された”前処理済み”スタックトレース項目のリストを返します。スタックトレースの代わりの書式設定を行うために役に立ちます。*limit* が省略されるか `None` の場合は、すべての項目が取り出されます。”前処理済み”スタックトレース項目とは 4 要素のタプル (*filename*, *line number*, *function name*, *text*) で、スタックトレースに対して通常出力される情報を表しています。*text* は前後の空白を取り除いた文字列です。ソースが利用できない場合は `None` です。

```
traceback.extract_stack([f[, limit]])
```

現在のスタックフレームから生のトレースバックを取り出します。戻り値は `extract_tb()` と同じ形式です。オプションの *f* と *limit* 引数は `print_stack()` と同じ意味を持ちます。

```
traceback.format_list(extracted_list)
```

`extract_tb()` または `extract_stack()` が返すタプルのリストが与えられると、出力の準備を整えた文字列のリストを返します。結果として生じるリストの中の各文字列は、引数リストの中の同じインデックスの要素に対応します。各文字列は末尾に改行が付いています。さらに、ソーステキスト行が `None` でないそれらの要素に対しては、文字列は内部に改行を含んでいるかもしれません。

```
traceback.format_exception_only(etype, value)
```

トレースバックの例外部分をフォーマットします。引数は `sys.last_type` と `sys.last_value` で与えられるような、例外の型 *etype* と値 *value* です。戻り値はそれぞれが改行で終わっている文字列のリストです。通常、リストは一つの文字列を含んでいます。しかし、`SyntaxError` 例外に対しては、(出力されるときに) 構文エラーが起きた場所についての詳細な情報を示す行をいくつか含んでいます。どの例外が起きたのかを示すメッセージは、常にリストの最後の文字列です。

```
traceback.format_exception(etype, value, tb[, limit])
```

スタックトレースと例外情報を書式化します。引数は `print_exception()` の対応する引数と同じ意味を持ちます。戻り値は文字列のリストで、それぞれの文字列は改行で終わり、そのいくつかは内部に改行を含みます。これらの行が連結されて出力される場合は、厳密に `print_exception()` と同じテキストが出力されます。

```
traceback.format_tb(tb[, limit])
```

`format_list(extract_tb(tb, limit))` の省略表現です。

```
traceback.format_stack([f[, limit]])
```

`format_list(extract_stack(f, limit))` の省略表現です。

```
traceback.tb_lineno(tb)
```

トレースバックオブジェクトに設定された現在の行番号を返します。Python 2.3 より前のバージョンでは、`-O` フラグが渡されたときに `tb.tb_lineno` が正しく更新されなかったため、この関数は必要でした。2.3 以降のバージョンでは不要です。

28.10.1 トレースバックの例

この簡単な例では基本的な read-eval-print ループを実装しています。標準的な Python の対話インタプリタループに似ていますが、Python のものより便利ではありません。インタプリタループのより完全な実装については、`code` モジュールを参照してください。

```
import sys, traceback

def run_user_code(envdir):
    source = raw_input(">>> ")
    try:
        exec source in envdir
    except:
        print "Exception in user code:"
        print '-'*60
        traceback.print_exc(file=sys.stdout)
        print '-'*60

envdir = {}
while 1:
    run_user_code(envdir)
```

次の例は例外とトレースバックの出力並びに形式が異なることを示します:

```
import sys, traceback

def lumberjack():
    bright_side_of_death()

def bright_side_of_death():
    return tuple()[0]

try:
    lumberjack()
except IndexError:
    exc_type, exc_value, exc_traceback = sys.exc_info()
    print "*** print_tb:"
    traceback.print_tb(exc_traceback, limit=1, file=sys.stdout)
    print "*** print_exception:"
    traceback.print_exception(exc_type, exc_value, exc_traceback,
                             limit=2, file=sys.stdout)
    print "*** print_exc:"
```

(次のページに続く)

(前のページからの続き)

```

traceback.print_exc()
print "*** format_exc, first and last line:"
formatted_lines = traceback.format_exc().splitlines()
print formatted_lines[0]
print formatted_lines[-1]
print "*** format_exception:"
print repr(traceback.format_exception(exc_type, exc_value,
                                     exc_traceback))

print "*** extract_tb:"
print repr(traceback.extract_tb(exc_traceback))
print "*** format_tb:"
print repr(traceback.format_tb(exc_traceback))
print "*** tb_lineno:", exc_traceback.tb_lineno

```

この例の出力は次のようになります:

```

*** print_tb:
  File "<doctest...>", line 10, in <module>
    lumberjack()
*** print_exception:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** print_exc:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** format_exc, first and last line:
Traceback (most recent call last):
IndexError: tuple index out of range
*** format_exception:
['Traceback (most recent call last):\n',
 '  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n',
 'IndexError: tuple index out of range\n']
*** extract_tb:
[('<doctest...>', 10, '<module>', 'lumberjack()'),
 ('<doctest...>', 4, 'lumberjack', 'bright_side_of_death()'),
 ('<doctest...>', 7, 'bright_side_of_death', 'return tuple()[0]')]
*** format_tb:
['  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n']
*** tb_lineno: 10

```

次の例は、スタックの print と format の違いを示しています:

```
>>> import traceback
>>> def another_function():
...     lumberstack()
...
>>> def lumberstack():
...     traceback.print_stack()
...     print repr(traceback.extract_stack())
...     print repr(traceback.format_stack())
...
>>> another_function()
File "<doctest>", line 10, in <module>
    another_function()
File "<doctest>", line 3, in another_function
    lumberstack()
File "<doctest>", line 6, in lumberstack
    traceback.print_stack()
[('<doctest>', 10, '<module>', 'another_function()'),
 ('<doctest>', 3, 'another_function', 'lumberstack()'),
 ('<doctest>', 7, 'lumberstack', 'print repr(traceback.extract_stack())')]
[' File "<doctest>", line 10, in <module>\n    another_function()\n',
 ' File "<doctest>", line 3, in another_function\n    lumberstack()\n',
 ' File "<doctest>", line 8, in lumberstack\n    print repr(traceback.format_
↳ stack())\n']
```

最後の例は、残りの幾つかの関数のデモをします:

```
>>> import traceback
>>> traceback.format_list([('spam.py', 3, '<module>', 'spam.eggs()'),
...                        ('eggs.py', 42, 'eggs', 'return "bacon"')])
[' File "spam.py", line 3, in <module>\n    spam.eggs()\n',
 ' File "eggs.py", line 42, in eggs\n    return "bacon"\n']
>>> an_error = IndexError('tuple index out of range')
>>> traceback.format_exception_only(type(an_error), an_error)
['IndexError: tuple index out of range\n']
```

28.11 `__future__` — `future` 文の定義

ソースコード: [Lib/_future_.py](#)

`__future__` は実際にモジュールであり、次の 3 つの役割があります:

- `import` 文を解析する既存のツールを混乱させることを避け、インポートしようとしているモジュールを見つけられるようにするため。
- 2.1 以前のリリースで `future` 文 が実行された場合に、最低でもランタイム例外を投げるようにするため (`__future__` のインポートは失敗します。なぜなら、2.1 以前にはそういう名前のモジュールはなかったからです)。

- 互換性のない変化がいつ言語に導入され、いつ言語の一部になる — あるいは、なった — のかを文書化するため。これは実行できる形式で書かれたドキュメントなので、`__future__` をインポートしてその中身を調べることでプログラムから調査することができます。

`__future__.py` のそれぞれの文は次のような形式をしています:

```
FeatureName = _Feature(OptionalRelease, MandatoryRelease,
                        CompilerFlag)
```

ここで、普通は、*OptionalRelease* は *MandatoryRelease* より小さく、2 つとも `sys.version_info` と同じフォーマットの 5 つのタプルからなります。

```
(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int
PY_MINOR_VERSION, # the 1; an int
PY_MICRO_VERSION, # the 0; an int
PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; string
PY_RELEASE_SERIAL # the 3; an int
)
```

OptionalRelease はその機能が導入された最初のリリースを記録します。

まだ時期が来ていない *MandatoryRelease* の場合、*MandatoryRelease* はその機能が言語の一部となるリリースを記します。

その他の場合、*MandatoryRelease* はその機能がいつ言語の一部になったのかを記録します。そのリリースから、あるいはそれ以降のリリースでは、この機能を使う際に `future` 文は必要ではありませんが、`future` 文を使い続けても構いません。

MandatoryRelease は `None` になるかもしれません。つまり、予定された機能が破棄されたということです。

`_Feature` クラスのインスタンスには対応する 2 つのメソッド、`getOptionalRelease()` と `getMandatoryRelease()` があります。

CompilerFlag は、動的にコンパイルされるコードでその機能を有効にするために、組み込み関数 `compile()` の第 4 引数に渡す (ビットフィールド) フラグです。このフラグは `_Feature` インスタンスの `compiler_flag` 属性に保存されています。

機能の記述が `__future__` から削除されたことはまだありません。Python 2.1 で `future` 文が導入されて以来、この仕組みを使って以下の機能が言語に導入されてきました:

feature	optional in	mandatory in	effect
nested_scopes	2.1.0b1	2.2	PEP 227: Statically Nested Scopes
generators	2.2.0a1	2.3	PEP 255: Simple Generators
division	2.2.0a2	3.0	PEP 238: Changing the Division Operator
absolute_import	2.5.0a1	3.0	PEP 328: Imports: Multi-Line and Absolute/Relative
with_statement	2.5.0a1	2.6	PEP 343: The "with" Statement
print_function	2.6.0a2	3.0	PEP 3105: Make print a function
unicode_literals	2.6.0a2	3.0	PEP 3112: Bytes literals in Python 3000

参考:

future コンパイラがどのように future インポートを扱うか。

28.12 gc — ガベージコレクタインターフェース

このモジュールは、循環ガベージコレクタの無効化・検出頻度の調整・デバッグオプションの設定などを行うインターフェースを提供します。また、検出した到達不能オブジェクトのうち、解放する事ができないオブジェクトを参照する事もできます。循環ガベージコレクタは Python の参照カウントを補うためのものなので、もしプログラム中で循環参照が発生しない事が明らかな場合には検出をする必要はありません。自動検出は、`gc.disable()` で停止する事ができます。メモリリークをデバッグするときには、`gc.set_debug(gc.DEBUG_LEAK)` とします。これは `gc.DEBUG_SAVEALL` を含んでいることに注意しましょう。ガベージとして検出されたオブジェクトは、インスペクション用に `gc.garbage` に保存されます。

`gc` モジュールは、以下の関数を提供しています:

`gc.enable()`

自動ガベージコレクションを有効にします。

`gc.disable()`

自動ガベージコレクションを無効にします。

`gc.isenabled()`

自動ガベージコレクションが有効なら真を返します。

`gc.collect([generation])`

引数を指定しない場合は、全ての検出を行います。オプションの引数 *generation* は、どの世代を検出するかを (0 から 2 までの) 整数値で指定します。無効な世代番号を指定した場合は `ValueError` が発生します。検出した到達不可オブジェクトの数を返します。

バージョン 2.5 で変更: オプションの引数 *generation* が追加されました。

バージョン 2.6 で変更: 幾つかの組み込み型のフリーリストは、最高 (第二) 世代の GC、あるいはフル GC が実行されるたびにクリアされます。幾つかの型 (特に `int` と `float`) では、実装によっては、フ

リーリスト内の全てのオブジェクトが解放されるとは限りません。

`gc.set_debug(flags)`

ガベージコレクションのデバッグフラグを設定します。デバッグ情報は `sys.stderr` に出力されます。デバッグフラグは、下の値の組み合わせを指定する事ができます。

`gc.get_debug()`

現在のデバッグフラグを返します。

`gc.get_objects()`

現在追跡しているオブジェクトのリストを返します。このリストには、戻り値のリスト自身は含まれていません。

バージョン 2.2 で追加。

`gc.set_threshold(threshold0[, threshold1[, threshold2]])`

ガベージコレクションの閾値 (検出頻度) を指定します。 `threshold0` を 0 にすると、検出は行われません。

GC は、オブジェクトを走査された回数に従って 3 世代に分類します。新しいオブジェクトは最も若い (0 世代) に分類されます。もし、そのオブジェクトがガベージコレクションで削除されなければ、次に古い世代に分類されます。もっとも古い世代は 2 世代で、この世代に属するオブジェクトは他の世代に移動しません。ガベージコレクタは、最後に検出を行ってから生成・削除したオブジェクトの数をカウントしており、この数によって検出を開始します。オブジェクトの生成数 - 削除数が `threshold0` より大きくなると、検出を開始します。最初は 0 世代のオブジェクトのみが検査されます。0 世代の検査が `threshold1` 回実行されると、1 世代のオブジェクトの検査を行います。同様に、1 世代が `threshold2` 回検査されると、2 世代の検査を行います。

`gc.get_count()`

現在の検出数を、(`count0`, `count1`, `count2`) のタプルで返します。

バージョン 2.5 で追加。

`gc.get_threshold()`

現在の検出閾値を、(`threshold0`, `threshold1`, `threshold2`) のタプルで返します。

`gc.get_referrers(*objs)`

`objs` で指定したオブジェクトのいずれかを参照しているオブジェクトのリストを返します。この関数では、ガベージコレクションをサポートしているコンテナのみを返します。他のオブジェクトを参照していても、ガベージコレクションをサポートしていない拡張型は含まれません。

尚、戻り値のリストには、すでに参照されなくなっているが、循環参照の一部でまだガベージコレクションで回収されていないオブジェクトも含まれるので注意が必要です。有効なオブジェクトのみを取得する場合、`get_referrers()` の前に `collect()` を呼び出してください。

`get_referrers()` から返されるオブジェクトは作りかけや利用できない状態である場合があるので、利用するには注意が必要です。`get_referrers()` をデバッグ以外の目的で利用するのは避けてください。

バージョン 2.2 で追加。

`gc.get_referents(*objs)`

引数で指定したオブジェクトのいずれかから参照されている、全てのオブジェクトのリストを返します。参照先のオブジェクトは、引数で指定したオブジェクトの C レベルメソッド `tp_traverse` で取得し、全てのオブジェクトが直接到達可能な全てのオブジェクトを返すわけではありません。 `tp_traverse` はガベージコレクションをサポートするオブジェクトのみが実装しており、ここで取得できるオブジェクトは循環参照の一部となる可能性のあるオブジェクトのみです。従って、例えば整数オブジェクトが直接到達可能であっても、このオブジェクトは戻り値には含まれません。

バージョン 2.3 で追加。

`gc.is_tracked(obj)`

`obj` が GC に管理されていたら `True` を返し、それ以外の場合は `False` を返します。一般的なルールとして、アトミックな (訳注:他のオブジェクトを参照しないで単一で値を表す型。 `int` や `str` など) 型のインスタンスは管理されず、それ以外の型 (コンテナ型、ユーザー定義型など) のインスタンスは管理されます。しかし、いくつかの型では専用の最適化を行い、シンプルなインスタンスの場合に GC のオーバーヘッドを減らしています。(例: 全ての `key` と `value` がアトミック型の値である `dict`)

```
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
False
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
False
>>> gc.is_tracked({"a": []})
True
```

バージョン 2.7 で追加。

以下の変数は読み込み専用です。(変更することはできますが、再バインドする事はできません。)

`gc.garbage`

到達不能であることが検出されたが、解放する事ができないオブジェクトのリスト (回収不能オブジェクト)。デフォルトでは、`__del__()` メソッドを持つオブジェクトのみが格納されます。^{*1} `__del__()` メソッドを持つオブジェクトが循環参照に含まれている場合、その循環参照全体と、循環参照からのみ到達する事ができるオブジェクトは回収不能となります。このような場合には、Python は安全に `__del__()` を呼び出す順番を決定する事ができないため、自動的に解放することはできません。もし安全な解放順序がわかるのであれば、`garbage` リストを参照して循環参照を破壊する事ができます。循環参照を破壊した後でも、そのオブジェクトは `garbage` リストから参照されているため、解放されません。解放するためには、循環参照を破壊した後、`del gc.garbage[:]` のように `garbage` からオブジェクトを削除する必要があります。一般的には `__del__()` を持つオブジェクトが循環参照の一部とはならないように配慮し、`garbage` はそのような循環参照が発生していない事を確認するために利用する方が良いでしょう。

^{*1} Python 2.2 より前のバージョンでは、`__del__()` メソッドを持つオブジェクトだけでなく、全ての到達不能オブジェクトが格納されていた。)

`DEBUG_SAVEALL` が設定されている場合、全ての到達不能オブジェクトは解放されずにこのリストに格納されます。

以下は `set_debug()` に指定することのできる定数です:

`gc.DEBUG_STATS`

検出中に統計情報を出力します。この情報は、検出頻度を最適化する際に有益です。

`gc.DEBUG_COLLECTABLE`

見つかった回収可能オブジェクトの情報を出力します。

`gc.DEBUG_UNCOLLECTABLE`

見つかった回収不能オブジェクト (到達不能だが、ガベージコレクションで解放する事ができないオブジェクト) の情報を出力します。回収不能オブジェクトは、`garbage` リストに追加されます。

`gc.DEBUG_INSTANCES`

`DEBUG_COLLECTABLE` か `DEBUG_UNCOLLECTABLE` が設定されている場合、見つかったインスタンスオブジェクトの情報を出力します。

`gc.DEBUG_OBJECTS`

`DEBUG_COLLECTABLE` か `DEBUG_UNCOLLECTABLE` が設定されている場合、見つかったインスタンスオブジェクト以外のオブジェクトの情報を出力します。

`gc.DEBUG_SAVEALL`

設定されている場合、全ての到達不能オブジェクトは解放されずに `garbage` に追加されます。これはプログラムのメモリリークをデバッグするときに便利です。

`gc.DEBUG_LEAK`

プログラムのメモリリークをデバッグするときに指定します。(`DEBUG_COLLECTABLE` | `DEBUG_UNCOLLECTABLE` | `DEBUG_INSTANCES` | `DEBUG_OBJECTS` | `DEBUG_SAVEALL` と同じ。)

脚注

28.13 `inspect` — 活動中のオブジェクトの情報を取得する

バージョン 2.1 で追加.

ソースコード: [Lib/inspect.py](#)

`inspect` は、活動中のオブジェクト (モジュール、クラス、メソッド、関数、トレースバック、フレームオブジェクト、コードオブジェクトなど) から情報を取得する関数を定義しており、クラスの内容を調べたり、メソッドのソースコードを取得したり、関数の引数リストを取り出して整形したり、詳細なトレースバックを表示するのに必要な情報を取得したりするために利用できます。

このモジュールの機能は 4 種類に分類することができます。型チェック、ソースコードの情報取得、クラスや関数からの情報取得、インタープリタのスタック情報の調査です。

28.13.1 型とメンバー

`getmembers()` は、クラスやモジュールなどのオブジェクトからメンバーを取得します。名前が`__is__`で始まる 16 個の関数は、主に `getmembers()` の第 2 引数として利用するために提供されています。以下のような特殊属性を参照できるかどうか調べる時にも使えるでしょう:

型	属性	説明
モジュール	<code>__doc__</code>	ドキュメント文字列
	<code>__file__</code>	ファイル名 (組み込みモジュールには存在しません)
クラス	<code>__doc__</code>	ドキュメント文字列
	<code>__module__</code>	クラスを定義しているモジュールの名前
メソッド	<code>__doc__</code>	ドキュメント文字列
	<code>__name__</code>	メソッドが定義された時の名前
	<code>im_class</code>	このメソッドを要求したクラスオブジェクト
	<code>im_func</code> または <code>__func__</code>	メソッドを実装している関数オブジェクト
	<code>im_self</code> または <code>__self__</code>	メソッドに結合しているインスタンス、または <code>None</code>
function	<code>__doc__</code>	ドキュメント文字列
	<code>__name__</code>	関数が定義された時の名前
	<code>func_code</code>	関数をコンパイルしたバイトコード (<i>bytecode</i>) を格納するコードオブジェクト
	<code>func_defaults</code>	引数のデフォルト値のタプル
	<code>func_doc</code>	(<code>__doc__</code> と同じ)
	<code>func_globals</code>	関数を定義した時のグローバル名前空間
	<code>func_name</code>	(<code>__name__</code> と同じ)
generator	<code>__iter__</code>	コンテナを通したイテレーションのために定義されます
	<code>close</code>	イテレーションを停止するために、ジェネレータの内部で <code>GeneratorExit</code> exception を発生させます
	<code>gi_code</code>	コードオブジェクト
	<code>gi_frame</code>	フレームオブジェクト。ジェネレータが終了したあとは <code>None</code> になることもあります
	<code>gi_running</code>	ジェネレータが実行中の時は 1 それ以外の場合は 0
	<code>next</code>	コンテナから次の要素を返します
	<code>send</code>	ジェネレータを再開して、現在の <code>yield</code> 式の結果となる値を ”送り” ます
	<code>throw</code>	ジェネレータ内部で例外を発生させるために用います
	<code>__throw__</code>	ジェネレータ内部で例外を発生させるために用います
トレースバック	<code>tb_frame</code>	このレベルのフレームオブジェクト
	<code>tb_lasti</code>	最後に実行しようとしたバイトコード中のインストラクションを示すインデックス
	<code>tb_lineno</code>	現在の Python ソースコードの行番号
	<code>tb_next</code>	このオブジェクトの内側 (このレベルから呼び出された) のトレースバックオブジェクト
frame	<code>f_back</code>	外側 (このフレームを呼び出した) のフレームオブジェクト
	<code>f_builtins</code>	このフレームで参照している組み込み名前空間
	<code>f_code</code>	このフレームで実行しているコードオブジェクト
	<code>f_exc_traceback</code>	このフレームで例外が発生した場合にはトレースバックオブジェクト、それ以外なら <code>None</code>
	<code>f_exc_type</code>	このフレームで例外が発生した場合には例外型、それ以外なら <code>None</code>
	<code>f_exc_value</code>	このフレームで例外が発生した場合には例外の値、それ以外なら <code>None</code>

表 1 – 前のページからの続き

型	属性	説明
	<code>f_globals</code>	このフレームで参照しているグローバル名前空間
	<code>f_lasti</code>	最後に実行しようとしたバイトコード中のインストラクションを示すインデックス
	<code>f_lineno</code>	現在の Python ソースコードの行番号
	<code>f_locals</code>	このフレームで参照しているローカル名前空間
	<code>f_restricted</code>	制限実行モードなら 1、それ以外なら 0
	<code>f_trace</code>	このフレームのトレース関数、または <code>None</code>
コード	<code>co_argcount</code>	引数の数 (* や ** 引数は含まない)
	<code>co_code</code>	コンパイルされたバイトコードそのままの文字列
	<code>co_consts</code>	バイトコード中で使用している定数のタプル
	<code>co_filename</code>	コードオブジェクトを生成したファイルのファイル名
	<code>co_firstlineno</code>	Python ソースコードの先頭行
	<code>co_flags</code>	以下の値の組み合わせ: 1=optimized 2=newlocals 4=*arg 8=**arg
	<code>co_lnotab</code>	行番号からバイトコードインデックスへの変換表を文字列にエンコードしたもの
	<code>co_name</code>	コードオブジェクトが定義されたときの名前
	<code>co_names</code>	ローカル変数名のタプル
	<code>co_nlocals</code>	ローカル変数の数
	<code>co_stacksize</code>	必要とされる仮想マシンのスタックスペース
	<code>co_varnames</code>	引数名とローカル変数名のタプル
builtin	<code>__doc__</code>	ドキュメント文字列
	<code>__name__</code>	関数、メソッドの元々の名前
	<code>__self__</code>	メソッドが結合しているインスタンス、または <code>None</code>

注釈:

- (1) バージョン 2.2 で変更: `im_class` は、以前はメソッドを定義しているクラスを指していました。

`inspect.getmembers(object[, predicate])`

オブジェクトの全メンバーを、(名前, 値) の組み合わせのリストで返します。リストはメンバー名でソートされています。 *predicate* が指定されている場合、 *predicate* の戻り値が真となる値のみを返します。

注釈: `getmembers()` は、引数がクラスの場合にメタクラス属性を返しません (この動作は `dir()` 関数に合わせてあります)。

`inspect.getmoduleinfo(path)`

path で指定したファイルがモジュールであれば、そのモジュールが Python でどのように解釈されるかを示す (name, suffix, mode, module_type) のタプルを返します。モジュールでなければ `None` を返します。 *name* はパッケージ名を含まないモジュール名、 *suffix* はファイル名からモジュール名を除いた残りの部分 (ドット区切りの拡張子であってはなりません)、 *mode* は `open()` で指定されるファイルモード ('r' または 'rb')、 *module_type* はモジュールタイプを示す整数で、 `imp` で定義している定数のいずれかが指定されます。モジュールタイプについては `imp` を参照してください。

バージョン 2.6 で変更: 名前付きタプル (*named tuple*) の `ModuleInfo(name, suffix, mode, module.type)` を返します。

`inspect.getmodulename(path)`

`path` で指定したファイルの、パッケージ名を含まないモジュール名を返します。この処理は、インタプリタがモジュールを検索する時と同じアルゴリズムで行われます。ファイルがこのアルゴリズムで見つからない場合には `None` が返ります。

`inspect.ismodule(object)`

オブジェクトがモジュールの場合は真を返します。

`inspect.isclass(object)`

オブジェクトが、ビルトインもしくは Python で作られたクラスの場合に真を返します。

`inspect.ismethod(object)`

オブジェクトが Python で実装された束縛メソッドもしくは非束縛メソッドの場合は真を返します。

`inspect.isfunction(object)`

オブジェクトが Python の関数 (*lambda* 式で生成されたものを含む) である場合に真を返します。

`inspect.isgeneratorfunction(object)`

`object` が Python のジェネレータ関数であるときに真を返します。

バージョン 2.6 で追加.

`inspect.isgenerator(object)`

`object` がジェネレータであるときに真を返します。

バージョン 2.6 で追加.

`inspect.istraceback(object)`

オブジェクトがトレースバックの場合は真を返します。

`inspect.isframe(object)`

オブジェクトがフレームの場合は真を返します。

`inspect.iscode(object)`

オブジェクトがコードの場合は真を返します。

`inspect.isbuiltin(object)`

オブジェクトが組み込み関数が束縛済みのビルトインメソッドの場合に真を返します。

`inspect.isroutine(object)`

オブジェクトがユーザ定義か組み込みの関数またはメソッドの場合は真を返します。

`inspect.isabstract(object)`

`object` が抽象規定型 (ABC) であるときに真を返します。

バージョン 2.6 で追加.

`inspect.ismethoddescriptor(object)`

オブジェクトがメソッドデスクリプタであり、 `ismethod()`, `isclass()`, `isfunction()`,

`isbuiltin()` でない場合に真を返します。

この機能は Python 2.2 から新たに追加されたもので、例えば `int.__add__` は真になります。このテストをパスするオブジェクトは `__get__()` メソッドを持ちますが `__set__()` メソッドを持ちません。それ以外の属性を持っているかもしれません。通常 `__name__` 属性を持っていますし、たいていは `__doc__` も持っています。

デスクリプタを使って実装されたメソッドで、上記のいずれかのテストもパスしているものは、`ismethoddescriptor()` では偽を返します。これは単に他のテストの方がもっと確実だからです。例えば、`ismethod()` をパスしたオブジェクトは `im_func` 属性などを持っていると期待できます。

`inspect.isdatadescriptor(object)`

オブジェクトがデータデスクリプタの場合に真を返します。

データデスクリプタは `__get__` および `__set__` 属性の両方を持ちます。データデスクリプタの例は (Python 上で定義された) プロパティや `getset` やメンバーです。後者のふたつは C で定義されており、個々の型に特有のテストも行います。そのため、Python の実装よりもより確実です。通常、データデスクリプタは `__name__` や `__doc__` 属性を持ちます (プロパティ、`getset`、メンバーは両方の属性を持っています) が、保証されているわけではありません。

バージョン 2.3 で追加。

`inspect.isgetsetdescriptor(object)`

オブジェクトが `getset` デスクリプタの場合に真を返します。

`getset` とは、拡張モジュールで `PyGetSetDef` 構造体を用いて定義された属性のことです。そのような型を持たない Python 実装の場合は、このメソッドは常に `False` を返します。

バージョン 2.5 で追加。

`inspect.ismemberdescriptor(object)`

オブジェクトがメンバーデスクリプタの場合に真を返します。

メンバーデスクリプタとは、拡張モジュールで `PyMemberDef` 構造体を用いて定義された属性のことです。そのような型を持たない Python 実装の場合は、このメソッドは常に `False` を返します。

バージョン 2.5 で追加。

28.13.2 ソースコードの情報取得

`inspect.getdoc(object)`

`cleandoc()` でクリーンアップされた、オブジェクトのドキュメンテーション文字列を取得します。

`inspect.getcomments(object)`

オブジェクトがクラス、関数、メソッドのいずれかの場合は、オブジェクトのソースコードの直後にあるコメント行 (複数行) を、単一の文字列として返します。オブジェクトがモジュールの場合、ソースファイルの先頭にあるコメントを返します。

`inspect.getfile(object)`

オブジェクトを定義している (テキストまたはバイナリの) ファイルの名前を返します。オブジェクトが組み込みモジュール、クラス、関数の場合は `TypeError` 例外が発生します。

`inspect.getmodule(object)`

オブジェクトを定義しているモジュールを推測します。

`inspect.getsourcefile(object)`

オブジェクトを定義している Python ソースファイルの名前を返します。オブジェクトが組み込みのモジュール、クラス、関数の場合には、`TypeError` 例外が発生します。

`inspect.getsourcelines(object)`

オブジェクトのソース行のリストと開始行番号を返します。引数にはモジュール、クラス、メソッド、関数、トレースバック、フレーム、コードオブジェクトを指定することができます。戻り値は指定したオブジェクトに対応するソースコードのソース行リストと元のソースファイル上での開始行となります。ソースコードを取得できない場合は `IOError` が発生します。

`inspect.getsource(object)`

オブジェクトのソースコードテキストを返します。引数にはモジュール、クラス、メソッド、関数、トレースバック、フレーム、コードオブジェクトを指定することができます。ソースコードは単一の文字列で返します。ソースコードを取得できない場合は `IOError` が発生します。

`inspect.cleandoc(doc)`

コードブロックと位置を合わせるためのインデントを `docstring` から削除します。

先頭行の行頭の空白文字は全て削除されます。2 行目以降では全行で同じ数の行頭の空白文字が、削除できるだけ削除されます。その後、先頭と末尾の空白行が削除され、全てのタブが空白に展開されます。

バージョン 2.6 で追加。

28.13.3 クラスと関数

`inspect.getclasstree(classes[, unique])`

リストで指定したクラスの継承関係から、ネストしたリストを作成します。ネストしたリストには、直前の要素から派生したクラスが格納されます。各要素は長さ 2 のタプルで、クラスと基底クラスのタプルを格納しています。`unique` が真の場合、各クラスは戻り値のリスト内に一つだけしか格納されません。真でなければ、多重継承を利用したクラスとその派生クラスは複数回格納される場合があります。

`inspect.getargspec(func)`

Python 関数の引数名とデフォルト値を取得します。戻り値は長さ 4 のタプルで、次の値を返します: (`args`, `varargs`, `keywords`, `defaults`)。 `args` は引数名のリストです (ネストしたリストが格納される場合があります)。 `varargs` と `keywords` は * 引数と ** 引数の名前で、引数がない場合は `None` となります。 `defaults` は引数のデフォルト値のタプルか、デフォルト値がない場合は `None` です。このタプルに `n` 個の要素があれば、各要素は `args` の後ろから `n` 個分の引数のデフォルト値となります。

バージョン 2.6 で変更: `ArgSpec(args, varargs, keywords, defaults)` 形式の名前付きタプル (*named tuple*) を返します。

`inspect.getargvalues (frame)`

指定したフレームに渡された引数の情報を取得します。戻り値は長さ 4 のタプルで、次の値を返します: (args, varargs, keywords, locals). *args* は引数名のリストです (ネストしたリストが格納される場合があります)。 *varargs* と *keywords* は * 引数と ** 引数の名前で、引数がなければ None となります。 *locals* は指定したフレームのローカル変数の辞書です。

バージョン 2.6 で変更: `ArgInfo(args, varargs, keywords, locals)` 形式の名前付きタプル (*named tuple*) を返します。

`inspect.formatargspec (args[, varargs, varkw, defaults, formatarg, formatvarargs, formatvarkw, formatvalue, join])`

`getargspec()` で取得した 4 つの値を読みやすく整形します。 *format** 引数はオプションで、名前と値を文字列に変換する整形関数を指定することができます。

`inspect.formatargvalues (args[, varargs, varkw, locals, formatarg, formatvarargs, formatvarkw, formatvalue, join])`

`getargvalues()` で取得した 4 つの値を読みやすく整形します。 *format** 引数はオプションで、名前と値を文字列に変換する整形関数を指定することができます。

`inspect.getmro (cls)`

cls クラスの基底クラス (*cls* 自身も含む) を、メソッドの優先順位順に並べたタプルを返します。結果のリスト内で各クラスは一度だけ格納されます。メソッドの優先順位はクラスの型によって異なります。非常に特殊なユーザ定義のメタクラスを使用していない限り、*cls* が戻り値の先頭要素となります。

`inspect.getcallargs (func[, *args][, **kwargs])`

args と *kwargs* を、Python の関数もしくはメソッド *func* を呼び出した場合と同じように引数名に束縛します。束縛済みメソッド (bound method) の場合、最初の引数 (慣習的に *self* という名前が付けられます) にも、関連づけられたインスタンスを束縛します。引数名 (* や ** 引数が存在すればその名前も) に *args* と *kwargs* からの値をマップした辞書を返します。 *func* を正しく呼び出せない場合、つまり `func(*args, **kwargs)` がシグネチャの不一致のために例外を投げるような場合には、それと同じ型で同じか似ているメッセージの例外を発生させます。例:

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
>>> getcallargs(f, 1, 2, 3)
{'a': 1, 'named': {}, 'b': 2, 'pos': (3,)}
>>> getcallargs(f, a=2, x=4)
{'a': 2, 'named': {'x': 4}, 'b': 1, 'pos': ()}
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() takes at least 1 argument (0 given)
```

バージョン 2.7 で追加.

28.13.4 インタープリタスタック

以下の関数には、戻り値として”フレームレコード”を返す関数があります。”フレームレコード”は長さ 6 のタプルで、以下の値を格納しています: フレームオブジェクト、ファイル名、実行中の行番号、関数名、コンテキストのソース行のリスト、ソース行のリストにおける実行中の行のインデックス。

注釈: フレームレコードの最初の要素などのフレームオブジェクトへの参照を保存すると、循環参照になってしまう場合があります。循環参照ができると、Python の循環参照検出機能を有効にしていたとしても関連するオブジェクトが参照しているすべてのオブジェクトが解放されにくくなり、明示的に参照を削除しないとメモリ消費量が増大する恐れがあります。

参照の削除を Python の循環参照検出機能にまかせることもできますが、`finally` 節で循環参照を解除すれば確実にフレーム (とそのローカル変数) は削除されます。また、循環参照検出機能は Python のコンパイルオプションや `gc.disable()` で無効とされている場合がありますので注意が必要です。例:

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
        del frame
```

以下の関数でオプション引数 `context` には、戻り値のソース行リストに何行分のソースを含めるかを指定します。ソース行リストには、実行中の行を中心として指定された行数分のリストを返します。

`inspect.getframeinfo(frame[, context])`

フレームまたはトレースバックオブジェクトの情報を取得します。フレームレコードの最後の 5 要素からなる長さ 5 のタプルを返します。

バージョン 2.6 で変更: `Traceback(filename, lineno, function, code_context, index)` 形式の名前付きタプル (*named tuple*) を返します。

`inspect.getouterframes(frame[, context])`

指定したフレームと、その外側の全フレームのフレームレコードを返します。外側のフレームとは `frame` が生成されるまでのすべての関数呼び出しを示します。戻り値のリストの先頭は `frame` のフレームレコードで、末尾の要素は `frame` のスタックにある最も外側のフレームのフレームレコードとなります。

`inspect.getinnerframes(traceback[, context])`

指定したフレームと、その内側の全フレームのフレームレコードを返します。内のフレームとは `frame` から続く一連の関数呼び出しを示します。戻り値のリストの先頭は `traceback` のフレームレコードで、末尾の要素は例外が発生した位置を示します。

`inspect.currentframe()`

呼び出し元のフレームオブジェクトを返します。

この関数はインタプリタの Python スタックフレームサポートに依存します。これは Python のすべて

の実装に存在している保証はありません。Python スタックフレームサポートのない環境では、この関数は `None` を返します。

`inspect.stack([context])`

呼び出し元スタックのフレームレコードのリストを返します。最初の要素は呼び出し元のフレームレコードで、末尾の要素はスタックにある最も外側のフレームのフレームレコードとなります。

`inspect.trace([context])`

実行中のフレームと処理中の例外が発生したフレームの間のフレームレコードのリストを返します。最初の要素は呼び出し元のフレームレコードで、末尾の要素は例外が発生した位置を示します。

28.14 site — サイト固有の設定フック

ソースコード: [Lib/site.py](#)

このモジュールは初期化中に自動的にインポートされます。自動インポートはインタプリタの `-S` オプションで禁止できます。

このモジュールをインポートすることで、サイト固有のパスをモジュール検索パスへ付け加え、また、いくつかのビルトインを追加します。

サイト固有のパスは、前部と後部から構築される最大で四つまでのディレクトリから始めます。前部には、`sys.prefix` と `sys.exec_prefix` を使用します。空の前部は省略されます。後部には、まず空文字列を使い、次に `lib/site-packages` (Windows) または `lib/pythonX.Y/site-packages`, そして `lib/site-python` (Unix と Macintosh) を使います。前部-後部の組み合わせのそれぞれに対して、それが存在するディレクトリを参照しているかどうかを調べ、もしそうならば `sys.path` へ追加します。そして、パス設定ファイルを新しく追加されたパスからも検索します。

パス設定ファイルは `name.pth` という形式の名前をもつファイルで、上の 4 つのディレクトリのひとつにあります。その内容は `sys.path` に追加される追加項目 (一行に一つ) です。存在しない項目は `sys.path` へは決して追加されませんが、項目がファイルではなくディレクトリを参照しているかどうかはチェックされません。項目が `sys.path` へ二回以上追加されることはありません。空行と `#` で始まる行は読み飛ばされます。 `import` で始まる (そしてその後ろにスペースかタブが続く) 行は実行されます。

バージョン 2.6 で変更: `import` キーワードの後ろにスペースかタブが必要になりました。

例えば、`sys.prefix` と `sys.exec_prefix` が `/usr/local` に設定されていると仮定します。そのとき Python X.Y ライブラリは `/usr/local/lib/pythonX.Y` にインストールされています。ここにはサブディレクトリ `/usr/local/lib/pythonX.Y/site-packages` があり、その中に三つのサブディレクトリ `foo`, `bar` および `spam` と二つのパス設定ファイル `foo.pth` と `bar.pth` をもつと仮定します。`foo.pth` には以下のものが記載されていると想定してください:

```
# foo package configuration

foo
bar
bletch
```

また、`bar.pth` には:

```
# bar package configuration

bar
```

が記載されているとします。そのとき、次のバージョンごとのディレクトリが `sys.path` へこの順番で追加されます:

```
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

`bletch` は存在しないため省略されるということに注意してください。 `bar` ディレクトリは `foo` ディレクトリの前に来ます。なぜなら、`bar.pth` がアルファベット順で `foo.pth` の前に来るからです。また、`spam` はどちらのパス設定ファイルにも記載されていないため、省略されます。

これらのパス操作の後に、`sitecustomize` という名前のモジュールをインポートしようします。そのモジュールは任意のサイト固有のカスタマイゼーションを行うことができます。典型的にはこれはシステム管理者によって `site-packages` ディレクトリに作成されます。`ImportError` 例外が発生してこのインポートに失敗した場合は、何も表示せずに無視されます。Windows での `pythonw.exe` (IDLE を開始するとデフォルトで使われます) のような、Python が出力ストリームが利用出来ない状態で開始された場合、`sitecustomize` から試みられた出力は無視されます。`ImportError` 以外のあらゆる例外は黙殺され、そしてそれはおそらく不可思議な失敗にみえるでしょう。

このあとで、`ENABLE_USER_SITE` が真であれば、任意のユーザ固有のカスタマイズを行うことが出来る `usercustomize` と名付けられたモジュールのインポートが試みられます。このファイルはユーザの `site-packages` ディレクトリ (下記参照) に作られることを意図していて、その場所はオプション `-s` によって無効にされない限りは `sys.path` に含まれます。`ImportError` は黙って無視されます。

いくつかの非 Unix システムでは、`sys.prefix` と `sys.exec_prefix` は空で、パス操作は省略されます。しかし、`sitecustomize` と `usercustomize` のインポートはそのときでも試みられます。

`site.PREFIXES`

site パッケージディレクトリの `prefix` のリスト。

バージョン 2.6 で追加.

`site.ENABLE_USER_SITE`

ユーザーサイトディレクトリのステータスを示すフラグ。 `True` の場合、ユーザーサイトディレクトリが有効で `sys.path` に追加されていることを意味しています。 `False` の場合、ユーザによるリクエスト (オプション `-s` か `PYTHONNOUSERSITE`) によって、`None` の場合セキュリティ上の理由 (ユーザまたはグループ ID と実効 ID の間のミスマッチ) あるいは管理者によって、ユーザーサイトディレクトリが無効になっていることを示しています。ユーザーサイトディレクトリのステータスを示すフラグ。 `True` の場合、ユーザーサイトディレクトリが有効で `sys.path` に追加されていることを意味しています。 `False` の場合、ユーザによるリクエスト (オプション `-s` か `PYTHONNOUSERSITE`) によって、`None` の場合セキュリティ上の理由 (ユーザまたはグループ ID と実効 ID の間のミスマッチ) あるいは管理者によって、ユーザーサイトディレクトリが無効になっていることを示しています。

バージョン 2.6 で追加.

site.USER_SITE

Python 実行時のユーザの site-packages へのパスです。 `getusersitepackages()` がまだ呼び出されていないければ `None` かもしれません。デフォルト値は UNIX と framework なしの Mac OS X ビルドでは `~/.local/lib/pythonX.Y/site-packages`、Mac framework ビルドでは `~/Library/Python/X.Y/lib/python/site-packages`、Windows では `%APPDATA%\Python\PythonXY\site-packages` です。このディレクトリは site ディレクトリなので、ここにいる `.pth` ファイルが処理されます。

バージョン 2.6 で追加。

site.USER_BASE

ユーザの site-packages のベースとなるディレクトリへのパスです。 `getuserbase()` がまだ呼び出されていないければ `None` かもしれません。デフォルト値は UNIX と framework なしの Mac OS X ビルドでは `~/.local`、Mac framework ビルドでは `~/Library/Python/X.Y`、Windows では `%APPDATA%\Python` です。この値は Distutils が、スクリプト、データファイル、Python モジュールなどのインストール先のディレクトリを user installation scheme で計算するのに使われます。PYTHONUSERBASE も参照してください。

バージョン 2.6 で追加。

site.addsitedir(sitedir, known_paths=None)

`sys.path` にディレクトリを追加し、その `.pth` ファイル群を処理します。典型的には `sitecustomize` か `usercustomize` 内で使われます (上述)。

site.getsitepackages()

全てのグローバルな site-packages ディレクトリ (と、おそらく site-python も) のリストを返します。

バージョン 2.7 で追加。

site.getuserbase()

ユーザのベースディレクトリへのパス `USER_BASE` を返します。未初期化であればこの関数は `PYTHONUSERBASE` を参考にして、設定もします。

バージョン 2.7 で追加。

site.getusersitepackages()

ユーザのベースディレクトリへのパス `USER_SITE` を返します。未初期化であればこの関数は `PYTHONNOUSERSITE` と `USER_BASE` を参考にして、設定もします。

バージョン 2.7 で追加。

`site` モジュールはユーザディレクトリをコマンドラインから得る手段も提供しています:

```
$ python -m site --user-site
/home/user/.local/lib/python2.7/site-packages
```

引数なしで呼び出された場合、`sys.path` の中身を表示し、続けて `USER_BASE` とそのディレクトリが存在するかどうか、`USER_SITE` とそのディレクトリが存在するかどうか、最後に `ENABLE_USER_SITE` の値を、標準出力に出力します。

--user-base

ユーザのベースディレクトリを表示します。

--user-site

ユーザの site-packages ディレクトリを表示します。

両方のオプションが指定された場合、ユーザのベースとユーザの site が (常にこの順序で) `os.pathsep` 区切りで表示されます。

いずれかのオプションが与えられた場合に、このスクリプトは次のいずれかの終了コードで終了します: ユーザの site-packages が有効ならば 0、ユーザにより無効にされていれば 1、セキュリティ的な理由あるいは管理者によって無効にされている場合 2、そして何かエラーがあった場合は 2 より大きな値。

参考:

PEP 370 – Per user site-packages directory

28.15 user — ユーザー設定のフック

バージョン 2.6 で非推奨: `user` モジュールは Python 3 で削除されました。

ポリシーとして、Python は起動時にユーザー毎の設定を行うコードを実行することはしません (ただし対話型セッションで環境変数 `PYTHONSTARTUP` が設定されていた場合にはそのスクリプトを実行します)。

しかしながら、プログラムやサイトによっては、プログラムが要求した時にユーザーごとの標準設定ファイルを実行できると便利なこともあります。このモジュールはそのような機構を実装しています。この機構を利用したいプログラムでは、以下の文を実行してください:

```
import user
```

`user` モジュールはユーザーのホームディレクトリの `.pythonrc.py` ファイルを探し、オープンできるなら (`user` モジュールの) グローバル名前空間で実行します (`execfile()` を利用します)。この段階で発生したエラーは catch されません。 `user` モジュールを import したプログラムに伝播します。ホームディレクトリは環境変数 `HOME` が仮定されていますが、もし設定されていなければカレントディレクトリが使われます。

ユーザーの `.pythonrc.py` では Python のバージョンに従って異なる動作を行うために `sys.version` のテストを行うことが考えられます。

ユーザーへの警告: `.pythonrc.py` ファイルに書く内容には慎重になってください。どのプログラムが利用しているかわからない状況で、標準のモジュールや関数の振る舞いを変えることはおすすめでできません。

この機構を使おうとするプログラマへの提案: あなたのパッケージ向けのオプションをユーザーが設定できるようにするシンプルな方法は、 `.pythonrc.py` ファイルで変数を定義して、あなたのプログラムでテストする方法です。たとえば、 `spam` モジュールでメッセージ出力のレベルを変える `user.spam_verbose` 変数を参照するには以下のようにします:

```
import user

verbose = bool(getattr(user, "spam_verbose", 0))
```

(ユーザが `spam_verbose` をファイル `.pythonrc.py` 内で定義していないことに備えて `getattr()` の 3 引数形式を使っています。)

大掛かりな設定の必要があるプログラムでは、プログラム固有の設定ファイルを読む方がずっと良いです。

セキュリティやプライバシーに配慮するプログラムではこのモジュールを `import` しないでください。このモジュールを使うと、ユーザーは `.pythonrc.py` に任意のコードを書くことで簡単にプログラムに侵入することができてしまいます。

汎用のモジュールではこのモジュールを `import` しないでください。 `import` したプログラムの動作にも干渉してしまいます。

参考:

`site` モジュール サイト毎のカスタマイズを行う機構

28.16 `fpectl` — 浮動小数点例外の制御

注釈: `fpectl` モジュールはデフォルトではビルドされません。このモジュールの利用は推奨されておらず、熟練者以外がこのモジュールを使うのは危険です。このモジュールの制限についての詳細は、[制限と他に考慮すべきこと](#) 節を参照してください。

ほとんどのコンピュータはいわゆる IEEE-754 標準に準拠した浮動小数点演算を実行します。実際のどんなコンピュータでも、浮動小数点演算が普通の浮動小数点数では表せない結果になることがあります。例えば、次を試してください

```
>>> import math
>>> math.exp(1000)
inf
>>> math.exp(1000) / math.exp(1000)
nan
```

(上の例は多くのプラットフォームで動作します。DEC Alpha は例外かもしれません。) ”Inf”は”infinity(無限)”を意味する IEEE-754 における特殊な非数値の値で、”nan”は”not a number(数ではない)”を意味します。ここで留意すべき点は、その計算を行うように Python に求めたときに非数値の結果以外に特別なことは何も起きないということです。事実、それは IEEE-754 標準に規定されたデフォルトのふるまいで、それで良ければここで読むのを止めてください。

いくつかの環境では、誤った演算がなされたところで例外を発生し、処理を止めることがより良いでしょう。`fpectl` モジュールはそんな状況で使うためのものです。いくつかのハードウェア製造メーカーの浮動小数点ユニットを制御できるようにします。つまり、IEEE-754 例外 Division by Zero、Overflow あるいは Invalid Operation が起きたときはいつでも SIGFPE が生成させるように、ユーザが切り替えられるようにします。あなたの python システムを構成している C コードの中へ挿入される一組のラッパーマクロと協力して、SIGFPE は捕捉され、Python `FloatingPointError` 例外へ変換されます。

`fpectl` モジュールは次の関数を定義しています。また、所定の例外を発生します:

`fpectl.turnon_sigfpe()`

SIGFPE を生成するように切り替え、適切なシグナルハンドラを設定します。

`fpectl.turnoff_sigfpe()`

浮動小数点例外のデフォルトの処理に再設定します。

exception `fpectl.FloatingPointError`

`turnon_sigfpe()` が実行された後に、IEEE-754 例外である Division by Zero、Overflow または Invalid operation の一つを発生する浮動小数点演算は、次にこの標準 Python 例外を発生します。

28.16.1 例

以下の例は `fpectl` モジュールの使用を開始する方法とモジュールのテスト演算について示しています。

```
>>> import fpectl
>>> import fpetest
>>> fpectl.turnon_sigfpe()
>>> fpetest.test()
overflow          PASS
FloatingPointError: Overflow

div by 0          PASS
FloatingPointError: Division by zero
[ more output from test elided ]
>>> import math
>>> math.exp(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FloatingPointError: in math_1
```

28.16.2 制限と他に考慮すべきこと

特定のプロセッサを IEEE-754 浮動小数点エラーを捕らえるように設定することは、現在アーキテクチャごとの基準に基づきカスタムコードを必要とします。あなたの特殊なハードウェアを制御するために `fpectl` を修正することもできます。

IEEE-754 例外の Python 例外への変換には、ラッパーマクロ `PyFPE_START_PROTECT` と `PyFPE_END_PROTECT` があなたのコードに適切な方法で挿入されていることが必要です。Python 自身は `fpectl` モジュールをサポートするために修正されていますが、数値解析にとって興味ある多くの他のコードはそうではありません。

`fpectl` モジュールはスレッドセーフではありません。

参考:

このモジュールがどのように動作するのかについてより学習するときに、ソースディストリビューションの中のいくつかのファイルは興味を引くものでしょう。インクルードファイル `Include/pyfpe.h` では、この

モジュールの実装について同じ長さで議論されています。 `Modules/fpetestmodule.c` には、いくつかの使い方の例があります。多くの追加の例が `Objects/floatobject.c` にあります。

第 29 章

カスタム Python インタプリタ

この章で解説されるモジュールで Python の対話インタプリタに似たインタフェースを書くことができます。もし Python そのもの以外に何か特殊な機能をサポートした Python インタプリタを作りたければ、`code` モジュールを参照してください。(`codeop` モジュールはより低レベルで、不完全 (かもしれない) Python コード断片のコンパイルをサポートするために使われます。)

この章で解説されるモジュールの完全な一覧は:

29.1 `code` — インタプリタ基底クラス

`code` モジュールは read-eval-print (読み込み-評価-表示) ループを Python で実装するための機能を提供します。対話的なインタプリタプロンプトを提供するアプリケーションを作るために使える二つのクラスと便利な関数が含まれています。

```
class code.InteractiveInterpreter ([locals])
```

このクラスは構文解析とインタプリタ状態 (ユーザの名前空間) を取り扱います。入力バッファリングやプロンプト出力、または入力ファイル指定を扱いません (ファイル名は常に明示的に渡されます)。オプションの `locals` 引数はその中でコードが実行される辞書を指定します。その初期値は、キー `'__name__'` が `'__console__'` に設定され、キー `'__doc__'` が `None` に設定された新しく作られた辞書です。

```
class code.InteractiveConsole ([locals[, filename]])
```

対話的な Python インタプリタの振る舞いを厳密にエミュレートします。このクラスは `InteractiveInterpreter` を元に作られていて、通常の `sys.ps1` と `sys.ps2` をつけたプロンプト出力と入力バッファリングが追加されています。

```
code.interact ([banner[, readfunc[, local]]])
```

read-eval-print ループを実行するための便利な関数。これは `InteractiveConsole` の新しいインスタンスを作り、`readfunc` が与えられた場合は `InteractiveConsole.raw_input()` メソッドとして使われるように設定します。`local` が与えられた場合は、インタプリタループのデフォルト名前空間として使うために `InteractiveConsole` コンストラクタへ渡されます。そして、インスタンスの `interact()` メソッドは見出しとして使うために渡される `banner` を受け取り実行されます。コンソールオブジェクトは使われた後捨てられます。

```
code.compile_command(source[, filename[, symbol]])
```

この関数は Python のインタプリタメインループ (別名、read-eval-print ループ) をエミュレートしようとするプログラムにとって役に立ちます。扱いにくい部分は、ユーザが (完全なコマンドや構文エラーではなく) さらにテキストを入力すれば完全になりうる不完全なコマンドを入力したときを決定することです。この関数はほとんどの場合に実際のインタプリタメインループと同じ決定を行います。

source はソース文字列です。*filename* はオプションのソースが読み出されたファイル名で、デフォルトで '<input>' です。*symbol* はオプションの文法の開始記号で、'single' (デフォルト) または 'eval' のどちらかにすべきです。

コマンドが完全で有効ならば、コードオブジェクトを返します (`compile(source, filename, symbol)` と同じ)。コマンドが完全でないならば、`None` を返します。コマンドが完全で構文エラーを含む場合は、`SyntaxError` を発生させます。または、コマンドが無効なりテラルを含む場合は、`OverflowError` もしくは `ValueError` を発生させます。

29.1.1 対話的なインタプリタオブジェクト

```
InteractiveInterpreter.runsource(source[, filename[, symbol]])
```

インタプリタ内のあるソースをコンパイルし実行します。引数は `compile_command()` のものと同じです。*filename* のデフォルトは '<input>' で、*symbol* は 'single' です。あるいはいくつかのことが起きる可能性があります:

- 入力が正しくない。`compile_command()` が例外 (`SyntaxError` か `OverflowError`) を起こした場合。`showsyntaxerror()` メソッドの呼び出によって、構文トレースバックが表示されるでしょう。`runsource()` は `False` を返します。
- 入力が完全でなく、さらに入力が必要。`compile_command()` が `None` を返した場合。`runsource()` は `True` を返します。
- 入力が完全。`compile_command()` がコードオブジェクトを返した場合。(`SystemExit` を除く実行時例外も処理する) `runcode()` を呼び出すことによって、コードは実行されます。`runsource()` は `False` を返します。

次の行を要求するために `sys.ps1` か `sys.ps2` のどちらを使うかを決定するために、戻り値を利用できます。

```
InteractiveInterpreter.runcode(code)
```

コードオブジェクトを実行します。例外が生じたときは、トレースバックを表示するために `showtraceback()` が呼び出されます。伝わることが許されている `SystemExit` を除くすべての例外が捉えられます。

`KeyboardInterrupt` についての注意。このコードの他の場所でこの例外が生じる可能性がありますし、常に捕らえることができるとは限りません。呼び出し側はそれを処理するために準備しておくべきです。

```
InteractiveInterpreter.showsyntaxerror([filename])
```

起きたばかりの構文エラーを表示します。複数の構文エラーに対して一つあるのではないため、これは

スタックトレースを表示しません。 *filename* が与えられた場合は、Python のパーサが与えるデフォルトのファイル名の代わりに例外の中へ入れられます。なぜなら、文字列から読み込んでいるときはパーサは常に '`<string>`' を使うからです。出力は `write()` メソッドによって書き込まれます。

`InteractiveInterpreter.showtraceback()`

起きたばかりの例外を表示します。スタックの最初の項目を取り除きます。なぜなら、それはインタプリタオブジェクトの実装の内部にあるからです。出力は `write()` メソッドによって書き込まれます。

`InteractiveInterpreter.write(data)`

文字列を標準エラー스트リーム (`sys.stderr`) へ書き込みます。必要に応じて適切な出力処理を提供するために、派生クラスはこれをオーバーライドすべきです。

29.1.2 対話的なコンソールオブジェクト

`InteractiveConsole` クラスは `InteractiveInterpreter` のサブクラスです。以下の追加メソッドだけでなく、インタプリタオブジェクトのすべてのメソッドも提供します。

`InteractiveConsole.interact([banner])`

対話的な Python コンソールをそっくりにエミュレートします。オプションの `banner` 引数は最初のやりとりの前に表示するバナーを指定します。デフォルトでは、標準 Python インタプリタが表示するものと同じようなバナーを表示します。それに続けて、実際のインタプリタと混乱しないように (とても似ているから!) 括弧の中にコンソールオブジェクトのクラス名を表示します。

`InteractiveConsole.push(line)`

ソーステキストの一行をインタプリタへ送ります。その行の末尾に改行がついてはいけません。内部に改行を持っているかもしれません。その行はバッファへ追加され、ソースとして連結された内容が渡されインタプリタの `runsource()` メソッドが呼び出されます。コマンドが実行されたか、有効であることをこれが示している場合は、バッファはリセットされます。そうでなければ、コマンドが不完全で、その行が付加された後のままバッファは残されます。さらに入力が必要ならば、戻り値は `True` です。その行がある方法で処理されたならば、`False` です (これは `runsource()` と同じです)。

`InteractiveConsole.resetbuffer()`

入力バッファから処理されていないソーステキストを取り除きます。

`InteractiveConsole.raw_input([prompt])`

プロンプトを書き込み、一行を読み込みます。返る行は末尾に改行を含みません。ユーザが EOF キーシーケンスを入力したときは、`EOFError` を発生させます。基本実装では、組み込み関数 `raw_input()` を使って読み込みます。サブクラスはこれを異なる実装と置き換えるかもしれません。

29.2 codeop — Python コードをコンパイルする

`codeop` モジュールは、`code` モジュールで行われているような Python の read-eval-print ループをエミュレートするユーティリティを提供します。そのため、このモジュールを直接利用する場面はあまり無いでしょう。プログラムにこのようなループを含めたい場合は、`code` モジュールの方が便利です。

この仕事には二つの部分があります:

1. 入力の一行が Python の文として完全であるかどうかを見分けられること: 簡単に言えば、次が `'>>>'` か、あるいは `'...'` かどうかを見分けます。
2. どの future 文をユーザが入力したのかを覚えていること。したがって、実質的にそれに続く入力をこれらとともにコンパイルすることができます。

`codeop` モジュールはこうしたことのそれぞれを行う方法とそれら両方を行う方法を提供します。

前者は実行するには:

```
codeop.compile_command(source[, filename[, symbol]])
```

Python コードの文字列であるべき *source* をコンパイルしてみて、*source* が有効な Python コードの場合はコードオブジェクトを返します。このような場合、コードオブジェクトのファイル名属性は、デフォルトで `'<input>'` である *filename* でしょう。*source* が有効な Python コードではないが、有効な Python コードの接頭語である場合には、`None` を返します。

source に問題がある場合は、例外を発生させます。無効な Python 構文がある場合は、`SyntaxError` を発生させます。また、無効なリテラルがある場合は、`OverflowError` または `ValueError` を発生させます。

symbol 引数は *source* が文としてコンパイルされるか (`'single'`、デフォルト)、または `式` としてコンパイルされるかを決定します (`'eval'`)。他のどんな値も `ValueError` を発生させる原因となります。

注釈: ソースの終わりに達する前に、成功した結果をもってパーサは構文解析を止めることがあります。このような場合、後ろに続く記号はエラーとならずに無視されます。例えば、バックスラッシュの後ろに改行が 2 つあって、その後ろにゴミがあるかもしれません。パーサの API がより良くなればすぐに、この挙動は修正されるでしょう。

class `codeop.Compile`

このクラスのインスタンスは組み込み関数 `compile()` とシグネチャが一致する `__call__()` メソッドを持っていますが、インスタンスが `__future__` 文を含むプログラムテキストをコンパイルする場合は、インスタンスは有効なその文とともに続くすべてのプログラムテキストを覚えていてコンパイルするという違いがあります。

class `codeop.CommandCompiler`

このクラスのインスタンスは `compile_command()` とシグネチャが一致する `__call__()` メソッドを持っています。インスタンスが `__future__` 文を含むプログラムテキストをコンパイルする場合に、インスタンスは有効なその文とともにそれに続くすべてのプログラムテキストを覚えていてコンパイルするという違いがあります。

バージョン間の互換性についての注意: `Compile` と `CommandCompiler` は Python 2.2 で導入されました。2.2 の future-tracking 機能を有効にするだけでなく、2.1 と Python のより以前のバージョンとの互換性も保ちたい場合は、次のように書くことができます

```
try:
    from codeop import CommandCompiler
    compile_command = CommandCompiler()
    del CommandCompiler
except ImportError:
    from codeop import compile_command
```

これは影響の小さい変更ですが、あなたのプログラムにおそらく望まれないグローバル状態を導入します。または、次のように書くこともできます:

```
try:
    from codeop import CommandCompiler
except ImportError:
    def CommandCompiler():
        from codeop import compile_command
        return compile_command
```

そして、新たなコンパイラオブジェクトが必要となるたびに `CommandCompiler` を呼び出します。

第 30 章

制限実行 (restricted execution)

警告: Python 2.3 で、既知の容易に修正できないセキュリティホールのために、これらのモジュールは無効にされています。 `rexec` や `Bastion` モジュールを使った古いコードを読むときに助けになるよう、モジュールのドキュメントだけは残されています。

制限実行 (*restricted execution*) とは、信頼できるコードと信頼できないコードを区別できるようにするための Python における基本的なフレームワークです。このフレームワークは、信頼できる Python コード (スーパーバイザ (*supervisor*)) が、パーミッションに制限のかけられた ”拘束セル (padded cell)” を生成し、このセル中で信頼のおけないコードを実行するという概念に基づいています。信頼のおけないコードはこの拘束セルを破ることができず、信頼されたコードで提供され、管理されたインタフェースを介してのみ、傷つきやすいシステムリソースとやりとりすることができます。 ”制限実行” という用語は、 ”安全な Python (safe-Python)” を裏から支えるものです。というのは、真の安全を定義することは難しく、制限された環境を生成する方法によって決められるからです。制限された環境は入れ子にすることができ、このとき内側のセルはより縮小されることはあるが決して拡大されることのない特権を持ったサブセルを生成します。

Python の制限実行モデルの興味深い側面は、信頼されないコードに提供されるインタフェースが、信頼されるコードに提供されるそれらと同じ名前を持つということです。このため、制限された環境で動作するよう設計されたコードを書く上で特殊なインタフェースを学ぶ必要がありません。また、拘束セルの厳密な性質はスーパーバイザによって決められるため、アプリケーションによって異なる制限を課することができます。例えば、信頼されないコードが指定したディレクトリ内の何らかのファイルを読み出すが決して書き込まないということが ”安全” と考えられるかもしれません。この場合、スーパーバイザは組み込みの `open()` 関数について、`mode` パラメタが `'w'` の時に例外を送出するように再定義できます。また例えば、 ”安全” とは、`filename` パラメタに対して `chroot()` に似た操作を施して、ルートパスがファイルシステム上の何らかの安全な ”砂場 (sandbox)” 領域に対する相対パスになるようにすることもかもしれません。この場合でも、信頼されないコードは依然として、もとの呼び出しインタフェースを持ったままの組み込みの `open()` 関数を制限環境中に見出します。ここでは、関数に対する意味付け (semantics) は同じですが、許可されないパラメタが使われようとしているとスーパーバイザが判断した場合には `IOError` が送出されます。

Python のランタイムシステムは、特定のコードブロックが制限実行モードかどうかを、グローバル変数の中の `__builtins__` オブジェクトの一意性をもとに判断します: オブジェクトが標準の `__builtin__` モジュール (の辞書) の場合、コードは非制限下にあるとみなされます。それ以外は制限下にあるとみなされます。

制限実行モードで動作する Python コードは、拘束セルから侵出しないように設計された数多くの制限に直面します。例えば、関数オブジェクト属性 `func_globals` や、クラスおよびインスタンスオブジェクトの属性 `__dict__` は利用できません。

二つのモジュールが、制限実行環境を立ち上げるためのフレームワークを提供しています:

30.1 `rexec` — 制限実行のフレームワーク

バージョン 2.6 で非推奨: `rexec` モジュールは Python 3 で削除されました。

バージョン 2.3 で変更: モジュールは無効化され、使えなくなりました。

警告: このドキュメントは、`rexec` モジュールを使用している古いコードを読む際の参照用として残されています。

このモジュールには `RExec` クラスが含まれています。このクラスは、`r_eval()`、`r_execfile()`、`r_exec()` および `r_import()` メソッドをサポートし、これらは標準の Python 関数 `eval()`、`execfile()` および `exec` と `import` 文の制限されたバージョンです。この制限された環境で実行されるコードは、安全であると見なされたモジュールや関数だけにアクセスします; `RExec` をサブクラス化すれば、望むように能力を追加および削除できます。

警告: `rexec` モジュールは、下記のように動作するべく設計されていますが、注意深く書かれたコードなら利用できてしまうかもしれない、既知の脆弱性がいくつかあります。従って、“製品レベル”のセキュリティを要する状況では、`rexec` の動作をあてにするべきではありません。製品レベルのセキュリティを求めるなら、サブプロセスを介した実行や、あるいは処理するコードとデータの両方に対する非常に注意深い“浄化”が必要でしょう。上記の代わりに、`rexec` の既知の脆弱性に対するパッチ当ての手伝いも歓迎します。

注釈: `RExec` クラスは、プログラムコードによるディスクファイルの読み書きや TCP/IP ソケットの利用といった、安全でない操作の実行を防ぐことができます。しかし、プログラムコードによる非常に大量のメモリや処理時間の消費に対して防御することはできません。

```
class rexec.RExec([hooks[, verbose]])
```

`RExec` クラスのインスタンスを返します。

`hooks` は、`RHooks` クラスあるいはそのサブクラスのインスタンスです。`hooks` が省略されているか `None` であれば、デフォルトの `RHooks` クラスがインスタンス化されます。`rexec` モジュールが(組み込みモジュールを含む)あるモジュールを探したり、あるモジュールのコードを読んだりする時は常に、`rexec` がじかにファイルシステムに出て行くことはありません。その代わりに、あらかじめ `RHooks` クラスに渡しておいたり、コンストラクタで生成された `RHooks` インスタンスのメソッドを

呼び出します。(実際には、`RExec` オブジェクトはこれらと呼ばれません — 呼び出しは、`RExec` オブジェクトの一部であるモジュールローダオブジェクトによって行われます。これによって別のレベルの柔軟性が実現されます。この柔軟性は、制限された環境内で `import` 機構を変更する時に役に立ちます。)

代替の `RHooks` オブジェクトを提供することで、モジュールをインポートする際に行われるファイルシステムへのアクセスを制御することができます。このとき、各々のアクセスが行われる順番を制御する実際のアルゴリズムは変更されません。例えば、`RHooks` オブジェクトを置き換えて、ILU のようなある種の RPC メカニズムを介することで、全てのファイルシステムの要求をどこかにあるファイルサーバに渡すことができます。Grail のアプレットローダは、アプレットを URL からディレクトリ上に `import` する際にこの機構を使っています。

もし `verbose` が `true` であれば、追加のデバッグ出力が標準出力に送られます。

制限された環境で実行するコードも、やはり `sys.exit()` 関数を呼ぶことができることを知っておくことは大事なことです。制限されたコードがインタプリタから抜けだすことを許さないためには、いつでも、制限されたコードが、`SystemExit` 例外をキャッチする `try/except` 文とともに実行するように、呼び出しを防御します。制限された環境から `sys.exit()` 関数を除去するだけでは不十分です – 制限されたコードは、やはり `raise SystemExit` を使うことができてしまいます。`SystemExit` を取り除くことも、合理的なオプションではありません; いくつかのライブラリコードはこれを使っていますし、これが利用できなくなると中断してしまうでしょう。

参考:

[Grail Home Page](#) Grail はすべて Python で書かれた Web ブラウザです。これは、`rexec` モジュールを、Python アプレットをサポートするのに使っていて、このモジュールの使用例として使うことができます。

30.1.1 RExec オブジェクト

`RExec` インスタンスには以下のメソッドがあります:

`RExec.r_eval(code)`

`code` は、Python の式を含む文字列か、あるいはコンパイルされたコードオブジェクトのどちらかでなければなりません。そしてこれらは制限された環境の `__main__` モジュールで評価されます。式あるいはコードオブジェクトの値が返されます。

`RExec.r_exec(code)`

`code` は、1 行以上の Python コードを含む文字列か、コンパイルされたコードオブジェクトのどちらかでなければなりません。そしてこれらは、制限された環境の `__main__` モジュールで実行されます。

`RExec.r_execfile(filename)`

ファイル `filename` 内の Python コードを、制限された環境の `__main__` モジュールで実行します。

名前が `s_` で始まるメソッドは、`r_` で始まる関数と同様ですが、そのコードは、標準 I/O ストリーム `sys.stdin`、`sys.stderr` および `sys.stdout` の制限されたバージョンへのアクセスが許されています。

`RExec.s_eval(code)`

`code` は、Python 式を含む文字列でなければなりません。そして制限された環境で評価されます。

`RExec.s_exec(code)`

`code` は、1 行以上の Python コードを含む文字列でなければなりません。そして制限された環境で実行されます。

`RExec.s_execfile(code)`

ファイル `filename` に含まれた Python コードを制限された環境で実行します。

`RExec` オブジェクトは、制限された環境で実行されるコードによって暗黙のうちに呼ばれる、さまざまなメソッドもサポートしなければなりません。これらのメソッドをサブクラス内でオーバーライドすることによって、制限された環境が強制するポリシーを変更します。

`RExec.r_import(modulename[, globals[, locals[, fromlist]]])`

モジュール `modulename` をインポートし、もしそのモジュールが安全でないとみなされるなら、`ImportError` 例外が発生します。

`RExec.r_open(filename[, mode[, bufsize]])`

`open()` が制限された環境で呼ばれるとき、呼ばれるメソッドです。引数は `open()` のものと同じであり、ファイルオブジェクト (あるいはファイルオブジェクトと互換性のあるクラスインスタンス) が返されます。`RExec` のデフォルトの動作は、任意のファイルを読み取り用にオープンすることを許可しますが、ファイルに書き込もうとすることは許しません。より制限の少ない `r_open()` の実装については、以下の例を見て下さい。

`RExec.r_reload(module)`

モジュールオブジェクト `module` を再ロードして、それを再解析し再初期化します。

`RExec.r_unload(module)`

モジュールオブジェクト `module` をアンロードします (それを制限された環境の `sys.modules` 辞書から取りのぞきます)。

および制限された標準 I/O ストリームへのアクセスが可能な同等のもの:

`RExec.s_import(modulename[, globals[, locals[, fromlist]]])`

モジュール `modulename` をインポートし、もしそのモジュールが安全でないとみなされるなら、`ImportError` 例外が発生します。

`RExec.s_reload(module)`

モジュールオブジェクト `module` を再ロードして、それを再解析し再初期化します。

`RExec.s_unload(module)`

モジュールオブジェクト `module` をアンロードします。

30.1.2 制限された環境を定義する

`RExec` クラスには以下のクラス属性があります。それらは、`__init__()` メソッドが使います。それらを既存のインスタンス上で変更しても何の効果もありません; そうする代わりに、`RExec` のサブクラスを作成し

て、そのクラス定義でそれらに新しい値を割り当てます。そうすると、新しいクラスのインスタンスは、これらの新しい値を使用します。これらの属性のすべては、文字列のタプルです。

`RExec.nok_builtin_names`

制限された環境で実行するプログラムでは利用できないであろう、組み込み関数の名前を格納しています。`RExec` に対する値は、`('open', 'reload', '__import__')` です。(これは例外です。というのは、組み込み関数のほとんど大多数は無害だからです。この変数をオーバーライドしたいサブクラスは、基本クラスからの値から始めて、追加した許されない関数を連結していかなければなりません – 危険な関数が新しく Python に追加された時は、それらも、このモジュールに追加します。)

`RExec.ok_builtin_modules`

安全にインポートできる組み込みモジュールの名前を格納しています。`RExec` に対する値は、`('audioop', 'array', 'binascii', 'cmath', 'errno', 'imageop', 'marshal', 'math', 'md5', 'operator', 'parser', 'regex', 'select', 'sha', '_sre', 'strop', 'struct', 'time')` です。この変数をオーバーライドする場合も、同様な注意が適用されます – 基本クラスからの値を使って始めます。

`RExec.ok_path`

`import` が制限された環境で実行される時に検索されるディレクトリーを格納しています。`RExec` に対する値は、(モジュールがロードされた時は) 制限されないコードの `sys.path` と同一です。

`RExec.ok_posix_names`

制限された環境で実行するプログラムで利用できる、`os` モジュール内の関数の名前を格納しています。`RExec` に対する値は、`('error', 'fstat', 'listdir', 'lstat', 'readlink', 'stat', 'times', 'uname', 'getpid', 'getppid', 'getcwd', 'getuid', 'getgid', 'geteuid', 'getegid')` です。

`RExec.ok_sys_names`

制限された環境で実行するプログラムで利用できる、`sys` モジュール内の関数名と変数名を格納しています。`RExec` に対する値は、`('ps1', 'ps2', 'copyright', 'version', 'platform', 'exit', 'maxint')` です。

`RExec.ok_file.types`

モジュールがロードすることを許されているファイルタイプを格納しています。各ファイルタイプは、`imp` モジュールで定義された整数定数です。意味のある値は、`PY_SOURCE`、`PY_COMPILED` および `C_EXTENSION` です。`RExec` に対する値は、`(C_EXTENSION, PY_SOURCE)` です。サブクラスで `PY_COMPILED` を追加することは推奨されません; 攻撃者が、バイトコンパイルしたでっちあげのファイル (`.pyc`) を、例えば、あなたの公開 FTP サーバの `/tmp` に書いたり、`/incoming` にアップロードしたりして、とにかくあなたのファイルシステム内に置くことで、制限された実行モードから抜け出ることができるかもしれないからです。

30.1.3 例

標準の `RExec` クラスよりも、若干、もっと緩めたポリシーを望んでいるとしましょう。例えば、もし `/tmp` 内のファイルへの書き込みを喜んで許すならば、`RExec` クラスを次のようにサブクラス化できます:

```
class TmpWriterRExec(rexec.RExec):
    def r_open(self, file, mode='r', buf=-1):
        if mode in ('r', 'rb'):
            pass
        elif mode in ('w', 'wb', 'a', 'ab'):
            # check filename: must begin with /tmp/
            if file[:5] != '/tmp/':
                raise IOError("can't write outside /tmp")
            elif (string.find(file, '/../') >= 0 or
                  file[:3] == '../' or file[-3:] == '../'):
                raise IOError("'..' in filename forbidden")
            else: raise IOError("Illegal open() mode")
        return open(file, mode, buf)
```

上のコードは、完全に正しいファイル名でも、時には禁止する場合があることに注意して下さい; 例えば、制限された環境でのコードでは、`/tmp/foo/../bar` というファイルはオープンできないかもしれません。これを修正するには、`r_open()` メソッドが、そのファイル名を `/tmp/bar` に単純化しなければなりません。そのためには、ファイル名を分割して、それにさまざまな操作を行う必要があります。セキュリティが重大な場合には、より複雑で微妙なセキュリティホールを抱え込むかもしれない一般性のあるコードよりも、制限が余りにあり過ぎるとしても単純なコードを書く方が、望ましいでしょう。

30.2 Bastion — オブジェクトに対するアクセスの制限

バージョン 2.6 で非推奨: *Bastion* モジュールは Python 3 で削除されました。

バージョン 2.3 で変更: モジュールは無効化され、使えなくなりました。

注釈: このドキュメントは、Bastion モジュールを使用している古いコードを読む際の参照用として残されています。

辞書によると、バスチオン (bastion、要塞) とは、“防衛された領域や地点”、または“最後の砦と考えられているもの”であり、オブジェクトの特定の属性へのアクセスを禁じる方法を提供するこのモジュールにふさわしい名前です。制限モード下のプログラムに対して、あるオブジェクトにおける特定の安全な属性へのアクセスを許可し、かつその他の安全でない属性へのアクセスを拒否するには、要塞オブジェクトは常に *rexec* モジュールと共に使われなければなりません。

`Bastion.Bastion(object[, filter[, name[, class]]])`

オブジェクト *object* を保護し、オブジェクトに対する要塞オブジェクトを返します。オブジェクトの属性に対するアクセスの試みは全て、*filter* 関数によって認可されなければなりません; アクセスが拒否された場合 *AttributeError* 例外が送出されます。

filter が存在する場合、この関数は属性名を含む文字列を受理し、その属性に対するアクセスが許可される場合には真を返さなければなりません; *filter* が偽を返す場合、アクセスは拒否されます。標準のフィルタは、アンダースコア ('_') で始まる全ての関数に対するアクセスを拒否します。 *name* の値が与えられた場合、要塞オブジェクトの文字列表現は `<Bastion for name>` になります; そうでない場

合、`repr(object)` が使われます。

`class` が存在する場合、`BastionClass` のサブクラスでなくてはなりません; 詳細は `bastion.py` のコードを参照してください。 `BastionClass` の標準設定を上書きする必要はほとんどないはずです。

class `Bastion.BastionClass` (*getfunc*, *name*)

実際に要塞オブジェクトを実装しているクラスです。このクラスは `Bastion()` によって使われる標準のクラスです。 `getfunc` 引数は関数で、唯一の引数である属性の名前を与えて呼び出した際、制限された実行環境に対して、開示すべき属性の値を返します。 `name` は `BastionClass` インスタンスの `repr()` を構築するために使われます。

参考:

[Grail Home Page](#) Python で書かれたインターネットブラウザ Grail です。Python で書かれたアプレットをサポートするために、上記のモジュールを使っています。 Grail における Python 制限実行モードの利用に関する詳しい情報は、Web サイトで入手することができます。

第 31 章

モジュールのインポート

この章で解説されるモジュールは他の Python モジュールをインポートする新しい方法と、インポート処理をカスタマイズするためのフックを提供します。

この章で解説されるモジュールの完全な一覧は:

31.1 `imp` — `import` 内部へアクセスする

このモジュールは `import` 文を実装するために使われているメカニズムへのインターフェイスを提供します。次の定数と関数が定義されています:

`imp.get_magic()`

バイトコンパイルされたコードファイル (`.pyc` ファイル) を認識するために使われるマジック文字列値を返します。(この値は Python の各バージョンで異なります。)

`imp.get_suffixes()`

3 要素のタブルのリストを返します。それぞれのタブルは特定の種類のモジュールを説明しています。各タブルは `(suffix, mode, type)` という形式です。ここで、`suffix` は探すファイル名を作るためにモジュール名に追加する文字列です。そのファイルをオープンするために、`mode` は組み込み `open()` 関数へ渡されるモード文字列です (これはテキストファイルに対しては `'r'`、バイナリファイルに対しては `'rb'` となります)。`type` はファイル型で、以下で説明する値 `PY_SOURCE`、`PY_COMPILED` あるいは、`C_EXTENSION` の一つを取ります。

`imp.find_module(name[, path])`

モジュール `name` を見つけようとしています。 `path` が省略されるか `None` ならば、`sys.path` によって与えられるディレクトリ名のリストが検索されます。しかし、最初にいくつか特別な場所を検索します。まず、所定の名前をもつ組み込みモジュール (`C_BUILTIN`) を見つけようとしています。それから、フリーズされたモジュール (`PY_FROZEN`)、そしていくつかのシステムでは他の場所が同様に検索されます (Windows では、特定のファイルを指すレジストリの中を見ます)。

それ以外の場合、`path` はディレクトリ名のリストでなければなりません。上の `get_suffixes()` が返す拡張子のいずれかを伴ったファイルを各ディレクトリの中で検索します。リスト内の有効でない名前は黙って無視されます (しかし、すべてのリスト項目は文字列でなければなりません)。

検索が成功すれば、戻り値は3要素のタプル (`file`, `pathname`, `description`) です:

`file` は先頭に位置合わせされたオープンファイルオブジェクトで、`pathname` は見つかったファイルのパス名です。そして、`description` は `get_suffixes()` が返すリストに含まれているような3要素のタプルで、見つかったモジュールの種類を説明しています。

モジュールがファイルとして存在していなければ、返された `file` は `None` で、`pathname` は空文字列、`description` タプルはその拡張子とモードに対して空文字列を含みます。モジュール型は上の括弧の中に示されます。検索が失敗すれば、`ImportError` が発生します。他の例外は引数または環境に問題があることを示唆します。

モジュールがパッケージならば、`file` は `None` で、`pathname` はパッケージのパスで `description` タプルの最後の項目は `PKG_DIRECTORY` です。

この関数は階層的なモジュール名 (ドットを含む名前) を扱いません。 `P.M`、すなわちパッケージ `P` のサブモジュール `M` を見つけるためには、まず `find_module()` と `load_module()` を使用してパッケージ `P` を見つけてロードして、次に `P.__path__` を `path` 引数にして `find_module()` を呼び出してください。もし `P` 自体がドット付きの名前を持つ場合、このレシピを再帰的に適用してください。

`imp.load_module(name, file, pathname, description)`

`find_module()` を使って (あるいは、互換性のある結果を作り出す検索を行って) 以前見つけたモジュールをロードします。この関数はモジュールをインポートするという以上のことを行います: モジュールが既にインポートされているならば、`reload()` と同じです! `name` 引数は (これがパッケージのサブモジュールならばパッケージ名を含む) 完全なモジュール名を示します。`file` 引数はオープンしたファイルで、`pathname` は対応するファイル名です。モジュールがパッケージであるかファイルからロードされようとしていないとき、これらはそれぞれ `None` と `''` であっても構いません。`get_suffixes()` が返すように `description` 引数はタプルで、どの種類のモジュールがロードされなければならないかを説明するものです。

ロードが成功したならば、戻り値はモジュールオブジェクトです。そうでなければ、例外 (たいていは `ImportError`) が発生します。

重要: `file` 引数が `None` でなければ、例外が発生した場合でも呼び出し側にはそれを閉じる責任があります。これを行うには、`try ... finally` 文を使うことが最も良いです。

`imp.new_module(name)`

`name` という名前の新しい空モジュールオブジェクトを返します。このオブジェクトは `sys.modules` に挿入されません。

`imp.lock_held()`

現在インポートロックが維持されているならば、`True` を返します。そうでなければ、`False` を返します。スレッドのないプラットフォームでは、常に `False` を返します。

スレッドのあるプラットフォームでは、インポートが完了するまでインポートを実行するスレッドは内部ロックを維持します。このロックは元のインポートが完了するまで他のスレッドがインポートすることを阻止します。言い換えると、元のスレッドがそのインポート (および、もしあるならば、それによって引き起こされるインポート) の途中で構築した不完全なモジュールオブジェクトを、他のスレッドが見られないようにします。

`imp.acquire_lock()`

実行中のスレッドでインタープリタのインポートロックを取得します。インポートフックは、スレッドセーフのためにこのロックを取得しなければなりません。

一旦スレッドがインポートロックを取得したら、その同じスレッドはブロックされることなくそのロックを再度取得できます。スレッドはロックを取得するのと同じだけ解放しなければなりません。

スレッドのないプラットフォームではこの関数は何もしません。

バージョン 2.3 で追加。

`imp.release_lock()`

インタープリタのインポートロックを解放します。スレッドのないプラットフォームではこの関数は何もしません。

バージョン 2.3 で追加。

整数値をもつ次の定数はこのモジュールの中で定義されており、`find_module()` の検索結果を表すために使われます。

`imp.PY_SOURCE`

ソースファイルとしてモジュールが発見された。

`imp.PY_COMPILED`

コンパイルされたコードオブジェクトファイルとしてモジュールが発見された。

`imp.C_EXTENSION`

動的にロード可能な共有ライブラリとしてモジュールが発見された。

`imp.PKG_DIRECTORY`

パッケージディレクトリとしてモジュールが発見された。

`imp.C_BUILTIN`

モジュールが組み込みモジュールとして発見された。

`imp.PY_FROZEN`

モジュールがフリーズされたモジュールとして発見された (`init_frozen()` を参照)。

以下の定数と関数は旧バージョンのもので、`find_module()` や `load_module()` を使えば同様の機能を利用できます。これらは後方互換性のために残されています:

`imp.SEARCH_ERROR`

使われていません。

`imp.init_builtin(name)`

`name` という名前の組み込みモジュールを初期化し、そのモジュールオブジェクトを `sys.modules` に格納した上で返します。モジュールが既に初期化されている場合は、再度初期化されます。再初期化は組み込みモジュールの `__dict__` を `sys.modules` のエントリーに結びつけられたキャッシュモジュールからコピーする過程を含みます。 `name` という名前の組み込みモジュールがない場合は、`None` を返します。

`imp.init_frozen(name)`

`name` という名前のフリーズされたモジュールを初期化し、モジュールオブジェクトを返します。モジュールが既に初期化されている場合は、再度 初期化されます。`name` という名前のフリーズされたモジュールがない場合は、`None` を返します。(フリーズされたモジュールは Python で書かれたモジュールで、そのコンパイルされたバイトコードオブジェクトが Python の **freeze** ユーティリティを使ってカスタムビルド版の Python インタープリタへ組み込まれています。差し当たり、`Tools/freeze/` を参照してください。)

`imp.is_builtin(name)`

`name` という名前の再初期化できる組み込みモジュールがある場合は、`1` を返します。`name` という名前の再初期化できない組み込みモジュールがある場合は、`-1` を返します(`init_builtin()` を参照してください)。`name` という名前の組み込みモジュールがない場合は、`0` を返します。

`imp.is_frozen(name)`

`name` という名前のフリーズされたモジュール(`init_frozen()` を参照)がある場合は、`True` を返します。または、そのようなモジュールがない場合は、`False` を返します。

`imp.load_compiled(name, pathname[, file])`

バイトコンパイルされたコードファイルとして実装されているモジュールをロードして初期化し、そのモジュールオブジェクトを返します。モジュールが既に初期化されている場合は、再度 初期化されます。`name` 引数はモジュールオブジェクトを作ったり、アクセスするために使います。`pathname` 引数はバイトコンパイルされたコードファイルを指します。`file` 引数はバイトコンパイルされたコードファイルで、バイナリモードでオープンされ、先頭からアクセスされます。現在は、ユーザ定義のファイルをエミュレートするクラスではなく、実際のファイルオブジェクトでなければなりません。

`imp.load_dynamic(name, pathname[, file])`

動的ロード可能な共有ライブラリとして実装されているモジュールをロードして初期化します。モジュールが既に初期化されている場合は、再度 初期化します。再初期化はモジュールのキャッシュされたインスタンスの `__dict__` 属性を `sys.modules` にキャッシュされたモジュールの中で使われた値に上書きコピーする過程を含みます。`pathname` 引数は共有ライブラリを指していなければなりません。`name` 引数は初期化関数の名前を作るために使われます。共有ライブラリの `initname()` という名前の外部 C 関数が呼び出されます。オプションの `file` 引数は無視されます。(注意: 共有ライブラリはシステムに大きく依存します。また、すべてのシステムがサポートしているわけではありません。)

インポートの内部処理は拡張モジュールをファイル名で識別します。ですから `foo = load_dynamic("foo", "mod.so")` と `bar = load_dynamic("bar", "mod.so")` では `foo` と `bar` のどちらも同じモジュールを参照します。`mod.so` が `initbar` 関数を公開しているかどうかは無関係にです。シンボリックリンクをサポートするシステムでも、それをする、各々のモジュール参照が異なったファイル名を使って同じ共有ライブラリからの多重インポートになりえます。

`imp.load_source(name, pathname[, file])`

Python ソースファイルとして実装されているモジュールをロードして初期化し、モジュールオブジェクトを返します。モジュールが既に初期化されている場合は、再度 初期化します。`name` 引数はモジュールオブジェクトを作成したり、アクセスしたりするために使われます。`pathname` 引数はソースファイルを指します。`file` 引数はソースファイルで、テキストとして読み込むためにオープンされ、先頭からアクセスされます。現在は、ユーザ定義のファイルをエミュレートするクラスではなく、実際のファイルオブジェクトでなければなりません。(拡張子 `.pyc` または `.pyo` をもつ) 正しく対応するバ

イトコンパイルされたファイルが存在する場合は、与えられたソースファイルを構文解析する代わりにそれが使われることに注意してください。

class `imp.NullImporter` (*path_string*)

`NullImporter` 型は [PEP 302](#) インポートフックで、何もモジュールが見つからなかったときの非ディレクトリパス文字列を処理します。この型を既存のディレクトリや空文字列に対してコールすると `ImportError` が発生します。それ以外の場合は `NullImporter` のインスタンスが返されます。

Python は、ディレクトリでなく `sys.path_hooks` のどのパスフックでも処理されていないすべてのパスエントリに対して、この型のインスタンスを `sys.path_importer_cache` に追加します。このインスタンスが持つメソッドは次のひとつです。

find_module (*fullname* [, *path*])

このメソッドは常に `None` を返し、要求されたモジュールが見つからなかったことを表します。

バージョン 2.5 で追加。

31.1.1 例

次の関数は Python 1.4 までの標準 `import` 文 (階層的なモジュール名がない) をエミュレートします。(この実装はそのバージョンでは動作しないでしょう。なぜなら、`find_module()` は拡張されており、また `load_module()` が 1.4 で追加されているからです。)

```
import imp
import sys

def __import__(name, globals=None, locals=None, fromlist=None):
    # Fast path: see if the module has already been imported.
    try:
        return sys.modules[name]
    except KeyError:
        pass

    # If any of the following calls raises an exception,
    # there's a problem we can't handle -- let the caller handle it.

    fp, pathname, description = imp.find_module(name)

    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        # Since we may exit via an exception, close fp explicitly.
        if fp:
            fp.close()
```

階層的なモジュール名を実装し、`reload()` 関数を含むより完全な例はモジュール `knee` にあります。 `knee` モジュールは Python のソースディストリビューションの中の `Demo/imputil/` にあります。

31.2 importlib — `__import__()` の便利なラッパー

バージョン 2.7 で追加.

このモジュールは、Python 3.1 にある `import` の完全な実装を提供している同じ名前のパッケージの小さなサブセットです。このモジュールが提供しているものは、2.7 から 3.1 への移行をしやすいするためのものです。

`importlib.import_module(name, package=None)`

モジュールをインポートします。 `name` 引数は、インポートするモジュールを指定する絶対形式もしくは相対形式の名前です。(例: `pkg.mod` か `..mod`) `name` が相対形式で与えられた場合、`package` 引数にパッケージ名を解決する基準点となるパッケージを指定しなければなりません。(例: `import_module('..mod', 'pkg.subpkg')` は `pkg.mod` をインポートします) 指定されたモジュールは `sys.modules` に追加され、返されます。

31.3 importlib — Import ユーティリティ

バージョン 2.6 で非推奨: `importlib` モジュールは Python 3 で削除されました。

このモジュールは手軽で役に立つ `import` フックを改造する機構を提供します。古い `ihooks` と比較すると、`importlib` は `import` 関数を改造するのに劇的に単純で直截なアプローチをとります。

`class importlib.ImportManager([fs_imp])`

import 処理を管理します。

`install([namespace])`

この ImportManager を指定された名前空間にインストールします。

`uninstall()`

以前の import 機構に戻します。

`add_suffix(suffix, importFunc)`

内緒です。

`class importlib.Importer`

標準 import 関数を置き換える基底クラス。

`import_top(name)`

トップレベルのモジュールを import します。

`get_code(parent, modname, fqname)`

与えられたモジュールに対するコードを見つけて取ってきます。

`parent` は import するコンテキストを定義する親モジュールを指定します。 `None` でも構いません。その場合探すべきコンテキストは特にないことを意味します。

`modname` は親の内部の (ドットの付かない) 単独のモジュールを指定します。

fqname には完全修飾 (fully-qualified) モジュール名を指定します。これは (潜在的に) ドット付けられた名前、モジュール名前空間の "root" から *modname* までの名前です。

parent が無ければ、*modname*==*fqname* です。

このメソッドは `None` または 3 要素タプルを返します。

- モジュールが見つからなければ `None` が返されます。
- 2 または 3 要素のタプルの最初の要素は整数の 0 か 1 で、見つかったモジュールがパッケージかそうでないかを指定します。
- 2 番目の要素はモジュールのコードオブジェクトです (このコードは新しいモジュールの名前空間の中で実行されます)。この要素はまた完全に読み込まれたモジュールオブジェクト (たとえば共有ライブラリから読み込まれたものなど) でもありえます。
- 3 番目の要素はコードオブジェクトが実行される前に新しいモジュールに挿入する名前/値ペアの辞書です。この要素はモジュールのコードが特定の値 (たとえばどこでモジュールが見つかったかといった) を予期している場合に供給されます。2 番目の要素がモジュールオブジェクトであるときは、これらの名前/値はモジュールが読み込まれ/初期化された 後で 挿入されます。

class `importlib.BuiltinImporter`

ビルトインおよび凍結されたモジュール用の `import` 機構をエミュレートします。 `Importer` クラスのサブクラスです。

get_code (*parent*, *modname*, *fqname*)

内緒です。

`importlib.py_suffix_importer` (*filename*, *finfo*, *fqname*)

内緒です。

class `importlib.DynLoadSuffixImporter` (*[desc]*)

内緒です。

import_file (*filename*, *finfo*, *fqname*)

内緒です。

31.3.1 例

これは階層的モジュール `import` の再実装です。

このコードは読むためのもので、実行するためのものではありません。しかしながら、まあ動きます – これを有効にするのに必要なのは "import knee" することだけです。

(名前はこのモジュールの不格好な前身 "ni" との語呂合わせです)

```
import sys, imp, __builtin__

# Replacement for __import__()
```

(次のページに続く)

(前のページからの続き)

```

def import_hook(name, globals=None, locals=None, fromlist=None):
    parent = determine_parent(globals)
    q, tail = find_head_package(parent, name)
    m = load_tail(q, tail)
    if not fromlist:
        return q
    if hasattr(m, "__path__"):
        ensure_fromlist(m, fromlist)
    return m

def determine_parent(globals):
    if not globals or not globals.has_key("__name__"):
        return None
    pname = globals['__name__']
    if globals.has_key("__path__"):
        parent = sys.modules[pname]
        assert globals is parent.__dict__
        return parent
    if '.' in pname:
        i = pname.rfind('.')
        pname = pname[:i]
        parent = sys.modules[pname]
        assert parent.__name__ == pname
        return parent
    return None

def find_head_package(parent, name):
    if '.' in name:
        i = name.find('.')
        head = name[:i]
        tail = name[i+1:]
    else:
        head = name
        tail = ""
    if parent:
        qname = "%s.%s" % (parent.__name__, head)
    else:
        qname = head
    q = import_module(head, qname, parent)
    if q: return q, tail
    if parent:
        qname = head
        parent = None
        q = import_module(head, qname, parent)
        if q: return q, tail
    raise ImportError("No module named " + qname)

def load_tail(q, tail):
    m = q
    while tail:
        i = tail.find('.')

```

(次のページに続く)

(前のページからの続き)

```

    if i < 0: i = len(tail)
    head, tail = tail[:i], tail[i+1:]
    mname = "%s.%s" % (m.__name__, head)
    m = import_module(head, mname, m)
    if not m:
        raise ImportError("No module named " + mname)
    return m

def ensure_fromlist(m, fromlist, recursive=0):
    for sub in fromlist:
        if sub == "*":
            if not recursive:
                try:
                    all = m.__all__
                except AttributeError:
                    pass
            else:
                ensure_fromlist(m, all, 1)
            continue
        if sub != "*" and not hasattr(m, sub):
            subname = "%s.%s" % (m.__name__, sub)
            submod = import_module(sub, subname, m)
            if not submod:
                raise ImportError("No module named " + subname)

def import_module(partname, fqname, parent):
    try:
        return sys.modules[fqname]
    except KeyError:
        pass
    try:
        fp, pathname, stuff = imp.find_module(partname,
                                                parent and parent.__path__)
    except ImportError:
        return None
    try:
        m = imp.load_module(fqname, fp, pathname, stuff)
    finally:
        if fp: fp.close()
    if parent:
        setattr(parent, partname, m)
    return m

# Replacement for reload()
def reload_hook(module):
    name = module.__name__
    if '.' not in name:
        return import_module(name, name, None)
    i = name.rfind('.')
    pname = name[:i]

```

(次のページに続く)

(前のページからの続き)

```
parent = sys.modules[pname]
return import_module(name[i+1:], name, parent)

# Save the original hooks
original_import = __builtin__.__import__
original_reload = __builtin__.reload

# Now install our hooks
__builtin__.__import__ = import_hook
__builtin__.reload = reload_hook
```

`importers` モジュール (Python の配布されているソースの `Demo/imputil/` の中にあります) ももう一つの例として参照して下さい。

31.4 zipimport — Zip アーカイブからモジュールを import する

バージョン 2.3 で追加.

このモジュールは、Python モジュール (`*.py`, `*.py[co]`) やパッケージを ZIP 形式のアーカイブから import できるようにします。通常、`zipimport` を明示的に使う必要はありません; 組み込みの `import` は、`sys.path` の要素が ZIP アーカイブへのパスを指している場合にこのモジュールを自動的に使います。

普通、`sys.path` はディレクトリ名の文字列からなるリストです。このモジュールを使うと、`sys.path` の要素に ZIP ファイルアーカイブを示す文字列を使えるようになります。ZIP アーカイブにはサブディレクトリ構造を含めることができ、パッケージの `import` をサポートさせたり、アーカイブ内のパスを指定してサブディレクトリ下から `import` を行わせたりできます。例えば、`example.zip/lib/` のように指定すると、アーカイブ中の `lib/` サブディレクトリ下だけから `import` を行います。

ZIP アーカイブ内にはどんなファイルを置いてもかまいませんが、`import` できるのは `.py` および `.py[co]` だけです。動的モジュール (`.pyd`, `.so`) の ZIP import は行えません。アーカイブ内に `.py` ファイルしかない場合、Python は対応する `.pyc` や `.pyo` ファイルを追加してがアーカイブを変更しようとはしません。つまり、ZIP アーカイブ内に `.pyc` がない場合は、`import` がやや低速になるかもしれないので注意してください。

ZIP アーカイブからロードしたモジュールに対して組み込み関数 `reload()` を呼び出すと失敗します; `reload()` が必要になることはあまり考えられません。なぜってそれは実行時に ZIP ファイルが置き換えられてしまうことを暗に言っているわけですから。

アーカイブコメント付きの ZIP アーカイブは現在のところサポートされていません。

参考:

PKZIP Application Note ZIP ファイルフォーマットおよびアルゴリズムを作成した Phil Katz によるドキュメント。

PEP 273 - Zip アーカイブからモジュールをインポートする このモジュールの実装も行った、James C. Ahlstrom による PEP です。Python 2.3 は PEP 273 の仕様に従っていますが、Just van Rossum の書いた `import` フックによる実装を使っています。`import` フックは PEP 302 で解説されています。

PEP 302 - 新たなインポートフック このモジュールを動作させる助けになっている import フックの追加を提案している PEP です。

このモジュールでは例外を一つ定義しています:

exception `zipimport.ZipImportError`

`zipimporter` オブジェクトが送出する例外です。 `ImportError` のサブクラスなので、 `ImportError` としても捕捉できます。

31.4.1 zipimporter オブジェクト

`zipimporter` は ZIP ファイルを import するためのクラスです。

class `zipimport.zipimporter (archivepath)`

新たな `zipimporter` インスタンスを生成します。 `archivepath` は ZIP ファイルへのパスまたは ZIP ファイル中の特定のパスへのパスでなければなりません。たとえば、 `foo/bar.zip/lib` という `archivepath` の場合、 `foo/bar.zip` という ZIP ファイルの中の `lib` ディレクトリにあるモジュールを (存在するものとして) 検索します。

`archivepath` が有効な ZIP アーカイブを指していない場合、 `ZipImportError` を送出します。

find_module (`fullname` [, `path`])

`fullname` で指定されたモジュールを検索します。 `fullname` は完全に修飾された (ドット表記の) モジュール名でなければなりません。モジュールが見つかった場合には `zipimporter` インスタンス自体を返し、そうでない場合には `None` を返します。オプションの `path` 引数は無視されます — この引数は `importer` プロトコルとの互換性を保つためのものです。

get_code (`fullname`)

`fullname` に指定したモジュールのコードオブジェクトを返します。モジュールがない場合には `ZipImportError` を送出します。

get_data (`pathname`)

`pathname` に関連付けられたデータを返します。該当するファイルが見つからなかった場合には `IOError` を送出します。

get_filename (`fullname`)

指定されたモジュールが import された場合、そのモジュールに設定した `__file__` の値を返します。モジュールが見つからない場合、 `ZipImportError` を送出します。

バージョン 2.7 で追加.

get_source (`fullname`)

`fullname` で指定されたモジュールのソースコードを返します。モジュールが見つからない場合、 `ZipImportError` を送出します。アーカイブにはモジュールがあるもののソースコードがない場合、 `None` を返します。

is_package (`fullname`)

`fullname` で指定されたモジュールがパッケージの場合 `True` を返します。モジュールを見つから

れない場合 `ZipImportError` を送出します。

load_module (*fullname*)

fullname で指定されたモジュールをロードします。 *fullname* は完全修飾された (ドット表記の) モジュール名でなければなりません。 `import` 済みのモジュールを返します。モジュールがない場合には `ZipImportError` を送出します。

archive

importer に関連付けられた ZIP ファイルのファイル名です。サブパスは含まれません。

prefix

モジュールを検索する ZIP ファイル中のサブパスです。この文字列は ZIP ファイルのルートを指している `zipimporter` オブジェクトでは空です。

スラッシュでつなげると、*archive* と *prefix* 属性は `zipimporter` コンストラクタに渡された元々の *archivepath* 引数と等しくなります。

31.4.2 例

モジュールを ZIP アーカイブから `import` する例を以下に示します - `zipimport` モジュールが明示的に使われていないことに注意してください。

```
$ unzip -l example.zip
Archive:  example.zip
  Length      Date    Time    Name
-----
  8467      11-26-02  22:30    jwzthreading.py
-----
  8467                      1 file

$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, 'example.zip') # Add .zip file to front of path
>>> import jwzthreading
>>> jwzthreading.__file__
'example.zip/jwzthreading.py'
```

31.5 pkgutil — パッケージ拡張ユーティリティ

バージョン 2.3 で追加.

ソースコード: [Lib/pkgutil.py](#)

このモジュールはインポートシステムの、特にパッケージサポートに関するユーティリティです。

`pkgutil.extend_path` (*path*, *name*)

パッケージを構成するモジュールの検索パスを拡張します。パッケージの `__init__.py` で次のように

書くことを意図したものです:

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

上記はパッケージの `__path__` に `sys.path` の全ディレクトリのサブディレクトリとしてパッケージ名と同じ名前を追加します。これは 1 つの論理的なパッケージの異なる部品を複数のディレクトリに分けて配布したいときに役立ちます。

同時に `*.pkg` の `*` の部分が `name` 引数に指定された文字列に一致するファイルの検索もおこないます。この機能は `import` で始まる特別な行がないことを除き `*.pth` ファイルに似ています (*site* の項を参照)。 `*.pkg` は重複のチェックを除き、信頼できるものとして扱われます。 `*.pkg` ファイルの中に見つかったエントリはファイルシステム上に実在するか否かを問わず、そのまますべてパスに追加されます。(このような仕様です。)

入力パスがリストでない場合 (フリーズされたパッケージのとき) は何もせずにリターンします。入力パスが変更されていなければ、アイテムを末尾に追加しただけのコピーを返します。

`sys.path` はシーケンスであることが前提になっています。 `sys.path` の要素の内、実在するディレクトリを指す (Unicode または 8 ビットの) 文字列となっていないものは無視されます。ファイル名として使ったときにエラーが発生する `sys.path` の Unicode 要素がある場合、この関数 (`os.path.isdir()` を実行している行) で例外が発生する可能性があります。

class `pkgutil.ImpImporter` (*dirname=None*)

Python の "クラシック" インポートアルゴリズムをラップする **PEP 302** Importer.

dirname が文字列の場合、そのディレクトリを検索する **PEP 302** importer を作成します。 *dirname* が `None` のとき、現在の `sys.path` とフリーズされた、あるいはビルトインの全てのモジュールを検索する **PEP 302** importer を作成します。

`ImpImporter` は現在のところ `sys.meta_path` に配置しての利用をサポートしていないことに注意してください。

class `pkgutil.ImpLoader` (*fullname, file, filename, etc*)

Python の "クラシック" インポートアルゴリズムをラップする **PEP 302** Loader.

`pkgutil.find_loader` (*fullname*)

fullname に対する **PEP 302** "loader" オブジェクトを検索します。

fullname にドットが含まれる場合、パスはそれを包含しているパッケージの `__path__` のはずです。モジュールが見つからない場合や `import` できない場合は `None` を返します。この関数は `iter_importers()` を利用しているので、それと同じように、Windows レジストリなどのプラットフォーム依存の特別な `import` 場所に関する制限があります。

`pkgutil.get_importer` (*path_item*)

指定された *path_item* に対する **PEP 302** importer を取得します。

path hook により新しい importer が作成された場合は、それは `sys.path_importer_cache` にキャッシュされます。

importer が存在しない場合、基本 import 機構のラッパーを返します。このラッパーは importer cache にはキャッシュされません (代わりに None が insert されます)。

キャッシュ (やその一部) は、`sys.path_hooks` のリスキャンが必要になった場合は手動でクリアすることができます。

`pkgutil.get_loader(module_or_name)`

`module_or_name` に対する **PEP 302** "loader" を取得します。

module か package が通常の import 機構によってアクセスできる場合、その機構の該当部分に対するラッパーを返します。モジュールが見つからなかったり import できない場合は None を返します。その名前のモジュールがまだ import されていない場合、そのモジュールを含むパッケージが (あれば) そのパッケージの `__path__` を確立するために import されます。

この関数は `iter_importers()` を利用しているので、それと同じように、Windows レジストリなどのプラットフォーム依存の特別な import 場所に関する制限があります。

`pkgutil.iter_importers(fullname="")`

指定された名前に対する **PEP 302** importer を yield します (—訳注: 原文は the given module name になっていますが、(わかりやすいかもしれませんが) 厳密にはこれは間違い。fullname に cc や aa.bb.cc を渡した場合の cc はモジュールでもパッケージでもよく、いずれにせよそれを含む親パッケージの importer が返ります。—)。

fullname が '.' を含む場合、返されるのは fullname を包含するパッケージの importer になり (—訳注: fullname="aaa.bbb.ccc" の場合 "aaa.bbb" の importer。—)、それ以外の場合は `sys.meta_path`, `sys.path`, そして Python の "クラシック" import 機構の順のどれかの importer になります。その名前のついたモジュールがパッケージ内に含まれている場合、この関数を実行した副作用としてそのパッケージが import されます。

標準の import 機構がファイルを別の場所から探す、**PEP 302** 以外の機構 (例: Windows レジストリ) はある程度までサポートされていますが、それは `sys.path` の後に 検索されます。通常はそれらの場所は、`sys.path` のエントリに隠されないように `sys.path` の前に 検索されます。

モジュールやパッケージ名が `sys.path` と非 **PEP 302** のファイルシステム機構のどちらからもアクセスできると、このことは目に見える振る舞いの違いをもたらします。この場合、エミュレート時は前者から、ビルトインの import 機構は後者からモジュールやパッケージを見つけます。

この動作の食い違いにより、以下の種類の要素が影響を受けるかもしれません: `imp.C_EXTENSION`, `imp.PY_SOURCE`, `imp.PY_COMPILED`, `imp.PKG_DIRECTORY`。

`pkgutil.iter_modules(path=None, prefix="")`

`path` を指定すればその全てのサブモジュールに対して、`path` が None なら `sys.path` の全てのトップレベルモジュールに対して、`(module_loader, name, ispkg)` のタプルを yield します。

`path` は None か、モジュールを検索する `path` のリストのどちらかでなければなりません。

`prefix` は出力の全てのモジュール名の頭に出力する文字列です。

`pkgutil.walk_packages(path=None, prefix="", onerror=None)`

`path` を指定すれば再帰的にその中のモジュール全てに対して、`path` が None ならばアクセスできる全

でのモジュールに対して、`(module_loader, name, ispkg)` のタプルを yield します。

`path` は `None` か、モジュールを検索する `path` のリストのどちらかでなければなりません。

`prefix` は出力の全てのモジュール名の頭に出力する文字列です。

この関数は与えられた `path` 上の全ての パッケージ (全てのモジュール ではない) を、サブモジュールを検索するのに必要な `__path__` 属性にアクセスするために import します。

`onerror` は、パッケージを import しようとしたときに何かの例外が発生した場合に、1 つの引数 (import しようとしていたパッケージの名前) で呼び出される関数です。 `onerror` 関数が提供されない場合、`ImportError` は補足され無視されます。それ以外の全ての例外は伝播し、検索を停止させます。

例:

```
# list all modules python can access
walk_packages()

# list all submodules of ctypes
walk_packages(ctypes.__path__, ctypes.__name__ + '.')

```

`pkgutil.get_data(package, resource)`

パッケージからリソースを取得します。

この関数は **PEP 302** ロードの `get_data()` API のラッパーです。 `package` 引数は標準的なモジュール形式 (`foo.bar`) のパッケージ名でなければなりません。 `resource` 引数は `/` をパス区切りに使った相対ファイル名の形式です。親ディレクトリを `..` としたり、ルートからの (`/` で始まる) 名前を使うことはできません。

この関数が返すのは指定されたリソースの内容であるバイナリ文字列です。

ファイルシステム中に位置するパッケージで既にインポートされているものに対しては、次と大体同じです:

```
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()

```

パッケージを見出せなかったりロードできなかったり、あるいは `get_data()` をサポートしない **PEP 302** ロードを使ったりした場合は、`None` が返されます。

バージョン 2.6 で追加.

31.6 modulefinder — スクリプト中で使われているモジュールを検索する

バージョン 2.3 で追加.

ソースコード: [Lib/modulefinder.py](#)

このモジュールでは、スクリプト中で `import` されているモジュールセットを調べるために使える `ModuleFinder` クラスを提供しています。 `modulefinder.py` はまた、Python スクリプトのファイル名を引数に指定してスクリプトとして実行し、 `import` されているモジュールのレポートを出力させることもできます。

`modulefinder.AddPackagePath(pkg_name, path)`

`pkg_name` という名前のパッケージの在り処が `path` であることを記録します。

`modulefinder.ReplacePackage(oldname, newname)`

実際にはパッケージ内で `oldname` という名前になっているモジュールを `newname` という名前で指定できるようにします。この関数の主な用途は、`_xmlplus` パッケージが `xml` パッケージに置き換わっている場合の処理でしょう。

class `modulefinder.ModuleFinder` (`[path=None, debug=0, excludes=[], replace_paths=[]]`)

このクラスでは `run_script()` および `report()` メソッドを提供しています。これらのメソッドは何らかのスクリプト中で `import` されているモジュールの集合を調べます。 `path` はモジュールを検索する先のディレクトリ名からなるリストです。 `path` を指定しない場合、 `sys.path` を使います。 `debug` にはデバッグレベルを設定します; 値を大きくすると、実行している内容を表すデバッグメッセージを出力します。 `excludes` は検索から除外するモジュール名です。 `replace_paths` には、モジュールパス内で置き換えられるパスをタプル (`oldpath`, `newpath`) からなるリストで指定します。

report()

スクリプトで `import` しているモジュールと、そのパスからなるリストを列挙したレポートを標準出力に出力します。モジュールを見つけられなかったり、モジュールがないように見える場合にも報告します。

run_script(pathname)

`pathname` に指定したファイルの内容を解析します。ファイルには Python コードが入っていないければなりません。

modules

モジュール名をモジュールに結びつける辞書。 [ModuleFinder の使用例](#) を参照して下さい。

31.6.1 ModuleFinder の使用例

解析対象のスクリプトはこれ (`bacon.py`) です:

```
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```


bacon.py のレポートを出力するスクリプトです:

```
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print 'Loaded modules:'
for name, mod in finder.modules.iteritems():
    print '%s: ' % name,
    print ','.join(mod.globalnames.keys()[:3])

print '-'*50
print 'Modules not imported:'
print '\n'.join(finder.badmodules.iterkeys())
```

出力例です (アーキテクチャによって違って来るかもしれません):

```
Loaded modules:
_types:
copy_reg:  _inverted_registry, _slotnames, __all__
sre_compile:  isstring, _sre, _optimize_unicode
_sre:
sre_constants:  REPEAT_ONE, makedict, AT_END_LINE
sys:
re:  __module__, finditer, _expand
itertools:
__main__:  re, itertools, baconhammeggs
sre_parse:  __getslice__, _PATTERNENDERS, SRE_FLAG_UNICODE
array:
types:  __module__, IntType, TypeType
-----
Modules not imported:
guido.python.ham
baconhammeggs
```

31.7 runpy — Python モジュールの位置特定と実行

バージョン 2.5 で追加.

ソースコード: [Lib/runpy.py](#)

`runpy` モジュールは Python のモジュールをインポートせずにその位置を特定したり実行したりするのに使われます。その主な目的はファイルシステムではなく Python のモジュール名前空間を使って位置を特定したスクリプトの実行を可能にする `-m` コマンドラインスイッチを実装することです。

`runpy` モジュールは 2 つの関数を提供しています:

`runpy.run_module(mod_name, init_globals=None, run_name=None, alter_sys=False)`

指定されたモジュールのコードを実行し、実行後のモジュールグローバル辞書を返します。モジュール

のコードはまず標準インポート機構 (詳細は [PEP 302](#) を参照) を使ってモジュールの位置を特定され、まっさらなモジュール名前空間で実行されます。

指定されたモジュール名が通常のモジュールではなくパッケージを参照していた場合、そのパッケージが import された後その中の `__main__` モジュールが実行され、実行後のモジュールグローバル辞書を返します。

オプションの辞書型引数 `init_globals` はコードを実行する前にモジュールグローバル辞書に前もって必要な設定しておくのに使われます。与えられた辞書は変更されません。その辞書の中に以下に挙げる特別なグローバル変数が定義されていたとしても、それらの定義は `run_module()` 関数によってオーバーライドされます。

特別なグローバル変数 `__name__`, `__file__`, `__loader__`, `__package__` はモジュールコードが実行される前にグローバル辞書にセットされます (この変数群は修正される最小セットです。これ以外の変数もインタプリタの実装の詳細として暗黙的に設定されるかもしれません)。

`__name__` は、オプション引数 `run_name` が `None` でない場合、指定されたモジュールがパッケージであれば `mod_name + '__main__'` に、そうでなければ `mod_name` 引数の値がセットされます。

`__file__` はモジュールローダにより与えられた名前がセットされます。もしローダがファイル名情報を取得可能にしなければ、この変数の値は `None` になります。

`__loader__` はモジュールのコードを取得するのに使われる [PEP 302](#) のモジュールローダがセットされます (このローダは標準のインポート機構に対するラッパーかもしれません)。

`__package__` は指定されたモジュールがパッケージだった場合は `mod_name` に、それ以外の場合は `mod_name.rpartition('.')[0]` に設定されます。

引数 `alter_sys` が与えられて `True` に評価されるならば、`sys.argv[0]` は `__file__` の値で更新され `sys.modules[__name__]` は実行されるモジュールの一時的モジュールオブジェクトで更新されます。`sys.argv[0]` と `sys.modules[__name__]` はどちらも関数が処理を戻す前にもとの値に復旧します。

この `sys` に対する操作はスレッドセーフではないということに注意してください。他のスレッドは部分的に初期化されたモジュールを見たり、入れ替えられた引数リストを見たりするかもしれません。この関数をスレッド化されたコードから起動するときは `sys` モジュールには手を触れないことが推奨されます。

参考:

コマンドラインから、`-m` オプションを与えることで同じ機能を実現出来ます。

バージョン 2.7 で変更: `__main__` サブモジュールを検索することによってパッケージを実行する機能が追加されました。

`runpy.run_path(file_path, init_globals=None, run_name=None)`

指定されたファイルシステム上の場所にあるコードを実行して、結果としてそのモジュールグローバル辞書を返します。通常の CPython 実行時にコマンドラインで指定するスクリプト名と同じく、指定できるパスは Python ソースファイル、コンパイルされたバイトコードファイル、もしくは `__main__` モ

ジュールを含む有効な `sys.path` エントリ (例: トップレベルに `__main__.py` ファイルを持つ zip ファイル) です。

シンプルなスクリプトを指定した場合は、指定されたコードはシンプルに新しいモジュール名前空間で実行されます。有効な `sys.path` エントリ (一般的には zip ファイルかディレクトリ) を指定した場合は、最初にそのエントリが `sys.path` の先頭に追加されます。次に更新した `sys.path` を元に `__main__` モジュールを検索して実行します。指定された場所に `__main__` モジュールが無かった時、`sys.path` のどこか他のエントリに存在する別の `__main__` を実行してしまう可能性があることに注意してください。

オプションの辞書型引数 `init_globals` はコードを実行する前にモジュールグローバル辞書に前もって必要な設定しておくのに使われます。与えられた辞書は変更されません。その辞書の中に以下に挙げる特別なグローバル変数が定義されていたとしても、それらの定義は `run_path()` 関数によってオーバーライドされます。

特別なグローバル変数 `__name__`, `__file__`, `__loader__`, `__package__` はモジュールコードが実行される前にグローバル辞書にセットされます (この変数群は修正される最小セットです。これ以外の変数もインタプリタの実装の詳細として暗黙的に設定されるかもしれません)。

`__name__` は、オプション引数 `run_name` が `None` でない場合、`run_name` に設定され、それ以外の場合は '`<run_path>`' に設定されます。

`__file__` はモジュールローダにより与えられた名前がセットされます。もしローダがファイル名情報を取得可能にしなければ、この変数の値は `None` になります。シンプルなスクリプトの場合、これは `file_path` になるでしょう。

`__loader__` はモジュールのコードを取得するのに使われる [PEP 302](#) のモジュールローダがセットされます (このローダは標準のインポート機構に対するラッパーかもしれません)。シンプルなスクリプトの場合、これは `None` になるでしょう。

`__package__` は `__name__.rpartition('.')[0]` に設定されます。

`sys` モジュールに対していくつかの変更操作が行われます。まず、`sys.path` が上記のように修正されます。`sys.argv[0]` は `file_path` に更新され、`sys.modules[__name__]` は実行されるモジュールのための一時モジュールオブジェクトに更新されます。`sys` の要素に対する全ての変更は、この関数から戻る前に元に戻されます。

`run_module()` と違い、`sys` に対する変更はオプションではありません。これらの変更は `sys.path` エントリの実行に必要な不可欠だからです。スレッドセーフ性に関する制限はこの関数にも存在します。この関数をマルチスレッドプログラムから実行する場合は、`import lock` によりシリアライズして実行するか、別プロセスに委譲してください。

参考:

コマンドラインから `using-on-interface-options` で同じ機能を使えます (`python path/to/script`)。

バージョン 2.7 で追加。

参考:

[PEP 338](#) - モジュールをスクリプトとして実行する Nick Coghlan によって書かれ実装された PEP。

PEP 366 - main モジュールの明示的な相対インポート Nick Coghlan によって書かれ実装された PEP。

using-on-general - CPython コマンドライン詳細

第 32 章

Python 言語サービス

Python には Python 言語を使って作業するときに役に立つモジュールがたくさん提供されています。これらのモジュールはトークンの切り出し、パース、構文解析、バイトコードのディスアセンブリおよびその他のさまざまな機能をサポートしています。

これらのモジュールには、次のものが含まれています：

32.1 `parser` — Python 解析木にアクセスする

`parser` モジュールは Python の内部パーサとバイトコード・コンパイラへのインターフェイスを提供します。このインターフェイスの第一の目的は、Python コードから Python の式の解析木を編集したり、これから実行可能なコードを作成したりできるようにすることです。これは任意の Python コードの断片を文字列として構文解析や変更を行うより良い方法です。なぜなら、構文解析がアプリケーションを作成するコードと同じ方法で実行されるからです。その上、高速です。

注釈：Python 2.5 以降、抽象構文木 (AST) の生成・コンパイルの段階に割り込むには `ast` モジュールを使うのがずっとお手軽です。

`parser` モジュールはこの文書に書かれている名前の `st` を `ast` に置き換えたものも使えるようにしてあります。こうした名前は Python 2.5 で AST を扱い始める前の AST が他になかった頃から名残です。このことはまた関数の引数が `st` ではなく `ast` と呼ばれていることの理由でもあります。`ast` 系関数は Python 3 で削除されました。

このモジュールについて注意すべきことが少しあります。それは作成したデータ構造を利用するために重要なことです。この文書は Python コードの解析木を編集するためのチュートリアルではありませんが、`parser` モジュールを使った例をいくつか示しています。

もっとも重要なことは、内部パーサが処理する Python の文法についてよく理解しておく必要があるということです。言語の文法に関する完全な情報については、`reference-index` を参照してください。標準の Python ディストリビューションに含まれるファイル `Grammar/Grammar` の中で定義されている文法仕様から、パーサ自身は作成されています。このモジュールが作成する ST オブジェクトの中に格納される解析木は、下で説明する `expr()` または `suite()` 関数によって作られるときに内部パーサから実際に出力されるものです。

`sequence2st()` が作る ST オブジェクトは忠実にこれらの構造をシミュレートしています。言語の形式文法が改訂されるために、“正しい”と考えられるシーケンスの値が Python のあるバージョンから別のバージョンで変化することがあるということに注意してください。しかし、Python のあるバージョンから別のバージョンへテキストのソースのままコードを移せば、目的のバージョンで正しい解析木を常に作成できます。ただし、インタプリタの古いバージョンへ移行する際に、最近の言語コンストラクトをサポートしていないことがあるという制限だけがあります。ソースコードが常に前方互換性があるのに対して、一般的に解析木はあるバージョンから別のバージョンへの互換性がありません。

`st2list()` または `st2tuple()` から返されるシーケンスのそれぞれの要素は単純な形式です。文法の非終端要素を表すシーケンスは常に一より大きい長さを持ちます。最初の要素は文法の生成規則を識別する整数です。これらの整数は C ヘッドファイル `Include/graminit.h` と Python モジュール `symbol` 中の特定のシンボル名です。シーケンスに付け加えられている各要素は、入力文字列の中で認識されたままの形で生成規則の構成要素を表しています: これらは常に親と同じ形式を持つシーケンスです。この構造の注意すべき重要な側面は、`if_stmt` 中のキーワード `if` のような親ノードの型を識別するために使われるキーワードがいかなる特別な扱いもなくノードツリーに含まれているということです。例えば、`if` キーワードはタプル `(1, 'if')` と表されます。ここで、`1` は、ユーザが定義した変数名と関数名を含むすべての `NAME` トークンに対応する数値です。行番号情報が必要となときに返される別の形式では、同じトークンが `(1, 'if', 12)` のように表されます。ここでは、`12` が終端記号の見つかった行番号を表しています。

終端要素は同じ方法で表現されますが、子の要素や識別されたソーステキストの追加は全くありません。上記の `if` キーワードの例が代表的なものです。終端記号のいろいろな型は、C ヘッドファイル `Include/token.h` と Python モジュール `token` で定義されています。

ST オブジェクトはこのモジュールの機能をサポートするために必要ありませんが、三つの目的から提供されています: アプリケーションが複雑な解析木を処理するコストを償却するため、Python のリストやタプル表現に比べてメモリ空間を保全する解析木表現を提供するため、解析木を操作する追加モジュールを C で作ることを簡単にするため。ST オブジェクトを使っていることを隠すために、簡単な“ラッパー”クラスを Python で作ることができます。

`parser` モジュールは二、三の別々の目的のために関数を定義しています。もっとも重要な目的は ST オブジェクトを作ることと、ST オブジェクトを解析木とコンパイルされたコードオブジェクトのような他の表現に変換することです。しかし、ST オブジェクトで表現された解析木の型を調べるために役に立つ関数もあります。

参考:

`symbol` モジュール 解析木の内部ノードを表す便利な定数。

`token` モジュール 便利な解析木の葉のノードを表す定数とノード値をテストするための関数。

32.1.1 ST オブジェクトを作成する

ST オブジェクトはソースコードあるいは解析木から作られます。ST オブジェクトをソースから作るときは、`'eval'` と `'exec'` 形式を作成するために別々の関数が使われます。

```
parser.expr(source)
```

まるで `compile(source, 'file.py', 'eval')` への入力であるかのように、`expr()` 関数は

パラメータ *source* を構文解析します。解析が成功した場合は、ST オブジェクトは内部解析木表現を保持するために作成されます。そうでなければ、適切な例外を発生させます。

`parser.suite(source)`

まるで `compile(source, 'file.py', 'exec')` への入力であるかのように、`suite()` 関数はパラメータ *source* を構文解析します。解析が成功した場合は、ST オブジェクトは内部解析木表現を保持するために作成されます。そうでなければ、適切な例外を発生させます。

`parser.sequence2st(sequence)`

この関数はシーケンスとして表現された解析木を受け取り、可能ならば内部表現を作ります。木が Python の文法に合っていることと、すべてのノードが Python のホストバージョンで有効なノード型であることを確認した場合は、ST オブジェクトが内部表現から作成されて呼び出し側へ返されます。内部表現の作成に問題があるならば、あるいは木が正しいと確認できないならば、`ParserError` 例外を発生します。この方法で作られた ST オブジェクトが正しくコンパイルできると決めつけない方がよいでしょう。ST オブジェクトが `compilest()` へ渡されたとき、コンパイルによって送出された通常の例外がまだ発生するかもしれません。これは (`MemoryError` 例外のような) 構文に関係していない問題を示すのかもしれないし、`del f(0)` を解析した結果のようなコンストラクトが原因であるかもしれません。このようなコンストラクトは Python のパーサを逃れますが、バイトコードインタープリタによってチェックされます。

終端トークンを表すシーケンスは、`(1, 'name')` 形式の二つの要素のリストか、または `(1, 'name', 56)` 形式の三つの要素のリストです。三番目の要素が存在する場合は、有効な行番号だとみなされます。行番号が指定されるのは、入力木の終端記号の一部に対してです。

`parser.tuple2st(sequence)`

これは `sequence2st()` と同じ関数です。このエントリポイントは後方互換性のために維持されています。

32.1.2 ST オブジェクトを変換する

作成するために使われた入力に関係なく、ST オブジェクトはリスト木またはタプル木として表される解析木へ変換されるか、または実行可能なオブジェクトへコンパイルされます。解析木は行番号情報を持って、あるいは持たずに抽出されます。

`parser.st2list(ast[, line_info])`

この関数は呼び出し側から *ast* に ST オブジェクトを受け取り、解析木と等価な Python のリストを返します。結果のリスト表現はインスペクションあるいはリスト形式の新しい解析木の作成に使うことができます。リスト表現を作るためにメモリが利用できる限り、この関数は失敗しません。解析木がインスペクションのためだけにつかわれるならば、メモリの消費量と断片化を減らすために `st2tuple()` を代わりに使うべきです。リスト表現が必要とされるとき、この関数はタプル表現を取り出して入れ子のリストに変換するよりかなり高速です。

line_info が真ならば、トークンを表すリストの三番目の要素として行番号情報がすべての終端トークンに含まれます。与えられた行番号はトークン が終わる 行を指定していることに注意してください。フラグが偽または省略された場合は、この情報は省かれます。

`parser.st2tuple(ast[, line_info])`

この関数は呼び出し側から *ast* に ST オブジェクトを受け取り、解析木と等価な Python のタプルを返します。リストの代わりにタプルを返す以外は、この関数は `st2list()` と同じです。

line_info が真ならば、トークンを表すリストの三番目の要素として行番号情報がすべての終端トークンに含まれます。フラグが偽または省略された場合は、この情報は省かれます。

`parser.compilest(ast, filename='<syntax-tree>')`

`exec` 文の一部として使える、あるいは、組み込み `eval()` 関数への呼び出しとして使えるコードオブジェクトを生成するために、Python バイトコードコンパイラを ST オブジェクトに対して呼び出すことができます。この関数はコンパイラへのインターフェイスを提供し、*filename* パラメータで指定されるソースファイル名を使って、*ast* からパーサへ内部解析木を渡します。*filename* に与えられるデフォルト値は、ソースが ST オブジェクトだったことを示唆しています。

ST オブジェクトをコンパイルすることは、コンパイルに関する例外を引き起こすことになるかもしれませんが。例としては、`del f(0)` の解析木によって発生させられる `SyntaxError` があります: この文は Python の形式文法としては正しいと考えられますが、正しい言語コンストラクトではありません。この状況に対して発生する `SyntaxError` は、実際には Python バイトコンパイラによって通常作り出されます。これが `parser` モジュールがこの時点で例外を発生できる理由です。解析木のインスペクションを行うことで、コンパイルが失敗するほとんどの原因をプログラムによって診断することができます。

32.1.3 ST オブジェクトに対する問い合わせ

ST が式または suite として作成されたかどうかをアプリケーションが決定できるようにする二つの関数が提供されています。これらの関数のどちらも、ST が `expr()` または `suite()` を通してソースコードから作られたかどうか、あるいは、`sequence2st()` を通して解析木から作られたかどうかを決定できません。

`parser.isexpr(ast)`

ast が 'eval' 形式を表している場合に、この関数は真を返します。そうでなければ、偽を返します。これは役に立ちます。なぜならば、通常は既存の組み込み関数を使ってもコードオブジェクトに対してこの情報を問い合わせることができないからです。このどちらのようにも `compilest()` によって作成されたコードオブジェクトに問い合わせることはできませんし、そのコードオブジェクトは組み込み `compile()` 関数によって作成されたコードオブジェクトと同じであることに注意してください。

`parser.issuite(ast)`

ST オブジェクトが (通常 "suite" として知られる) 'exec' 形式を表しているかどうかを報告するという点で、この関数は `isexpr()` に酷似しています。追加の構文が将来サポートされるかもしれないので、この関数が `not isexpr(ast)` と等価であるとみなすのは安全ではありません。

32.1.4 例外とエラー処理

`parser` モジュールは例外を一つ定義していますが、Python ランタイム環境の他の部分が提供する別の組み込み例外を発生させることもあります。各関数が発生させる例外の情報については、それぞれ関数を参照してください。

exception `parser.ParserError`

`parser` モジュール内部で障害が起きたときに発生する例外。普通の構文解析中に発生する組み込みの `SyntaxError` ではなく、一般的に妥当性確認が失敗した場合に引き起こされます。例外の引数としては、障害の理由を説明する文字列である場合と、`sequence2st()` へ渡される解析木の中の障害を引き起こすシーケンスを含むタプルと説明用の文字列である場合があります。モジュール内の他の関数の呼び出しは単純な文字列値を検出すればよいだけですが、`sequence2st()` の呼び出しはどちらの例外の型も処理できる必要があります。

普通は構文解析とコンパイル処理によって発生する例外を、関数 `compilest()`、`expr()` および `suite()` が発生させることに注意してください。このような例外には組み込み例外 `MemoryError`、`OverflowError`、`SyntaxError` および `SystemError` が含まれます。こうした場合には、これらの例外が通常その例外に関係する全ての意味を伝えます。詳細については、各関数の説明を参照してください。

32.1.5 ST オブジェクト

ST オブジェクト間の順序と等値性の比較がサポートされています。(`pickle` モジュールを使った) ST オブジェクトのピクルス化もサポートされています。

`parser.STType`

`expr()`、`suite()` と `sequence2st()` が返すオブジェクトの型。

ST オブジェクトは次のメソッドを持っています:

`ST.compile([filename])`

`compilest(st, filename)` と同じ。

`ST.isexpr()`

`isexpr(st)` と同じ。

`ST.issuite()`

`issuite(st)` と同じ。

`ST.tolist([line_info])`

`st2list(st, line_info)` と同じ。

`ST.totuple([line_info])`

`st2tuple(st, line_info)` と同じ。

32.1.6 例: `compile()` のエミュレーション

たくさんの有用な演算を構文解析とバイトコード生成の間に行うことができますが、もっとも単純な演算は何もしないことです。このため、`parser` モジュールを使って中間データ構造を作ることは次のコードと等価です

```
>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
```

(次のページに続く)

(前のページからの続き)

```
>>> eval(code)
10
```

`parser` モジュールを使った等価な演算はやや長くなりますが、ST オブジェクトとして中間内部解析木が維持されるようにします:

```
>>> import parser
>>> st = parser.expr('a + 5')
>>> code = st.compile('file.py')
>>> a = 5
>>> eval(code)
10
```

ST とコードオブジェクトの両方が必要なアプリケーションでは、このコードを簡単に利用できる関数にまとめることができます:

```
import parser

def load_suite(source_string):
    st = parser.suite(source_string)
    return st, st.compile()

def load_expression(source_string):
    st = parser.expr(source_string)
    return st, st.compile()
```

32.2 ast — 抽象構文木

バージョン 2.5 で追加: 低級モジュール `_ast` はノードクラスだけを含みます。

バージョン 2.6 で追加: 高級モジュール `ast` は全てのヘルパーを含みます。

ソースコード: [Lib/ast.py](#)

`ast` モジュールは、Python アプリケーションで Python の抽象構文木を処理しやすくするものです。抽象構文そのものは、Python のリリースごとに変化する可能性があります。このモジュールを使用すると、現在の文法をプログラム上で知る助けになるでしょう。

抽象構文木を作成するには、`ast.PyCF_ONLY_AST` を組み込み関数 `compile()` のフラグとして渡すか、あるいはこのモジュールで提供されているヘルパー関数 `parse()` を使います。その結果は、`ast.AST` を継承したクラスのオブジェクトのツリーとなります。抽象構文木は組み込み関数 `compile()` を使って Python コード・オブジェクトにコンパイルすることができます。

32.2.1 Node クラス

class `ast.AST`

このクラスは全ての AST ノード・クラスの基底です。実際のノード・クラスは [後ほど](#) 示す `Parser/Python.asdl` ファイルから派生したものです。これらのクラスは `ast` モジュールで定義され、`ast` にもエクスポートし直されています。

抽象文法の左辺のシンボル一つずつにそれぞれ一つのクラスがあります (たとえば `ast.stmt` や `ast.expr`)。それに加えて、右辺のコンストラクター一つずつにそれぞれ一つのクラスがあり、これらのクラスは左辺のツリーのクラスを継承しています。たとえば、`ast.BinOp` は `ast.expr` から継承しています。代替を伴った生成規則 (production rules with alternatives) (別名 "sums") の場合、左辺は抽象クラスとなり、特定のコンストラクタ・ノードのインスタンスのみが作成されます。

`_fields`

各具象クラスは属性 `_fields` を持っており、すべての子ノードの名前をそこに保持しています。

具象クラスのインスタンスは、各子ノードに対してそれぞれひとつの属性を持っています。この属性は、文法で定義された型となります。たとえば `ast.BinOp` のインスタンスは `left` という属性を持っており、その型は `ast.expr` です。

これらの属性が、文法上 (クエスションマークを用いて) オプションであるとマークされている場合は、その値が `None` となることもあります。属性が 0 個以上の複数の値をとりうる場合 (アスタリスクでマークされている場合) は、値は Python のリストで表されます。全ての属性は `AST` を `compile()` でコンパイルする際には存在しなければならず、そして妥当な値でなければなりません。

`lineno`

`col_offset`

`ast.expr` や `ast.stmt` のサブクラスのインスタンスにはさらに `lineno` や `col_offset` といった属性があります。 `lineno` はソーステキスト上の行番号 (1 から数え始めるので、最初の行の行番号は 1 となります)、そして `col_offset` はノードが生成した最初のトークンの UTF-8 バイトオフセットとなります。UTF-8 オフセットが記録される理由は、パーサが内部で UTF-8 を使用するからです。

クラス `ast.T` のコンストラクタは引数を次のように解析します:

- 位置による引数があるとすれば、`T._fields` にあるのと同じだけの個数が無ければなりません。これらの引数はそこにある名前を持った属性として割り当てられます。
- キーワード引数があるとすれば、それらはその名前の属性にその値を割り当てられます。

たとえば、`ast.UnaryOp` ノードを生成して属性を埋めるには、次のようにすることができます

```
node = ast.UnaryOp()
node.op = ast.USub()
node.operand = ast.Num()
node.operand.n = 5
node.operand.lineno = 0
node.operand.col_offset = 0
```

(次のページに続く)

(前のページからの続き)

```
node.lineno = 0
node.col_offset = 0
```

もしくはよりコンパクトにも書けます

```
node = ast.UnaryOp(ast.USub(), ast.Num(5, lineno=0, col_offset=0),
                  lineno=0, col_offset=0)
```

バージョン 2.6 で追加: 上で説明したコンストラクタが付け加えられました。Python 2.5 においてはノードは引数なしのコンストラクタを呼び出して生成した後、全ての属性をセットしていかなければなりませんでした。

32.2.2 抽象文法 (Abstract Grammar)

このモジュールでは文字列定数 `__version__` を定義しています。これは、以下に示すファイルの 10 進の Subversion リビジョン番号です。

抽象文法は、現在次のように定義されています:

```
-- ASDL's five builtin types are identifier, int, string, object, bool

module Python version "$Revision$"
{
    mod = Module(stmt* body)
        | Interactive(stmt* body)
        | Expression(expr body)

    -- not really an actual node but useful in Jython's typesystem.
    | Suite(stmt* body)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list)
        | ClassDef(identifier name, expr* bases, stmt* body, expr* decorator_
↪list)
        | Return(expr? value)

        | Delete(expr* targets)
        | Assign(expr* targets, expr value)
        | AugAssign(expr target, operator op, expr value)

    -- not sure if bool is allowed, can always use int
    | Print(expr? dest, expr* values, bool nl)

    -- use 'orelse' because else is a keyword in target languages
    | For(expr target, expr iter, stmt* body, stmt* orelse)
    | While(expr test, stmt* body, stmt* orelse)
    | If(expr test, stmt* body, stmt* orelse)
    | With(expr context_expr, expr? optional_vars, stmt* body)
```

(次のページに続く)

(前のページからの続き)

```

-- 'type' is a bad name
| Raise(expr? type, expr? inst, expr? tback)
| TryExcept(stmt* body, excepthandler* handlers, stmt* orelse)
| TryFinally(stmt* body, stmt* finalbody)
| Assert(expr test, expr? msg)

| Import(alias* names)
| ImportFrom(identifier? module, alias* names, int? level)

-- Doesn't capture requirement that locals must be
-- defined if globals is
-- still supports use as a function!
| Exec(expr body, expr? globals, expr? locals)

| Global(identifier* names)
| Expr(expr value)
| Pass | Break | Continue

-- XXX Jython will be different
-- col_offset is the byte offset in the utf8 string the parser uses
attributes (int lineno, int col_offset)

-- BoolOp() can use left & right?
expr = BoolOp(boolop op, expr* values)
| BinOp(expr left, operator op, expr right)
| UnaryOp(unaryop op, expr operand)
| Lambda(arguments args, expr body)
| IfExp(expr test, expr body, expr orelse)
| Dict(expr* keys, expr* values)
| Set(expr* elts)
| ListComp(expr elt, comprehension* generators)
| SetComp(expr elt, comprehension* generators)
| DictComp(expr key, expr value, comprehension* generators)
| GeneratorExp(expr elt, comprehension* generators)
-- the grammar constrains where yield expressions can occur
| Yield(expr? value)
-- need sequences for compare to distinguish between
-- x < 4 < 3 and (x < 4) < 3
| Compare(expr left, cmpop* ops, expr* comparators)
| Call(expr func, expr* args, keyword* keywords,
        expr? starargs, expr? kwargs)
| Repr(expr value)
| Num(object n) -- a number as a PyObject.
| Str(string s) -- need to specify raw, unicode, etc?
-- other literals? bools?

-- the following expression can appear in assignment context
| Attribute(expr value, identifier attr, expr_context ctx)
| Subscript(expr value, slice slice, expr_context ctx)
| Name(identifier id, expr_context ctx)
| List(expr* elts, expr_context ctx)

```

(次のページに続く)

(前のページからの続き)

```

    | Tuple(expr* elts, expr_context ctx)

    -- col_offset is the byte offset in the utf8 string the parser uses
    attributes (int lineno, int col_offset)

    expr_context = Load | Store | Del | AugLoad | AugStore | Param

    slice = Ellipsis | Slice(expr? lower, expr? upper, expr? step)
           | ExtSlice(slice* dims)
           | Index(expr value)

    boolop = And | Or

    operator = Add | Sub | Mult | Div | Mod | Pow | LShift
              | RShift | BitOr | BitXor | BitAnd | FloorDiv

    unaryop = Invert | Not | UAdd | USub

    cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

    comprehension = (expr target, expr iter, expr* ifs)

    -- not sure what to call the first argument for raise and except
    excepthandler = ExceptHandler(expr? type, expr? name, stmt* body)
                   attributes (int lineno, int col_offset)

    arguments = (expr* args, identifier? vararg,
                 identifier? kwarg, expr* defaults)

    -- keyword arguments supplied to call
    keyword = (identifier arg, expr value)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)
}

```

32.2.3 ast ヘルパー

バージョン 2.6 で追加.

ノード・クラスの他に、`ast` モジュールは以下のような抽象構文木をトラバースするためのユーティリティ関数やクラスも定義しています:

```
ast.parse(source, filename='<unknown>', mode='exec')
```

`source` を解析して AST ノードにします。compile(source, filename, mode, ast.PyCF_ONLY_AST) と等価です。

警告: 十分に大きい文字列や複雑な文字列によって Python の抽象構文木コンパイラのスタックの深さの限界を越えることで、Python インタプリタをクラッシュさせることができます。

`ast.literal_eval (node_or_string)`

式ノード、または Python の式を表す Unicode または *Latin-1* エンコード文字列、コンテナのディスプレイ表現を表す文字列、を安全に評価します。与えられる文字列またはノードは次のリテラルのみからなるものに限られます: 文字列, 数, タプル, リスト, 辞書, ブール値, `None`。

この関数は Python の式を含んだ信頼出来ない出どころからの文字列を、値自身を解析することなしに安全に評価するのに使えます。この関数は、例えば演算や添え字を含んだ任意の複雑な表現を評価するのには使えません。

警告: 十分に大きい文字列や複雑な文字列によって Python の抽象構文木コンパイラのスタックの深さの限界を越えることで、Python インタプリタをクラッシュさせることができます。

`ast.get_docstring (node, clean=True)`

与えられた *node* (これは `FunctionDef`, `ClassDef`, `Module` のいずれかのノードでなければなりません) のドキュメント文字列を返します。もしドキュメント文字列が無ければ `None` を返します。*clean* が真ならば、ドキュメント文字列のインデントを `inspect.cleandoc()` を用いて一掃します。

`ast.fix_missing_locations (node)`

`compile()` はノード・ツリーをコンパイルする際、`lineno` と `col_offset` 両属性をサポートする全てのノードに対しそれが存在するものと想定します。生成されたノードに対しこれらを埋めて回るのはどちらかという退屈な作業なので、このヘルパーが再帰的に二つの属性がセットされていないものに親ノードと同じ値をセットしていきます。再帰の出発点が *node* です。

`ast.increment_lineno (node, n=1)`

node から始まるツリーの全てのノードの行番号を *n* ずつ増やします。これはファイルの中で別の場所に「コードを動かす」ときに便利です。

`ast.copy_location (new_node, old_node)`

ソースの場所 (`lineno` と `col_offset`) を *old_node* から *new_node* に可能ならばコピーし、*new_node* を返します。

`ast.iter_fields (node)`

node にある `node._fields` のそれぞれのフィールドを (フィールド名, 値) のタプルとして yield します。

`ast.iter_child_nodes (node)`

node の直接の子ノード全てを yield します。すなわち、yield されるのは、ノードであるような全てのフィールドおよびノードのリストであるようなフィールドの全てのアイテムです。

`ast.walk (node)`

node の全ての子孫ノード (*node* 自体を含む) を再帰的に yield します。順番は決められていません。こ

の関数はノードをその場で変更するだけで文脈を気にしないような場合に便利です。

class `ast.NodeVisitor`

抽象構文木を渡り歩いてビジター関数を見つけたノードごとに呼び出すノード・ビジターの基底クラスです。この関数は `visit()` メソッドに送られる値を返してもかまいません。

このクラスはビジター・メソッドを付け加えたサブクラスを派生させることを意図しています。

visit (*node*)

ノードを訪れます。デフォルトの実装では `self.visit_classname` というメソッド (ここで `classname` はノードのクラス名です) を呼び出すか、そのメソッドがなければ `generic_visit()` を呼び出します。

generic_visit (*node*)

このビジターはノードの全ての子について `visit()` を呼び出します。

注意して欲しいのは、専用のビジター・メソッドを具えたノードの子ノードは、このビジターが `generic_visit()` を呼び出すかそれ自身で子ノードを訪れない限り訪れられないということです。

トラバースの途中でノードを変化させたいならば `NodeVisitor` を使ってはいけません。そうした目的のために変更を許す特別なビジター (`NodeTransformer`) があります。

class `ast.NodeTransformer`

`NodeVisitor` のサブクラスで抽象構文木を渡り歩きながらノードを変更することを許すものです。

`NodeTransformer` は抽象構文木 (AST) を渡り歩き、ビジター・メソッドの戻り値を使って古いノードを置き換えたり削除したりします。ビジター・メソッドの戻り値が `None` ならば、ノードはその場から取り去られ、そうでなければ戻り値で置き換えられます。置き換えない場合は戻り値が元のノードそのものであってもかまいません。

それでは例を示しましょう。Name (たとえば `foo`) を見つけるたび全て `data['foo']` に書き換える変換器 (transformer) です:

```
class RewriteName (NodeTransformer):

    def visit_Name(self, node):
        return copy_location (Subscript (
            value=Name (id='data', ctx=Load()),
            slice=Index (value=Str (s=node.id)),
            ctx=node.ctx
        ), node)
```

操作しようとしているノードが子ノードを持つならば、その子ノードの変形も自分で行うか、またはそのノードに対し最初に `generic_visit()` メソッドを呼び出すか、それを行うのはあなたの責任だということを肝に銘じましょう。

文のコレクションであるようなノード (全ての文のノードが当てはまります) に対して、このビジターは単独のノードではなくノードのリストを返すかもしれません。

たいてい、変換器の使い方は次のようになります:

```
node = YourTransformer().visit(node)
```

`ast.dump(node, annotate_fields=True, include_attributes=False)`

`node` 中のツリーのフォーマットされたダンプを返します。主な使い道はデバッグです。返される文字列は名前とフィールドの値を表示します。これを使うとコードは評価できなくなりますので、評価が必要ならば `annotate_fields` に `False` をセットしなければなりません。行番号や列オフセットのような属性はデフォルトではダンプされません。これが欲しければ、`include_attributes` を `True` にセットすることができます。

32.3 symtable — コンパイラの記号表へのアクセス

ソースコード: [Lib/symtable.py](#)

記号表 (symbol table) は、コンパイラが AST からバイトコードを生成する直前に作られます。記号表はコード中の全ての識別子のスコープの算出に責任を持ちます。 `symtable` はこうした記号表を調べるインターフェイスを提供します。

32.3.1 記号表の生成

`symtable.symtable(code, filename, compile_type)`

Python ソース `code` に対するトップレベルの `SymbolTable` を返します。 `filename` はコードを収めているファイルの名前です。 `compile_type` は `compile()` の `mode` 引数のようなものです。

32.3.2 記号表の検査

`class symtable.SymbolTable`

ブロックに対する名前空間の記号表。コンストラクタはパブリックではありません。

`get_type()`

記号表の型を返します。有り得る値は `'class'`, `'module'`, `'function'` です。

`get_id()`

記号表の識別子を返します。

`get_name()`

記号表の名前を返します。この名前は記号表がクラスに対するものであればクラス名であり、関数に対するものであれば関数名であり、グローバルな (`get_type()` が `'module'` を返す) 記号表であれば `'top'` です。

`get_lineno()`

この記号表に対応するブロックの一行目の行番号を返します。

`is_optimized()`

この記号表に含まれるローカル変数が最適化できるならば `True` を返します。

is.nested()

ブロックが入れ子のクラスまたは関数のとき `True` を返します。

has_children()

ブロックが入れ子の名前空間を含んでいるならば `True` を返します。入れ子の名前空間は `get_children()` で得られます。

has_exec()

ブロックの中で `exec` が使われているならば `True` を返します。

has_import_star()

ブロックの中で `*` を使った `from-import` が使われているならば `True` を返します。

get_identifiers()

この記号表にある記号の名前のリストを返します。

lookup(name)

記号表から名前 `name` を見つけ出して `Symbol` インスタンスとして返します。

get_symbols()

記号表中の名前を表す `Symbol` インスタンスのリストを返します。

get_children()

入れ子になった記号表のリストを返します。

class symtable.Function

関数またはメソッドの名前空間。このクラスは `SymbolTable` を継承しています。

get_parameters()

この関数の引数名からなるタプルを返します。

get_locals()

この関数のローカル変数の名前からなるタプルを返します。

get_globals()

この関数のグローバル変数の名前からなるタプルを返します。

get_frees()

この関数の自由変数の名前からなるタプルを返します。

class symtable.Class

クラスの名前空間。このクラスは `SymbolTable` を継承しています。

get_methods()

このクラスで宣言されているメソッド名からなるタプルを返します。

class symtable.Symbol

`SymbolTable` のエントリーでソースの識別子に対応するものです。コンストラクタはパブリックではありません。

get_name()

記号の名前を返します。

is_referenced()記号がそのブロックの中で使われていれば `True` を返します。**is_imported()**記号が `import` 文で作られたものならば `True` を返します。**is_parameter()**記号がパラメータならば `True` を返します。**is_global()**記号がグローバルならば `True` を返します。**is_declared_global()**記号が `global` 文によってグローバルであると宣言されているなら `True` を返します。**is_local()**記号がそのブロックに対してローカルならば `True` を返します。**is_free()**記号がそのブロックの中で参照されていて、しかし代入は行われないならば `True` を返します。**is_assigned()**記号がそのブロックの中で代入されているならば `True` を返します。**is_namespace()**名前の束縛が新たな名前空間を導入するならば `True` を返します。名前が関数または `class` 文のターゲットとして使われるならば、真です。

例えば:

```
>>> table = symtable.symtable("def some_func(): pass", "string", "exec")
>>> table.lookup("some_func").is_namespace()
True
```

一つの名前が複数のオブジェクトに束縛されうことに注意しましょう。結果が `True` であったとしても、その名前は他のオブジェクトにも束縛されるかも知れず、それがたとえば整数やリストであれば、そこでは新たな名前空間は導入されません。

get_namespaces()

この名前に束縛された名前空間のリストを返します。

get_namespace()

この名前に束縛されたただ一つの名前空間を返します。束縛された名前空間が一つより多くあれば `ValueError` が送出されます。

32.4 symbol — Python 解析木と共に使われる定数

ソースコード: [Lib/symbol.py](#)

このモジュールは解析木の内部ノードの数値を表す定数を提供します。ほとんどの Python 定数とは違い、これらは小文字の名前を使います。言語の文法のコンテキストにおける名前の定義については、Python ディストリビューションのファイル `Grammar/Grammar` を参照してください。名前がマップする特定の数値は Python のバージョン間で変わります。

このモジュールには、データオブジェクトも一つ付け加えられています:

`symbol.sym_name`

ディクショナリはこのモジュールで定義されている定数の数値を名前の文字列へマップし、より人が読みやすいように解析木を表現します。

32.5 token — Python 解析木と共に使われる定数

ソースコード: [Lib/token.py](#)

このモジュールは解析木の葉ノード (終端記号) の数値を表す定数を提供します。言語の文法のコンテキストにおける名前の定義については、Python ディストリビューションのファイル `Grammar/Grammar` を参照してください。名前がマップする特定の数値は Python のバージョン間で変わります。

このモジュールは、数値コードから名前へのマッピングと、いくつかの関数も提供しています。関数は Python の C ヘッドファイルの定義を反映します。

`token.tok_name`

ディクショナリはこのモジュールで定義されている定数の数値を名前の文字列へマップし、より人が読みやすいように解析木を表現します。

`token.ISTERMINAL` (*x*)

終端トークンの値に対して真を返します。

`token.ISNONTERMINAL` (*x*)

非終端トークンの値に対して真を返します。

`token.ISEOF` (*x*)

x が入力の終わりを示すマーカーならば、真を返します。

token の定数一覧:

`token.ENDMARKER`

`token.NAME`

`token.NUMBER`

`token.STRING`

`token.NEWLINE`

`token.INDENT`

`token.DEDENT`

`token.LPAR`

`token.RPAR`

`token.LSQB`

`token.RSQB`
`token.COLON`
`token.COMMA`
`token.SEMI`
`token.PLUS`
`token.MINUS`
`token.STAR`
`token.SLASH`
`token.VBAR`
`token.AMPER`
`token.LESS`
`token.GREATER`
`token.EQUAL`
`token.DOT`
`token.PERCENT`
`token.BACKQUOTE`
`token.LBRACE`
`token.RBRACE`
`token.EQEQUAL`
`token.NOTEQUAL`
`token.LESSEQUAL`
`token.GREATEREQUAL`
`token.TILDE`
`token.CIRCUMFLEX`
`token.LEFTSHIFT`
`token.RIGHTSHIFT`
`token.DOUBLESTAR`
`token.PLUSEQUAL`
`token.MINEQUAL`
`token.STAREQUAL`
`token.SLASHEQUAL`
`token.PERCENTEQUAL`
`token.AMPEREQUAL`
`token.VBAREQUAL`
`token.CIRCUMFLEXEQUAL`
`token.LEFTSHIFTEQUAL`
`token.RIGHTSHIFTEQUAL`
`token.DOUBLESTAREQUAL`
`token.DOUBLESASH`
`token.DOUBLESASHEQUAL`
`token.AT`
`token.OP`
`token.ERRORTOKEN`
`token.N_TOKENS`

`token.NT_OFFSET`

参考:

`parser` モジュール `parser` モジュールの二番目の例で、`symbol` モジュールの使い方を示しています。

32.6 keyword — Python キーワードチェック

ソースコード: [Lib/keyword.py](#)

このモジュールでは、Python プログラムで文字列がキーワードか否かをチェックする機能を提供します。

`keyword.iskeyword(s)`

`s` が Python のキーワードであれば真を返します。

`keyword.kwlist`

インタプリタで定義している全てのキーワードのシーケンス。特定の `__future__` 宣言がなければ有効ではないキーワードでもこのリストには含まれます。

32.7 tokenize — Python ソースのためのトークナイザ

ソースコード: [Lib/tokenize.py](#)

`tokenize` モジュールでは、Python で実装された Python ソースコードの字句解析器を提供します。さらに、このモジュールの字句解析器はコメントもトークンとして返します。このため、このモジュールはスクリーン上で表示する際の色付け機能 (colorizers) を含む "清書出力器 (pretty-printer)" を実装する上で便利です。

トークンストリームの扱いをシンプルにするために、全ての operators と delimiters トークンはジェネリックな `token.OP` トークンタイプとして返されます。正確な型は `tokenize.generate_tokens()` からの戻り値のタプルの 2 番目のフィールド (実際のマッチしたトークン文字列を含んでいます) を特定の演算子トークンを識別する文字シーケンスかチェックすれば決定出来ます。

第一のエントリポイントはジェネレータ (*generator*) です:

`tokenize.generate_tokens(readline)`

`generate_tokens()` ジェネレータは一つの引数 `readline` を必要とします。この引数は呼び出し可能オブジェクトで、組み込みファイルオブジェクトにおける `readline()` メソッドと同じインタフェースを提供していなければなりません ([ファイルオブジェクト](#) 節を参照してください)。この関数は呼び出しのたびに入力内の一行を文字列で返さなければなりません。もしくは、`readline` を `StopIteration` を送出することで完了を知らせる呼び出し可能オブジェクトとすることもできます。

このジェネレータは次の 5 要素のタプルを返します; トークンタイプ; トークン文字列; ソース内でそのトークンがどの行、列で開始するかを示す int の (`srow`, `scol`) タプル; どの行、列で終了するかを示す int の (`erow`, `ecol`) タプル; トークンが見つかった行。 (タプルの最後の要素にある) 行は 論理行で、継続行が含まれます。

バージョン 2.2 で追加.

後方互換性のために古いエントリポイントが残されています:

`tokenize.tokenize(readline[, tokeneater])`

`tokenize()` 関数は二つのパラメータを取ります: 一つは入力ストリームを表し、もう一つは `tokenize()` のための出力メカニズムを与えます。

最初のパラメータ、`readline` は、組み込みファイルオブジェクトの `readline()` メソッドと同じインタフェースを提供する呼び出し可能オブジェクトでなければなりません (`ファイルオブジェクト` 節を参照)。この関数は呼び出しのたびに入力内の一行を文字列で返さなければなりません。もしくは、`readline` を `StopIteration` を送出することで完了を知らせる呼び出し可能オブジェクトとすることもできます。

バージョン 2.5 で変更: `StopIteration` サポートの追加。

二番目のパラメータ `tokeneater` も呼び出し可能オブジェクトでなければなりません。この関数は各トークンに対して一度だけ呼び出され、`generate_tokens()` が生成するタプルに対応する 5 つの引数をとります。

`token` モジュールの全ての定数は `tokenize` でも公開されており、これに加え、以下の二つのトークン値が `tokenize()` の `tokeneater` 関数に渡される可能性があります:

`tokenize.COMMENT`

コメントであることを表すために使われるトークン値です。

`tokenize.NL`

終わりではない改行を表すために使われるトークン値。NEWLINE トークンは Python コードの論理行の終わりを表します。NL トークンはコードの論理行が複数の物理行にわたって続いているときに作られます。

もう一つの関数がトークン化プロセスを逆転するために提供されています。これは、スクリプトを字句解析し、トークンのストリームに変更を加え、変更されたスクリプトを書き戻すようなツールを作成する際に便利です。

`tokenize.untokenize(iterable)`

トークンの列を Python ソースコードに変換します。 `iterable` は少なくとも二つの要素、トークンタイプおよびトークン文字列、からなるシーケンスを返さなければいけません。 その他のシーケンスの要素は無視されます。

再構築されたスクリプトは一つの文字列として返されます。得られる結果はもう一度字句解析すると入力と一致することが保証されるので、変換がロスレスでありラウンドトリップできることは間違いありません。この保証はトークン型およびトークン文字列に対してのものでトークン間のスペース (コラム位置) のようなものは変わることがあり得ます。

バージョン 2.5 で追加.

exception `tokenize.TokenError`

docstring や複数行にわたることが許される式がファイル内のどこかで終わっていない場合に送出されます。例えば:

```
"""Beginning of
docstring
```

もしくは:

```
[1,
 2,
 3
```

閉じていないシングルクォート文字列は送出されるべきエラーの原因とならないことに注意してください。それらは `ERRORTOKEN` とトークン化され、続いてその内容がトークン化されます。

スクリプト書き換えの例で、浮動小数点数リテラルを `Decimal` オブジェクトに変換します:

```
def decistmt(s):
    """Substitute Decimals for floats in a string of statements.

    >>> from decimal import Decimal
    >>> s = 'print +21.3e-5*-.1234/81.7'
    >>> decistmt(s)
    "print +Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7') "

    >>> exec(s)
    -3.21716034272e-007
    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7

    """
    result = []
    g = generate_tokens(StringIO(s).readline)    # tokenize the string
    for toknum, tokval, _, _, _ in g:
        if toknum == NUMBER and '.' in tokval:  # replace NUMBER tokens
            result.extend([
                (NAME, 'Decimal'),
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')
            ])
        else:
            result.append((toknum, tokval))
    return untokenize(result)
```

32.8 tabnanny — あいまいなインデントの検出

ソースコード: `Lib/tabnanny.py`

差し当たり、このモジュールはスクリプトとして呼び出すことを意図しています。しかし、IDE 上にインポートして下で説明する関数 `check()` を使うことができます。

注釈: このモジュールが提供する API を将来のリリースで変更する可能性が高いです。このような変更は後方互換性がないかもしれません。

`tabnanny.check(file_or_dir)`

`file_or_dir` がディレクトリであってシンボリックリンクでないときに、`file_or_dir` という名前のディレクトリツリーを再帰的に下って行き、この通り道に沿ってすべての `.py` ファイルをチェックします。`file_or_dir` が通常の Python ソースファイルの場合には、問題のある空白をチェックします。診断メッセージは `print` 文を使って標準出力に書き込まれます。

`tabnanny.verbose`

冗長なメッセージをプリントするかどうかを示すフラグ。スクリプトとして呼び出された場合は、`-v` オプションによって増加します。

`tabnanny.filename_only`

問題のある空白を含むファイルのファイル名のみをプリントするかどうかを示すフラグ。スクリプトとして呼び出された場合は、`-q` オプションによって真に設定されます。

exception `tabnanny.NannyNag`

あいまいなインデントを検出した場合に `process_tokens()` が送出します。この例外は `check()` で捕捉され処理されます。

`tabnanny.process_tokens(tokens)`

この関数は、`tokenize` モジュールによって生成されたトークンを `check()` が処理するために使われます。

参考:

`tokenize` モジュール Python ソースコードの字句解析器。

32.9 pyc1br — Python クラスブラウザサポート

ソースコード: [Lib/pyc1br.py](#)

この `pyc1br` モジュールはモジュールで定義されたクラス、メソッド、およびトップレベルの関数について、限られた量の情報を取得するのに使われます。伝統的な 3 ペイン形式のクラスブラウザを実装するのに十分な情報を提供します。情報は、モジュールのインポートによらず、ソースコードから抽出します。このため、このモジュールは信用できないソースコードに対して利用しても安全です。この制限から、多くの標準モジュールやオプションの拡張モジュールを含む、Python で実装されていないモジュールに対して利用することはできません。

`pyc1br.readmodule(module, path=None)`

モジュールを読み込み、クラス名からクラス記述オブジェクトにマップする辞書を返します。パラメタ `module` はモジュール名を表す文字列でなくてはなりません; パッケージ内のモジュール名でもかまいません。`path` パラメタはシーケンス型でなくてはならず、モジュールのソースコードがある場所を特定する際に `sys.path` の値に加えて使われます。

`pyclbr.readmodule_ex(module, path=None)`

`readmodule()` に似ていますが、返される辞書は、クラス名からクラス記述オブジェクトへの対応付けに加えて、トップレベル関数から関数記述オブジェクトへの対応付けも行っています。さらに、読み出し対象のモジュールがパッケージである場合、返される辞書はキー `'__path__'` を持ち、その値はパッケージの検索パスが入ったリストになります。

32.9.1 Class オブジェクト

Class オブジェクトは、`readmodule()` や `readmodule_ex()` が返す辞書の値として使われており、以下のデータ属性を提供しています:

Class.module

クラス記述オブジェクトが記述している対象のクラスを定義しているモジュールの名前です。

Class.name

クラスの名前です。

Class.super

記述しようとしている対象クラスの、直接の基底クラス群について記述している Class オブジェクトのリストです。スーパークラスとして挙げられているが `readmodule()` が見つけられなかったクラスは、Class オブジェクトではなくクラス名の文字列としてリストに挙げられます。

Class.methods

メソッド名を行番号に対応付ける辞書です。

Class.file

クラスを定義している `class` 文が入っているファイルの名前です。

Class.lineno

`file` で指定されたファイルにおける `class` 文の行番号です。

32.9.2 Function オブジェクト

Function オブジェクトは、`readmodule_ex()` が返す辞書内でキーに対応する値として使われており、以下のデータ属性を提供しています:

Function.module

関数記述オブジェクトが記述している対象の関数を定義しているモジュールの名前です。

Function.name

関数の名前です。

Function.file

関数を定義してる `def` 文が入っているファイルの名前です。

Function.lineno

`file` で指定されたファイルにおける `def` 文の行番号です。

32.10 `py_compile` — Python ソースファイルのコンパイル

ソースコード: `Lib/py_compile.py`

`py_compile` モジュールには、ソースファイルからバイトコードファイルを作る関数と、モジュールのソースファイルがスクリプトとして呼び出される時に使用される関数が定義されています。

頻繁に必要なわけではありませんが、共有ライブラリとしてモジュールをインストールする場合や、特にソースコードのあるディレクトリにバイトコードのキャッシュファイルを書き込む権限がないユーザがいるときには、この関数は役に立ちます。

exception `py_compile.PyCompileError`

ファイルをコンパイル中にエラーが発生すると送出される例外。

`py_compile.compile(file[, cfile[, dfile[, doraise]]])`

ソースファイルをバイトコードにコンパイルして、バイトコードのキャッシュファイルに書き出します。ソースコードは `file` という名前のファイルから読み込みます。バイトコードはファイル `cfile` に書き込まれ、デフォルトでは `file + '.c'` (使用しているインタプリタで最適化が可能なら `'.o'`) です。もし `dfile` が指定されたら、`file` の代わりにソースファイルの名前としてエラーメッセージの中で使われます。`doraise` が真の場合、コンパイル中にエラーが発生すると `PyCompileError` を送出します。`doraise` が偽の場合 (デフォルト) はエラーメッセージは `sys.stderr` に出力されますが、例外は送出しません。

`py_compile.main([args])`

いくつか複数のソースファイルをコンパイルします。`args` で (あるいは `args` で指定されなかったらコマンドラインで) 指定されたファイルをコンパイルし、できたバイトコードを通常の方法で保存します。この関数はソースファイルの存在するディレクトリを検索しません。指定されたファイルをコンパイルするだけです。`args` が `'-'` 1 つだけだった場合、ファイルのリストは標準入力から取られます。

バージョン 2.7 で変更: `'-'` のサポートが追加されました。

このモジュールがスクリプトとして実行されると、`main()` がコマンドラインで指定されたファイルを全てコンパイルします。一つでもコンパイルできないファイルがあると終了ステータスが 0 でない値になります。

バージョン 2.6 で変更: モジュールがスクリプトとして実行された場合の 0 でない終了ステータスが追加されました。

参考:

`compileall` モジュール ディレクトリツリー内の Python ソースファイルを全てコンパイルするライブラリ。

32.11 `compileall` — Python ライブラリをバイトコンパイル

ソースコード: `Lib/compileall.py`

このモジュールは、Python ライブラリのインストールを助けるユーティリティ関数群を提供します。この関数群は、ディレクトリツリー内の Python ソースファイルをコンパイルします。このモジュールを使って、キャッシュされたバイトコードファイルをライブラリのインストール時に生成することで、ライブラリディレクトリに書き込み権限をもたないユーザでも、これらを利用できるようになります。

32.11.1 コマンドラインでの使用

このモジュールは、(`python -m compileall` を使って) Python ソースをコンパイルするスクリプトとして機能します。

directory ...

file ...

位置引数は、コンパイルするファイル群か、再帰的に横断されるディレクトリでソースファイル群を含むものです。引数が与えられなければ、`-l <directories from sys.path>` を渡したのと同じように動作します。

-l

サブディレクトリを再帰処理せず、指名または暗示されたディレクトリ群に含まれるソースコードファイル群だけをコンパイルします。

-f

タイムスタンプが最新であってもリビルドを強制します。

-q

コンパイルされたファイルを一覧表示せず、エラーメッセージのみ表示します。

-d *destdir*

コンパイルされるそれぞれのファイルへのパスの先頭に、ディレクトリを追加します。これはコンパイル時トレースバックに使われ、バイトコードファイルが実行される時点でソースファイルが存在しない場合に、トレースバックやその他のメッセージに使われるバイトコードファイルにもコンパイルされます。

-x *regex*

regex を使って、コンパイル候補のそれぞれのファイルのフルパスを検索し、*regex* がマッチしたファイルを除外します。

-i *list*

ファイル *list* を読み込み、そのファイルのそれぞれの行を、コンパイルするファイルとディレクトリのリストに加えます。*list* が `-` なら、`stdin` の行を読み込みます。

バージョン 2.7 で変更: `-i` オプションが追加されました。

32.11.2 パブリックな関数

`compileall.compile_dir(dir[, maxlevels[, ddir[, force[, rx[, quiet]]]])`

dir で指定されたディレクトリを再帰的に下降し、見つかった `.py` を全てコンパイルします。

maxlevels パラメタは、再帰する深さの限度に使われ、デフォルトは 10 です。

ddir が与えられれば、コンパイルされるそれぞれのファイルへのパスの先頭に、そのディレクトリを追加します。これはコンパイル時トレースバックに使われ、バイトコードファイルが実行される時点でソースファイルが存在しない場合に、トレースバックやその他のメッセージに使われるバイトコードファイルにもコンパイルされます。

force が真の場合、モジュールはファイルの更新日付に関わりなく再コンパイルされます。

rx が与えられれば、コンパイル候補のそれぞれのファイルのフルパスに対して検索メソッドが呼び出され、それが真値を返したら、そのファイルは除外されます。

quiet が真の場合、エラーが起こらない限り標準出力に何も表示しません。

`compileall.compile_file(fullname[, ddir[, force[, rx[, quiet]]])`

パス *fullname* のファイルをコンパイルします。

ddir が与えられれば、コンパイルされるファイルへのパスの先頭に、そのディレクトリを追加します。これはコンパイル時トレースバックに使われ、バイトコードファイルが実行される時点でソースファイルが存在しない場合に、トレースバックやその他のメッセージに使われるバイトコードファイルにもコンパイルされます。

rx が与えられれば、コンパイル候補のファイルのフルパスに対して検索メソッドが呼び出され、それが真値を返したら、ファイルはコンパイルされず、`True` が返されます。

quiet が真の場合、エラーが起こらない限り標準出力に何も表示しません。

バージョン 2.7 で追加。

`compileall.compile_path([skip_curdir[, maxlevels[, force]]])`

`sys.path` に含まれる、全ての `.py` ファイルをバイトコンパイルします。 *skip_curdir* が真 (デフォルト) の時、カレントディレクトリは検索されません。その他のパラメタはすべて `compile_dir()` 関数に渡されます。なお、他のコンパイル関数とは違い、*maxlevels** のデフォルトは 0 です。

Lib/ ディレクトリ以下にある全ての `.py` ファイルを強制的に再コンパイルするには、以下のようにします:

```
import compileall

compileall.compile_dir('Lib/', force=True)

# Perform same compilation, excluding files in .svn directories.
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\\][.]svn'), force=True)
```

参考:

Module `py_compile` 一つのソースファイルをバイトコンパイルします。

32.12 `dis` — Python バイトコードの逆アセンブラ

Source code: [Lib/dis.py](#)

`dis` モジュールは CPython バイトコード (*bytecode*) を逆アセンブルすることでバイトコードの解析をサポートします。このモジュールがとして受け取る CPython バイトコードはファイル `Include/opcode.h` に定義されており、コンパイラとインタプリタが使用しています。

バイトコードは CPython インタプリタの実装詳細です! Python のバージョン間でバイトコードの追加や、削除、変更がないという保証はありません。このモジュールを使用することによって Python の異なる VM または異なるリリースの間で動作すると考えるべきではありません。

例: 以下の関数 `myfunc()` を考えると

```
def myfunc(alist):  
    return len(alist)
```

`myfunc()` の逆アセンブル結果を得るために次のコマンドを使うことができます

```
>>> dis.dis(myfunc)  
2          0 LOAD_GLOBAL              0 (len)  
          3 LOAD_FAST                0 (alist)  
          6 CALL_FUNCTION             1  
          9 RETURN_VALUE
```

("2" は行番号です)。

`dis` モジュールは次の関数と定数を定義します:

`dis.dis([bytesource])`

bytesource オブジェクトを逆アセンブルします。 *bytesource* はモジュール、クラス、関数、あるいはコードオブジェクトのいずれかを示します。モジュールに対しては、すべての関数を逆アセンブルします。クラスに対しては、すべてのメソッドを逆アセンブルします。単一のコード列に対しては、バイトコード命令ごとに 1 行を出力します。オブジェクトが与えられない場合は、最後のトレースバックを逆アセンブルします。

`dis.distb([tb])`

トレースバックのスタックの先頭の関数を逆アセンブルします。 `None` が渡された場合は最後のトレースバックを使います。例外を引き起こした命令が表示されます。

`dis.disassemble(code[, lasti])`

コードオブジェクトを逆アセンブルします。 *lasti* が与えられた場合は、最後の命令を示します。出力は次のようなカラムに分割されます:

1. 各行の最初の命令に対する行番号。
2. 現在の命令。 `-->` として示されます。
3. ラベル付けされた命令。 `>>` とともに表示されます。

4. 命令のアドレス。
5. 命令コード名。
6. 命令パラメタ。
7. パラメタの解釈を括弧で囲んだもの。

パラメタの解釈は、ローカル変数とグローバル変数の名前、定数の値、分岐先、比較命令を認識します。

`dis.disco(code[, lasti])`

`disassemble()` の別名。よりタイプしやすく、以前の Python リリースと互換性があります。

`dis.findlinestarts(code)`

このジェネレータ関数は、コードオブジェクト `code` の `co_firstlineno` と `co_lnotab` 属性を使い、ソースコード内の行が始まる場所であるオフセットを求めます。これらは `(offset, lineno)` の対として生成されます。

`dis.findlabels(code)`

ジャンプ先であるコードオブジェクト `code` のすべてのオフセットを求め、これらのオフセットのリストを返します。

`dis.opname`

命令コード名のリスト。バイトコードをインデクスにを使って参照できます。

`dis.opmap`

命令コード名をバイトコードに対応づける辞書。

`dis.cmp_op`

すべての比較命令の名前のリスト。

`dis.hasconst`

Sequence of bytecodes that access a constant.

`dis.hasfree`

自由変数にアクセスするバイトコードのリスト。

`dis.hasname`

名前によって属性にアクセスするバイトコードのリスト。

`dis.hasjrel`

相対ジャンプ先を持つバイトコードのリスト。

`dis.hasjabs`

絶対ジャンプ先を持つバイトコードのリスト。

`dis.haslocal`

ローカル変数にアクセスするバイトコードのリスト。

`dis.hascompare`

ブール命令のバイトコードのリスト。

32.12.1 Python バイトコード命令

現在 Python コンパイラは次のバイトコード命令を生成します。

STOP_CODE()

コンパイラにコードの終わりを知らせます。インタプリタでは使われません。

NOP()

なにもしないコード。バイトコードオプティマイザでブレースホルダとして使われます。

POP_TOP()

スタックの先頭 (TOS) の要素を取り除きます。

ROT_TWO()

スタックの先頭の 2 つの要素を入れ替えます。

ROT_THREE()

スタックの二番目と三番目の要素の位置を 1 つ上げ、先頭を三番目へ下げます。

ROT_FOUR()

スタックの二番目、三番目および四番目の位置を 1 つ上げ、先頭を四番目に下げます。

DUP_TOP()

スタックの先頭にある参照の複製を作ります。

単項命令はスタックの先頭を取り出して操作を適用し、結果をスタックへプッシュし戻します。

UNARY_POSITIVE()

$TOS = +TOS$ に対応します。

UNARY_NEGATIVE()

$TOS = -TOS$ に対応します。

UNARY_NOT()

$TOS = \text{not } TOS$ に対応します。

UNARY_CONVERT()

$TOS = \text{`TOS`}$ に対応します。

UNARY_INVERT()

$TOS = \sim TOS$ に対応します。

GET_ITER()

$TOS = \text{iter}(TOS)$ に対応します。

二項命令はスタックの先頭 (TOS) と先頭から二番目の要素をスタックから取り除きます。命令を実行し、スタックへ結果をプッシュし戻します。

BINARY_POWER()

$TOS = TOS1 ** TOS$ に対応します。

BINARY_MULTIPLY()

TOS = TOS1 * TOS に対応します。

BINARY_DIVIDE()

from __future__ import division が有効でないときの TOS = TOS1 / TOS に対応します。

BINARY_FLOOR_DIVIDE()

TOS = TOS1 // TOS に対応します。

BINARY_TRUE_DIVIDE()

from __future__ import division が有効なときの TOS = TOS1 / TOS に対応します。

BINARY_MODULO()

TOS = TOS1 % TOS に対応します。

BINARY_ADD()

TOS = TOS1 + TOS に対応します。

BINARY_SUBTRACT()

TOS = TOS1 - TOS に対応します。

BINARY_SUBSCR()

TOS = TOS1[TOS] に対応します。

BINARY_LSHIFT()

TOS = TOS1 << TOS に対応します。

BINARY_RSHIFT()

TOS = TOS1 >> TOS に対応します。

BINARY_AND()

TOS = TOS1 & TOS に対応します。

BINARY_XOR()

TOS = TOS1 ^ TOS に対応します。

BINARY_OR()

TOS = TOS1 | TOS に対応します。

インプレース命令は TOS と TOS1 を取り除いて結果をスタックへプッシュするという点で二項命令と似ています。しかし、TOS1 がインプレース命令をサポートしている場合には操作が直接 TOS1 に行われます。また、操作結果の TOS は (常に同じというわけではありませんが) 元の TOS1 と同じオブジェクトになることが多いです。

INPLACE_POWER()

インプレースの TOS = TOS1 ** TOS に対応します。

INPLACE_MULTIPLY()

インプレースの TOS = TOS1 * TOS に対応します。

INPLACE_DIVIDE()

from __future__ import division が有効でないときのインプレースの TOS = TOS1 / TOS

に対応します。

INPLACE_FLOOR_DIVIDE()

インプレースの $TOS = TOS1 // TOS$ に対応します。

INPLACE_TRUE_DIVIDE()

`from __future__ import division` が有効なときのインプレースの $TOS = TOS1 / TOS$ に対応します。

INPLACE_MODULO()

インプレースの $TOS = TOS1 \% TOS$ に対応します。

INPLACE_ADD()

インプレースの $TOS = TOS1 + TOS$ に対応します。

INPLACE_SUBTRACT()

インプレースの $TOS = TOS1 - TOS$ に対応します。

INPLACE_LSHIFT()

インプレースの $TOS = TOS1 << TOS$ に対応します。

INPLACE_RSHIFT()

インプレースの $TOS = TOS1 >> TOS$ に対応します。

INPLACE_AND()

インプレースの $TOS = TOS1 \& TOS$ に対応します。

INPLACE_XOR()

インプレースの $TOS = TOS1 \wedge TOS$ に対応します。

INPLACE_OR()

インプレースの $TOS = TOS1 | TOS$ に対応します。

スライス命令コードは最大 3 つのパラメタを取ります。

SLICE+0()

$TOS = TOS[:]$ に対応します。

SLICE+1()

$TOS = TOS1[TOS:]$ に対応します。

SLICE+2()

$TOS = TOS1[:TOS]$ に対応します。

SLICE+3()

$TOS = TOS2[TOS1:TOS]$ に対応します。

スライス代入はさらにもう 1 つのパラメタを必要とします。他の文と同じく、これらはスタックに何もプッシュしません。

STORE_SLICE+0()

$TOS[:] = TOS1$ に対応します。

STORE_SLICE+1()

TOS1[TOS:] = TOS2 に対応します。

STORE_SLICE+2()

TOS1[:TOS] = TOS2 に対応します。

STORE_SLICE+3()

TOS2[TOS1:TOS] = TOS3 に対応します。

DELETE_SLICE+0()

del TOS[:] に対応します。

DELETE_SLICE+1()

del TOS1[TOS:] に対応します。

DELETE_SLICE+2()

del TOS1[:TOS] に対応します。

DELETE_SLICE+3()

del TOS2[TOS1:TOS] に対応します。

STORE_SUBSCR()

TOS1[TOS] = TOS2 に対応します。

DELETE_SUBSCR()

del TOS1[TOS] に対応します。

その他の命令コード。

PRINT_EXPR()

対話モードのための式文に対応します。TOS はスタックから取り除かれ表示されます。非対話モードにおいては、式文は *POP_TOP* で終了しています。

PRINT_ITEM()

sys.stdout に束縛されたファイル互換オブジェクトに対して TOS を出力します。print 文の各要素に対してこのような命令が一つずつあります。

PRINT_ITEM.TO()

PRINT_ITEM と似ていますが、TOS から二番目の要素を TOS にあるファイル互換オブジェクトへ出力します。これは拡張 print 文で使われます。

PRINT_NEWLINE()

sys.stdout へ改行を表示します。これは print 文がコンマで終わっていない場合に print 文の最後の命令として生成されます。

PRINT_NEWLINE.TO()

PRINT_NEWLINE と似ていますが、TOS のファイル互換オブジェクトに改行を表示します。これは拡張 print 文で使われます。

BREAK_LOOP()

break 文によってループを終了します。

CONTINUE_LOOP (*target*)

`continue` 文によってループを継続します。 *target* はジャンプするアドレスです (アドレスは `FOR_ITER` 命令でなければなりません)。

LIST_APPEND (*i*)

`list.append(TOS[-i], TOS)` を呼びます。リスト内包表記を実装するために使われます。要素が取り除かれる間、リストオブジェクトはスタックに残るので、ループの反復をさらに行えます。

LOAD_LOCALS ()

現在のスコープのローカルな名前空間 (`locals`) への参照をスタックにプッシュします。これはクラス定義のためのコードで使われます: クラス本体が評価された後、`locals` はクラス定義へ渡されます。

RETURN_VALUE ()

関数の呼び出し元へ `TOS` を返します。

YIELD_VALUE ()

`TOS` をポップし、それをジェネレータ (*generator*) から `yield` します。

IMPORT_STAR ()

'_'で始まっていないすべてのシンボルをモジュール `TOS` から直接ローカル名前空間へロードします。モジュールはすべての名前をロードした後にポップされます。この命令コードは `from module import *` に対応します。

EXEC_STMT ()

`exec TOS2, TOS1, TOS` に対応します。コンパイラは指定されなかったオプションのパラメタを `None` で埋めます。

POP_BLOCK ()

ブロックスタックからブロックを一つ取り除きます。フレームごとにブロックのスタックがあり、ネストしたループや `try` 文などを表しています。

END_FINALLY ()

`finally` 節を終了します。インタプリタは例外を再送出しなければならないかどうか、あるいは、関数から `return` して外側の次のブロックに続くかどうかを再度判断します。

BUILD_CLASS ()

新しいクラスオブジェクトを作成します。`TOS` はメソッド辞書、`TOS1` は基底クラスの名前のタプル、`TOS2` はクラス名です。

SETUP_WITH (*delta*)

この命令コードは、`with` ブロックが開始する前にいくつかの命令を行います。まず、コンテキストマネージャから `__exit__()` をロードし、後から `WITH_CLEANUP` で使うためにスタックにプッシュします。そして、`__enter__()` が呼び出され、*delta* を指す `finally` ブロックがプッシュされます。最後に、`enter` メソッドを呼び出した結果がスタックにプッシュされます。次の命令コードはこれを見捨て (`POP_TOP`) するか、変数に保存 (`STORE_FAST`, `STORE_NAME`, または `UNPACK_SEQUENCE`) します。

WITH_CLEANUP ()

`with` 式ブロックを抜けるときに、スタックをクリーンアップします。スタックの先頭は 1-3 個の値で、それらはなぜ/どのように `finally` 節に到達したかを表しています:

- `TOP = None`
- `(TOP, SECOND) = (WHY_{RETURN, CONTINUE}), retval`
- `TOP = WHY_*`; no `retval` below it
- `(TOP, SECOND, THIRD) = exc_info()`

これらの値の下には、コンテキストマネージャーの `__exit__()` 結合メソッド (bound method) である `EXIT` があります。

最後のケースでは、`EXIT(TOP, SECOND, THIRD)` が呼ばれ、それ以外では `EXIT(None, None, None)` が呼ばれます。

`EXIT` はスタックから取り除かれ、その上の値は順序を維持したまま残されます。加えて、スタックが例外処理中であることを示し、かつ関数呼び出しが `true` 値を返した場合、`END_FINALLY` が例外を再送出することを防ぐため、この情報は削除されます ("zapped")。 (しかし、`non-local goto` は再開されます)

以下の命令コードはすべて引数を必要とします。引数は 2 バイトで、最上位バイトが後になります。

STORE_NAME (*namei*)

`name` = `TOS` に対応します。 *namei* はコードオブジェクトの属性 `co_names` における *name* のインデクスです。コンパイラは可能ならば `STORE_FAST` または `STORE_GLOBAL` を使おうとします。

DELETE_NAME (*namei*)

`del name` に対応します。 *namei* はコードオブジェクトの `co_names` 属性へのインデクスです。

UNPACK_SEQUENCE (*count*)

`TOS` を *count* 個の個別の値にアンパックして、右から左の順にスタックに置きます。

DUP_TOPX (*count*)

count 個の要素を順番を保ちながら複製します。実装上の制限から、*count* は 1 以上 から 5 以下でなければなりません。

STORE_ATTR (*namei*)

`TOS.name` = `TOS1` に対応します。 *namei* は `co_names` における名前のインデクスです。

DELETE_ATTR (*namei*)

`del TOS.name` に対応します。 `co_names` へのインデクスとして *namei* を使います。

STORE_GLOBAL (*namei*)

`STORE_NAME` と同じように動作しますが、`name` をグローバルとして保存します。

DELETE_GLOBAL (*namei*)

`DELETE_NAME` と同じように動作しますが、グローバルの `name` を削除します。

LOAD_CONST (*consti*)

`co_consts[consti]` をスタックにプッシュします。

LOAD_NAME (*namei*)

`co_names[namei]` に関連付けられた値をスタックにプッシュします。

BUILD_TUPLE (*count*)

スタックから *count* 個の要素を消費してタプルを作り出し、できたタプルをスタックにプッシュします。

BUILD_LIST (*count*)

BUILD_TUPLE と同じように動作しますが、リストを作り出します。

BUILD_SET (*count*)

BUILD_TUPLE と同じように動作しますが、set を作り出します。

バージョン 2.7 で追加.

BUILD_MAP (*count*)

スタックに新しい辞書オブジェクトをプッシュします。辞書は *count* 個のエントリを持つサイズに設定されます。

LOAD_ATTR (*namei*)

TOS を `getattr(TOS, co_names[namei])` と入れ替えます。

COMPARE_OP (*opname*)

ブール命令を実行します。命令名は `cmp_op[opname]` にあります。

IMPORT_NAME (*namei*)

モジュール `co_names[namei]` をインポートします。TOS と TOS1 がポップされ、`__import__()` の *fromlist* と *level* 引数になります。モジュールオブジェクトはスタックへプッシュされます。現在の名前空間は影響されません: 適切な import 文のためには、後続の **STORE_FAST** 命令が名前空間を変更します。

IMPORT_FROM (*namei*)

TOS にあるモジュールから属性 `co_names[namei]` をロードします。作成されたオブジェクトはスタックにプッシュされ、後続の **STORE_FAST** 命令によって保存されます。

JUMP_FORWARD (*delta*)

バイトコードカウンタを *delta* だけ増加させます。

POP_JUMP_IF_TRUE (*target*)

TOS が真ならば、バイトコードカウンタを *target* に設定します。TOS はポップされます。

POP_JUMP_IF_FALSE (*target*)

TOS が偽ならば、バイトコードカウンタを *target* に設定します。TOS はポップされます。

JUMP_IF_TRUE_OR_POP (*target*)

TOS が真ならば、バイトコードカウンタを *target* に設定し、TOS はスタックに残されます。そうでない (TOS が偽) なら、TOS はポップされます。

JUMP_IF_FALSE_OR_POP (*target*)

TOS が偽ならば、バイトコードカウンタを *target* に設定し、TOS はスタックに残されます。そうでない (TOS が真) なら、TOS はポップされます。

JUMP_ABSOLUTE (*target*)

バイトコードカウンタを *target* に設定します。

FOR_ITER (*delta*)

TOS はイテレータです。その `next()` メソッドを呼び出します。新しい値が `yield` された場合は、それをスタックにプッシュします (イテレータはその下に残されます)。イテレータの呼び出しで要素が尽きたことが示された場合は、TOS がポップされます。そして、バイトコードカウンタが *delta* だけ増やされます。

LOAD_GLOBAL (*namei*)

`co_names[namei]` という名前のグローバルをスタック上にロードします。

SETUP_LOOP (*delta*)

ループのためのブロックをブロックスタックにプッシュします。ブロックは現在の命令から *delta* バイトの大きさを占めます。

SETUP_EXCEPT (*delta*)

try-except 節から try ブロックをブロックスタックにプッシュします。*delta* は最初の except ブロックを指します。

SETUP_FINALLY (*delta*)

try-except 節から try ブロックをブロックスタックにプッシュします。*delta* は finally ブロックを指します。

STORE_MAP ()

key, value のペアを辞書に格納します。key と value をポップする一方、辞書はスタックに残されます。

LOAD_FAST (*var_num*)

ローカルな `co_varnames[var_num]` への参照をスタックにプッシュします。

STORE_FAST (*var_num*)

TOS をローカルな `co_varnames[var_num]` の中に保存します。

DELETE_FAST (*var_num*)

ローカルな `co_varnames[var_num]` を削除します。

LOAD_CLOSURE (*i*)

セルと自由変数の記憶領域のスロット *i* に含まれるセルへの参照をプッシュします。*i* が `co_cellvars` の長さより小さければ、変数の名前は `co_cellvars[i]` です。そうでなければ `co_freevars[i - len(co_cellvars)]` です。

LOAD_DEREF (*i*)

セルと自由変数の記憶領域のスロット *i* に含まれるセルをロードします。セルが持つオブジェクトへの参照をスタックにプッシュします。

STORE_DEREF (*i*)

セルと自由変数の記憶領域のスロット *i* に含まれるセルへ TOS を保存します。

SET_LINENO (*lineno*)

この命令コードは廃止されました。

RAISE_VARARGS (*argc*)

Raises an exception. *argc* indicates the number of arguments to the raise statement, ranging from 0 to 3.

The handler will find the traceback as TOS2, the parameter as TOS1, and the exception as TOS.

CALL_FUNCTION (*argc*)

Calls a callable object. The low byte of *argc* indicates the number of positional arguments, the high byte the number of keyword arguments. The stack contains keyword arguments on top (if any), then the positional arguments below that (if any), then the callable object to call below that. Each keyword argument is represented with two values on the stack: the argument's name, and its value, with the argument's value above the name on the stack. The positional arguments are pushed in the order that they are passed in to the callable object, with the right-most positional argument on top. `CALL_FUNCTION` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

MAKE_FUNCTION (*argc*)

新しい関数オブジェクトをスタックにプッシュします。TOS は関数に関連付けられたコードです。関数オブジェクトは TOS の下にある *argc* デフォルトパラメタをもつように定義されます。

MAKE_CLOSURE (*argc*)

新しい関数オブジェクトを作り出し、その *func_closure* スロットを設定し、スタックにプッシュします。TOS は関数に関連付けられたコードで、TOS1 はクロージャの自由変数に対するセルを格納したタプルです。関数はセルの前にある *argc* デフォルトパラメタも持っています。

BUILD_SLICE (*argc*)

スライスオブジェクトをスタックにプッシュします。*argc* は 2 あるいは 3 でなければなりません。2 ならば `slice(TOS1, TOS)` がプッシュされます。3 ならば `slice(TOS2, TOS1, TOS)` がプッシュされます。これ以上の情報については、`slice()` 組み込み関数を参照してください。

EXTENDED_ARG (*ext*)

デフォルトの 2 バイトに収まりきらない大きな引数を持つあらゆる命令コードの前に置かれます。*ext* は追加の 2 バイトを保持し、後続の命令コードの引数と組み合わせられます。それらは 4 バイト引数を構成し、*ext* はその最上位バイトです。

CALL_FUNCTION_VAR (*argc*)

Calls a callable object, similarly to `CALL_FUNCTION`. *argc* represents the number of keyword and positional arguments, identically to `CALL_FUNCTION`. The top of the stack contains an iterable object containing additional positional arguments. Below that are keyword arguments (if any), positional arguments (if any) and a callable object, identically to `CALL_FUNCTION`. Before the callable object is called, the iterable object is "unpacked" and its contents are appended to the positional arguments passed in. The iterable object is ignored when computing the value of *argc*.

CALL_FUNCTION_KW (*argc*)

Calls a callable object, similarly to `CALL_FUNCTION`. *argc* represents the number of keyword and positional arguments, identically to `CALL_FUNCTION`. The top of the stack contains a mapping object containing additional keyword arguments. Below that are keyword arguments (if any), positional arguments (if any) and a callable object, identically to `CALL_FUNCTION`. Before the callable is called, the mapping object at the top of the stack is "unpacked" and its contents are appended to the keyword arguments passed in. The mapping object at the top of the stack is ignored when computing the value of *argc*.

CALL_FUNCTION_VAR_KW (*argc*)

Calls a callable object, similarly to `CALL_FUNCTION_VAR` and `CALL_FUNCTION_KW`. `argc` represents the number of keyword and positional arguments, identically to `CALL_FUNCTION`. The top of the stack contains a mapping object, as per `CALL_FUNCTION_KW`. Below that is an iterable object, as per `CALL_FUNCTION_VAR`. Below that are keyword arguments (if any), positional arguments (if any) and a callable object, identically to `CALL_FUNCTION`. Before the callable is called, the mapping object and iterable object are each “unpacked” and their contents passed in as keyword and positional arguments respectively, identically to `CALL_FUNCTION_VAR` and `CALL_FUNCTION_KW`. The mapping object and iterable object are both ignored when computing the value of `argc`.

HAVE_ARGUMENT()

これは実際の命令コードではありません。引数を取らない命令コード `< HAVE_ARGUMENT` と、引数を取る命令コード `>= HAVE_ARGUMENT` の分割行を表します。

32.13 `pickletools` — `pickle` 開発者のためのツール群

バージョン 2.3 で追加.

ソースコード: `Lib/pickletools.py`

このモジュールには、`pickle` モジュールの詳細に関わる様々な定数や実装に関する長大なコメント、そして `pickle` 化されたデータを解析する上で有用な関数をいくつか定義しています。このモジュールの内容は `pickle`, `cPickle` の実装に関わっている Python コア開発者にとって有用なものです; 普通の `pickle` 利用者にとっては、`pickletools` モジュールはおそらく関係ないものでしょう。

`pickletools.dis(pickle, out=None, memo=None, indentlevel=4)`

`pickle` の抽象的な逆アセンブリを file-like オブジェクト `out` (デフォルトは `sys.stdout`) に出力します。 `pickle` は文字列または file-like オブジェクトです。 `memo` は Python の辞書で、 `pickle` のメモとして使われます; これは、 `pickle` 処理を行う 1 つのオブジェクトが、複数の `pickle` にわたって逆アセンブルを行うために使われます。ストリーム上の `MARK` 命令コードが示す後続のレベルは、 `indentlevel` 個の空白でインデントされます。

`pickletools.genops(pickle)`

`pickle` 内の全ての opcode を取り出すイテレータ (*iterator*) を返します。このイテレータは (`opcode`, `arg`, `pos`) の三つ組みからなる配列を返します。 `opcode` は `OpcodeInfo` クラスのインスタンスのクラスです。 `arg` は `opcode` の引数としてデコードされた Python オブジェクトの値です。 `pos` は `opcode` の場所を表す値です。 `pickle` は文字列でもファイル類似オブジェクトでもかまいません。

`pickletools.optimize(picklestring)`

使われていない `PUT` 命令コードを除去した上で、その新しい `pickle` 文字列を返します。最適化された `pickle` は、長さがより短く、転送時間がより少なく、必要とするストレージ領域がより狭く、`unpickle` 化がより効率的になります。

バージョン 2.6 で追加.

第 33 章

Python コンパイラパッケージ

バージョン 2.6 で非推奨: `compiler` モジュールは Python 3 で削除されました。

Python コンパイラパッケージは Python のソースコードを分析したり Python バイトコードを生成するためのツールです。compiler は Python のソースコードから抽象的な構文木を生成し、その構文木から Python バイトコード (*bytecode*) を生成するライブラリをそなえています。

`compiler` パッケージは、Python で書かれた Python ソースコードからバイトコードへの変換プログラムです。これは組み込みの構文解析器と標準ライブラリの `parser` モジュールを使って、具象構文木を生成します。この構文木から抽象構文木 AST (Abstract Syntax Tree) が生成され、その後 Python バイトコードが得られます。

このパッケージの機能は、Python インタプリタに内蔵されている組み込みのコンパイラがすべて含んでいるものです。これはその機能と正確に同じものになるよう意図してつくられています。なぜ同じことをするコンパイラをもうひとつ作る必要があるのでしょうか? このパッケージはいろいろな目的に使うことができます。これは組み込みのコンパイラよりも簡単に変更できますし、これが生成する AST は Python ソースコードを解析するのに有用です。

この章では `compiler` パッケージのいろいろなコンポーネントがどのように動作するのかを説明します。そのため説明はリファレンスマニュアル的なものと、チュートリアル的な要素がまざったものになっています。

33.1 基本的なインターフェイス

このパッケージのトップレベルでは 4 つの関数が定義されています。 `compiler` モジュールを import すると、これらの関数およびこのパッケージに含まれている一連のモジュールが使用可能になります。

`compiler.parse(buf)`

`buf` 中の Python ソースコードから得られた抽象構文木 AST を返します。ソースコード中にエラーがある場合、この関数は `SyntaxError` を発生させます。返り値は `compiler.ast.Module` インスタンスであり、この中に構文木が格納されています。

`compiler.parseFile(path)`

`path` で指定されたファイル中の Python ソースコードから得られた抽象構文木 AST を返します。これは `parse(open(path).read())` と等価な働きをします。

`compiler.walk(ast, visitor[, verbose])`

ast に格納された抽象構文木の各ノードを先行順序 (pre-order) でたどっていきます。各ノードごとに *visitor* インスタンスの該当するメソッドが呼ばれます。

`compiler.compile(source, filename, mode, flags=None, dont_inherit=None)`

文字列 *source*、Python モジュール、文あるいは式を `exec` 文あるいは `eval()` 関数で実行可能なバイトコードオブジェクトにコンパイルします。この関数は組み込みの `compile()` 関数を置き換えるものです。

filename は実行時のエラーメッセージに使用されます。

mode は、モジュールをコンパイルする場合は `'exec'`、(対話的に実行される) 単一の文をコンパイルする場合は `'single'`、式をコンパイルする場合には `'eval'` を渡します。

引数 *flags* および *dont_inherit* は将来的に使用される文に影響しますが、いまのところはサポートされていません。

`compiler.compileFile(source)`

ファイル *source* をコンパイルし、`.pyc` ファイルを生成します。

`compiler` パッケージは以下のモジュールを含んでいます: `ast`, `consts`, `future`, `misc`, `pyassem`, `pycodegen`, `symbols`, `transformer`, そして `visitor`。

33.2 制限

`compiler` パッケージにはエラーチェックにいくつか問題が存在します。構文エラーはインタプリタの2つの別々のフェーズによって認識されます。ひとつはインタプリタのパーザによって認識されるもので、もうひとつはコンパイラによって認識されるものです。`compiler` パッケージはインタプリタのパーザに依存しているので、最初の段階のエラーチェックは労せずして実現できています。しかしその次の段階は、実装されてはいますが、その実装は不完全です。たとえば `compiler` パッケージは引数に同じ名前が2度以上出てきてもエラーを出しません: `def f(x, x): ...`

`compiler` の将来のバージョンでは、これらの問題は修正される予定です。

33.3 Python 抽象構文

`compiler.ast` モジュールは Python の抽象構文木 AST を定義します。AST では各ノードがそれぞれの構文要素をあらわします。木の根は `Module` オブジェクトです。

抽象構文木 AST オブジェクトは、パーズされた Python ソースコードに対する高水準のインターフェイスを提供します。Python インタプリタにおける `parser` モジュールとコンパイラは C で書かれおり、具象構文木を使っています。具象構文木は Python のパーザ中で使われている構文と密接に関連しています。ひとつの要素に単一のノードを割り当てる代わりに、ここでは Python の優先順位に従って、何層にもわたるネストしたノードがしばしば使われています。

抽象構文木 AST は、`compiler.transformer` (変換器) モジュールによって生成されます。`transformer`

は組み込みの Python パーザに依存しており、これを使って具象構文木をまず生成します。つぎにそこから抽象構文木 AST を生成します。

`transformer` モジュールは、実験的な Python-to-C コンパイラ用に Greg Stein と Bill Tutt によって作られました。現行のバージョンではいくつかの修正と改良がなされていますが、抽象構文木 AST と `transformer` の基本的な構造は Stein と Tutt によるものです。

33.3.1 AST ノード

`compiler.ast` モジュールは、各ノードのタイプとその要素を記述したテキストファイルからつくられます。各ノードのタイプはクラスとして表現され、そのクラスは抽象基底クラス `compiler.ast.Node` を継承し子ノードの名前属性を定義しています。

class `compiler.ast.Node`

`Node` インスタンスはパーザジェネレータによって自動的に作成されます。ある特定の `Node` インスタンスに対する推奨されるインターフェイスとは、子ノードにアクセスするために `public` な属性を使うことです。 `public` な属性は単一のノード、あるいは一連のノードのシーケンスに束縛されているかもしれませんが、これは `Node` のタイプによって違います。たとえば `Class` ノードの `bases` 属性は基底クラスのノードのリストに束縛されており、 `doc` 属性は単一のノードのみに束縛されている、といった具合です。

各 `Node` インスタンスは `lineno` 属性をもっており、これは `None` かもしれません。 XXX どういったノードが使用可能な `lineno` をもっているかの規則は定かではない。

全ての `Node` オブジェクトは以下のメソッドを提供します:

getChildren()

子ノードと子オブジェクトを、これらが出てきた順で、平らなリスト形式にして返します。とくにノードの順序は、Python 文法中に現れるものと同じになっています。すべての子が `Node` インスタンスなわけではありません。たとえば関数名やクラス名といったものは、ただの文字列として表されます。

getChildNodes()

子ノードをこれらが出てきた順で平らなリスト形式にして返します。このメソッドは `getChildren()` に似ていますが、 `Node` インスタンスしか返さないという点で異なります。

`Node` クラスの一般的な構造を説明するため、以下に 2 つの例を示します。 `while` 文は以下のような文法規則により定義されています:

```
while_stmt:      "while" expression ":" suite
                ["else" ":" suite]
```

While ノードは 3 つの属性をもっています: `test`, `body` および `else_` です。(ある属性にふさわしい名前が Python の予約語としてすでに使われているとき、その名前を属性名にすることはできません。そのため、ここでは名前が正規のものとして受けつけられるようにアンダースコアを後につけてあります、そのため `else_` は `else` のかわりです。)

`if` 文はもっとこみ入っています。なぜならこれはいくつもの条件判定を含む可能性があるからです。

```
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
```

If ノードでは、`tests` および `else_` の 2 つだけの属性が定義されています。 `tests` 属性には条件式とその後の動作のタプルがリスト形式で入っています。おのおのの `if/elif` 節ごとに 1 タプルです。各タプルの最初の要素は条件式で、2 番目の要素はもしその式が真ならば実行されるコードをふくんだ `Stmt` ノードになっています。

If の `getChildren()` メソッドは、子ノードの平らなリストを返します。 `if/elif` 節が 3 つあって `else` 節がない場合なら、`getChildren()` は 6 要素のリストを返すでしょう: 最初の条件式、最初の `Stmt`、2 番目の条件式...といった具合です。

以下の表は `compiler.ast` で定義されている `Node` サブクラスと、それらのインスタンスに対して使用可能なパブリックな属性です。ほとんどの属性の値自体は `Node` インスタンスか、インスタンスのリストです。この値がインスタンス型以外の場合、その型は備考の中で記されています。これら属性の順序は、`getChildren()` および `getChildNodes()` が返す順です。

ノード型	属性	値
Add	<code>left</code>	左オペランド
	<code>right</code>	右オペランド
And	<code>nodes</code>	オペランドのリスト
AssAttr		代入のターゲットとなる属性
	<code>expr</code>	ドットの左側の式
	<code>attrname</code>	属性名, 文字列
	<code>flags</code>	XXX
AssList	<code>nodes</code>	代入先のリスト要素のリスト
AssName	<code>name</code>	代入先の名前
	<code>flags</code>	XXX
AssTuple	<code>nodes</code>	代入先のタプル要素のリスト
Assert	<code>test</code>	テストされる式
	<code>fail</code>	<code>AssertionError</code> の値
Assign	<code>nodes</code>	代入ターゲットのリスト、等号ごとに一つ
	<code>expr</code>	代入される値
AugAssign	<code>node</code>	
	<code>op</code>	
	<code>expr</code>	
Backquote	<code>expr</code>	
Bitand	<code>nodes</code>	
Bitor	<code>nodes</code>	
Bitxor	<code>nodes</code>	
Break		
CallFunc	<code>node</code>	呼び出される式

次のページに続く

表 1 – 前のページからの続き

ノード型	属性	値
	args	引数のリスト
	star_args	拡張 *-引数の値
	dstar_args	拡張 **-引数の値
Class	name	クラス名, 文字列
	bases	基底クラスのリスト
	doc	ドキュメント文字列, 文字列または None
	<i>code</i>	クラス文の本体
Compare	expr	
	ops	
Const	value	
Continue		
Decorators	nodes	関数デコレータ式のリスト
Dict	items	
Discard	expr	
Div	left	
	right	
<i>Ellipsis</i>		
Expression	node	
Exec	expr	
	<i>locals</i>	
	<i>globals</i>	
FloorDiv	left	
	right	
For	assign	
	list	
	body	
	else_	
From	modname	
	names	
Function	decorators	Decorators または None
	name	def に使われた名前, 文字列
	argnames	引数名の文字列としてのリスト
	defaults	デフォルト値のリスト
	flags	xxx
	doc	ドキュメント文字列, 文字列または None
	<i>code</i>	関数の本体
GenExpr	<i>code</i>	
GenExprFor	assign	
	<i>iter</i>	
	ifs	

次のページに続く

表 1 – 前のページからの続き

ノード型	属性	値
GenExprIf	<i>test</i>	
GenExprInner	expr	
	quals	
Getattr	expr	
	attrname	
Global	names	
If	tests	
	else_	
Import	names	
Invert	expr	
Keyword	name	
	expr	
Lambda	argnames	
	defaults	
	flags	
	<i>code</i>	
LeftShift	left	
	right	
List	nodes	
ListComp	expr	
	quals	
ListCompFor	assign	
	list	
	ifs	
ListCompIf	<i>test</i>	
Mod	left	
	right	
Module	doc	ドキュメント文字列, 文字列または None
	node	モジュールの本体, Stmt
Mul	left	
	right	
Name	name	
Not	expr	
Or	nodes	
Pass		
Power	left	
	right	
Print	nodes	
	dest	
Printnl	nodes	

次のページに続く

表 1 – 前のページからの続き

ノード型	属性	値
	dest	
Raise	expr1	
	expr2	
	expr3	
Return	value	
RightShift	left	
	right	
Slice	expr	
	flags	
	lower	
	upper	
Sliceobj	nodes	文のリスト
Stmt	nodes	
Sub	left	
	right	
Subscript	expr	
	flags	
	subs	
TryExcept	body	
	handlers	
	else_	
TryFinally	body	
	final	
Tuple	nodes	
UnaryAdd	expr	
UnarySub	expr	
While	<i>test</i>	
	body	
	else_	
With	expr	
	<i>vars</i>	
	body	
Yield	value	

33.3.2 代入ノード

代入をあらわすのに使われる一群のノードが存在します。ソースコードにおけるそれぞれの代入文は、抽象構文木 AST では単一のノード `Assign` になっています。 `nodes` 属性は各代入の対象にたいするノードのリストです。これが必要なのは、たとえば `a = b = 2` のように代入が連鎖的に起こるためです。このリスト中における各 *Node* は、次のうちどれかのクラスになります: `AssAttr`, `AssList`, `AssName` または `AssTuple`

。

代入対象の各ノードには代入されるオブジェクトの種類が記録されています。AssName は `a = 1` などの単純な変数名、AssAttr は `a.x = 1` などの属性に対する代入、AssList および AssTuple はそれぞれ、`a, b, c = a_tuple` などのようなリストとタプルの展開をあらわします。

代入対象ノードはまた、そのノードが代入で使われるのか、それとも削除文で使われるのかをあらわす属性 flags も持っています。AssName は `del x` などのような削除文をあらわすのにも使われます。

ある式がいくつかの属性への参照をふくんでいるときは、代入あるいは削除文はただひとつだけの AssAttr ノードをもちます – 最終的な属性への参照としてです。それ以外の属性への参照は AssAttr インスタンスの `expr` 属性にある Getattr ノードによってあらわされます。

33.3.3 例

この節では、Python ソースコードに対する抽象構文木 AST のかんたんな例をいくつかご紹介します。これらの例では `parse()` 関数をどうやって使うか、AST の repr 表現はどんなふうになっているか、そしてある AST ノードの属性にアクセスするにはどうするかを説明します。

最初のモジュールでは単一の関数を定義しています。かりにこれは `/tmp/doublelib.py` に格納されていると仮定しましょう。

```
"""This is an example module.

This is the docstring.
"""

def double(x):
    "Return twice the argument"
    return x * 2
```

以下の対話的インタプリタのセッションでは、見やすさのため長い AST の repr を整形しなおしてあります。AST の repr では qualify されていないクラス名が使われています。repr 表現からインスタンスを作成したい場合は、`compiler.ast` モジュールからそれらのクラス名を import しなければなりません。

```
>>> import compiler
>>> mod = compiler.parseFile("doublelib.py")
>>> mod
Module('This is an example module.\n\nThis is the docstring.\n',
      Stmt([Function(None, 'double', ['x'], [], 0,
                    'Return twice the argument',
                    Stmt([Return(Mul((Name('x'), Const(2))))]))]))
>>> from compiler.ast import *
>>> Module('This is an example module.\n\nThis is the docstring.\n',
...      Stmt([Function(None, 'double', ['x'], [], 0,
...                    'Return twice the argument',
...                    Stmt([Return(Mul((Name('x'), Const(2))))]))]))
Module('This is an example module.\n\nThis is the docstring.\n',
      Stmt([Function(None, 'double', ['x'], [], 0,
```

(次のページに続く)

(前のページからの続き)

```

        'Return twice the argument',
        Stmt([Return(Mul((Name('x'), Const(2))))]))))
>>> mod.doc
'This is an example module.\n\nThis is the docstring.\n'
>>> for node in mod.node.nodes:
...     print node
...
Function(None, 'double', ['x'], [], 0, 'Return twice the argument',
        Stmt([Return(Mul((Name('x'), Const(2))))]))
>>> func = mod.node.nodes[0]
>>> func.code
Stmt([Return(Mul((Name('x'), Const(2))))])

```

33.4 Visitor を使って AST をわたり歩く

visitor パターンは... *compiler* パッケージは、Python のイントロスペクション機能を利用して visitor のために必要な大部分のインフラを省略した、visitor パターンの変種を使っています。

visit されるクラスは、visitor を受け入れるようにプログラムされている必要はありません。visitor が必要なのはただそれがとくに興味あるクラスに対して visit メソッドを定義することだけです。それ以外はデフォルトの visit メソッドが処理します。

XXX visitor 用の魔法の visit() メソッド。

```
compiler.visitor.walk(tree, visitor[, verbose])
```

class compiler.visitor.ASTVisitor

ASTVisitor は構文木を正しい順序でわたり歩くようにします。それぞれのノードはまず *preorder()* の呼び出しではじまります。各ノードに対して、これは 'visitNodeType' という名前のメソッドに対する *preorder()* 関数への visitor 引数をチェックします。ここで NodeType の部分はそのノードのクラス名です。たとえば While ノードなら、visitWhile() が呼ばれるわけです。もしそのメソッドが存在している場合、それはそのノードを第一引数として呼び出されます。

ある特定のノード型に対する visitor メソッドでは、その子ノードをどのようにわたり歩くかが制御できます。*ASTVisitor* は visitor に visit メソッドを追加することで、その visitor 引数を修正します；このメソッドは特定の子ノードを訪問するのに使われるでしょう。特定のノード型に対する visitor が存在しない場合、*default()* メソッドが呼び出されます。

ASTVisitor オブジェクトには以下のようなメソッドがあります：

XXX 追加の引数を記述

```
default(node[, ...])
```

```
dispatch(node[, ...])
```

```
preorder(tree, visitor)
```

33.5 バイトコード生成

バイトコード生成器はバイトコードを出力する visitor です。visit メソッドが呼ばれるたびにこれは emit() メソッドを呼び出し、バイトコードを出力します。基本的なバイトコード生成器はモジュール、クラス、および関数のために特殊化されます。アセンブラがこれらの出力された命令を低レベルのバイトコードに変換します。これはコードオブジェクトからなる定数のリスト生成や、分岐のオフセット計算といった処理をおこないます。

第 34 章

各種サービス

この章では、Python のすべてのバージョンで利用可能な各種サービスについて説明します。以下に概要を示します:

34.1 `formatter` — 汎用の出力書式化機構

このモジュールでは、二つのインタフェース定義を提供しており、それらの各インタフェースについて複数の実装を提供しています。`formatter` インタフェースは `htmllib` モジュールの `HTMLParser` クラスで使われており、`writer` インタフェースは `formatter` インタフェースを使う上で必要です。

`formatter` オブジェクトはある抽象化された書式イベントの流れを `writer` オブジェクト上の特定の出力イベントに変換します。`formatter` はいくつかのスタック構造を管理することで、`writer` オブジェクトの様々な属性を変更したり復元したりできるようにしています; このため、`writer` は相対的な変更や "元に戻す" 操作を処理できなくてもかまいません。`writer` の特定のプロパティのうち、`formatter` オブジェクトを介して制御できるのは、水平方向の字揃え、フォント、そして左マージンの字下げです。任意の、非排他的なスタイル設定を `writer` に提供するためのメカニズムも提供されています。さらに、段落分割のように、可逆でない書式化イベントの機能を提供するインタフェースもあります。

`writer` オブジェクトはデバイスインタフェースをカプセル化します。ファイル形式のような抽象デバイスも物理デバイス同様にサポートされています。ここで提供されている実装内容はすべて抽象デバイス上で動作します。デバイスインタフェースは `formatter` オブジェクトが管理しているプロパティを設定し、データを出力端に書き込めるようにします。

34.1.1 `formatter` インタフェース

`formatter` を作成するためのインタフェースは、インスタンス化しようとする個々の `formatter` クラスに依存します。以下で解説するのは、インスタンス化された全ての `formatter` がサポートしなければならないインタフェースです。

モジュールレベルではデータ要素を一つ定義しています:

`formatter.AS_IS`

後に述べる `push_font()` メソッドでフォント指定をする時に使える値です。また、その他の

`push_property()` メソッドの新しい値として使うことができます。AS_IS の値をスタックに置くと、どのプロパティが変更されたかの追跡を行わずに、対応する `pop_property()` メソッドが呼び出されるようになります。

`formatter` インスタンスオブジェクトには以下の属性が定義されています:

`formatter.writer`

`formatter` とやり取りを行う `writer` インスタンスです。

`formatter.end_paragraph(blanklines)`

開かれている段落があれば閉じ、次の段落との間に少なくとも *blanklines* が挿入されるようにします。

`formatter.add_line_break()`

強制改行挿入します。既に強制改行がある場合は挿入しません。論理的な段落は中断しません。

`formatter.add_hor_rule(*args, **kw)`

出力に水平罫線を挿入します。現在の段落に何らかのデータがある場合、強制改行が挿入されますが、論理的な段落は中断しません。引数とキーワードは `writer` の `send_line_break()` メソッドに渡されます。

`formatter.add_flowling_data(data)`

空白を折りたたんで書式化しなければならないデータを提供します。空白の折りたたみでは、直前や直後の `add_flowling_data()` 呼び出しに入っている空白も考慮されます。このメソッドに渡されたデータは出力デバイスで行末の折り返し (word-wrap) されるものと想定されています。出力デバイスでの要求やフォント情報に応じて、`writer` オブジェクトでも何らかの行末折り返しが行われなければならないので注意してください。

`formatter.add_literal_data(data)`

変更を加えずに `writer` に渡さなければならないデータを提供します。改行およびタブを含む空白を *data* の値にしても問題ありません。

`formatter.add_label_data(format, counter)`

現在の左マージン位置の左側に配置されるラベルを挿入します。このラベルは箇条書き、数字つき箇条書きの書式を構築する際に使われます。 *format* の値が文字列の場合、整数の値 *counter* の書式指定として解釈されます。 *format* の値が文字列の場合、整数の値をとる *counter* の書式化指定として解釈されます。書式化された文字列はラベルの値になります; *format* が文字列でない場合、ラベルの値として直接使われます。ラベルの値は `writer` の `send_label_data()` メソッドの唯一の引数として渡されます。非文字列のラベル値をどう解釈するかは関連付けられた `writer` に依存します。

書式化指定は文字列からなり、*counter* の値と合わせてラベルの値を算出するために使われます。書式文字列の各文字はラベル値にコピーされます。このときいくつかの文字は *counter* 値を変換を指すものとして認識されます。特に、文字 '1' はアラビア数字の *counter* 値を表し、'A' と 'a' はそれぞれ大文字および小文字のアルファベットによる *counter* 値を表し、'I' と 'i' はそれぞれ大文字および小文字のローマ数字による *counter* 値を表します。アルファベットおよびローマ数字への変換の際には、*counter* の値はゼロ以上である必要があるので注意してください。

`formatter.flush_softspace()`

以前の `add_flowling_data()` 呼び出しでバッファされている出力待ちの空白を、関連付けられてい

る `writer` オブジェクトに送信します。このメソッドは `writer` オブジェクトに対するあらゆる直接操作の前に呼び出さなければなりません。

`formatter.push_alignment(align)`

新たな字揃え (*alignment*) 設定を字揃えスタックの上にプッシュします。変更を行いたくない場合には *AS-IS* にすることができます。字揃え設定値が以前の設定から変更された場合、`writer` の `new_alignment()` メソッドが *align* の値と共に呼び出されます。

`formatter.pop_alignment()`

以前の字揃え設定を復元します。

`formatter.push_font((size, italic, bold, teletype))`

`writer` オブジェクトのフォントプロパティのうち、一部または全てを変更します。*AS-IS* に設定されていないプロパティは引数で渡された値に設定され、その他の値は現在の設定を維持します。`writer` の `new_font()` メソッドは完全に設定解決されたフォント指定で呼び出されます。

`formatter.pop_font()`

以前のフォント設定を復元します。

`formatter.push_margin(margin)`

左マージンのインデント数を一つ増やし、論理タグ *margin* を新たなインデントに関連付けます。マージンレベルの初期値は 0 です。変更された論理タグの値は真値とならなければなりません; *AS-IS* 以外の偽の値はマージンの変更としては不適切です。

`formatter.pop_margin()`

以前のマージン設定を復元します。

`formatter.push_style(*styles)`

任意のスタイル指定をスタックにプッシュします。全てのスタイルはスタイルスタックに順番にプッシュされます。*AS-IS* 値を含み、スタック全体を表すタプルは `writer` の `new_styles()` メソッドに渡されます。

`formatter.pop_style([n=1])`

`push_style()` に渡された最新 *n* 個のスタイル指定をポップします。*AS-IS* 値を含み、変更されたスタックを表すタプルは `writer` の `new_styles()` メソッドに渡されます。

`formatter.set_spacing(spacing)`

`writer` の割り付けスタイル (*spacing style*) を設定します。

`formatter.assert_line_data([flag=1])`

現在の段落にデータが予期せず追加されたことを `formatter` に知らせます。このメソッドは `writer` を直接操作した際に使わなければなりません。`writer` 操作の結果、出力の末尾が強制改行となった場合、オプションの *flag* 引数を偽に設定することができます。

34.1.2 formatter 実装

このモジュールでは、`formatter` オブジェクトに関して二つの実装を提供しています。ほとんどのアプリケーションではこれらのクラスを変更したりサブクラス化することなく使うことができます。

class `formatter.NullFormatter` (`[writer]`)

何も行わない formatter です。 `writer` を省略すると、 `NullWriter` インスタンスが生成されます。 `NullFormatter` インスタンスは、 `writer` のメソッドを全く呼び出しません。 `writer` へのインタフェースを実装する場合にはこのクラスのインタフェースを継承する必要がありますが、実装を継承する必要は全くありません。

class `formatter.AbstractFormatter` (`writer`)

標準の formatter です。この formatter 実装は広範な `writer` で適用できることが実証されており、ほとんどの状況で直接使うことができます。高機能の WWW ブラウザを実装するために使われたこともあります。

34.1.3 writer インタフェース

`writer` を作成するためのインタフェースは、インスタンス化しようとする個々の `writer` クラスに依存します。以下で解説するのは、インスタンス化された全ての `writer` がサポートしなければならないインタフェースです。ほとんどのアプリケーションでは `AbstractFormatter` クラスを `formatter` として使うことができますが、通常 `writer` はアプリケーション側で与えなければならないので注意してください。

`writer.flush()`

バッファに蓄積されている出力データやデバイス制御イベントをフラッシュします。

`writer.new_alignment` (`align`)

字揃えのスタイルを設定します。 `align` の値は任意のオブジェクトを取りえますが、慣習的な値は文字列または `None` で、 `None` は `writer` の ”好む” 字揃えを使うことを表します。慣習的な `align` の値は `'left'`、`'center'`、`'right'`、および `'justify'` です。

`writer.new_font` (`font`)

フォントスタイルを設定します。 `font` は、デバイスの標準のフォントが使われることを示す `None` か、 (`size`, `italic`, `bold`, `teletype`) の形式をとるタプルになります。 `size` はフォントサイズを示す文字列になります; 特定の文字列やその解釈はアプリケーション側で定義します。 `italic`、`bold`、および `teletype` といった値はブール値で、それらの属性を使うかどうかを指定します。

`writer.new_margin` (`margin`, `level`)

マージンレベルを整数値 `level` に設定し、論理タグ (logical tag) を `margin` に設定します。論理タグの解釈は `writer` の判断に任されます; 論理タグの値に対する唯一の制限は `level` が非ゼロの値の際に偽であってはならないということです。

`writer.new_spacing` (`spacing`)

割り付けスタイル (spacing style) を `spacing` に設定します。

`writer.new_styles` (`styles`)

追加のスタイルを設定します。 `styles` の値は任意の値からなるタプルです; `AS_IS` 値は無視されます。 `styles` タプルはアプリケーションや `writer` の実装上の都合により、集合としても、スタックとしても解釈され得ます。

`writer.send_line_break()`

現在の行を改行します。

`writer.send_paragraph(blankline)`

少なくとも *blankline* 空行分の間隔か、空行そのもので段落を分割します。 *blankline* の値は整数になります。 `writer` の実装では、改行を行う必要がある場合、このメソッドの呼び出しに先立って `send_line_break()` の呼び出しを受ける必要があります; このメソッドには段落の最後の行を閉じる機能は含まれておらず、段落間に垂直スペースを空ける役割しかありません。

`writer.send_hor_rule(*args, **kw)`

水平罫線を出力デバイスに表示します。このメソッドへの引数は全てアプリケーションおよび `writer` 特有のもので、注意して解釈する必要があります。このメソッドの実装では、すでに改行が `send_line_break()` によってなされているものと仮定しています。

`writer.send_flowring_data(data)`

行端が折り返され、必要に応じて再割り付け解析を行った (re-flowed) 文字データを出力します。このメソッドを連続して呼び出す上では、`writer` は複数の空白文字は単一のスペース文字に縮約されていると仮定することがあります。

`writer.send_literal_data(data)`

すでに表示用に書式化された文字データを出力します。これは通常、改行文字で表された改行を保存し、新たに改行を持ち込まないことを意味します。 `send_formatted_data()` インタフェースと違って、データには改行やタブ文字が埋め込まれていてもかまいません。

`writer.send_label_data(data)`

可能ならば、*data* を現在の左マージンの左側に設定します。 *data* の値には制限がありません; 文字列でない値の扱い方はアプリケーションや `writer` に完全に依存します。このメソッドは行の先頭でのみ呼び出されます。

34.1.4 writer 実装

このモジュールでは、3 種類の `writer` オブジェクトインタフェース実装を提供しています。ほとんどのアプリケーションでは、 `NullWriter` から新しい `writer` クラスを派生する必要があるでしょう。

class `formatter.NullWriter`

インタフェース定義だけを提供する `writer` クラスです; どのメソッドも何ら処理を行いません。このクラスは、メソッド実装をまったく継承する必要のない `writer` 全ての基底クラスになります。

class `formatter.AbstractWriter`

この `writer` は `formatter` をデバッグするのに利用できますが、それ以外に利用できるほどのものではありません。各メソッドを呼び出すと、メソッド名と引数を標準出力に印字して呼び出されたことを示します。

class `formatter.DumbWriter(file=None, maxcol=72)`

単純な `writer` クラスで *file* に渡されたファイルオブジェクトか *file* が `None` の場合には標準出力に出力を書き込みます。出力は *maxcol* で指定されたカラム数で単純な行端折り返しが行われます。このクラスは連続した段落を再割り付けするのに適しています。

第 35 章

MS Windows 固有のサービス

この章では、MS Windows プラットフォーム上でのみ利用可能なモジュール群について記述します。

35.1 msilib — Microsoft インストーラーファイルの読み書き

バージョン 2.5 で追加.

`msilib` モジュールは Microsoft インストーラー (.msi) の作成を支援します。このファイルは大抵の場合埋め込まれた「キャビネット」ファイル (.cab) を含むので、CAB ファイル作成用の API も公開されています。現在のところ .cab ファイルの読み出しは現時点では実装されていませんが、.msi データベースの読み出しは可能です。

このパッケージの目的は .msi ファイルにある全てのテーブルへの完全なアクセスの提供であり、提供されているものは非常に低レベルな API です。このパッケージの二つの主要な使用目的は `distutils` の `bdist_msi` コマンドと、Python インストーラーパッケージそれ自体 (と言いつつ現在は別バージョンの `msilib` を使っているのですが) です。

パッケージの内容は大きく四つのパートに分けられます。低レベル CAB ルーチン、低レベル MSI ルーチン、少し高レベルの MSI ルーチン、標準的なテーブル構造、の四つです。

`msilib.FCICreate(cabname, files)`

新しい CAB ファイルを *cabname* という名前で作ります。 *files* はタプルのリストで、それぞれのタプルはディスク上のファイル名と CAB ファイルで付けられるファイル名で構成されなければなりません。

ファイルはリストに現れた順番で CAB ファイルに追加されます。全てのファイルは MSZIP 圧縮アルゴリズムを使って一つの CAB ファイルに追加されます。

MSI 作成の様々なステップに対する Python コールバックは現在公開されていません。

`msilib.UuidCreate()`

新しい一意な識別子の文字列表現を返します。この関数は Windows API の関数 `UuidCreate()` と `UuidToString()` をラップしたものです。

`msilib.OpenDatabase(path, persist)`

`MsiOpenDatabase` を呼び出して新しいデータベースオブジェクトを返します。 *path* は MSI ファイ

ルのファイル名です。 *persist* は五つの定数 `MSIDBOPEN_CREATEDIRECT`, `MSIDBOPEN_CREATE`, `MSIDBOPEN_DIRECT`, `MSIDBOPEN_READONLY`, `MSIDBOPEN_TRANSACT` のどれか一つで、フラグ `MSIDBOPEN_PATCHFILE` を含めても構いません。これらのフラグの意味は Microsoft のドキュメントを参照してください。フラグに従って、既存のデータベースを開いたり新しいデータベースを作成したりします。

`msilib.CreateRecord(count)`

`MSICreateRecord()` を呼び出して新しいレコードオブジェクトを返します。 *count* はレコードのフィールドの数です。

`msilib.init_database(name, schema, ProductName, ProductCode, ProductVersion, Manufacturer)`

name という名前の新しいデータベースを作り、*schema* で初期化し、プロパティ *ProductName*, *ProductCode*, *ProductVersion*, *Manufacturer* をセットして、返します。

schema は `tables` と `_Validation_records` という属性をもったモジュールオブジェクトでなければなりません。大抵の場合、 `msilib.schema` を使うべきです。

データベースはこの関数から返された時点でスキーマとバリデーションレコードだけが収められています。

`msilib.add_data(database, table, records)`

全ての *records* を *database* の *table* テーブルに追加します。

table 引数は MSI スキーマで事前に定義されたテーブルでなければなりません。例えば、 `'Feature'`, `'File'`, `'Component'`, `'Dialog'`, `'Control'`, などです。

records はタプルのリストで、それぞれのタプルにはテーブルのスキーマに従ったレコードの全てのフィールドを含んでいるものでなければなりません。オプションのフィールドには `None` を渡すことができます。

フィールドの値には、整数・長整数・文字列・Binary クラスのインスタンスが使えます。

`class msilib.Binary(filename)`

Binary テーブル中のエントリーを表わします。 `add_data()` を使ってこのクラスのオブジェクトを挿入するときには *filename* という名前のファイルをテーブルに読み込みます。

`msilib.add_tables(database, module)`

module の全てのテーブルの内容を *database* に追加します。 *module* は `tables` という内容が追加されるべき全てのテーブルのリストと、テーブルごとに一つある実際の内容を持っている属性とを含んでいなければなりません。

この関数は典型的にシーケンステーブルをインストールするために使われます。

`msilib.add_stream(database, name, path)`

database の `_Stream` テーブルに、ファイル *path* を *name* というストリーム名で追加します。

`msilib.gen_uuid()`

新しい UUID を、MSI が通常要求する形式 (つまり、中括弧で囲み、16 進数は大文字) で返します。

参考:

FCICreateFile UuidCreate UuidToString

35.1.1 データベースオブジェクト

`Database.OpenView (sql)`

`MSIDatabaseOpenView()` を呼び出して取得したビューオブジェクトを返します。 *sql* は実行される SQL ステートメントです。

`Database.Commit ()`

`MSIDatabaseCommit()` を呼び出して現在のトランザクションで保留されている変更をコミットします。

`Database.GetSummaryInformation (count)`

`MsiGetSummaryInformation()` を呼び出して新しいサマリー情報オブジェクトを返します。 *count* は更新された値の最大数です。

参考:

[MSIDatabaseOpenView](#) [MSIDatabaseCommit](#) [MsiGetSummaryInformation](#)

35.1.2 ビューオブジェクト

`View.Execute (params)`

`MsiViewExecute()` を通してビューに対する SQL 問い合わせを実行します。 *params* が `None` でない場合、クエリ中のパラメータトークンの実際の値を与えるものです。

`View.GetColumnInfo (kind)`

`MsiViewGetColumnInfo()` の呼び出しを通してビューのカラムを説明するレコードを返します。 *kind* は `MSICOLINFO_NAMES` または `MSICOLINFO_TYPES` です。

`View.Fetch ()`

`MsiViewFetch()` の呼び出しを通してクエリの結果レコードを返します。

`View.Modify (kind, data)`

`MsiViewModify()` を呼び出してビューを変更します。 *kind* は `MSIMODIFY_SEEK`, `MSIMODIFY_REFRESH`, `MSIMODIFY_INSERT`, `MSIMODIFY_UPDATE`, `MSIMODIFY_ASSIGN`, `MSIMODIFY_REPLACE`, `MSIMODIFY_MERGE`, `MSIMODIFY_DELETE`, `MSIMODIFY_INSERT_TEMPORARY`, `MSIMODIFY_VALIDATE`, `MSIMODIFY_VALIDATE_NEW`, `MSIMODIFY_VALIDATE_FIELD`, `MSIMODIFY_VALIDATE_DELETE` のいずれかです。

data は新しいデータを表わすレコードでなければなりません。

`View.Close ()`

`MsiViewClose()` を通してビューを閉じます。

参考:

[MsiViewExecute](#) [MsiViewGetColumnInfo](#) [MsiViewFetch](#) [MsiViewModify](#) [MsiViewClose](#)

35.1.3 サマリー情報オブジェクト

`SummaryInformation.GetProperty (field)`

`MsiSummaryInfoGetProperty()` を通してサマリーのプロパティを返します。 *field* はプロパティ名で、定数 `PID_CODEPAGE`, `PID_TITLE`, `PID_SUBJECT`, `PID_AUTHOR`, `PID_KEYWORDS`, `PID_COMMENTS`, `PID_TEMPLATE`, `PID_LASTAUTHOR`, `PID_REVNUMBER`, `PID_LASTPRINTED`, `PID_CREATE_DTM`, `PID_LASTSAVE_DTM`, `PID_PAGECOUNT`, `PID_WORDCOUNT`, `PID_CHARCOUNT`, `PID_APPNAME`, `PID_SECURITY` のいずれかです。

`SummaryInformation.GetPropertyCount ()`

`MsiSummaryInfoGetPropertyCount()` を通してサマリープロパティの個数を返します。

`SummaryInformation.SetProperty (field, value)`

`MsiSummaryInfoSetProperty()` を通してプロパティをセットします。 *field* は `GetProperty()` におけるものと同じ値をとります。 *value* はプロパティの新しい値です。許される値の型は整数と文字列です。

`SummaryInformation.Persist ()`

`MsiSummaryInfoPersist()` を使って変更されたプロパティをサマリー情報ストリームに書き込みます。

参考:

[MsiSummaryInfoGetProperty](#) [MsiSummaryInfoGetPropertyCount](#) [MsiSummaryInfoSetProperty](#) [MsiSummaryInfoPersist](#)

35.1.4 レコードオブジェクト

`Record.GetFieldCount ()`

`MsiRecordGetFieldCount()` を通してレコードのフィールド数を返します。

`Record.GetInteger (field)`

field の値を可能なら整数として返します。 *field* は整数でなければなりません。

`Record.GetString (field)`

field の値を可能なら文字列として返します。 *field* は整数でなければなりません。

`Record.SetString (field, value)`

`MsiRecordSetString()` を通して *field* を *value* にセットします。 *field* は整数、 *value* は文字列でなければなりません。

`Record.SetStream (field, value)`

`MsiRecordSetStream()` を通して *field* を *value* という名のファイルの内容にセットします。 *field* は整数、 *value* は文字列でなければなりません。

`Record.SetInteger (field, value)`

`MsiRecordSetInteger()` を通して *field* を *value* にセットします。 *field* も *value* も整数でなければなりません。

`Record.ClearData()`

`MsiRecordClearData()` を通してレコードの全てのフィールドを 0 にセットします。

参考:

`MsiRecordGetFieldCount` `MsiRecordSetString` `MsiRecordSetStream` `MsiRecordSetInteger` `MsiRecordClear`

35.1.5 エラー

All wrappers around MSI functions raise `MSIError`; the string inside the exception will contain more detail.

35.1.6 CAB オブジェクト

class `msilib.CAB` (*name*)

`CAB` クラスは CAB ファイルを表わすものです。MSI 構築中、ファイルは `Files` テーブルと CAB ファイルとに同時に追加されます。そして、全てのファイルを追加し終えたら、CAB ファイルは書き込まれることが可能になり、MSI ファイルに追加されます。

name は MSI ファイル中の CAB ファイルの名前です。

append (*full, file, logical*)

パス名 *full* のファイルを CAB ファイルに *logical* という名で追加します。*logical* という名が既に存在したならば、新しいファイル名が作られます。

ファイルの CAB ファイル中のインデクスと新しいファイル名を返します。

commit (*database*)

CAB ファイルを作り、MSI ファイルにストリームとして追加し、`Media` テーブルに送り込み、作ったファイルはディスクから削除します。

35.1.7 ディレクトリオブジェクト

class `msilib.Directory` (*database, cab, basedir, physical, logical, default[, componentflags]*)

新しいディレクトリを `Directory` テーブルに作成します。ディレクトリには各時点で現在のコンポーネントがあり、それは `start_component()` を使って明ら様に作成されたかまたは最初にファイルが追加された際に暗黙裡に作成されたものです。ファイルは現在のコンポーネントと cab ファイルに追加されます。ディレクトリを作成するには親ディレクトリオブジェクト (`None` でも可)、物理的ディレクトリへのパス、論理的ディレクトリ名を指定する必要があります。*default* はディレクトリテーブルの `DefaultDir` スロットを指定します。*componentflags* は新しいコンポーネントが得るデフォルトのフラグを指定します。

start_component ([*component[, feature[, flags[, keyfile[, uuid]]]]])*

エントリを `Component` テーブルに追加し、このコンポーネントをこのディレクトリの現在のコンポーネントにします。もしコンポーネント名が与えられなければディレクトリ名が使われます。

feature が与えられなければ、ディレクトリのデフォルトフラグが使われます。*keyfile* が与えられなければ、Component テーブルの KeyPath は null のままになります。

add_file (*file* [, *src* [, *version* [, *language*]]])

ファイルをディレクトリの現在のコンポーネントに追加します。このとき現在のコンポーネントがなければ新しいものを開始します。デフォルトではソースとファイルテーブルのファイル名は同じになります。*src* ファイルが与えられたならば、それは現在のディレクトリから相対的に解釈されます。オプションで *version* と *language* を File テーブルのエントリ用に指定することができます。

glob (*pattern* [, *exclude*])

現在のコンポーネントに glob パターンで指定されたファイルのリストを追加します。個々のファイルを *exclude* リストで除外することができます。

remove_pyc ()

アンインストールの際に .pyc / .pyo を削除します。

参考:

[Directory Table](#) [File Table](#) [Component Table](#) [FeatureComponents Table](#)

35.1.8 機能

class `msilib.Feature` (*database*, *id*, *title*, *desc*, *display* [, *level=1* [, *parent* [, *directory* [, *attributes=0*]]]])

id, *parent.id*, *title*, *desc*, *display*, *level*, *directory*, *attributes* の値を使って、新しいレコードを Feature テーブルに追加します。出来上がったフィーチャーオブジェクトは [Directory](#) の `start_component ()` メソッドに渡すことができます。

set_current ()

このフィーチャーを [msilib](#) の現在のフィーチャーにします。フィーチャーが明ら様に指定されない限り、新しいコンポーネントが自動的にデフォルトのフィーチャーに追加されます。

参考:

[Feature Table](#)

35.1.9 GUI クラス

[msilib](#) は MSI データベースにある GUI テーブルをラップしたいくつかのクラスを提供します。しかし、標準的なユーザーインタフェースは提供されません。ユーザーインタフェースを備えた Python パッケージをインストールする MSI を作成するには `bdist_msi` を使用してください。

class `msilib.Control` (*dlg*, *name*)

ダイアログコントロールの基底クラス。*dlg* はコントロールの属するダイアログオブジェクト、*name* はコントロールの名前です。

event (*event*, *argument* [, *condition=1* [, *ordering*]])

このコントロールの ControlEvent テーブルにエントリを作ります。

mapping (*event, attribute*)

このコントロールの `EventMapping` テーブルにエントリを作ります。

condition (*action, condition*)

このコントロールの `ControlCondition` テーブルにエントリを作ります。

class `msilib.RadioButtonGroup` (*dlg, name, property*)

name という名前のラジオボタンコントロールを作成します。 *property* はラジオボタンが選ばれたときにセットされるインストーラプロパティです。

add (*name, x, y, width, height, text* [, *value*])

グループに *name* という名前で、座標 *x, y* に大きさが *width, height* で *text* というラベルの付いたラジオボタンを追加します。 *value* が省略されると、デフォルトは *name* になります。

class `msilib.Dialog` (*db, name, x, y, w, h, attr, title, first, default, cancel*)

新しい `Dialog` オブジェクトを返します。 `Dialog` テーブルを以下の引数に渡された情報を元に作成します: 座標、ダイアログ属性、タイトル、 `first`・`default`・`cancel` という三つのコントロールに対する名前。

control (*name, type, x, y, width, height, attributes, property, text, control_next, help*)

新しい `Control` オブジェクトを返します。 `Control` テーブルに指定されたパラメータのエントリが作られます。

これは汎用のメソッドで、特定の型に対しては特化したメソッドが提供されています。

text (*name, x, y, width, height, attributes, text*)

`Text` コントロールを追加して返します。

bitmap (*name, x, y, width, height, text*)

`Bitmap` コントロールを追加して返します。

line (*name, x, y, width, height*)

`Line` コントロールを追加して返します。

pushbutton (*name, x, y, width, height, attributes, text, next_control*)

`PushButton` コントロールを追加して返します。

radiogroup (*name, x, y, width, height, attributes, property, text, next_control*)

`RadioButtonGroup` コントロールを追加して返します。

checkbox (*name, x, y, width, height, attributes, property, text, next_control*)

`CheckBox` コントロールを追加して返します。

参考:

[Dialog Table](#) [Control Table](#) [Control Types](#) [ControlCondition Table](#) [ControlEvents Table](#) [EventMapping Table](#) [RadioButton Table](#)

35.1.10 事前に計算されたテーブル

`msilib` はスキーマとテーブル定義だけから成るサブパッケージをいくつか提供しています。現在のところ、これらの定義は MSI バージョン 2.0 に基づいています。

`msilib.schema`

これは MSI 2.0 用の標準 MSI スキーマで、テーブル定義のリストを提供する `tables` 変数と、MSI バリデーション用のデータを提供する `_Validation_records` 変数があります。

`msilib.sequence`

このモジュールは標準シーケンステーブルのテーブル内容を含んでいます。`AdminExecuteSequence`, `AdminUISequence`, `AdvtExecuteSequence`, `InstallExecuteSequence`, `InstallUISequence` が含まれています。

`msilib.text`

このモジュールは標準的なインストーラーのアクションのための `UIText` および `ActionText` テーブルの定義を含んでいます。

35.2 msvcrt — MS VC++ 実行時システムの有用なルーチン群

このモジュールの関数は、Windows プラットフォームの便利な機能のいくつかに対するアクセス機構を提供しています。高レベルモジュールのいくつかは、提供するサービスを Windows で実装するために、これらの関数を使っています。例えば、`getpass` モジュールは関数 `getpass()` を実装するためにこのモジュールの関数を使います。

ここに挙げた関数の詳細なドキュメントについては、プラットフォーム API ドキュメントで見つけることができます。

このモジュールは、通常版とワイド文字列版の両方のコンソール I/O API を実装しています。通常版の API は ASCII 文字列のためのもので、国際化アプリケーションでは利用が制限されます。可能な限りワイド文字列版 API を利用すべきです。

35.2.1 ファイル操作関連

`msvcrt.locking` (*fd*, *mode*, *nbytes*)

C 言語による実行時システムにおけるファイル記述子 *fd* に基づいて、ファイルの一部にロックをかけます。ロックされるファイルの領域は、現在のファイル位置から *nbytes* バイトで、ファイルの末端まで延長することができます。*mode* は以下に列挙する `LK_*` のいずれか一つでなければなりません。一つのファイルの複数の領域を同時にロックすることは可能ですが、領域が重複してはなりません。接続する領域をまとめて指定することはできません; それらの領域は個別にロック解除しなければなりません。

`msvcrt.LK_LOCK`

`msvcrt.LK_RLOCK`

指定されたバイト列にロックをかけます。指定領域がロックできなかった場合、プログラムは 1 秒後に再度ロックを試みます。10 回再試行した後もロックをかけられない場合、`IOError` が送出されます。

`msvcrt.LK_NBLCK`

`msvcrt.LK_NBLCK`

指定されたバイト列にロックをかけます。指定領域がロックできなかった場合、`IOError` が送出されます。

`msvcrt.LK_UNLCK`

指定されたバイト列のロックを解除します。指定領域はあらかじめロックされていなければなりません。

`msvcrt.setmode(fd, flags)`

ファイル記述子 `fd` に対して、行末文字の変換モードを設定します。テキストモードに設定するには、`flags` を `os.O_TEXT` にします; バイナリモードにするには `os.O_BINARY` にします。

`msvcrt.open_osfhandle(handle, flags)`

C 言語による実行時システムにおけるファイル記述子をファイルハンドル `handle` から生成します。`flags` パラメタは `os.O_APPEND`、`os.O_RDONLY`、および `os.O_TEXT` をビット単位で OR したものに なります。返されるファイル記述子は `os.fdopen()` でファイルオブジェクトを生成するために使うことができます。

`msvcrt.get_osfhandle(fd)`

ファイル記述子 `fd` のファイルハンドルを返します。`fd` が認識できない場合、`IOError` を送出します。

35.2.2 コンソール I/O 関連

`msvcrt.kbhit()`

読み出し待ちの打鍵イベントが存在する場合に真を返します。

`msvcrt.getch()`

打鍵を読み取り、読み出された文字を返します。コンソールには何もエコーバックされません。この関数呼び出しは読み出し可能な打鍵がない場合にはブロックしますが、文字を読み出せるようにするために Enter の打鍵を待つ必要はありません。打鍵されたキーが特殊機能キー (function key) である場合、この関数は `'\000'` または `'\xe0'` を返します; キーコードは次に関数を呼び出した際に返されます。この関数で Control-C の打鍵を読み出すことはできません。

`msvcrt.getwch()`

`getch()` のワイド文字列版。Unicode の値を返します。

バージョン 2.6 で追加。

`msvcrt.getche()`

`getch()` に似ていますが、打鍵した字が印字可能な文字の場合エコーバックされます。

`msvcrt.getwche()`

`getche()` のワイド文字列版。Unicode の値を返します。

バージョン 2.6 で追加。

`msvcrt.putch(char)`

キャラクタ *char* をバッファリングを行わないでコンソールに出力します。

`msvcrt.putwch(unicode_char)`

`putch()` のワイド文字列版。Unicode の値を引数に取ります。

バージョン 2.6 で追加。

`msvcrt.ungetch(char)`

キャラクタ *char* をコンソールバッファに “押し戻し (push back)” ます; これにより、押し戻された文字は `getch()` や `getche()` で次に読み出される文字になります。

`msvcrt.ungetwch(unicode_char)`

`ungetch()` のワイド文字列版。Unicode の値を引数に取ります。

バージョン 2.6 で追加。

35.2.3 その多の関数

`msvcrt.heapmin()`

`malloc()` されたヒープ領域を強制的に消去させて、未使用のメモリブロックをオペレーティングシステムに返します。失敗した場合、`IOError` を送出します。

35.3 _winreg — Windows レジストリへのアクセス

注釈: `_winreg` モジュールは、Python 3 では `winreg` にリネームされました。 `2to3` ツールが自動的にソースコードの `import` を修正します。

バージョン 2.0 で追加。

これらの関数は Windows レジストリ API を Python から使えるようにします。プログラマがレジストリハンドルのクローズを失念した場合でも、確実にハンドルがクローズされるようにするために、整数値をレジストリハンドルとして使う代わりに [ハンドルオブジェクト](#) が使われます。

このモジュールでは以下の関数を提供します:

`_winreg.CloseKey(hkey)`

以前開かれたレジストリキーを閉じます。 *hkey* 引数には以前開かれたレジストリキーを特定します。

注釈: このメソッドを使って (または `hkey.Close()` によって) *hkey* が閉じられなかった場合、Python が *hkey* オブジェクトを破壊する際に閉じられます。

`_winreg.ConnectRegistry(computer_name, key)`

他の計算機上にある既定のレジストリハンドル接続を確立し、 [ハンドルオブジェクト](#) を返します。

`computer_name` はリモートコンピュータの名前で、`r"\\computername"` の形式をとります。None の場合、ローカルの計算機が使われます。

`key` は接続したい既定のハンドルです。

戻り値は開かれたキーのハンドルです。関数が失敗した場合、`WindowsError` 例外が送出されます。

`_winreg.CreateKey(key, sub_key)`

特定のキーを生成するか開き、[ハンドルオブジェクト](#) を返します。

`key` はすでに開かれたキーか、既定の `HKEY_* 定数` のうちの一つです。

`sub_key` はこのメソッドが開く、または新規作成するキーの名前です。

`key` が既定のキーの一つなら、`sub_key` は None でかまいません。この場合、返されるハンドルは関数に渡されたのと同じキーハンドルです。

キーがすでに存在する場合、この関数は既に存在するキーを開きます。

戻り値は開かれたキーのハンドルです。関数が失敗した場合、`WindowsError` 例外が送出されます。

`_winreg.CreateKeyEx(key, sub_key[, res[, sam]])`

特定のキーを生成するか開き、[ハンドルオブジェクト](#) を返します。

`key` はすでに開かれたキーか、既定の `HKEY_* 定数` のうちの一つです。

`sub_key` はこのメソッドが開く、または新規作成するキーの名前です。

`res` は予約された整数で、0 でなくてはなりません。デフォルト値は 0 です。

`sam` は、`key` に対して想定するセキュリティアクセスを示すアクセスマスクを指定します。デフォルトは `KEY_ALL_ACCESS` です。利用可能な値については[アクセス権](#)を参照してください。

`key` が既定のキーの一つなら、`sub_key` は None でかまいません。この場合、返されるハンドルは関数に渡されたのと同じキーハンドルです。

キーがすでに存在する場合、この関数は既に存在するキーを開きます。

戻り値は開かれたキーのハンドルです。関数が失敗した場合、`WindowsError` 例外が送出されます。

バージョン 2.7 で追加.

`_winreg.DeleteKey(key, sub_key)`

特定のキーを削除します。

`key` はすでに開かれたキーか、既定の `HKEY_* 定数` のうちの一つです。

`sub_key` は文字列で、`key` パラメタによって特定されたキーのサブキーでなければなりません。この値は None であってはならず、キーはサブキーを持っていないてもかまいません。

このメソッドはサブキーをもつキーを削除することはできません。

このメソッドの実行が成功すると、キー全体が、その値すべてを含めて削除されます。このメソッドが失敗した場合、`WindowsError` 例外が送出されます。

`_winreg.DeleteKeyEx(key, sub_key[, sam[, res]])`

特定のキーを削除します。

注釈: The `DeleteKeyEx()` 関数は、`RegDeleteKeyEx` Windows API 関数で実装されています。これは Windows の 64-bit バージョンに特有です。 [RegDeleteKeyEx documentation](#) を参照してください。

`key` はすでに開かれたキーか、既定の `HKEY_*` 定数のうちの一つです。

`sub_key` は `key` 引数によって指定された `key` の subkey でなければなりません。この値は `None` であってはなりません。また、`key` は subkey を持たないかもしれません。

`res` は予約された整数で、0 でなくてはなりません。デフォルト値は 0 です。

`sam` は、`key` に対して想定するセキュリティアクセスを示すアクセスマスクを指定します。デフォルトは `KEY_WOW64_64KEY` です。利用可能な値については [アクセス権](#) を参照してください。

このメソッドはサブキーをもつキーを削除することはできません。

このメソッドの実行が成功すると、キー全体が、その値すべてを含めて削除されます。このメソッドが失敗した場合、`WindowsError` 例外が送出されます。

サポートされていない Windows バージョンでは、`NotImplementedError` 例外が発生させます。

バージョン 2.7 で追加。

`_winreg.DeleteValue(key, value)`

レジストリキーから指定された名前付きの値を削除します。

`key` はすでに開かれたキーか、既定の `HKEY_*` 定数のうちの一つです。

`value` は削除したい値を指定するための文字列です。

`_winreg.EnumKey(key, index)`

開かれているレジストリキーのサブキーを列挙し、文字列で返します。

`key` はすでに開かれたキーか、既定の `HKEY_*` 定数のうちの一つです。

`index` は整数値で、取得するキーのインデックスを特定します。

この関数は呼び出されるたびに一つのサブキーの名前を取得します。この関数は通常、これ以上値がないことを示す `WindowsError` 例外が送出されるまで繰り返し呼び出されます。

`_winreg.EnumValue(key, index)`

開かれているレジストリキーの値を列挙し、タプルで返します。

`key` はすでに開かれたキーか、既定の `HKEY_*` 定数のうちの一つです。

`index` は整数値で、取得する値のインデックスを特定します。

この関数は呼び出されるたびに一つのサブキーの名前を取得します。この関数は通常、これ以上値がないことを示す `WindowsError` 例外が送出されるまで繰り返し呼び出されます。

結果は 3 要素のタプルになります:

インデックス	意味
0	値の名前を特定する文字列
1	値のデータを保持するためのオブジェクトで、その型は背後のレジストリ型に依存します
2	値のデータ型を特定する整数です (<code>SetValueEx()</code> のドキュメント内のテーブルを参照してください)

`_winreg.ExpandEnvironmentStrings(unicode)`

`REG_EXPAND_SZ` のように、環境変数プレースホルダ `%NAME%` を Unicode 文字列で展開します:

```
>>> ExpandEnvironmentStrings(u"%windir%")
u"C:\\Windows"
```

バージョン 2.6 で追加.

`_winreg.FlushKey(key)`

キーのすべての属性をレジストリに書き込みます。

`key` はすでに開かれたキーか、既定の `HKEY_*` 定数のうちの一つです。

キーを変更するために `FlushKey()` を呼ぶ必要はありません。レジストリの変更は怠惰なフラッシュ機構 (lazy flusher) を使ってフラッシュされます。また、システムの遮断時にもディスクにフラッシュされます。 `CloseKey()` と違って、 `FlushKey()` メソッドはレジストリに全てのデータを書き終えたときにのみ返ります。アプリケーションは、レジストリへの変更を絶対に確実にディスク上に反映させる必要がある場合にのみ、 `FlushKey()` を呼ぶべきです。

注釈: `FlushKey()` を呼び出す必要があるかどうか分からない場合、おそらくその必要はありません。

`_winreg.LoadKey(key, sub_key, file_name)`

指定されたキーの下にサブキーを生成し、サブキーに指定されたファイルのレジストリ情報を記録します。

`key` は `ConnectRegistry()` が返したハンドルか、定数 `HKEY_USERS` と `HKEY_LOCAL_MACHINE` のどちらかです。

`sub_key` は記録先のサブキーを指定する文字列です。

`file_name` はレジストリデータを読み出すためのファイル名です。このファイルは `SaveKey()` 関数で生成されたファイルでなくてはなりません。ファイル割り当てテーブル (FAT) ファイルシステム下では、ファイル名は拡張子を持っていてはなりません。

この関数を呼び出しているプロセスが `SE_RESTORE_PRIVILEGE` 特権を持たない場合には

`LoadKey()` は失敗します。この特権はファイル許可とは違うので注意してください - 詳細は [RegLoadKey documentation](#) を参照してください。

`key` が `ConnectRegistry()` によって返されたハンドルの場合、`file_name` に指定されたパスは遠隔計算機に対する相対パス名になります。

`_winreg.OpenKey(key, sub_key[, res[, sam]])`

指定されたキーを開き、[ハンドルオブジェクト](#) を返します。

`key` はすでに開かれたキーか、既定の [HKEY_* 定数](#) のうちの一つです。

`sub_key` は開きたいサブキーを特定する文字列です。

`res` は予約されている整数値で、ゼロでなくてはなりません。デフォルトはゼロです。

`sam` は必要なキーへのセキュリティアクセスを記述する、アクセスマスクを指定する整数です。標準の値は [KEY_READ](#) です。その他の利用できる値については [アクセス権限](#) を参照してください。

指定されたキーへの新しいハンドルが返されます。

この関数が失敗すると、[WindowsError](#) が送出されます。

`_winreg.OpenKeyEx()`

`OpenKeyEx()` の機能は `OpenKey()` を標準の引数で使うことで提供されています。

`_winreg.QueryInfoKey(key)`

キーに関数情報をタプルとして返します。

`key` はすでに開かれたキーか、既定の [HKEY_* 定数](#) のうちの一つです。

結果は 3 要素のタプルになります:

インデックス	意味
0	結果は以下の 3 要素からなるタプルです。
1	このキーが持つサブキーの数を表す整数。
2	最後のキーの変更が (あれば) いつだったかを表す長整数で、1601 年 1 月 1 日からの 100 ナノ秒単位で数えたもの。

`_winreg.QueryValue(key, sub_key)`

キーに対する、名前付けられていない値を文字列で取得します。

`key` はすでに開かれたキーか、既定の [HKEY_* 定数](#) のうちの一つです。

`sub_key` は値が関連付けられているサブキーの名前を保持する文字列です。この引数が `None` または空文字列の場合、この関数は `key` で特定されるキーに対して `SetValue()` メソッドで設定された値を取得します。

レジストリ中の値は名前、型、およびデータから構成されています。このメソッドはあるキーのデータ中で、名前 `NULL` をもつ最初の値を取得します。しかし背後の API 呼び出しは型情報を返しません。なので、可能ならいつでも `QueryValueEx()` を使うべきです。

`_winreg.QueryValueEx` (*key*, *value_name*)

開かれたレジストリキーに関連付けられている、指定した名前の値に対して、型およびデータを取得します。

key はすでに開かれたキーか、既定の [HKEY_* 定数](#) のうちの一つです。

value_name は要求する値を指定する文字列です。

結果は 2 つの要素からなるタプルです:

インデックス	意味
0	レジストリ要素の値。
1	この値のレジストリ型を表す整数。 (SetValueEx() のドキュメント内のテーブルを参照してください。)

`_winreg.SaveKey` (*key*, *file_name*)

指定されたキーと、そのサブキー全てを指定したファイルに保存します。

key はすでに開かれたキーか、既定の [HKEY_* 定数](#) のうちの一つです。

file_name はレジストリデータを保存するファイルの名前です、このファイルはすでに存在してはいけません。このファイル名が拡張子を含んでいる場合、[LoadKey\(\)](#) メソッドは、FAT ファイルシステムを使うことができません。

key が遠隔の計算機上にあるキーを表す場合、*file_name* で記述されているパスは遠隔の計算機に対して相対的なパスになります。このメソッドの呼び出し側は `SeBackupPrivilege` セキュリティ特権を保有していなければなりません。この特権はファイルパーミッションとは異なります - 詳細は [Conflicts Between User Rights and Permissions documentation](#) を参照してください。

この関数は *security_attributes* を NULL にして API に渡します。

`_winreg.SetValue` (*key*, *sub_key*, *type*, *value*)

値を指定したキーに関連付けます。

key はすでに開かれたキーか、既定の [HKEY_* 定数](#) のうちの一つです。

sub_key は値が関連付けられているサブキーの名前を表す文字列です。

type はデータの型を指定する整数です。現状では、この値は [REG_SZ](#) でなければならず、これは文字列だけがサポートされていることを示します。他のデータ型をサポートするには [SetValueEx\(\)](#) を使ってください。

value は新たな値を指定する文字列です。

sub_key 引数で指定されたキーが存在しなければ、`SetValue` 関数で生成されます。

値の長さは利用可能なメモリによって制限されます。(2048 バイト以上の) 長い値はファイルに保存して、そのファイル名を設定レジストリに保存するべきです。そうすればレジストリを効率的に動作させる役に立ちます。

`key` 引数に指定されたキーは `KEY_SET_VALUE` アクセスで開かれていなければなりません。

`_winreg.SetValueEx(key, value_name, reserved, type, value)`

開かれたレジストリキーの値フィールドにデータを記録します。

`key` はすでに開かれたキーか、既定の `HKEY_*` 定数のうちの一つです。

`value_name` は値が関連付けられているサブキーの名前を表す文字列です。

`type` はデータの型を指定する整数です。利用できる型については [値の型](#) を参照してください。

`reserved` は何もしません - API には常にゼロが渡されます。

`value` は新たな値を指定する文字列です。

このメソッドではまた、指定されたキーに対して、さらに別の値や型情報を設定することができます。
`key` 引数で指定されたキーは `KEY_SET_VALUE` アクセスで開かれていなければなりません。

キーを開くには、`CreateKey()` または `OpenKey()` メソッドを使ってください。

値の長さは利用可能なメモリによって制限されます。(2048 バイト以上の) 長い値はファイルに保存して、そのファイル名を設定レジストリに保存するべきです。そうすればレジストリを効率的に動作させる役に立ちます。

`_winreg.DisableReflectionKey(key)`

64 ビット OS 上で動作している 32bit プロセスに対するレジストリリフレクションを無効にします。

`key` はすでに開かれたキーか、既定の `HKEY_*` 定数のうちの一つです。

32bit OS 上では一般的に `NotImplementedError` 例外を発生させます。

`key` がリフレクションリストに無い場合は、この関数は成功しますが効果はありません。あるキーのリフレクションを無効にしても、その全てのサブキーのリフレクションには影響しません。

`_winreg.EnableReflectionKey(key)`

指定された、リフレクションが無効にされたキーのリフレクションを再び有効にします。

`key` はすでに開かれたキーか、既定の `HKEY_*` 定数のうちの一つです。

32bit OS 上では一般的に `NotImplementedError` 例外を発生させます。

あるキーのリフレクションを再開しても、その全てのサブキーには影響しません。

`_winreg.QueryReflectionKey(key)`

指定されたキーのリフレクション状態を確認します。

`key` はすでに開かれたキーか、既定の `HKEY_*` 定数のうちの一つです。

リフレクションが無効になっている場合、`True` を返します。

32bit OS 上では一般的に `NotImplementedError` 例外を発生させます。

35.3.1 定数

`_winreg` の多くの関数で利用するために以下の定数が定義されています。

HKEY_* 定数

`_winreg.HKEY_CLASSES_ROOT`

このキー以下のレジストリエントリは、ドキュメントのタイプ（またはクラス）や、それに関連付けられたプロパティを定義しています。シェルと COM アプリケーションがこの情報を利用します。

`_winreg.HKEY_CURRENT_USER`

このキー以下のレジストリエントリは、現在のユーザーの設定を定義します。この設定には、環境変数、プログラムグループに関するデータ、カラー、プリンター、ネットワーク接続、アプリケーション設定などが含まれます。

`_winreg.HKEY_LOCAL_MACHINE`

このキー以下のレジストリエントリは、コンピュータの物理的な状態を定義します。これには、バスタイプ、システムメモリ、インストールされているソフトウェアやハードウェアが含まれます。

`_winreg.HKEY_USERS`

このキー以下のレジストリエントリは、ローカルコンピュータの新規ユーザーのためのデフォルト設定や、現在のユーザーの設定を定義しています。

`_winreg.HKEY_PERFORMANCE_DATA`

このキー以下のレジストリエントリは、パフォーマンスデータへのアクセスを可能にしています。実際にはデータはレジストリには格納されていません。レジストリ関数がシステムにソースからデータを収集させます。

`_winreg.HKEY_CURRENT_CONFIG`

ローカルコンピュータシステムの現在のハードウェアプロファイルに関する情報を含みます。

`_winreg.HKEY_DYN_DATA`

このキーは Windows の 98 以降のバージョンでは利用されていません。

アクセス権限

より詳しい情報については、[Registry Key Security and Access](#) を参照してください。

`_winreg.KEY_ALL_ACCESS`

`STANDARD_RIGHTS_REQUIRED` (`KEY_QUERY_VALUE`, `KEY_SET_VALUE`, `KEY_CREATE_SUB_KEY`, `KEY_ENUMERATE_SUB_KEYS`, `KEY_NOTIFY`, `KEY_CREATE_LINK`) アクセス権限の組み合わせ。

`_winreg.KEY_WRITE`

`STANDARD_RIGHTS_WRITE` (`KEY_SET_VALUE`, `KEY_CREATE_SUB_KEY`) アクセス権限の組み合わせ。

`_winreg.KEY_READ`

`STANDARD_RIGHTS_READ` (`KEY_QUERY_VALUE`, `KEY_ENUMERATE_SUB_KEYS`, `KEY_NOTIFY`) ア

クセス権限の組み合わせ。

`_winreg.KEY_EXECUTE`

`KEY_READ` と同じ。

`_winreg.KEY_QUERY_VALUE`

レジストリキーの値を問い合わせるのに必要。

`_winreg.KEY_SET_VALUE`

レジストリの値を作成、削除、設定するのに必要。

`_winreg.KEY_CREATE_SUB_KEY`

レジストリキーのサブキーを作るのに必要。

`_winreg.KEY_ENUMERATE_SUB_KEYS`

レジストリキーのサブキーを列挙するのに必要。

`_winreg.KEY_NOTIFY`

レジストリキーやそのサブキーに対する変更通知を要求するのに必要。

`_winreg.KEY_CREATE_LINK`

システムでの利用のために予約されている。

64-bit 特有のアクセス権

より詳しい情報については、[Accessing an Alternate Registry View](#) を参照してください。

`_winreg.KEY_WOW64_64KEY`

64 bit Windows 上のアプリケーションが、64 bit のレジストリビュー上で操作する事を示します。

`_winreg.KEY_WOW64_32KEY`

64 bit Windows 上のアプリケーションが、32 bit のレジストリビュー上で操作する事を示します。

値の型

より詳しい情報は、[Registry Value Types](#) を参照してください。

`_winreg.REG_BINARY`

何らかの形式のバイナリデータ。

`_winreg.REG_DWORD`

32 ビットの数。

`_winreg.REG_DWORD_LITTLE_ENDIAN`

32 ビットのリトルエンディアン形式の数。

`_winreg.REG_DWORD_BIG_ENDIAN`

32 ビットのビッグエンディアン形式の数。

`_winreg.REG_EXPAND_SZ`

環境変数を参照している、ヌル文字で終端された文字列。(`%PATH%`)。

`_winreg.REG_LINK`

Unicode のシンボリックリンク。

`_winreg.REG_MULTI_SZ`

ヌル文字で終端された文字列からなり、二つのヌル文字で終端されている配列。(Python はこの終端の処理を自動的に行います。)

`_winreg.REG_NONE`

定義されていない値の形式。

`_winreg.REG_RESOURCE_LIST`

デバイスドライバリソースのリスト。

`_winreg.REG_FULL_RESOURCE_DESCRIPTOR`

ハードウェアセッティング。

`_winreg.REG_RESOURCE_REQUIREMENTS_LIST`

ハードウェアリソースリスト。

`_winreg.REG_SZ`

ヌル文字で終端された文字列。

35.3.2 レジストリハンドルオブジェクト

このオブジェクトは Windows の HKEY オブジェクトをラップし、オブジェクトが破壊されたときに自動的にハンドルを閉じます。オブジェクトの `Close()` メソッドと `CloseKey()` 関数のどちらも、後始末がきちんと行われることを保証するために呼び出すことができます。

このモジュールのレジストリ関数は全て、これらのハンドルオブジェクトの一つを返します。

このモジュールのレジストリ関数でハンドルオブジェクトを受け取るものは全て整数も受け取りますが、ハンドルオブジェクトを利用することを推奨します。

ハンドルオブジェクトは `__nonzero__()` のセマンティクスを持ちます - すなわち

```
if handle:
    print "Yes"
```

は、ハンドルが現在有効な (閉じられたり切り離されたりしていない) 場合には `Yes` となります。

ハンドルオブジェクトはまた、比較の意味構成もサポートしています。このため、背後の Windows ハンドル値が同じものを複数のハンドルオブジェクトが参照している場合、それらの比較は真になります。

ハンドルオブジェクトは (例えば組み込みの `int()` 関数を使って) 整数に変換することができます。この場合、背後の Windows ハンドル値が返されます、また、`Detach()` メソッドを使って整数のハンドル値を返させると同時に、ハンドルオブジェクトから Windows ハンドルを切り離すこともできます。

`PyHKEY.Close()`

背後の Windows ハンドルを閉じます。

ハンドルがすでに閉じられていてもエラーは送出されません。

`PyHKEY.Detach()`

ハンドルオブジェクトから Windows ハンドルを切り離します。

切り離される以前にそのハンドルを保持していた整数 (または、64 ビット Windows では長整数) オブジェクトが返されます。ハンドルがすでに切り離されていたり閉じられていたりした場合、ゼロが返されます。

この関数を呼び出した後、ハンドルは確実に無効化されますが、閉じられるわけではありません。背後の Win32 ハンドルがハンドルオブジェクトよりも長く維持される必要がある場合にはこの関数を呼び出すとよいでしょう。

`PyHKEY.__enter__()`

`PyHKEY.__exit__(*exc_info)`

HKEY オブジェクトは `__enter__()`, `__exit__()` メソッドを実装していて、`with` 文のためのコンテキストプロトコルをサポートしています:

```
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ... # work with key
```

このコードは、`with` ブロックから抜けるときに自動的に `key` を閉じます。

バージョン 2.6 で追加.

35.4 winsound — Windows 用の音声再生インタフェース

バージョン 1.5.2 で追加.

`winsound` モジュールは Windows プラットフォーム上で提供されている基本的な音声再生機構へのアクセス手段を提供します。このモジュールではいくつかの関数と定数が定義されています。

`winsound.Beep(frequency, duration)`

PC のスピーカを鳴らします。引数 `frequency` は鳴らす音の周波数の指定で、単位は Hz です。値は 37 から 32,767 でなくてはなりません。引数 `duration` は音を何ミリ秒鳴らすかの指定です。システムがスピーカを鳴らすことができない場合、例外 `RuntimeError` が送出されます。

バージョン 1.6 で追加.

`winsound.PlaySound(sound, flags)`

プラットフォームの API から関数 `PlaySound()` を呼び出します。引数 `sound` はファイル名、音声データの文字列、または `None` をとり得ます。 `sound` の解釈は `flags` の値に依存します。この値は以下に述べる定数をビット単位 OR して組み合わせたものになります。 `sound` 引数が `None` だった場合、現在再生中の Wave 形式サウンドの再生を停止します。システムのエラーが発生した場合、例外 `RuntimeError` が送出されます。

`winsound.MessageBeep([type=MB_OK])`

根底にある `MessageBeep()` 関数をプラットフォームの API から呼び出します。この関数は音声をレジストリの指定に従って再生します。 `type` 引数はどの音声を再生するかを指定します; とり得る値は `-1`, `MB_ICONASTERISK`, `MB_ICONEXCLAMATION`, `MB_ICONHAND`, `MB_ICONQUESTION`, および `MB_OK` で、全て以下に記述されています。値 `-1` は "単純なビーブ音" を再生します; この値は他の場合で音声を再生することができなかった際の最終的な代替音です。

バージョン 2.3 で追加。

`winsound.SND_FILENAME`

`sound` パラメタが WAV ファイル名であることを示します。 `SND_ALIAS` と同時に使ってはいけません。

`winsound.SND_ALIAS`

引数 `sound` はレジストリにある音声データに関連付けられた名前であることを示します。指定した名前がレジストリ上にない場合、定数 `SND_NODEFAULT` が同時に指定されていない限り、システム標準の音声データが再生されます。標準の音声データが登録されていない場合、例外 `RuntimeError` が送出されます。 `SND_FILENAME` と同時に使ってはいけません。

全ての Win32 システムは少なくとも以下の名前をサポートします; ほとんどのシステムでは他に多数あります:

<code>PlaySound()</code> <i>name</i>	対応するコントロールパネルでの音声名
'SystemAsterisk'	Asterisk
'SystemExclamation'	Exclamation
'SystemExit'	Exit Windows
'SystemHand'	Critical Stop
'SystemQuestion'	Question

例えば:

```
import winsound
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered (because
# "*" probably isn't the registered name of any sound).
winsound.PlaySound("*", winsound.SND_ALIAS)
```

`winsound.SND_LOOP`

音声データを繰り返し再生します。システムがブロックしないようにするため、 `SND_ASYNC` フラグを同時に使わなくてはなりません。 `SND_MEMORY` と同時に使うことはできません。

`winsound.SND_MEMORY`

`PlaySound()` の引数 `sound` が文字列の形式をとった WAV ファイルのメモリ上のイメージであることを示します。

注釈: このモジュールはメモリ上のイメージを非同期に再生する機能をサポートしていません。従って、このフラグと `SND_ASYNC` を組み合わせると例外 `RuntimeError` が送出されます。

`winsound.SND_PURGE`

指定した音声の全てのインスタンスについて再生処理を停止します。

注釈: このフラグは現代の Windows プラットフォームではサポートされていません。

`winsound.SND_ASYNC`

音声を非同期に再生するようにして、関数呼び出しを即座に返します。

`winsound.SND_NODEFAULT`

指定した音声が見つからなかった場合にシステム標準の音声を鳴らさないようにします。

`winsound.SND_NOSTOP`

現在鳴っている音声を中断させないようにします。

`winsound.SND_NOWAIT`

サウンドドライバがビジー状態にある場合、関数がすぐ返るようにします。

注釈: このフラグは現代の Windows プラットフォームではサポートされていません。

`winsound.MB_ICONASTERISK`

音声 `SystemDefault` を再生します。

`winsound.MB_ICONEXCLAMATION`

音声 `SystemExclamation` を再生します。

`winsound.MB_ICONHAND`

音声 `SystemHand` を再生します。

`winsound.MB_ICONQUESTION`

音声 `SystemQuestion` を再生します。

`winsound.MB_OK`

音声 `SystemDefault` を再生します。

第 36 章

Unix 固有のサービス

本章に記述されたモジュールは、Unix オペレーティングシステム、あるいはそれから派生した多くのものに固有の機能のためのインタフェースを提供します。その概要を以下に述べます:

36.1 `posix` — 最も一般的な POSIX システムコール群

このモジュールはオペレーティングシステムの機能のうち、C 言語標準および (Unix インタフェースをほんの少し隠蔽した) POSIX 標準で標準化されている機能に対するアクセス機構を提供します。

このモジュールを直接インポートしてはいけません。その代わりに、このインタフェースのポータブル版である `os` モジュールをインポートしてください。Unix では、`os` モジュールは `posix` インタフェースのスーパーセットを提供しています。非 Unix オペレーティングシステムでは、`posix` モジュールは利用できませんが、その一部分は `os` インタフェースを通して常に利用可能です。一度 `os` をインポートし `posix` の代わりにそれを使えば、パフォーマンス上の代償はありません。加えて `os` は、`os.environ` の要素が変更されたときに `putenv()` を呼び出すなどの追加の機能も提供します。

エラーは例外として報告されます; よくある例外は型エラーです。一方、システムコールから報告されたエラーは以下に述べるように `OSError` を送出します。

36.1.1 ラージファイルのサポート

いくつかのオペレーティングシステム (AIX, HP-UX, Irix および Solaris が含まれます) は、`int` および `long` を 32 ビット値とする C プログラムモデルで 2GB を超えるサイズのファイルのサポートを提供しています。このサポートは典型的には関連するサイズとオフセットの組合せを 64-bit 値として定義することで実現しています。このようなファイルは時にラージファイル (*large files*) と呼ばれます。

Python では、`offset` のサイズが `long` より大きく、かつ `long long` 型を利用することができて、少なくとも `offset` 型と同じくらい大きなサイズである場合、ラージファイルのサポートが有効になります。この場合、ファイルのサイズ、オフセットおよび Python の通常整数型の範囲を超えるような値の表現には Python の長整数型が使われます。このモードを有効にするのに、`configure` で Python のコンパイルに特定のコンパイルフラグを必要とするかもしれません。例えば、ラージファイルのサポートは Irix の最近のバージョンでは標準で有効ですが、Solaris 2.6 および 2.7 では、以下のようにする必要があります:

```
CFLAGS="\`getconf LFS_CFLAGS`" OPT="-g -O2 $CFLAGS" \
./configure
```

ラージファイル対応の Linux システムでは、以下のようにすれば良いでしょう:

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT="-g -O2 $CFLAGS" \
./configure
```

36.1.2 注目すべきモジュールの内容

`os` モジュールのドキュメントで説明されている多数の関数に加え、`posix` では以下のデータ項目を定義しています:

`posix.environ`

インタプリタが起動した時点の環境変数文字列を表現する辞書です。例えば、`environ['HOME']` はホームディレクトリのパス名で、C 言語の `getenv("HOME")` と等価です。

この辞書を編集しても、`execv()`、`popen()`、`system()` で渡された環境変数文字列には影響は与えません; 環境変数を変更したい場合は、`environ` を `execve()` に渡すか、変数への代入文と `export` 文を `system()` や `popen()` に渡すコマンド文字列に追加してください。

注釈: `os` モジュールでは、もう一つの `environ` 実装を提供しており、環境変数が変更された場合、その内容を更新するようになっています。`os.environ` を更新した場合、この辞書は古い内容を表していることになってしまうので、このことにも注意してください。`posix` モジュール版を直接アクセスするよりも、`os` モジュール版を使う方が推奨されています。

36.2 `pwd` — パスワードデータベースへのアクセスを提供する

このモジュールは Unix のユーザアカウントとパスワードのデータベースへのアクセスを提供します。全ての Unix 系 OS で利用できます。

パスワードデータベースの各エントリはタプルのようなオブジェクトで提供され、それぞれの属性は `passwd` 構造体のメンバに対応しています (下の属性欄については、`<pwd.h>` を見てください):

インデックス	属性	意味
0	<code>pw_name</code>	ログイン名
1	<code>pw_passwd</code>	暗号化されたパスワード (optional))
2	<code>pw_uid</code>	ユーザ ID (UID)
3	<code>pw_gid</code>	グループ ID (GID)
4	<code>pw_gecos</code>	実名またはコメント
5	<code>pw_dir</code>	ホームディレクトリ
6	<code>pw_shell</code>	シェル

UID と GID は整数で、それ以外は全て文字列です。検索したエントリが見つからないと `KeyError` が発生します。

注釈: 伝統的な Unix では、`pw_passwd` フィールドは DES 由来のアルゴリズムで暗号化されたパスワード (`crypt` モジュールをぐらんください) が含まれています。しかし、近代的な UNIX 系 OS では シャドウパスワード とよばれる仕組みを利用しています。この場合には `pw_passwd` フィールドにはアスタリスク ('*') か、'x' という一文字だけが含まれており、暗号化されたパスワードは、一般には見えない `/etc/shadow` というファイルに入っています。`pw_passwd` フィールドに有用な値が入っているかはシステムに依存します。利用可能なら、暗号化されたパスワードへのアクセスが必要なときには `spwd` モジュールを利用してください。

このモジュールでは以下の内容を定義しています:

`pwd.getpwuid(uid)`

与えられた UID に対応するパスワードデータベースのエントリを返します。

`pwd.getpwnam(name)`

与えられたユーザ名に対応するパスワードデータベースのエントリを返します。

`pwd.getpwall()`

パスワードデータベースの全てのエントリを、任意の順番で並べたリストを返します。

参考:

`grp` モジュール このモジュールに似た、グループデータベースへのアクセスを提供するモジュール。

`spwd` モジュール このモジュールと類似の、シャドウパスワードデータベースへのインタフェース。

36.3 spwd — シャドウパスワードデータベース

バージョン 2.5 で追加.

このモジュールは Unix のシャドウパスワードデータベースへのアクセスを提供します。様々な Unix 環境で利用できます。

シャドウパスワードデータベースへアクセスできる権限が必要 (大抵の場合 root である必要があります) です。

シャドウパスワードデータベースのエントリはタプル状のオブジェクトで提供され、その属性は `spwd` 構造のメンバーに対応しています (以下を参照してください。 <shadow.h> を参照):

インデックス	属性	意味
0	<code>sp_nam</code>	ログイン名
1	<code>sp_pwd</code>	暗号化されたパスワード
2	<code>sp_lstchg</code>	最終更新日
3	<code>sp_min</code>	パスワード変更が出来るようになるまでの最小日数
4	<code>sp_max</code>	パスワードを変更しなくても良い最大日数
5	<code>sp_warn</code>	パスワードが期限切れになる前に、期限切れが近づいている旨の警告をユーザに出しはじめる日数
6	<code>sp_inact</code>	パスワードが期限切れになってから、アカウントが <code>inactive</code> となり使用できなくなるまでの日数
7	<code>sp_expire</code>	1970-01-01 からアカウントが使用できなくなるまでの日数
8	<code>sp_flag</code>	将来のために予約

`sp_nam` と `sp_pwd` は文字列で、他は全て整数です。エントリが見つからなかった時は `KeyError` が起きます。

このモジュールでは以下の内容を定義しています:

`spwd.getspnam(name)`

エントリが見つからなかった時は `KeyError` が起きます。

`spwd.getspall()`

このモジュールでは以下を定義しています。

参考:

`grp` モジュール このモジュールに似た、グループデータベースへのアクセスを提供するモジュール。

`pwd` モジュール このモジュールに似た通常のパスワードデータベースへのインタフェース。

36.4 grp — グループデータベースへのアクセス

このモジュールでは Unix グループ (group) データベースへのアクセス機構を提供します。全ての Unix バージョンで利用可能です。

このモジュールはグループデータベースのエントリをタプルに似たオブジェクトとして報告されます。このオブジェクトの属性は `group` 構造体の各メンバ (以下の属性フィールド、`<pwd.h>` を参照) に対応します:

インデックス	属性	意味
0	<code>gr_name</code>	グループ名
1	<code>gr_passwd</code>	(暗号化された) グループパスワード; しばしば空文字列になります
2	<code>gr_gid</code>	数字のグループ ID
3	<code>gr_mem</code>	グループメンバの全てのユーザ名

`gid` は整数、名前およびパスワードは文字列、そしてメンバリストは文字列からなるリストです。(ほとんど

のユーザは、パスワードデータベースで自分が入れられているグループのメンバとしてグループデータベース内では明示的に列挙されていないので注意してください。完全なメンバ情報を取得するには両方のデータベースを調べてください。また、`+` や `-` で始まる `gr_name` は YP/NIS 参照である可能性があり、`getgrnam()` や `getgrgid()` でアクセスできないかもしれないことにも注意してください。)

このモジュールでは以下の内容を定義しています:

`grp.getgrgid(gid)`

与えられたグループ ID に対するグループデータベースエントリを返します。要求したエントリが見つからなかった場合、`KeyError` が送出されます。

`grp.getgrnam(name)`

与えられたグループ名に対するグループデータベースエントリを返します。要求したエントリが見つからなかった場合、`KeyError` が送出されます。

`grp.getgrall()`

全ての入手可能なグループエントリを返します。順番は決まっていません。

参考:

`pwd` モジュール このモジュールと類似の、ユーザデータベースへのインタフェース。

`spwd` モジュール このモジュールと類似の、シャドウパスワードデータベースへのインタフェース。

36.5 crypt — Unix パスワードをチェックするための関数

このモジュールは修正 DES アルゴリズムに基づいた一方向ハッシュ関数である `crypt(3)` ルーチンを実装しています。詳細については Unix マニュアルページを参照してください。このモジュールは、Python スクリプトがユーザから入力されたパスワードを受理できるようにしたり、Unix パスワードに (脆弱性検査のための) 辞書攻撃を試みるのに使えます。

このモジュールは実行環境の `crypt(3)` の実装に依存しています。そのため、現在の実装で利用可能な拡張を、このモジュールでもそのまま利用できます。

`crypt.crypt(word, salt)`

`word` は通常はユーザのパスワードで、プロンプトやグラフィカルインタフェースからタイプ入力されます。`salt` は通常ランダムな 2 文字からなる文字列で、DES アルゴリズムに 4096 通りのうち 1 つの方法で外乱を与えるために使われます。`salt` に使う文字は集合 `[./a-zA-Z0-9]` の要素でなければなりません。ハッシュされたパスワードを文字列として返します。パスワード文字列は `salt` と同じ文字集合に含まれる文字からなります (最初の 2 文字は `salt` 自体です)。

いくつかの拡張された `crypt(3)` は異なる値と `salt` の長さを許しているので、パスワードをチェックする際には `crypt` されたパスワード文字列全体を `salt` として渡すよう勧めます。

典型的な使用例のサンプルコード:

```
import crypt, getpass, pwd
```

(次のページに続く)

(前のページからの続き)

```
def login():
    username = raw_input('Python login:')
    cryptedpasswd = pwd.getpwnam(username)[1]
    if cryptedpasswd:
        if cryptedpasswd == 'x' or cryptedpasswd == '*':
            raise NotImplementedError(
                "Sorry, currently no support for shadow passwords")
        cleartext = getpass.getpass()
        return crypt.crypt(cleartext, cryptedpasswd) == cryptedpasswd
    else:
        return 1
```

36.6 dl — 共有オブジェクトの C 関数の呼び出し

バージョン 2.6 で非推奨: `dl` モジュールは Python 3 で削除されました。代わりに `ctypes` モジュールを使ってください。

`dl` モジュールは `dlopen()` 関数へのインタフェースを定義します。これはダイナミックリンクライブラリを処理するための Unix プラットフォーム上の最も一般的なインタフェースです。これによりそのライブラリの任意の関数を呼ぶプログラムが書けます。

警告: `dl` モジュールは Python の型システムとエラー処理をバイパスしています。もし間違えて使用すれば、セグメンテーションフォルト、クラッシュ、その他の不正な動作を起こします。

注釈: このモジュールは `sizeof(int) == sizeof(long) == sizeof(char *)` でなければ動きません。そうでなければ `import` するときに `SystemError` が送出されるでしょう。

`dl` モジュールは次の関数を定義します:

`dl.open(name[, mode=RTLD_LAZY])`

共有オブジェクトファイルを開いて、ハンドルを返します。モードは遅延結合 (`RTLD_LAZY`) または即時結合 (`RTLD_NOW`) を表します。デフォルトは `RTLD_LAZY` です。いくつかのシステムは `RTLD_NOW` をサポートしていないことに注意してください。

戻り値は `dlobjct` です。

`dl` モジュールは次の定数を定義します:

`dl.RTLD_LAZY`

`open()` の引数として使います。

`dl.RTLD_NOW`

`open()` の引数として使います。即時結合をサポートしないシステムでは、この定数がモジュールに現

われないうことに注意してください。最大のポータビリティを求めるならば、システムが即時結合をサポートするかどうかを決定するために `hasattr()` を使用してください。

`dl` モジュールは次の例外を定義します:

exception `dl.error`

動的なロードやリンクルーチンの内部でエラーが生じたときに送出される例外です。

例:

```
>>> import dl, time
>>> a=dl.open('/lib/libc.so.6')
>>> a.call('time'), time.time()
(929723914, 929723914.498)
```

この例は Debian GNU/Linux システム上で行なったもので、このモジュールの使用はたいてい悪い選択肢であるという事実のよい例です。

36.6.1 DI オブジェクト

`open()` によって返された DI オブジェクトは次のメソッドを持っています:

`dl.close()`

メモリーを除く全てのリソースを解放します。

`dl.sym(name)`

`name` という名前の関数が参照された共有オブジェクトに存在する場合、そのポインタ (整数値) を返します。存在しない場合 `None` を返します。これは次のように使えます:

```
>>> if a.sym('time'):
...     a.call('time')
... else:
...     time.time()
```

(0 は `NULL` ポインターであるので、この関数は 0 でない数を返すだろうということに注意してください)

`dl.call(name[, arg1[, arg2...]])`

参照された共有オブジェクトの `name` という名前の関数を呼出します。引数は、Python 整数 (そのまま渡される)、Python 文字列 (ポインタが渡される)、`None` (`NULL` として渡される) のどれかでなければいけません。Python はその文字列が変化させられるのを好まないで、文字列は `const char*` として関数に渡されるべきであることに注意してください。

最大で 10 個の引数が渡すことができ、与えられない引数は `None` として扱われます。関数の返り値は `C long` (Python 整数である) です。

36.7 termios — POSIX スタイルの端末制御

このモジュールでは端末 I/O 制御のための POSIX 準拠の関数呼び出しインタフェースを提供します。これら呼び出しのための完全な記述については、Unix マニュアルページの *termios(2)* を参照してください。これは POSIX *termios* 形式の端末制御をサポートしていてインストール時に有効にした Unix のバージョンでのみ利用可能です。

このモジュールの関数は全て、ファイル記述子 *fd* を最初の引数としてとります。この値は、`sys.stdin.fileno()` が返すような整数のファイル記述子でも、`sys.stdin` 自体のような *file object* でもかまいません。

このモジュールではまた、モジュールで提供されている関数を使う上で必要となる全ての定数を定義しています; これらの定数は C の対応する関数と同じ名前を持っています。これらの端末制御インタフェースを利用する上でのさらなる情報については、あなたのシステムのドキュメンテーションを参考にしてください。

このモジュールでは以下の関数を定義しています:

`termios.tcgetattr(fd)`

ファイル記述子 *fd* の端末属性を含むリストを返します。その形式は: `[iflag, oflag, cflag, lflag, ispeed, ospeed, cc]` です。*cc* は端末特殊文字のリストです (それぞれ長さ 1 の文字列です。ただしインデクス *VMIN* および *VTIME* の内容は、それらのフィールドが定義されていた場合整数の値となります)。端末設定フラグおよび端末速度の解釈、および配列 *cc* のインデクス検索は、*termios* で定義されているシンボル定数を使って行わなければなりません。

`termios.tcsetattr(fd, when, attributes)`

ファイル記述子 *fd* の端末属性を *attributes* から取り出して設定します。*attributes* は `tcgetattr()` が返すようなリストです。引数 *when* は属性がいつ変更されるかを決定します: *TCSANOW* は即時変更を行い、*TCSADRAIN* は現在キューされている出力を全て転送した後に変更を行い、*TCSAFLUSH* は現在キューされている出力を全て転送し、全てのキューされている入力を無視した後に変更を行います。

`termios.tcsendbreak(fd, duration)`

ファイル記述子 *fd* にブレークを送信します。*duration* をゼロにすると、0.25–0.5 秒間のブレークを送信します; *duration* の値がゼロでない場合、その意味はシステム依存です。

`termios.tcdrain(fd)`

ファイル記述子 *fd* に書き込まれた全ての出力が転送されるまで待ちます。

`termios.tcflush(fd, queue)`

ファイル記述子 *fd* にキューされたデータを無視します。どのキューかは *queue* セレクタで指定します: *TCIFLUSH* は入力キュー、*TCOFLUSH* は出力キュー、*TCIOFLUSH* は両方のキューです。

`termios.tcflow(fd, action)`

ファイル記述子 *fd* の入力または出力をサスペンドしたりレジュームしたりします。引数 *action* は出力をサスペンドする *TCOOFF*、出力をレジュームする *TCOON*、入力をサスペンドする *TCIOFF*、入力をレジュームする *TCION* をとることができます。

参考:

tty モジュール 一般的な端末制御操作のための便利な関数。

36.7.1 例

以下はエコーバックを切った状態でパスワード入力を促す関数です。ユーザの入力に関わらず以前の端末属性を正確に回復するために、二つの `tcgetattr()` と `try ... finally` 文によるテクニックが使われています:

```
def getpass(prompt="Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] = new[3] & ~termios.ECHO          # lflags
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        passwd = raw_input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```

36.8 tty — 端末制御のための関数群

`tty` モジュールは端末を `cbreak` および `raw` モードにするための関数を定義しています。

このモジュールは `termios` モジュールを必要とするため、Unix でしか動作しません。

`tty` モジュールでは、以下の関数を定義しています:

`tty.setraw(fd[, when])`

ファイル記述子 `fd` のモードを `raw` モードに変えます。 `when` を省略すると標準の値は `termios.TCSAFLUSH` になり、 `termios.tcsetattr()` に渡されます。

`tty.setcbreak(fd[, when])`

ファイル記述子 `fd` のモードを `cbreak` モードに変えます。 `when` を省略すると標準の値は `termios.TCSAFLUSH` になり、 `termios.tcsetattr()` に渡されます。

参考:

モジュール `termios` 低レベル端末制御インタフェース。

36.9 pty — 擬似端末ユーティリティ

`pty` モジュールは擬似端末 (他のプロセスを実行してその制御をしている端末をプログラムで読み書きする) を制御する操作を定義しています。

擬似端末の制御はプラットフォームに強く依存するので、Linux 用のコードしか存在していません。(Linux 用のコードは他のプラットフォームでも動作するように作られていますがテストされていません。)

`pty` モジュールでは以下の関数を定義しています:

`pty.fork()`

`fork` します。子プロセスの制御端末を擬似端末に接続します。返り値は `(pid, fd)` です。子プロセスは `pid` として 0、`fd` として `invalid` をそれぞれ受けとります。親プロセスは `pid` として子プロセスの PID、`fd` として子プロセスの制御端末 (子プロセスの標準入出力に接続されている) のファイル記述子を受けとります。

`pty.openpty()`

新しい擬似端末のペアを開きます。利用できるなら `os.openpty()` を使い、利用できなければ一般的な Unix システム用のエミュレーションコードを使います。マスター、スレーブそれぞれのためのファイル記述子、`(master, slave)` のタプルを返します。

`pty.spawn(argv[, master_read[, stdin_read]])`

プロセスを生成して制御端末を現在のプロセスの標準入出力に接続します。これは制御端末を読もうとするプログラムをごまかすために利用されます。

`master_read` と `stdin_read` にはファイル記述子から読み込む関数を指定してください。デフォルトでは呼ばれるたびに 1024 バイトずつ読み込もうとします。

36.10 fcntl — fcntl および ioctl システムコール

このモジュールでは、ファイル記述子 (file descriptor) に基づいたファイル制御および I/O 制御を実現します。このモジュールは、Unix のルーチンである `fcntl()` および `ioctl()` へのインタフェースです。これらのシステムコールの完全な説明は、`fcntl(2)` と `ioctl(2)` の Unix マニュアルページを参照してください。

このモジュール内の全ての関数はファイル記述子 `fd` を最初の引数に取ります。この値は `sys.stdin.fileno()` が返すような整数のファイル記述子でも、`sys.stdin` 自体のような、純粋にファイル記述子だけを返す `fileno()` メソッドを提供しているファイルオブジェクトでもかまいません。

このモジュールでは以下の関数を定義しています:

`fcntl.fcntl(fd, op[, arg])`

操作 `op` をファイル記述子 `fd` (または `fileno()` メソッドを提供しているファイルオブジェクト) に対して実行します。 `op` として用いられる値はオペレーティングシステム依存で、`fcntl` モジュール内に関連する C ヘッダファイルと同じ名前が使われている定数の形で利用出来ます。引数 `arg` はオプションで、標準では整数値 0 です。この引数を与える場合、整数か文字列の値をとります。引数が無い場合、この関数の戻り値は C 言語の `fcntl()` を呼び出した際の整数の戻り値になります。引数が文字列の場合には、`struct.pack()` で作られるようなバイナリの構造体を表します。バイナリデータはバッファにコピーされ、そのアドレスが C 言語の `fcntl()` 呼び出しに渡されます。呼び出しが成功した後に戻される値はバッファの内容で、文字列オブジェクトに変換されています。返される文字列は `arg` 引数と同じ長さになります。この値は 1024 バイトに制限されています。オペレーティングシステムからバッファに返される情報の長さが 1024 バイトよりも大きい場合、大抵はセグメンテーション違反となるか、より不可思議なデータの破損を引き起こします。

`fcntl()` が失敗した場合、`IOError` が送出されます。

`fcntl.ioctl(fd, op[, arg[, mutate_flag]])`

この関数は `fcntl()` 関数と同じですが、操作が通常ライブラリモジュール `termios` で定義されてお

り、引数の扱いがより複雑であるところが異なります。

パラメータ `op` は 32 ビットに収まる値に制限されます。 `op` 引数として使うのに関係のある追加の定数は `termios` モジュールにあって、関連する C ヘッドファイルで使われているのと同じ名前が付けられています。

パラメータ `arg` は整数か、存在しない (整数 0 と等価なものとして扱われます) か、(通常の Python 文字列のような) 読み出し専用のバッファインタフェースをサポートするオブジェクトか、読み書きバッファインタフェースをサポートするオブジェクトです。

最後の型のオブジェクトを除き、動作は `fcntl()` 関数と同じです。

可変なバッファが渡された場合、動作は `mutate_flag` 引数の値で決定されます。

この値が偽の場合、バッファの可変性は無視され、動作は読み出しバッファの場合と同じになります。が、上で述べた 1024 バイトの制限は回避されます – 従って、オペレーティングシステムが希望するバッファ長までであれば正しく動作します。

`mutate_flag` が真の場合、バッファは (実際には) 根底にある `ioctl()` システムコールに渡され、後者の戻り値が呼び出し側の Python に引き渡され、バッファの新たな内容は `ioctl()` の動作を反映します。この説明はやや単純化されています。というのは、与えられたバッファが 1024 バイト長より短い場合、バッファはまず 1024 バイト長の静的なバッファにコピーされてから `ioctl()` に渡され、その後引数で与えたバッファに戻しコピーされるからです。

`mutate_flag` が与えられなかった場合、2.3 ではこの値は偽となります。この仕様は今後のいくつかのバージョンを経た Python で変更される予定です: 2.4 では、`mutate_flag` を提供し忘れると警告が出されますが同じ動作を行い、2.5 ではデフォルトの値が真となるはずですが。

`ioctl()` が失敗すると、`IOError` 例外が送出されます。

以下に例を示します:

```
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPRG, " "))[0]
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPRG, buf, 1)
0
>>> buf
array('h', [13341])
```

`fcntl.flock(fd, op)`

ファイル記述子 `fd` (`fileno()` メソッドを提供しているファイルオブジェクトも含む) に対してロック操作 `op` を実行します。詳細は Unix マニュアルの `flock(2)` を参照してください (システムによっては、この関数は `fcntl()` を使ってエミュレーションされています)。

`flock()` が失敗すると、`IOError` 例外が送出されます。

`fcntl.lockf(fd, operation[, length[, start[, whence]]])`

本質的に `fcntl()` によるロッキングの呼び出しをラップしたものです。 `fd` はロックまたはアンロックするファイルのファイル記述子で、 `operation` は以下の値のうちいずれかになります:

- `LOCK_UN` – アンロック
- `LOCK_SH` – 共有ロックを取得
- `LOCK_EX` – 排他的ロックを取得

`operation` が `LOCK_SH` または `LOCK_EX` の場合、 `LOCK_NB` とビット OR にすることでロック取得時にブロックしないようにすることができます。 `LOCK_NB` が使われ、ロックが取得できなかった場合、 `IOError` が送出され、例外は `errno` 属性を持ち、その値は `EACCESS` または `EAGAIN` になります (オペレーティングシステムに依存します; 可搬性のため、両方の値をチェックしてください)。少なくともいくつかのシステムでは、ファイル記述子が参照しているファイルが書き込みのために開かれている場合、 `LOCK_EX` だけしか使うことができません。

`length` はロックを行いたいバイト数、 `start` はロック領域先頭の `whence` からの相対的なバイトオフセット、 `whence` は `io.IOBase.seek()` と同じで、具体的には:

- 0 – ファイル先頭からの相対位置 (`os.SEEK_SET`)
- 1 – 現在のバッファ位置からの相対位置 (`os.SEEK_CUR`)
- 2 – ファイルの末尾からの相対位置 (`os.SEEK_END`)

`start` の標準の値は 0 で、ファイルの先頭から開始することを意味します。 `whence` の標準の値も 0 です。

以下に (全ての SVR4 互換システムでの) 例を示します:

```
import struct, fcntl, os

f = open(...)
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

最初の例では、戻り値 `rv` は整数値を保持しています; 二つ目の例では文字列値を保持しています。 `lockdata` 変数の構造体レイアウトはシステム依存です — 従って `flock()` を呼ぶ方がベターです。

参考:

`os` モジュール もし `os` モジュールに `os.O_SHLOCK` と `os.O_EXLOCK` が存在する場合 (BSD のみ)、 `os.open()` 関数は `lockf()` や `flock()` 関数を代替できます。

36.11 pipes — シェルパイプラインへのインタフェース

ソースコード: [Lib/pipes.py](#)

`pipes` モジュールでは、*pipeline* の概念 — あるファイルを別のファイルに変換する機構の直列接続 — を抽象化するためのクラスを定義しています。

このモジュールは `/bin/sh` コマンドラインを利用するため、`os.system()` および `os.popen()` のための POSIX 準拠のシェル、または互換のシェルが必要です。

class pipes.Template

パイプラインを抽象化したクラス。

例:

```
>>> import pipes
>>> t = pipes.Template()
>>> t.append('tr a-z A-Z', '--')
>>> f = t.open('pipefile', 'w')
>>> f.write('hello world')
>>> f.close()
>>> open('pipefile').read()
'HELLO WORLD'
```

`pipes.quote(s)`

バージョン 2.7 で非推奨: Python 2.7 のある時点まで、この関数は公式にドキュメントされてきませんでした。これは Python 3.3 でようやく、ただし `shlex` モジュール内の `quote` 関数として公になりました。

文字列 `s` をシェルエスケープして返します。戻り値は、リストを使えないようなケースで、シェルコマンドライン内で一つのトークンとして安全に利用出来る文字列です。

以下のイディオムは安全ではないかもしれません:

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print command # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

`quote()` がそのセキュリティの穴をふさぎます:

```
>>> command = 'ls -l {}'.format(quote(filename))
>>> print command
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print remote_command
ssh home 'ls -l "'"somefile; rm -rf ~'"'
```

この引用方法は UNIX シェル、`shlex.split()` と互換です:

```
>>> remote_command = shlex.split(remote_command)
>>> remote_command
['ssh', 'home', "ls -l 'somefile; rm -rf ~'"]
>>> command = shlex.split(remote_command[-1])
```

(次のページに続く)

(前のページからの続き)

```
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

36.11.1 テンプレートオブジェクト

テンプレートオブジェクトは以下のメソッドを持っています:

`Template.reset()`

パイプラインテンプレートを初期状態に戻します。

`Template.clone()`

元のパイプラインテンプレートと等価の新しいオブジェクトを返します。

`Template.debug(flag)`

flag が真の場合、デバッグをオンにします。そうでない場合、デバッグをオフにします。デバッグがオンの時には、実行されるコマンドが印字され、より多くのメッセージを出力するようにするために、シェルに `set -x` 命令を与えます。

`Template.append(cmd, kind)`

新たなアクションをパイプラインの末尾に追加します。*cmd* 変数は有効な bourne shell 命令でなければなりません。*kind* 変数は二つの文字からなります。

最初の文字は `'r'` (コマンドが標準入力からデータを読み出すことを意味します)、`'f'` (コマンドがコマンドライン上で与えたファイルからデータを読み出すことを意味します)、あるいは `'.'` (コマンドは入力を読まないことを意味します、従ってパイプラインの先頭になります)、のいずれかになります。

同様に、二つ目の文字は `'r'` (コマンドが標準出力に結果を書き込むことを意味します)、`'f'` (コマンドがコマンドライン上で指定したファイルに結果を書き込むことを意味します)、あるいは `'.'` (コマンドはファイルを書き込まないことを意味し、パイプラインの末尾になります)、のいずれかになります。

`Template.prepend(cmd, kind)`

パイプラインの先頭に新しいアクションを追加します。引数の説明については [append\(\)](#) を参照してください。

`Template.open(file, mode)`

ファイル類似のオブジェクトを返します。このオブジェクトは *file* を開いていますが、パイプラインを通して読み書きするようになっています。*mode* には `'r'` または `'w'` のいずれか一つしか与えることができないので注意してください。

`Template.copy(infile, outfile)`

パイプを通して *infile* を *outfile* にコピーします。

36.12 posixfile — ロック機構をサポートするファイル類似オブジェクト

バージョン 1.5 で非推奨: このモジュールが提供しているよりもうまく処理ができ、可搬性も高いロック操作が `fcntl.lockf()` で提供されています。

このモジュールでは、組み込みのファイルオブジェクトの上にいくつかの追加機能を実装しています。特に、このオブジェクトはファイルのロック機構、ファイルフラグへの操作、およびファイルオブジェクトを複製するための簡単なインタフェースを実装しています。このモジュールは新たなファイルオブジェクト `posixfile` を定義しています。このオブジェクトは全ての標準ファイルオブジェクトのメソッドに加え、下記に述べるメソッドを持っています。このモジュールはファイルのロック機構に `fcntl.fcntl()` を用いるため、ある種の Unix でしか動作しません。

`posixfile` オブジェクトをインスタンス化するには、`posixfile.open()` 関数を使います。生成されるオブジェクトは標準ファイルオブジェクトとだいたい同じロック&フィールドです。

`posixfile` モジュールでは、以下の定数を定義しています:

`posixfile.SEEK_SET`

オフセットをファイルの先頭から計算します。

`posixfile.SEEK_CUR`

オフセットを現在のファイル位置から計算します。

`posixfile.SEEK_END`

オフセットをファイルの末尾から計算します。

`posixfile` モジュールでは以下の関数を定義しています:

`posixfile.open(filename[, mode[, bufsize]])`

指定したファイル名とモードで新しい `posixfile` オブジェクトを作成します。 `filename`、 `mode` および `bufsize` 引数は組み込みの `open()` 関数と同じように解釈されます。

`posixfile.fileopen(fileobject)`

指定した標準ファイルオブジェクトで新しい `posixfile` オブジェクトを作成します。作成されるオブジェクトは元のファイルオブジェクトと同じファイル名およびモードを持っています。

`posixfile` オブジェクトでは以下の追加メソッドを定義しています:

`posixfile.lock(fmt[, len[, start[, whence]]])`

ファイルオブジェクトが参照しているファイルの指定部分にロックをかけます。指定の書式は下のテーブルで説明されています。 `len` 引数にはロックする部分の長さを指定します。標準の値は 0 です。 `start` にはロックする部分の先頭オフセットを指定し、その標準値は 0 です。 `whence` 引数はオフセットをどこからの相対位置にするかを指定します。この値は定数 `SEEK_SET`、`SEEK_CUR`、または `SEEK_END` のいずれかになります。標準の値は `SEEK_SET` です。引数についてのより詳しい情報はシステムの `fcntl(2)` マニュアルページを参照してください。

`posixfile.flags([flags])`

ファイルオブジェクトが参照しているファイルに指定したフラグを設定します。新しいフラグは特に指

定しない限り以前のフラグと OR されます。指定書式は下のテーブルで説明されています。 *flags* 引数なしの場合、現在のフラグを示す文字列が返されます (? 修飾子と同じです)。フラグについてのより詳しい情報はシステムの *fcntl(2)* マニュアルページを参照してください。

`posixfile.dup()`
ファイルオブジェクトと、背後のファイルポインタおよびファイル記述子を複製します。返されるオブジェクトは新たに開かれたファイルのように振舞います。

`posixfile.dup2(fd)`
ファイルオブジェクトと、背後のファイルポインタおよびファイル記述子を複製します。新たなオブジェクトは指定したファイル記述子を持ちます。それ以外の点では、返されるオブジェクトは新たに開かれたファイルのように振舞います。

`posixfile.file()`
`posixfile` オブジェクトが参照している標準ファイルオブジェクトを返します。この関数は標準ファイルオブジェクトを使うよう強制している関数を使う場合に時々必要になります。

全てのメソッドで、要求された操作が失敗した場合には *IOError* が送出されます。

`lock()` の書式指定文字には以下のような意味があります:

フォーマット	意味
u	指定領域のロックを解除します
r	指定領域の読み出しロックを要求します
w	指定領域の書き込みロックを要求します

これに加え、以下の修飾子を書式に追加できます:

修飾子	意味	注釈
	ロック操作が許可されるまで待ちます	
?	要求されたロックと衝突している第一のロックを返すか、衝突がない場合には <code>None</code> を返します。	(1)

注釈:

- (1) 返されるロックは `(mode, len, start, whence, pid)` の形式で、*mode* はロックのタイプを表す文字 ('r' または 'w') です。この修飾子はロック要求の許可を行わせません; すなわち、問い合わせの目的にしか使えません。

`flags()` の書式指定文字には以下のような意味があります:

フォーマット	意味
a	追記のみ (append only) フラグ
c	実行時クローズ (close on exec) フラグ
n	無遅延 (no delay) フラグ (非ブロック (non-blocking) フラグとも呼ばれます)
s	同期 (synchronization) フラグ

これに加え、以下の修飾子を書式に追加できます:

修飾子	意味	注釈
!	指定したフラグを通常の ' オン ' にせず ' オフ ' にします	(1)
=	フラグを標準の ' OR ' 操作ではなく置換します。	(1)
?	設定されているフラグを表現する文字からなる文字列を返します。	(2)

注釈:

- (1) ! および = 修飾子は互いに排他的の関係にあります。
- (2) この文字列が表すフラグは同じ呼び出しによってフラグが置き換えられた後のものです。

例:

```
import posixfile

file = posixfile.open('testfile', 'w')
file.lock('w|')
...
file.lock('u')
file.close()
```

36.13 resource — リソース使用状態の情報

このモジュールでは、プログラムによって使用されているシステムリソースを計測したり制御するための基本的なメカニズムを提供します。

特定のシステムリソースを指定したり、現在のプロセスやその子プロセスのリソース使用情報を要求するためにシンボル定数が使われます。

エラーを表すための例外が一つ定義されています:

exception resource.error

下に述べる関数は、背後にあるシステムコールが予期せず失敗した場合、このエラーを送出するかもしれません。

36.13.1 リソースの制限

リソースの使用は下に述べる `setrlimit()` 関数を使って制限することができます。各リソースは二つ組の制限値: ソフトリミット (soft limit)、およびハードリミット (hard limit)、で制御されます。ソフトリミットは現在の制限値で、時間とともにプロセスによって下げたり上げたりできます。ソフトリミットはハードリミットを超えることはできません。ハードリミットはソフトリミットよりも高い任意の値まで下げることができません、上げることはできません。(スーパーユーザの有効な UID を持つプロセスのみがハードリミットを上げることができます。)

制限をかけるべく指定できるリソースはシステムに依存します。指定できるリソースは `getrlimit(2)` マニュアルページで解説されています。以下に列挙するリソースは背後のオペレーティングシステムがサポートする場合にサポートされています; オペレーティングシステム側で値を調べたり制御したりできないリソースは、そのプラットフォーム向けのこのモジュール内では定義されていません。

`resource.RLIM_INFINITY`

無制限のリソースの上限を示すための定数です。

`resource.getrlimit(resource)`

`resource` の現在のソフトおよびハードリミットを表すタプル (soft, hard) を返します。無効なリソースが指定された場合には `ValueError` が、背後のシステムコールが予期せず失敗した場合には `error` が送出されます。

`resource.setrlimit(resource, limits)`

`resource` の新たな消費制限を設定します。 `limits` 引数には、タプル (soft, hard) による二つの整数で、新たな制限を記述しなければなりません。 `RLIM_INFINITY` を指定することで、無制限を要求することが出来ます。

無効なリソースが指定された場合、ソフトリミットの値がハードリミットの値を超えている場合、プロセスがハードリミットを引き上げようとした場合には `ValueError` が送出されます。リソースのハードリミットやシステムリミットが無制限でないのに `RLIM_INFINITY` を指定した場合も、`ValueError` になります。スーパーユーザの実効 UID を持ったプロセスは無制限を含めあらゆる妥当な制限値を要求出来ますが、システムが課している制限を超過した要求ではやはり `ValueError` となります。

`setrlimit` は背後のシステムコールが予期せず失敗した場合に、 `error` を送出する場合があります。

以下のシンボルは、後に述べる関数 `setrlimit()` および `getrlimit()` を使って消費量を制御することができるリソースを定義しています。これらのシンボルの値は、C プログラムで使われているシンボルと全く同じです。

`getrlimit(2)` の Unix マニュアルページには、指定可能なリソースが列挙されています。全てのシステムで同じシンボルが使われているわけではなく、また同じリソースを表すために同じ値が使われているとも限らないので注意してください。このモジュールはプラットフォーム間の相違を隠蔽しようとはしていません — あるプラットフォームで定義されていないシンボルは、そのプラットフォーム向けの本モジュールでは利用することができません。

`resource.RLIMIT_CORE`

現在のプロセスが生成できるコアファイルの最大 (バイト) サイズです。プロセスの全体イメージを入

れるためにこの値より大きなサイズのコアファイルが要求された結果、部分的なコアファイルが生成される可能性があります。

`resource.RLIMIT_CPU`

プロセッサが利用することができる最大プロセッサ時間 (秒) です。この制限を超えた場合、SIGXCPU シグナルがプロセスに送られます。(どのようにしてシグナルを捕捉したり、例えば開かれているファイルをディスクにフラッシュするといった有用な処理を行うかについての情報は、`signal` モジュールのドキュメントを参照してください)

`resource.RLIMIT_FSIZE`

プロセスが作成するファイルの最大サイズです。

`resource.RLIMIT_DATA`

プロセスのヒープの最大 (バイト) サイズです。

`resource.RLIMIT_STACK`

現在のプロセスのコールスタックの最大サイズ (バイト単位) です。これはマルチスレッドプロセスのメインスレッドのスタックのみに影響します。

`resource.RLIMIT_RSS`

プロセスが取りうる最大 RAM 常駐ページサイズ (resident set size) です。

`resource.RLIMIT_NPROC`

現在のプロセスが生成できるプロセスの上限です。

`resource.RLIMIT_NOFILE`

現在のプロセスが開けるファイル記述子の上限です。

`resource.RLIMIT_OFILE`

`RLIMIT_NOFILE` の BSD での名称です。

`resource.RLIMIT_MEMLOCK`

メモリ中でロックできる最大アドレス空間です。

`resource.RLIMIT_VMEM`

プロセスが占有できるマップメモリの最大領域です。

`resource.RLIMIT_AS`

アドレス空間でプロセスが占有できる最大領域 (バイト) です。

36.13.2 リソースの使用状態

以下の関数はリソース使用情報を取得するために使われます:

`resource.getrusage(who)`

この関数は、`who` 引数で指定される、現プロセスおよびその子プロセスによって消費されているリソースを記述するオブジェクトを返します。`who` 引数は以下に記述される `RUSAGE_*` 定数のいずれかを使って指定します。

返される値の各フィールドはそれぞれ、個々のシステムリソースがどれくらい使用されているか、例えばユーザモードでの実行に費やされた時間やプロセスが主記憶からスワップアウトされた回数、を示しています。幾つかの値、例えばプロセスが使用しているメモリ量は、内部時計の最小単位に依存します。

以前のバージョンとの互換性のため、返される値は 16 要素からなるタプルとしてアクセスすることもできます。

戻り値のフィールド `ru_utime` および `ru_stime` は浮動小数点数で、それぞれユーザモードでの実行に費やされた時間、およびシステムモードでの実行に費やされた時間を表します。それ以外の値は整数です。これらの値に関する詳しい情報は `getrusage(2)` を調べてください。以下に簡単な概要を示します:

インデックス	フィールド	リソース
0	<code>ru_utime</code>	ユーザモード実行時間 (float)
1	<code>ru_stime</code>	システムモード実行時間 (float)
2	<code>ru_maxrss</code>	最大常駐ページサイズ
3	<code>ru_ixrss</code>	共有メモリサイズ
4	<code>ru_idrss</code>	非共有メモリサイズ
5	<code>ru_isrss</code>	非共有スタックサイズ
6	<code>ru_minflt</code>	I/O を必要としないページフォールト数
7	<code>ru_majflt</code>	I/O を必要とするページフォールト数
8	<code>ru_nswap</code>	スワップアウト回数
9	<code>ru_inblock</code>	ブロック入力操作数
10	<code>ru_oublock</code>	ブロック出力操作数
11	<code>ru_msgsnd</code>	送信メッセージ数
12	<code>ru_msgrcv</code>	受信メッセージ数
13	<code>ru_nsignals</code>	受信シグナル数
14	<code>ru_nvcsw</code>	自発的な実行コンテキスト切り替え数
15	<code>ru_nivcsw</code>	非自発的な実行コンテキスト切り替え数

この関数は無効な `who` 引数を指定した場合には `ValueError` を送出します。また、異常が発生した場合には `error` 例外が送出される可能性があります。

バージョン 2.3 で変更: 値へのアクセスを戻り値の属性として追加しました。

```
resource.getpagesize()
```

システムページ内のバイト数を返します。(ハードウェアページサイズと同じとは限りません。)

以下の `RUSAGE_*` シンボルはどのプロセスの情報を提供させるかを指定するために関数 `getrusage()` に渡されます。

```
resource.RUSAGE_SELF
```

`RUSAGE_SELF` はプロセス自体のみに属する情報を要求するために使われます。

```
resource.RUSAGE_CHILDREN
```

`getrusage()` に渡すと呼び出し側プロセスの子プロセスのリソース情報を要求します。

`resource.RUSAGE_BOTH`

`getrusage()` に渡すと現在のプロセスおよび子プロセスの両方が消費しているリソースを要求します。全てのシステムで利用可能なわけではありません。

36.14 nis — Sun の NIS (Yellow Pages) へのインタフェース

`nis` モジュールは複数のホストを集中管理する上で便利な NIS ライブラリを薄くラップします。

NIS は Unix システム上にしかないので、このモジュールは Unix でしか利用できません。

`nis` モジュールでは以下の関数を定義しています:

`nis.match(key, mapname[, domain=default_domain])`

`mapname` 中で `key` に一致するものを返すか、見つからない場合にはエラー (`nis.error`) を送出します。両方の引数とも文字列で、`key` は 8 ビットクリーンです。返される値は (NULL その他を含む可能性のある) 任意のバイト列です。

`mapname` は他の名前の別名になっていないか最初にチェックされます。

バージョン 2.5 で変更: `domain` 引数は検索に使う NIS ドメインを上書きできます。指定されなければ、デフォルト NIS ドメイン内が検索されます。

`nis.cat(mapname[, domain=default_domain])`

`match(key, mapname)==value` となる `key` を `value` に対応付ける辞書を返します。辞書内のキーと値は共に任意のバイト列なので注意してください。

`mapname` は他の名前の別名になっていないか最初にチェックされます。

バージョン 2.5 で変更: `domain` 引数は検索に使う NIS ドメインを上書きできます。指定されなければ、デフォルト NIS ドメイン内が検索されます。

`nis.maps([domain=default_domain])`

有効なマップのリストを返します。

バージョン 2.5 で変更: `domain` 引数は検索に使う NIS ドメインを上書きできます。指定されなければ、デフォルト NIS ドメイン内が検索されます。

`nis.get_default_domain()`

システムのデフォルト NIS ドメインを返します。

バージョン 2.5 で追加.

`nis` モジュールは以下の例外を定義しています:

exception `nis.error`

NIS 関数がエラーコードを返した場合に送出されます。

36.15 syslog — Unix syslog ライブラリルーチン群

このモジュールでは Unix `syslog` ライブラリルーチン群へのインタフェースを提供します。`syslog` の便宜レベルに関する詳細な記述は Unix マニュアルページを参照してください。

このモジュールはシステムの `syslog` ファミリのルーチンをラップしています。`syslog` サーバーと通信できる pure Python のライブラリが、`logging.handlers` モジュールの `SysLogHandler` にあります。

このモジュールでは以下の関数を定義しています:

`syslog.syslog(message)`

`syslog.syslog(priority, message)`

文字列 `message` をシステムログ機構に送信します。末尾の改行文字は必要に応じて追加されます。各メッセージは `facility` および `level` からなる優先度でタグ付けされます。オプションの `priority` 引数はメッセージの優先度を定義します。標準の値は `LOG_INFO` です。`priority` 中に、便宜レベルが (`LOG_INFO` | `LOG_USER` のように) 論理和を使ってコード化されていない場合、`openlog()` を呼び出した際の値が使われます。

`syslog()` が呼び出される前に `openlog()` が呼び出されなかった場合、`openlog()` が引数なしで呼び出されます。

`syslog.openlog([ident[, logoption[, facility]]])`

`openlog()` 関数を呼び出すことで以降の `syslog()` の呼び出しに対するログオプションを設定することができます。ログがまだ開かれていない状態で `syslog()` を呼び出すと `openlog()` が引数なしで呼び出されます。

オプションの `ident` キーワード引数は全てのメッセージの先頭に付く文字列で、デフォルトでは `sys.argv[0]` から前方のパス部分を取り除いたものです。オプションの `logoption` キーワード引数 (デフォルトは 0) はビットフィールドです。組み合わせられる値については下記を参照してください。オプションの `facility` キーワード引数 (デフォルトは `LOG_USER`) は明示的に `facility` が encode されていないメッセージに設定される `facility` です。

`syslog.closelog()`

`syslog` モジュールの値をリセットし、システムライブラリの `closelog()` を呼び出します。

この関数を呼ぶと、モジュールが最初に import されたときと同じようにふるまいます。例えば、(`openlog()` を呼び出さないで) `syslog()` を最初に呼び出したときに、`openlog()` が呼び出され、`ident` やその他の `openlog()` の引数はデフォルト値にリセットされます。

`syslog.setlogmask(maskpri)`

優先度マスクを `maskpri` に設定し、以前のマスク値を返します。`maskpri` に設定されていない優先度レベルを持った `syslog()` の呼び出しは無視されます。標準では全ての優先度をログ出力します。関数 `LOG_MASK(pri)` は個々の優先度 `pri` に対する優先度マスクを計算します。関数 `LOG_UPTO(pri)` は優先度 `pri` までの全ての優先度を含むようなマスクを計算します。

このモジュールでは以下の定数を定義しています:

優先度 (降順): `LOG_EMERG`、`LOG_ALERT`、`LOG_CRIT`、`LOG_ERR`、`LOG_WARNING`、`LOG_NOTICE`、`LOG_INFO`、`LOG_DEBUG`。

機能: LOG_KERN, LOG_USER, LOG_MAIL, LOG_DAEMON, LOG_AUTH, LOG_LPR, LOG_NEWS, LOG_UUCP, LOG_CRON, LOG_SYSLOG および、LOG_LOCAL0 から LOG_LOCAL7。

ログオプション: <syslog.h> で定義されている場合、LOG_PID、LOG_CONS、LOG_NDELAY、LOG_NOWAIT、および LOG_PERROR。

36.15.1 例

シンプルな例

1 つ目のシンプルな例:

```
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

いくつかのログオプションを設定する例。ログメッセージにプロセス ID を含み、メッセージをメールのログ用の facility にメッセージを書きます:

```
syslog.openlog(logoption=syslog.LOG_PID, facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```

36.16 commands — コマンド実行ユーティリティ

バージョン 2.6 で非推奨: *commands* モジュールは Python 3 で削除されました。代わりに *subprocess* モジュールを使ってください。

commands は、システムへコマンド文字列を渡して実行する *os.popen()* のラッパー関数を含んでいるモジュールです。外部で実行したコマンドの結果や、その終了ステータスを扱います。

subprocess がプロセスを生成してその結果を取得するためのより強力な手段を提供しています。*subprocess* モジュールを使う方が *commands* モジュールを使うより好ましいです。

注釈: Python 3.x において、*getstatus()* および二つの隠し関数 (*mk2arg()* と *mkarg()*) は削除されました。また、*getstatusoutput()* と *getoutput()* は *subprocess* モジュールに移動されました。

commands モジュールは以下の関数を定義しています。

commands.getstatusoutput(cmd)

文字列 *cmd* を *os.popen()* を使いシェル上で実行し、タプル (*status*, *output*) を返します。実際には { *cmd*; } 2>&1 と実行されるため、標準出力とエラー出力が混合されます。また、出力の最後の改行文字は取り除かれます。コマンドの終了ステータスは C 言語関数の *wait()* の規則に従って解釈することができます。

`commands.getoutput (cmd)`

`getstatusoutput ()` に似ていますが、終了ステータスは無視され、コマンドの出力のみを返します。

`commands.getstatus (file)`

`ls -ld file` の出力を文字列で返します。この関数は `getoutput ()` を使い、引数内のバックスラッシュ記号とドル記号を適切にエスケープします。

バージョン 2.6 で非推奨: この関数は明らかでないですし役立たずです。名前も `getstatusoutput ()` の前では誤解を招くものです。

例:

```
>>> import commands
>>> commands.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> commands.getstatusoutput('cat /bin/junk')
(256, 'cat: /bin/junk: No such file or directory')
>>> commands.getstatusoutput('/bin/junk')
(256, 'sh: /bin/junk: not found')
>>> commands.getoutput('ls /bin/ls')
'/bin/ls'
>>> commands.getstatus('/bin/ls')
'-rwxr-xr-x  1 root          13352 Oct 14  1994 /bin/ls'
```

参考:

`subprocess` モジュール サブプロセスの生成と管理のためのモジュール。

第 37 章

Mac OS X 固有のサービス

この章では Mac OS X プラットフォームでのみ利用可能なモジュールについて説明します。

さらに多くのモジュールについて [MacPython OSA モジュール](#) や 文書化されていない [Mac OS モジュール](#) も参照してください。また HOWTO の [using-on-mac](#) は Mac 固有の Python プログラミングについての一般的な入門編になっています。

注釈: 最新バージョンの OS X では、このモジュールが使っている OS X API のほとんどが非推奨になっているか、もしくは削除されています。多くは Python が 64 ビットモードで動作している場合には利用出来ません。このモジュールは Python 3 では削除済みです。Python 2 でもこのモジュールの利用は避けるべきです。

37.1 `ic` — Mac OS X のインターネット設定へのアクセス

このモジュールでは、システム設定 や **Finder** で設定したインターネット関連の設定へのアクセス機能を提供しています。

注釈: このモジュールは Python 3.x で削除されました。

このモジュールには、`icglue` という低水準の関連モジュールがあり、インターネット設定への基本的なアクセス機能を提供しています。この低水準のモジュールはドキュメント化されていませんが、各ルーチンの docstring にはパラメタの説明があり、ルーチン名には Internet Config に対する Pascal や C のインタフェースと同じ名前を使っているため、このモジュールが必要な場合には標準の IC プログラマドキュメントを利用できます。

`ic` モジュールでは、例外 `error` と、インターネット設定から生じる全てのエラーコードに対するシンボル名を定義しています。詳しくはソースコードを参照してください。

exception `ic.error`

`ic` モジュール内部でエラーが生じたときに送出される例外です。

`ic` モジュールは以下のクラスと関数を定義しています:


```
class ic.IC([signature[, ic]])
```

インターネット設定オブジェクトを作成します。 *signature* は、IC の設定に影響を及ぼす可能性のある現在のアプリケーションを表す 4 文字のクリエイターコード (デフォルトは 'Pyth') です。オプションの引数 *ic* は低水準モジュールであらかじめ作成しておいた `icglue.icinstance` で、別の設定ファイルなどから設定を得る場合に便利です。

```
ic.launchurl(url[, hint])
ic.parseurl(data[, start[, end[, hint]]])
ic.mapfile(file)
ic.maptypescreator(type, creator[, filename])
ic.settypescreator(file)
```

これらの関数は、後述する同名のメソッドへの「ショートカット」です。

37.1.1 IC オブジェクト

IC オブジェクトはマップ型のインターフェースを持っているので、メールアドレスの取得は単に `ic['MailAddress']` でできます。値の代入もでき、設定ファイルのオプションを変更できます。

このモジュールは各種のデータ型を知っていて、IC 内部の表現を「論理的な」Python データ構造に変換します。 `ic` モジュールを単体で実行すると、テストプログラムが実行されて IC データベースにある全てのキーと値のペアをリスト表示するので、文書代わりになります。

モジュールがデータの表現方法を推測できなかった場合、 `data` 属性に生のデータが入った `ICOpaqueData` 型のインスタンスを返します。この型のオブジェクトも代入に利用できます。

IC には辞書型のインターフェースの他にも以下のようなメソッドがあります:

```
IC.launchurl(url[, hint])
```

与えられた URL を解析し、適切なアプリケーションを起動して URL を渡します。省略可能な *hint* は、 'mailto:' などのスキーム名で、不完全な URL はこのスキームにあわせて補完します。 *hint* を指定していない場合、不完全な URL は無効になります。

```
IC.parseurl(data[, start[, end[, hint]]])
```

data の中から URL を検索し、URL の開始位置、終了位置、URL そのものを返します。オプションの引数 *start* と *end* を使うと検索範囲を制限できます。例えば、ユーザーが長いテキストフィールドをクリックした場合に、このルーチンにテキストフィールド全体とクリック位置 *start* を渡すことで、ユーザーがクリックした場所にある URL 全体を返させられます。先に述べたように、 *hint* はオプションで、不完全な URL を補完するためのスキームです。

```
IC.mapfile(file)
```

file に対するマッピングエントリを返します。 *file* にはファイル名か `FSSpec()` の戻り値を渡せます。実在しないファイルであってもかまいません。

マッピングエントリは (version, type, creator, postcreator, flags, extension, appname, postappname, mimetype, entryname) からなるタプルで返されます。 *version* はエントリーのバージョン番号、 *type* は 4 文字のファイルタイプ、 *creator* は 4 文字のクリエイタータイプ、 *postcreator* はファイルのダウンロード後にオプションとして起動され、後処理を行うアプリケー

ションの 4 文字のクリエイターコードです。 *flags* は、転送をバイナリで行うかアスキーで行うか、などの様々なフラグビットからなる値です。 *extension* はこのファイルタイプに対するファイル名の拡張子、 *appname* はファイルが属するアプリケーションの印字可能な名前、 *postappname* は後処理用アプリケーション、 *mimetype* はこのファイルの MIME タイプ、最後の *entryname* はこのエントリの名前です。

IC. **maptypecreator** (*type*, *creator* [, *filename*])

4 文字の *type* と *creator* コードを持つファイルに対するマッピングエントリを返します。(クリエイターが '????' であるような場合に) 正しいエントリが見つかりやすいようにオプションの *filename* を指定できます。

マッピングエントリーは *mapfile* と同じフォーマットで返されます。

IC. **settypecreator** (*file*)

実在のファイル *file* に対して、拡張子に基づいて適切なクリエイターとタイプを設定します。 *file* の指定は、ファイル名でも `FSSpec()` の戻り値でもかまいません。変更は Finder に通知されるので、Finder 上のアイコンは即座に更新されます。

37.2 MacOS — Mac OS インタプリタ機能へのアクセス

このモジュールは、Python インタプリタ内の MacOS 固有の機能に対するアクセスを提供します。例えば、インタプリタのイベントループ関数などです。十分注意して利用してください。

注釈: このモジュールは Python 3.x で削除されました。

モジュール名が大文字で始まることに注意してください。これは昔からの約束です。

MacOS.runtimeModel

Python 2.4 以降は常に 'macho' です。それより前のバージョンの Python では、古い Mac OS 8 ランタイムモデルの場合は 'ppc'、Mac OS 9 ランタイムモデルの場合は 'carbon' となります。

MacOS.linkModel

インタプリタがどのような方法でリンクされているかを返します。拡張モジュールがリンクモデル間で非互換性かもしれない場合、パッケージはより多くの適切なエラーメッセージを伝えるためにこの情報を使用することができます。値は静的リンクした Python は 'static'、Mac OS X framework で構築した Python は 'framework'、標準の Unix 共有ライブラリ (shared library) で構築された Python は 'shared' となります。古いバージョンの Python の場合、Mac OS 9 互換の Python では 'cfm' となります。

exception MacOS.Error

MacOS でエラーがあると、このモジュールの関数が、Mac 固有なツールボックスインターフェースモジュールから、この例外が生成されます。引数は、整数エラーコード (OSErr 値) とテキストで記述されたエラーコードです。分かっている全てのエラーコードのシンボル名は、標準モジュール *macerrors* で定義されています。

`MacOS.GetErrorString(errno)`

MacOS のエラーコード *errno* のテキスト表現を返します。

`MacOS.DebugStr(message[, object])`

Mac OS X 上では、文字列を単純に標準エラー出力に送るだけです (古いバージョンの Mac OS では、より複雑な機能が使用できました)。しかし、低水準のデバグ (gdb など) 用にブレークポイントを設定する場所も適切に用意しています。

注釈: 64 ビットモードでは使用できません。

`MacOS.SysBeep()`

ベルを鳴らします。

注釈: 64 ビットモードでは使用できません。

`MacOS.GetTicks()`

システム起動時からのティック数 (clock ticks、1/60 秒) を得ます。

`MacOS.GetCreatorAndType(file)`

2 つの 4 文字の文字列としてファイルクリエイターおよびファイルタイプを返します。 *file* 引数はパスもしくは、FSSpec、FSRef オブジェクトを与える事ができます。

注釈: FSSpec は 64 ビットモードでは使うことができません。

`MacOS.SetCreatorAndType(file, creator, type)`

ファイルクリエイターおよびファイルタイプを設定します。 *file* 引数はパスもしくは、FSSpec、FSRef オブジェクトを与える事ができます。 *creator* と *type* は 4 文字の文字列が必要です。

注釈: FSSpec は 64 ビットモードでは使うことができません。

`MacOS.openrf(name[, mode])`

ファイルのリソースフォークを開きます。引数は組み込み関数 `open()` と同じです。返されたオブジェクトはファイルのように見えるかもしれませんが、これは Python のファイルオブジェクトではありませんので扱いに微妙な違いがあります。

`MacOS.WMAvailable()`

現在のプロセスがウィンドウマネージャへのアクセスを持っているかどうかをチェックします。メソッドはウィンドウマネージャが存在しない場合は `False` を返します。これは例えば Mac OS X サーバー上で動作している、あるいは SSH でログインしている、もしくは現在のインタープリタがフルブローンアプリケーションバンドル (fullblown application bundle) から起動されていないなどの場合に起こり

ます。スクリプトは `python` ではなく `pythonw` で開始する際か、アプレットとして実行する際のどちらかの場合に、アプリケーションバンドルから起動します。

`MacOS.splash([resourceid])`

リソース id でスプラッシュスクリーンを開きます。スプラッシュスクリーンを閉じるには resourceid 0 を使います。

注釈: 64 ビットモードでは使用できません。

37.3 macostools — ファイル操作を便利にするルーチン集

このモジュールには、Macintosh 上でのファイル操作を便利にするためのルーチンがいくつか入っています。全てファイルパラメタは、パス名か `FSRef` または `FSSpec` オブジェクトで指定できます。このモジュールは、リソースフォークつきファイル (forked file) をサポートするファイルシステムを想定しているので、UFS パーティションに使ってはなりません。

注釈: このモジュールは Python 3 で削除されました。

`macostools` モジュールでは以下の関数を定義しています。

`macostools.copy(src, dst[, createpath[, copytypes]])`

ファイル `src` を `dst` へコピーします。 `createpath` が真なら、必要に応じて `dst` に至るまでのフォルダを作成します。このメソッドはデータとリソースフォーク、そしていくつかのファインダ情報 (クリエータ、タイプ、フラグ) をコピーします。オプションの `copytypes` を指定すると、作成日、修正日、バックアップ日の情報のコピー (デフォルトではコピーします) を制御できます。カスタムアイコン、コメント、アイコン位置はコピーされません。

注釈: この関数は 64 bit モードで利用できない API を使用するので、64 bit のコードでは動きません。

`macostools.copytree(src, dst)`

`src` から `dst` へ再帰的にファイルのツリーをコピーします。必要に応じてフォルダを作成してゆきます。`src` と `dst` はパス名で指定しなければなりません。

注釈: この関数は 64 bit モードで利用できない API を使用するので、64 bit のコードでは動きません。

`macostools.mkalias(src, dst)`

`src` を示すファインダエイリアス `dst` を作成します。

注釈: この関数は 64 bit モードで利用できない API を使用するので、64 bit のコードでは動きません。

`macostools.touched(dst)`

ファイル *dst* のクリエータやタイプなどのファインダ情報が変わったことをファインダに知らせます。ファイルはパス名か `FSSpec` で指定できます。この呼び出しは、ファインダにアイコンを再描画させるよう命令します。

バージョン 2.6 で非推奨: この関数は OS X では何もしません。

`macostools.BUFSIZ`

`copy` に用いるバッファサイズで、デフォルトは 1 メガバイトです。

Apple のドキュメントでは、ファインダエイリアスの作成プロセスを規定していません。そのため、`mkalias()` で作成したエイリアスが互換性のない振る舞いをする可能性があるので注意してください。

37.4 findertools — finder の Apple Events インターフェース

このモジュールのルーチンを使うと、Python プログラムからファインダが持ついくつかの機能へアクセスできます。これらの機能はファインダへの `AppleEvent` インタフェースのラップとして実装されています。

全てのファイルとフォルダのパラメータは、フルパス名、あるいは `FSRef` か `FSSpec` オブジェクトで指定できます。

`findertools` モジュールは以下の関数を定義しています。

`findertools.launch(file)`

ファインダに *file* を起動するように命令します。起動が意味するものは *file* に依存します。アプリケーションなら起動しますし、フォルダなら開かれ、文書なら適切なアプリケーションで開かれます。

`findertools.Print(file)`

ファインダにファイルを印刷するよう命令します。実際の動作はファイルを選択し、ファインダのファイルメニューから印刷コマンドを使うのと同じです。

`findertools.copy(file, destdir)`

ファインダにファイルかフォルダである *file* をフォルダ *destdir* にコピーするよう命令します。この関数は新しいファイルを示す `Alias` オブジェクトを返します。

`findertools.move(file, destdir)`

ファインダにファイルかフォルダである *file* をフォルダ *destdir* に移動するよう命令します。この関数は新しいファイルを示す `Alias` オブジェクトを返します。

`findertools.sleep()`

マシンがサポートしていれば、ファインダに Macintosh をスリープさせるよう命令します。

`findertools.restart()`

ファインダに、マシンを適切に再起動するよう命令します。

```
findertools.shutdown()
```

ファインダに、マシンを適切にシャットダウンするよう命令します。

37.5 EasyDialogs — 基本的な Macintosh ダイアログ

EasyDialogs モジュールには、Macintosh で単純なダイアログ操作を行うためのルーチンが入っています。ダイアログはドックに現れる別のアプリケーションとして起動され、ダイアログが表示されるにはクリックされなければなりません。全てのルーチンは、オプションとしてリソース ID パラメタ *id* をとります。デフォルトの DIALOG のリソース (タイプとアイテムナンバの両方) が一致するようなダイアログがあれば、*id* を使ってダイアログ操作に使われるダイアログオブジェクト情報を上書きできます。詳細はソースコードを参照してください。

注釈: このモジュールは Python 3.x で削除されました。

EasyDialogs モジュールでは以下の関数を定義しています。

```
EasyDialogs.Message(str[, id[, ok]])
```

メッセージテキスト *str* 付きのモーダルダイアログを表示します。テキストの長さは最大 255 文字です。ボタンのテキストはデフォルトでは "OK" ですが、文字列の引数 *ok* を指定して変更できます。ユーザが "OK" ボタンをクリックすると処理を戻します。

```
EasyDialogs.AskString(prompt[, default[, id[, ok[, cancel]]]])
```

ユーザに文字列値の入力を促すモーダルダイアログを表示します。 *prompt* はプロンプトメッセージで、オプションの *default* 引数は入力文字列の初期値です (指定しなければ "" を使います)。 "OK" と "Cancel" ボタンの文字列は *ok* と *cancel* の引数で変更できます。文字列の長さは全て最大 255 文字です。入力された文字列か、ユーザがキャンセルした場合には *None* を返します。

```
EasyDialogs.AskPassword(prompt[, default[, id[, ok[, cancel]]]])
```

ユーザに文字列値の入力を促すモーダルダイアログを表示します。 *AskString()* に似ていますが、ユーザの入力したテキストは点で表示されます。引数は *AskString()* のものと同じ意味です。

```
EasyDialogs.AskYesNoCancel(question[, default[, yes[, no[, cancel[, id]]]])
```

プロンプト *question* と "Yes"、"No"、"Cancel" というラベルの 3 つボタンが付いたダイアログを表示します。ユーザが "Yes" を押した場合には 1 を、"No" ならば 0 を、"Cancel" ならば -1 を返します。RETURN キーを押した場合は *default* の値 (*default* を指定しない場合は 0) を返します。ボタンのテキストはそれぞれ引数 *yes*、*no*、*cancel* で変更できます。ボタンを表示したくなければ対応する引数に "" を指定します。

```
EasyDialogs.ProgressBar([title[, maxval[, label[, id]]]])
```

プログレスバー付きのモードレスダイアログを表示します。これは後で述べる *ProgressBar* クラスのコンストラクタです。 *title* はダイアログに表示するテキスト文字列 (デフォルトの値は "Working...") で、*maxval* は処理が完了するときの値です (デフォルトは 0 で、残りの作業量が不確定であることを示します)。 *label* はプログレスバー自体の上に表示するテキストです。

```
EasyDialogs.GetArgv([optionlist[, commandlist[, addoldfile[, addnewfile[, addfolder[, id]]]])
```


コマンドライン引数リストの作成を補助するためのダイアログを表示します。得られた引数リストを `sys.argv` の形式にします。これは `getopt.getopt()` の引数として渡すのに適した形式です。`addoldfile`、`addnewfile`、`addfolder` はブール型の引数です。これらの引数が真の場合、それぞれ、実在のファイル、まだ(おそらく)存在しないファイル、フォルダへのパスをコマンドラインのパスとして設定できます。(注意: `getopt.getopt()` がファイルやフォルダ引数を認識できるようにするためには、オプションの引数がそれらより前に現れるようにしなければなりません。) 空白を含む引数は、空白をシングルクォートあるいはダブルクォートで囲んで指定できます。ユーザが "Cancel" ボタンを押した場合、`SystemExit` 例外を送出します。

`optionlist` には、ポップアップメニューで選べる選択肢を定義したリストを指定します。ポップアップメニューの要素には、次の 2 つの形式、`optstr` または `(optstr, descr)` があります。`descr` に短い説明文字列を指定すると、該当の選択肢をポップアップメニューで選択している間その文字列をダイアログに表示します。`optstr` とコマンドライン引数の対応を以下に示します:

<code>optstr</code> 形式	コマンドライン形式
<code>x</code>	<code>-x</code> (short option)
<code>x: あるいは x=</code>	<code>-x</code> (short option with value)
<code>xyz</code>	<code>--xyz</code> (long option)
<code>xyz: あるいは xyz=</code>	<code>--xyz</code> (long option with value)

`commandlist` は `cmdstr` あるいは `(cmdstr, descr)` の形のアイテムからなるリストです。`descr` は上と同じです。`cmdstr` はポップアップメニューに表示されます。メニューを選択すると `cmdstr` はコマンドラインに追加されますが、それに続く `:` や `=` は (存在していれば) 取り除かれます。

バージョン 2.0 で追加.

```
EasyDialogs.AskFileForOpen ([message] [, typeList] [, defaultLocation] [, defaultOptionFlags]
                               [, location] [, clientName] [, windowTitle] [, actionButtonLa-
                               bel] [, cancelButtonLabel] [, preferenceKey] [, popupExtension]
                               [, eventProc] [, previewProc] [, filterProc] [, wanted] )
```

どのファイルを開くかをユーザに尋ねるダイアログを表示し、ユーザが選択したファイルを返します。ユーザがダイアログをキャンセルした場合には `None` を返します。`message` はダイアログに表示するテキストメッセージです。`typeList` は選択できるファイルタイプを表す 4 文字の文字列からなるリスト、`defaultLocation` は最初に表示するフォルダで、パス名、`FSSpec` あるいは `FSRef` で指定します。`location` はダイアログを表示するスクリーン上の位置 (`x`, `y`) です。`actionButtonLabel` は OK ボタンの位置に "Open" の代わりに表示する文字列、`cancelButtonLabel` は "Cancel" ボタンの位置に "Cancel" の代わりに表示する文字列です。`wanted` は返したい値のタイプで、`str`、`unicode`、`FSSpec`、`FSRef` およびそれらのサブタイプを指定できます。

その他の引数の説明については Apple Navigation Services のドキュメントと `EasyDialogs` のソースコードを参照してください。


```
EasyDialogs.AskFileForSave ([message] [, savedFileName] [, defaultLocation] [, defaultOptionFlags] [, location] [, clientName] [, windowTitle] [, actionButtonLabel] [, cancelButtonLabel] [, preferenceKey] [, popupExtension] [, fileType] [, fileCreator] [, eventProc] [, wanted]
```

)
保存先のファイルをユーザに尋ねるダイアログを表示して、ユーザが選択したファイルを返します。ユーザがダイアログをキャンセルした場合には *None* を返します。 *savedFileName* は保存先のファイル名 (戻り値) のデフォルト値です。その他の引数の説明については [AskFileForOpen\(\)](#) を参照してください。

```
EasyDialogs.AskFolder ([message] [, defaultLocation] [, defaultOptionFlags] [, location] [, clientName] [, windowTitle] [, actionButtonLabel] [, cancelButtonLabel] [, preferenceKey] [, popupExtension] [, eventProc] [, filterProc] [, wanted])
```

フォルダの選択をユーザに促すダイアログを表示して、ユーザが選択したフォルダを返します。ユーザがダイアログをキャンセルした場合には *None* を返します。引数についての説明は [AskFileForOpen\(\)](#) を参照してください。

参考:

[Navigation Services Reference](#) Programmer's reference documentation の Carbon framework の Navigation Services の項。

37.5.1 プログレスバーオブジェクト

ProgressBar オブジェクトでは、モードレスなプログレスバーダイアログのサポートを提供しています。定量プログレスバー (温度計スタイル) と不定量プログレスバー (床屋の螺旋看板スタイル) がサポートされています。プログレスバーの最大値がゼロ以上の場合には定量インジケータに、そうでない場合は不定量インジケータになります。

バージョン 2.2 で変更: 不定量プログレスバーのサポートを追加しました。

ダイアログは作られるとすぐに表示されます。ダイアログの "Cancel" ボタンを押すか、Cmd-*.* (コマンドキーを押しながらピリオドを押す) か、あるいは ESC をタイプすると、ダイアログウィンドウを非表示にして *KeyboardInterrupt* を送出します (ただし、この応答は次にプログレスバーを更新するときまで、すなわち次に *inc()* または *set()* を呼び出してダイアログを更新するまで発生しません)。それ以外の場合、プログレスバーは *ProgressBar* オブジェクトを廃棄するまで表示されたままになります。

ProgressBar オブジェクトには以下の属性とメソッドがあります。

ProgressBar.**curval**

プログレスバーの現在の値 (整数型あるいは長整数型) です。プログレスバーの通常のアクセスのメソッドによって *curval* を 0 と *maxval* の間にします。この属性を直接変更してはなりません。

ProgressBar.**maxval**

プログレスバーの最大値 (整数型あるいは長整数型) です; プログレスバー (温度計スタイル) では、*curval* が *maxval* に等しい時に全量に到達します。 *maxval* が 0 の場合、不定量プログレスバー (床屋の螺旋看板スタイル) になります。この属性を直接変更してはなりません。

`ProgressBar.title([newstr])`

プログレスダイアログのタイトルバーのテキストを *newstr* に設定します。

`ProgressBar.label([newstr])`

プログレスダイアログ中のプログレスボックスのテキストを *newstr* に設定します。

`ProgressBar.set(value[, max])`

プログレスバーの現在値 *curval* を *value* に設定します。 *max* も指定した場合、 *maxval* を *max* にします。 *value* は前もって 0 と *maxval* の間になるよう強制的に設定されます。温度計バーの場合、変更内容を反映するよう表示を更新します。変更によって定量プログレスバーから不定量プログレスバーへ、あるいはその逆への推移が起こります。

`ProgressBar.inc([n])`

プログレスバーの *curval* を *n* だけ増やします。 *n* を指定しなければ 1 だけ増やします。 (*n* は負にもでき、その場合は *curval* を減少させます。) 変更内容を反映するようプログレスバーの表示を更新します。プログレスバーが不定量プログレスバーの場合、床屋の螺旋看板模様を 1 度「回転」させます。増減によって *curval* が 0 から *maxval* までの範囲を超えた場合、 0 と *maxval* の範囲に収まるよう強制的に値を設定します。

37.6 FrameWork — 対話型アプリケーション・フレームワーク

FrameWork モジュールは、対話型 Macintosh アプリケーションのクラスで、同時にフレームワークを提供します。プログラマは、サブクラスを作って基底クラスの様々なメソッドをオーバーライドし、必要な機能を実装することでアプリケーションを組み立てられます。機能のオーバーライドは、時によって様々な異なるレベルで行われます。つまり、ある一つのダイアログウィンドウでクリックの処理を普段と違う方法で行うには、完全なイベント処理をオーバーライドする必要はありません。

注釈: このモジュールは Python 3.x で削除されました。

FrameWork の開発は事実上停止しています。現在では PyObjC を使用すれば Python から Cocoa の全機能を使用することができます。このドキュメントでは最も重要な機能だけしか記述していませんし、それさえも論理的な形で書かれてもいません。ソースが例題を詳しく見てください。次にあげるのは、MacPython ニュースグループにポストされたコメントで、*FrameWork* の強力さと限界について述べています。

FrameWork の最大の強みは、制御の流れをたくさんの異なる部分に分割できることです。例えば *W* を使って、いろいろな方法でメニューをオン/オフしたり、残りをいじらずにうまくプラグインさせることができます。*FrameWork* の弱点は、コマンドインタフェースが抽象化されていないこと (といっても難しいわけではないですが)、ダイアログサポートが最低限しかないこと、それからコントロール/ツールバーサポートが全くないことです。

FrameWork モジュールは以下の関数を定義しています:

`FrameWork.Application()`

アプリケーション全体を表現しているオブジェクト。メソッドについての詳細は以下の記述を参照し

てください。デフォルト `__init__()` ルーチンは、空のウィンドウ辞書とアップルメニュー付きのメニューバーを作成します。

`FrameWork.MenuBar()`

メニューバーを表現するオブジェクト。このオブジェクトは普通はユーザは作成しません。

`FrameWork.Menu (bar, title[, after])`

メニューを表現するオブジェクト。生成時には、メニューが現われる `MenuBar` と、`title` 文字列、メニューが表示されるべき (1 から始まる) 位置 `after` (デフォルトは末尾) を渡します。

`FrameWork.MenuItem (menu, title[, shortcut, callback])`

メニューアイテムオブジェクトを作成します。引数は作成するメニューと、アイテムのタイトル文字列、オプションのキーボードショートカット、コールバックルーチンです。コールバックは、メニュー ID、メニュー内のアイテム番号 (1 から数える)、現在のフロントウィンドウ、イベントレコードを引数に呼ばれます。

呼び出し可能なオブジェクトのかわりに、コールバックは文字列でも良いです。この場合、メニューの選択は、最前面のウィンドウとアプリケーションの中でメソッド探索を引き起こします。メソッド名は、コールバック文字列の前に `'domenu_'` を付けたものです。

`MenuBar` の `fixmenudimstate()` メソッドを呼び出すと、現在のフロントウィンドウにもとづいて、適切なディム化を全てのメニューアイテムに対して施します。

`FrameWork.Separator (menu)`

メニューの最後にセパレータを追加します。

`FrameWork.SubMenu (menu, label)`

`label` の名前のサブメニューを、メニュー `menu` の下に作成します。メニューオブジェクトが返されます。

`FrameWork.Window (parent)`

(モードレス) ウィンドウを作成します。 `Parent` は、ウィンドウが属するアプリケーションオブジェクトです。作成されたウィンドウはまだ表示されません。

`FrameWork.DialogWindow (parent)`

モードレスダイアログウィンドウを作成します。

`FrameWork.windowbounds (width, height)`

与えた幅と高さのウィンドウを作成するのに必要な、 (`left`, `top`, `right`, `bottom`) からなるタプルを返します。ウィンドウは以前のウィンドウに対して位置をずらして作成され、全体のウィンドウが画面からなるべく外れないようにします。しかし、ウィンドウはいつでも与えたのと全く同じサイズで、そのため一部は画面から隠れる場合もあります。

`FrameWork.setwatchcursor()`

マウスカーソルを時計型に設定します。

`FrameWork.setarrowcursor()`

マウスカーソルを矢印型に設定します。

37.6.1 Application オブジェクト

Application オブジェクトのメソッドは各種ありますが、次のメソッドをあげておきます。

`Application.makeusermenus()`

アプリケーションでメニューを使う必要がある場合、このメソッドをオーバーライドします。属性 `menubar` にメニューを追加します。

`Application.getabouttext()`

このメソッドをオーバーライドすることで、アプリケーションの説明を記述するテキスト文字列を返します。代わりに、`do_about()` メソッドをオーバーライドすれば、もっと凝った "about" (...について) メッセージを出す事ができます。

`Application.mainloop([mask[, wait]])`

このルーチンがメインイベントループで、作成したアプリケーションが動き出すためにはこれと呼ぶことになります。 *Mask* は操作したいイベントを選択するマスクです。 *wait* は並行に動作しているアプリケーションに割り当てたいティック数 (1/60 秒) です (デフォルトで 0 ですが、あまり良い値ではありません)。 *self* フラグを立ててメインループを抜ける方法はまだサポートされていますが、これはお勧めできません。代わりに `self._quit()` を呼んでください。

イベントループは小さなパーツに分割されていて、各々をオーバーライドできるようになっています。これらのメソッドは、デフォルトでウィンドウとダイアログや、ドラッグとリサイズの手続き、AppleEvent、非 Framework のウィンドウに関するウィンドウの操作などに関するイベントをディスパッチすることなどまで面倒をみてくれます。

原則として、全てのイベントハンドラは、イベントが完全に取扱われた場合は 1 を返さなくてはなりませんし、それ以外では 0 を返さなくてはなりません (例えば、前面のウィンドウは Framework ウィンドウではない場合を考えてください)。こうしなくてはいけない理由は、アップデートイベントなどが Sioux コンソールウィンドウなどの他のウィンドウにきちんと渡されるようにするためです。 *our_dispatch* やその呼び出し元の内部から `MacOS.HandleEvent()` を呼んではいけません。そうしたコードが Python の内部ループのイベントハンドラを経由して呼ばれると、無限ループになりかねないからです。

`Application.asynccevents(onoff)`

非同期でイベント操作をしたい場合は、非ゼロの引数でこのメソッドを呼んでください。こうすることで、イベントが生じた時に、内部のインタプリタのループで、アプリケーションイベントハンドラ *async_dispatch* が呼ばれることになります。すると、長時間の計算を行っている場合でも、Framework ウィンドウがアップデートされ、ユーザーインターフェースが動き続けるようになります。ただし、インタプリタの動作が減速し、非リアルタイムのコード (例えば Framework 自身など) に奇妙な動作が見られるかもしれません。デフォルトでは *async_dispatch* はすぐに *our_dispatch* を呼びますが、このメソッドをオーバーライドすると、特定のイベントを非同期で操作しても良くなります。処理しないイベントは Sioux などに渡されることになります。

戻り値は以前の on あるいは off 値です。

`Application._quit()`

実行中の *mainloop()* 呼び出しを、次の適当なタイミングで終了させます。

`Application.do_char(c, event)`

ユーザーが文字 *c* をタイプした時に呼ばれます。イベントの全詳細は `event` 構造体の中にあります。このメソッドは "Window" オブジェクト内で使うためにも提供されています。このオブジェクトのウィンドウが最前面にある場合にアプリケーション全体としてのハンドラとして使われます。

`Application.do_dialogevent(event)`

イベントループ内部で最初のほうで呼ばれて、モードレスダイアログイベントを処理します。デフォルトではメソッドは単にイベントを適切なダイアログにディスパッチするだけです (関連した `DialogWindow` オブジェクトを経由してではありません)。特別にダイアログイベント (キーボードショートカットなど) を処理する必要がある場合にオーバーライドしてください。

`Application.idle(event)`

イベントが無い場合にメインイベントループから呼ばれます。 `null` イベントも渡されます (つまりマウス位置などを監視することができます)。

37.6.2 Window オブジェクト

Window オブジェクトは特に次のメソッドを持ちます:

`Window.open()`

ウィンドウを開く時はこのメソッドをオーバーライドします。Mac OS ウィンドウ ID を `self.wid` に入れて `do_postopen()` メソッドを呼ぶと、親アプリケーションにウィンドウを登録します。

`Window.close()`

ウィンドウを閉じるときに特別な処理をする場合はこのメソッドをオーバーライドします。親アプリケーション状態をクリーンアップするには、`do_postclose()` を呼びます。

`Window.do_postresize(width, height, macoswindowid)`

ウィンドウがリサイズされた後に呼ばれます。 `InvalidRect` を呼び出す以外にもすることがある場合はこれをオーバーライドします。

`Window.do_contentclick(local, modifiers, event)`

ウィンドウのコンテンツ部分をユーザーがクリックすると呼ばれます。引数は位置座標 (ウィンドウを基準)、キーモディファイア、生のイベントです。

`Window.do_update(macoswindowid, event)`

ウィンドウの更新イベントが受信された時に呼ばれます。ウィンドウを再描画します。

`Window.do_activate(activate, event)`

ウィンドウがアクティブ化 (`activate == 1`)、非アクティブ化 (`activate == 0`) する際に呼ばれます。フォーカスのハイライトなどを処理します。

37.6.3 ControlsWindow オブジェクト

ControlsWindow オブジェクトには Window オブジェクトのメソッドの他に次のメソッドがあります:

`ControlsWindow.do_controlhit (window, control, pcode, event)`

コントロール *control* のパートコード *pcode* がユーザにヒットされた場合に呼ばれます。トラッキングなどは任せておいてかまいません。

37.6.4 ScrolledWindow オブジェクト

ScrolledWindow オブジェクトは、次のメソッドを追加した ControlsWindow オブジェクトです。

`ScrolledWindow.scrollbars ([wantx[, wanty]])`

水平スクロールバーと垂直スクロールバーを作成します (あるいは破棄します)。引数はどちらが欲しいか指定します (デフォルトは両方)。スクロールバーは常に最小値 0、最大値 32767 です。

`ScrolledWindow.getscrollbarvalues ()`

このメソッドは必ず作っておかなくてははいけません。現在のスクロールバーの位置を与えるタプル (*x*, *y*) を (0 の 32767 間で) 返してください。バーの方向について文書全体が可視状態であること知らせるため `None` を返す事もできます。

`ScrolledWindow.updatescrollbars ()`

文書に変更があった場合はこのメソッドを呼びます。このメソッドは `getscrollbarvalues ()` を呼んでスクロールバーを更新します。

`ScrolledWindow.scrollbar_callback (which, what, value)`

あらかじめ与えておくメソッドで、ユーザーとの対話により呼ばれます。*which* は 'x' か 'y'、*what* は '-', '--', 'set', '++', '+' のどれかです。'set' の場合は、*value* に新しいスクロールバー位置を入れておきます。

`ScrolledWindow.scalebarvalues (absmin, absmax, curmin, curmax)`

`getscrollbarvalues ()` の結果から値を計算するのを助ける補助的なメソッドです。文書の最小値と最大値、可視部分に関する上端値 (左端値) と下端値 (右端値) を渡すと、正しい数か `None` を返します。

`ScrolledWindow.do_activate (onoff, event)`

ウィンドウが最前面になった時、スクロールバーのディム (dimming)/ハイライトの面倒をみます。このメソッドをオーバーライドするなら、オーバーライドしたメソッドの最後でオリジナルのメソッドを呼んでください。

`ScrolledWindow.do_postresize (width, height, window)`

スクロールバーを正しい位置に移動させます。オーバーライドする時は、オーバーライドしたメソッドの一番最初でオリジナルのメソッドを呼んでください。

`ScrolledWindow.do_controlhit (window, control, pcode, event)`

スクロールバーのインタラクションを処理します。これをオーバーライドする時は、オリジナルのメソッドを最初に呼び出してください。非ゼロの返り値はスクロールバー内がヒットされたことを意味し、実際に処理が進むことになります。

37.6.5 DialogWindow オブジェクト

DialogWindow オブジェクトには、Window オブジェクトのメソッドの他に次のメソッドがあります:

`DialogWindow.open(resid)`

ID *resid* の DLOG リソースからダイアログウィンドウを作成します。ダイアログオブジェクトは `self.wid` に保存されます。

`DialogWindow.do_itemhit(item, event)`

アイテム番号 *item* がヒットされた時に呼ばれます。トグルボタンなどの再描画は自分で処理してください。

37.7 autoGIL — イベントループ中のグローバルインタープリタの取り扱い

`autoGIL` モジュールは、イベントループを実行する際に、自動的に Python のグローバルインタープリタロック (*Global Interpreter Lock*) をロックしたり解除したりするための関数 `installAutoGIL()` を提供します。

注釈: このモジュールは Python 3.x で削除されました。

exception `autoGIL.AutoGILError`

例えば現在のスレッドがループしていないなど、オブザーバにコールバックをインストールできない場合に発生します。

`autoGIL.installAutoGIL()`

現在のスレッドのイベントループ (CFRunLoop) 中のオブザーバにコールバックをインストールし、適切な時にグローバルインタープリタロック (GIL) を、イベントループがアイドルの間、他の Python スレッドの起動ができるようにロックしたり、ロックの解除をしたりします。

利用できる環境: OSX 10.1 以降

37.8 Mac OS ツールボックスモジュール

各種のレガシーな Mac OS ツールボックスへのインターフェースを与えるモジュール群があります。対応するモジュールがあるなら、そのモジュールではツールボックスで宣言された各種の構造体の Python オブジェクトが定義され、操作は定義されたオブジェクトのメソッドとして実装されています。その他の操作はモジュールの関数として実装されています。C で可能な操作がすべて Python で可能なわけではありません (コールバックはよく問題になります)、パラメータが Python だと違ってしまうことはよくあります (特に入力バッファや出力バッファ)。全てのメソッドと関数は `__doc__` 文字列があるので、引数と返り値の説明を得る事ができます。他の情報源としては、[Inside Macintosh](#)などを参照してください。

これらのモジュールは全て Carbon パッケージに含まれています。この名前にもかかわらずそれら全てが Carbon フレームワークの一部ではありません。CF は、CoreFoundation フレームワークの中に実際は

ありますし、Qt は QuickTime フレームワークにあります。ツールボックスモジュールは普通以下のようにして利用します:

```
from Carbon import AE
```

注釈: 最新バージョンの OS X では、このモジュールが使っている OS X API のほとんどが非推奨になっているか、もしくは削除されています。多くは Python が 64 ビットモードで動作している場合には利用出来ません。Carbon モジュール群は Python 3 では削除済みです。Python 2 でもこれらのモジュールの利用は避けるべきです。

37.8.1 Carbon.AE — Apple Events

37.8.2 Carbon.AH — Apple ヘルプ

37.8.3 Carbon.App — Appearance Manager

37.8.4 Carbon.Appearance — Appearance Manager 定数

37.8.5 Carbon.CF — Core Foundation

CFBase, CFArray, CFData, CFDictionary, CFString と CFURL オブジェクトがいくらか部分的にサポートされています。

37.8.6 Carbon.CG — Core Graphics

37.8.7 Carbon.CarbonEvt — Carbon Event Manager

37.8.8 Carbon.CarbonEvents — Carbon Event Manager 定数

37.8.9 Carbon.Cm — Component Manager

37.8.10 Carbon.Components — Component Manager 定数

37.8.11 Carbon.ControlAccessor — Control Manager アクセッサ

37.8.12 Carbon.Controls — Control Manager 定数

37.8.13 Carbon.CoreFounation — CoreFounation 定数

37.8.14 Carbon.CoreGraphics — CoreGraphics 定数

37.8.15 Carbon.Ctl — Control Manager

37.8.16 Carbon.Dialogs — Dialog Manager 定数

37.8.17 Carbon.Dlg — Dialog Manager

37.8.18 Carbon.Drag — Drag and Drop Manager

37.8.19 Carbon.Dragconst — Drag and Drop Manager 定数

37.8.20 Carbon.Events — Event Manager 定数

37.8.21 Carbon.Evt — Event Manager

37.8.22 Carbon.File — File Manager

37.8.23 Carbon.Files — File Manager 定数

37.8.24 Carbon.Fm — Font Manager

37.8.25 Carbon.Folder — Folder Manager

37.8.26 Carbon.Folders — Folder Manager 定数

37.8.27 Carbon.Fonts — Font Manager 定数

37.8.28 Carbon.Help — Help Manager

スクラップマネージャは Macintosh 上でのカット & ペースト操作の最もシンプルな形式をサポートします。アプリケーション間とアプリケーション内での両方のクリップボード操作が可能です。

Scrap モジュールはスクラップマネージャの関数へのローレベルでのアクセスを提供します。以下の関数が定義されています:

`Carbon.Scrap.InfoScrap()`

スクラップについて現在の情報を返します。この情報は (size, handle, count, state, path) を含むタプルでエンコードされます。

フィールド	意味
<i>size</i>	スクラップのサイズをバイト数で示したもの。
<i>handle</i>	スクラップを表現するリソースオブジェクト。
<i>count</i>	スクラップの内容のシリアルナンバー。
<i>state</i>	整数。メモリー内にあるなら正、ディスク上にあるなら 0、初期化されていないなら負。
<i>path</i>	ディスク上に保存されているなら、そのスクラップのファイルネーム。

参考:

[Scrap Manager](#) Apple のスクラップマネージャに関する文書には、アプリケーションでスクラップマネージャを使用する上での便利な情報がたくさんあります。

37.8.53 `Carbon.Snd` — Sound Manager

37.8.54 `Carbon.Sound` — Sound Manager 定数

37.8.55 `Carbon.TE` — TextEdit

37.8.56 `Carbon.TextEdit` — TextEdit 定数

37.8.57 `Carbon.Win` — Window Manager

37.8.58 `Carbon.Windows` — Window Manager 定数

37.9 `ColorPicker` — 色選択ダイアログ

`ColorPicker` モジュールは標準色選択ダイアログへのアクセスを提供します。

注釈: このモジュールは Python 3.x で削除されました。

`ColorPicker.GetColor(prompt, rgb)`

標準色選択ダイアログを表示し、ユーザが色を選択することを可能にします。 *prompt* の文字列によりユーザに指示を与えられ、デフォルトの選択色を *rgb* で設定する事ができます。 *rgb* は赤、緑、青の色要素のタプルで与えてください。 `GetColor()` はユーザが選択した色のタプルと色が選択されたか、取り消されたかを示すフラグを返します。

第 38 章

MacPython OSA モジュール

この章では、オープンスクリプティングアーキテクチャ (Open Scripting Architecture、OSA、一般的には AppleScript として知られている) の現在の Python 用実装について説明します。これを使うとスクリプト可能なアプリケーションを Python プログラムから実に python らしいインタフェースとともに制御することができます。このモジュール群の開発は停止しました。

AppleScript や OSA の様々なコンポーネントの説明、およびそのアーキテクチャや用語の理解のために、Apple のドキュメントを読んでおく方がよいでしょう。"Applescript Language Guide" は概念モデルと用語を説明し、標準スイートについて文書にまとめてあります。"Open Scripting Architecture" はアプリケーションプログラマの視点から、OSA を使用する方法について説明しています。Apple ヘルプビューワにおいてこれらは Developer Documentation, Core Technologies セクションで見つかります。

アプリケーションでスクリプト制御する例として、以下の AppleScript コードは、最前面にある **Finder** のウィンドウの名前を取得して表示させます:

```
tell application "Finder"
    get name of window 1
end tell
```

Python では、以下のコードで同じ事ができます:

```
import Finder

f = Finder.Finder()
print f.get(f.window(1).name)
```

配布されている Python ライブラリには、標準スイートを実装したパッケージに加えて、いくつかのよくあるアプリケーションへのインタフェースを実装したパッケージが含まれています。

AppleEvent をアプリケーションに送るためには、最初にアプリケーションの用語 (Script Editor が「辞書」と呼んでいるもの) を話せる Python パッケージを作らなければなりません。この作業は PythonIDE の中から行うこともできますし、コマンドラインから gensuitemodule.py モジュールをスタンドアロンのプログラムとして実行することでもできます。

作成されるのはいくつものモジュールからなるパッケージで、それぞれのモジュールはプログラムで使われるスイートであり __init__ モジュールがそれらを取りまとめています。Python の継承グラフは AppleScript

の継承グラフに従っていますので、プログラムの辞書が標準スイートのサポートを含みつつ、一つ二つ動詞を追加の引数で拡張するように指定しているならば、出力されるスイートは `Standard_Suite` という `StdSuites.Standard_Suite` からすべてをインポートしてエクスポートし直しつつ追加された機能を持つようにメソッドをオーバーライドしたモジュールを含みます。 `gensuitemodule` の出力は非常に読み易く、また元々の AppleScript 辞書にあったドキュメントを Python 文書化文字列 (docstring) 中に含みますので、それを読むことは有用な情報源となります。

出力されたパッケージはパッケージと同じ名前のメインクラスを実装しており、これは全ての AppleScript 動詞を直接のオブジェクトは第 1 引数で、オプションのパラメータはキーワード引数で受けるメソッドとして含みます。 AppleScript クラスも Python クラスとして実装されたり、その他諸々も同様です。

動詞を実装しているメインの Python クラスはまた AppleScript の "application" クラスで宣言されたプロパティおよび要素へのアクセスも許します。現在のリリースではこれはオブジェクト指向的というには程遠く、上の例で見たように `f.get(f.window(1).name)` と書かねばならず、より Python らしい `f.window(1).name.get()` という書き方はできません。

AppleScript の識別子が Python の識別子として扱えない場合以下の少数のルールで変換します:

- 空白はアンダースコアに置き換えられます
- その他の英数字以外の文字は `_xx_` に置き換えられます。ここで `xx` はその文字の 16 進値です。
- Python の予約語にはアンダースコアが後ろに付けられます

Python はスクリプト可能なアプリケーションを Python で作成することもサポートしていますが、以下のモジュールは MacPython の AppleScript サポートに関係するモジュールのみです:

38.1 gensuitemodule — OSA スタブ作成パッケージ

`gensuitemodule` モジュールは AppleScript 辞書によって特定のアプリケーションに実装されている AppleScript 群のためのスタブコードを実装した Python パッケージを作成します。

このモジュールは、通常は **PythonIDE** からユーザによって起動されますが、コマンドラインからスクリプトとして実行する (オプションとしてヘルプに `--help` を与えてみてください) こともできますし、Python コードでインポートして利用する事もできます。使用例として、どのようにして標準ライブラリに含まれているスタブパッケージを生成するか、`Mac/scripts/genallsuites.py` にあるソースを見てください。

このモジュールは次の公開関数を定義しています。

`gensuitemodule.is_scriptable(application)`

`application` としてパス名を与えたアプリケーションがスクリプト可能でありそうな場合、真を返します。返り値はやや不確実な場合があります。 **Internet Explorer** はスクリプト不可能なように見えてしまいが、実際はスクリプト可能です。

`gensuitemodule.processfile(application[, output, basepkgname, edit_modnames, creatorsignature, dump, verbose])`

フルパス名として渡された `application` のためのスタブパッケージを作成します。 `.app` として一つのパッケージにまとめてあるプログラム群のために内部の実行プログラムそのものではなくパッケー

ジへのパス名を渡すだけでよくなっています。パッケージ化されていない CFM アプリケーションではアプリケーションバイナリのファイル名を渡す事もできます。

この関数は、アプリケーションの OSA 用語リソースを搜し、これらのリソースを読み取り、その結果データをクライアントスタブを実装した Python コードパッケージを作成するために使用します。

`output` は作成結果のパッケージを保存するパス名で、指定しない場合は標準の「別名で保存 (save file as)」ダイアログが表示されます。 `basepkgname` はこのパッケージの基盤となるパッケージを指定します。デフォルトは `StdSuites` になります。 `StdSuites` 自体を生成する場合だけ、このオプションを指定する必要があります。 `edit_modnames` は自動生成によって作成されてあまり綺麗ではないモジュール名を変更するために使用することができる辞書です。 `creator_signature` はパッケージ中の `PkgInfo` ファイル、あるいは CFM ファイルクリエイタ署名から通常得られる 4 文字クリエイタコードを上書きするために使用することができます。 `dump` にはファイルオブジェクトを与え、これを指定すると `processfile` はリソースを読取った後に停止し、コード化した用語リソースの Python 表現をダンプします。 `verbose` にもまたファイルオブジェクトを与え、これを指定すると `processfile` の行になっている処理の詳細を出力します。

```
gensuitemodule.processfile_fromresource(application[, output, basepkgname,
                                         edit_modnames, creatorsignature, dump,
                                         verbose])
```

この関数は、用語リソースを得るのに異なる方法を使用する以外は、`processfile` と同じです。この関数では、リソースファイルとして `application` を開き、このファイルから "aete" および "aet" リソースをすべて読み込む事で、AppleScript 用語リソース読み込みを行ないます。

38.2 aetools — OSA クライアントのサポート

`aetools` モジュールは Python で AppleScript クライアントとしての機能をサポートするアプリケーションを構築するための基本的な機能を含んでいます。さらに、このモジュールは、`aetypes` および `aepack` モジュールの中核機能をインポートし再エクスポートします。 `gensuitemodule` によって生成されたスタブパッケージは `aetools` のかなり適切な部分をインポートするので、通常はそれを明示的にインポートする必要はありません。生成されたパッケージ群を使用することができない場合と、スクリプト対応のためにより低いレベルのアクセスを必要としている場合、例外が発生します。

`aetools` モジュールはそれ自身、`Carbon.AE` モジュールによって提供される AppleEvent サポートを利用します。このモジュールにはウィンドウマネージャへのアクセスを必要とするという 1 つの欠点があります。詳細は `osx-gui-scripts` を見てください。この制限は将来のリリースで撤廃されるかもしれません。

注釈: このモジュールは Python 3.x で削除されました。

`aetools` モジュールは下記の関数を定義しています。

`aetools.packevent(ae, parameters, attributes)`

あらかじめ作成された `Carbon.AE.AEDesc` オブジェクト中のパラメータおよび属性を保存します。 `parameters` と `attributes` は Python オブジェクトの 4 文字の OSA パラメータのキーを写像した辞書です。このオブジェクトをパックするには `aepack.pack()` を使います。

`aetools.unpackevent` (*ae*[, *formodulename*])

再帰的に、`Carbon.AE.AEDesc` イベントを Python オブジェクトへアンパックします。関数は引数の辞書および属性の辞書を返します。`formodulename` 引数は AppleScript クラスをどこに捜しに行くか制御するために、生成されたスタブパッケージにより使用されます。

`aetools.keysubst` (*arguments*, *keydict*)

Python キーワード引数辞書 *arguments* を、写像による 4 文字の OSA キーとして *keydict* の中で指定された Python 識別子であるキーの交換により `packevent` によって要求されるフォーマットへ変換します。生成されたパッケージ群によって使用されます。

`aetools.enumsbst` (*arguments*, *key*, *edict*)

arguments 辞書が *key* へのエントリーを含んでいる場合、辞書 *edict* のエントリーに見合う値に変換します。これは人間に判読可能のように Python 列挙名を OSA 4 文字のコードに変換します。生成されたパッケージ群によって使用されます。

`aetools` モジュールは次のクラスを定義しています。

class `aetools.TalkTo` ([*signature=None*, *start=0*, *timeout=0*])

アプリケーションとの対話に利用する代理の基底クラスです。`signature` はクラス属性 `_signature` (サブクラスによって通常設定される) を上書きした、対話するアプリケーションを定義する 4 文字 クリエイトコードです。`start` にはクラスインスタンス上でアプリケーションを実行することを可能にするために、真を設定する事ができます。`timeout` を明示的に設定する事で、`AppleEvent` の返答を待つデフォルトのタイムアウト時間を変更する事ができます。

`TalkTo._start` ()

アプリケーションが起動しているか確認し、起動していなければ起動しようとします。

`TalkTo.send` (*code*, *subcode*[, *parameters*, *attributes*])

OSA 指示子 *code*, *subcode* (いずれも通常 4 文字の文字列です) を持った変数のために、*parameters* をパックし、*attributes* に戻し、目標アプリケーションにそれを送って、返答を待ち、`unpackevent` を含んだ返答をアンパックし、`AppleEvent` の返答を返し、辞書としてアンパックした値と属性を返して、`AppleEvent Carbon.AE.AEDesc` を作成します。

38.3 aepack — Python 変数と AppleEvent データコンテナ間の変換

`aepack` モジュールは、Python 変数から `AppleEvent` デスクリプタへの変換 (パック) と、その逆に変換 (アンパック) する関数を定義しています。Python 内では `AppleEvent` デスクリプタは、組み込み型である `AEDesc` の Python オブジェクトとして扱われます。`AEDesc` は `Carbon.AE` モジュールで定義されています。

注釈: このモジュールは Python 3.x で削除されました。

`aepack` モジュールは次の関数を定義しています。

`aepack.pack` (*x*[, *forcetype*])

Python 値 *x* を変換した値を保持する `AEDesc` オブジェクトを返します。*forcetype* を与えることで、結

果のデスクリプタ型を指定できます。それ以外では、Python 型から Apple Event デスクリプタ型へのデフォルトのマッピングが使われます。マッピングは次の通りとなります:

Python の型	デスクリプタの型
FSSpec	typeFSS
FSRef	typeFSRef
Alias	typeAlias
integer	typeLong (32 ビット整数)
float	typeFloat (64 ビット浮動小数点数)
string	typeText
unicode	typeUnicodeText
list	typeAEList
dictionary	typeAERecord
instance	下記参照

x が Python インスタンスなら、この関数は `__aepack__()` メソッドを呼びだそうとします。このメソッドは `AEDesc` オブジェクトを返します。

x の変換が上で定義されていない場合は、この関数は、テキストデスクリプタとしてエンコードされた、値の (`repr()` 関数による) Python 文字列表現が返されます。

`aepack.unpack(x[, formodulename])`

x は `AEDesc` タイプのオブジェクトでなければいけません。この関数は、Apple Event デスクリプタ x のデータの Python オブジェクト表現を返します。単純な AppleEvent データ型 (整数、テキスト、浮動小数点数) の、対応する Python 型が返されます。Apple Event リストは Python リストとして返され、リストの要素は再帰的にアンパックされます。 `formodulename` の指定がない場合、オブジェクト参照 (例: line 3 of document 1) が、 `aetypes.ObjectSpecifier` のインスタンスとして返されます。デスクリプタ型が `typeFSS` である AppleEvent デスクリプタは、 `FSSpec` オブジェクトとして返されます。 AppleEvent レコードデスクリプタは、再帰的にアンパックされた、型の 4 文字キーと要素を持つ Python 辞書として返されます。

オプションの `formodulename` 引数は `gensuitemodule` より作成されるスタブパッケージにより利用され、オブジェクト指定子のための OSA クラスをモジュールの中で見つけられることを保証します。これは、例えば、ファインダがウィンドウに対してオブジェクト指定子を返す場合、 `Finder.Window` のインスタンスが得られ、 `aetypes.Window` が得られないことを保証します。前者は、ファインダ上のウィンドウが持っている、すべての特性および要素のことを知っています。一方、後者のものはそれらのことを知りません。

参考:

`Carbon.AE` モジュール Apple Event マネージャー・チェーンへの組み込みアクセス

`aetypes` モジュール Apple Event デスクリプタ型としてコードされた Python 定義

38.4 aetypes — AppleEvent オブジェクト

`aetypes` では、Apple Event データデスクリプタ (data descriptor) や Apple Event オブジェクト指定子 (object specifier) を表現するクラスを定義しています。

Apple Event データはデスクリプタに含まれていて、これらのデスクリプタは型付けされています。多くのデスクリプタは、単に対応する Python の型で表現されています。例えば、OSA 中の `typeText` は Python 文字列型で、`typeFloat` は浮動小数点型になる、といった具合です。このモジュールでは、OSA の型のうち、直接的に対応する Python の型がないもののためにクラスを定義しています。そのようなクラスのインスタンスに対するパックやアンパック操作は、`aepack` モジュール自動的に処理します。

オブジェクト指定子は、本質的には Apple Event サーバ中に実装されているオブジェクトへのアドレスです。Apple Event 指定子は、Apple Event のオブジェクトそのものとして、あるいはオプションパラメタの引数として使われます。`aetypes` モジュールには OSA クラスやプロパティを表現するための基底クラスが入っています。これらのクラスは、`gensuitemodule` が生成するパッケージ内で、目的に応じてクラスやプロパティを増やす際に使われます。

以前のバージョンとの互換性や、スタブパッケージを生成していないようなアプリケーションをスクリプトで書く必要がある場合のために、このモジュールには `Document`、`Window`、`Character`、といったよく使われる OSA クラスのいくつかを指定できるオブジェクト指定子も入っています。

注釈: このモジュールは Python 3.x で削除されました。

`AEObjects` モジュールでは、以下のようなクラスを定義して、Apple Event デスクリプタデータを表現しています:

class `aetypes.Unknown` (*type, data*)

`aepack` や `aetypes` がサポートしていない OSA の デスクリプタデータ、すなわち、このモジュールで扱っている他のクラスや、Python の組み込み型の値で表現されていないようなデータを表現するクラスです。

class `aetypes.Enum` (*enum*)

列挙値 (enumeration value) を表すクラスです。値は 4 文字の文字列型になります。

class `aetypes.InsertionLoc` (*of, pos*)

オブジェクト *of* の中の *pos* の位置を表すクラスです。

class `aetypes.Boolean` (*bool*)

ブール値 (真偽値) を表すクラスです。

class `aetypes.StyledText` (*style, text*)

スタイル情報 (フォント、タイプフェイスなど) つきのテキストを表すクラスです。

class `aetypes.AEText` (*script, style, text*)

スクリプトシステム (script system) およびスタイル情報の入ったテキストを表すクラスです。

class `aetypes.IntlText` (*script, language, text*)

スクリプトシステムと言語情報 (language information) の入ったテキストを表すクラスです。

class `aetypes.IntlWritingCode` (*script, language*)

スクリプトシステムと言語情報を表すクラスです。

class `aetypes.QDPoint` (*v, h*)

QuickDraw の点を表すクラスです。

class `aetypes.QDRectangle` (*v0, h0, v1, h1*)

QuickDraw の矩形を表すクラスです。

class `aetypes.RGBColor` (*r, g, b*)

色を表すクラスです。

class `aetypes.Type` (*type*)

OSA の型 (type value) を表すクラスです。4 文字からなる名前を値に持ちます。

class `aetypes.Keyword` (*name*)

OSA のキーワードです。4 文字からなる名前を値に持ちます。

class `aetypes.Range` (*start, stop*)

範囲を表すクラスです。

class `aetypes.Ordinal` (*abso*)

先頭を表す "firs" や中央を表す "midd" のように、数値でない絶対位置指定子を表すクラスです。

class `aetypes.Logical` (*logc, term*)

演算子 logc を term に適用したときの論理式を表すクラスです。

class `aetypes.Comparison` (*obj1, relo, obj2*)

obj1 と obj2 の relo による比較を表すクラスです。

以下のクラスは、生成されたスタブパッケージが、AppleScript のクラスやプロパティを Python で表現する上で基底クラスとして利用します。

class `aetypes.ComponentItem` (*which[, fr]*)

OSA クラス用の抽象基底クラスです。サブクラスでは、クラス属性 `want` を 4 文字の OSA クラスコードに設定せねばなりません。このクラスのサブクラスのインスタンスは AppleScript オブジェクト指定子と同じになります。インスタンス化を行う際には、`which` にセレクタを渡さねばなりません。また、任意で親オブジェクトを `fr` に渡せます。

class `aetypes.NProperty` (*fr*)

OSA プロパティ用の抽象基底クラスです。サブクラスでは、クラス属性 `want` と `which` を設定して、どのプロパティを表しているかを指定せねばなりません。このクラスのサブクラスのインスタンスはオブジェクト指定子と同じになります。

class `aetypes.ObjectSpecifier` (*want, form, seld[, fr]*)

`ComponentItem` と `NProperty` の基底クラスで、汎用の OSA オブジェクト指定子を表します。パラメタの説明は Apple Open Scripting Architecture のドキュメントを参照してください。このクラスは抽象クラスではないので注意してください。

38.5 MiniAEFrame — オープンスクリプティングアーキテクチャサーバのサポート

MiniAEFrame モジュールは、アプリケーションにオープンスクリプティングアーキテクチャ (OSA) サーバ機能を持たせるためのフレームワークを提供します。つまり、AppleEvents の受信と処理を行わせます。*FrameWork* と連携させてもいいし、単独でも使えます。実例として、このモジュールは *PythonCGISlave* の中で使われています。

MiniAEFrame には以下のクラスが定義されています:

class *MiniAEFrame.AEServer*

AppleEvent の分岐を処理するクラス。作成するアプリケーションはこのクラスと、*MiniApplication* あるいは *FrameWork.Application* のサブクラスでなければなりません。サブクラス化したクラスでは `__init__()` メソッドで、継承した両方のクラスの `__init__()` メソッドを呼びださなければなりません。

class *MiniAEFrame.MinApplication*

FrameWork.Application とある程度互換なクラスですが、機能は少ないです。このクラスのイベントループはアップルメニュー、Cmd-(コマンドキーを押しながらピリオド.を押す)、AppleEvent をサポートします。他のイベントは Python インタープリタが Sioux (CodeWarrior のコンソールシステム) に渡されます。作成するアプリケーションで *AEServer* を使いたいが、独自のウィンドウなどを持たない場合に便利です。

38.5.1 AEServer オブジェクト

AEServer.installaehandler (*classe*, *type*, *callback*)

AppleEvent ハンドラをインストールします。 *classe* と *type* は 4 文字の OSA クラスとタイプの指定子で、ワイルドカード '****' も使えます。対応する AppleEvent を受けるとパラメータがデコードされ、与えたコールバックが呼び出されます。

AEServer.callback (*_object*, ***kwargs*)

与えたコールバックは、OSA ダイレクトオブジェクトを 1 番目のパラメータとして呼び出されます。他のパラメータは 4 文字の指定子を名前にしたキーワード引数として渡されます。他に 3 つのキーワード・パラメータが渡されます。つまり、`_class` と `_type` はクラスとタイプ指定子で、`_attributes` は AppleEvent 属性を持つ辞書です。

与えたメソッドの戻り値は *aetools.packer.event()* でパックされ、リプライとして送られます。

現在のクラス設計にはいくつか重大な問題があることに注意してください。引数に名前ではない 4 文字の指定子を持つ AppleEvent はまだ実装されていないし、イベントの送信側にエラーを返すこともできません。この問題は将来のリリースまで先送りされています。

他に、以下のサポートモジュールが事前に生成されています: *Finder*, *Terminal*, *Explorer*, *Netscape*, *CodeWarrior*, *SystemEvents*, *StdSuites*。

第 39 章

SGI IRIX 固有のサービス

この章で記述されているモジュールは、SGI の IRIX オペレーティングシステム (バージョン 4 と 5) 固有の機能へのインターフェイスを提供します。

39.1 al — SGI のオーディオ機能

バージョン 2.6 で非推奨: `al` モジュールは Python 3 で削除されました。

このモジュールを使うと、SGI Indy と Indigo ワークステーションのオーディオ装置にアクセスできます。詳しくは IRIX の man ページのセクション 3A を参照してください。ここに書かれた関数が何をするかを理解するには、man ページを読む必要があります！ IRIX のリリース 4.0.5 より前のものでは使えない関数もあります。お使いのプラットフォームで特定の関数が使えるかどうか、マニュアルで確認してください。

このモジュールで定義された関数とメソッドは全て、名前に `AL` の接頭辞を付けた C の関数と同義です。

C のヘッダーファイル `<audio.h>` のシンボル定数は標準モジュール `AL` に定義されています。下記を参照してください。

警告: オーディオライブラリの現在のバージョンは、不正な引数が渡されるとエラーステータスが返るのではなく、core を吐き出すことがあります。残念ながら、この現象が確実に起こる環境は述べられていないし、確認することは難しいので、Python インターフェースでこの種の問題に対して防御することはできません。(一つの例は過大なキューサイズを特定することです — 上限については記載されていません。)

このモジュールには、以下の関数が定義されています:

`al.openport (name, direction[, config])`

引数 `name` と `direction` は文字列です。省略可能な引数 `config` は、`newconfig()` で返されるコンフィギュレーションオブジェクトです。返り値は *audio port object* です；オーディオポートオブジェクトのメソッドは下に書かれています。

`al.newconfig()`

返り値は新しい *audio configuration object* です；オーディオコンフィギュレーションオブジェクトのメソッドは下に書かれています。

`al.queryparams(device)`

引数 *device* は整数です。返り値は `ALqueryparams()` で返されるデータを含む整数のリストです。

`al.getparams(device, list)`

引数 *device* は整数です。引数 *list* は `queryparams()` で返されるようなリストです;
`queryparams()` を適切に (!) 修正して使うことができます。

`al.setparams(device, list)`

引数 *device* は整数です。引数 *list* は `queryparams()` で返されるようなリストです。

39.1.1 コンフィギュレーションオブジェクト

`newconfig()` で返されるコンフィギュレーションオブジェクトには以下のメソッドがあります：

`audio configuration.getqueuesize()`

キューサイズを返します。

`audio configuration.setqueuesize(size)`

キューサイズを設定します。

`audio configuration.getwidth()`

サンプルサイズを返します。

`audio configuration.setwidth(width)`

サンプルサイズを設定します。

`audio configuration.getchannels()`

チャンネル数を返します。

`audio configuration.setchannels(nchannels)`

チャンネル数を設定します。

`audio configuration.getsampfmt()`

サンプルのフォーマットを返します。

`audio configuration.setsampfmt(sampfmt)`

サンプルのフォーマットを設定します。

`audio configuration.getfloatmax()`

浮動小数点数でサンプルデータの最大値を返します。

`audio configuration.setfloatmax(floatmax)`

浮動小数点数でサンプルデータの最大値を設定します。

39.1.2 ポートオブジェクト

`openport()` で返されるポートオブジェクトには以下のメソッドがあります：

`audio port.closeport()`

ポートを閉じます。

`audio port.getfd()`

ファイルデスクリプタを整数で返します。

`audio port.getfilled()`

バッファに存在するサンプルの数を返します。

`audio port.getfillable()`

バッファの空きに入れることのできるサンプルの数を返します。

`audio port.readsamps(nsamples)`

必要ならブロックして、キューから指定のサンプル数を読み込みます。生データを文字列として（例えば、サンプルサイズが 2 バイトならサンプル当たり 2 バイトが big-endian (high byte、low byte) で）返します。

`audio port.writesamps(samples)`

必要ならブロックして、キューにサンプルを書き込みます。サンプルは `readsamps()` で返される値のようにエンコードされていなければなりません。

`audio port.getfillpoint()`

'fill point' を返します。

`audio port.setfillpoint(fillpoint)`

'fill point' を設定します。

`audio port.getconfig()`

現在のポートのコンフィギュレーションを含んだコンフィギュレーションオブジェクトを返します。

`audio port.setconfig(config)`

コンフィギュレーションを引数に取り、そのコンフィギュレーションに設定します。

`audio port.getstatus(list)`

最後のエラーについてのステータスの情報を返します。

39.2 AL — al モジュールで使われる定数

バージョン 2.6 で非推奨: `AL` モジュールは Python 3 で削除されました。

このモジュールには、組み込みモジュール `al` (上記参照) を使用するのに必要とされるシンボリック定数が定義されています。定数の名前は C の include ファイル `<audioio.h>` で接頭辞 `AL_` を除いたものと同じです。定義されている名前の完全なリストについてはモジュールのソースを参照してください。お勤めの使い方は以下の通りです：

```
import al
from AL import *
```

39.3 cd — SGI システムの CD-ROM へのアクセス

バージョン 2.6 で非推奨: `cd` モジュールは Python 3 で削除されました。

このモジュールは Silicon Graphics CD ライブラリへのインターフェースを提供します。Silicon Graphics システムだけで利用可能です。

ライブラリは次のように使われます。CD-ROM デバイスを `open()` で開き、`createparser()` で CD からデータをパースするためのパーザを作ります。`open()` で返されるオブジェクトは CD からデータを読み込むのに使われますが、CD-ROM デバイスのステータス情報や、CD の情報、たとえば目次などを得るのにも使われます。CD から得たデータはパーザに渡され、パーザはフレームをパースし、あらかじめ加えられたコールバック関数を呼び出します。

オーディオ CD はトラック *tracks* あるいはプログラム *programs* (同じ意味で、どちらかの用語が使われます) に分けられます。トラックはさらにインデックス *indices* に分けられます。オーディオ CD は、CD 上の各トラックのスタート位置を示す目次 *table of contents* を持っています。インデックス 0 は普通、トラックの始まりの前のポーズです。目次から得られるトラックのスタート位置は通常、インデックス 1 のスタート位置です。

CD 上の位置は 2 通りの方法で得ることができます。それはフレームナンバーと、分、秒、フレームの 3 つの値からなるタプルの 2 つです。ほとんどの関数は後者を使います。位置は CD の開始位置とトラックの開始位置の両方に相対的になります。

`cd` モジュールは次の関数と定数を定義します:

`cd.createparser()`

不透明なパーザオブジェクトを作って返します。パーザオブジェクトのメソッドは下に記載されています。

`cd.msftoframe(minutes, seconds, frames)`

絶対的なタイムコードである (minutes, seconds, frames) の三つ組の表現を、相当する CD のフレームナンバーに変換します。

`cd.open([device[, mode]])`

CD-ROM デバイスを開きます。不透明なプレーヤーオブジェクトを返します; プレーヤーオブジェクトのメソッドは下に記載されています。デバイス *device* は SCSI デバイスファイルの名前で、例えば `'/dev/scsi/sc0d410'` あるいは `None` です。もし省略したり、`None` なら、ハードウェアが検索されて CD-ROM デバイスを割り当てます。*mode* は、省略しないなら `'r'` にすべきです。

このモジュールでは以下の変数を定義しています:

exception `cd.error`

様々なエラーについて発生する例外です。

`cd.DATASIZE`

オーディオデータの 1 フレームのサイズです。これは `audio` タイプのコールバックへ渡されるオーディオデータのサイズです。

`cd.BLOCKSIZE`

オーディオデータが読み取られていないフレーム 1 つのサイズです。

以下の変数は `getstatus()` で返されるステータス情報です:

`cd.READY`

オーディオ CD がロードされて、ドライブが操作可能であることを示します。

`cd.NODISC`

ドライブに CD がロードされていないことを示します。

`cd.CDROM`

ドライブに CD-ROM がロードされていることを示します。続いて `play` あるいは `read` の操作をすると、I/O エラーを返します。

`cd.ERROR`

ディスクや目次を読み込もうとしているときに起こるエラー。

`cd.PLAYING`

ドライブがオーディオ CD を CD プレーヤーモードでオーディオ端子から再生していることを示します。

`cd.PAUSED`

ドライブが CD プレーヤーモードで、再生を一時停止していることを示します。

`cd.STILL`

[`PAUSED`](#) と同じですが、古いモデル (non 3301) である Toshiba CD-ROM ドライブのもので、このドライブはもう SGI から出荷されていません。

`cd.audio`

`cd.pnum`

`cd.index`

`cd.ptime`

`cd.atime`

`cd.catalog`

`cd.ident`

`cd.control`

これらは整数の定数で、パーザのいろいろなタイプのコールバックを示しています。コールバックは CD パーザオブジェクトの `addcallback()` で設定できます (下記参照)。

39.3.1 Player オブジェクト

プレーヤーオブジェクト (`open()` で返されます) には以下のメソッドがあります:

CD `player.allowremoval()`

CD-ROM ドライブのイジェクトボタンのロックを解除して、ユーザが CD キャディを排出するのを許可します。

CD `player.bestreadsize()`

メソッド `readda()` のパラメータ `num_frames` として最適の値を返します。最適値は CD-ROM ドライブからの連続したデータフローが許可される値が定義されます。

CD `player.close()`

プレーヤーオブジェクトと関連付けられたリソースを解放します。 `close()` を呼び出したあとでは、そのオブジェクトに対するメソッドは使用できません。

CD `player.eject()`

CD-ROM ドライブからキャディを排出します。

CD `player.getstatus()`

CD-ROM ドライブの現在の状態に関する情報を返します。返される情報は以下の値からなるタプルです: `state`、`track`、`rtime`、`atime`、`ttime`、`first`、`last`、`scsi_audio`、`cur_block`。 `rtime` は現在のトラックの初めからの相対的な時間; `atime` はディスクの初めからの相対的な時間; `ttime` はディスクの全時間です。それぞれの値の詳細については、マニュアルページ `CDgetstatus(3dm)` を参照してください。 `state` の値は以下のうちのどれか一つです: `ERROR`、`NODISC`、`READY`、`PLAYING`、`PAUSED`、`STILL`、`CDROM`。

CD `player.gettrackinfo(track)`

特定のトラックについての情報を返します。返される情報は、トラックの開始時刻とトラックの時間の長さの二つの要素からなるタプルです。

CD `player.msftoblock(min, sec, frame)`

分、秒、フレームの 3 つからなる絶対的なタイムコードを、与えられた CD-ROM ドライブの相当する論理ブロック番号に変換します。時刻を比較するには `msftoblock()` よりも `msftoframe()` を使うべきです。論理ブロック番号は、CD-ROM ドライブによって必要とされるオフセット値が異なるため、フレームナンバーと異なります。

CD `player.play(start, play)`

CD-ROM ドライブのオーディオ CD の特定のトラックから再生を開始します。CD-ROM ドライブのヘッドフォン端子と (備えているなら) オーディオ端子から出力されます。ディスクの最後で再生は停止します。 `start` は再生を開始する CD のトラックナンバーです; `play` が 0 なら、CD は最初の一時的停止状態になります。その状態からメソッド `togglepause()` で再生を開始できます。

CD `player.playabs(minutes, seconds, frames, play)`

`play()` と似ていますが、開始位置をトラックナンバーの代わりに分、秒、フレームで与えます。

CD `player.playtrack(start, play)`

`play()` と似ていますが、トラックの終わりで再生を停止します。

CD `player.playtrackabs(track, minutes, seconds, frames, play)`

`play()` と似ていますが、指定した絶対的な時刻から再生を開始して、指定したトラックで終了します。

CD `player.preventremoval()`

CD-ROM ドライブのイジェクトボタンをロックして、ユーザが CD キャディを排出できないようにします。

CD `player.reada(num_frames)`

CD-ROM ドライブにマウントされたオーディオ CD から、指定したフレーム数を読み込みます。オーディオフレームのデータを示す文字列を返します。この文字列はそのままパーザオブジェクトのメソッド `parseframe()` へ渡すことができます。

CD `player.seek(minutes, seconds, frames)`

CD-ROM から次にデジタルオーディオデータを読み込む開始位置のポインタを設定します。ポインタは *minutes*、*seconds*、*frames* で指定した絶対的なタイムコードの位置に設定されます。返される値はポインタが設定された論理ブロック番号です。

CD `player.seekblock(block)`

CD-ROM から次にデジタルオーディオデータを読み込む開始位置のポインタを設定します。ポインタは指定した論理ブロック番号に設定されます。返される値はポインタが設定された論理ブロック番号です。

CD `player.seektrack(track)`

CD-ROM から次にデジタルオーディオデータを読み込む開始位置のポインタを設定します。ポインタは指定したトラックに設定されます。返される値はポインタが設定された論理ブロック番号です。

CD `player.stop()`

現在実行中の再生を停止します。

CD `player.togglepause()`

再生中なら CD を一時停止し、一時停止中なら再生します。

39.3.2 Parser オブジェクト

パーザオブジェクト (`createparser()` で返されます) には以下のメソッドがあります:

CD `parser.addcallback(type, func, arg)`

パーザにコールバックを加えます。デジタルオーディオストリームの 8 つの異なるデータタイプのためのコールバックをパーザは持っています。これらのタイプのための定数は `cd` モジュールのレベルで定義されています (上記参照)。コールバックは以下のように呼び出されます: `func(arg, type, data)`、ここで *arg* はユーザが与えた引数、*type* はコールバックの特定のタイプ、*data* はこの *type* のコールバックに渡されるデータです。データのタイプは以下のようにコールバックのタイプによって決まります:

型	値
<code>audio</code>	<code>al.writesamps()</code> へそのまま渡すことのできる文字列。
<code>pnum</code>	プログラム (トラック) ナンバーを示す整数。
<code>index</code>	インデックスナンバーを示す整数。
<code>ptime</code>	プログラムの時間を示す分、秒、フレームからなるタプル。
<code>atime</code>	絶対的な時刻を示す分、秒、フレームからなるタプル。
<code>catalog</code>	CD のカタログナンバーを示す 13 文字の文字列。
<code>ident</code>	録音の ISRC 識別番号を示す 12 文字の文字列。文字列は 2 文字の国別コード、3 文字の所有者コード、2 文字の年、5 文字のシリアルナンバーからなります。
<code>control</code>	CD のサブコードデータのコントロールビットを示す整数。

CD `parser.deleteparser()`

パーザを消去して、使用していたメモリを解放します。この呼び出しのあと、オブジェクトは使用できません。オブジェクトへの最後の参照が削除されると、自動的にこのメソッドが呼び出されます。

CD `parser.parseframe(frame)`

`readdata()` などから返されたデジタルオーディオ CD のデータの 1 つあるいはそれ以上のフレームをパースします。データ内にどのようなサブコードがあるかを決定します。その前のフレームからサブコードが変化していたら、`parseframe()` は対応するタイプのコールバックを起動して、フレーム内のサブコードデータをコールバックに渡します。C の関数とは違って、1 つ以上のデジタルオーディオデータのフレームをこのメソッドに渡すことができます。

CD `parser.removecallback(type)`

指定した *type* のコールバックを削除します。

CD `parser.resetparser()`

サブコードを追跡しているパーザのフィールドをリセットして、初期状態にします。ディスクを交換したあと、`resetparser()` を呼び出さなければなりません。

39.4 `fl` — グラフィカルユーザーインターフェースのための FORMS ライブラリ

バージョン 2.6 で非推奨: `fl` モジュールは Python 3 で削除されました。

このモジュールは、Mark Overmars による FORMS ライブラリへのインターフェースを提供します。FORMS ライブラリのソースは anonymous FTP `ftp.cs.ruu.nl` の SGI/FORMS ディレクトリから入手できます。最新のテストはバージョン 2.0b で行いました。

ほとんどの関数は接頭辞の `fl_` を取ると、対応する C の関数名になります。ライブラリで使われる定数は後述の `FL` モジュールで定義されています。

Python でこのオブジェクトを作る方法は C とは少し違っています: ライブラリに保持された '現在のフォーム' に新しい FORMS オブジェクトを加えるのではなく、フォームに FORMS オブジェクトを加えるには、フォームを示す Python オブジェクトのメソッドで全て行います。したがって、C の関数の `fl_addto_form()` と `fl_end_form()` に相当するものは Python にはありませんし、`fl_bgn_form()` に相当するものとしては `fl.make_form()` を呼び出します。

用語のちょっとした混乱に注意してください: FORMS ではフォームの中に置くことができるボタン、スライダーなどに *object* の用語を使います。Python では全ての値が 'オブジェクト' です。FORMS への Python のインターフェースによって、2 つの新しいタイプの Python オブジェクト: フォームオブジェクト (フォーム全体を示します) と FORMS オブジェクト (ボタン、スライダーなどの一つひとつを示します) を作ります。おそらく、混乱するほどのことではありません。

FORMS への Python インターフェースに 'フリーオブジェクト' はありませんし、Python でオブジェクトクラスを書いて加える簡単な方法もありません。しかし、GL イベントハンドルへの FORMS インターフェースが利用可能で、純粋な GL ウィンドウに FORMS を組み合わせることができます。

注意: `fl` をインポートすると、GL の関数 `foreground()` と FORMS のルーチン `fl_init()` を呼び出します。

39.4.1 fl モジュールに定義されている関数

`fl` モジュールには以下の関数が定義されています。これらの関数の働きに関する詳しい情報については、FORMS ドキュメントで対応する C の関数の説明を参照してください。

`fl.make_form(type, width, height)`

与えられたタイプ、幅、高さでフォームを作ります。これは *form* オブジェクトを返します。このオブジェクトは後述のメソッドを持ちます。

`fl.do_forms()`

標準の FORMS のメインループです。ユーザからの応答が必要な FORMS オブジェクトを示す Python オブジェクト、あるいは特別な値 `FL.EVENT` を返します。

`fl.check_forms()`

FORMS イベントを確認します。 `do_forms()` が返すもの、あるいはユーザからの応答をすぐに必要とするイベントがないなら `None` を返します。

`fl.set_event_call_back(function)`

イベントのコールバック関数を設定します。

`fl.set_graphics_mode(rgbmode, doublebuffering)`

グラフィックモードを設定します。

`fl.get_rgbmode()`

現在の RGB モードを返します。これは C のグローバル変数 `fl_rgbmode` の値です。

`fl.show_message(str1, str2, str3)`

3 行のメッセージと OK ボタンのあるダイアログボックスを表示します。

`fl.show_question(str1, str2, str3)`

3 行のメッセージと YES、NO のボタンのあるダイアログボックスを表示します。ユーザによって YES が押されたら 1、NO が押されたら 0 を返します。

`fl.show_choice(str1, str2, str3, but1[, but2[, but3]])`

3 行のメッセージと最大 3 つまでのボタンのあるダイアログボックスを表示します。ユーザによって押されたボタンの数値を返します (それぞれ 1、2、3)。

`fl.show_input(prompt, default)`

1 行のプロンプトメッセージと、ユーザが入力できるテキストフィールドを持つダイアログボックスを表示します。2 番目の引数はデフォルトで表示される入力文字列です。ユーザが入力した文字列が返されます。

`fl.show_file_selector(message, directory, pattern, default)`

ファイル選択ダイアログを表示します。ユーザによって選択されたファイルの絶対パス、あるいはユーザが Cancel ボタンを押した場合は `None` を返します。

```
fl.get_directory()
```

```
fl.get_pattern()
```

```
fl.get_filename()
```

これらの関数は最後にユーザが `show_file_selector()` で選択したディレクトリ、パターン、ファイル名 (パスの末尾のみ) を返します。

```
fl.qdevice(dev)
```

```
fl.unqdevice(dev)
```

```
fl.isqueued(dev)
```

```
fl.qtest()
```

```
fl.qread()
```

```
fl.qreset()
```

```
fl.qenter(dev, val)
```

```
fl.get_mouse()
```

```
fl.tie(button, valuator1, valuator2)
```

これらの関数は対応する GL 関数への FORMS のインターフェースです。 `fl.do_events()` を使っていて、自分で何か GL イベントを操作したいときにこれらを使います。FORMS が扱うことのできない GL イベントが検出されたら `fl.do_forms()` が特別の値 `FL.EVENT` を返すので、`fl.qread()` を呼び出して、キューからイベントを読み込むべきです。対応する GL の関数は使わないでください!

```
fl.color()
```

```
fl.mapcolor()
```

```
fl.getmcolor()
```

FORMS ドキュメントにある `fl_color()`、`fl_mapcolor()`、`fl_getmcolor()` の記述を参照してください。

39.4.2 Form オブジェクト

フォームオブジェクト (上で述べた `make_form()` で返されます) には下記のメソッドがあります。各メソッドは名前の接頭辞に `fl_` を付けた C の関数に対応します; また、最初の引数はフォームのポインタです; 説明は FORMS の公式文書を参照してください。

全ての `add_*()` メソッドは、FORMS オブジェクトを示す Python オブジェクトを返します。FORMS オブジェクトのメソッドを以下に記載します。ほとんどの FORMS オブジェクトは、そのオブジェクトの種類ごとに特有のメソッドもいくつか持っています。

```
form.show_form(placement, bordertype, name)
```

フォームを表示します。

```
form.hide_form()
```

フォームを隠します。

```
form.redraw_form()
```

フォームを再描画します。

```
form.set_form_position(x, y)
```

フォームの位置を設定します。

`form.freeze_form()`

フォームを固定します。

`form.unfreeze_form()`

固定したフォームの固定を解除します。

`form.activate_form()`

フォームをアクティベートします。

`form.deactivate_form()`

フォームをディアクティベートします。

`form.bgn_group()`

新しいオブジェクトのグループを作ります；グループオブジェクトを返します。

`form.end_group()`

現在のオブジェクトのグループを終了します。

`form.find_first()`

フォームの中の最初のオブジェクトを見つけます。

`form.find_last()`

フォームの中の最後のオブジェクトを見つけます。

`form.add_box(type, x, y, w, h, name)`

フォームにボックスオブジェクトを加えます。特別な追加のメソッドはありません。

`form.add_text(type, x, y, w, h, name)`

フォームにテキストオブジェクトを加えます。特別な追加のメソッドはありません。

`form.add_clock(type, x, y, w, h, name)`

フォームにクロックオブジェクトを加えます。 — メソッド: `get_clock()`。

`form.add_button(type, x, y, w, h, name)`

フォームにボタンオブジェクトを加えます。 — メソッド: `get_button()`、`set_button()`。

`form.add_lightbutton(type, x, y, w, h, name)`

フォームにライトボタンオブジェクトを加えます。 — メソッド: `get_button()`、`set_button()`。
。

`form.add_roundbutton(type, x, y, w, h, name)`

フォームにラウンドボタンオブジェクトを加えます。 — メソッド: `get_button()`、`set_button()`。
。

`form.add_slider(type, x, y, w, h, name)`

フォームにスライダーオブジェクトを加えます。 — メソッド: `set_slider_value()`、`get_slider_value()`、`set_slider_bounds()`、`get_slider_bounds()`、`set_slider_return()`、`set_slider_size()`、`set_slider_precision()`、`set_slider_step()`。

`form.add_valslider` (*type, x, y, w, h, name*)

フォームにバリュースライダーオブジェクトを加えます。 — メソッド: `set_slider_value()`、`get_slider_value()`、`set_slider_bounds()`、`get_slider_bounds()`、`set_slider_return()`、`set_slider_size()`、`set_slider_precision()`、`set_slider_step()`。

`form.add_dial` (*type, x, y, w, h, name*)

フォームにダイヤルオブジェクトを加えます。 — メソッド: `set_dial_value()`、`get_dial_value()`、`set_dial_bounds()`、`get_dial_bounds()`。

`form.add_positioner` (*type, x, y, w, h, name*)

フォームに 2 次元ポジショナーオブジェクトを加えます。 — メソッド: `set_positioner_xvalue()`、`set_positioner_yvalue()`、`set_positioner_xbounds()`、`set_positioner_ybounds()`、`get_positioner_xvalue()`、`get_positioner_yvalue()`、`get_positioner_xbounds()`、`get_positioner_ybounds()`。

`form.add_counter` (*type, x, y, w, h, name*)

フォームにカウンタオブジェクトを加えます。 — メソッド: `set_counter_value()`、`get_counter_value()`、`set_counter_bounds()`、`set_counter_step()`、`set_counter_precision()`、`set_counter_return()`。

`form.add_input` (*type, x, y, w, h, name*)

フォームにインプットオブジェクトを加えます。 — メソッド: `set_input()`、`get_input()`、`set_input_color()`、`set_input_return()`。

`form.add_menu` (*type, x, y, w, h, name*)

フォームにメニューオブジェクトを加えます。 — メソッド: `set_menu()`、`get_menu()`、`addto_menu()`。

`form.add_choice` (*type, x, y, w, h, name*)

フォームにチョイスオブジェクトを加えます。 — メソッド: `set_choice()`、`get_choice()`、`clear_choice()`、`addto_choice()`、`replace_choice()`、`delete_choice()`、`get_choice_text()`、`set_choice_fontsize()`、`set_choice_fontstyle()`。

`form.add_browser` (*type, x, y, w, h, name*)

フォームにブラウザオブジェクトを加えます。 — メソッド: `set_browser_topline()`、`clear_browser()`、`add_browser_line()`、`addto_browser()`、`insert_browser_line()`、`delete_browser_line()`、`replace_browser_line()`、`get_browser_line()`、`load_browser()`、`get_browser_maxline()`、`select_browser_line()`、`deselect_browser_line()`、`deselect_browser()`、`isselected_browser_line()`、`get_browser()`、`set_browser_fontsize()`、`set_browser_fontstyle()`、`set_browser_specialkey()`。

`form.add_timer` (*type, x, y, w, h, name*)

フォームにタイマーオブジェクトを加えます。 — メソッド: `set_timer()`、`get_timer()`。

フォームオブジェクトには以下のデータ属性があります; FORMS ドキュメントを参照してください:

名前	C の型	意味
window	int (read-only)	GL ウィンドウの id
w	float	フォームの幅
h	float	フォームの高さ
x	float	フォーム左肩の x 座標
y	float	フォーム左肩の y 座標
deactivated	int	フォームがディアクティベートされているなら非ゼロ
visible	int	フォームが可視なら非ゼロ
frozen	int	フォームが固定されているなら非ゼロ
doublebuf	int	ダブルバッファリングがオンなら非ゼロ

39.4.3 FORMS オブジェクト

FORMS オブジェクトの種類ごとに特有のメソッドの他に、全ての FORMS オブジェクトは以下のメソッドも持っています:

FORMS `object.set_callback(function, argument)`

オブジェクトのコールバック関数と引数を設定します。オブジェクトがユーザからの応答を必要とするときには、コールバック関数は 2 つの引数、オブジェクトとコールバックの引数とともに呼び出されます。(コールバック関数のない FORMS オブジェクトは、ユーザからの応答を必要とするときには `fl.do_forms()` あるいは `fl.check_forms()` によって返されます。) 引数なしにこのメソッドを呼び出すと、コールバック関数を削除します。

FORMS `object.delete_object()`

オブジェクトを削除します。

FORMS `object.show_object()`

オブジェクトを表示します。

FORMS `object.hide_object()`

オブジェクトを隠します。

FORMS `object.redraw_object()`

オブジェクトを再描画します。

FORMS `object.freeze_object()`

オブジェクトを固定します。

FORMS `object.unfreeze_object()`

固定したオブジェクトの固定を解除します。

FORMS オブジェクトには以下のデータ属性があります; FORMS ドキュメントを参照してください。

名前	C の型	意味
objclass	int (read-only)	オブジェクトクラス
<i>type</i>	int (read-only)	オブジェクトタイプ
boxtype	int	ボックスタイプ
x	float	左肩の x 座標
y	float	左肩の y 座標
w	float	幅
h	float	高さ
col1	int	第 1 の色
col2	int	第 2 の色
align	int	配置
lcol	int	ラベルの色
lsize	float	ラベルのフォントサイズ
label	string	ラベルの文字列
lstyle	int	ラベルのスタイル
pushed	int (read-only)	(FORMS ドキュメント参照)
focus	int (read-only)	(FORMS ドキュメント参照)
belowmouse	int (read-only)	(FORMS ドキュメント参照)
frozen	int (read-only)	(FORMS ドキュメント参照)
active	int (read-only)	(FORMS ドキュメント参照)
<i>input</i>	int (read-only)	(FORMS ドキュメント参照)
visible	int (read-only)	(FORMS ドキュメント参照)
radio	int (read-only)	(FORMS ドキュメント参照)
automatic	int (read-only)	(FORMS ドキュメント参照)

39.5 FL — fl モジュールで使用される定数

バージョン 2.6 で非推奨: *FL* モジュールは Python 3 で削除されました。

このモジュールには、組み込みモジュール *fl* を使うのに必要なシンボル定数が定義されています (上記参照) ; これらは名前の接頭辞 *FL_* が省かれていることを除いて、C のヘッダファイル `<forms.h>` に定義されているものと同じです。定義されている名称の完全なリストについては、モジュールのソースをご覧ください。お勧めする使い方は以下の通りです :

```
import fl
from FL import *
```

39.6 flp — 保存された FORMS デザインをロードする関数

バージョン 2.6 で非推奨: *flp* モジュールは Python 3 で削除されました。

このモジュールには、FORMS ライブラリ (上記の *fl* モジュールを参照してください) とともに配布される '

フォームデザイナー' (**fdesign**) プログラムで作られたフォームの定義を読み込む関数が定義されています。

今のところは、詳しくは Python ライブラリソースのディレクトリの中の `flp.doc` を参照してください。

XXX A complete description should be inserted here!

39.7 fm — *Font Manager* インターフェース

バージョン 2.6 で非推奨: `fm` モジュールは Python 3 で削除されました。

このモジュールは IRIS *Font Manager* ライブラリへのアクセスを提供します。Silicon Graphics マシン上だけで利用可能です。次も参照してください: *4Sight User's Guide*, section 1, chapter 5: "Using the IRIS Font Manager".

このモジュールは、まだ IRIS Font Manager への完全なインタフェースではありません。サポートされていない機能は次のものです: matrix operations; cache operations; character operations (代わりに string operations を使ってください); font info のうちのいくつか; individual glyph metrics; printer matching.

以下の操作をサポートしています:

`fm.init()`

関数を初期化します。 `fminit()` を呼び出します。この関数は `fm` モジュールを最初にインポートすると自動的に呼び出されるので、普通、呼び出す必要はありません。

`fm.findfont(fontname)`

フォントハンドルオブジェクトを返します。 `fmfindfont(fontname)` を呼び出します。

`fm.enumerate()`

利用可能なフォント名のリストを返します。この関数は `fmenumerate()` へのインタフェースです。

`fm.prstr(string)`

現在のフォントを使って文字列をレンダリングします (下のフォントハンドルメソッド `setfont()` を参照)。 `fmprstr(string)` を呼び出します。

`fm.setpath(string)`

フォントの検索パスを設定します。 `fmsetpath(string)` を呼び出します。(XXX 機能しない!?!)

`fm.fontpath()`

現在のフォント検索パスを返します。

フォントハンドルオブジェクトは以下の操作をサポートします:

`font handle.scalefont(factor)`

このフォントを拡大 / 縮小したハンドルを返します。 `fmscalefont(fh, factor)` を呼び出します。

`font handle.setfont()`

このフォントを現在のフォントに設定します。注意: フォントハンドルオブジェクトが削除されると、設定は告知なしに元に戻ります。 `fmsetfont(fh)` を呼び出します。

font handle.getFontname()

このフォントの名前を返します。 `fmgetfontname(fh)` を呼び出します。

font handle.getcomment()

このフォントに関連付けられたコメント文字列を返します。コメント文字列が何もない場合は例外を返します。 `fmgetcomment(fh)` を呼び出します。

font handle.getFontinfo()

このフォントに関連したデータを含むタプルを返します。これは `fmgetfontinfo()` へのインタフェースです。以下の数値を含むタプルを返します: (`printermatched`, `fixedwidth`, `xorig`, `yorig`, `xsize`, `ysize`, `height`, `nglyphs`)。

font handle.getstrwidth(string)

このフォントで *string* を描いたときの幅をピクセル数で返します。 `fmgetstrwidth(fh, string)` を呼び出します。

39.8 gl — Graphics Library インターフェース

バージョン 2.6 で非推奨: *gl* モジュールは Python 3 で削除されました。

このモジュールは Silicon Graphics の *Graphics Library* へのアクセスを提供します。Silicon Graphics マシン上だけで利用可能です。

警告: GL ライブラリの不適切な呼び出しによっては、Python インタープリタがコアダンプすることがあります。特に、GL のほとんどの関数では最初のウィンドウを開く前に呼び出すのは安全ではありません。

このモジュールはとても大きいので、ここに全てを記述することはできませんが、以下の説明で出発点としては十分でしょう。C の関数のパラメータは、以下のような決まりに従って Python に翻訳されます:

- 全ての (short、long、unsigned) int は Python の整数に相当します。
- 全ての浮動小数点数と倍精度浮動小数点数は Python の浮動小数点数に相当します。たいていの場合、Python の整数も使えます。
- 全ての配列は Python の一次元のリストに相当します。たいていの場合、タプルも使えます。
- 全ての文字列と文字の引数は、Python の文字列に相当します。例えば、`winopen('Hi There!')` と `rotate(900, 'z')`。
- 配列である引数の長さを特定するためだけに使われる全ての (short、long、unsigned) 整数値の引数あるいは返り値は省略されます。例えば、C の呼び出しで、

```
lmdef(deftype, index, np, props)
```

これは Python では、こうなります。

```
lmdef(deftype, index, props)
```

- 出力のための引数は、引数のリストから省略されています；代わりにこれらは関数の返り値として渡されます。もし 1 つ以上の値が返されるのなら、返り値はタプルです。もし C の関数が通常の返り値 (先のルールによって省略されません) と、出力のための引数の両方を取るなら、返り値はタプルの最初に来ます。例：C の呼び出しで、

```
getmcolor(i, &red, &green, &blue)
```

これは Python では、こうなります。

```
red, green, blue = getmcolor(i)
```

以下の関数は一般的でないか、引数に特別な決まりを持っています:

`gl.varray (argument)`

`v3d()` の呼び出しに相当しますが、それよりも速いです。 *argument* は座標のリスト (あるいはタプル) です。各座標は (x, y, z) あるいは (x, y) の座標のタプルでなければなりません。座標は 2 次元あるいは 3 次元が可能ですが、全て同次元でなければなりません。ですが、浮動小数点数と整数を混合して使えます。座標は (マニュアルページにあるように) 必要であれば $z = 0.0$ と仮定して、常に 3 次元の精密な座標に変換され、各座標について `v3d()` が呼び出されます。

`gl.nvarray ()`

`n3f` と `v3f` の呼び出しに相当しますが、それらよりも速いです。引数は法線と座標とのペアからなるシーケンス (リストあるいはタプル) です。各ペアは座標と、その座標からの法線とのタプルです。各座標と各法線は (x, y, z) からなるタプルでなければなりません。3 つの座標が渡されなければなりません。浮動小数点数と整数を混合して使えます。各ペアについて、法線に対して `n3f()` が呼び出され、座標に対して `v3f()` が呼び出されます。

`gl.vnarray ()`

`nvarray()` と似ていますが、各ペアは始めに座標を、2 番目に法線を持っています。

`gl.nurbssurface (s_k, t_k, ctl, s_ord, t_ord, type)`

`nurbs` (非均一有理 B スプライン) 曲面を定義します。 `ctl[][]` の次元は以下のように計算されます：
`[len(s_k) - s_ord]`、`[len(t_k) - t_ord]`。

`gl.nurbscurve (knots, ctlpoints, order, type)`

`nurbs` (非均一有理 B スプライン) 曲線を定義します。 `ctlpoints` の長さは、`len(knots) - order` です。

`gl.pwlcurve (points, type)`

区分線形曲線 (piecewise-linear curve) を定義します。 *points* は座標のリストです。 *type* は `N_ST` でなければなりません。

`gl.pick (n)`

`gl.select (n)`

これらの関数はただ一つの引数を取り、`pick/select` に使うバッファのサイズを設定します。

```
gl.endpick()
gl.endselect()
```

これらの関数は引数を取りません。 pick/select に使われているバッファの大きさを示す整数のリストを返します。バッファがあふれているを検出するメソッドはありません。

小さいですが完全な Python の GL プログラムの例をここに挙げます：

```
import gl, GL, time

def main():
    gl.foreground()
    gl.prefposition(500, 900, 500, 900)
    w = gl.winopen('CrissCross')
    gl.ortho2(0.0, 400.0, 0.0, 400.0)
    gl.color(GL.WHITE)
    gl.clear()
    gl.color(GL.RED)
    gl.bgnline()
    gl.v2f(0.0, 0.0)
    gl.v2f(400.0, 400.0)
    gl.endline()
    gl.bgnline()
    gl.v2f(400.0, 0.0)
    gl.v2f(0.0, 400.0)
    gl.endline()
    time.sleep(5)

main()
```

参考:

PyOpenGL: The Python OpenGL Binding OpenGL へのインタフェースが利用できます；詳しくは **Py-OpenGL** プロジェクト <http://pyopengl.sourceforge.net/> から情報を入手できます。これは、SGI のハードウェアが 1996 年頃より前である必要がないので、OpenGL の方が良い選択かもしれません。

39.9 DEVICE — gl モジュールで使われる定数

バージョン 2.6 で非推奨: *DEVICE* モジュールは Python 3 で削除されました。

このモジュールには、Silicon Graphics の *Graphics Library* で使われる定数が定義されています。これらは C のプログラマーがヘッダーファイル `<gl/device.h>` の中から使っているものです。詳しくはモジュールのソースファイルをご覧ください。

39.10 GL — gl モジュールで使われる定数

バージョン 2.6 で非推奨: *GL* モジュールは Python 3 で削除されました。

このモジュールには Silicon Graphics の *Graphics Library* で使われる C のヘッダーファイル `<gl/gl.h>` の

定数が定義されています。詳しくはモジュールのソースファイルをご覧ください。

39.11 `imgfile` — SGI `imglib` ファイルのサポート

バージョン 2.6 で非推奨: `imgfile` モジュールは Python 3 で削除されました。

`imgfile` モジュールは、Python プログラムが SGI `imglib` 画像ファイル (`.rgb` ファイルとしても知られています) にアクセスできるようにします。このモジュールは完全なものにはほど遠いですが、その機能はある状況で十分役に立つものなので、ライブラリで提供されています。現在、カラーマップ形式のファイルはサポートされていません。

このモジュールでは以下の変数と関数を定義しています:

exception `imgfile.error`

この例外は、サポートされていないファイル形式の場合のような全てのエラーで送出されます。

`imgfile.getsizes (file)`

この関数はタプル (`x`, `y`, `z`) を返します。`x` および `y` は画像のサイズをピクセルで表したもので、`z` はピクセルあたりのバイト長です。3 バイトの RGB ピクセルと 1 バイトのグレイスケールピクセルのみが現在サポートされています。

`imgfile.read (file)`

この関数は指定されたファイル上の画像を読み出して復号化し、Python 文字列として返します。この文字列は 1 バイトのグレイスケールピクセルか、4 バイトの RGBA ピクセルによるものです。左下のピクセルが文字列中の最初のピクセルになります。これは `gl.rectwrite()` に渡すのに適した形式です。

`imgfile.readscaled (file, x, y, filter[, blur])`

この関数は `read` と同じですが、`x` および `y` のサイズにスケールされた画像を返します。`filter` および `blur` パラメタが省略された場合、単にピクセルデータを捨てたり複製したりすることによってスケール操作が行われるので、処理結果は、特に計算機上で合成した画像の場合にはおよそ完璧とはいえないものになります。

そうする代わりに、スケール操作後に画像を平滑化するために用いるフィルタを指定することができます。サポートされているフィルタの形式は `'impulse'`、`'box'`、`'triangle'`、`'quadratic'`、および `'gaussian'` です。フィルタを指定する場合、`blur` はオプションのパラメタで、フィルタの不明瞭化度を指定します。デフォルトの値は 1.0 です。

`readscaled()` は正しいアスペクト比をまったく維持しようとしないので、それはユーザの責任になります。

`imgfile.ttob (flag)`

この関数は画像のスキャンラインの読み書きを下から上に向かって行う (フラグがゼロの場合で、SGI GL 互換です) か、上から下に向かって行う (フラグが 1 の場合で、X 互換です) かを決定する大域的なフラグを設定します。デフォルトの値はゼロです。

`imgfile.write (file, data, x, y, z)`

この関数は `data` 中の RGB またはグレイスケールのデータを画像ファイル `file` に書き込みます。`x` およ

び `y` には画像のサイズを与え、`z` は 1 バイトグレースケール画像の場合には 1 で、RGB 画像の場合には 3 (4 バイトの値として記憶され、下位 3 バイトが使われます) です。これらは `gl.lrectread()` が返すデータの形式です。

39.12 jpeg — JPEG ファイルの読み書きを行う

バージョン 2.6 で非推奨: `jpeg` モジュールは Python 3 で削除されました。

この `jpeg` モジュールは Independent JPEG Group (IJG) によって書かれた JPEG 圧縮及び展開アルゴリズムを提供します。JPEG 形式は写真等の画像圧縮で標準的に利用され、ISO 10918 で定義されています。JPEG、あるいは Independent JPEG Group ソフトウェアの詳細は、標準 JPEG、もしくは提供されるソフトウェアのドキュメントを参照してください。

JPEG ファイルを扱うポータブルなインタフェースは Fredrik Lundh による Python Imaging Library (PIL) があります。PIL の情報は <http://www.pythonware.com/products/pil/> で見つけることができます。

モジュール `jpeg` では、一つの例外といくつかの関数を定義しています。

exception `jpeg.error`

関数 `compress()` または `decompress()` のエラーで上げられる例外です。

`jpeg.compress(data, w, h, b)`

データを、ピクセルマップが幅 `w`、高さ `h`、1 ピクセルあたりのバイト数 `b` として扱います。データは SGI GL 順になっていて、最初のピクセルは左下端になります。また、これは `gl.lrectread()` が返す値をすぐに `compress()` にかけるためです。現在は、1 バイトもしくは 4 バイトのピクセルを取り扱うことができます。前者はグレースケール、後者は RGB カラーを扱います。`compress()` は、圧縮された JFIF 形式のイメージが含まれた文字列を返します。

`jpeg.decompress(data)`

データは圧縮された JFIF 形式のイメージが含まれた文字列で、この関数はタプル (`data`, `width`, `height`, `bytesperpixel`) を返します。このデータも圧縮でのそれ同様に `gl.lrectwrite()` に渡すのに適したものです。

`jpeg.setoption(name, value)`

`compress()` と `decompress()` を呼ぶための様々なオプションをセットします。次のオプションが利用できます:

オプション	効果
'forcegray'	入力が RGB でも強制的にグレースケールを出力します。
'quality'	圧縮後イメージの品質を 0 から 100 の間の値で指定します (デフォルトは 75 です)。これは圧縮にのみ影響します。
'optimize'	ハフマンテーブルを最適化します。時間がかかりますが、高圧縮になります。これは圧縮にのみ影響します。
'smooth'	圧縮されていないイメージ上でインターブロックスムーシングを行います。低品質イメージに役立ちます。これは展開にのみ影響します。

参考:

JPEG Still Image Data Compression Standard Pennebaker と Mitchell による、JPEG 画像フォーマットの正統的な参考文献。

[Information Technology - Digital Compression and Coding of Continuous-tone Still Images - Requirements and Guidelines](#)

JPEG の ISO 標準は ITU T.81 としても発行されています。これは PDF 形式でオンラインから取得できます。

第 40 章

SunOS 固有のサービス

この章では、SunOS オペレーティングシステムバージョン 5 (Solaris バージョン 2) に固有の機能を解説します。

40.1 sunaudiodev — Sun オーディオハードウェアへのアクセス

バージョン 2.6 で非推奨: `sunaudiodev` モジュールは Python 3 で削除されました。

このモジュールを使うと、Sun のオーディオインターフェースにアクセスできます。Sun オーディオハードウェアは、1 秒あたり 8k のサンプリングレート、u-LAW フォーマットでオーディオデータを録音、再生できます。完全な説明文書はマニュアルページ `audio(7I)` にあります。

モジュール `SUNAUDIODEV` には、このモジュールで使われる定数が定義されています。

このモジュールには、以下の変数と関数が定義されています:

exception `sunaudiodev.error`

この例外は、全てのエラーについて発生します。引数は誤りを説明する文字列です。

`sunaudiodev.open(mode)`

この関数はオーディオデバイスを開き、Sun オーディオデバイスのオブジェクトを返します。こうすることで、オブジェクトが I/O に使用できるようになります。パラメータ `mode` は次のうちのいずれかで、録音のみには 'r'、再生のみには 'w'、録音と再生両方には 'rw'、コントロールデバイスへのアクセスには 'control' です。レコーダーやプレーヤーには同時に 1 つのプロセスしかアクセスが許されていないので、必要な動作についてだけデバイスをオープンするのがいい考えです。詳しくは `audio(7I)` を参照してください。

マニュアルページにあるように、このモジュールは環境変数 `AUDIODEV` の中のベースオーディオデバイスファイルネームを初めに参照します。見つからない場合は `/dev/audio` を参照します。コントロールデバイスについては、ベースオーディオデバイスに "ctl" を加えて扱われます。

40.1.1 オーディオデバイスオブジェクト

オーディオデバイスオブジェクトは `open()` で返され、このオブジェクトには以下のメソッドが定義されています (control オブジェクトは除きます。これには `getinfo()`、`setinfo()`、`fileno()`、`drain()` だけが定義されています) :

`audio device.close()`

このメソッドはデバイスを明示的に閉じます。オブジェクトを削除しても、それを参照しているものがあって、すぐに閉じてくれない場合に便利です。閉じられたデバイスを使うことはできません。

`audio device.fileno()`

デバイスに関連づけられたファイルディスクリプタを返します。これは、後述の `SIGPOLL` の通知を組み立てるのに使われます。

`audio device.drain()`

このメソッドは全ての出力中のプロセスが終了するまで待って、それから制御が戻ります。このメソッドの呼び出しはそう必要ではありません：オブジェクトを削除すると自動的にオーディオデバイスを閉じて、暗黙のうちに吐き出します。

`audio device.flush()`

このメソッドは全ての出力中のものを捨て去ります。ユーザの停止命令に対する反応の遅れ (1 秒までの音声のバッファリングによって起こります) を避けるのに使われます。

`audio device.getinfo()`

このメソッドは入出力のボリューム値などの情報を引き出して、オーディオステータスのオブジェクト形式で返します。このオブジェクトには何もメソッドはありませんが、現在のデバイスの状態を示す多くの属性が含まれます。属性の名称と意味は `<sun/audioio.h>` と `audio(7I)` に記載があります。メンバー名は相当する C のものとは少し違っています：ステータスオブジェクトは 1 つの構造体です。その中の構造体である `play` のメンバーには名前の初めに `o_` がついていて、`record` には `i_` がついています。そのため、C のメンバーである `play.sample_rate` は `o.sample_rate` として、`record.gain` は `i_gain` として参照され、`monitor_gain` はそのまま `monitor_gain` で参照されます。

`audio device.ibufcount()`

このメソッドは録音側でバッファリングされるサンプル数を返します。つまり、プログラムは同じ大きさのサンプルに対する `read()` の呼び出しをブロックしません。

`audio device.obufcount()`

このメソッドは再生側でバッファリングされるサンプル数を返します。残念ながら、この数値はブロックなしに書き込めるサンプル数を調べるのには使えません。というのは、カーネルの出力キューの長さは可変だからです。

`audio device.read(size)`

このメソッドはオーディオ入力から `size` のサイズのサンプルを読み込んで、Python の文字列として返します。この関数は必要なデータが得られるまで他の操作をブロックします。

`audio device.setinfo(status)`

This method sets the audio device status parameters. The *status* parameter is a device status object as

returned by `getinfo()` and possibly modified by the program.

`audio device.write(samples)`

パラメータとしてオーディオサンプルを Python 文字列を受け取り、再生します。もし十分なバッファの空きがあればすぐに制御が戻り、そうでないならブロックされます。

オーディオデバイスは `SIGPOLL` を介して様々なイベントの非同期通知に対応しています。Python でこれをどのようにしたらできるか、例を挙げます：

```
def handle_sigpoll(signum, frame):
    print 'I got a SIGPOLL update'

import fcntl, signal, STROPTS

signal.signal(signal.SIGPOLL, handle_sigpoll)
fcntl.ioctl(audio_obj.fileno(), STROPTS.I_SETSIG, STROPTS.S_MSG)
```

40.2 SUNAUDIODEV — `sunaudiodev` で使われる定数

バージョン 2.6 で非推奨: `SUNAUDIODEV` モジュールは Python 3 で削除されました。

これは `sunaudiodev` に付随するモジュールで、`MIN_GAIN`、`MAX_GAIN`、`SPEAKER` などの便利なシンボル定数を定義しています。定数の名前は C の include ファイル `<sun/audioio.h>` のものと同じで、最初の文字列 `AUDIO_` を除いたものです。

第 41 章

ドキュメント化されていないモジュール

現在ドキュメント化されていないが、ドキュメント化すべきモジュールを以下にざっと列挙します。どうぞこれらのドキュメントを寄稿してください！（電子メールで docs@python.org に送ってください。）

この章のアイデアと元の文章内容は Fredrik Lundh のポストによるものです；この章の特定の内容は実際には改訂されてきています。

41.1 雑多な有用ユーティリティ

以下のいくつかは非常に古かったり、あまり頑健でなかったりします；”ふーむ” 印付きです。

`ihooks` — `import` フックのサポートです (*rexec* のためのものです；撤廃されるかもしれません)。Python 3.x で削除されました。

41.2 プラットフォーム固有のモジュール

これらのモジュールは `os.path` モジュールを実装するために用いられていますが、ここで触れる内容を超えてドキュメントされていません。これらはもう少しドキュメント化する必要があります。

`ntpath` — Win32、Win64、WinCE、および OS/2 プラットフォームにおける `os.path` 実装です。

`posixpath` — POSIX における `os.path` 実装です。

`bsddb185` — まだ BerkeleyDB 1.85 を使用しているシステムで後方互換性を保つためのモジュール。通常、特定の BSD Unix ベースのシステムでのみ利用可能。直接使用しないで下さい。

41.3 マルチメディア関連

`audiodev` — 音声データを再生するためのプラットフォーム非依存の API です。Python 3.x で削除されました。

`linuxaudiodev` — Linux 音声デバイスで音声データを再生します。Python 2.3 では `ossaudiodev` モジュールと置き換えられました。Python 3.x で削除されました。

`sunaudio` — Sun 音声データヘッダを解釈します (撤廃されるか、ツール/デモになるかもしれません)。
Python 3.x で削除されました。

`toaiiff` — ”任意の” 音声ファイルを AIFF ファイルに変換します; おそらくツールかデモになるはずです。
外部プログラム `sox` が必要です。Python 3.x で削除されました。

41.4 文書化されていない Mac OS モジュール

41.4.1 `applesingle` — `AppleSingle` デコーダー

バージョン 2.6 で非推奨.

41.4.2 `buildtools` — `BuildApplet` とその仲間のヘルパーモジュール

バージョン 2.4 で非推奨.

41.4.3 `cfmfile` — コードフラグメントリソースを扱うモジュール

`cfmfile` は、コードフラグメントと関連する ”cfrg” リソースを処理するモジュールです。このモジュールでコードフラグメントを分解やマージできて、全てのプラグインモジュールをまとめて、一つの実行可能ファイルにするため、`BuildApplication` によって利用されます。

バージョン 2.4 で非推奨.

41.4.4 `icopen` — `open()` を `Internet Config` のために置き換える

`icopen` をインポートすると、組み込み `open()` を新しいファイル用にファイルタイプおよびクリエーターを設定するために `Internet Config` を使用するバージョンに置き換えます。

バージョン 2.6 で非推奨.

41.4.5 `macerrors` — `MacOS` のエラー

`macerrors` は、`MacOS` エラーコードを意味する定数定義を含みます。

バージョン 2.6 で非推奨.

41.4.6 `macresource` — スクリプトのリソースを見つける

`macresource` はスクリプトが `MacPython` 上で `MacPython` アプレットとして起動されていたり、`OSX Python` 上で起動されている場合に、特別な処理をせずにダイアログやメニューなどのようなリソースを見つけるためのヘルパースクリプトです。

バージョン 2.6 で非推奨.

41.4.7 `Nav` — `NavServices` の呼び出し

Navigation Services の低レベルインターフェース。

バージョン 2.6 で非推奨.

41.4.8 `PixMapWrapper` — `PixMap` オブジェクトのラッパー

`PixMapWrapper` は `PixMap` オブジェクトを Python オブジェクトでラップしたもので、各フィールドに対し名前アクセスできるようになります。PIL 画像との相互の変換をするメソッドも用意されています。

バージョン 2.6 で非推奨.

41.4.9 `videoreader` — `QuickTime` ムービーの読み込み

`videoreader` は QuickTime ムービーを読み込み、デコードし、プログラムへ渡せます。このモジュールはさらにオーディオトラックをサポートしています。

バージョン 2.6 で非推奨.

41.4.10 `W` — `FrameWork` 上に作られたウィジェット

`W` ウィジェットは、`IDE` で頻繁に使われています。

バージョン 2.6 で非推奨.

41.5 撤廃されたもの

これらのモジュールは通常 `import` して利用できません; 利用できるようにするには作業を行わなければなりません。

これらの拡張モジュールのうち C で書かれたものは、標準の設定ではビルドされません。Unix でこれらのモジュールを有効にするには、ビルドツリー内の `Modules/Setup` の適切な行のコメントアウトを外して、モジュールを静的リンクするなら Python をビルドしなおし、動的にロードされる拡張を使うなら共有オブジェクトをビルドしてインストールする必要があります。

`timing` — 高い精度で経過時間を計測します (`time.clock()` を使ってください)。Python 3.x で削除されました。

41.6 SGI 固有の拡張モジュール

以下は SGI 固有のモジュールで、現在のバージョンの SGI の実情が反映されていないかもしれません。

- `cl` — SGI 圧縮ライブラリへのインタフェースです。
- `sv` — SGI Indigo 上の "simple video" ボード (旧式のハードウェアです) へのインタフェースです。Python 3.x で削除されました。

付録 A

用語集

>>> インタラクティブシェルにおけるデフォルトの Python プロンプトです。インタプリタでインタラクティブに実行されるコード例でよく出てきます。

... インタラクティブシェルにおいて、インデントされたコードブロック、対応する左右の区切り文字の組 (丸括弧 `()`、角括弧 `[]`、波括弧 `{}`、三重引用符) の内側、デコレーターの後に、コードを入力する際に表示されるデフォルトの Python プロンプトです。

2to3 Python 2.x のコードを Python 3.x のコードに変換するツールです。ソースコードを解析してその解析木を巡回 (traverse) することで検知できる、非互換性の大部分を処理します。

2to3 は標準ライブラリの `lib2to3` として利用できます。単体のツールとしての使えるスクリプトが `Tools/scripts/2to3` として提供されています。 [2to3 - Python 2 から 3 への自動コード変換](#) を参照してください。

abstract base class (抽象基底クラス) `abstract-base-classes` は [duck-typing](#) を補完するもので、`hasattr()` などの別のテクニックでは不恰好になる場合に インタフェースを定義する方法を提供します。Python は沢山のビルトイン ABCs を、(`collections` モジュールで) データ構造、(`numbers` モジュールで) 数値型、(`io` モジュールで) ストリーム型で提供しています。 `abc` モジュールを利用して独自の ABC を作成することもできます。

argument 関数を呼び出す際に、[function](#) (または [method](#)) に渡す値です。引数には 2 種類あります。

- キーワード引数: 関数呼び出しの際に引数の前に識別子がついたもの (例: `name=`) や、`**` に続けた辞書の中の値として渡された引数。例えば、次の `complex()` の呼び出しでは、`3` と `5` がキーワード引数です:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- 位置引数: キーワード引数以外の引数。位置引数は引数リストの先頭に書くことができ、また `*` に続けた [iterable](#) の要素として渡すことができます。例えば、次の例では `3` と `5` は両方共位置引数です:

```
complex(3, 5)
complex(*(3, 5))
```

実引数は関数の実体において名前付きのローカル変数に割り当てられます。割り当てを行う規則については `calls` を参照してください。シンタックスにおいて実引数を表すためにあらゆる式を使うことが出来ます。評価された値はローカル変数に割り当てられます。

[仮引数](#)、FAQ の [実引数と仮引数の違いは何ですか?](#) を参照してください。

`attribute` (属性) オブジェクトに関連付けられ、ドット表記式によって名前参照される値です。例えば、オブジェクト `o` が属性 `a` を持っているとき、その属性は `o.a` で参照されます。

`BDFL` 慈悲深き終身独裁者 (Benevolent Dictator For Life) の略です。Python の作者、[Guido van Rossum](#) のことです。

`bytes-like object` (バイト様オブジェクト) `str`、`bytearray` や `memoryview` といった buffer protocol をサポートするオブジェクトです。バイト様オブジェクトは圧縮、バイナリーデータの保存、ソケット経由の送信等の様々な操作に用いることが出来ます。操作によってはバイナリーデータはミューテブルでなくてはなりませんが、その場合全てのバイト様オブジェクトを用いることが出来るわけではありません。

`bytecode` (バイトコード) Python のソースコードはバイトコードへとコンパイルされます。バイトコードは Python プログラムの CPython インタプリタ内部での表現です。バイトコードはまた、`.pyc` や `.pyo` ファイルにキャッシュされ、同じファイルの実行が二度目にはより高速になります (ソースコードからバイトコードへの再度のコンパイルは回避されます)。このバイトコードは、各々のバイトコードに対応するサブルーチン呼び出すような "仮想機械 (*virtual machine*)" で動作する "中間言語 (*intermediate language*)" といえます。重要な注意として、バイトコードは異なる Python 仮想マシン間で動作することは期待されていませんし、Python リリース間で安定でもありません。

バイトコードの命令一覧は [dis モジュール](#) にあります。

`class` (クラス) ユーザー定義オブジェクトを作成するためのテンプレートです。クラス定義は普通、そのクラスのインスタンス上の操作をするメソッドの定義を含みます。

`classic class` (旧スタイルクラス) `object` を継承していないクラス全てを指します。新スタイルクラス (*new-style class*) も参照してください。旧スタイルクラスは Python 3 で削除されます。

`coercion` (型強制) 同じ型の 2 つの引数を要する演算の最中に、ある型のインスタンスを別の型に暗黙のうちに変換することです。例えば、`int(3.15)` は浮動小数点数を整数の 3 にします。しかし、`3+4.5` の場合、各引数は型が異なっていて (一つは整数、一つは浮動小数点数)、加算をする前に同じ型に変換しなければいけません。そうでないと、`TypeError` 例外が投げられます。2 つの被演算子間の型強制は組み込み関数の `coerce` を使って行えます。従って、`3+4.5` は `operator.add(*coerce(3, 4.5))` を呼び出すことに等しく、`operator.add(3.0, 4.5)` という結果になります。型強制を行わない場合、たとえ互換性のある型であっても、すべての引数はプログラマーが、単に `3+4.5` とするのではなく、`float(3)+4.5` というように、同じ型に正規化しなければいけません。

`complex number` (複素数) よく知られている実数系を拡張したもので、すべての数は実部と虚部の和として表されます。虚数は虚数単位 (-1 の平方根) に実数を掛けたもので、一般に数学では i と書かれ、工学では j と書かれます。Python は複素数に組み込みで対応し、後者の表記を取っています。虚部は末尾に `j` をつけて書きます。例えば `3+1j` です。`math` モジュールの複素数版を利用するには、`cmath` を使います。複素数の使用はかなり高度な数学の機能です。必要性を感じなければ、ほぼ間違いなく無視してしまってよいでしょう。

context manager (コンテキストマネージャ) `with` 文で扱われる、`__enter__()` と `__exit__()` メソッドを定義することで環境を制御するオブジェクトです。PEP 343 を参照してください。

CPython python.org で配布されている、Python プログラミング言語の標準的な実装です。”CPython” という単語は、この実装を Jython や IronPython といった他の実装と区別する必要がある場合に利用されます。

decorator (デコレータ) 別の関数を返す関数で、通常、`@wrapper` 構文で関数変換として適用されます。デコレータの一般的な利用例は、`classmethod()` と `staticmethod()` です。

デコレータの文法はシンタックスシュガーです。次の 2 つの関数定義は意味的に同じものです:

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

同じ概念がクラスにも存在しますが、あまり使われません。デコレータについて詳しくは、関数定義およびクラス定義のドキュメントを参照してください。

descriptor (デスクリプタ) メソッド `__get__()`, `__set__()`, あるいは `__delete__()` が定義されている新スタイル (*new-style*) のオブジェクトです。あるクラス属性がデスクリプタである場合、その属性を参照するときに、そのデスクリプタに束縛されている特別な動作を呼び出します。通常、`get`, `set`, `delete` のために `a.b` と書くと、`a` のクラス辞書内でオブジェクト `b` を検索しますが、`b` がデスクリプタの場合にはデスクリプタで定義されたメソッドを呼び出します。デスクリプタの理解は、Python を深く理解する上で鍵となります。というのは、デスクリプタこそが、関数、メソッド、プロパティ、クラスメソッド、静的メソッド、そしてスーパクラスの参照といった多くの機能の基盤だからです。

デスクリプタのメソッドに関して詳しくは、`descriptors` を参照してください。

dictionary (辞書) 任意のキーを値に対応付ける連想配列です。`__hash__()` メソッドと `__eq__()` メソッドを実装した任意のオブジェクトをキーにできます。Perl ではハッシュ (`hash`) と呼ばれています。

dictionary view (ビュー) `dict.viewkeys()`, `dict.viewvalues()`, `dict.viewitems()` が返すオブジェクトを辞書ビュー (dictionary view) と呼びます。これらは辞書のエントリに対する動的なビューで、つまり辞書の変更されるとビューはそれら変更を反映します。辞書ビューを完全なリストにするには `list(dictview)` としてください。辞書ビューオブジェクトを参照してください。

docstring クラス、関数、モジュールの最初の式である文字列リテラルです。そのスイートの実行時には無視されますが、コンパイラによって識別され、そのクラス、関数、モジュールの `__doc__` 属性として保存されます。イントロスペクションできる (訳注: 属性として参照できる) ので、オブジェクトのドキュメントを書く標準的な場所です。

duck-typing あるオブジェクトが正しいインタフェースを持っているかを決定するのにオブジェクトの型を見ないプログラミングスタイルです。代わりに、単純にオブジェクトのメソッドや属性が呼ばれたり使われたりします。(「アヒルのように見えて、アヒルのように鳴けば、それはアヒルである。’) インタフェースを型より重視することで、上手くデザインされたコードは、ポリモーフィックな代替を許し

て柔軟性を向上させます。ダックタイピングは `type()` や `isinstance()` による判定を避けます。(ただし、ダックタイピングを **抽象基底クラス** で補完することもできます。) その代わり、典型的に `hasattr()` 判定や **EAFP** プログラミングを利用します。

EAFP 「認可をとるより許しを請う方が容易 (easier to ask for forgiveness than permission、マーフィーの法則)」の略です。この Python で広く使われているコーディングスタイルでは、通常は有効なキーや属性が存在するものと仮定し、その仮定が誤っていた場合に例外を捕捉します。この簡潔で手早く書けるコーディングスタイルには、`try` 文および `except` 文がたくさんあるのが特徴です。このテクニックは、C のような言語でよく使われている **LBYL** スタイルと対照的なものです。

expression (式) 何かの値に評価される、一つづきの構文 (a piece of syntax). 言い換えると、リテラル、名前、属性アクセス、演算子や関数呼び出しといった、値を返す式の要素の組み合わせ。他の多くの言語と違い、Python は言語の全ての構成要素が式というわけではありません。 `print` や `if` のように、式にはならない、文 (*statement*) もあります。代入も式ではなく文です。

extension module (拡張モジュール) C や C++ で書かれたモジュールで、Python の C API を利用して Python コアやユーザーコードとやりとりします。

file object (ファイルオブジェクト) 下位のリソースへのファイル志向 API (`read()` や `write()` メソッドを持つもの) を公開しているオブジェクトです。ファイルオブジェクトは、作成された手段によって、実際のディスク上のファイルや、その他のタイプのストレージや通信デバイス (例えば、標準入出力、インメモリバッファ、ソケット、パイプ、等) へのアクセスを媒介できます。ファイルオブジェクトは *file-like objects* や *streams* と呼ばれます。

ファイルオブジェクトには 3 つの種類があります: 生のバイナリーファイル、バッファされたバイナリーファイル、そしてテキストファイルです。これらのインターフェースは `io` モジュール内で定義されています。ファイルオブジェクトを作成する標準的な方法は `open()` 関数を利用することです。

file-like object *file object* と同義です。

finder モジュールの *loader* を探すオブジェクト。 `findmodule()` という名前のメソッドを実装していなければなりません。詳細については **PEP 302** を参照してください。

floor division 一番近い小さい整数に丸める数学除算。floor division 演算子は `//` です。例えば、 `11 // 4` は 2 になり、 `float` の true division の結果 `2.75` と異なります。 `(-11) // 4` は `-2.75` を小さい方に丸めるので `-3` になることに注意してください。 **PEP 238** を参照してください。

関数 (関数) 呼び出し側に値を返す一連の文のことです。関数には 0 以上の **実引数** を渡すことが出来ます。実体の実行時に引数を使用することが出来ます。 **仮引数**、**メソッド**、`function` を参照してください。

`__future__` 互換性のない新たな機能を現在のインタプリタで有効にするためにプログラマが利用できる擬似モジュールです。例えば、式 `11/4` は現状では 2 になります。この式を実行しているモジュールで

```
from __future__ import division
```

を行って 真の除算操作 (*true division*) を有効にすると、式 `11/4` は `2.75` になります。実際に `__future__` モジュールを `import` してその変数を評価すれば、新たな機能が初めて追加されたのがいつで、いつデフォルトの機能になる予定かわかります。

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection (ガベージコレクション) それ以上使われなくなったメモリを解放する処理です。Python は、参照カウントと、循環参照を見つけて破壊する循環ガベージコレクタと、を使ってガベージコレクションを行います。

generator A function which returns an iterator. It looks like a normal function except that it contains `yield` statements for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function. Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

generator expression (ジェネレータ式) イテレータを返す式です。普通の式に、ループ変数を定義する `for` 式、範囲、そして省略可能な `if` 式がつづいているように見えます。こうして構成された式は、外側の関数に向けて値を生成します:

```
>>> sum(i*i for i in range(10))           # sum of squares 0, 1, 4, ... 81
285
```

GIL [global interpreter lock](#) を参照してください。

global interpreter lock (グローバルインタプリタロック) *CPython* インタプリタが利用している、一度に Python のバイトコード (*bytecode*) 実行するスレッドは一つだけであることを保証する仕組みです。これにより (*dict* などの重要な組み込み型を含む) などのオブジェクトモデルが同時アクセスに対して暗黙的に安全になるので、CPython の実装がシンプルになります。インタプリタ全体をロックすることで、マルチプロセッサマシンが生じる並列化のコストに対して、楽にインタプリタをマルチスレッド化できます。

ただし、標準あるいは外部のいくつかの拡張モジュールは、圧縮やハッシュ計算などの計算の重い処理をするときに GIL を解放するように設計されています。また、I/O 処理をするときも GIL は常に解放されます。

過去に”自由なマルチスレッド化”したインタプリタ (供用されるデータを細かい粒度でロックする) が開発されましたが、一般的なシングルスプロセッサの場合のパフォーマンスが悪かったので成功しませんでした。このパフォーマンスの問題を克服しようとする、実装がより複雑になり保守コストが増加すると考えられています。

hashable (ハッシュ可能) ハッシュ可能なオブジェクトとは、生存期間中変わらないハッシュ値を持ち (`__hash__()` メソッドが必要)、他のオブジェクトと比較ができる (`__eq__()` か `__cmp__()` メソッドが必要) オブジェクトです。同値なハッシュ可能オブジェクトは必ず同じハッシュ値を持つ必要があります。

ハッシュ可能なオブジェクトは辞書のキーや集合のメンバーとして使えます。辞書や集合のデータ構造は内部でハッシュ値を使っているからです。

Python のイミュータブルな組み込みオブジェクトはハッシュ可能ですが、ミュータブルなコンテ

ナ (例えばリストや辞書) はハッシュ不可能です。ユーザー定義のクラスのインスタンスであるようなオブジェクトはデフォルトでハッシュ可能です。それらは全て非等価を比較し (自身を除いて)、ハッシュ値は `id()` より得られます。

IDLE Python の統合開発環境 (Integrated DeveLopment Environment) です。IDLE は Python の標準的な配布に同梱されている基本的な機能のエディタとインタプリタ環境です。

immutable (イミュータブル) 固定の値を持ったオブジェクトです。イミュータブルなオブジェクトには、数値、文字列、およびタプルなどがあります。これらのオブジェクトは値を変えられません。別の値を記憶させる際には、新たなオブジェクトを作成しなければなりません。イミュータブルなオブジェクトは、固定のハッシュ値が必要となる状況で重要な役割を果たします。辞書のキーがその例です。

integer division (整数除算) 剰余を考慮しない数学的除算です。例えば、式 $11/4$ は現状では 2.75 ではなく 2 になります。これは切り捨て除算 (*floor division*) とも呼ばれます。二つの整数間で除算を行うと、結果は (端数切捨て関数が適用されて) 常に整数になります。しかし、被演算子の一方が (*float* のような) 別の数値型の場合、演算の結果は共通の型に型強制されます (型強制 (*coercion*) 参照)。例えば、浮動小数点数で整数を除算すると結果は浮動小数点になり、場合によっては端数部分を伴います。// 演算子を / の代わりに使うと、整数除算を強制できます。 `__future__` も参照してください。

importing あるモジュールの Python コードが別のモジュールの Python コードで使えるようにする処理です。

importer モジュールを探してロードするオブジェクト。 *finder* と *loader* のどちらでもあるオブジェクト。

interactive (対話的) Python には対話的インタプリタがあり、文や式をインタプリタのプロンプトに入力すると即座に実行されて結果を見ることができます。 `python` と何も引数を与えずに実行してください。(コンピュータのメインメニューから Python の対話的インタプリタを起動できるかもしれません。) 対話的インタプリタは、新しいアイデアを試してみたり、モジュールやパッケージの中を覗いてみる (`help(x)` を覚えておいてください) のに非常に便利なツールです。

interpreted Python はインタプリタ形式の言語であり、コンパイラ言語の対極に位置します。(バイトコードコンパイラがあるために、この区別は曖昧ですが。) ここでのインタプリタ言語とは、ソースコードのファイルを、まず実行可能形式にしてから実行させるといった操作なしに、直接実行できることを意味します。インタプリタ形式の言語は通常、コンパイラ形式の言語よりも開発 / デバッグのサイクルは短いものの、プログラムの実行は一般に遅いです。対話的 (*interactive*) も参照してください。

iterable (反復可能オブジェクト) 要素を一つずつ返せるオブジェクトです。反復可能オブジェクトの例には、 (`list`, `str`, `tuple` といった) 全てのシーケンス型や、 *dict* や *file* といった幾つかの非シーケンス型、あるいは `__iter__()` か `__getitem__()` メソッドを実装したクラスのインスタンスが含まれます。反復可能オブジェクトは `for` ループ内やその他多くのシーケンス (訳注: ここでのシーケンスとは、シーケンス型ではなくただの列という意味) が必要となる状況 (`zip()`, `map()`, ...) で利用できます。反復可能オブジェクトを組み込み関数 `iter()` の引数として渡すと、オブジェクトに対するイテレータを返します。このイテレータは一連の値を引き渡す際に便利です。反復可能オブジェクトを使う際には、通常 `iter()` を呼んだり、イテレータオブジェクトを自分で扱う必要はありません。 `for` 文ではこの操作を自動的に扱い、無名の変数を作成してループの間イテレータを記憶します。イテレータ (*iterator*) シーケンス (*sequence*), およびジェネレータ (*generator*) も参照してください。

iterator (イテレータ) データの流れを表現するオブジェクトです。イテレータの `next()` メソッドを繰り返し呼び出すと、流れの中の要素を一つずつ返します。データがなくなると、代わりに `StopIteration`

例外を送出します。その時点で、イテレータオブジェクトは尽きており、それ以降は `next()` を何度呼んでも `StopIteration` を送じます。イテレータは、そのイテレータオブジェクト自体を返す `__iter__()` メソッドを実装しなければならないので、イテレータは他の iterable を受理するほとんどの場所で利用できます。はっきりとした例外は複数の反復を行うようなコードです。(list のような) コンテナオブジェクトは、自身を `iter()` 関数にオブジェクトに渡したり for ループ内で使うたびに、新たな未使用のイテレータを生成します。これをイテレータで行おうとすると、前回のイテレーションで使用済みの同じイテレータオブジェクトを単純に返すため、空のコンテナのようになってしまいます。

詳細な情報は [イテレータ型](#) にあります。

key function (キー関数) キー関数、あるいは照合関数とは、ソートや順序比較のための値を返す呼び出し可能オブジェクト (callable) です。例えば、`locale.strxfrm()` をキー関数に使用すれば、ロケール依存のソートの慣習にのっとったソートキーを返します。

Python には、キー関数を受け付けて要素がどのように順序付けられたりグループ化されたりするかを制御するような、いくつかのツールがあります。例えば、`min()`, `max()`, `sorted()`, `list.sort()`, `heapq.nsmallest()`, `heapq.nlargest()`, `itertools.groupby()` です。

キー関数を作る方法はいくつかあります。例えば、`str.lower()` メソッドをキー関数として使って大文字小文字を区別しないソートができます。他には、lambda 式を使って `lambda r: (r[0], r[2])` のようなアドホックなキー関数を作ることができます。また、`operator` モジュールは `attrgetter()`, `itemgetter()`, `methodcaller()` というキー関数コンストラクタを提供しています。キー関数の作り方、使い方に関する例は、Sorting HOW TO を参照してください。

keyword argument [実引数](#) を参照してください。

lambda (ラムダ) 無名のインライン関数で、関数が呼び出されたときに評価される 1 つの式 (*expression*) を含みます。ラムダ関数を作る構文は `lambda [parameters]: expression` です。

LBYL 「ころばぬ先の杖 (look before you leap)」の略です。このコーディングスタイルでは、呼び出しや検索を行う前に、明示的に前提条件 (pre-condition) 判定を行います。[EAFP](#) アプローチと対照的で、if 文がたくさん使われるのが特徴的です。

マルチスレッド化された環境では、LBYL アプローチは「見る」過程と「飛ぶ」過程の競合状態を引き起こすリスクがあります。例えば、`if key in mapping: return mapping[key]` というコードは、判定の後、別のスレッドが探索の前に `mapping` から `key` を取り除くと失敗します。この問題は、ロックするか EAFP アプローチを使うことで解決できます。

list (リスト) Python の組み込みのシーケンス (*sequence*) です。リストという名前ですが、リンクリストではなく、他の言語で言う配列 (array) と同種のもので、要素へのアクセスは $O(1)$ です。

list comprehension (リスト内包表記) シーケンス内の全てあるいは一部の要素を処理して、その結果からなるリストを返す、コンパクトな書き方です。`result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` とすると、0 から 255 までの偶数を 16 進数表記 (0x..) した文字列からなるリストを生成します。if 節はオプションです。if 節がない場合、`range(256)` の全ての要素が処理されます。

loader モジュールをロードするオブジェクト。`load_module()` という名前のメソッドを定義していなけ

ればなりません。詳細は [PEP 302](#) を参照してください。

magic method [special method](#) のくだけた同義語です。

mapping (マップ、マッピング) 任意のキーに対する検索をサポートしていて、`Mapping` が `MutableMapping` の抽象基底クラスを実装しているコンテナオブジェクト。例えば、`dict`, `collections.defaultdict`, `collections.OrderedDict`, `collections.Counter` はマップ型です。

metaclass (メタクラス) クラスのクラスです。クラス定義は、クラス名、クラスの辞書と、基底クラスのリストを作ります。メタクラスは、それら 3 つを引数として受け取り、クラスを作る責任を負います。ほとんどのオブジェクト指向言語は (訳注:メタクラスの) デフォルトの実装を提供しています。Python が特別なのはカスタムのメタクラスを作成できる点です。ほとんどのユーザーにとって、メタクラスは全く必要のないものです。しかし、一部の場面では、メタクラスは強力でエレガントな方法を提供します。たとえば属性アクセスのログを取ったり、スレッドセーフ性を追加したり、オブジェクトの生成を追跡したり、シングルトンを実装するなど、多くの場面で利用されます。

詳細は [metaclasses](#) を参照してください。

method (メソッド) クラス本体の中で定義された関数。そのクラスのインスタンスの属性として呼び出された場合、メソッドはインスタンスオブジェクトを第一引数 (*argument*) として受け取ります (この第一引数は通常 `self` と呼ばれます)。関数 と [ネストされたスコープ](#) も参照してください。

method resolution order (メソッド解決順序) 探索中に基底クラスが構成要素を検索される順番です。2.3 以降の Python インタープリタが使用するアルゴリズムの詳細については [The Python 2.3 Method Resolution Order](#) を参照してください。

モジュール (モジュール) Python コードの組織単位としてはたらくオブジェクトです。モジュールは任意の Python オブジェクトを含む名前空間を持ちます。モジュールは [importing](#) の処理によって Python に読み込まれます。

[パッケージ](#) を参照してください。

MRO [method resolution order](#) を参照してください。

mutable (ミュータブル) ミュータブルなオブジェクトは、`id()` を変えることなく値を変更できます。イミュータブル (*immutable*) も参照してください。

named tuple (名前付きタプル) タプルに似ていて、インデックス指定できる要素に名前付き属性でもアクセス出来るクラスです (例えば、`time.localtime()` はタプルに似たオブジェクトを返し、その `year` には `t[0]` のようなインデックスによるアクセスと、`t.tm_year` のような名前付き要素としてのアクセスが可能です)。

名前付きタプルには、`time.struct_time` のような組み込み型もありますし、通常のクラス定義によって作成することもできます。名前付きタプルを `collections.namedtuple()` ファクトリ関数で作成することもできます。最後の方法で作った名前付きタプルには自動的に、`Employee(name='jones', title='programmer')` のような自己ドキュメント表現 (self-documenting representation) などの機能が付いてきます。

namespace (名前空間) 変数を記憶している場所です。名前空間は辞書を用いて実装されています。名前

空間には、ローカル、グローバル、組み込み名前空間、そして(メソッド内の)オブジェクトのネストされた名前空間があります。例えば、関数 `_builtin_.open()` と `os.open()` は名前空間で区別されます。名前空間はまた、ある関数をどのモジュールが実装しているかをはっきりさせることで、可読性やメンテナンス性に寄与します。例えば、`random.seed()`、`itertools.izip()` と書くことで、これらの関数がそれぞれ `random` モジュールや `itertools` モジュールで実装されていることがはっきりします。

nested scope (ネストされたスコープ) 外側で定義されている変数を参照する機能。具体的に言えば、ある関数が別の関数の中で定義されている場合、内側の関数は外側の関数中の変数を参照できます。ネストされたスコープからは変数の参照だけができ、外側の変数を変更できないことに注意してください。それは常に最も内側のスコープに対する書き込みになります。対照的に、ローカル変数は最も内側のスコープ内の読み書き両方します。同様に、グローバル変数を使うとグローバル名前空間の値を読み書きします。

new-style class (新スタイルクラス) `object` から継承したクラス全てを指します。これには `list` や `dict` のような全ての組み込み型が含まれます。`__slots__()`、デスクリプタ、プロパティ、`__getattr__()` といった、Python の新しい機能を使うのは新スタイルクラスだけです。

より詳しい情報は `newstyle` を参照してください。

object (オブジェクト) 状態(属性や値)と定義された振る舞い(メソッド)をもつ全てのデータ。もしくは、全ての新スタイルクラス (`new-style class`) の究極の基底クラスのこと。

package (パッケージ) サブモジュールや再帰的にサブパッケージを含むことの出来る `module` のことです。専門的には、パッケージは `__path__` 属性を持つ Python オブジェクトです。

parameter (仮引数) 名前付の実体で `term:関数<function>` (や `term:メソッド<method>`) の定義において関数が受ける **実引数** を明示します。仮引数には 4 種類あります:

- 位置またはキーワード: **位置** または **キーワード引数** として渡すことができる引数を指定します。これはたとえば以下の `foo` や `bar` のように、デフォルトの仮引数の種類です:

```
def func(foo, bar=None): ...
```

- 位置専用: 位置によってのみ与えられる引数を指定します。Python に位置専用仮引数を定義する文法はありません。しかし、組み込み関数には位置専用仮引数を持つもの(例: `abs()`)があります。
- 可変長位置: (他の仮引数で既に受けられた任意の位置引数に加えて) 任意の個数の位置引数が与えられることを指定します。このような仮引数は、以下の `args` のように仮引数名の前に `*` をつけることで定義できます:

```
def func(*args, **kwargs): ...
```

- 可変長キーワード: (他の仮引数で既に受けられた任意のキーワード引数に加えて) 任意の個数のキーワード引数が与えられることを指定します。このような仮引数は、上の例の `kwargs` のように仮引数名の前に `**` をつけることで定義できます。

仮引数はオプションと必須の引数のどちらも指定でき、オプションの引数にはデフォルト値も指定できます。

argument、FAQ の 実引数と仮引数の違いは何ですか?、function を参照してください。

PEP Python Enhancement Proposal。PEP は、Python コミュニティに対して情報を提供する、あるいは Python の新機能やその過程や環境について記述する設計文書です。PEP は、機能についての簡潔な技術的仕様と提案する機能の論拠 (理論) を伝えるべきです。

PEP は、新機能の提案にかかる、コミュニティによる問題提起の集積と Python になされる設計決断の文書化のための最上位の機構となることを意図しています。PEP の著者にはコミュニティ内の合意形成を行うこと、反対意見を文書化することの責務があります。

PEP 1 を参照してください。

位置引数 実引数 を参照してください。

Python 3000 Python 3.x リリースラインのニックネームです。(Python 3 が遠い将来の話だった頃に作られた言葉です。) ”Py3k” と略されることもあります。

Pythonic 他の言語で一般的な考え方で書かれたコードではなく、Python の特に一般的なイディオムに従った考え方やコード片。例えば、Python の一般的なイディオムでは `for` 文を使ってイテラブルのすべての要素に渡ってループします。他の多くの言語にはこの仕組みはないので、Python に慣れていない人は代わりに数値のカウンターを使うかもしれません:

```
for i in range(len(food)):\n    print food[i]
```

これに対し、きれいな Pythonic な方法は:

```
for piece in food:\n    print piece
```

reference count (参照カウント) あるオブジェクトに対する参照の数。参照カウントが 0 になったとき、そのオブジェクトは破棄されます。参照カウントは通常は Python のコード上には現れませんが、*CPython* 実装の重要な要素です。*sys* モジュールは、プログラマーが任意のオブジェクトの参照カウントを知るための *getrefcount()* 関数を提供しています。

__slots__ 新スタイルクラス (*new-style class*) 内で、インスタンス属性の記憶に必要な領域をあらかじめ定義しておき、それとひきかえにインスタンス辞書を排除してメモリの節約を行うための宣言です。これはよく使われるテクニックですが、正しく動作させるのには少々手際を要するので、例えばメモリが死活問題となるようなアプリケーション内にインスタンスが大量に存在するといった稀なケースを除き、使わないのがベストです。

sequence (シーケンス) 特殊メソッド *__getitem__()* で整数インデックスによる効率的な要素へのアクセスをサポートし、*len()* で長さを返すような反復可能オブジェクト (*iterable*) です。組み込みシーケンス型には、*list*、*str*、*tuple*、*unicode* などがあります。*dict* は *__getitem__()* と *__len__()* もサポートしますが、検索の際に任意の変更不能 (*immutable*) なキーを使うため、シーケンスではなくマップ (mapping) とみなされているので注意してください。

slice (スライス) 多くの場合、シーケンス (*sequence*) の一部を含むオブジェクト。スライスは、添字記号 `[]` で数字の間にコロンを書いたときに作られます。例えば、`variable_name[1:3:5]` です。添字記

号は `slice` オブジェクトを内部で利用しています。(もしくは、古いバージョンの、`__getslice__()` と `__setslice__()` を利用します。)

special method (特殊メソッド) ある型に特定の操作、例えば加算をするために Python から暗黙に呼び出されるメソッド。この種類のメソッドは、メソッド名の最初と最後にアンダースコア 2 つがついていません。特殊メソッドについては `specialnames` で解説されています。

statement (文) 文はスイート(コードの”ブロック”)に不可欠な要素です。文は `式` かキーワードから構成されるもののどちらかです。後者には `if`、`while`、`for` があります。

struct sequence A tuple with named elements. Struct sequences expose an interface similiar to *named tuple* in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like `make()` or `asdict()`. Examples of struct sequences include `sys.float_info` and the return value of `os.stat()`.

triple-quoted string (三重クオート文字列) 3 つの連続したクオート記号 (") かアポストロフィー (') で囲まれた文字列。通常の (一重) クオート文字列に比べて表現できる文字列に違いはありませんが、幾つかの理由で有用です。1 つか 2 つの連続したクオート記号をエスケープ無しに書くことができますし、行継続文字 (\) を使わなくても複数行にまたがることのできる、ドキュメンテーション文字列を書く時に特に便利です。

型 (型) Python オブジェクトの型はオブジェクトがどのようなものかを決めます。あらゆるオブジェクトは型を持っています。オブジェクトの型は `__class__` 属性でアクセスしたり、`type(obj)` で取得したり出来ます。

universal newlines テキストストリームの解釈法の一つで、以下のすべてを行末と認識します: Unix の行末規定 '\n'、Windows の規定 '\r\n'、古い Macintosh の規定 '\r'。利用法について詳しくは、**PEP 278** と **PEP 3116**、さらに `str.splitlines()` も参照してください。

virtual environment (仮想環境) 協調的に切り離された実行環境です。これにより Python ユーザとアプリケーションは同じシステム上で動いている他の Python アプリケーションの挙動に干渉することなく Python パッケージのインストールと更新を行うことができます。

virtual machine (仮想マシン) 完全にソフトウェアにより定義されたコンピュータ。Python の仮想マシンは、バイトコードコンパイラが出力したバイトコード (*bytecode*) を実行します。

Zen of Python (Python の悟り) Python を理解し利用する上での導きとなる、Python の設計原則と哲学をリストにしたものです。対話プロンプトで `import this` とするとこのリストを読めます。

付録 B

このドキュメントについて

このドキュメントは、Python のドキュメントを主要な目的として作られた ドキュメントプロセッサの [Sphinx](#) を利用して、[reStructuredText](#) 形式のソースから生成されました。

ドキュメントとそのツールセットの開発は、Python 自身と同様に完全にボランティアの努力です。もしあなたが貢献したいなら、どのようにすればよいかについて [reporting-bugs](#) ページをご覧ください。新しいボランティアはいつでも歓迎です!

多大な感謝を:

- Fred L. Drake, Jr., オリジナルの Python ドキュメントツールセットの作成者で、ドキュメントの多くを書きました。
- [Docutils](#) プロジェクト. [reStructuredText](#) と [docutils](#) ツールセットを作成しました。
- Fredrik Lundh の [Alternative Python Reference](#) プロジェクトから Sphinx は多くのアイデアを得ました。

B.1 Python ドキュメント 貢献者

多くの方々が Python 言語、Python 標準ライブラリ、そして Python ドキュメンテーションに貢献してくれています。ソース配布物の [Misc/ACKS](#) に、それら貢献してくれた人々を部分的にはありますがリストアップしてあります。

Python コミュニティからの情報提供と貢献がなければこの素晴らしいドキュメンテーションは生まれませんでした – ありがとう!

付録 C

歴史とライセンス

C.1 Python の歴史

Python は 1990 年代の始め、オランダにある Stichting Mathematisch Centrum (CWI, <https://www.cwi.nl/> 参照) で Guido van Rossum によって ABC と呼ばれる言語の後継言語として生み出されました。その後多くの人々が Python に貢献していますが、Guido は今日でも Python 製作者の先頭に立っています。

1995 年、Guido は米国ヴァージニア州レストンにある Corporation for National Reserch Initiatives (CNRI, <https://www.cnri.reston.va.us/> 参照) で Python の開発に携わり、いくつかのバージョンをリリースしました。

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen Python-Labs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see <https://www.zope.org/>). In 2001, the Python Software Foundation (PSF, see <https://www.python.org/psf/>) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

Python のリリースは全てオープンソース (オープンソースの定義は <https://opensource.org/> を参照してください) です。歴史的にみて、ごく一部を除くほとんどの Python リリースは GPL 互換になっています; 各リリースについては下表にまとめてあります。

リリース	ベース	西暦年	権利	GPL 互換
0.9.0 - 1.2	n/a	1991-1995	CWI	yes
1.3 - 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 以降	2.1.1	2001-現在	PSF	yes

注釈: 「GPL 互換」という表現は、Python が GPL で配布されているという意味ではありません。Python のライセンスは全て、GPL と違い、変更したバージョンを配布する際に変更をオープンソースにしなくてもかまいません。GPL 互換のライセンスの下では、GPL でリリースされている他のソフトウェアと Python を組み合わせられますが、それ以外のライセンスではそうではありません。

Guido の指示の下、これらのリリースを可能にくださった多くのボランティアのみなさんに感謝します。

C.2 Terms and conditions for accessing or otherwise using Python

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 2.7.17

1. This LICENSE AGREEMENT is between the Python Software Foundation,
→ ("PSF"), and
the Individual or Organization ("Licensee") accessing and otherwise,
→ using Python
2.7.17 software in source or binary form and its associated,
→ documentation.
2. Subject to the terms and conditions of this License Agreement, PSF,
→ hereby
grants Licensee a nonexclusive, royalty-free, world-wide license to,
→ reproduce,
analyze, test, perform and/or display publicly, prepare derivative,
→ works,
distribute, and otherwise use Python 2.7.17 alone or in any derivative
version, provided, however, that PSF's License Agreement and PSF's,
→ notice of
copyright, i.e., "Copyright ^c2^a9 2001-2019 Python Software,
→ Foundation; All Rights
Reserved" are retained in Python 2.7.17 alone or in any derivative,
→ version
prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or
incorporates Python 2.7.17 or any part thereof, and wants to make the
derivative work available to others as provided herein, then Licensee,
→ hereby
agrees to include in any such work a brief summary of the changes made,
→ to Python
2.7.17.

4. PSF is making Python 2.7.17 available to Licensee on an "AS IS" basis.
 PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY
 ↳OF
 EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY
 ↳REPRESENTATION OR
 WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR
 ↳THAT THE
 USE OF PYTHON 2.7.17 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.7.17
 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A
 ↳RESULT OF
 MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.7.17, OR ANY
 ↳DERIVATIVE
 THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material
 ↳breach of
 its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any
 ↳relationship
 of agency, partnership, or joint venture between PSF and Licensee. ↳
 ↳This License
 Agreement does not grant permission to use PSF trademarks or trade name
 ↳in a
 trademark sense to endorse or promote products or services of Licensee,
 or any
 third party.
8. By copying, installing or otherwise using Python 2.7.17, Licensee agrees
 to be bound by the terms and conditions of this License Agreement.

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license

(次のページに続く)

(前のページからの続き)

- to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
 4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
 5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
 6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.
 7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement,

(次のページに続く)

(前のページからの続き)

Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: <http://hdl.handle.net/1895.22/1013>."

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

(次のページに続く)

(前のページからの続き)

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3 取り入れられているソフトウェアのライセンスと謝辞

この節では、Python 配布物に取り入れられているサードパーティソフトウェアのライセンスと謝辞の、不完全だが成長し続けるリストを記述します。

C.3.1 メルセンヌツイスター (Mersenne Twister)

`random` モジュールは、<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html> からダウンロードしたコードに基づいたコードベースを含んでいます。オリジナルのコードのコメントをそのまま次に示します:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
```

(次のページに続く)

(前のページからの続き)

3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

C.3.2 ソケット

`socket` モジュールは、WIDE プロジェクト <http://www.wide.ad.jp/> の別々のソースファイルにコーディングされていた `getaddrinfo()` 関数と `getnameinfo()` 関数を使っています:

Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)

(次のページに続く)

(前のページからの続き)

HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.3 浮動小数点数例外の制御

fpectl モジュールのソースコードは次の告知を含んでいます:

```
-----  
/                               Copyright (c) 1996.                               \  
|                               The Regents of the University of California.          \  
|                               All rights reserved.                                \  
|                                                                                 \  
| Permission to use, copy, modify, and distribute this software for                \  
| any purpose without fee is hereby granted, provided that this en-               \  
| tire notice is included in all copies of any software which is or               \  
| includes a copy or modification of this software and in all                    \  
| copies of the supporting documentation for such software.                       \  
|                                                                                 \  
| This work was produced at the University of California, Lawrence                 \  
| Livermore National Laboratory under contract no. W-7405-ENG-48                  \  
| between the U.S. Department of Energy and The Regents of the                   \  
| University of California for the operation of UC LLNL.                         \  
|                                                                                 \  
|                               DISCLAIMER                                           \  
|                                                                                 \  
| This software was prepared as an account of work sponsored by an                \  
| agency of the United States Government. Neither the United States               \  
| Government nor the University of California nor any of their em-                \  
| ployees, makes any warranty, express or implied, or assumes any                 \  
| liability or responsibility for the accuracy, completeness, or                  \  
| usefulness of any information, apparatus, product, or process                   \  
| disclosed, or represents that its use would not infringe                       \  
| privately-owned rights. Reference herein to any specific commer-               \  
| cial products, process, or service by trade name, trademark,                   \  
| manufacturer, or otherwise, does not necessarily constitute or                 \  
| imply its endorsement, recommendation, or favoring by the United              \  
| States Government or the University of California. The views and                \  
| opinions of authors expressed herein do not necessarily state or               \  
| reflect those of the United States Government or the University                 \  
| of California, and shall not be used for advertising or product                 \  
| \ endorsement purposes.                                                         \  
-----
```

C.3.4 MD5 メッセージダイジェストアルゴリズム

md5 モジュールのソースコードは次の告知を含んでいます:

Copyright (C) 1999, 2002 Aladdin Enterprises. All rights reserved.

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose text is available at

<http://www.ietf.org/rfc/rfc1321.txt>

The code is derived from the text of the RFC, including the test suite (section A.5) but excluding the rest of Appendix A. It does not include any code or documentation that is identified in the RFC as being copyrighted.

The original and principal author of md5.h is L. Peter Deutsch <ghost@aladdin.com>. Other authors are noted in the change history that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed references to Ghostscript; clarified derivation from RFC 1321; now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5); added conditionalization for C++ compilation from Martin Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.

C.3.5 非同期ソケットサービス

asynchat モジュールと *asyncore* モジュールは次の告知を含んでいます:

Copyright 1996 by Sam Rushing

(次のページに続く)

(前のページからの続き)

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Sam Rushing not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.6 Cookie の管理

`Cookie` モジュールは次の告知を含んでいます:

Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Timothy O'Malley not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.7 実行の追跡

`trace` モジュールは次の告知を含んでいます:

portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err... reserved and offered to the public under the terms of the
Python 2.2 license.

Author: Zooko O'Whielacronx

<http://zooko.com/>

<mailto:zooko@zooko.com>

Copyright 2000, Mojam Media, Inc., all rights reserved.

Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.

Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.

Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.

Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.

C.3.8 UUencode 関数と UUdecode 関数

`uu` モジュールは次の告知を含んでいます:

Copyright 1994 by Lance Ellinghouse

Cathedral City, California Republic, United States of America.

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.

LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:

(次のページに続く)

(前のページからの続き)

- Use `binascii` module to do the actual line-by-line conversion between `ascii` and `binary`. This results in a 1000-fold speedup. The C version is still 5 times faster, though.
- Arguments more compliant with Python standard

C.3.9 XML リモートプロシージャコール

`xmlrpclib` モジュールは次の告知を含んでいます:

The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB

Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and its associated documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Secret Labs AB or the author not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.10 `test_epoll`

`test_epoll` は次の告知を含んでいます:

Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to

(次のページに続く)

(前のページからの続き)

permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

C.3.11 Select kqueue

`select` は `kqueue` インターフェースについての次の告知を含んでいます:

Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.12 strtod と dtoa

The file `Python/dtoa.c`, which supplies C functions `dtoa` and `strtod` for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from <http://www.netlib.org/fp/>. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:


```

/*****
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 *****/

```

C.3.13 OpenSSL

The modules *hashlib*, *posix*, *ssl*, *crypt* use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and Mac OS X installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here:

```

LICENSE ISSUES
=====

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
-----

/* =====
 * Copyright (c) 1998-2008 The OpenSSL Project. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the

```

(次のページに続く)

(前のページからの続き)

```

*   distribution.
*
* 3. All advertising materials mentioning features or use of this
*   software must display the following acknowledgment:
*   "This product includes software developed by the OpenSSL Project
*   for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
*
* 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
*   endorse or promote products derived from this software without
*   prior written permission. For written permission, please contact
*   openssl-core@openssl.org.
*
* 5. Products derived from this software may not be called "OpenSSL"
*   nor may "OpenSSL" appear in their names without prior written
*   permission of the OpenSSL Project.
*
* 6. Redistributions of any form whatsoever must retain the following
*   acknowledgment:
*   "This product includes software developed by the OpenSSL Project
*   for use in the OpenSSL Toolkit (http://www.openssl.org/)"
*
* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
* EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
* OF THE POSSIBILITY OF SUCH DAMAGE.
* =====
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com).  This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/

```

Original SSLeay License

```

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
* All rights reserved.
*
* This package is an SSL implementation written
* by Eric Young (eay@cryptsoft.com).
* The implementation was written so as to conform with Netscapes SSL.
*

```

(次のページに続く)

(前のページからの続き)

```
* This library is free for commercial and non-commercial use as long as
* the following conditions are aheared to. The following conditions
* apply to all code found in this distribution, be it the RC4, RSA,
* lhash, DES, etc., code; not just the SSL code. The SSL documentation
* included with this distribution is covered by the same copyright terms
* except that the holder is Tim Hudson (tjh@cryptsoft.com).
*
* Copyright remains Eric Young's, and as such any Copyright notices in
* the code are not to be removed.
* If this package is used in a product, Eric Young should be given attribution
* as the author of the parts of the library used.
* This can be in the form of a textual message at program startup or
* in documentation (online or textual) provided with the package.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the copyright
*    notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
* 3. All advertising materials mentioning features or use of this software
*    must display the following acknowledgement:
*    "This product includes cryptographic software written by
*      Eric Young (eay@cryptsoft.com)"
*    The word 'cryptographic' can be left out if the rouines from the library
*    being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof) from
*    the apps directory (application code) you must include an acknowledgement:
*    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed. i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/
```

C.3.14 expat

The pyexpat extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.15 libffi

The _ctypes extension is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi`:

```
Copyright (c) 1996-2008 Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
`Software'), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
```

(次のページに続く)

(前のページからの続き)

WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.

C.3.16 zlib

The `zlib` extension is built using an included copy of the zlib sources if the zlib version found on the system is too old to be used for the build:

Copyright (C) 1995-2010 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly
jloup@gzip.org

Mark Adler
madler@alumni.caltech.edu

付録 D

Copyright

Python and this documentation is:

Copyright 2001-2019 Python Software Foundation. All rights reserved.

Copyright 2000 BeOpen.com. All rights reserved.

Copyright 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

ライセンスおよび許諾に関する完全な情報は、[歴史とライセンス](#) を参照してください。

参考文献

- [C99] ISO/IEC 9899:1999. "Programming languages – C." この標準のパブリックドラフトが参照できます:
<http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf>。

Python モジュール索引

—
 __builtin__, 1546
 __future__, 1566
 __main__, 1548
 _winreg (*Windows*), 1684

a

abc, 1557
 aepack (*Mac*), 1744
 aetools (*Mac*), 1743
 aetypes (*Mac*), 1746
 aifc, 1264
 al (*IRIX*), 1749
 AL (*IRIX*), 1751
 anydbm, 405
 applesingle (*Mac*), 1776
 argparse, 567
 array, 236
 ast, 1626
 asynchat, 926
 asyncore, 921
 atexit, 1560
 audioop, 1259
 autoGIL (*Mac*), 1735

b

base64, 1025
 BaseHTTPServer, 1219
 Bastion, 1598
 bdb, 1489
 binascii, 1029
 binhex, 1028
 bisect, 233
 bsddb, 412
 buildtools (*Mac*), 1776
 bz2, 447

C

calendar, 205
 Carbon.AE (*Mac*), 1736
 Carbon.AH (*Mac*), 1736
 Carbon.App (*Mac*), 1736
 Carbon.Appearance (*Mac*), 1736
 Carbon.CarbonEvents (*Mac*), 1738
 Carbon.CarbonEvt (*Mac*), 1738
 Carbon.CF (*Mac*), 1736
 Carbon.CG (*Mac*), 1738
 Carbon.Cm (*Mac*), 1738
 Carbon.Components (*Mac*), 1738
 Carbon.ControlAccessor (*Mac*), 1738
 Carbon.Controls (*Mac*), 1738
 Carbon.CoreFoundation (*Mac*), 1738
 Carbon.CoreGraphics (*Mac*), 1738
 Carbon.Ctl (*Mac*), 1738
 Carbon.Dialogs (*Mac*), 1738
 Carbon.Dlg (*Mac*), 1738
 Carbon.Drag (*Mac*), 1738
 Carbon.Dragconst (*Mac*), 1738
 Carbon.Events (*Mac*), 1738
 Carbon.Evt (*Mac*), 1738

Carbon.File (*Mac*), 1738
 Carbon.Files (*Mac*), 1738
 Carbon.Fm (*Mac*), 1738
 Carbon.Folder (*Mac*), 1738
 Carbon.Folders (*Mac*), 1738
 Carbon.Fonts (*Mac*), 1738
 Carbon.Help (*Mac*), 1738
 Carbon.IBCarbon (*Mac*), 1738
 Carbon.IBCarbonRuntime (*Mac*), 1738
 Carbon.Icns (*Mac*), 1738
 Carbon.Icons (*Mac*), 1738
 Carbon.Launch (*Mac*), 1738
 Carbon.LaunchServices (*Mac*), 1738
 Carbon.List (*Mac*), 1738
 Carbon.Lists (*Mac*), 1738
 Carbon.MacHelp (*Mac*), 1738
 Carbon.MediaDescr (*Mac*), 1738
 Carbon.Menu (*Mac*), 1738
 Carbon.Menus (*Mac*), 1738
 Carbon.Mlte (*Mac*), 1738
 Carbon.OSA (*Mac*), 1738
 Carbon.OSAconst (*Mac*), 1738
 Carbon.Qd (*Mac*), 1738
 Carbon.Qdoffs (*Mac*), 1738
 Carbon.QDOffscreen (*Mac*), 1738
 Carbon.Qt (*Mac*), 1738
 Carbon.QuickDraw (*Mac*), 1738
 Carbon.QuickTime (*Mac*), 1738
 Carbon.Res (*Mac*), 1738
 Carbon.Resources (*Mac*), 1738
 Carbon.Scrap (*Mac*), 1738
 Carbon.Snd (*Mac*), 1739
 Carbon.Sound (*Mac*), 1739
 Carbon.TE (*Mac*), 1739
 Carbon.TextEdit (*Mac*), 1739
 Carbon.Win (*Mac*), 1739
 Carbon.Windows (*Mac*), 1739
 cd (*IRIX*), 1752
 cfmfile (*Mac*), 1776
 cgi, 1116
 CGIHTTPServer, 1225
 cgitb, 1125
 chunk, 1274
 cmath, 280
 cmd, 1307
 code, 1587
 codecs, 150
 codeop, 1589
 collections, 210
 ColorPicker (*Mac*), 1739
 colorsys, 1275
 commands (*Unix*), 1719
 compileall, 1643
 compiler, 1659
 compiler.ast, 1661
 compiler.visitor, 1667
 ConfigParser, 481
 contextlib, 1555
 Cookie, 1238
 cookielib, 1227
 copy, 263
 copy-reg, 399

cPickle, 399
 cProfile, 1504
 crypt (*Unix*), 1701
 cStringIO, 146
 csv, 471
 ctypes, 716
 curses (*Unix*), 676
 curses.ascii, 700
 curses.panel, 703
 curses.textpad, 698

d

datetime, 175
 dbhash, 410
 dbm (*Unix*), 408
 decimal, 284
 DEVICE (*IRIX*), 1766
 difflib, 132
 dircache, 382
 dis, 1646
 distutils, 1523
 dl (*Unix*), 1702
 doctest, 1410
 DocXMLRPCServer, 1257
 dumbdbm, 415
 dummy_thread, 779
 dummy_threading, 779

e

EasyDialogs (*Mac*), 1727
 email, 931
 email.charset, 952
 email.encoders, 956
 email.errors, 957
 email.generator, 944
 email.header, 950
 email.iterators, 961
 email.message, 932
 email.mime, 946
 email.parser, 941
 email.utils, 958
 encodings.idna, 169
 encodings.utf-8.sig, 169
 ensurepip, 1524
 errno, 708
 exceptions, 81

f

fcntl (*Unix*), 1706
 filecmp, 367
 fileinput, 358
 findertools (*Mac*), 1726
 fl (*IRIX*), 1756
 FL (*IRIX*), 1762
 flp (*IRIX*), 1762
 fm (*IRIX*), 1763
 fnmatch, 374
 formatter, 1669
 fpectl (*Unix*), 1583
 fpformat, 174
 fractions, 315
 Framework (*Mac*), 1730
 ftplib, 1169
 functools, 339
 future.builtins, 1547

g

gc, 1568
 gdbm (*Unix*), 409
 gensuitemodule (*Mac*), 1742
 getopt, 634

getpass, 676
 gettext, 1285
 gl (*IRIX*), 1764
 GL (*IRIX*), 1766
 glob, 373
 grp (*Unix*), 1700
 gzip, 445

h

hashlib, 499
 heapq, 228
 hmac, 502
 hotshot, 1511
 hotshot.stats, 1512
 htmlentitydefs, 1047
 htmllib, 1045
 HTMLParser, 1035
 httplib, 1161

i

ic (*Mac*), 1721
 icopen (*Mac*), 1776
 imageop, 1263
 imaplib, 1178
 imgfile (*IRIX*), 1767
 imghdr, 1276
 imp, 1601
 importlib, 1606
 imputil, 1606
 inspect, 1571
 io, 545
 itertools, 323

j

jpeg (*IRIX*), 1768
 json, 972

k

keyword, 1638

l

lib2to3, 1479
 linecache, 376
 locale, 1297
 logging, 637
 logging.config, 653
 logging.handlers, 665

m

macerrors (*Mac*), 1776
 MacOS (*Mac*), 1723
 macostools (*Mac*), 1725
 macpath, 383
 macresource (*Mac*), 1776
 mailbox, 984
 mailcap, 982
 marshal, 403
 math, 275
 md5, 503
 mhtml, 1008
 mimetools, 1010
 mimetypes, 1012
 MimeTypeWriter, 1015
 mimic, 1016
 MiniAEFrame (*Mac*), 1748
 mmap, 842
 modulefinder, 1615
 msilib (*Windows*), 1675
 msvcrt (*Windows*), 1682

multifile, 1018
 multiprocessing, 780
 multiprocessing.connection, 812
 multiprocessing.dummy, 816
 multiprocessing.managers, 802
 multiprocessing.pool, 809
 multiprocessing.sharedctypes, 799
 mutex, 246

n

Nav (*Mac*), 1777
 netrc, 490
 new, 262
 nis (*Unix*), 1717
 nntplib, 1185
 numbers, 271

o

operator, 342
 optparse, 601
 os, 507
 os.path, 353
 ossaudiodev (*Linux*, *FreeBSD*), 1278

p

parser, 1621
 pdb, 1495
 pickle, 385
 pickletools, 1657
 pipes (*Unix*), 1708
 PixMapWrapper (*Mac*), 1777
 pkgutil, 1612
 platform, 704
 plistlib, 495
 popen2, 918
 poplib, 1175
 posix (*Unix*), 1697
 posixfile (*Unix*), 1711
 pprint, 264
 profile, 1504
 pstats, 1506
 pty (*Linux*), 1705
 pwd (*Unix*), 1698
 py_compile, 1643
 pycldr, 1641
 pydoc, 1409

q

Queue, 247
 quopri, 1031

r

random, 317
 re, 106
 readline (*Unix*), 846
 resource (*Unix*), 1713
 rexec, 1594
 rfc822, 1020
 rlcompleter, 851
 robotparser, 489
 runpy, 1617

s

sched, 244
 ScrolledText (*Tk*), 1358
 select, 757
 sets, 240
 sgmlib, 1041
 sha, 505
 shelve, 400
 shlex, 1310

shutil, 376
 signal, 914
 SimpleHTTPServer, 1223
 SimpleXMLRPCServer, 1252
 site, 1579
 smtpd, 1196
 smtplib, 1190
 sndhdr, 1277
 socket, 868
 SocketServer, 1210
 spwd (*Unix*), 1699
 sqlite3, 416
 ssl, 883
 stat, 361
 statvfs, 366
 string, 91
 StringIO, 145
 stringprep, 172
 struct, 126
 subprocess, 853
 sunau, 1267
 sunaudiodev (*SunOS*), 1771
 SUNAUDIODEV (*SunOS*), 1773
 symbol, 1635
 symtable, 1633
 sys, 1527
 sysconfig, 1542
 syslog (*Unix*), 1718

t

tabnanny, 1640
 tarfile, 458
 telnetlib, 1197
 tempfile, 369
 termios (*Unix*), 1704
 test, 1479
 test.support, 1482
 textwrap, 147
 thread, 777
 threading, 763
 time, 559
 timeit, 1513
 Tix, 1352
 Tkinter, 1315
 token, 1636
 tokenize, 1638
 trace, 1518
 traceback, 1562
 ttk, 1329
 tty (*Unix*), 1705
 turtle, 1359
 types, 258

u

unicodedata, 170
 unittest, 1441
 urllib, 1137
 urllib2, 1146
 urlparse, 1204
 user, 1582
 UserDict, 255
 UserList, 256
 UserString, 257
 uu, 1032
 uuid, 1200

v

videoreader (*Mac*), 1777

w

W (*Mac*), 1777

warnings, 1548
wave, 1271
weakref, 250
webbrowser, 1113
whichdb, 407
winsound (*Windows*), 1694
wsgiref, 1126
wsgiref.handlers, 1133
wsgiref.headers, 1129
wsgiref.simple_server, 1130
wsgiref.util, 1126
wsgiref.validate, 1132

X

xdrlib, 491
xml, 1047
xml.dom, 1065
xml.dom.minidom, 1079
xml.dom.pulldom, 1084
xml.etree.ElementTree, 1049
xml.parsers.expat, 1100
xml.sax, 1085
xml.sax.handler, 1087
xml.sax.saxutils, 1094
xml.sax.xmlreader, 1095
xmlrpclib, 1243

Z

zipfile, 450
zipimport, 1610
zlib, 441

索引

```

!=
    演算子, 38
*
    演算子, 39
**
    演算子, 39
+
    演算子, 39
-
    演算子, 39
--buffer
    unittest command line option, 1445
--catch
    unittest command line option, 1445
--clock
    timeit command line option, 1516
--count
    trace command line option, 1519
--coverdir=<dir>
    trace command line option, 1520
--failfast
    unittest command line option, 1445
--file=<file>
    trace command line option, 1519
--help
    timeit command line option, 1516
    trace command line option, 1519
--ignore-dir=<dir>
    trace command line option, 1520
--ignore-module=<mod>
    trace command line option, 1520
--listfuncs
    trace command line option, 1519
--missing
    trace command line option, 1520
--no-report
    trace command line option, 1520
--number=N
    timeit command line option, 1516
--pattern pattern
    unittest-discover command line option, 1445
--repeat=N
    timeit command line option, 1516
--report
    trace command line option, 1519
--setup=S
    timeit command line option, 1516
--start-directory directory
    unittest-discover command line option, 1445
--summary
    trace command line option, 1520
--time
    timeit command line option, 1516
--timing
    trace command line option, 1520
--top-level-directory directory
    unittest-discover command line option, 1446
--trace
    trace command line option, 1519
--trackcalls
    trace command line option, 1519
--user-base

```

```

    site command line option, 1581
--user-site
    site command line option, 1582
--verbose
    timeit command line option, 1516
    unittest-discover command line option, 1445
--version
    trace command line option, 1519
-b
    unittest command line option, 1445
-C
    trace command line option, 1520
-c
    timeit command line option, 1516
    trace command line option, 1519
    unittest command line option, 1445
-c <zipfile> <source1> ... <sourceN>
    zipfile command line option, 458
-d destdir
    compileall command line option, 1644
-e <zipfile> <output_dir>
    zipfile command line option, 458
-f
    compileall command line option, 1644
    trace command line option, 1519
    unittest command line option, 1445
-g
    trace command line option, 1520
-h
    timeit command line option, 1516
-i list
    compileall command line option, 1644
-l
    compileall command line option, 1644
    trace command line option, 1519
-l <zipfile>
    zipfile command line option, 458
-m
    trace command line option, 1520
-n N
    timeit command line option, 1516
-p
    unittest-discover command line option, 1445
-q
    compileall command line option, 1644
-R
    trace command line option, 1520
-r
    trace command line option, 1519
-r N
    timeit command line option, 1516
-s
    trace command line option, 1520
    unittest-discover command line option, 1445
-s S
    timeit command line option, 1516
-T
    trace command line option, 1519
-t
    timeit command line option, 1516
    trace command line option, 1519
    unittest-discover command line option, 1446

```



```

-t <zipfile>
    zipfile command line option, 458
-v
    timeit command line option, 1516
    unittest-discover command line option, 1445
-x regex
    compileall command line option, 1644
..., 1779
.ini
    file, 481
.pdbrc
    file, 1498
.pythonrc.py
    file, 1582
/
    演算子, 39
//
    演算子, 39
==
    演算子, 38
%
    演算子, 39
% formatting, 55
% interpolation, 55
&
    演算子, 41
__abs__() (operator モジュール), 343
__add__() (operator モジュール), 343
__add__() (rfc822.AddressList のメソッド), 1025
__and__() (operator モジュール), 343
__bases__ (class の属性), 80
__builtin__ (モジュール), 1546
__class__ (instance の属性), 80
__cmp__() (instance method), 39
__concat__() (operator モジュール), 344
__contains__() (email.message.Message のメソッド), 935
__contains__() (mailbox.Mailbox のメソッド), 986
__contains__() (operator モジュール), 345
__copy__() (copy protocol), 264
__debug__ (組み込み変数), 35
__deepcopy__() (copy protocol), 264
__del__() (io.IOBase のメソッド), 550
__delitem__() (email.message.Message のメソッド), 935
__delitem__() (mailbox.Mailbox のメソッド), 985
__delitem__() (mailbox.MH のメソッド), 992
__delitem__() (operator モジュール), 345
__delslice__() (operator モジュール), 345
__dict__ (object の属性), 80
__displayhook__ (sys モジュール), 1529
__div__() (operator モジュール), 343
__enter__() (_winreg.PyHKEY のメソッド), 1694
__enter__() (contextmanager のメソッド), 76
__eq__() (email.charset.Charset のメソッド), 955
__eq__() (email.header.Header のメソッド), 951
__eq__() (instance method), 39
__eq__() (operator モジュール), 342
__excepthook__ (sys モジュール), 1529
__exit__() (_winreg.PyHKEY のメソッド), 1694
__exit__() (contextmanager のメソッド), 76
__floordiv__() (operator モジュール), 343
__format__, 13
__format__() (datetime.date のメソッド), 183
__format__() (datetime.datetime のメソッド), 191
__format__() (datetime.time のメソッド), 194
__future__, 1782
__future__ (モジュール), 1566
__ge__() (instance method), 39
__ge__() (operator モジュール), 342
__getinitargs__() (object のメソッド), 392
__getitem__() (email.message.Message のメソッド), 935
__getitem__() (mailbox.Mailbox のメソッド), 986
__getitem__() (operator モジュール), 345
__getnewargs__() (object のメソッド), 392
__getslice__() (operator モジュール), 345
__getstate__() (object のメソッド), 392
__gt__() (instance method), 39
__gt__() (operator モジュール), 342
__iadd__() (operator モジュール), 346
__iadd__() (rfc822.AddressList のメソッド), 1025
__iand__() (operator モジュール), 346
__iconcat__() (operator モジュール), 346
__idiv__() (operator モジュール), 346
__ifloordiv__() (operator モジュール), 346
__ilshift__() (operator モジュール), 346
__imod__() (operator モジュール), 347
__import__() (組み込み関数), 30
__imul__() (operator モジュール), 347
__index__() (operator モジュール), 343
__init__() (logging.Handler のメソッド), 642
__inv__() (operator モジュール), 343
__invert__() (operator モジュール), 343
__ior__() (operator モジュール), 347
__ipow__() (operator モジュール), 347
__irepeat__() (operator モジュール), 347
__irshift__() (operator モジュール), 347
__isub__() (operator モジュール), 347
__isub__() (rfc822.AddressList のメソッド), 1025
__iter__() (container のメソッド), 44
__iter__() (iterator のメソッド), 44
__iter__() (mailbox.Mailbox のメソッド), 985
__iter__() (unittest.TestSuite のメソッド), 1463
__itruediv__() (operator モジュール), 347
__ixor__() (operator モジュール), 348
__le__() (instance method), 39
__le__() (operator モジュール), 342
__len__() (email.message.Message のメソッド), 934
__len__() (mailbox.Mailbox のメソッド), 987
__len__() (rfc822.AddressList のメソッド), 1025
__lshift__() (operator モジュール), 343
__lt__() (instance method), 39
__lt__() (operator モジュール), 342
__main__
    モジュール, 1617, 1618
__main__ (モジュール), 1548
__members__ (object の属性), 80
__methods__ (object の属性), 80
__missing__(), 64
__missing__() (collections.defaultdict のメソッド), 217
__mod__() (operator モジュール), 344
__mro__ (class の属性), 80
__mul__() (operator モジュール), 344
__name__ (definition の属性), 80
__ne__() (email.charset.Charset のメソッド), 955
__ne__() (email.header.Header のメソッド), 952
__ne__() (instance method), 39
__ne__() (operator モジュール), 342
__neg__() (operator モジュール), 344
__not__() (operator モジュール), 342
__or__() (operator モジュール), 344
__pos__() (operator モジュール), 344
__pow__() (operator モジュール), 344
__reduce__() (object のメソッド), 393
__reduce_ex__() (object のメソッド), 394
__repeat__() (operator モジュール), 345
__repr__() (multiprocessing.managers.BaseProxy のメソッド),
    809
__repr__() (netrc.netrc のメソッド), 491
__rshift__() (operator モジュール), 344
__setitem__() (email.message.Message のメソッド), 935
__setitem__() (mailbox.Mailbox のメソッド), 985
__setitem__() (mailbox.Maildir のメソッド), 989
__setitem__() (operator モジュール), 345
__setslice__() (operator モジュール), 345
__setstate__() (object のメソッド), 392
__slots__, 1788
__stderr__ (sys モジュール), 1541

```

```

__stdin__ (sys モジュール), 1541
__stdout__ (sys モジュール), 1541
__str__ () (datetime.date のメソッド), 183
__str__ () (datetime.datetime のメソッド), 190
__str__ () (datetime.time のメソッド), 194
__str__ () (email.charset.Charset のメソッド), 955
__str__ () (email.header.Header のメソッド), 951
__str__ () (email.message.Message のメソッド), 933
__str__ () (multiprocessing.managers.BaseProxy のメソッド),
    809
__str__ () (rfc822.AddressList のメソッド), 1025
__sub__ () (operator モジュール), 344
__sub__ () (rfc822.AddressList のメソッド), 1025
__subclasses__ () (class のメソッド), 80
__subclasshook__ () (abc.ABCMeta のメソッド), 1558
__truediv__ () (operator モジュール), 344
__unicode__ () (email.header.Header のメソッド), 951
__xor__ () (operator モジュール), 344
.anonymous_ (ctypes.Structure の属性), 755
.asdict () (collections.somenamedtuple のメソッド), 221
.b_base_ (ctypes._CData の属性), 750
.b_needsfree_ (ctypes._CData の属性), 750
.callmethod () (multiprocessing.managers.BaseProxy のメソッ
    ド), 808
._CData (ctypes のクラス), 749
._clear_type_cache () (sys モジュール), 1528
._current_frames () (sys モジュール), 1528
._exit () (os モジュール), 533
._fields (ast.AST の属性), 1627
._fields (collections.somenamedtuple の属性), 222
._fields_ (ctypes.Structure の属性), 754
._flush () (wsgiref.handlers.BaseHandler のメソッド), 1134
._FuncPtr (ctypes のクラス), 742
._getframe () (sys モジュール), 1534
._getvalue () (multiprocessing.managers.BaseProxy のメソッ
    ド), 809
._handle (ctypes.PyDLL の属性), 741
._https_verify_certificates () (ssl モジュール), 888
._length_ (ctypes.Array の属性), 756
._locale
    モジュール, 1297
._make () (collections.somenamedtuple のクラスメソッド), 221
._makeResult () (unittest.TextTestRunner のメソッド), 1468
._name (ctypes.PyDLL の属性), 741
._objects (ctypes._CData の属性), 750
._pack_ (ctypes.Structure の属性), 755
._parse () (gettext.NullTranslations のメソッド), 1289
._Pointer (ctypes のクラス), 756
._quit () (FrameWork.Application のメソッド), 1732
._replace () (collections.somenamedtuple のメソッド), 222
._setroot () (xml.etree.ElementTree.ElementTree のメソッド),
    1061
._SimpleCData (ctypes のクラス), 751
._start () (aetools.TalkTo のメソッド), 1744
._structure () (email.iterators モジュール), 961
._type_ (ctypes._Pointer の属性), 756
._type_ (ctypes.Array の属性), 756
._url opener (urllib モジュール), 1140
._winreg (モジュール), 1684
._write () (wsgiref.handlers.BaseHandler のメソッド), 1134
^
    演算子, 41
~
    演算子, 41
|
    演算子, 41
>>>, 1779
>
    演算子, 38
>=
    演算子, 38
>>
    演算子, 41

```

```

<
    演算子, 38
<=
    演算子, 38
<<
    演算子, 41
<protocol>_proxy, 1149
2to3, 1779

A-LAW, 1267, 1277
a-LAW, 1259
a2b_base64 () (binascii モジュール), 1029
a2b_hex () (binascii モジュール), 1031
a2b_hqx () (binascii モジュール), 1030
a2b_qp () (binascii モジュール), 1029
a2b_uu () (binascii モジュール), 1029
abc (モジュール), 1557
ABCMeta (abc のクラス), 1557
abort () (ftplib.FTP のメソッド), 1172
abort () (os モジュール), 533
above () (curses.panel.Panel のメソッド), 703
abs () (decimal.Context のメソッド), 301
abs () (operator モジュール), 343
abs () (組み込み関数), 5
abspath () (os.path モジュール), 353
abstract base class, 1779
AbstractBasicAuthHandler (urllib2 のクラス), 1150
AbstractDigestAuthHandler (urllib2 のクラス), 1150
AbstractFormatter (formatter のクラス), 1672
abstractmethod () (abc モジュール), 1559
abstractproperty () (abc モジュール), 1560
AbstractWriter (formatter のクラス), 1673
accept () (asyncore.dispatcher のメソッド), 924
accept () (multiprocessing.connection.Listener のメソッド), 813
accept () (socket.socket のメソッド), 876
accept2dayear (time モジュール), 561
access () (os モジュール), 520
acos () (cmath モジュール), 282
acos () (math モジュール), 278
acosh () (cmath モジュール), 282
acosh () (math モジュール), 278
acquire () (logging.Handler のメソッド), 642
acquire () (multiprocessing.Lock のメソッド), 796
acquire () (multiprocessing.RLock のメソッド), 797
acquire () (thread.lock のメソッド), 778
acquire () (threading.Condition のメソッド), 772
acquire () (threading.Lock のメソッド), 770
acquire () (threading.RLock のメソッド), 771
acquire () (threading.Semaphore のメソッド), 774
acquire_lock () (imp モジュール), 1602
Action (argparse のクラス), 588
action (optparse.Option の属性), 617
ACTIONS (optparse.Option の属性), 633
activate_form () (fl.form のメソッド), 1759
active_children () (multiprocessing モジュール), 792
active_count () (threading モジュール), 764
activeCount () (threading モジュール), 764
add () (audioop モジュール), 1259
add () (decimal.Context のメソッド), 302
add () (frozenset のメソッド), 63
add () (mailbox.Mailbox のメソッド), 985
add () (mailbox.Maildir のメソッド), 989
add () (msilib.RadioButtonGroup のメソッド), 1681
add () (operator モジュール), 343
add () (pstats.Stats のメソッド), 1506
add () (tarfile.TarFile のメソッド), 464
add () (tk.Notebook のメソッド), 1337
add_alias () (email.charset モジュール), 956
add_argument () (argparse.ArgumentParser のメソッド), 577
add_argument_group () (argparse.ArgumentParser のメソッ
    ド), 597
add_box () (fl.form のメソッド), 1759
add_browser () (fl.form のメソッド), 1760

```

add.button() (*fl.form* のメソッド), 1759
 add.cgi_vars() (*wsgiref.handlers.BaseHandler* のメソッド), 1134
 add.charset() (*email.charset* モジュール), 955
 add.choice() (*fl.form* のメソッド), 1760
 add.clock() (*fl.form* のメソッド), 1759
 add.codec() (*email.charset* モジュール), 956
 add.cookie_header() (*cookielib.CookieJar* のメソッド), 1229
 add.counter() (*fl.form* のメソッド), 1760
 add.data() (*msilib* モジュール), 1676
 add.data() (*urllib2.Request* のメソッド), 1151
 add.dial() (*fl.form* のメソッド), 1760
 add.fallback() (*gettext.NullTranslations* のメソッド), 1289
 add.file() (*msilib.Directory* のメソッド), 1680
 add.flag() (*mailbox.MaildirMessage* のメソッド), 995
 add.flag() (*mailbox.mboxMessage* のメソッド), 998
 add.flag() (*mailbox.MMDfMessage* のメソッド), 1003
 add.floating.data() (*formatter.formatter* のメソッド), 1670
 add.folder() (*mailbox.Maildir* のメソッド), 989
 add.folder() (*mailbox.MH* のメソッド), 991
 add.handler() (*urllib2.OpenerDirector* のメソッド), 1152
 add.header() (*email.message.Message* のメソッド), 936
 add.header() (*urllib2.Request* のメソッド), 1151
 add.header() (*wsgiref.headers.Headers* のメソッド), 1129
 add.history() (*readline* モジュール), 848
 add.hor_rule() (*formatter.formatter* のメソッド), 1670
 add.input() (*fl.form* のメソッド), 1760
 add.label() (*mailbox.BabylMessage* のメソッド), 1001
 add.label.data() (*formatter.formatter* のメソッド), 1670
 add.lightbutton() (*fl.form* のメソッド), 1759
 add.line_break() (*formatter.formatter* のメソッド), 1670
 add.literal.data() (*formatter.formatter* のメソッド), 1670
 add.menu() (*fl.form* のメソッド), 1760
 add.mutually_exclusive_group() (*argparse.ArgumentParser* のメソッド), 597
 add.option() (*optparse.OptionParser* のメソッド), 616
 add.parent() (*urllib2.BaseHandler* のメソッド), 1154
 add.password() (*urllib2.HTTPPasswordMgr* のメソッド), 1157
 add.positioner() (*fl.form* のメソッド), 1760
 add.roundbutton() (*fl.form* のメソッド), 1759
 add.section() (*ConfigParser.RawConfigParser* のメソッド), 484
 add.sequence() (*mailbox.MHMessage* のメソッド), 999
 add.slider() (*fl.form* のメソッド), 1759
 add.stream() (*msilib* モジュール), 1676
 add.subparsers() (*argparse.ArgumentParser* のメソッド), 593
 add.suffix() (*imputil.ImportManager* のメソッド), 1606
 add.tables() (*msilib* モジュール), 1676
 add.text() (*fl.form* のメソッド), 1759
 add.timer() (*fl.form* のメソッド), 1760
 add.type() (*mimetypes* モジュール), 1013
 add.unredirected_header() (*urllib2.Request* のメソッド), 1151
 add.valslider() (*fl.form* のメソッド), 1759
 addch() (*curses.window* のメソッド), 685
 addCleanup() (*unittest.TestCase* のメソッド), 1461
 addComponent() (*turtle.Shape* のメソッド), 1391
 addError() (*unittest.TestResult* のメソッド), 1467
 addExpectedFailure() (*unittest.TestResult* のメソッド), 1468
 addFailure() (*unittest.TestResult* のメソッド), 1467
 addfile() (*tarfile.TarFile* のメソッド), 464
 addFilter() (*logging.Handler* のメソッド), 642
 addFilter() (*logging.Logger* のメソッド), 641
 addHandler() (*logging.Logger* のメソッド), 641
 addheader() (*MimeWriter.MimeWriter* のメソッド), 1016
 addinfo() (*hotshot.Profile* のメソッド), 1512
 addLevelName() (*logging* モジュール), 651
 addnstr() (*curses.window* のメソッド), 685
 AddPackagePath() (*modulefinder* モジュール), 1616
 address (*multiprocessing.connection.Listener* の属性), 813

address (*multiprocessing.managers.BaseManager* の属性), 804
 address_family (*SocketServer.BaseServer* の属性), 1213
 address_string() (*BaseHTTPServer.BaseHTTPRequestHandler* のメソッド), 1223
 AddressList (*rfc822* のクラス), 1021
 addresslist (*rfc822.AddressList* の属性), 1025
 addressof() (*ctypes* モジュール), 746
 addshape() (*turtle* モジュール), 1388
 addsitedir() (*site* モジュール), 1581
 addSkip() (*unittest.TestResult* のメソッド), 1467
 addstr() (*curses.window* のメソッド), 685
 addSuccess() (*unittest.TestResult* のメソッド), 1467
 addTest() (*unittest.TestSuite* のメソッド), 1462
 addTests() (*unittest.TestSuite* のメソッド), 1462
 addTypeEqualityFunc() (*unittest.TestCase* のメソッド), 1459
 addUnexpectedSuccess() (*unittest.TestResult* のメソッド), 1468
 adjusted() (*decimal.Decimal* のメソッド), 290
 Adler32() (*zlib* モジュール), 441
 ADPCM, Intel/DVI, 1259
 adpcm2lin() (*audioop* モジュール), 1259
 aepack (モジュール), 1744
 AEServer (*MiniAEFrame* のクラス), 1748
 AEText (*aetypes* のクラス), 1746
 aetools (モジュール), 1743
 aetypes (モジュール), 1746
 AF_INET (*socket* モジュール), 870
 AF_INET6 (*socket* モジュール), 870
 AF_UNIX (*socket* モジュール), 870
 aifc (モジュール), 1264
 aifc() (*aifc.aifc* のメソッド), 1266
 AIFF, 1264, 1274
 aiff() (*aifc.aifc* のメソッド), 1266
 AIFF-C, 1264, 1274
 AL
 モジュール, 1749
 AL (モジュール), 1751
 al (モジュール), 1749
 alarm() (*signal* モジュール), 916
 alaw2lin() (*audioop* モジュール), 1259
 ALERT_DESCRIPTION_HANDSHAKE_FAILURE (*ssl* モジュール), 897
 ALERT_DESCRIPTION_INTERNAL_ERROR (*ssl* モジュール), 897
 algorithms_available (*hashlib* モジュール), 500
 algorithms_guaranteed (*hashlib* モジュール), 500
 alignment() (*ctypes* モジュール), 746
 all() (組み込み関数), 5
 all_errors (*ftplib* モジュール), 1171
 all_features (*xml.sax.handler* モジュール), 1089
 all_properties (*xml.sax.handler* モジュール), 1090
 allocate_lock() (*thread* モジュール), 777
 allow_reuse_address (*SocketServer.BaseServer* の属性), 1213
 allowed_domains() (*cookielib.DefaultCookiePolicy* のメソッド), 1234
 alt() (*curses.ascii* モジュール), 702
 ALT_DIGITS (*locale* モジュール), 1301
 altsep (*os* モジュール), 544
 altzone (*time* モジュール), 561
 ALWAYS_TYPED_ACTIONS (*optparse.Option* の属性), 633
 AMPER (*token* モジュール), 1636
 AMPEREQUAL (*token* モジュール), 1636
 anchor_bgn() (*htmlib.HTMLParser* のメソッド), 1046
 anchor_end() (*htmlib.HTMLParser* のメソッド), 1046
 and
 演算子, 38
 and_() (*operator* モジュール), 343
 annotate() (*dircache* モジュール), 382
 answer_challenge() (*multiprocessing.connection* モジュール), 812

any() (組み込み関数), 6
 anydbm (モジュール), 405
 api.version (sys モジュール), 1542
 apop() (poplib.POP3 のメソッド), 1176
 append() (array.array のメソッド), 237
 append() (collections.deque のメソッド), 214
 append() (email.header.Header のメソッド), 951
 append() (imaplib.IMAP4 のメソッド), 1180
 append() (list method), 58
 append() (msilib.CAB のメソッド), 1679
 append() (pipes.Template のメソッド), 1710
 append() (xml.etree.ElementTree.Element のメソッド), 1060
 appendChild() (xml.dom.Node のメソッド), 1070
 appendleft() (collections.deque のメソッド), 214
 AppleEvents, 1726, 1748
 applesingle (モジュール), 1776
 Application() (FrameWork モジュール), 1730
 application.uri() (wsgiref.util モジュール), 1127
 apply (2to3 fixer), 1475
 apply() (multiprocessing.pool.multiprocessing.Pool のメソッド), 810
 apply() (組み込み関数), 33
 apply_async() (multiprocessing.pool.multiprocessing.Pool のメソッド), 810
 architecture() (platform モジュール), 705
 archive (zipimport.zipimporter の属性), 1612
 aRepr (repr モジュール), 268
 argparse (モジュール), 567
 args (exceptions.BaseException の属性), 81
 args (functools.partial の属性), 341
 argtypes (ctypes.FuncPtr の属性), 742
 argument, 1779
 ArgumentDefaultsHelpFormatter (argparse のクラス), 573
 ArgumentError, 743
 ArgumentParser (argparse のクラス), 569
 argv (sys モジュール), 1527
 arithmetic, 39
 ArithmeticError, 82
 array (array のクラス), 236
 Array (ctypes のクラス), 756
 array (モジュール), 236
 Array() (multiprocessing モジュール), 799
 Array() (multiprocessing.managers.SyncManager のメソッド), 804
 Array() (multiprocessing.sharedctypes モジュール), 800
 arrays, 236
 ArrayType (array モジュール), 237
 article() (nntplib.NNTP のメソッド), 1189
 as.integer_ratio() (float のメソッド), 42
 AS_IS (formatter モジュール), 1669
 as.string() (email.message.Message のメソッド), 932
 as.tuple() (decimal.Decimal のメソッド), 290
 ascii() (curses.ascii モジュール), 702
 ascii() (future.builtins モジュール), 1547
 ascii_letters (string モジュール), 91
 ascii_lowercase (string モジュール), 91
 ascii_uppercase (string モジュール), 91
 asctime() (time モジュール), 561
 asin() (cmath モジュール), 282
 asin() (math モジュール), 278
 asinh() (cmath モジュール), 282
 asinh() (math モジュール), 278
 AskFileForOpen() (EasyDialogs モジュール), 1728
 AskFileForSave() (EasyDialogs モジュール), 1728
 AskFolder() (EasyDialogs モジュール), 1729
 AskPassword() (EasyDialogs モジュール), 1727
 AskString() (EasyDialogs モジュール), 1727
 AskYesNoCancel() (EasyDialogs モジュール), 1727
 assert
 文, 82
 assert.line_data() (formatter.formatter のメソッド), 1671
 assertAlmostEqual() (unittest.TestCase のメソッド), 1457
 assertDictContainsSubset() (unittest.TestCase のメソッド), 1458
 assertDictEqual() (unittest.TestCase のメソッド), 1460
 assertEqual() (unittest.TestCase のメソッド), 1455
 assertFalse() (unittest.TestCase のメソッド), 1455
 assertGreater() (unittest.TestCase のメソッド), 1458
 assertGreaterEqual() (unittest.TestCase のメソッド), 1458
 assertIn() (unittest.TestCase のメソッド), 1455
 AssertionError, 82
 assertIs() (unittest.TestCase のメソッド), 1455
 assertIsInstance() (unittest.TestCase のメソッド), 1455
 assertIsNone() (unittest.TestCase のメソッド), 1455
 assertIsNot() (unittest.TestCase のメソッド), 1455
 assertIsNotNone() (unittest.TestCase のメソッド), 1455
 assertItemsEqual() (unittest.TestCase のメソッド), 1458
 assertLess() (unittest.TestCase のメソッド), 1458
 assertLessEqual() (unittest.TestCase のメソッド), 1458
 assertListEqual() (unittest.TestCase のメソッド), 1459
 assertMultiLineEqual() (unittest.TestCase のメソッド), 1459
 assertNotAlmostEqual() (unittest.TestCase のメソッド), 1457
 assertNotEqual() (unittest.TestCase のメソッド), 1455
 assertNotIn() (unittest.TestCase のメソッド), 1455
 assertNotIsInstance() (unittest.TestCase のメソッド), 1455
 assertNotRegexpMatches() (unittest.TestCase のメソッド), 1458
 assertRaises() (unittest.TestCase のメソッド), 1456
 assertRaisesRegexp() (unittest.TestCase のメソッド), 1456
 assertRegexpMatches() (unittest.TestCase のメソッド), 1458
 asserts (2to3 fixer), 1475
 assertSequenceEqual() (unittest.TestCase のメソッド), 1459
 assertSetEqual() (unittest.TestCase のメソッド), 1460
 assertTrue() (unittest.TestCase のメソッド), 1455
 assertTupleEqual() (unittest.TestCase のメソッド), 1459
 assignment
 extended slice, 58
 slice, 58
 subscript, 58
 AST (ast のクラス), 1627
 ast (モジュール), 1626
 astimezone() (datetime.datetime のメソッド), 188
 ASTVisitor (compiler.visitor のクラス), 1667
 async_chat (asynchat のクラス), 926
 async_chat.ac.in.buffer.size (asynchat モジュール), 926
 async_chat.ac.out.buffer.size (asynchat モジュール), 926
 asyncevents() (FrameWork.Application のメソッド), 1732
 asynchat (モジュール), 926
 asyncore (モジュール), 921
 AsyncResult (multiprocessing.pool のクラス), 811
 AT (token モジュール), 1636
 atan() (cmath モジュール), 282
 atan() (math モジュール), 278
 atan2() (math モジュール), 278
 atanh() (cmath モジュール), 282
 atanh() (math モジュール), 279
 atexit (モジュール), 1560
 atime (cd モジュール), 1753
 atof() (locale モジュール), 1303
 atof() (string モジュール), 103
 atoi() (locale モジュール), 1303
 atoi() (string モジュール), 104
 atol() (string モジュール), 104
 attach() (email.message.Message のメソッド), 933
 AttlistDeclHandler() (xml.parsers.expat.xmlparser のメソッド), 1105
 attrgetter() (operator モジュール), 348
 attrib (xml.etree.ElementTree.Element の属性), 1059
 attribute, 1780

- AttributeError, 82
- attributes (*xml.dom.Node* の属性), 1069
- AttributesImpl (*xml.sax.xmlreader* のクラス), 1096
- AttributesNSImpl (*xml.sax.xmlreader* のクラス), 1096
- attroff () (*curses.window* のメソッド), 686
- attron () (*curses.window* のメソッド), 686
- attrset () (*curses.window* のメソッド), 686
- audio (*cd* モジュール), 1753
- Audio Interchange File Format, 1264, 1274
- AUDIO_FILE_ENCODING_ADPCM_G721 (*sunau* モジュール), 1269
- AUDIO_FILE_ENCODING_ADPCM_G722 (*sunau* モジュール), 1269
- AUDIO_FILE_ENCODING_ADPCM_G723_3 (*sunau* モジュール), 1269
- AUDIO_FILE_ENCODING_ADPCM_G723_5 (*sunau* モジュール), 1269
- AUDIO_FILE_ENCODING_ALAW_8 (*sunau* モジュール), 1268
- AUDIO_FILE_ENCODING_DOUBLE (*sunau* モジュール), 1269
- AUDIO_FILE_ENCODING_FLOAT (*sunau* モジュール), 1269
- AUDIO_FILE_ENCODING_LINEAR_16 (*sunau* モジュール), 1268
- AUDIO_FILE_ENCODING_LINEAR_24 (*sunau* モジュール), 1268
- AUDIO_FILE_ENCODING_LINEAR_32 (*sunau* モジュール), 1268
- AUDIO_FILE_ENCODING_LINEAR_8 (*sunau* モジュール), 1268
- AUDIO_FILE_ENCODING_MULAW_8 (*sunau* モジュール), 1268
- AUDIO_FILE_MAGIC (*sunau* モジュール), 1268
- AUDIODEV, 1278
- audioop (モジュール), 1259
- auth () (*ftplib.FTP.TLS* のメソッド), 1175
- authenticate () (*imaplib.IMAP4* のメソッド), 1180
- AuthenticationError, 813
- authenticators () (*netrc.netrc* のメソッド), 490
- authkey (*multiprocessing.Process* の属性), 787
- autoGIL (モジュール), 1735
- AutoGILError, 1735
- avg () (*audioop* モジュール), 1260
- avggp () (*audioop* モジュール), 1260
- b16decode () (*base64* モジュール), 1027
- b16encode () (*base64* モジュール), 1027
- b2a_base64 () (*binascii* モジュール), 1029
- b2a_hex () (*binascii* モジュール), 1031
- b2a_hqx () (*binascii* モジュール), 1030
- b2a_qp () (*binascii* モジュール), 1030
- b2a_uu () (*binascii* モジュール), 1029
- b32decode () (*base64* モジュール), 1026
- b32encode () (*base64* モジュール), 1026
- b64decode () (*base64* モジュール), 1026
- b64encode () (*base64* モジュール), 1026
- Babyl (*mailbox* のクラス), 992
- BabylMailbox (*mailbox* のクラス), 1006
- BabylMessage (*mailbox* のクラス), 1000
- back () (*turtle* モジュール), 1364
- BACKQUOTE (*token* モジュール), 1636
- backslashreplace.errors () (*codecs* モジュール), 153
- backward () (*turtle* モジュール), 1364
- backward.compatible (*imageop* モジュール), 1264
- BadStatusLine, 1164
- BadZipfile, 450
- Balloon (*Tix* のクラス), 1353
- base64
 - encoding, 1025
 - モジュール, 1029
- base64 (モジュール), 1025
- BaseCGIHandler (*wsgiref.handlers* のクラス), 1133
- BaseCookie (*Cookie* のクラス), 1239
- BaseException, 81
- BaseHandler (*urllib2* のクラス), 1149
- BaseHandler (*wsgiref.handlers* のクラス), 1133
- BaseHTTPRequestHandler (*BaseHTTPServer* のクラス), 1220
- BaseHTTPServer (モジュール), 1219
- BaseManager (*multiprocessing.managers* のクラス), 802
- basename () (*os.path* モジュール), 354
- BaseProxy (*multiprocessing.managers* のクラス), 808
- BaseRequestHandler (*SocketServer* のクラス), 1214
- BaseServer (*SocketServer* のクラス), 1212
- basestring (2to3 fixer), 1475
- basestring () (組み込み関数), 6
- basicConfig () (*logging* モジュール), 651
- BasicContext (*decimal* のクラス), 299
- Bastion (モジュール), 1598
- Bastion () (*Bastion* モジュール), 1598
- BastionClass (*Bastion* のクラス), 1599
- baudrate () (*curses* モジュール), 677
- bbox () (*ttk.Treeview* のメソッド), 1344
- bdb
 - モジュール, 1495
- Bdb (*bdb* のクラス), 1490
- bdb (モジュール), 1489
- BdbQuit, 1489
- BDFL, 1780
- beep () (*curses* モジュール), 678
- Beep () (*winsound* モジュール), 1694
- begin_fill () (*turtle* モジュール), 1375
- begin_poly () (*turtle* モジュール), 1381
- below () (*curses.panel.Panel* のメソッド), 704
- Benchmarking, 1513
- benchmarking, 561
- betavariate () (*random* モジュール), 321
- bgcolor () (*turtle* モジュール), 1383
- bgn.group () (*fl.form* のメソッド), 1759
- bgpic () (*turtle* モジュール), 1383
- bias () (*audioop* モジュール), 1260
- bidirectional () (*unicodedata* モジュール), 170
- BigEndianStructure (*ctypes* のクラス), 754
- bin () (組み込み関数), 6
- binary
 - data, packing, 126
- Binary (*msilib* のクラス), 1676
- Binary (*xmlrpclib* のクラス), 1247
- binary semaphores, 777
- BINARY_ADD (*opcode*), 1649
- BINARY_AND (*opcode*), 1649
- BINARY_DIVIDE (*opcode*), 1649
- BINARY_FLOOR_DIVIDE (*opcode*), 1649
- BINARY_LSHIFT (*opcode*), 1649
- BINARY_MODULO (*opcode*), 1649
- BINARY_MULTIPLY (*opcode*), 1648
- BINARY_OR (*opcode*), 1648
- BINARY_POWER (*opcode*), 1648
- BINARY_RSHIFT (*opcode*), 1649
- BINARY_SUBSCR (*opcode*), 1649
- BINARY_SUBTRACT (*opcode*), 1649
- BINARY_TRUE_DIVIDE (*opcode*), 1649
- BINARY_XOR (*opcode*), 1649
- binascii (モジュール), 1029
- bind (widgets), 1326
- bind () (*asyncore.dispatcher* のメソッド), 924
- bind () (*socket.socket* のメソッド), 876
- bind.textdomain.codeset () (*gettext* モジュール), 1286
- bindtextdomain () (*gettext* モジュール), 1285
- bindtextdomain () (*locale* モジュール), 1306
- binhex
 - モジュール, 1029
- binhex (モジュール), 1028
- binhex () (*binhex* モジュール), 1028
- bisect (モジュール), 233
- bisect () (*bisect* モジュール), 234
- bisect.left () (*bisect* モジュール), 233
- bisect.right () (*bisect* モジュール), 234
- bit.length () (*int* のメソッド), 42
- bit.length () (*long* のメソッド), 42
- bitmap () (*msilib.Dialog* のメソッド), 1681
- bitwise
 - operations, 41

bk () (*turtle* モジュール), 1364
 bkgd () (*curses.window* のメソッド), 686
 bkgdset () (*curses.window* のメソッド), 686
 blocked_domains () (*cookielib.DefaultCookiePolicy* のメソッド), 1234
 BlockingIOError, 548
 BLOCKSIZE (*cd* モジュール), 1752
 blocksize (*sha* モジュール), 505
 body () (*nntplib.NNTP* のメソッド), 1189
 body_encode () (*email.charset.Charset* のメソッド), 955
 body_encoding (*email.charset.Charset* の属性), 953
 body_line_iterator () (*email.iterators* モジュール), 961
 BOM (*codecs* モジュール), 155
 BOM_BE (*codecs* モジュール), 155
 BOM_LE (*codecs* モジュール), 155
 BOM_UTF16 (*codecs* モジュール), 155
 BOM_UTF16_BE (*codecs* モジュール), 155
 BOM_UTF16_LE (*codecs* モジュール), 155
 BOM_UTF32 (*codecs* モジュール), 155
 BOM_UTF32_BE (*codecs* モジュール), 155
 BOM_UTF32_LE (*codecs* モジュール), 155
 BOM_UTF8 (*codecs* モジュール), 155
 bool (組み込みクラス), 6
 Boolean
 operations, 37, 38
 type, 6
 values, 79
 オブジェクト, 39
 Boolean (*aetypes* のクラス), 1746
 boolean () (*xmlrpclib* モジュール), 1251
 BooleanType (*types* モジュール), 259
 bootstrap () (*ensurepip* モジュール), 1525
 border () (*curses.window* のメソッド), 686
 bottom () (*curses.panel.Panel* のメソッド), 704
 bottom_panel () (*curses.panel* モジュール), 703
 BoundaryError, 957
 BoundedSemaphore (*multiprocessing* のクラス), 795
 BoundedSemaphore ()
 (*multiprocessing.managers.SyncManager* のメソッド), 804
 BoundedSemaphore () (*threading* モジュール), 765
 box () (*curses.window* のメソッド), 687
 break_anywhere () (*bdb.Bdb* のメソッド), 1492
 break_here () (*bdb.Bdb* のメソッド), 1492
 break_long_words (*textwrap.TextWrapper* の属性), 149
 BREAK_LOOP (*opcode*), 1651
 break_on_hyphens (*textwrap.TextWrapper* の属性), 150
 Breakpoint (*bdb* のクラス), 1489
 breakpoints, 1400
 BROWSER, 1113, 1114
 bsddb
 モジュール, 401, 405, 410
 bsddb (モジュール), 412
 BsdDbShelf (*shelve* のクラス), 402
 btopen () (*bsddb* モジュール), 413
 buffer
 オブジェクト, 45
 組み込み関数, 261
 buffer (2to3 fixer), 1475
 buffer (*io.TextIOBase* の属性), 556
 buffer (*unittest.TestResult* の属性), 1466
 buffer size, I/O, 18
 buffer () (組み込み関数), 33
 buffer_info () (*array.array* のメソッド), 237
 buffer_size (*xml.parsers.expat.xmlparser* の属性), 1103
 buffer_text (*xml.parsers.expat.xmlparser* の属性), 1103
 buffer_used (*xml.parsers.expat.xmlparser* の属性), 1103
 BufferedIOBase (*io* のクラス), 551
 BufferedRandom (*io* のクラス), 555
 BufferedReader (*io* のクラス), 554
 BufferedRWPair (*io* のクラス), 555
 BufferedWriter (*io* のクラス), 554
 BufferError, 82

BufferingHandler (*logging.handlers* のクラス), 674
 BufferTooShort, 788, 813
 BufferType (*types* モジュール), 261
 BUFSIZ (*macostools* モジュール), 1726
 bufsize () (*ossaudiodev.oss.audio.device* のメソッド), 1281
 BUILD_CLASS (*opcode*), 1652
 BUILD_LIST (*opcode*), 1654
 BUILD_MAP (*opcode*), 1654
 build_opener () (*urllib2* モジュール), 1147
 BUILD_SET (*opcode*), 1654
 BUILD_SLICE (*opcode*), 1656
 BUILD_TUPLE (*opcode*), 1653
 buildtools (モジュール), 1776
 built-in
 types, 37
 builtin_module_names (*sys* モジュール), 1527
 BuiltinFunctionType (*types* モジュール), 260
 BuiltinImporter (*imputil* のクラス), 1607
 BuiltinMethodType (*types* モジュール), 260
 ButtonBox (*Tix* のクラス), 1353
 bye () (*turtle* モジュール), 1389
 byref () (*ctypes* モジュール), 746
 byte-code
 file, 1601, 1604, 1643
 bytearray
 オブジェクト, 45
 bytearray (組み込みクラス), 6
 bytecode, 1780
 byteorder (*sys* モジュール), 1527
 bytes (*uuid.UUID* の属性), 1201
 bytes-like object, 1780
 bytes_le (*uuid.UUID* の属性), 1201
 BytesIO (*io* のクラス), 553
 byteswap () (*array.array* のメソッド), 237
 BytesWarning, 88
 bz2 (モジュール), 447
 BZ2Compressor (*bz2* のクラス), 449
 BZ2Decompressor (*bz2* のクラス), 450
 BZ2File (*bz2* のクラス), 448
 C
 language, 39
 structures, 126
 c_bool (*ctypes* のクラス), 753
 C_BUILTIN (*imp* モジュール), 1603
 c_byte (*ctypes* のクラス), 751
 c_char (*ctypes* のクラス), 751
 c_char_p (*ctypes* のクラス), 751
 c_double (*ctypes* のクラス), 751
 C_EXTENSION (*imp* モジュール), 1603
 c_float (*ctypes* のクラス), 752
 c_int (*ctypes* のクラス), 752
 c_int16 (*ctypes* のクラス), 752
 c_int32 (*ctypes* のクラス), 752
 c_int64 (*ctypes* のクラス), 752
 c_int8 (*ctypes* のクラス), 752
 c_long (*ctypes* のクラス), 752
 c_longdouble (*ctypes* のクラス), 751
 c_longlong (*ctypes* のクラス), 752
 c_short (*ctypes* のクラス), 752
 c_size_t (*ctypes* のクラス), 752
 c_ssize_t (*ctypes* のクラス), 752
 c_ubyte (*ctypes* のクラス), 752
 c_uint (*ctypes* のクラス), 752
 c_uint16 (*ctypes* のクラス), 753
 c_uint32 (*ctypes* のクラス), 753
 c_uint64 (*ctypes* のクラス), 753
 c_uint8 (*ctypes* のクラス), 753
 c_ulong (*ctypes* のクラス), 753
 c_ulonglong (*ctypes* のクラス), 753
 c_ushort (*ctypes* のクラス), 753
 c_void_p (*ctypes* のクラス), 753
 c_wchar (*ctypes* のクラス), 753

c_wchar_p (*ctypes* のクラス), 753
CAB (*msilib* のクラス), 1679
CacheFTPHandler (*urllib2* のクラス), 1150
calcsize() (*struct* モジュール), 127
Calendar (*calendar* のクラス), 205
calendar (モジュール), 205
calendar() (*calendar* モジュール), 209
call() (*dl.dl* のメソッド), 1703
call() (*subprocess* モジュール), 854
CALL_FUNCTION (*opcode*), 1656
CALL_FUNCTION_KW (*opcode*), 1656
CALL_FUNCTION_VAR (*opcode*), 1656
CALL_FUNCTION_VAR_KW (*opcode*), 1656
call_tracing() (*sys* モジュール), 1527
Callable (*collections* のクラス), 226
callable() (組み込み関数), 7
CallableProxyType (*weakref* モジュール), 253
callback (*optparse.Option* の属性), 618
callback() (*MiniAEFrame.AEServer* のメソッド), 1748
callback_args (*optparse.Option* の属性), 618
callback_kwargs (*optparse.Option* の属性), 618
CalledProcessError, 856
can_change_color() (*curses* モジュール), 678
can_fetch() (*robotparser.RobotFileParser* のメソッド), 489
cancel() (*sched.scheduler* のメソッド), 245
cancel() (*threading.Timer* のメソッド), 776
cancel_join_thread() (*multiprocessing.Queue* のメソッド), 791
CannotSendHeader, 1164
CannotSendRequest, 1164
canonic() (*bdb.Bdb* のメソッド), 1490
canonical() (*decimal.Context* のメソッド), 302
canonical() (*decimal.Decimal* のメソッド), 290
capitalize() (*str* のメソッド), 47
capitalize() (*string* モジュール), 104
captured_stdout() (*test.support* モジュール), 1485
captureWarnings() (*logging* モジュール), 652
capwords() (*string* モジュール), 103
Carbon.AE (モジュール), 1736
Carbon.AH (モジュール), 1736
Carbon.App (モジュール), 1736
Carbon.Appearance (モジュール), 1736
Carbon.CarbonEvents (モジュール), 1738
Carbon.CarbonEvt (モジュール), 1738
Carbon.CF (モジュール), 1736
Carbon.CG (モジュール), 1738
Carbon.Cm (モジュール), 1738
Carbon.Components (モジュール), 1738
Carbon.ControlAccessor (モジュール), 1738
Carbon.Controls (モジュール), 1738
Carbon.CoreFoundation (モジュール), 1738
Carbon.CoreGraphics (モジュール), 1738
Carbon.Ctl (モジュール), 1738
Carbon.Dialogs (モジュール), 1738
Carbon.Dlg (モジュール), 1738
Carbon.Drag (モジュール), 1738
Carbon.DragConst (モジュール), 1738
Carbon.Events (モジュール), 1738
Carbon.Evt (モジュール), 1738
Carbon.File (モジュール), 1738
Carbon.Files (モジュール), 1738
Carbon.Fm (モジュール), 1738
Carbon.Folder (モジュール), 1738
Carbon.Folders (モジュール), 1738
Carbon.Fonts (モジュール), 1738
Carbon.Help (モジュール), 1738
Carbon.IBCarbon (モジュール), 1738
Carbon.IBCarbonRuntime (モジュール), 1738
Carbon.Icons (モジュール), 1738
Carbon.Icons (モジュール), 1738
Carbon.Launch (モジュール), 1738
Carbon.LaunchServices (モジュール), 1738
Carbon.List (モジュール), 1738

Carbon.Lists (モジュール), 1738
Carbon.MacHelp (モジュール), 1738
Carbon.MediaDescr (モジュール), 1738
Carbon.Menu (モジュール), 1738
Carbon.Menus (モジュール), 1738
Carbon.Mlte (モジュール), 1738
Carbon.OSA (モジュール), 1738
Carbon.OSAconst (モジュール), 1738
Carbon.Qd (モジュール), 1738
Carbon.Qdoffs (モジュール), 1738
Carbon.QDOffscreen (モジュール), 1738
Carbon.Qt (モジュール), 1738
Carbon.QuickDraw (モジュール), 1738
Carbon.QuickTime (モジュール), 1738
Carbon.Res (モジュール), 1738
Carbon.Resources (モジュール), 1738
Carbon.Scrap (モジュール), 1738
Carbon.Snd (モジュール), 1739
Carbon.Sound (モジュール), 1739
Carbon.TE (モジュール), 1739
Carbon.TextEdit (モジュール), 1739
Carbon.Win (モジュール), 1739
Carbon.Windows (モジュール), 1739
cast() (*ctypes* モジュール), 746
cat() (*nis* モジュール), 1717
catalog (*cd* モジュール), 1753
catch_warnings (*warnings* のクラス), 1554
category() (*unicodedata* モジュール), 170
cbreak() (*curses* モジュール), 678
cd (モジュール), 1752
CDLL (*ctypes* のクラス), 739
CDROM (*cd* モジュール), 1753
ceil() (*in module math*), 40
ceil() (*math* モジュール), 275
center() (*str* のメソッド), 47
center() (*string* モジュール), 106
CERT.NONE (*ssl* モジュール), 892
CERT.OPTIONAL (*ssl* モジュール), 892
CERT.REQUIRED (*ssl* モジュール), 892
cert_store_stats() (*ssl.SSLContext* のメソッド), 901
cert_time_to_seconds() (*ssl* モジュール), 890
CertificateError, 885
certificates, 906
cfmfile (モジュール), 1776
CFUNCTYPE() (*ctypes* モジュール), 743
CGI
 debugging, 1123
 exceptions, 1125
 protocol, 1116
 security, 1122
 tracebacks, 1125
cgi (モジュール), 1116
cgi_directories (*CGIHTTPServer.CGIHTTPRequestHandler* の属性), 1226
CGIHandler (*wsgiref.handlers* のクラス), 1133
CGIHTTPRequestHandler (*CGIHTTPServer* のクラス), 1226
CGIHTTPServer
 モジュール, 1219
CGIHTTPServer (モジュール), 1225
cglib (モジュール), 1125
CGIXMLRPCRequestHandler (*SimpleXMLRPCServer* のクラス), 1253
chain() (*itertools* モジュール), 325
chaining
 comparisons, 38
CHANNEL.BINDING_TYPES (*ssl* モジュール), 896
channels() (*ossaudiodev.oss.audio.device* のメソッド), 1280
CHAR.MAX (*locale* モジュール), 1304
character, 170
CharacterDataHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1105
characters() (*xml.sax.handler.ContentHandler* のメソッド), 1092

characters-written (*io.BlockingIOError* の属性), 548
 Charset (*email.charset* のクラス), 952
 CHARSET (*mimify* モジュール), 1017
 charset () (*gettext.NullTranslations* のメソッド), 1290
 chdir () (*os* モジュール), 521
 check () (*imaplib.IMAP4* のメソッド), 1180
 check () (*tabnanny* モジュール), 1641
 check_call () (*subprocess* モジュール), 854
 check_forms () (*fl* モジュール), 1757
 check_hostname (*ssl.SSLContext* の属性), 905
 check_output () (*doctest.OutputChecker* のメソッド), 1436
 check_output () (*subprocess* モジュール), 855
 check_py3k_warnings () (*test.support* モジュール), 1485
 check_unused_args () (*string.Formatter* のメソッド), 93
 check_warnings () (*test.support* モジュール), 1484
 checkbox () (*msilib.Dialog* のメソッド), 1681
 checkcache () (*linecache* モジュール), 376
 checkfuncname () (*bdb* モジュール), 1494
 CheckList (*Tix* のクラス), 1355
 checksum
 Cyclic Redundancy Check, 442
 MD5, 503
 SHA, 505
 chflags () (*os* モジュール), 522
 chgat () (*curses.window* のメソッド), 687
 childerr (*popen2.Popen3* の属性), 920
 childNodes (*xml.dom.Node* の属性), 1069
 chmod () (*os* モジュール), 522
 choice () (*random* モジュール), 320
 choices (*optparse.Option* の属性), 618
 choose_boundary () (*mimetools* モジュール), 1010
 chown () (*os* モジュール), 523
 chr () (組み込み関数), 7
 chroot () (*os* モジュール), 522
 Chunk (*chunk* のクラス), 1274
 chunk (モジュール), 1274
 cipher
 DES, 1701
 cipher () (*ssl.SSLSocket* のメソッド), 899
 circle () (*turtle* モジュール), 1366
 CIRCUMFLEX (*token* モジュール), 1636
 CIRCUMFLEXEQUAL (*token* モジュール), 1636
 Clamped (*decimal* のクラス), 306
 class, 1780
 Class (*symtable* のクラス), 1634
 Class browser, 1396
 classic class, 1780
 classmethod () (組み込み関数), 7
 classobj () (*new* モジュール), 262
 ClassType (*types* モジュール), 260
 clean () (*mailbox.Maildir* のメソッド), 989
 cleandoc () (*inspect* モジュール), 1576
 Clear Breakpoint, 1400
 clear () (*collections.deque* のメソッド), 214
 clear () (*cookielib.CookieJar* のメソッド), 1230
 clear () (*curses.window* のメソッド), 687
 clear () (*dict* のメソッド), 65
 clear () (*frozenset* のメソッド), 63
 clear () (*mailbox.Mailbox* のメソッド), 987
 clear () (*threading.Event* のメソッド), 775
 clear () (*turtle* モジュール), 1376, 1383
 clear () (*xml.etree.ElementTree.Element* のメソッド), 1059
 clear_all_breaks () (*bdb.Bdb* のメソッド), 1493
 clear_all_file_breaks () (*bdb.Bdb* のメソッド), 1493
 clear_bppnumber () (*bdb.Bdb* のメソッド), 1493
 clear_break () (*bdb.Bdb* のメソッド), 1493
 clear_flags () (*decimal.Context* のメソッド), 300
 clear_history () (*readline* モジュール), 848
 clear_memo () (*pickle.Pickler* のメソッド), 389
 clear_session_cookies () (*cookielib.CookieJar* のメソッド), 1230
 clearcache () (*linecache* モジュール), 376
 ClearData () (*msilib.Record* のメソッド), 1678

clearok () (*curses.window* のメソッド), 687
 clearscreen () (*turtle* モジュール), 1383
 clearstamp () (*turtle* モジュール), 1367
 clearstamps () (*turtle* モジュール), 1368
 Client () (*multiprocessing.connection* モジュール), 812
 client_address (*BaseHTTPServer.BaseHTTPRequestHandler* の属性), 1220
 clock () (*time* モジュール), 561
 clone () (*email.generator.Generator* のメソッド), 945
 clone () (*pipes.Template* のメソッド), 1710
 clone () (*turtle* モジュール), 1381
 cloneNode () (*xml.dom.minidom.Node* のメソッド), 1082
 cloneNode () (*xml.dom.Node* のメソッド), 1071
 Close () (*_winreg.PyHKEY* のメソッド), 1693
 close () (*aifc.aifc* のメソッド), 1266, 1267
 close () (*anydbm* モジュール), 407
 close () (*asyncore.dispatcher* のメソッド), 924
 close () (*bsddb.bsddbobject* のメソッド), 414
 close () (*bz2.BZ2File* のメソッド), 448
 close () (*chunk.Chunk* のメソッド), 1274
 close () (*Connection* のメソッド), 794
 close () (*dbm* モジュール), 408
 close () (*dl.dl* のメソッド), 1703
 close () (*dumbdbm* モジュール), 416
 close () (*email.parser.FeedParser* のメソッド), 942
 close () (*file* のメソッド), 69
 close () (*fileinput* モジュール), 360
 close () (*FrameWork.Window* のメソッド), 1733
 close () (*ftplib.FTP* のメソッド), 1174
 close () (*gdbm* モジュール), 410
 close () (*hotshot.Profile* のメソッド), 1512
 close () (*HTMLParser.HTMLParser* のメソッド), 1037
 close () (*httplib.HTTPConnection* のメソッド), 1166
 close () (*imaplib.IMAP4* のメソッド), 1180
 close () (*io.IOBase* のメソッド), 549
 close () (*logging.FileHandler* のメソッド), 666
 close () (*logging.Handler* のメソッド), 643
 close () (*logging.handlers.MemoryHandler* のメソッド), 675
 close () (*logging.handlers.NTEventLogHandler* のメソッド), 673
 close () (*logging.handlers.SocketHandler* のメソッド), 669
 close () (*logging.handlers.SysLogHandler* のメソッド), 671
 close () (*mailbox.Mailbox* のメソッド), 987
 close () (*mailbox.Maildir* のメソッド), 989
 close () (*mailbox.MH* のメソッド), 992
 close () (*mmap.mmap* のメソッド), 845
 Close () (*msilib.View* のメソッド), 1677
 close () (*multiprocessing.connection.Listener* のメソッド), 813
 close () (*multiprocessing.pool.multiprocessing.Pool* のメソッド), 810
 close () (*multiprocessing.Queue* のメソッド), 791
 close () (*os* モジュール), 515
 close () (*ossaudiodev.oss_audio_device* のメソッド), 1279
 close () (*ossaudiodev.oss_mixer_device* のメソッド), 1282
 close () (*select.epoll* のメソッド), 759
 close () (*select.kqueue* のメソッド), 761
 close () (*sgmlib.SGMLParser* のメソッド), 1042
 close () (*shelve.Shelf* のメソッド), 401
 close () (*socket.socket* のメソッド), 876
 close () (*sqlite3.Connection* のメソッド), 421
 close () (*StringIO.StringIO* のメソッド), 145
 close () (*sunau.AU_read* のメソッド), 1269
 close () (*sunau.AU_write* のメソッド), 1271
 close () (*tarfile.TarFile* のメソッド), 465
 close () (*telnetlib.Telnet* のメソッド), 1199
 close () (*urllib2.BaseHandler* のメソッド), 1154
 close () (*wave.Wave_read* のメソッド), 1272
 close () (*wave.Wave_write* のメソッド), 1273
 close () (*xml.etree.ElementTree.TreeBuilder* のメソッド), 1063
 close () (*xml.etree.ElementTree.XMLParser* のメソッド), 1064
 close () (*xml.sax.xmlreader.IncrementalParser* のメソッド), 1097
 close () (*zipfile.ZipFile* のメソッド), 452

close_when_done() (*asynchat.async_chat* のメソッド), 927
 closed (*file* の属性), 73
 closed (*io.IOBase* の属性), 549
 closed (*ossaudiodev.oss_audio_device* の属性), 1282
 CloseKey() (*.winreg* モジュール), 1684
 closelog() (*syslog* モジュール), 1718
 closerange() (*os* モジュール), 516
 closing() (*contextlib* モジュール), 1556
 clrtobot() (*curses.window* のメソッド), 687
 clrtoeol() (*curses.window* のメソッド), 687
 cmath (モジュール), 280
 cmd
 モジュール, 1495
 Cmd (*cmd* のクラス), 1307
 cmd (*subprocess.CalledProcessError* の属性), 856
 cmd (モジュール), 1307
 cmdloop() (*cmd.Cmd* のメソッド), 1307
 cmdqueue (*cmd.Cmd* の属性), 1309
 cmp
 組み込み関数, 1303
 cmp() (*filecmp* モジュール), 367
 cmp() (組み込み関数), 8
 cmp_op (*dis* モジュール), 1647
 cmp_to_key() (*functools* モジュール), 339
 cmpfiles() (*filecmp* モジュール), 367
 code (*urllib2.HTTPError* の属性), 1148
 code (*xml.parsers.expat.ExpatError* の属性), 1107
 code (モジュール), 1587
 code object, 78, 404
 code() (*new* モジュール), 262
 Codecs, 150
 decode, 150
 encode, 150
 codecs (モジュール), 150
 coded_value (*Cookie.Morsel* の属性), 1241
 codeop (モジュール), 1589
 codepoint2name (*htmlentitydefs* モジュール), 1047
 CODESET (*locale* モジュール), 1300
 CodeType (*types* モジュール), 260
 coerce() (組み込み関数), 33
 coercion, 1780
 col_offset (*ast.AST* の属性), 1627
 collapse_rfc2231_value() (*email.utils* モジュール), 960
 collect() (*gc* モジュール), 1568
 collect_incoming_data() (*asynchat.async_chat* のメソッド), 927
 collections (モジュール), 210
 COLON (*token* モジュール), 1636
 color() (*fl* モジュール), 1758
 color() (*turtle* モジュール), 1374
 color_content() (*curses* モジュール), 678
 color_pair() (*curses* モジュール), 678
 colormode() (*turtle* モジュール), 1387
 ColorPicker (モジュール), 1739
 colorsys (モジュール), 1275
 column() (*ttk.Treeview* のメソッド), 1344
 COLUMNS, 684
 combinations() (*itertools* モジュール), 325
 combinations_with_replacement() (*itertools* モジュール), 326
 combine() (*datetime.datetime* のクラスメソッド), 186
 combining() (*unicodedata* モジュール), 170
 ComboBox (*Tix* のクラス), 1354
 Combobox (*ttk* のクラス), 1335
 COMMA (*token* モジュール), 1636
 command (*BaseHTTPServer.BaseHTTPRequestHandler* の属性), 1220
 CommandCompiler (*codeop* のクラス), 1590
 commands (モジュール), 1719
 comment (*cookielib.Cookie* の属性), 1237
 COMMENT (*tokenize* モジュール), 1639
 comment (*zipfile.ZipFile* の属性), 455
 comment (*zipfile.ZipInfo* の属性), 457

Comment() (*xml.etree.ElementTree* モジュール), 1056
 comment_url (*cookielib.Cookie* の属性), 1237
 commenters (*shlex.shlex* の属性), 1312
 CommentHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1106
 commit() (*msilib.CAB* のメソッド), 1679
 Commit() (*msilib.Database* のメソッド), 1677
 commit() (*sqlite3.Connection* のメソッド), 421
 common (*filecmp.dircmp* の属性), 368
 Common Gateway Interface, 1116
 common_dirs (*filecmp.dircmp* の属性), 368
 common_files (*filecmp.dircmp* の属性), 368
 common_funny (*filecmp.dircmp* の属性), 368
 common_types (*mimetypes* モジュール), 1013
 commonprefix() (*os.path* モジュール), 354
 communicate() (*subprocess.Popen* のメソッド), 861
 compare() (*decimal.Context* のメソッド), 302
 compare() (*decimal.Decimal* のメソッド), 290
 compare() (*difflib.Differ* のメソッド), 142
 compare_digest() (*hmac* モジュール), 503
 COMPARE_OP (*opcode*), 1654
 compare_signal() (*decimal.Context* のメソッド), 302
 compare_signal() (*decimal.Decimal* のメソッド), 291
 compare_total() (*decimal.Context* のメソッド), 302
 compare_total() (*decimal.Decimal* のメソッド), 291
 compare_total_mag() (*decimal.Context* のメソッド), 302
 compare_total_mag() (*decimal.Decimal* のメソッド), 291
 comparing
 objects, 39
 comparison
 operator, 38
 Comparison (*aetypes* のクラス), 1747
 COMPARISON_FLAGS (*doctest* モジュール), 1421
 comparisons
 chaining, 38
 compile
 組み込み関数, 78, 260, 1624
 Compile (*codeop* のクラス), 1590
 compile() (*compiler* モジュール), 1660
 compile() (*parser.ST* のメソッド), 1625
 compile() (*py.compile* モジュール), 1643
 compile() (*re* モジュール), 113
 compile() (組み込み関数), 8
 compile_command() (*code* モジュール), 1587
 compile_command() (*codeop* モジュール), 1590
 compile_dir() (*compileall* モジュール), 1644
 compile_file() (*compileall* モジュール), 1645
 compile_path() (*compileall* モジュール), 1645
 compileall (モジュール), 1643
 compileall command line option
 -d destdir, 1644
 -f, 1644
 -i list, 1644
 -l, 1644
 -q, 1644
 -x regex, 1644
 directory ..., 1644
 file ..., 1644
 compileFile() (*compiler* モジュール), 1660
 compiler (モジュール), 1659
 compiler.ast (モジュール), 1661
 compiler.visitor (モジュール), 1667
 compilest() (*parser* モジュール), 1624
 complete() (*rlcompleter.Completer* のメソッド), 851
 complete_statement() (*sqlite3* モジュール), 420
 completedefault() (*cmd.Cmd* のメソッド), 1309
 complex
 組み込み関数, 39
 Complex (*numbers* のクラス), 271
 complex (組み込みクラス), 9
 complex number, 1780
 complex number
 literals, 39

オブジェクト, 39

ComplexType (*types* モジュール), 259
 ComponentItem (*aetypes* のクラス), 1747
 compress () (*bz2* モジュール), 450
 compress () (*bz2.BZ2Compressor* のメソッド), 449
 compress () (*itertools* モジュール), 327
 compress () (*jpeg* モジュール), 1768
 compress () (*zlib* モジュール), 442
 compress () (*zlib.Compress* のメソッド), 444
 compress_size (*zipfile.ZipInfo* の属性), 457
 compress_type (*zipfile.ZipInfo* の属性), 456
 compression () (*ssl.SSLSocket* のメソッド), 899
 CompressionError, 460
 compressobj () (*zlib* モジュール), 442
 COMSPEC, 540, 859
 concat () (*operator* モジュール), 344
 concatenation
 operation, 46
 Condition (*multiprocessing* のクラス), 796
 Condition (*threading* のクラス), 772
 condition () (*msilib.Control* のメソッド), 1681
 Condition () (*multiprocessing.managers.SyncManager* のメソッド), 804
 ConfigParser (*ConfigParser* のクラス), 482
 ConfigParser (モジュール), 481
 configuration
 file, 481
 file, debugger, 1498
 file, path, 1579
 file, user, 1582
 configuration information, 1542
 configure () (*tk.Style* のメソッド), 1348
 confstr () (*os* モジュール), 542
 confstr.names (*os* モジュール), 543
 conjugate () (*complex number method*), 40
 conjugate () (*decimal.Decimal* のメソッド), 291
 conjugate () (*numbers.Complex* のメソッド), 271
 connect () (*asyncore.dispatcher* のメソッド), 923
 connect () (*ftplib.FTP* のメソッド), 1171
 connect () (*httplib.HTTPConnection* のメソッド), 1166
 connect () (*multiprocessing.managers.BaseManager* のメソッド), 803
 connect () (*smtpplib.SMTP* のメソッド), 1193
 connect () (*socket.socket* のメソッド), 876
 connect () (*sqlite3* モジュール), 419
 connect_ex () (*socket.socket* のメソッド), 876
 Connection (*sqlite3* のクラス), 421
 connection (*sqlite3.Cursor* の属性), 431
 Connection (組み込みクラス), 793
 ConnectRegistry () (*winreg* モジュール), 1684
 const (*optparse.Option* の属性), 618
 constructor () (*copy_reg* モジュール), 399
 container
 iteration over, 43
 Container (*collections* のクラス), 226
 contains () (*operator* モジュール), 345
 content type
 MIME, 1012
 ContentHandler (*xml.sax.handler* のクラス), 1088
 contents (*ctypes.Pointer* の属性), 756
 ContentTooShortError, 1144
 Context (*decimal* のクラス), 299
 context (*ssl.SSLSocket* の属性), 900
 context management protocol, 76
 context manager, 1781
 context manager, 76
 context.diff () (*difflib* モジュール), 134
 contextlib (モジュール), 1555
 contextmanager () (*contextlib* モジュール), 1555
 CONTINUE_LOOP (*opcode*), 1651
 control (*cd* モジュール), 1753
 Control (*msilib* のクラス), 1680
 Control (*Tix* のクラス), 1354

control () (*msilib.Dialog* のメソッド), 1681
 control () (*select.kqueue* のメソッド), 761
 controlnames (*curses.ascii* モジュール), 703
 controls () (*ossaudiodev.oss_mixer_device* のメソッド), 1282
 ConversionError, 494
 conversions
 numeric, 40
 convert () (*email.charset.Charset* のメソッド), 954
 convert_arg_line_to_args () (*argparse.ArgumentParser* のメソッド), 600
 convert_charref () (*sgmlib.SGMLParser* のメソッド), 1043
 convert_codepoint () (*sgmlib.SGMLParser* のメソッド), 1043
 convert_entityref () (*sgmlib.SGMLParser* のメソッド), 1043
 convert_field () (*string.Formatter* のメソッド), 94
 Cookie (*cookielib* のクラス), 1228
 Cookie (モジュール), 1238
 CookieError, 1239
 CookieJar (*cookielib* のクラス), 1227
 cookiejar (*urllib2.HTTPCookieProcessor* の属性), 1156
 cookielib (モジュール), 1227
 CookiePolicy (*cookielib* のクラス), 1228
 Coordinated Universal Time, 560
 Copy, 1400
 copy
 モジュール, 399
 copy (モジュール), 263
 copy () (*copy* モジュール), 263
 copy () (*decimal.Context* のメソッド), 300
 copy () (*dict* のメソッド), 65
 copy () (*findertools* モジュール), 1726
 copy () (*frozenset* のメソッド), 61
 copy () (*hashlib.hash* のメソッド), 501
 copy () (*hmac.HMAC* のメソッド), 503
 copy () (*imaplib.IMAP4* のメソッド), 1180
 copy () (*macostools* モジュール), 1725
 copy () (*md5.md5* のメソッド), 504
 copy () (*multiprocessing.sharedctypes* モジュール), 800
 copy () (*pipes.Template* のメソッド), 1710
 copy () (*sha.sha* のメソッド), 505
 copy () (*shutil* モジュール), 377
 copy () (*zlib.Compress* のメソッド), 444
 copy () (*zlib.Decompress* のメソッド), 445
 copy_abs () (*decimal.Context* のメソッド), 302
 copy_abs () (*decimal.Decimal* のメソッド), 291
 copy_decimal () (*decimal.Context* のメソッド), 300
 copy_location () (*ast* モジュール), 1631
 copy_negate () (*decimal.Context* のメソッド), 302
 copy_negate () (*decimal.Decimal* のメソッド), 291
 copy_reg (モジュール), 399
 copy_sign () (*decimal.Context* のメソッド), 302
 copy_sign () (*decimal.Decimal* のメソッド), 292
 copy2 () (*shutil* モジュール), 377
 copybinary () (*mimetools* モジュール), 1011
 copyfile () (*shutil* モジュール), 377
 copyfileobj () (*shutil* モジュール), 377
 copying files, 376
 copyliteral () (*mimetools* モジュール), 1011
 copymessage () (*mhlib.Folder* のメソッド), 1010
 copymode () (*shutil* モジュール), 377
 copyright (*sys* モジュール), 1527
 copyright (組み込み変数), 36
 copysign () (*math* モジュール), 275
 copystat () (*shutil* モジュール), 377
 copytree () (*macostools* モジュール), 1725
 copytree () (*shutil* モジュール), 378
 cos () (*cmath* モジュール), 282
 cos () (*math* モジュール), 278
 cosh () (*cmath* モジュール), 282
 cosh () (*math* モジュール), 279
 count () (*array.array* のメソッド), 237
 count () (*collections.deque* のメソッド), 214

count () (*itertools* モジュール), 327
count () (*list method*), 58
count () (*str* のメソッド), 47
count () (*string* モジュール), 104
Counter (*collections* のクラス), 211
countOf () (*operator* モジュール), 345
countTestCases () (*unittest.TestCase* のメソッド), 1461
countTestCases () (*unittest.TestSuite* のメソッド), 1463
CoverageResults (*trace* のクラス), 1521
cPickle
モジュール, 399
cPickle (モジュール), 399
cProfile (モジュール), 1504
CPU time, 561
cpu.count () (*multiprocessing* モジュール), 792
CPython, 1781
CRC (*zipfile.ZipInfo* の属性), 457
crc.hqx () (*binascii* モジュール), 1030
crc32 () (*binascii* モジュール), 1030
crc32 () (*zlib* モジュール), 442
create () (*imaplib.IMAP4* のメソッド), 1180
create.aggregate () (*sqlite3.Connection* のメソッド), 422
create.collation () (*sqlite3.Connection* のメソッド), 423
create.connection () (*socket* モジュール), 871
create.decimal () (*decimal.Context* のメソッド), 300
create.decimal.from.float () (*decimal.Context* のメソッド), 301
create.default_context () (*ssl* モジュール), 887
create.function () (*sqlite3.Connection* のメソッド), 422
CREATE_NEW_CONSOLE (*subprocess* モジュール), 864
CREATE_NEW_PROCESS_GROUP (*subprocess* モジュール), 864
create.socket () (*asyncore.dispatcher* のメソッド), 923
create.stats () (*profile.Profile* のメソッド), 1505
create.string.buffer () (*ctypes* モジュール), 746
create.system (*zipfile.ZipInfo* の属性), 457
create.unicode.buffer () (*ctypes* モジュール), 747
create.version (*zipfile.ZipInfo* の属性), 457
createAttribute () (*xml.dom.Document* のメソッド), 1073
createAttributeNS () (*xml.dom.Document* のメソッド), 1073
createComment () (*xml.dom.Document* のメソッド), 1072
createDocument () (*xml.dom.DOMImplementation* のメソッド), 1068
createDocumentType () (*xml.dom.DOMImplementation* のメソッド), 1068
createElement () (*xml.dom.Document* のメソッド), 1072
createElementNS () (*xml.dom.Document* のメソッド), 1072
createfilehandler () (*Tkinter.Widget.tk* のメソッド), 1329
CreateKey () (*._winreg* モジュール), 1685
CreateKeyEx () (*._winreg* モジュール), 1685
createLock () (*logging.Handler* のメソッド), 642
createLock () (*logging.NullHandler* のメソッド), 666
createparser () (*cd* モジュール), 1752
createProcessingInstruction () (*xml.dom.Document* のメソッド), 1072
CreateRecord () (*msilib* モジュール), 1676
createSocket () (*logging.handlers.SocketHandler* のメソッド), 670
createTextNode () (*xml.dom.Document* のメソッド), 1072
credits (組み込み変数), 36
critical () (*logging* モジュール), 650
critical () (*logging.Logger* のメソッド), 640
CRNCYSTR (*locale* モジュール), 1301
crop () (*imageop* モジュール), 1263
cross () (*audioop* モジュール), 1260
crypt
モジュール, 1699
crypt (モジュール), 1701
crypt () (*crypt* モジュール), 1701
crypt (3), 1701
cryptography, 499
cStringIO (モジュール), 146
csv, 471

csv (モジュール), 471
ctermid () (*os* モジュール), 509
ctime () (*datetime.date* のメソッド), 183
ctime () (*datetime.datetime* のメソッド), 190
ctime () (*time* モジュール), 561
ctrl () (*curses.ascii* モジュール), 702
CTRL_BREAK_EVENT (*signal* モジュール), 915
CTRL_C_EVENT (*signal* モジュール), 915
ctypes (モジュール), 716
curdir (*os* モジュール), 543
currency () (*locale* モジュール), 1303
current () (*tk.ComboBox* のメソッド), 1335
current.process () (*multiprocessing* モジュール), 792
current.thread () (*threading* モジュール), 764
CurrentByteIndex (*xml.parsers.expat.xmlparser* の属性), 1104
CurrentColumnNumber (*xml.parsers.expat.xmlparser* の属性), 1104
currentframe () (*inspect* モジュール), 1578
CurrentLineNumber (*xml.parsers.expat.xmlparser* の属性), 1104
currentThread () (*threading* モジュール), 764
curs.set () (*curses* モジュール), 678
curses (モジュール), 676
curses.ascii (モジュール), 700
curses.panel (モジュール), 703
curses.textpad (モジュール), 698
Cursor (*sqlite3* のクラス), 427
cursor () (*sqlite3.Connection* のメソッド), 421
cursyncup () (*curses.window* のメソッド), 687
curval (*EasyDialogs.ProgressBar* の属性), 1729
Cut, 1400
cwd () (*ftplib.FTP* のメソッド), 1174
cycle () (*itertools* モジュール), 328
Cyclic Redundancy Check, 442
D_FMT (*locale* モジュール), 1300
D_T_FMT (*locale* モジュール), 1300
daemon (*multiprocessing.Process* の属性), 787
daemon (*threading.Thread* の属性), 769
data
packing binary, 126
tabular, 471
Data (*plistlib* のクラス), 496
data (*select.kevent* の属性), 763
data (*UserDict.IterableUserDict* の属性), 256
data (*UserList.UserList* の属性), 257
data (*UserString.MutableString* の属性), 258
data (*xml.dom.Comment* の属性), 1075
data (*xml.dom.ProcessingInstruction* の属性), 1076
data (*xml.dom.Text* の属性), 1075
data (*xmlrpclib.Binary* の属性), 1247
data () (*xml.etree.ElementTree.TreeBuilder* のメソッド), 1063
database
Unicode, 170
databases, 415
DatagramHandler (*logging.handlers* のクラス), 670
DatagramRequestHandler (*SocketServer* のクラス), 1215
DATASIZE (*cd* モジュール), 1752
date (*datetime* のクラス), 180
date () (*datetime.datetime* のメソッド), 188
date () (*nntplib.NNTP* のメソッド), 1190
date.time (*zipfile.ZipInfo* の属性), 456
date.time.string ()
(*BaseHTTPServer.BaseHTTPRequestHandler* のメソッド), 1222
datetime (*datetime* のクラス), 184
DateTime (*xmlrpclib* のクラス), 1247
datetime (モジュール), 175
day (*datetime.date* の属性), 181
day (*datetime.datetime* の属性), 186
day.abbrev (*calendar* モジュール), 209
day.name (*calendar* モジュール), 209

- daylight (*time* モジュール), 561
Daylight Saving Time, 560
DbfilenameShelf (*shelve* のクラス), 402
dbhash
 モジュール, 405
dbhash (モジュール), 410
dbm
 モジュール, 401, 405, 409
dbm (モジュール), 408
dcgettext () (*locale* モジュール), 1305
deactivate_form () (*fl.form* のメソッド), 1759
debug (*imaplib.IMAP4* の属性), 1185
DEBUG (*re* モジュール), 113
debug (*shlex.shlex* の属性), 1313
debug (*zipfile.ZipFile* の属性), 455
debug () (*doctest* モジュール), 1438
debug () (*logging* モジュール), 649
debug () (*logging.Logger* のメソッド), 639
debug () (*pipes.Template* のメソッド), 1710
debug () (*unittest.TestCase* のメソッド), 1454
debug () (*unittest.TestSuite* のメソッド), 1463
DEBUG_COLLECTABLE (*gc* モジュール), 1571
DEBUG_INSTANCES (*gc* モジュール), 1571
DEBUG_LEAK (*gc* モジュール), 1571
DEBUG_OBJECTS (*gc* モジュール), 1571
DEBUG_SAVEALL (*gc* モジュール), 1571
debug_src () (*doctest* モジュール), 1439
DEBUG_STATS (*gc* モジュール), 1571
DEBUG_UNCOLLECTABLE (*gc* モジュール), 1571
debugger, 1399, 1534, 1540
 configuration file, 1498
debugging, 1495
 CGI, 1123
DebuggingServer (*smtpd* のクラス), 1196
DebugRunner (*doctest* のクラス), 1439
DebugStr () (*MacOS* モジュール), 1724
Decimal (*decimal* のクラス), 288
decimal (モジュール), 284
decimal () (*unicodedata* モジュール), 170
DecimalException (*decimal* のクラス), 306
decode
 Codecs, 150
decode () (*base64* モジュール), 1027
decode () (*codecs* モジュール), 150
decode () (*codecs.Codec* のメソッド), 156
decode () (*codecs.IncrementalDecoder* のメソッド), 158
decode () (*json.JSONDecoder* のメソッド), 978
decode () (*mimetools* モジュール), 1010
decode () (*quopri* モジュール), 1032
decode () (*str* のメソッド), 47
decode () (*uu* モジュール), 1033
decode () (*xmlrpclib.Binary* のメソッド), 1248
decode () (*xmlrpclib.DateTime* のメソッド), 1247
decode.header () (*email.header* モジュール), 952
decode.params () (*email.utils* モジュール), 960
decode.rfc2231 () (*email.utils* モジュール), 960
DecodedGenerator (*email.generator* のクラス), 946
decodestring () (*base64* モジュール), 1027
decodestring () (*quopri* モジュール), 1032
decomposition () (*unicodedata* モジュール), 171
decompress () (*bz2* モジュール), 450
decompress () (*bz2.BZ2Decompressor* のメソッド), 450
decompress () (*jpeg* モジュール), 1768
decompress () (*zlib* モジュール), 443
decompress () (*zlib.Decompress* のメソッド), 444
decompressobj () (*zlib* モジュール), 443
decorator, 1781
DEDENT (*token* モジュール), 1636
dedent () (*textwrap* モジュール), 147
deepcopy () (*copy* モジュール), 263
def_prog_mode () (*curses* モジュール), 678
def_shell_mode () (*curses* モジュール), 678
default (*optparse.Option* の属性), 618
default () (*cmd.Cmd* のメソッド), 1308
default () (*compiler.visitor.ASTVisitor* のメソッド), 1667
default () (*json.JSONEncoder* のメソッド), 980
DEFAULT_BUFFER_SIZE (*io* モジュール), 545
default.bufsize (*xml.dom.pulldom* モジュール), 1085
default_factory (*collections.defaultdict* の属性), 218
DEFAULT_FORMAT (*tarfile* モジュール), 461
default_open () (*urllib2.BaseHandler* のメソッド), 1154
default.timer () (*timeit* モジュール), 1514
DefaultContext (*decimal* のクラス), 299
DefaultCookiePolicy (*cookielib* のクラス), 1228
defaultdict (*collections* のクラス), 217
DefaultHandler () (*xml.parsers.expat.xmlparser* のメソッド), 1106
DefaultHandlerExpand () (*xml.parsers.expat.xmlparser* のメソッド), 1106
defaults () (*ConfigParser.RawConfigParser* のメソッド), 484
defaultTestLoader (*unittest* モジュール), 1468
defaultTestResult () (*unittest.TestCase* のメソッド), 1461
defects (*email.message.Message* の属性), 941
defpath (*os* モジュール), 544
degrees () (*math* モジュール), 278
degrees () (*turtle* モジュール), 1371
del
 文, 58, 63
del_param () (*email.message.Message* のメソッド), 938
delattr () (組み込み関数), 9
delay () (*turtle* モジュール), 1385
delay_output () (*curses* モジュール), 678
delayload (*cookielib.FileCookieJar* の属性), 1231
delch () (*curses.window* のメソッド), 687
dele () (*poplib.POP3* のメソッド), 1177
delete () (*ftplib.FTP* のメソッド), 1174
delete () (*imaplib.IMAP4* のメソッド), 1180
delete () (*ttk.Treeview* のメソッド), 1344
DELETE_ATTR (*opcode*), 1653
DELETE_FAST (*opcode*), 1655
DELETE_GLOBAL (*opcode*), 1653
DELETE_NAME (*opcode*), 1653
DELETE_SLICE+0 (*opcode*), 1651
DELETE_SLICE+1 (*opcode*), 1651
DELETE_SLICE+2 (*opcode*), 1651
DELETE_SLICE+3 (*opcode*), 1651
DELETE_SUBSCR (*opcode*), 1651
deleteacl () (*imaplib.IMAP4* のメソッド), 1180
deletefilehandler () (*Tkinter.Widget.tk* のメソッド), 1329
deletefolder () (*mhlib.MH* のメソッド), 1009
DeleteKey () (*winreg* モジュール), 1685
DeleteKeyEx () (*winreg* モジュール), 1685
deleteln () (*curses.window* のメソッド), 687
deleteMe () (*bdb.Breakpoint* のメソッド), 1489
DeleteValue () (*winreg* モジュール), 1686
delimiter (*csv.Dialect* の属性), 476
delitem () (*operator* モジュール), 345
deliver_challenge () (*multiprocessing.connection* モジュール), 812
delslice () (*operator* モジュール), 345
demo_app () (*wsgiref.simple_server* モジュール), 1130
denominator (*numbers.Rational* の属性), 272
DeprecationWarning, 87
deque (*collections* のクラス), 213
DER_cert_to_PEM_cert () (*ssl* モジュール), 890
derwin () (*curses.window* のメソッド), 687
DES
 cipher, 1701
description (*sqlite3.Cursor* の属性), 430
description () (*nntplib.NNTP* のメソッド), 1188
descriptions () (*nntplib.NNTP* のメソッド), 1188
descriptor, 1781
 descriptor
 file, 70
dest (*optparse.Option* の属性), 618
Detach () (*winreg.PyHKEY* のメソッド), 1694

- `detach()` (*io.BufferedIOBase* のメソッド), 552
- `detach()` (*io.TextIOBase* のメソッド), 556
- `detach()` (*tk.Treeview* のメソッド), 1345
- deterministic profiling, 1501
- DEVICE (モジュール), 1766
- devnull (*os* モジュール), 544
- dgettext() (*gettext* モジュール), 1286
- dgettext() (*locale* モジュール), 1305
- Dialect (*csv* のクラス), 474
- dialect (*csv.csvreader* の属性), 477
- dialect (*csv.csvwriter* の属性), 478
- Dialog (*msilib* のクラス), 1681
- DialogWindow() (*FrameWork* モジュール), 1731
- dict (*2to3 fixer*), 1476
- dict (組み込みクラス), 63
- dict() (*multiprocessing.managers.SyncManager* のメソッド), 804
- dictConfig() (*logging.config* モジュール), 653
- dictionary, **1781**
- dictionary
 - type, operations on, 63
 - オブジェクト, 63
- dictionary view, **1781**
- DictionaryType (*types* モジュール), 260
- DictMixin (*UserDict* のクラス), 256
- DictProxyType (*types* モジュール), 261
- DictReader (*csv* のクラス), 473
- DictType (*types* モジュール), 260
- DictWriter (*csv* のクラス), 474
- diff_files (*filecmp.dircmp* の属性), 369
- Differ (*difflib* のクラス), 133, 142
- difference() (*frozenset* のメソッド), 61
- difference.update() (*frozenset* のメソッド), 62
- difflib (モジュール), 132
- digest() (*hashlib.hash* のメソッド), 501
- digest() (*hmac.HMAC* のメソッド), 502
- digest() (*md5.md5* のメソッド), 504
- digest() (*sha.sha* のメソッド), 505
- digest.size (*md5* モジュール), 504
- digest.size (*sha* モジュール), 505
- digit() (*unicodedata* モジュール), 170
- digits (*string* モジュール), 92
- dir() (*ftplib.FTP* のメソッド), 1174
- dir() (組み込み関数), 9
- dircache (モジュール), 382
- dircmp (*filecmp* のクラス), 367
- directory
 - changing, 521
 - creating, 525
 - deleting, 378, 526
 - site-packages, 1579
 - site-python, 1579
 - traversal, 531
 - walking, 531
- Directory (*msilib* のクラス), 1679
- directory ...
 - compileall command line option, 1644
- DirList (*Tix* のクラス), 1354
- dirname() (*os.path* モジュール), 354
- DirSelectBox (*Tix* のクラス), 1355
- DirSelectDialog (*Tix* のクラス), 1355
- DirTree (*Tix* のクラス), 1354
- dis (モジュール), 1646
- dis() (*dis* モジュール), 1646
- dis() (*pickletools* モジュール), 1657
- disable() (*bdb.Breakpoint* のメソッド), 1490
- disable() (*gc* モジュール), 1568
- disable() (*logging* モジュール), 650
- disable() (*profile.Profile* のメソッド), 1505
- disable.interspersed.args() (*optparse.OptionParser* のメソッド), 623
- DisableReflectionKey() (*_winreg* モジュール), 1690
- disassemble() (*dis* モジュール), 1646
- discard (*cookielib.Cookie* の属性), 1237
- discard() (*frozenset* のメソッド), 63
- discard() (*mailbox.Mailbox* のメソッド), 985
- discard() (*mailbox.MH* のメソッド), 992
- discard.buffers() (*asynchat.async_chat* のメソッド), 927
- disco() (*dis* モジュール), 1647
- discover() (*unittest.TestLoader* のメソッド), 1464
- dispatch() (*compiler.visitor.ASTVisitor* のメソッド), 1667
- dispatch.call() (*bdb.Bdb* のメソッド), 1491
- dispatch.exception() (*bdb.Bdb* のメソッド), 1491
- dispatch.line() (*bdb.Bdb* のメソッド), 1491
- dispatch.return() (*bdb.Bdb* のメソッド), 1491
- dispatcher (*asyncore* のクラス), 922
- dispatcher.with.send (*asyncore* のクラス), 924
- displayhook() (*sys* モジュール), 1528
- dist() (*platform* モジュール), 708
- distance() (*turtle* モジュール), 1370
- distb() (*dis* モジュール), 1646
- distutils (モジュール), 1523
- dither2grey2() (*imageop* モジュール), 1264
- dither2mono() (*imageop* モジュール), 1264
- div() (*operator* モジュール), 343
- divide() (*decimal.Context* のメソッド), 302
- divide.int() (*decimal.Context* のメソッド), 302
- division
 - integer, 40
 - long integer, 40
- DivisionByZero (*decimal* のクラス), 306
- divmod() (*decimal.Context* のメソッド), 302
- divmod() (組み込み関数), 10
- dl (モジュール), 1702
- DllCanUnloadNow() (*ctypes* モジュール), 747
- DllGetClassObject() (*ctypes* モジュール), 747
- dllhandle (*sys* モジュール), 1528
- dngettext() (*gettext* モジュール), 1287
- do.activate() (*FrameWork.ScrolledWindow* のメソッド), 1734
- do.activate() (*FrameWork.Window* のメソッド), 1733
- do.char() (*FrameWork.Application* のメソッド), 1732
- do.clear() (*bdb.Bdb* のメソッド), 1492
- do.command() (*curses.textpad.Textbox* のメソッド), 699
- do.contentclick() (*FrameWork.Window* のメソッド), 1733
- do.controlhit() (*FrameWork.ControlsWindow* のメソッド), 1733
- do.controlhit() (*FrameWork.ScrolledWindow* のメソッド), 1734
- do.dialogevent() (*FrameWork.Application* のメソッド), 1733
- do.forms() (*fl* モジュール), 1757
- do.GET() (*SimpleHTTPServer.SimpleHTTPRequestHandler* のメソッド), 1224
- do.handshake() (*ssl.SSLSocket* のメソッド), 898
- do.HEAD() (*SimpleHTTPServer.SimpleHTTPRequestHandler* のメソッド), 1224
- do.itemhit() (*FrameWork.DialogWindow* のメソッド), 1735
- do.POST() (*CGIHTTPServer.CGIHTTPRequestHandler* のメソッド), 1226
- do.postresize() (*FrameWork.ScrolledWindow* のメソッド), 1734
- do.postresize() (*FrameWork.Window* のメソッド), 1733
- do.update() (*FrameWork.Window* のメソッド), 1733
- doc.header (*cmd.Cmd* の属性), 1309
- DocCGIXMLRPCRequestHandler (*DocXMLRPCServer* のクラス), 1257
- DocFileSuite() (*doctest* モジュール), 1427
- doCleanups() (*unittest.TestCase* のメソッド), 1461
- docmd() (*smtplib.SMTP* のメソッド), 1192
- docstring, **1781**
- docstring (*doctest.DocTest* の属性), 1431
- DocTest (*doctest* のクラス), 1431
- doctest (モジュール), 1410
- DocTestFailure, 1439
- DocTestFinder (*doctest* のクラス), 1432

- DocTestParser (*doctest* のクラス), 1433
 DocTestRunner (*doctest* のクラス), 1434
 DocTestSuite () (*doctest* モジュール), 1429
 doctype () (*xml.etree.ElementTree.TreeBuilder* のメソッド), 1063
 doctype () (*xml.etree.ElementTree.XMLParser* のメソッド), 1064
 documentation
 generation, 1409
 online, 1409
 documentElement (*xml.dom.Document* の属性), 1072
 DocXMLRPCRequestHandler (*DocXMLRPCServer* のクラス), 1258
 DocXMLRPCServer (*DocXMLRPCServer* のクラス), 1257
 DocXMLRPCServer (モジュール), 1257
 domain.initial.dot (*cookielib.Cookie* の属性), 1237
 domain.return.ok () (*cookielib.CookiePolicy* のメソッド), 1232
 domain.specified (*cookielib.Cookie* の属性), 1237
 DomainLiberal (*cookielib.DefaultCookiePolicy* の属性), 1236
 DomainRFC2965Match (*cookielib.DefaultCookiePolicy* の属性), 1236
 DomainStrict (*cookielib.DefaultCookiePolicy* の属性), 1236
 DomainStrictNoDots (*cookielib.DefaultCookiePolicy* の属性), 1235
 DomainStrictNonDomain (*cookielib.DefaultCookiePolicy* の属性), 1236
 DOMEventStream (*xml.dom.pulldom* のクラス), 1085
 DOMException, 1076
 DomStringSizeErr, 1076
 done () (*turtle* モジュール), 1380
 done () (*xdrlib.Unpacker* のメソッド), 493
 DONT_ACCEPT_BLANKLINE (*doctest* モジュール), 1420
 DONT_ACCEPT_TRUE_FOR_1 (*doctest* モジュール), 1419
 dont.write.bytecode (*sys* モジュール), 1528
 doRollover () (*logging.handlers.RotatingFileHandler* のメソッド), 668
 doRollover () (*logging.handlers.TimedRotatingFileHandler* のメソッド), 669
 DOT (*token* モジュール), 1636
 dot () (*turtle* モジュール), 1367
 DOTALL (*re* モジュール), 114
 doublequote (*csv.Dialect* の属性), 476
 DOUBLESASH (*token* モジュール), 1636
 DOUBLESASHEQUAL (*token* モジュール), 1636
 DOUBLESTAR (*token* モジュール), 1636
 DOUBLESTAREQUAL (*token* モジュール), 1636
 doupdate () (*curses* モジュール), 678
 down () (*turtle* モジュール), 1371
 drop.whitespace (*textwrap.TextWrapper* の属性), 149
 dropwhile () (*itertools* モジュール), 328
 dst () (*datetime.datetime* のメソッド), 189
 dst () (*datetime.time* のメソッド), 195
 dst () (*datetime.tzinfo* のメソッド), 196
 DTDHandler (*xml.sax.handler* のクラス), 1088
 duck-typing, 1781
 dumbdbm
 モジュール, 405
 dumbdbm (モジュール), 415
 DumbWriter (*formatter* のクラス), 1673
 dummy.thread (モジュール), 779
 dummy.threading (モジュール), 779
 dump () (*ast* モジュール), 1633
 dump () (*json* モジュール), 975
 dump () (*marshal* モジュール), 404
 dump () (*pickle* モジュール), 387
 dump () (*pickle.Pickler* のメソッド), 389
 dump () (*xml.etree.ElementTree* モジュール), 1056
 dump.address.pair () (*rfc822* モジュール), 1022
 dump.stats () (*profile.Profile* のメソッド), 1505
 dump.stats () (*pstats.Stats* のメソッド), 1506
 dumps () (*json* モジュール), 976
 dumps () (*marshal* モジュール), 405
 dumps () (*pickle* モジュール), 388
 dumps () (*xmlrpclib* モジュール), 1251
 dup () (*os* モジュール), 516
 dup () (*posixfile.posixfile* のメソッド), 1712
 DUP_TOP (*opcode*), 1648
 DUP_TOPX (*opcode*), 1653
 dup2 () (*os* モジュール), 516
 dup2 () (*posixfile.posixfile* のメソッド), 1712
 DuplicateSectionError, 483
 dwFlags (*subprocess.STARTUPINFO* の属性), 862
 DynLoadSuffixImporter (*imputil* のクラス), 1607
 e (*cmath* モジュール), 283
 e (*math* モジュール), 279
 E2BIG (*errno* モジュール), 709
 EACCESS (*errno* モジュール), 709
 EADDRINUSE (*errno* モジュール), 714
 EADDRNOTAVAIL (*errno* モジュール), 714
 EADV (*errno* モジュール), 713
 EAFNOSUPPORT (*errno* モジュール), 714
 EAFP, 1782
 EAGAIN (*errno* モジュール), 709
 EALREADY (*errno* モジュール), 715
 east_asian_width () (*unicodedata* モジュール), 170
 EasyDialogs (モジュール), 1727
 EBADF (*errno* モジュール), 712
 EBADF (*errno* モジュール), 709
 EBADFD (*errno* モジュール), 713
 EBADMSG (*errno* モジュール), 713
 EBADR (*errno* モジュール), 712
 EBADRQC (*errno* モジュール), 712
 EBADSLT (*errno* モジュール), 712
 EBFONT (*errno* モジュール), 712
 EBUSY (*errno* モジュール), 710
 ECHILD (*errno* モジュール), 709
 echo () (*curses* モジュール), 679
 echochar () (*curses.window* のメソッド), 688
 ECHRN (*errno* モジュール), 711
 ECOMM (*errno* モジュール), 713
 ECONNABORTED (*errno* モジュール), 715
 ECONNREFUSED (*errno* モジュール), 715
 ECONNRESET (*errno* モジュール), 715
 EDEADLK (*errno* モジュール), 711
 EDEADLOCK (*errno* モジュール), 712
 EDESTADDRREQ (*errno* モジュール), 714
 edit () (*curses.textpad.Textbox* のメソッド), 699
 EDOM (*errno* モジュール), 711
 EDOTDOT (*errno* モジュール), 713
 EDQUOTE (*errno* モジュール), 716
 EEXIST (*errno* モジュール), 710
 EFAULT (*errno* モジュール), 709
 EFBIG (*errno* モジュール), 710
 effective () (*bdb* モジュール), 1494
 ehlo () (*smtpplib.SMTP* のメソッド), 1193
 ehlo.or.helo.if.needed () (*smtpplib.SMTP* のメソッド), 1193
 EHOSTDOWN (*errno* モジュール), 715
 EHOSTUNREACH (*errno* モジュール), 715
 EIDRM (*errno* モジュール), 711
 EILSEQ (*errno* モジュール), 714
 EINPROGRESS (*errno* モジュール), 715
 EINTR (*errno* モジュール), 709
 EINVAL (*errno* モジュール), 710
 EIO (*errno* モジュール), 709
 EISCONN (*errno* モジュール), 715
 EISDIR (*errno* モジュール), 710
 EISNAM (*errno* モジュール), 716
 EL2HLT (*errno* モジュール), 712
 EL2NSYNC (*errno* モジュール), 711
 EL3HLT (*errno* モジュール), 711
 EL3RST (*errno* モジュール), 711
 Element (*xml.etree.ElementTree* のクラス), 1059
 element.create () (*ttk.Style* のメソッド), 1350

`element_names()` (*ttk.Style* のメソッド), 1350
`element_options()` (*ttk.Style* のメソッド), 1350
`ElementDeclHandler()` (*xml.parsers.expat.xmlparser* のメソッド), 1105
`elements()` (*collections.Counter* のメソッド), 211
`ElementTree` (*xml.etree.ElementTree* のクラス), 1061
`ELIBACC` (*errno* モジュール), 713
`ELIBBAD` (*errno* モジュール), 713
`ELIBEXEC` (*errno* モジュール), 713
`ELIBMAX` (*errno* モジュール), 713
`ELIBSCN` (*errno* モジュール), 713
`Ellinghouse, Lance`, 1033
`ELLIPSIS` (*doctest* モジュール), 1420
`Ellipsis` (組み込み変数), 35
`EllipsisType` (*types* モジュール), 261
`ELNRNG` (*errno* モジュール), 711
`ELOOP` (*errno* モジュール), 711
`email` (モジュール), 931
`email.charset` (モジュール), 952
`email.encoders` (モジュール), 956
`email.errors` (モジュール), 957
`email.generator` (モジュール), 944
`email.header` (モジュール), 950
`email.iterators` (モジュール), 961
`email.message` (モジュール), 932
`email.mime` (モジュール), 946
`email.parser` (モジュール), 941
`email.utils` (モジュール), 958
`EMFILE` (*errno* モジュール), 710
`emit()` (*logging.FileHandler* のメソッド), 666
`emit()` (*logging.Handler* のメソッド), 643
`emit()` (*logging.handlers.BufferingHandler* のメソッド), 674
`emit()` (*logging.handlers.DatagramHandler* のメソッド), 670
`emit()` (*logging.handlers.HTTPHandler* のメソッド), 675
`emit()` (*logging.handlers.NTEventLogHandler* のメソッド), 673
`emit()` (*logging.handlers.RotatingFileHandler* のメソッド), 668
`emit()` (*logging.handlers.SMTPHandler* のメソッド), 674
`emit()` (*logging.handlers.SocketHandler* のメソッド), 669
`emit()` (*logging.handlers.SysLogHandler* のメソッド), 671
`emit()` (*logging.handlers.TimedRotatingFileHandler* のメソッド), 669
`emit()` (*logging.handlers.WatchedFileHandler* のメソッド), 667
`emit()` (*logging.NullHandler* のメソッド), 666
`emit()` (*logging.StreamHandler* のメソッド), 665
`EMLINK` (*errno* モジュール), 710
`Empty`, 248
`empty()` (*multiprocessing.multiprocessing.queues.SimpleQueue* のメソッド), 792
`empty()` (*multiprocessing.Queue* のメソッド), 790
`empty()` (*Queue.Queue* のメソッド), 248
`empty()` (*sched.scheduler* のメソッド), 245
`EMPTY_NAMESPACE` (*xml.dom* モジュール), 1066
`emptyline()` (*cmd.Cmd* のメソッド), 1308
`EMSGSIZE` (*errno* モジュール), 714
`EMULTIHOP` (*errno* モジュール), 713
`enable()` (*bdb.Breakpoint* のメソッド), 1490
`enable()` (*cgiib* モジュール), 1125
`enable()` (*gc* モジュール), 1568
`enable()` (*profile.Profile* のメソッド), 1505
`enable_callback_tracebacks()` (*sqlite3* モジュール), 421
`enable_interspersed_args()` (*optparse.OptionParser* のメソッド), 624
`enable_load_extension()` (*sqlite3.Connection* のメソッド), 424
`enable_traversal()` (*ttk.Notebook* のメソッド), 1338
`ENABLE_USER_SITE` (*site* モジュール), 1580
`EnableReflectionKey()` (*winreg* モジュール), 1690
`ENAMETOOLONG` (*errno* モジュール), 711
`ENAVAIL` (*errno* モジュール), 716
`enclose()` (*curses.window* のメソッド), 688
`encode`
 Codecs, 150
`encode()` (*base64* モジュール), 1027

`encode()` (*codecs* モジュール), 150
`encode()` (*codecs.Codec* のメソッド), 156
`encode()` (*codecs.IncrementalEncoder* のメソッド), 157
`encode()` (*email.header.Header* のメソッド), 951
`encode()` (*json.JSONEncoder* のメソッド), 980
`encode()` (*mimetools* モジュール), 1010
`encode()` (*quopri* モジュール), 1032
`encode()` (*str* のメソッド), 47
`encode()` (*uu* モジュール), 1033
`encode()` (*xmlrpclib.Binary* のメソッド), 1248
`encode()` (*xmlrpclib.Boolean* のメソッド), 1246
`encode()` (*xmlrpclib.DateTime* のメソッド), 1247
`encode_7or8bit()` (*email.encoders* モジュール), 957
`encode_base64()` (*email.encoders* モジュール), 957
`encode_noop()` (*email.encoders* モジュール), 957
`encode_quopri()` (*email.encoders* モジュール), 956
`encode_rfc2231()` (*email.utils* モジュール), 960
`encode_threshold`
 (*SimpleXMLRPCServer.SimpleXMLRPCRequestHandler* の属性), 1254
`encoded_header_len()` (*email.charset.Charset* のメソッド), 954
`EncodedFile()` (*codecs* モジュール), 154
`encodePriority()` (*logging.handlers.SysLogHandler* のメソッド), 671
`encodestring()` (*base64* モジュール), 1027
`encodestring()` (*quopri* モジュール), 1032
`encoding`
 base64, 1025
 quoted-printable, 1031
`encoding` (*exceptions.UnicodeError* の属性), 86
`encoding` (*file* の属性), 73
`encoding` (*io.TextIOBase* の属性), 555
`ENCODING` (*tarfile* モジュール), 461
`encodings.idna` (モジュール), 169
`encodings.utf_8_sig` (モジュール), 169
`encodings_map` (*mimetypes* モジュール), 1013
`encodings_map` (*mimetypes.MimeTypes* の属性), 1014
`end` (*exceptions.UnicodeError* の属性), 86
`end()` (*re.MatchObject* のメソッド), 121
`end()` (*xml.etree.ElementTree.TreeBuilder* のメソッド), 1063
`end_fill()` (*turtle* モジュール), 1375
`END_FINALY` (*opcode*), 1652
`end_group()` (*fl.form* のメソッド), 1759
`end_headers()` (*BaseHTTPServer.BaseHTTPRequestHandler* のメソッド), 1222
`end_marker()` (*multifile.MultiFile* のメソッド), 1019
`end_paragraph()` (*formatter.formatter* のメソッド), 1670
`end_poly()` (*turtle* モジュール), 1381
`EndCdataSectionHandler()` (*xml.parsers.expat.xmlparser* のメソッド), 1106
`EndDoctypeDeclHandler()` (*xml.parsers.expat.xmlparser* のメソッド), 1105
`endDocument()` (*xml.sax.handler.ContentHandler* のメソッド), 1091
`endElement()` (*xml.sax.handler.ContentHandler* のメソッド), 1091
`EndElementHandler()` (*xml.parsers.expat.xmlparser* のメソッド), 1105
`endElementNS()` (*xml.sax.handler.ContentHandler* のメソッド), 1092
`endheaders()` (*httplib.HTTPConnection* のメソッド), 1167
`ENDMARKER` (*token* モジュール), 1636
`EndNamespaceDeclHandler()` (*xml.parsers.expat.xmlparser* のメソッド), 1106
`endpick()` (*gl* モジュール), 1765
`endpos` (*re.MatchObject* の属性), 121
`endPrefixMapping()` (*xml.sax.handler.ContentHandler* のメソッド), 1091
`endselect()` (*gl* モジュール), 1765
`endswith()` (*str* のメソッド), 48
`endwin()` (*curses* モジュール), 679
`ENETDOWN` (*errno* モジュール), 714

- ENETRESET (*errno* モジュール), 715
- ENETUNREACH (*errno* モジュール), 715
- ENFILE (*errno* モジュール), 710
- ENOANO (*errno* モジュール), 712
- ENOBUFFS (*errno* モジュール), 715
- ENOCSI (*errno* モジュール), 712
- ENODATA (*errno* モジュール), 712
- ENODEV (*errno* モジュール), 710
- ENOENT (*errno* モジュール), 709
- ENOEXEC (*errno* モジュール), 709
- ENOLCK (*errno* モジュール), 711
- ENOLINK (*errno* モジュール), 713
- ENOMEM (*errno* モジュール), 709
- ENOMSG (*errno* モジュール), 711
- ENONET (*errno* モジュール), 712
- ENOPKG (*errno* モジュール), 712
- ENOPROTOOPT (*errno* モジュール), 714
- ENOSPC (*errno* モジュール), 710
- ENOSR (*errno* モジュール), 712
- ENOSTR (*errno* モジュール), 712
- ENOSYS (*errno* モジュール), 711
- ENOTBLK (*errno* モジュール), 709
- ENOTCONN (*errno* モジュール), 715
- ENOTDIR (*errno* モジュール), 710
- ENOTEMPTY (*errno* モジュール), 711
- ENOTNAM (*errno* モジュール), 716
- ENOTSOCK (*errno* モジュール), 714
- ENOTTY (*errno* モジュール), 710
- ENOTUNIQ (*errno* モジュール), 713
- ensurepip (モジュール), 1524
- enter () (*sched.scheduler* のメソッド), 245
- enterabs () (*sched.scheduler* のメソッド), 245
- entities (*xml.dom.DocumentType* の属性), 1072
- EntityDeclHandler () (*xml.parsers.expat.xmlparser* のメソッド), 1106
- entitydefs (*htmlentitydefs* モジュール), 1047
- EntityResolver (*xml.sax.handler* のクラス), 1088
- Enum (*aetypes* のクラス), 1746
- enum_certificates () (*ssl* モジュール), 891
- enum_crls () (*ssl* モジュール), 891
- enumerate () (*fn* モジュール), 1763
- enumerate () (*threading* モジュール), 764
- enumerate () (組み込み関数), 10
- EnumKey () (*_winreg* モジュール), 1686
- enumsbst () (*aetools* モジュール), 1744
- EnumValue () (*_winreg* モジュール), 1686
- environ (*os* モジュール), 508
- environ (*posix* モジュール), 1698
- environment variables
 - deleting, 513
 - setting, 511
- EnvironmentError, 82
- EnvironmentVarGuard (*test.support* のクラス), 1486
- ENXIO (*errno* モジュール), 709
- eof (*shlex.shlex* の属性), 1313
- EOFError, 82
- EOPNOTSUPP (*errno* モジュール), 714
- EOVERFLOW (*errno* モジュール), 713
- EPERM (*errno* モジュール), 709
- EPFNOSUPPORT (*errno* モジュール), 714
- epilogue (*email.message.Message* の属性), 940
- EPIPE (*errno* モジュール), 710
- epoch, 560
- epoll () (*select* モジュール), 757
- EPROTO (*errno* モジュール), 713
- EPROTONOSUPPORT (*errno* モジュール), 714
- EPROTOTYPE (*errno* モジュール), 714
- eq () (*operator* モジュール), 342
- EQEQUAL (*token* モジュール), 1636
- EQUAL (*token* モジュール), 1636
- ERA (*locale* モジュール), 1301
- ERA_D_FMT (*locale* モジュール), 1301
- ERA_D_T_FMT (*locale* モジュール), 1301
- ERA_T_FMT (*locale* モジュール), 1301
- ERANGE (*errno* モジュール), 711
- erase () (*curses.window* のメソッド), 688
- erasechar () (*curses* モジュール), 679
- EREMCHG (*errno* モジュール), 713
- EREMOTE (*errno* モジュール), 712
- EREMOTEIO (*errno* モジュール), 716
- ERESTART (*errno* モジュール), 714
- erf () (*math* モジュール), 279
- erfc () (*math* モジュール), 279
- EROFS (*errno* モジュール), 710
- ERR (*curses* モジュール), 693
- errcheck (*ctypes.FuncPtr* の属性), 743
- errcode (*xmlrpclib.ProtocolError* の属性), 1249
- errmsg (*xmlrpclib.ProtocolError* の属性), 1249
- errno
 - モジュール, 84, 869
- errno (モジュール), 708
- Error, 379, 476, 483, 494, 1004, 1028, 1031, 1033, 1114, 1268, 1271, 1297, 1723
- error, 117, 127, 263, 406, 408, 409, 411, 415, 441, 507, 636, 677, 757, 777, 869, 1100, 1259, 1263, 1703, 1713, 1717, 1721, 1752, 1767, 1768, 1771
- ERROR (*cd* モジュール), 1753
- error () (*argparse.ArgumentParser* のメソッド), 600
- error () (*logging* モジュール), 650
- error () (*logging.Logger* のメソッド), 640
- error () (*mhlib.Folder* のメソッド), 1009
- error () (*mhlib.MH* のメソッド), 1008
- error () (*urllib2.OpenerDirector* のメソッド), 1153
- error () (*xml.sax.handler.ErrorHandler* のメソッド), 1093
- error_body (*wsgiref.handlers.BaseHandler* の属性), 1136
- error_content_type
 - (*BaseHTTPServer.BaseHTTPRequestHandler* の属性), 1221
- error_headers (*wsgiref.handlers.BaseHandler* の属性), 1136
- error_leader () (*shlex.shlex* のメソッド), 1312
- error_message_format
 - (*BaseHTTPServer.BaseHTTPRequestHandler* の属性), 1221
- error_output () (*wsgiref.handlers.BaseHandler* のメソッド), 1135
- error_perm, 1171
- error_proto, 1171, 1176
- error_reply, 1171
- error_status (*wsgiref.handlers.BaseHandler* の属性), 1136
- error_temp, 1171
- ErrorByteIndex (*xml.parsers.expat.xmlparser* の属性), 1104
- errorcode (*errno* モジュール), 708
- ErrorCode (*xml.parsers.expat.xmlparser* の属性), 1104
- ErrorColumnNumber (*xml.parsers.expat.xmlparser* の属性), 1104
- ErrorHandler (*xml.sax.handler* のクラス), 1088
- ErrorLineNumber (*xml.parsers.expat.xmlparser* の属性), 1104
- Errors
 - logging, 637
- errors (*file* の属性), 73
- errors (*io.TextIOBase* の属性), 556
- errors (*unittest.TestResult* の属性), 1465
- ErrorString () (*xml.parsers.expat* モジュール), 1101
- ERRORTOKEN (*token* モジュール), 1636
- escape (*shlex.shlex* の属性), 1312
- escape () (*cgi* モジュール), 1122
- escape () (*re* モジュール), 117
- escape () (*xml.sax.saxutils* モジュール), 1094
- escapechar (*csv.Dialect* の属性), 476
- escapedquotes (*shlex.shlex* の属性), 1313
- ESHUTDOWN (*errno* モジュール), 715
- ESOCKTNOSUPPORT (*errno* モジュール), 714
- ESPIPE (*errno* モジュール), 710
- ESRCH (*errno* モジュール), 709
- ESRMNT (*errno* モジュール), 713
- ESTALE (*errno* モジュール), 715

- ESTRPIPE (*errno* モジュール), 714
- ETIME (*errno* モジュール), 712
- ETIMEDOUT (*errno* モジュール), 715
- Etiny () (*decimal.Context* のメソッド), 301
- ETOOMANYREFS (*errno* モジュール), 715
- Etop () (*decimal.Context* のメソッド), 301
- ETXTBSY (*errno* モジュール), 710
- EUCLEAN (*errno* モジュール), 715
- EUNATCH (*errno* モジュール), 711
- EUSERS (*errno* モジュール), 714
- eval
 - 組み込み関数, 78, 104, 265, 266, 1624
- eval () (組み込み関数), 11
- Event (*multiprocessing* のクラス), 796
- Event (*threading* のクラス), 775
- event scheduling, 244
- event () (*msilib.Control* のメソッド), 1680
- Event () (*multiprocessing.managers.SyncManager* のメソッド), 804
- events (*widgets*), 1326
- EWOLDBLOCK (*errno* モジュール), 711
- EX.CANTCREAT (*os* モジュール), 535
- EX.CONFIG (*os* モジュール), 536
- EX.DATAERR (*os* モジュール), 534
- EX.IOERR (*os* モジュール), 535
- EX.NOHOST (*os* モジュール), 534
- EX.NOINPUT (*os* モジュール), 534
- EX.NOPERM (*os* モジュール), 536
- EX.NOTFOUND (*os* モジュール), 536
- EX.NOUSER (*os* モジュール), 534
- EX.OK (*os* モジュール), 534
- EX.OSERR (*os* モジュール), 535
- EX.OSFILE (*os* モジュール), 535
- EX.PROTOCOL (*os* モジュール), 536
- EX.SOFTWARE (*os* モジュール), 535
- EX.TEMPFAIL (*os* モジュール), 535
- EX.UNAVAILABLE (*os* モジュール), 535
- EX.USAGE (*os* モジュール), 534
- Example (*doctest* のクラス), 1431
- example (*doctest.DocTestFailure* の属性), 1439
- example (*doctest.UnexpectedException* の属性), 1440
- examples (*doctest.DocTest* の属性), 1431
- exc_clear () (*sys* モジュール), 1529
- exc_info (*doctest.UnexpectedException* の属性), 1440
- exc_info () (*sys* モジュール), 1529
- exc_msg (*doctest.Example* の属性), 1432
- exc_traceback (*sys* モジュール), 1530
- exc_type (*sys* モジュール), 1530
- exc_value (*sys* モジュール), 1530
- excel (*csv* のクラス), 475
- excel_tab (*csv* のクラス), 475
- except
 - 文, 81
- except (2to3 fixer), 1476
- excepthook () (*in module sys*), 1125
- excepthook () (*sys* モジュール), 1528
- Exception, 82
- EXCEPTION (*Tkinter* モジュール), 1329
- exception () (*logging* モジュール), 650
- exception () (*logging.Logger* のメソッド), 641
- exceptions
 - in CGI scripts, 1125
- exceptions (モジュール), 81
- EXDEV (*errno* モジュール), 710
- exec
 - 文, 78
- exec (2to3 fixer), 1476
- exec_prefix (*sys* モジュール), 1530
- EXEC_STMT (*opcode*), 1652
- execfile
 - 組み込み関数, 1582
- execfile (2to3 fixer), 1476
- execfile () (組み込み関数), 11
- execl () (*os* モジュール), 533
- execle () (*os* モジュール), 533
- execlp () (*os* モジュール), 533
- execvp () (*os* モジュール), 533
- executable (*sys* モジュール), 1530
- Execute () (*msilib.View* のメソッド), 1677
- execute () (*sqlite3.Connection* のメソッド), 421
- execute () (*sqlite3.Cursor* のメソッド), 427
- executemany () (*sqlite3.Connection* のメソッド), 422
- executemany () (*sqlite3.Cursor* のメソッド), 428
- executescript () (*sqlite3.Connection* のメソッド), 422
- executescript () (*sqlite3.Cursor* のメソッド), 429
- execv () (*os* モジュール), 533
- execve () (*os* モジュール), 533
- execvp () (*os* モジュール), 533
- execvpe () (*os* モジュール), 533
- ExFileSelectBox (*Tix* のクラス), 1355
- EXFULL (*errno* モジュール), 712
- exists () (*os.path* モジュール), 354
- exists () (*tk.Treeview* のメソッド), 1345
- exit (組み込み変数), 36
- exit () (*argparse.ArgumentParser* のメソッド), 600
- exit () (*sys* モジュール), 1530
- exit () (*thread* モジュール), 777
- exitcode (*multiprocessing.Process* の属性), 787
- exitfunc (2to3 fixer), 1476
- exitfunc (*in sys*), 1560
- exitfunc (*sys* モジュール), 1530
- exitonclick () (*turtle* モジュール), 1389
- exp () (*cmath* モジュール), 281
- exp () (*decimal.Context* のメソッド), 302
- exp () (*decimal.Decimal* のメソッド), 292
- exp () (*math* モジュール), 277
- expand () (*re.MatchObject* のメソッド), 119
- expandtabs (*textwrap.TextWrapper* の属性), 148
- ExpandEnvironmentStrings () (*_winreg* モジュール), 1687
- expandNode () (*xml.dom.pulldom.DOMEventStream* のメソッド), 1085
- expandtabs () (*str* のメソッド), 48
- expandtabs () (*string* モジュール), 104
- expanduser () (*os.path* モジュール), 354
- expandvars () (*os.path* モジュール), 354
- Expat, 1100
- ExpatriError, 1100
- expect () (*telnetlib.Telnet* のメソッド), 1199
- expectedFailure () (*unittest* モジュール), 1452
- expectedFailures (*unittest.TestResult* の属性), 1466
- expires (*cookielib.Cookie* の属性), 1237
- expml () (*math* モジュール), 277
- expovariate () (*random* モジュール), 321
- expr () (*parser* モジュール), 1622
- expression, 1782
- expunge () (*imaplib.IMAP4* のメソッド), 1180
- extend () (*array.array* のメソッド), 237
- extend () (*collections.deque* のメソッド), 214
- extend () (*list method*), 58
- extend () (*xml.etree.ElementTree.Element* のメソッド), 1060
- extend.path () (*pkgutil* モジュール), 1612
- extended slice
 - assignment, 58
 - operation, 46
- EXTENDED_ARG (*opcode*), 1656
- ExtendedContext (*decimal* のクラス), 299
- extendleft () (*collections.deque* のメソッド), 214
- extension module, 1782
- extensions_map
 - (*SimpleHTTPServer.SimpleHTTPRequestHandler* の属性), 1224
- External Data Representation, 386, 491
- external_attr (*zipfile.ZipInfo* の属性), 457
- ExternalClashError, 1004
- ExternalEntityParserCreate ()
 - (*xml.parsers.expat.xmlparser* のメソッド), 1102

- ExternalEntityRefHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1107
- extra (*zipfile.ZipInfo* の属性), 457
- extract() (*tarfile.TarFile* のメソッド), 463
- extract() (*zipfile.ZipFile* のメソッド), 453
- extract_cookies() (*cookielib.CookieJar* のメソッド), 1229
- extract_stack() (*traceback* モジュール), 1563
- extract_tb() (*traceback* モジュール), 1563
- extract.version (*zipfile.ZipInfo* の属性), 457
- extractall() (*tarfile.TarFile* のメソッド), 463
- extractall() (*zipfile.ZipFile* のメソッド), 453
- ExtractError, 460
- extractfile() (*tarfile.TarFile* のメソッド), 464
- extsep (*os* モジュール), 544
- F_BAVAIL (*statvfs* モジュール), 366
- F_BFREE (*statvfs* モジュール), 366
- F_BLOCKS (*statvfs* モジュール), 366
- F_BSIZE (*statvfs* モジュール), 366
- F_FAVAIL (*statvfs* モジュール), 366
- F_FFREE (*statvfs* モジュール), 366
- F_FILES (*statvfs* モジュール), 366
- F_FLAG (*statvfs* モジュール), 366
- F_FRSIZE (*statvfs* モジュール), 366
- F_NAMEMAX (*statvfs* モジュール), 366
- F_OK (*os* モジュール), 521
- fabs() (*math* モジュール), 275
- factorial() (*math* モジュール), 275
- fail() (*unittest.TestCase* のメソッド), 1460
- failfast (*unittest.TestResult* の属性), 1466
- failureException (*unittest.TestCase* の属性), 1460
- failures (*unittest.TestResult* の属性), 1466
- False, 38, 79
- false, 37
- False (*Built-in object*), 37
- False (組み込み変数), 35
- family (*socket.socket* の属性), 880
- FancyURLOpener (*urllib* のクラス), 1143
- fatalError() (*xml.sax.handler.ErrorHandler* のメソッド), 1093
- Fault (*xmlrpclib* のクラス), 1248
- faultCode (*xmlrpclib.Fault* の属性), 1248
- faultString (*xmlrpclib.Fault* の属性), 1248
- fchdir() (*os* モジュール), 521
- fchmod() (*os* モジュール), 516
- fchown() (*os* モジュール), 516
- FCICreate() (*msilib* モジュール), 1675
- fcntl
- モジュール, 70
- fcntl (モジュール), 1706
- fcntl() (*fcntl* モジュール), 1706
- fcntl() (*in module fcntl*), 1711
- fd() (*turtle* モジュール), 1363
- fdasync() (*os* モジュール), 516
- fdopen() (*os* モジュール), 513
- Feature (*msilib* のクラス), 1680
- feature_external_ges (*xml.sax.handler* モジュール), 1089
- feature_external_pes (*xml.sax.handler* モジュール), 1089
- feature_namespace_prefixes (*xml.sax.handler* モジュール), 1088
- feature_namespaces (*xml.sax.handler* モジュール), 1088
- feature_string_interning (*xml.sax.handler* モジュール), 1088
- feature_validation (*xml.sax.handler* モジュール), 1089
- feed() (*email.parser.FeedParser* のメソッド), 942
- feed() (*HTMLParser.HTMLParser* のメソッド), 1037
- feed() (*sgmlib.SGMLParser* のメソッド), 1042
- feed() (*xml.etree.ElementTree.XMLParser* のメソッド), 1064
- feed() (*xml.sax.xmlreader.IncrementalParser* のメソッド), 1097
- FeedParser (*email.parser* のクラス), 942
- fetch() (*imaplib.IMAP4* のメソッド), 1181
- Fetch() (*msilib.View* のメソッド), 1677
- fetchall() (*sqlite3.Cursor* のメソッド), 430
- fetchmany() (*sqlite3.Cursor* のメソッド), 430
- fetchone() (*sqlite3.Cursor* のメソッド), 430
- fflags (*select.kevent* の属性), 762
- field_size_limit() (*csv* モジュール), 473
- fieldnames (*csv.csvreader* の属性), 477
- fields (*uuid.UUID* の属性), 1201
- fifo (*asyncchat* のクラス), 928
- file
- .ini, 481
- .pdbrc, 1498
- .pythonrc.py, 1582
- byte-code, 1601, 1604, 1643
- configuration, 481
- copying, 376
- debugger configuration, 1498
- descriptor, 70
- large files, 1697
- mime.types, 1013
- path configuration, 1579
- plist, 495
- temporary, 369
- user configuration, 1582
- オブジェクト, 69
- 組み込み関数, 69
- file (*pyclbr.Class* の属性), 1642
- file (*pyclbr.Function* の属性), 1642
- file ...
- compileall command line option, 1644
- file control
- UNIX, 1706
- file name
- temporary, 369
- file object, 1782
- file object
- POSIX, 1711
- file() (*posixfile.posixfile* のメソッド), 1712
- file() (組み込み関数), 12
- file-like object, 1782
- file_dispatcher (*asyncore* のクラス), 924
- file.open() (*urllib2.FileHandler* のメソッド), 1158
- file.size (*zipfile.ZipInfo* の属性), 457
- file.wrapper (*asyncore* のクラス), 924
- filecmp (モジュール), 367
- fileConfig() (*logging.config* モジュール), 654
- FileCookieJar (*cookielib* のクラス), 1228
- FileEntry (*Tix* のクラス), 1355
- FileHandler (*logging* のクラス), 666
- FileHandler (*urllib2* のクラス), 1150
- FileInput (*fileinput* のクラス), 360
- fileinput (モジュール), 358
- FileIO (*io* のクラス), 553
- filelineno() (*fileinput* モジュール), 359
- filename (*cookielib.FileCookieJar* の属性), 1231
- filename (*doctest.DocTest* の属性), 1431
- filename (*zipfile.ZipInfo* の属性), 456
- filename() (*fileinput* モジュール), 359
- filename_only (*tabnanny* モジュール), 1641
- filenames
- pathname expansion, 373
- wildcard expansion, 374
- fileno() (*Connection* のメソッド), 794
- fileno() (*file* のメソッド), 70
- fileno() (*fileinput* モジュール), 359
- fileno() (*hotshot.Profile* のメソッド), 1512
- fileno() (*httplib.HTTPResponse* のメソッド), 1167
- fileno() (*io.IOBase* のメソッド), 549
- fileno() (*ossaudiodev.oss_audio.device* のメソッド), 1279
- fileno() (*ossaudiodev.oss_mixer.device* のメソッド), 1282
- fileno() (*select.epoll* のメソッド), 759
- fileno() (*select.kqueue* のメソッド), 761
- fileno() (*socket.socket* のメソッド), 877
- fileno() (*SocketServer.BaseServer* のメソッド), 1212
- fileno() (*telnetlib.Telnet* のメソッド), 1199

- `fileopen()` (*posixfile* モジュール), 1711
- `FileSelectBox` (*Tix* のクラス), 1355
- `FileType` (*argparse* のクラス), 596
- `FileType` (*types* モジュール), 260
- `FileWrapper` (*wsgiref.util* のクラス), 1128
- `fill()` (*textwrap* モジュール), 147
- `fill()` (*textwrap.TextWrapper* のメソッド), 150
- `fill()` (*turtle* モジュール), 1375
- `fillcolor()` (*turtle* モジュール), 1374
- `filter` (*2to3 fixer*), 1476
- `Filter` (*logging* のクラス), 645
- `filter` (*select.kevent* の属性), 761
- `filter()` (*curses* モジュール), 679
- `filter()` (*fnmatch* モジュール), 375
- `filter()` (*future.builtins* モジュール), 1547
- `filter()` (*logging.Filter* のメソッド), 645
- `filter()` (*logging.Handler* のメソッド), 643
- `filter()` (*logging.Logger* のメソッド), 641
- `filter()` (組み込み関数), 12
- `filterwarnings()` (*warnings* モジュール), 1553
- `find()` (*doctest.DocTestFinder* のメソッド), 1433
- `find()` (*gettext* モジュール), 1287
- `find()` (*mmap.mmap* のメソッド), 845
- `find()` (*str* のメソッド), 48
- `find()` (*string* モジュール), 104
- `find()` (*xml.etree.ElementTree.Element* のメソッド), 1060
- `find()` (*xml.etree.ElementTree.ElementTree* のメソッド), 1061
- `find.first()` (*fl.form* のメソッド), 1759
- `find.global()` (*pickle protocol*), 396
- `find.last()` (*fl.form* のメソッド), 1759
- `find.library()` (*ctypes.util* モジュール), 747
- `find.loader()` (*pkgutil* モジュール), 1613
- `find.longest_match()` (*difflib.SequenceMatcher* のメソッド), 138
- `find.module()` (*imp* モジュール), 1601
- `find.module()` (*imp.NullImporter* のメソッド), 1605
- `find.module()` (*zipimport.zipimporter* のメソッド), 1611
- `find.msvcr()` (*ctypes.util* モジュール), 747
- `find.user.password()` (*urllib2.HTTPPasswordMgr* のメソッド), 1157
- `findall()` (*re* モジュール), 115
- `findall()` (*re.RegexObject* のメソッド), 118
- `findall()` (*xml.etree.ElementTree.Element* のメソッド), 1060
- `findall()` (*xml.etree.ElementTree.ElementTree* のメソッド), 1061
- `findCaller()` (*logging.Logger* のメソッド), 641
- `finder`, 1782
- `findertools` (モジュール), 1726
- `findfactor()` (*audioop* モジュール), 1260
- `findfile()` (*test.support* モジュール), 1483
- `findfit()` (*audioop* モジュール), 1260
- `findfont()` (*fm* モジュール), 1763
- `finditer()` (*re* モジュール), 116
- `finditer()` (*re.RegexObject* のメソッド), 118
- `findlabels()` (*dis* モジュール), 1647
- `findlinestarts()` (*dis* モジュール), 1647
- `findmatch()` (*mailcap* モジュール), 983
- `findmax()` (*audioop* モジュール), 1260
- `findtext()` (*xml.etree.ElementTree.Element* のメソッド), 1060
- `findtext()` (*xml.etree.ElementTree.ElementTree* のメソッド), 1062
- `finish()` (*SocketServer.BaseRequestHandler* のメソッド), 1215
- `finish.request()` (*SocketServer.BaseServer* のメソッド), 1214
- `first()` (*asynchat.fifo* のメソッド), 928
- `first()` (*bsddb.bsddbobject* のメソッド), 414
- `first()` (*dbhash.dbhash* のメソッド), 411
- `firstChild` (*xml.dom.Node* の属性), 1069
- `firstkey()` (*gdbm* モジュール), 410
- `firstweekday()` (*calendar* モジュール), 208
- `fix()` (*fppformat* モジュール), 174
- `fix.missing.locations()` (*ast* モジュール), 1631
- `fix.sentence.endings` (*textwrap.TextWrapper* の属性), 149
- `FL` (モジュール), 1762
- `fl` (モジュール), 1756
- `flag.bits` (*zipfile.ZipInfo* の属性), 457
- `flags` (*re.RegexObject* の属性), 119
- `flags` (*select.kevent* の属性), 762
- `flags` (*sys* モジュール), 1531
- `flags()` (*posixfile.posixfile* のメソッド), 1711
- `flash()` (*curses* モジュール), 679
- `flatten()` (*email.generator.Generator* のメソッド), 945
- `flattening`
 - objects, 385
- `float`
 - 組み込み関数, 39, 103
- `float` (組み込みクラス), 12
- `float.info` (*sys* モジュール), 1531
- `float.repr_style` (*sys* モジュール), 1532
- `floating point`
 - literals, 39
 - オブジェクト, 39
- `FloatingPointError`, 83, 1584
- `FloatType` (*types* モジュール), 259
- `flock()` (*fcntl* モジュール), 1707
- `floor division`, 1782
- `floor()` (*in module math*), 40
- `floor()` (*math* モジュール), 275
- `floordiv()` (*operator* モジュール), 343
- `flp` (モジュール), 1762
- `flush()` (*bz2.BZ2Compressor* のメソッド), 449
- `flush()` (*file* のメソッド), 70
- `flush()` (*formatter.writer* のメソッド), 1672
- `flush()` (*io.BufferedWriter* のメソッド), 554
- `flush()` (*io.IOBase* のメソッド), 549
- `flush()` (*logging.Handler* のメソッド), 643
- `flush()` (*logging.handlers.BufferingHandler* のメソッド), 674
- `flush()` (*logging.handlers.MemoryHandler* のメソッド), 675
- `flush()` (*logging.StreamHandler* のメソッド), 665
- `flush()` (*mailbox.Mailbox* のメソッド), 987
- `flush()` (*mailbox.Maildir* のメソッド), 989
- `flush()` (*mailbox.MH* のメソッド), 992
- `flush()` (*mmap.mmap* のメソッド), 845
- `flush()` (*zlib.Compress* のメソッド), 444
- `flush()` (*zlib.Decompress* のメソッド), 445
- `flush.softspace()` (*formatter.formatter* のメソッド), 1670
- `flushheaders()` (*MimeWriter.MimeWriter* のメソッド), 1016
- `flushinp()` (*curses* モジュール), 679
- `FlushKey()` (*winreg* モジュール), 1687
- `fm` (モジュール), 1763
- `fma()` (*decimal.Context* のメソッド), 302
- `fma()` (*decimal.Decimal* のメソッド), 292
- `fmod()` (*math* モジュール), 275
- `fnmatch` (モジュール), 374
- `fnmatch()` (*fnmatch* モジュール), 375
- `fnmatchcase()` (*fnmatch* モジュール), 375
- `focus()` (*ttk.Treeview* のメソッド), 1345
- `Folder` (*mhlib* のクラス), 1008
- `Font Manager`, IRIS, 1763
- `fontpath()` (*fm* モジュール), 1763
- `FOR_ITER` (*opcode*), 1654
- `forget()` (*test.support* モジュール), 1483
- `forget()` (*ttk.Notebook* のメソッド), 1338
- `fork()` (*os* モジュール), 536
- `fork()` (*pty* モジュール), 1705
- `ForkingMixIn` (*SocketServer* のクラス), 1211
- `ForkingTCPServer` (*SocketServer* のクラス), 1211
- `ForkingUDPServer` (*SocketServer* のクラス), 1211
- `forkpty()` (*os* モジュール), 536
- `Form` (*Tix* のクラス), 1356
- `format`
 - str, 13
- `format` (*memoryview* の属性), 75
- `format` (*struct.Struct* の属性), 132
- `format()` (*locale* モジュール), 1303
- `format()` (*logging.Formatter* のメソッド), 644

- `format()` (`logging.Handler` のメソッド), 643
- `format()` (`pprint.PrettyPrinter` のメソッド), 266
- `format()` (`str` のメソッド), 49
- `format()` (`string.Formatter` のメソッド), 92
- `format()` (組み込み関数), 13
- `format_exc()` (`traceback` モジュール), 1562
- `format_exception()` (`traceback` モジュール), 1563
- `format_exception_only()` (`traceback` モジュール), 1563
- `format_field()` (`string.Formatter` のメソッド), 94
- `format_help()` (`argparse.ArgumentParser` のメソッド), 599
- `format_list()` (`traceback` モジュール), 1563
- `format_stack()` (`traceback` モジュール), 1563
- `format_stack_entry()` (`bdb.Bdb` のメソッド), 1494
- `format_string()` (`locale` モジュール), 1303
- `format_tb()` (`traceback` モジュール), 1563
- `format_usage()` (`argparse.ArgumentParser` のメソッド), 599
- `formataddr()` (`email.utils` モジュール), 959
- `formatargspec()` (`inspect` モジュール), 1577
- `formatargvalues()` (`inspect` モジュール), 1577
- `formatdate()` (`email.utils` モジュール), 959
- `FormatError`, 1004
- `FormatError()` (`ctypes` モジュール), 748
- `formatException()` (`logging.Formatter` のメソッド), 644
- `formatmonth()` (`calendar.HTMLCalendar` のメソッド), 207
- `formatmonth()` (`calendar.TextCalendar` のメソッド), 207
- `formatter`
 - モジュール, 1045
- `formatter` (`htmlib.HTMLParser` の属性), 1046
- `Formatter` (`logging` のクラス), 644
- `Formatter` (`string` のクラス), 92
- `formatter` (モジュール), 1669
- `formatTime()` (`logging.Formatter` のメソッド), 644
- `formatting`, `string (%)`, 55
- `formatwarning()` (`warnings` モジュール), 1553
- `formatyear()` (`calendar.HTMLCalendar` のメソッド), 207
- `formatyear()` (`calendar.TextCalendar` のメソッド), 207
- `formatyearpage()` (`calendar.HTMLCalendar` のメソッド), 207
- `FORMS Library`, 1756
- `forward()` (`turtle` モジュール), 1363
- `found_terminator()` (`asynchat.async_chat` のメソッド), 927
- `fp` (`rfc822.Message` の属性), 1024
- `fpathconf()` (`os` モジュール), 516
- `fpectl` (モジュール), 1583
- `fpformat` (モジュール), 174
- `Fraction` (`fractions` のクラス), 315
- `fractions` (モジュール), 315
- `frame` (`ScrolledText.ScrolledText` の属性), 1359
- `FrameType` (`types` モジュール), 261
- `FrameWork`
 - モジュール, 1748
- `FrameWork` (モジュール), 1730
- `freeze_form()` (`fl.form` のメソッド), 1759
- `freeze_support()` (`multiprocessing` モジュール), 793
- `frexp()` (`math` モジュール), 275
- `from_address()` (`ctypes._CData` のメソッド), 750
- `from_buffer()` (`ctypes._CData` のメソッド), 749
- `from_buffer_copy()` (`ctypes._CData` のメソッド), 750
- `from_decimal()` (`fractions.Fraction` のメソッド), 317
- `from_float()` (`decimal.Decimal` のメソッド), 292
- `from_float()` (`fractions.Fraction` のメソッド), 316
- `from_iterable()` (`itertools.chain` のクラスメソッド), 325
- `from_param()` (`ctypes._CData` のメソッド), 750
- `from_splittable()` (`email.charset.Charset` のメソッド), 954
- `frombuf()` (`tarfile.TarInfo` のメソッド), 465
- `fromchild` (`popen2.Popen3` の属性), 920
- `fromfd()` (`select.epoll` のメソッド), 759
- `fromfd()` (`select.kqueue` のメソッド), 761
- `fromfd()` (`socket` モジュール), 874
- `fromfile()` (`array.array` のメソッド), 238
- `fromhex()` (`float` のメソッド), 43
- `fromkeys()` (`collections.Counter` のメソッド), 212
- `fromkeys()` (`dict` のメソッド), 65
- `fromlist()` (`array.array` のメソッド), 238
- `fromordinal()` (`datetime.date` のクラスメソッド), 180
- `fromordinal()` (`datetime.datetime` のクラスメソッド), 186
- `fromstring()` (`array.array` のメソッド), 238
- `fromstring()` (`xml.etree.ElementTree` モジュール), 1057
- `fromstringlist()` (`xml.etree.ElementTree` モジュール), 1057
- `fromtarfile()` (`tarfile.TarInfo` のメソッド), 466
- `fromtimestamp()` (`datetime.date` のクラスメソッド), 180
- `fromtimestamp()` (`datetime.datetime` のクラスメソッド), 185
- `fromunicode()` (`array.array` のメソッド), 238
- `fromutc()` (`datetime.tzinfo` のメソッド), 197
- `frozenset` (組み込みクラス), 60
- `fstat()` (`os` モジュール), 517
- `fstatvfs()` (`os` モジュール), 517
- `fsum()` (`math` モジュール), 276
- `fsync()` (`os` モジュール), 517
- `FTP`, 1145
 - `ftplib` (standard module), 1169
 - protocol, 1144, 1169
- `FTP` (`ftplib` のクラス), 1170
- `ftp_open()` (`urllib2.FTPHandler` のメソッド), 1159
- `ftp_proxy`, 1139
- `FTP.TLS` (`ftplib` のクラス), 1170
- `FTPHandler` (`urllib2` のクラス), 1150
- `ftplib` (モジュール), 1169
- `ftpmirror.py`, 1171
- `ftruncate()` (`os` モジュール), 517
- `Full`, 248
- `full()` (`multiprocessing.Queue` のメソッド), 790
- `full()` (`Queue.Queue` のメソッド), 248
- `func` (`functools.partial` の属性), 341
- `func.code` (function object attribute), 78
- `funcattrs` (2to3 fixer), 1476
- `Function` (`symtable` のクラス), 1634
- `function()` (`new` モジュール), 262
- `FunctionTestCase` (`unittest` のクラス), 1461
- `FunctionType` (`types` モジュール), 260
- `functools` (モジュール), 339
- `funny_files` (`filecmp.dircmp` の属性), 369
- `future` (2to3 fixer), 1476
- `future.builtins` (モジュール), 1547
- `FutureWarning`, 87
-
- `G.722`, 1267
- `gaierror`, 870
- `gamma()` (`math` モジュール), 279
- `gammavariate()` (`random` モジュール), 321
- `garbage` (`gc` モジュール), 1570
- `garbage collection`, 1783
- `gather()` (`curses.textpad.Textbox` のメソッド), 700
- `gauss()` (`random` モジュール), 321
- `gc` (モジュール), 1568
- `gcd()` (`fractions` モジュール), 317
- `gdbm`
 - モジュール, 401, 405
- `gdbm` (モジュール), 409
- `ge()` (`operator` モジュール), 342
- `gen_uuid()` (`msilib` モジュール), 1676
- `generate_tokens()` (`tokenize` モジュール), 1638
- `generator`, 1783
- `generator`, 1783
- `Generator` (`email.generator` のクラス), 945
- `generator expression`, 1783
- `generator expression`, 1783
- `GeneratorExit`, 83
- `GeneratorType` (`types` モジュール), 260
- `generic_visit()` (`ast.NodeVisitor` のメソッド), 1632
- `genops()` (`pickletools` モジュール), 1657
- `gensuitemodule` (モジュール), 1742
- `get()` (`ConfigParser.ConfigParser` のメソッド), 486
- `get()` (`ConfigParser.RawConfigParser` のメソッド), 485
- `get()` (`dict` のメソッド), 65
- `get()` (`email.message.Message` のメソッド), 935

`get()` (*mailbox.Mailbox* のメソッド), 986
`get()` (*multiprocessing.multiprocessing.queues.SimpleQueue* のメソッド), 792
`get()` (*multiprocessing.pool.AsyncResult* のメソッド), 811
`get()` (*multiprocessing.Queue* のメソッド), 791
`get()` (*ossaudiodev.oss_mixer_device* のメソッド), 1283
`get()` (*Queue.Queue* のメソッド), 249
`get()` (*rfc822.Message* のメソッド), 1023
`get()` (*ttk.Combobox* のメソッド), 1335
`get()` (*webbrowser* モジュール), 1114
`get()` (*xml.etree.ElementTree.Element* のメソッド), 1059
`get_all()` (*email.message.Message* のメソッド), 936
`get_all()` (*wsgiref.headers.Headers* のメソッド), 1129
`get_all_breaks()` (*bdb.Bdb* のメソッド), 1494
`get_app()` (*wsgiref.simple_server.WSGIServer* のメソッド), 1131
`get_archive_formats()` (*shutil* モジュール), 381
`get_begidx()` (*readline* モジュール), 849
`get_body_encoding()` (*email.charset.Charset* のメソッド), 954
`get_boundary()` (*email.message.Message* のメソッド), 939
`get_break()` (*bdb.Bdb* のメソッド), 1493
`get_breaks()` (*bdb.Bdb* のメソッド), 1493
`get_buffer()` (*xdrlib.Packer* のメソッド), 492
`get_buffer()` (*xdrlib.Unpacker* のメソッド), 493
`get_ca_certs()` (*ssl.SSLContext* のメソッド), 902
`get_channel_binding()` (*ssl.SSLSocket* のメソッド), 899
`get_charset()` (*email.message.Message* のメソッド), 934
`get_charsets()` (*email.message.Message* のメソッド), 939
`get_children()` (*symtable.SymbolTable* のメソッド), 1634
`get_children()` (*ttk.Treeview* のメソッド), 1344
`get_close_matches()` (*difflib* モジュール), 135
`get_code()` (*importlib.BuiltinImporter* のメソッド), 1607
`get_code()` (*importlib.Importer* のメソッド), 1606
`get_code()` (*zipimport.zipimporter* のメソッド), 1611
`get_completer()` (*readline* モジュール), 849
`get_completer_delims()` (*readline* モジュール), 849
`get_completion_type()` (*readline* モジュール), 849
`get_config_h_filename()` (*sysconfig* モジュール), 1546
`get_config_var()` (*sysconfig* モジュール), 1543
`get_config_vars()` (*sysconfig* モジュール), 1543
`get_content_charset()` (*email.message.Message* のメソッド), 939
`get_content_maintype()` (*email.message.Message* のメソッド), 937
`get_content_subtype()` (*email.message.Message* のメソッド), 937
`get_content_type()` (*email.message.Message* のメソッド), 936
`get_count()` (*gc* モジュール), 1569
`get_current_history_length()` (*readline* モジュール), 848
`get_data()` (*pkgutil* モジュール), 1615
`get_data()` (*urllib2.Request* のメソッド), 1151
`get_data()` (*zipimport.zipimporter* のメソッド), 1611
`get_date()` (*mailbox.MaildirMessage* のメソッド), 996
`get_debug()` (*gc* モジュール), 1569
`get_default()` (*argparse.ArgumentParser* のメソッド), 598
`get_default_domain()` (*nis* モジュール), 1717
`get_default_type()` (*email.message.Message* のメソッド), 937
`get_default_verify_paths()` (*ssl* モジュール), 891
`get_dialect()` (*csv* モジュール), 473
`get_directory()` (*fl* モジュール), 1757
`get_docstring()` (*ast* モジュール), 1631
`get_doctest()` (*doctest.DocTestParser* のメソッド), 1433
`get_endidx()` (*readline* モジュール), 849
`get_environ()` (*wsgiref.simple_server.WSGIRequestHandler* のメソッド), 1131
`get_errno()` (*ctypes* モジュール), 748
`get_examples()` (*doctest.DocTestParser* のメソッド), 1434
`get_field()` (*string.Formatter* のメソッド), 93
`get_file()` (*mailbox.Babyl* のメソッド), 993
`get_file()` (*mailbox.Mailbox* のメソッド), 986
`get_file()` (*mailbox.Maildir* のメソッド), 989

`get_file()` (*mailbox.mbox* のメソッド), 990
`get_file()` (*mailbox.MH* のメソッド), 992
`get_file()` (*mailbox.MMDF* のメソッド), 994
`get_file_breaks()` (*bdb.Bdb* のメソッド), 1493
`get_filename()` (*email.message.Message* のメソッド), 939
`get_filename()` (*fl* モジュール), 1757
`get_filename()` (*zipimport.zipimporter* のメソッド), 1611
`get_flags()` (*mailbox.MaildirMessage* のメソッド), 995
`get_flags()` (*mailbox.mboxMessage* のメソッド), 997
`get_flags()` (*mailbox.MMDFMessage* のメソッド), 1002
`get_folder()` (*mailbox.Maildir* のメソッド), 989
`get_folder()` (*mailbox.MH* のメソッド), 991
`get_frees()` (*symtable.Function* のメソッド), 1634
`get_from()` (*mailbox.mboxMessage* のメソッド), 997
`get_from()` (*mailbox.MMDFMessage* のメソッド), 1002
`get_full_url()` (*urllib2.Request* のメソッド), 1151
`get_globals()` (*symtable.Function* のメソッド), 1634
`get_grouped_opcodes()` (*difflib.SequenceMatcher* のメソッド), 140
`get_header()` (*urllib2.Request* のメソッド), 1152
`get_history_item()` (*readline* モジュール), 848
`get_history_length()` (*readline* モジュール), 847
`get_host()` (*urllib2.Request* のメソッド), 1152
`get_id()` (*symtable.SymbolTable* のメソッド), 1633
`get_ident()` (*thread* モジュール), 777
`get_identifiers()` (*symtable.SymbolTable* のメソッド), 1634
`get_importer()` (*pkgutil* モジュール), 1613
`get_info()` (*mailbox.MaildirMessage* のメソッド), 996
`GET_ITER(opcode)`, 1648
`get_labels()` (*mailbox.Babyl* のメソッド), 993
`get_labels()` (*mailbox.BabylMessage* のメソッド), 1001
`get_last_error()` (*ctypes* モジュール), 748
`get_line_buffer()` (*readline* モジュール), 847
`get_lineno()` (*symtable.SymbolTable* のメソッド), 1633
`get_loader()` (*pkgutil* モジュール), 1614
`get_locals()` (*symtable.Function* のメソッド), 1634
`get_logger()` (*multiprocessing* モジュール), 815
`get_magic()` (*imp* モジュール), 1601
`get_makefile_filename()` (*sysconfig* モジュール), 1546
`get_matching_blocks()` (*difflib.SequenceMatcher* のメソッド), 139
`get_message()` (*mailbox.Mailbox* のメソッド), 986
`get_method()` (*urllib2.Request* のメソッド), 1151
`get_methods()` (*symtable.Class* のメソッド), 1634
`get_mouse()` (*fl* モジュール), 1758
`get_name()` (*symtable.Symbol* のメソッド), 1634
`get_name()` (*symtable.SymbolTable* のメソッド), 1633
`get_namespace()` (*symtable.Symbol* のメソッド), 1635
`get_namespaces()` (*symtable.Symbol* のメソッド), 1635
`get_nonstandard_attr()` (*cookielib.Cookie* のメソッド), 1237
`get_nowait()` (*multiprocessing.Queue* のメソッド), 791
`get_nowait()` (*Queue.Queue* のメソッド), 249
`get_objects()` (*gc* モジュール), 1569
`get_opcodes()` (*difflib.SequenceMatcher* のメソッド), 139
`get_option()` (*optparse.OptionParser* のメソッド), 624
`get_option_group()` (*optparse.OptionParser* のメソッド), 612
`get_origin_req_host()` (*urllib2.Request* のメソッド), 1152
`get_osfhandle()` (*msvcrt* モジュール), 1683
`get_output_charset()` (*email.charset.Charset* のメソッド), 954
`get_param()` (*email.message.Message* のメソッド), 937
`get_parameters()` (*symtable.Function* のメソッド), 1634
`get_params()` (*email.message.Message* のメソッド), 937
`get_path()` (*sysconfig* モジュール), 1544
`get_path_names()` (*sysconfig* モジュール), 1544
`get_paths()` (*sysconfig* モジュール), 1544
`get_pattern()` (*fl* モジュール), 1757
`get_payload()` (*email.message.Message* のメソッド), 933
`get_platform()` (*sysconfig* モジュール), 1545
`get_poly()` (*turtle* モジュール), 1381
`get_position()` (*xdrlib.Unpacker* のメソッド), 493

get.python.version() (sysconfig モジュール), 1545
 get.recsrc() (ossaudiodev.oss.mixer_device のメソッド), 1283
 get.referents() (gc モジュール), 1569
 get.referrers() (gc モジュール), 1569
 get.request() (SocketServer.BaseServer のメソッド), 1214
 get.rgbmode() (fl モジュール), 1757
 get.scheme() (wsgiref.handlers.BaseHandler のメソッド), 1135
 get.scheme.names() (sysconfig モジュール), 1544
 get.selector() (urllib2.Request のメソッド), 1152
 get.sequences() (mailbox.MH のメソッド), 991
 get.sequences() (mailbox.MHMessage のメソッド), 999
 get.server() (multiprocessing.managers.BaseManager のメソッド), 802
 get.server.certificate() (ssl モジュール), 890
 get.socket() (telnetlib.Telnet のメソッド), 1199
 get.source() (zipimport.zipimporter のメソッド), 1611
 get.stack() (bdb.Bdb のメソッド), 1494
 get.starttagtext() (HTMLParser.HTMLParser のメソッド), 1037
 get.starttagtext() (sgmlib.SGMLParser のメソッド), 1042
 get.stderr() (wsgiref.handlers.BaseHandler のメソッド), 1134
 get.stderr() (wsgiref.simple_server.WSGIRequestHandler のメソッド), 1131
 get.stdin() (wsgiref.handlers.BaseHandler のメソッド), 1134
 get.string() (mailbox.Mailbox のメソッド), 986
 get.subdir() (mailbox.MaildirMessage のメソッド), 995
 get.suffixes() (imp モジュール), 1601
 get.symbols() (symtable.SymbolTable のメソッド), 1634
 get.terminator() (asynchat.async_chat のメソッド), 927
 get.threshold() (gc モジュール), 1569
 get.token() (shlex.shlex のメソッド), 1311
 get.type() (symtable.SymbolTable のメソッド), 1633
 get.type() (urllib2.Request のメソッド), 1152
 get.unixfrom() (email.message.Message のメソッド), 933
 get.usage() (optparse.OptionParser のメソッド), 626
 get.value() (string.Formatter のメソッド), 93
 get.version() (optparse.OptionParser のメソッド), 613
 get.visible() (mailbox.BabylMessage のメソッド), 1001
 getabouttext() (FrameWork.Application のメソッド), 1732
 getacl() (imaplib.IMAP4 のメソッド), 1181
 getaddr() (rfc822.Message のメソッド), 1023
 getaddresses() (email.utils モジュール), 959
 getaddrinfo() (socket モジュール), 871
 getaddrlist() (rfc822.Message のメソッド), 1024
 getallmatchingheaders() (rfc822.Message のメソッド), 1023
 getannotation() (imaplib.IMAP4 のメソッド), 1181
 getargspec() (inspect モジュール), 1576
 GetArgv() (EasyDialogs モジュール), 1727
 getargvalues() (inspect モジュール), 1576
 getatime() (os.path モジュール), 355
 getattr() (組み込み関数), 13
 getAttribute() (xml.dom.Element のメソッド), 1073
 getAttributeNode() (xml.dom.Element のメソッド), 1073
 getAttributeNodeNS() (xml.dom.Element のメソッド), 1074
 getAttributeNS() (xml.dom.Element のメソッド), 1073
 GetBase() (xml.parsers.expat.xmlparser のメソッド), 1102
 getbegyx() (curses.window のメソッド), 688
 getbkgd() (curses.window のメソッド), 688
 getboolean() (ConfigParser.RawConfigParser のメソッド), 485
 getByteStream() (xml.sax.xmlreader.InputSource のメソッド), 1099
 getcallargs() (inspect モジュール), 1577
 getcanvas() (turtle モジュール), 1388
 getcaps() (mailcap モジュール), 983
 getch() (curses.window のメソッド), 688
 getch() (msvcrt モジュール), 1683
 getCharacterStream() (xml.sax.xmlreader.InputSource のメソッド), 1099

getche() (msvcrt モジュール), 1683
 getcheckinterval() (sys モジュール), 1532
 getChild() (logging.Logger のメソッド), 639
 getChildNodes() (compiler.ast.Node のメソッド), 1661
 getChildren() (compiler.ast.Node のメソッド), 1661
 getchildren() (xml.etree.ElementTree.Element のメソッド), 1060
 getclasstree() (inspect モジュール), 1576
 GetColor() (ColorPicker モジュール), 1739
 GetColumnInfo() (msilib.View のメソッド), 1677
 getColumnNumber() (xml.sax.xmlreader.Locator のメソッド), 1098
 getcomments() (inspect モジュール), 1575
 getcompname() (aifc.aifc のメソッド), 1265
 getcompname() (sunau.AU_read のメソッド), 1269
 getcompname() (wave.Wave_read のメソッド), 1272
 getcomptype() (aifc.aifc のメソッド), 1265
 getcomptype() (sunau.AU_read のメソッド), 1269
 getcomptype() (wave.Wave_read のメソッド), 1272
 getContentHandler() (xml.sax.xmlreader.XMLReader のメソッド), 1096
 getcontext() (decimal モジュール), 298
 getcontext() (mllib.MH のメソッド), 1008
 GetCreatorAndType() (MacOS モジュール), 1724
 getctime() (os.path モジュール), 355
 getcurrent() (mllib.Folder のメソッド), 1009
 getcwd() (os モジュール), 521
 getcwd() (2to3 fixer), 1476
 getcwd() (os モジュール), 522
 getdate() (rfc822.Message のメソッド), 1024
 getdate.tz() (rfc822.Message のメソッド), 1024
 getdecoder() (codecs モジュール), 152
 getdefaultencoding() (sys モジュール), 1533
 getdefaultlocale() (locale モジュール), 1301
 getdefaulttimeout() (socket モジュール), 875
 getdlopenflags() (sys モジュール), 1533
 getdoc() (inspect モジュール), 1575
 getDOMImplementation() (xml.dom モジュール), 1066
 getDTDHandler() (xml.sax.xmlreader.XMLReader のメソッド), 1096
 getEffectiveLevel() (logging.Logger のメソッド), 639
 getegid() (os モジュール), 509
 getElementsByTagName() (xml.dom.Document のメソッド), 1073
 getElementsByTagName() (xml.dom.Element のメソッド), 1073
 getElementsByTagNameNS() (xml.dom.Document のメソッド), 1073
 getElementsByTagNameNS() (xml.dom.Element のメソッド), 1073
 getencoder() (codecs モジュール), 152
 getencoding() (mimetools.Message のメソッド), 1011
 getEncoding() (xml.sax.xmlreader.InputSource のメソッド), 1098
 getEntityResolver() (xml.sax.xmlreader.XMLReader のメソッド), 1096
 getenv() (os モジュール), 510
 getErrorHandler() (xml.sax.xmlreader.XMLReader のメソッド), 1097
 GetErrorString() (MacOS モジュール), 1723
 geteuid() (os モジュール), 509
 getEvent() (xml.dom.pulldom.DOMEventStream のメソッド), 1085
 getEventCategory() (logging.handlers.NTEventLogHandler のメソッド), 673
 getEventType() (logging.handlers.NTEventLogHandler のメソッド), 673
 getException() (xml.sax.SAXException のメソッド), 1087
 getFeature() (xml.sax.xmlreader.XMLReader のメソッド), 1097
 GetFieldCount() (msilib.Record のメソッド), 1678
 getfile() (inspect モジュール), 1575
 getfilesystemencoding() (sys モジュール), 1533

[getfirst\(\)](#) (*cgi.FieldStorage* のメソッド), 1120
[getfirstmatchingheader\(\)](#) (*rfc822.Message* のメソッド), 1023
[getfloat\(\)](#) (*ConfigParser.RawConfigParser* のメソッド), 485
[getfmmts\(\)](#) (*ossaudiodev.oss_audio_device* のメソッド), 1280
[getfqdn\(\)](#) (*socket* モジュール), 872
[getframeinfo\(\)](#) (*inspect* モジュール), 1578
[getframerate\(\)](#) (*aifc.aifc* のメソッド), 1265
[getframerate\(\)](#) (*sunau.AU_read* のメソッド), 1269
[getframerate\(\)](#) (*wave.Wave_read* のメソッド), 1272
[getfullname\(\)](#) (*mhlib.Folder* のメソッド), 1009
[getgid\(\)](#) (*os* モジュール), 509
[getgrall\(\)](#) (*grp* モジュール), 1701
[getgrgid\(\)](#) (*grp* モジュール), 1701
[getgrnam\(\)](#) (*grp* モジュール), 1701
[getgroups\(\)](#) (*os* モジュール), 509
[getheader\(\)](#) (*httplib.HTTPResponse* のメソッド), 1167
[getheader\(\)](#) (*rfc822.Message* のメソッド), 1023
[getheaders\(\)](#) (*httplib.HTTPResponse* のメソッド), 1167
[gethostbyaddr\(\)](#) (*in module socket*), 513
[gethostbyaddr\(\)](#) (*socket* モジュール), 873
[gethostbyname\(\)](#) (*socket* モジュール), 872
[gethostbyname_ex\(\)](#) (*socket* モジュール), 872
[gethostname\(\)](#) (*in module socket*), 513
[gethostname\(\)](#) (*socket* モジュール), 872
[getincrementaldecoder\(\)](#) (*codecs* モジュール), 152
[getincrementalencoder\(\)](#) (*codecs* モジュール), 152
[getinfo\(\)](#) (*zipfile.ZipFile* のメソッド), 452
[getinnerframes\(\)](#) (*inspect* モジュール), 1578
[GetInputContext\(\)](#) (*xml.parsers.expat.xmlparser* のメソッド), 1102
[getint\(\)](#) (*ConfigParser.RawConfigParser* のメソッド), 485
[GetInteger\(\)](#) (*msilib.Record* のメソッド), 1678
[getitem\(\)](#) (*operator* モジュール), 345
[getiterator\(\)](#) (*xml.etree.ElementTree.Element* のメソッド), 1060
[getiterator\(\)](#) (*xml.etree.ElementTree.ElementTree* のメソッド), 1062
[getitimer\(\)](#) (*signal* モジュール), 917
[getkey\(\)](#) (*curses.window* のメソッド), 688
[getlast\(\)](#) (*mhlib.Folder* のメソッド), 1009
[GetLastError\(\)](#) (*ctypes* モジュール), 748
[getLength\(\)](#) (*xml.sax.xmlreader.Attributes* のメソッド), 1099
[getLevelName\(\)](#) (*logging* モジュール), 651
[getline\(\)](#) (*linecache* モジュール), 376
[getLineNumber\(\)](#) (*xml.sax.xmlreader.Locator* のメソッド), 1098
[getlist\(\)](#) (*cgi.FieldStorage* のメソッド), 1120
[getloadavg\(\)](#) (*os* モジュール), 543
[getlocale\(\)](#) (*locale* モジュール), 1302
[getLogger\(\)](#) (*logging* モジュール), 648
[getLoggerClass\(\)](#) (*logging* モジュール), 649
[getlogin\(\)](#) (*os* モジュール), 509
[getmaintype\(\)](#) (*mimetools.Message* のメソッド), 1011
[getmark\(\)](#) (*aifc.aifc* のメソッド), 1266
[getmark\(\)](#) (*sunau.AU_read* のメソッド), 1270
[getmark\(\)](#) (*wave.Wave_read* のメソッド), 1272
[getmarkers\(\)](#) (*aifc.aifc* のメソッド), 1266
[getmarkers\(\)](#) (*sunau.AU_read* のメソッド), 1270
[getmarkers\(\)](#) (*wave.Wave_read* のメソッド), 1272
[getmaxyx\(\)](#) (*curses.window* のメソッド), 688
[getmcolor\(\)](#) (*fl* モジュール), 1758
[getmember\(\)](#) (*tarfile.TarFile* のメソッド), 463
[getmembers\(\)](#) (*inspect* モジュール), 1573
[getmembers\(\)](#) (*tarfile.TarFile* のメソッド), 463
[getMessage\(\)](#) (*logging.LogRecord* のメソッド), 646
[getMessage\(\)](#) (*xml.sax.SAXException* のメソッド), 1087
[getMessagefilename\(\)](#) (*mhlib.Folder* のメソッド), 1009
[getMessageID\(\)](#) (*logging.handlers.NTEventLogHandler* のメソッド), 673
[getmodule\(\)](#) (*inspect* モジュール), 1576
[getmoduleinfo\(\)](#) (*inspect* モジュール), 1573
[getmodulename\(\)](#) (*inspect* モジュール), 1574

[getmouse\(\)](#) (*curses* モジュール), 679
[getmro\(\)](#) (*inspect* モジュール), 1577
[getmtime\(\)](#) (*os.path* モジュール), 355
[getname\(\)](#) (*chunk.Chunk* のメソッド), 1274
[getName\(\)](#) (*threading.Thread* のメソッド), 769
[getNameByQName\(\)](#) (*xml.sax.xmlreader.AttributesNS* のメソッド), 1100
[getnameinfo\(\)](#) (*socket* モジュール), 873
[getnames\(\)](#) (*tarfile.TarFile* のメソッド), 463
[getNames\(\)](#) (*xml.sax.xmlreader.Attributes* のメソッド), 1099
[getnchannels\(\)](#) (*aifc.aifc* のメソッド), 1265
[getnchannels\(\)](#) (*sunau.AU_read* のメソッド), 1269
[getnchannels\(\)](#) (*wave.Wave_read* のメソッド), 1272
[getnframes\(\)](#) (*aifc.aifc* のメソッド), 1265
[getnframes\(\)](#) (*sunau.AU_read* のメソッド), 1269
[getnframes\(\)](#) (*wave.Wave_read* のメソッド), 1272
[getnode, 1202](#)
[getnode\(\)](#) (*uuid* モジュール), 1202
[getopt \(モジュール\), 634](#)
[getopt\(\)](#) (*getopt* モジュール), 635
[GetoptError, 635](#)
[getouterframes\(\)](#) (*inspect* モジュール), 1578
[getoutput\(\)](#) (*commands* モジュール), 1719
[getpagesize\(\)](#) (*resource* モジュール), 1716
[getparam\(\)](#) (*mimetools.Message* のメソッド), 1011
[getparams\(\)](#) (*aifc.aifc* のメソッド), 1265
[getparams\(\)](#) (*al* モジュール), 1750
[getparams\(\)](#) (*sunau.AU_read* のメソッド), 1269
[getparams\(\)](#) (*wave.Wave_read* のメソッド), 1272
[getparyx\(\)](#) (*curses.window* のメソッド), 688
[getpass \(モジュール\), 676](#)
[getpass\(\)](#) (*getpass* モジュール), 676
[GetPassWarning, 676](#)
[getpath\(\)](#) (*mhlib.MH* のメソッド), 1008
[getpeercert\(\)](#) (*ssl.SSLSocket* のメソッド), 898
[getpeername\(\)](#) (*socket.socket* のメソッド), 877
[getpen\(\)](#) (*turtle* モジュール), 1381
[getpgid\(\)](#) (*os* モジュール), 510
[getpgrp\(\)](#) (*os* モジュール), 510
[getpid\(\)](#) (*os* モジュール), 510
[getplist\(\)](#) (*mimetools.Message* のメソッド), 1011
[getpos\(\)](#) (*HTMLParser.HTMLParser* のメソッド), 1037
[getppid\(\)](#) (*os* モジュール), 510
[getpreferredencoding\(\)](#) (*locale* モジュール), 1302
[getprofile\(\)](#) (*mhlib.MH* のメソッド), 1008
[getprofile\(\)](#) (*sys* モジュール), 1534
[GetProperty\(\)](#) (*msilib.SummaryInformation* のメソッド), 1678
[getProperty\(\)](#) (*xml.sax.xmlreader.XMLReader* のメソッド), 1097
[GetPropertyCount\(\)](#) (*msilib.SummaryInformation* のメソッド), 1678
[getprotobyname\(\)](#) (*socket* モジュール), 873
[getproxies\(\)](#) (*urllib* モジュール), 1142
[getPublicId\(\)](#) (*xml.sax.xmlreader.InputSource* のメソッド), 1098
[getPublicId\(\)](#) (*xml.sax.xmlreader.Locator* のメソッド), 1098
[getpwall\(\)](#) (*pwd* モジュール), 1699
[getpwnam\(\)](#) (*pwd* モジュール), 1699
[getpwuid\(\)](#) (*pwd* モジュール), 1699
[getQNameByQName\(\)](#) (*xml.sax.xmlreader.AttributesNS* のメソッド), 1100
[getQNames\(\)](#) (*xml.sax.xmlreader.AttributesNS* のメソッド), 1100
[getquota\(\)](#) (*imaplib.IMAP4* のメソッド), 1181
[getquotaroot\(\)](#) (*imaplib.IMAP4* のメソッド), 1181
[getrandbits\(\)](#) (*random* モジュール), 319
[getrawheader\(\)](#) (*rfc822.Message* のメソッド), 1023
[getreader\(\)](#) (*codecs* モジュール), 152
[getrecursionlimit\(\)](#) (*sys* モジュール), 1533
[getrefcount\(\)](#) (*sys* モジュール), 1533
[getresgid\(\)](#) (*os* モジュール), 510
[getresponse\(\)](#) (*httplib.HTTPConnection* のメソッド), 1166

- `getresuid()` (*os* モジュール), 510
- `getrlimit()` (*resource* モジュール), 1714
- `getroot()` (*xml.etree.ElementTree.ElementTree* のメソッド), 1062
- `getrusage()` (*resource* モジュール), 1715
- `getsample()` (*audioop* モジュール), 1260
- `getsampwidth()` (*aifc.aifc* のメソッド), 1265
- `getsampwidth()` (*sunau.AU_read* のメソッド), 1269
- `getsampwidth()` (*wave.Wave_read* のメソッド), 1272
- `getscreen()` (*turtle* モジュール), 1381
- `getscrollbarvalues()` (*FrameWork.ScrolledWindow* のメソッド), 1734
- `getsequences()` (*mhlib.Folder* のメソッド), 1009
- `getsequencesfilename()` (*mhlib.Folder* のメソッド), 1009
- `getservbyname()` (*socket* モジュール), 873
- `getservbyport()` (*socket* モジュール), 873
- `GetSetDescriptorType` (*types* モジュール), 261
- `getshapes()` (*turtle* モジュール), 1388
- `getsid()` (*os* モジュール), 512
- `getsignal()` (*signal* モジュール), 916
- `getsitpackages()` (*site* モジュール), 1581
- `getsize()` (*chunk.Chunk* のメソッド), 1274
- `getsize()` (*os.path* モジュール), 355
- `getsizeof()` (*sys* モジュール), 1533
- `getsizes()` (*imgfile* モジュール), 1767
- `getslice()` (*operator* モジュール), 345
- `getsockname()` (*socket.socket* のメソッド), 877
- `getsockopt()` (*socket.socket* のメソッド), 877
- `getsource()` (*inspect* モジュール), 1576
- `getsourcefile()` (*inspect* モジュール), 1576
- `getsourcelines()` (*inspect* モジュール), 1576
- `getspall()` (*spwd* モジュール), 1700
- `getspnam()` (*spwd* モジュール), 1700
- `getstate()` (*random* モジュール), 319
- `getstatus()` (*commands* モジュール), 1720
- `getstatusoutput()` (*commands* モジュール), 1719
- `getstr()` (*curses.window* のメソッド), 688
- `GetString()` (*msilib.Record* のメソッド), 1678
- `getSubject()` (*logging.handlers.SMTPHandler* のメソッド), 674
- `getsubtype()` (*mimetools.Message* のメソッド), 1011
- `GetSummaryInformation()` (*msilib.Database* のメソッド), 1677
- `getSystemId()` (*xml.sax.xmlreader.InputSource* のメソッド), 1098
- `getSystemId()` (*xml.sax.xmlreader.Locator* のメソッド), 1098
- `getsyx()` (*curses* モジュール), 679
- `gettartinio()` (*tarfile.TarFile* のメソッド), 465
- `gettempdir()` (*tempfile* モジュール), 373
- `gettempprefix()` (*tempfile* モジュール), 373
- `getTestCaseNames()` (*unittest.TestLoader* のメソッド), 1464
- `gettext` (モジュール), 1285
- `gettext()` (*gettext* モジュール), 1286
- `gettext()` (*gettext.GNUTranslations* のメソッド), 1291
- `gettext()` (*gettext.NullTranslations* のメソッド), 1289
- `gettext()` (*locale* モジュール), 1305
- `GetTicks()` (*MacOS* モジュール), 1724
- `gettimeout()` (*socket.socket* のメソッド), 879
- `gettrace()` (*sys* モジュール), 1534
- `getturtle()` (*turtle* モジュール), 1381
- `gettype()` (*mimetools.Message* のメソッド), 1011
- `getType()` (*xml.sax.xmlreader.Attributes* のメソッド), 1099
- `getuid()` (*os* モジュール), 510
- `geturl()` (*urlparse.ParseResult* のメソッド), 1209
- `getuser()` (*getpass* モジュール), 676
- `getuserbase()` (*site* モジュール), 1581
- `getusersitepackages()` (*site* モジュール), 1581
- `getvalue()` (*io.BytesIO* のメソッド), 553
- `getvalue()` (*io.StringIO* のメソッド), 558
- `getvalue()` (*StringIO.StringIO* のメソッド), 145
- `getValue()` (*xml.sax.xmlreader.Attributes* のメソッド), 1099
- `getValueByQName()` (*xml.sax.xmlreader.AttributesNS* のメソッド), 1100
- `getwch()` (*msvcrt* モジュール), 1683
- `getwche()` (*msvcrt* モジュール), 1683
- `getweakrefcount()` (*weakref* モジュール), 252
- `getweakrefs()` (*weakref* モジュール), 252
- `getwelcome()` (*ftplib.FTP* のメソッド), 1172
- `getwelcome()` (*nnplib.NNTP* のメソッド), 1187
- `getwelcome()` (*poplib.POP3* のメソッド), 1176
- `getwin()` (*curses* モジュール), 679
- `getwindowsversion()` (*sys* モジュール), 1534
- `getwriter()` (*codecs* モジュール), 153
- `getyx()` (*curses.window* のメソッド), 688
- `gid` (*tarfile.TarInfo* の属性), 466
- GIL, 1783
- GL (モジュール), 1766
- gl (モジュール), 1764
- glob
 - モジュール, 374
- glob (モジュール), 373
- glob() (*glob* モジュール), 373
- glob() (*msilib.Directory* のメソッド), 1680
- global interpreter lock, 1783
- globals() (組み込み関数), 13
- globs (*doctest.DocTest* の属性), 1431
- gmtime() (*time* モジュール), 562
- gname (*tarfile.TarInfo* の属性), 466
- GNOME, 1292
- GNU_FORMAT (*tarfile* モジュール), 461
- gnu_getopt() (*getopt* モジュール), 635
- got (*doctest.DocTestFailure* の属性), 1440
- goto() (*turtle* モジュール), 1364
- Graphical User Interface, 1315
- GREATER (*token* モジュール), 1636
- GREATEREQUAL (*token* モジュール), 1636
- Greenwich Mean Time, 560
- grey22grey() (*imageop* モジュール), 1264
- grey2grey2() (*imageop* モジュール), 1264
- grey2grey4() (*imageop* モジュール), 1264
- grey2mono() (*imageop* モジュール), 1263
- grey42grey() (*imageop* モジュール), 1264
- group() (*nnplib.NNTP* のメソッド), 1188
- group() (*re.MatchObject* のメソッド), 119
- groupby() (*itertools* モジュール), 328
- groupdict() (*re.MatchObject* のメソッド), 121
- groupindex (*re.RegexObject* の属性), 119
- groups (*re.RegexObject* の属性), 119
- groups() (*re.MatchObject* のメソッド), 120
- grp (モジュール), 1700
- gt() (*operator* モジュール), 342
- guess_all_extensions() (*mimetypes* モジュール), 1012
- guess_all_extensions() (*mimetypes.MimeTypes* のメソッド), 1015
- guess_extension() (*mimetypes* モジュール), 1012
- guess_extension() (*mimetypes.MimeTypes* のメソッド), 1014
- guess_scheme() (*wsgiref.util* モジュール), 1126
- guess_type() (*mimetypes* モジュール), 1012
- guess_type() (*mimetypes.MimeTypes* のメソッド), 1015
- GUI, 1315
- gzip (モジュール), 445
- GzipFile (*gzip* のクラス), 445
- halfdelay() (*curses* モジュール), 680
- handle() (*BaseHTTPServer.BaseHTTPRequestHandler* のメソッド), 1222
- handle() (*logging.Handler* のメソッド), 643
- handle() (*logging.Logger* のメソッド), 641
- handle() (*logging.NullHandler* のメソッド), 666
- handle() (*SocketServer.BaseRequestHandler* のメソッド), 1215
- handle() (*wsgiref.simple_server.WSGIRequestHandler* のメソッド), 1131
- handle_accept() (*asyncore.dispatcher* のメソッド), 923
- handle_charref() (*HTMLParser.HTMLParser* のメソッド), 1038

[handle_charref\(\)](#) (*sgmlib.SGMLParser* のメソッド), 1042
[handle_close\(\)](#) (*asyncore.dispatcher* のメソッド), 923
[handle_comment\(\)](#) (*HTMLParser.HTMLParser* のメソッド), 1038
[handle_comment\(\)](#) (*sgmlib.SGMLParser* のメソッド), 1043
[handle_connect\(\)](#) (*asyncore.dispatcher* のメソッド), 923
[handle_data\(\)](#) (*HTMLParser.HTMLParser* のメソッド), 1038
[handle_data\(\)](#) (*sgmlib.SGMLParser* のメソッド), 1042
[handle_decl\(\)](#) (*HTMLParser.HTMLParser* のメソッド), 1038
[handle_decl\(\)](#) (*sgmlib.SGMLParser* のメソッド), 1043
[handle_endtag\(\)](#) (*HTMLParser.HTMLParser* のメソッド), 1037
[handle_endtag\(\)](#) (*sgmlib.SGMLParser* のメソッド), 1042
[handle_entityref\(\)](#) (*HTMLParser.HTMLParser* のメソッド), 1038
[handle_entityref\(\)](#) (*sgmlib.SGMLParser* のメソッド), 1043
[handle_error\(\)](#) (*asyncore.dispatcher* のメソッド), 923
[handle_error\(\)](#) (*SocketServer.BaseServer* のメソッド), 1214
[handle_expt\(\)](#) (*asyncore.dispatcher* のメソッド), 923
[handle_image\(\)](#) (*htmlib.HTMLParser* のメソッド), 1046
[handle_one_request\(\)](#) (*BaseHTTPServer.BaseHTTPRequestHandler* のメソッド), 1222
[handle_pi\(\)](#) (*HTMLParser.HTMLParser* のメソッド), 1038
[handle_read\(\)](#) (*asyncore.dispatcher* のメソッド), 922
[handle_request\(\)](#) (*SimpleXMLRPCServer.CGIXMLRPCRequestHandler* のメソッド), 1257
[handle_request\(\)](#) (*SocketServer.BaseServer* のメソッド), 1212
[handle_startendtag\(\)](#) (*HTMLParser.HTMLParser* のメソッド), 1038
[handle_starttag\(\)](#) (*HTMLParser.HTMLParser* のメソッド), 1037
[handle_starttag\(\)](#) (*sgmlib.SGMLParser* のメソッド), 1042
[handle_timeout\(\)](#) (*SocketServer.BaseServer* のメソッド), 1214
[handle_write\(\)](#) (*asyncore.dispatcher* のメソッド), 922
[handleError\(\)](#) (*logging.Handler* のメソッド), 643
[handleError\(\)](#) (*logging.handlers.SocketHandler* のメソッド), 669
[handler\(\)](#) (*cgitb* モジュール), 1126
[HAS_ALPN](#) (*ssl* モジュール), 895
[has_children\(\)](#) (*symtable.SymbolTable* のメソッド), 1634
[has_colors\(\)](#) (*curses* モジュール), 680
[has_data\(\)](#) (*urllib2.Request* のメソッド), 1151
[HAS_ECDH](#) (*ssl* モジュール), 896
[has_exec\(\)](#) (*symtable.SymbolTable* のメソッド), 1634
[has_extn\(\)](#) (*smtpplib.SMTP* のメソッド), 1193
[has_header\(\)](#) (*csv.Sniffer* のメソッド), 475
[has_header\(\)](#) (*urllib2.Request* のメソッド), 1151
[has_ic\(\)](#) (*curses* モジュール), 680
[has_il\(\)](#) (*curses* モジュール), 680
[has_import_star\(\)](#) (*symtable.SymbolTable* のメソッド), 1634
[has_ipv6](#) (*socket* モジュール), 871
[has_key](#) (*2to3 fixer*), 1476
[has_key\(\)](#) (*bsddb.bsddbobject* のメソッド), 414
[has_key\(\)](#) (*curses* モジュール), 680
[has_key\(\)](#) (*dict* のメソッド), 65
[has_key\(\)](#) (*email.message.Message* のメソッド), 935
[has_key\(\)](#) (*mailbox.Mailbox* のメソッド), 986
[has_nonstandard_attr\(\)](#) (*cookielib.Cookie* のメソッド), 1237
[HAS_NPN](#) (*ssl* モジュール), 896
[has_option\(\)](#) (*ConfigParser.RawConfigParser* のメソッド), 484
[has_option\(\)](#) (*optparse.OptionParser* のメソッド), 624
[has_section\(\)](#) (*ConfigParser.RawConfigParser* のメソッド), 484
[HAS_SNI](#) (*ssl* モジュール), 896
[HAS_TLSv1.3](#) (*ssl* モジュール), 896
[hasattr\(\)](#) (組み込み関数), 13

[hasAttribute\(\)](#) (*xml.dom.Element* のメソッド), 1073
[hasAttributeNS\(\)](#) (*xml.dom.Element* のメソッド), 1073
[hasAttributes\(\)](#) (*xml.dom.Node* のメソッド), 1070
[hasChildNodes\(\)](#) (*xml.dom.Node* のメソッド), 1070
[hascompare](#) (*dis* モジュール), 1647
[hasconst](#) (*dis* モジュール), 1647
[hasFeature\(\)](#) (*xml.dom.DOMImplementation* のメソッド), 1068
[hasfree](#) (*dis* モジュール), 1647
[hash\(\)](#) (組み込み関数), 14
[hash.block_size](#) (*hashlib* モジュール), 501
[hash.digest_size](#) (*hashlib* モジュール), 501
[hashable](#), 1783
[Hashable](#) (*collections* のクラス), 226
[hashlib](#) (モジュール), 499
[hashlib.algorithms](#) (*hashlib* モジュール), 500
[hashopen\(\)](#) (*bsddb* モジュール), 413
[hasjabs](#) (*dis* モジュール), 1647
[hasjrel](#) (*dis* モジュール), 1647
[haslocal](#) (*dis* モジュール), 1647
[hasname](#) (*dis* モジュール), 1647
[HAVE_ARGUMENT](#) (*opcode*), 1657
[have_unicode](#) (*test.support* モジュール), 1483
[head\(\)](#) (*nntplib.NNTP* のメソッド), 1189
[Header](#) (*email.header* のクラス), 950
[header_encode\(\)](#) (*email.charset.Charset* のメソッド), 955
[header_encoding](#) (*email.charset.Charset* の属性), 953
[header_items\(\)](#) (*urllib2.Request* のメソッド), 1152
[header_offset](#) (*zipfile.ZipInfo* の属性), 457
[HeaderError](#), 461
[HeaderParseError](#), 957
[headers](#)
 MIME, 1012, 1116
[headers](#) (*BaseHTTPServer.BaseHTTPRequestHandler* の属性), 1220
[headers](#) (*rfc822.Message* の属性), 1024
[Headers](#) (*wsgiref.headers* のクラス), 1129
[headers](#) (*xmlrpclib.ProtocolError* の属性), 1249
[heading\(\)](#) (*tk.Treeview* のメソッド), 1345
[heading\(\)](#) (*turtle* モジュール), 1370
[heapify\(\)](#) (*heapq* モジュール), 229
[heapmin\(\)](#) (*msvcrt* モジュール), 1684
[heappop\(\)](#) (*heapq* モジュール), 229
[heappush\(\)](#) (*heapq* モジュール), 229
[heappushpop\(\)](#) (*heapq* モジュール), 229
[heapq](#) (モジュール), 228
[heapreplace\(\)](#) (*heapq* モジュール), 229
[hello\(\)](#) (*smtpplib.SMTP* のメソッド), 1193
[help](#)
 online, 1409
[help](#) (*optparse.Option* の属性), 618
[help\(\)](#) (*nntplib.NNTP* のメソッド), 1188
[help\(\)](#) (組み込み関数), 14
[herror](#), 869
[hex](#) (*uuid.UUID* の属性), 1202
[hex\(\)](#) (*float* のメソッド), 43
[hex\(\)](#) (*future.builtins* モジュール), 1547
[hex\(\)](#) (組み込み関数), 14
[hexadecimal](#)
 literals, 39
[hexbin\(\)](#) (*binhex* モジュール), 1028
[hexdigest\(\)](#) (*hashlib.hash* のメソッド), 501
[hexdigest\(\)](#) (*hmac.HMAC* のメソッド), 503
[hexdigest\(\)](#) (*md5.md5* のメソッド), 504
[hexdigest\(\)](#) (*sha.sha* のメソッド), 505
[hexdigits](#) (*string* モジュール), 92
[hexlify\(\)](#) (*binascii* モジュール), 1031
[hexversion](#) (*sys* モジュール), 1535
[hidden\(\)](#) (*curses.panel.Panel* のメソッド), 704
[hide\(\)](#) (*curses.panel.Panel* のメソッド), 704
[hide\(\)](#) (*tk.Notebook* のメソッド), 1338
[hide_cookie2](#) (*cookielib.CookiePolicy* の属性), 1233
[hide_form\(\)](#) (*fl.form* のメソッド), 1758

hideturtle() (*turtle* モジュール), 1377
HierarchyRequestErr, 1076
HIGHEST_PROTOCOL (*pickle* モジュール), 387
HKEY_CLASSES_ROOT (*winreg* モジュール), 1691
HKEY_CURRENT_CONFIG (*winreg* モジュール), 1691
HKEY_CURRENT_USER (*winreg* モジュール), 1691
HKEY_DYN_DATA (*winreg* モジュール), 1691
HKEY_LOCAL_MACHINE (*winreg* モジュール), 1691
HKEY_PERFORMANCE_DATA (*winreg* モジュール), 1691
HKEY_USERS (*winreg* モジュール), 1691
hline() (*curses.window* のメソッド), 688
HList (*Tix* のクラス), 1355
hls.to.rgb() (*colorsys* モジュール), 1276
hmac (モジュール), 502
HOME, 354, 1582
home() (*turtle* モジュール), 1366
HOMEDRIVE, 354
HOMEPATH, 354
hook_compressed() (*fileinput* モジュール), 360
hook_encoded() (*fileinput* モジュール), 361
hosts (*netrc.netrc* の属性), 491
hotshot (モジュール), 1511
hotshot.stats (モジュール), 1512
hour (*datetime.datetime* の属性), 186
hour (*datetime.time* の属性), 193
HRESULT (*ctypes* のクラス), 753
hStdError (*subprocess.STARTUPINFO* の属性), 863
hStdInput (*subprocess.STARTUPINFO* の属性), 863
hStdOutput (*subprocess.STARTUPINFO* の属性), 863
hsv.to.rgb() (*colorsys* モジュール), 1276
ht() (*turtle* モジュール), 1377
HTML, 1035, 1045, 1145
HTMLCalendar (*calendar* のクラス), 207
HtmlDiff (*difflib* のクラス), 133
HtmlDiff.__init__() (*difflib* モジュール), 133
HtmlDiff.make_file() (*difflib* モジュール), 134
HtmlDiff.make_table() (*difflib* モジュール), 134
htmlentitydefs (モジュール), 1047
htmllib
 モジュール, 1145
htmllib (モジュール), 1045
HTMLParseError, 1036, 1045
HTMLParser (*class in htmlib*), 1669
HTMLParser (*htmlib* のクラス), 1045
HTMLParser (*HTMLParser* のクラス), 1035
HTMLParser (モジュール), 1035
htonl() (*socket* モジュール), 874
htons() (*socket* モジュール), 874
HTTP
 httplib (standard module), 1161
 protocol, 1116, 1144, 1145, 1161, 1219
http.error_301() (*urllib2.HTTPRedirectHandler* のメソッド), 1156
http.error_302() (*urllib2.HTTPRedirectHandler* のメソッド), 1156
http.error_303() (*urllib2.HTTPRedirectHandler* のメソッド), 1156
http.error_307() (*urllib2.HTTPRedirectHandler* のメソッド), 1156
http.error_401() (*urllib2.HTTPBasicAuthHandler* のメソッド), 1157
http.error_401() (*urllib2.HTTPDigestAuthHandler* のメソッド), 1158
http.error_407() (*urllib2.ProxyBasicAuthHandler* のメソッド), 1158
http.error_407() (*urllib2.ProxyDigestAuthHandler* のメソッド), 1158
http.error_auth_reqed()
 (*urllib2.AbstractBasicAuthHandler* のメソッド), 1157
http.error_auth_reqed()
 (*urllib2.AbstractDigestAuthHandler* のメソッド), 1158
http.error_default() (*urllib2.BaseHandler* のメソッド), 1154

http.error_nnn() (*urllib2.BaseHandler* のメソッド), 1155
http.open() (*urllib2.HTTPHandler* のメソッド), 1158
HTTP_PORT (*httplib* モジュール), 1164
http.proxy, 1139, 1147, 1160
http.response() (*urllib2.HTTPErrorProcessor* のメソッド), 1159
http.version (*wsgiref.handlers.BaseHandler* の属性), 1136
HTTPBasicAuthHandler (*urllib2* のクラス), 1150
HTTPConnection (*httplib* のクラス), 1162
HTTPCookieProcessor (*urllib2* のクラス), 1149
httpd, 1219
HTTPDefaultErrorHandler (*urllib2* のクラス), 1149
HTTPDigestAuthHandler (*urllib2* のクラス), 1150
HTTPError, 1148
HTTPErrorProcessor (*urllib2* のクラス), 1151
HTTPException, 1163
HTTPHandler (*logging.handlers* のクラス), 675
HTTPHandler (*urllib2* のクラス), 1150
httplib (モジュール), 1161
HTTPMessage (*httplib* のクラス), 1163
HTTPPasswordMgr (*urllib2* のクラス), 1149
HTTPPasswordMgrWithDefaultRealm (*urllib2* のクラス), 1149
HTTPRedirectHandler (*urllib2* のクラス), 1149
HTTPResponse (*httplib* のクラス), 1163
https.open() (*urllib2.HTTPSHandler* のメソッド), 1158
HTTPS_PORT (*httplib* モジュール), 1164
https.response() (*urllib2.HTTPErrorProcessor* のメソッド), 1159
HTTPSConnection (*httplib* のクラス), 1162
HTTPServer (*BaseHTTPServer* のクラス), 1220
HTTPSHandler (*urllib2* のクラス), 1150
hypertext, 1045
hypot() (*math* モジュール), 278

I (*re* モジュール), 114
I/O control
 buffering, 18, 513, 878
 POSIX, 1704
 tty, 1704
 UNIX, 1706
iadd() (*operator* モジュール), 346
iand() (*operator* モジュール), 346
IC (*ic* のクラス), 1721
ic (モジュール), 1721
icglue
 モジュール, 1721
iconcat() (*operator* モジュール), 346
icopen (モジュール), 1776
id() (*unittest.TestCase* のメソッド), 1461
id() (組み込み関数), 14
idcok() (*curses.window* のメソッド), 688
ident (*cd* モジュール), 1753
ident (*select.kevent* の属性), 761
ident (*threading.Thread* の属性), 769
identchars (*cmd.Cmd* の属性), 1309
identify() (*ttk.Notebook* のメソッド), 1338
identify() (*ttk.Treeview* のメソッド), 1345
identify() (*ttk.Widget* のメソッド), 1334
identify_column() (*ttk.Treeview* のメソッド), 1345
identify_element() (*ttk.Treeview* のメソッド), 1346
identify_region() (*ttk.Treeview* のメソッド), 1345
identify_row() (*ttk.Treeview* のメソッド), 1345
idioms (*2to3 fixer*), 1476
idiv() (*operator* モジュール), 346
IDLE, 1784
IDLE, 1396
idle() (*FrameWork.Application* のメソッド), 1733
IDLESTARTUP, 1404
idlok() (*curses.window* のメソッド), 689
IEEE-754, 1583
if
 文, 37

ifilter() (*itertools* モジュール), 329
 ifilterfalse() (*itertools* モジュール), 329
 ifloordiv() (*operator* モジュール), 346
 iglob() (*glob* モジュール), 374
 ignoreableWhitespace() (*xml.sax.handler.ContentHandler* のメソッド), 1092
 ignore_errors() (*codecs* モジュール), 153
 IGNORE_EXCEPTION_DETAIL (*doctest* モジュール), 1420
 ignore_patterns() (*shutil* モジュール), 377
 IGNORECASE (*re* モジュール), 114
 ihave() (*nnlib.NNTP* のメソッド), 1189
 ilshift() (*operator* モジュール), 346
 imag (*numbers.Complex* の属性), 271
 imageop (モジュール), 1263
 imap() (*itertools* モジュール), 330
 imap() (*multiprocessing.pool.multiprocessing.Pool* のメソッド), 810
 imap_unordered() (*multiprocessing.pool.multiprocessing.Pool* のメソッド), 810
 IMAP4
 protocol, 1178
 IMAP4 (*imaplib* のクラス), 1178
 IMAP4.abort, 1178
 IMAP4.error, 1178
 IMAP4.readonly, 1178
 IMAP4_SSL
 protocol, 1178
 IMAP4_SSL (*imaplib* のクラス), 1178
 IMAP4.stream
 protocol, 1178
 IMAP4.stream (*imaplib* のクラス), 1179
 imaplib (モジュール), 1178
 imgfile (モジュール), 1767
 imghdr (モジュール), 1276
 immedok() (*curses.window* のメソッド), 689
 immutable, 1784
 ImmutableSet (*sets* のクラス), 240
 imod() (*operator* モジュール), 347
 imp
 モジュール, 30
 imp (モジュール), 1601
 ImpImporter (*pkgutil* のクラス), 1613
 ImpLoader (*pkgutil* のクラス), 1613
 import
 文, 30, 1601, 1606
 import (2to3 fixer), 1477
 import_file() (*imputil.DynLoadSuffixImporter* のメソッド), 1607
 import_fresh_module() (*test.support* モジュール), 1485
 IMPORT_FROM (*opcode*), 1654
 import_module() (*importlib* モジュール), 1606
 import_module() (*test.support* モジュール), 1485
 IMPORT_NAME (*opcode*), 1654
 IMPORT_STAR (*opcode*), 1652
 import_top() (*imputil.Importer* のメソッド), 1606
 importer, 1784
 Importer (*imputil* のクラス), 1606
 ImportError, 83
 importing, 1784
 importlib (モジュール), 1606
 ImportManager (*imputil* のクラス), 1606
 imports (2to3 fixer), 1477
 imports2 (2to3 fixer), 1477
 ImportWarning, 87
 ImproperConnectionState, 1164
 imputil (モジュール), 1606
 imul() (*operator* モジュール), 347
 in
 演算子, 39, 46
 in_dll() (*ctypes.CData* のメソッド), 750
 in_table_al() (*stringprep* モジュール), 172
 in_table_b1() (*stringprep* モジュール), 172
 in_table_c11() (*stringprep* モジュール), 173
 in_table_c11_c12() (*stringprep* モジュール), 173
 in_table_c12() (*stringprep* モジュール), 173
 in_table_c21() (*stringprep* モジュール), 173
 in_table_c21_c22() (*stringprep* モジュール), 173
 in_table_c22() (*stringprep* モジュール), 173
 in_table_c3() (*stringprep* モジュール), 173
 in_table_c4() (*stringprep* モジュール), 173
 in_table_c5() (*stringprep* モジュール), 173
 in_table_c6() (*stringprep* モジュール), 173
 in_table_c7() (*stringprep* モジュール), 173
 in_table_c8() (*stringprep* モジュール), 173
 in_table_c9() (*stringprep* モジュール), 173
 in_table_d1() (*stringprep* モジュール), 173
 in_table_d2() (*stringprep* モジュール), 174
 inc() (*EasyDialogs.ProgressBar* のメソッド), 1730
 inch() (*curses.window* のメソッド), 689
 Incomplete, 1031
 IncompleteRead, 1163
 increment_lineno() (*ast* モジュール), 1631
 IncrementalDecoder (*codecs* のクラス), 158
 IncrementalEncoder (*codecs* のクラス), 157
 IncrementalNewlineDecoder (*io* のクラス), 558
 IncrementalParser (*xml.sax.xmlreader* のクラス), 1095
 indent (*doctest.Example* の属性), 1432
 INDENT (*token* モジュール), 1636
 IndentationError, 85
 Independent JPEG Group, 1768
 index (*cd* モジュール), 1753
 index() (*array.array* のメソッド), 238
 index() (*list* method), 58
 index() (*operator* モジュール), 343
 index() (*str* のメソッド), 49
 index() (*string* モジュール), 104
 index() (*ttk.Notebook* のメソッド), 1338
 index() (*ttk.Treeview* のメソッド), 1346
 IndexError, 83
 index_of() (*operator* モジュール), 345
 IndexSizeErr, 1076
 inet_aton() (*socket* モジュール), 874
 inet_ntoa() (*socket* モジュール), 874
 inet_ntop() (*socket* モジュール), 875
 inet_pton() (*socket* モジュール), 875
 Inexact (*decimal* のクラス), 306
 infile (*shlex.shlex* の属性), 1313
 Infinity, 13, 103
 info() (*gettext.NullTranslations* のメソッド), 1290
 info() (*logging* モジュール), 650
 info() (*logging.Logger* のメソッド), 640
 infolist() (*zipfile.ZipFile* のメソッド), 452
 InfoScrap() (*Carbon.Scrap* モジュール), 1739
 ini file, 481
 init() (*fm* モジュール), 1763
 init() (*mimetypes* モジュール), 1013
 init_built_in() (*imp* モジュール), 1603
 init_color() (*curses* モジュール), 680
 init_database() (*msilib* モジュール), 1676
 init_frozen() (*imp* モジュール), 1603
 init_pair() (*curses* モジュール), 680
 inited (*mimetypes* モジュール), 1013
 initgroups() (*os* モジュール), 509
 initial_indent (*textwrap.TextWrapper* の属性), 149
 initscr() (*curses* モジュール), 680
 INPLACE_ADD (*opcode*), 1650
 INPLACE_AND (*opcode*), 1650
 INPLACE_DIVIDE (*opcode*), 1649
 INPLACE_FLOOR_DIVIDE (*opcode*), 1650
 INPLACE_LSHIFT (*opcode*), 1650
 INPLACE_MODULO (*opcode*), 1650
 INPLACE_MULTIPLY (*opcode*), 1649
 INPLACE_OR (*opcode*), 1650
 INPLACE_POWER (*opcode*), 1649
 INPLACE_RSHIFT (*opcode*), 1650
 INPLACE_SUBTRACT (*opcode*), 1650

- INPLACE_TRUE_DIVIDE (*opcode*), 1650
- INPLACE_XOR (*opcode*), 1650
- input
 - 組み込み関数, 1541
- input (*2to3 fixer*), 1477
- input () (*fileinput* モジュール), 359
- input () (組み込み関数), 14
- input_charset (*email.charset.Charset* の属性), 953
- input_codec (*email.charset.Charset* の属性), 953
- InputOnly (*Tix* のクラス), 1356
- InputSource (*xml.sax.xmlreader* のクラス), 1095
- InputType (*cStringIO* モジュール), 146
- insch () (*curses.window* のメソッド), 689
- insdelln () (*curses.window* のメソッド), 689
- insert () (*array.array* のメソッド), 238
- insert () (*list* method), 58
- insert () (*ttk.Notebook* のメソッド), 1338
- insert () (*ttk.Treeview* のメソッド), 1346
- insert () (*xml.etree.ElementTree.Element* のメソッド), 1060
- insert_text () (*readline* モジュール), 847
- insertBefore () (*xml.dom.Node* のメソッド), 1070
- InsertionLoc (*aetypes* のクラス), 1746
- insertln () (*curses.window* のメソッド), 689
- insnstr () (*curses.window* のメソッド), 689
- insort () (*bisect* モジュール), 234
- insort_left () (*bisect* モジュール), 234
- insort_right () (*bisect* モジュール), 234
- inspect (モジュール), 1571
- insstr () (*curses.window* のメソッド), 689
- install () (*gettext* モジュール), 1288
- install () (*gettext.NullTranslations* のメソッド), 1290
- install () (*imputil.ImportManager* のメソッド), 1606
- install_opener () (*urllib2* モジュール), 1147
- installaehandler () (*MiniAEFrame.AEServer* のメソッド), 1748
- installAutoGIL () (*autoGIL* モジュール), 1735
- installHandler () (*unittest* モジュール), 1472
- instance () (*new* モジュール), 262
- instancemethod () (*new* モジュール), 262
- InstanceType (*types* モジュール), 260
- instate () (*ttk.Widget* のメソッド), 1334
- instr () (*curses.window* のメソッド), 689
- istream (*shlex.shlex* の属性), 1313
- int
 - 組み込み関数, 39
- int (*uuid.UUID* の属性), 1202
- int (組み込みクラス), 15
- Int2AP () (*imaplib* モジュール), 1179
- integer
 - division, 40
 - division, long, 40
 - literals, 39
 - literals, long, 39
 - types, operations on, 41
 - オブジェクト, 39
- integer division, 1784
- Integral (*numbers* のクラス), 272
- Integrated Development Environment, 1396
- Intel/DVI ADPCM, 1259
- interact () (*code* モジュール), 1587
- interact () (*code.InteractiveConsole* のメソッド), 1589
- interact () (*telnetlib.Telnet* のメソッド), 1199
- interactive, 1784
- InteractiveConsole (*code* のクラス), 1587
- InteractiveInterpreter (*code* のクラス), 1587
- intern (*2to3 fixer*), 1477
- intern () (組み込み関数), 33
- internal_attr (*zipfile.ZipInfo* の属性), 457
- Internaldate2tuple () (*imaplib* モジュール), 1179
- internalSubset (*xml.dom.DocumentType* の属性), 1072
- Internet, 1113
- Internet Config, 1139
- interpolation, string (%), 55
- InterpolationDepthError, 483
- InterpolationError, 483
- InterpolationMissingOptionError, 483
- InterpolationSyntaxError, 483
- interpreted, 1784
- interpreter prompts, 1538
- interrupt () (*sqlite3.Connection* のメソッド), 423
- interrupt_main () (*thread* モジュール), 777
- intersection () (*frozenset* のメソッド), 61
- intersection.update () (*frozenset* のメソッド), 62
- IntlText (*aetypes* のクラス), 1746
- IntlWritingCode (*aetypes* のクラス), 1747
- intro (*cmd.Cmd* の属性), 1309
- IntType (*types* モジュール), 259
- InuseAttributeErr, 1076
- inv () (*operator* モジュール), 343
- InvalidAccessErr, 1076
- InvalidCharacterErr, 1076
- InvalidModificationErr, 1076
- InvalidOperation (*decimal* のクラス), 306
- InvalidStateErr, 1076
- InvalidURL, 1163
- invert () (*operator* モジュール), 343
- io (モジュール), 545
- IOBase (*io* のクラス), 548
- ioctl () (*fcntl* モジュール), 1706
- ioctl () (*socket.socket* のメソッド), 877
- IOError, 83
- ior () (*operator* モジュール), 347
- ipow () (*operator* モジュール), 347
- irepeat () (*operator* モジュール), 347
- IRIS Font Manager, 1763
- IRIX
 - threads, 779
- irshift () (*operator* モジュール), 347
- is
 - 演算子, 38
- is not
 - 演算子, 38
- is_ () (*operator* モジュール), 343
- is_alive () (*multiprocessing.Process* のメソッド), 787
- is_alive () (*threading.Thread* のメソッド), 769
- is_assigned () (*symtable.Symbol* のメソッド), 1635
- is_blocked () (*cookielib.DefaultCookiePolicy* のメソッド), 1234
- is_builtin () (*imp* モジュール), 1604
- is_canonical () (*decimal.Context* のメソッド), 302
- is_canonical () (*decimal.Decimal* のメソッド), 293
- IS_CHARACTER_JUNK () (*difflib* モジュール), 137
- is_data () (*multifile.MultiFile* のメソッド), 1019
- is_declared_global () (*symtable.Symbol* のメソッド), 1635
- is_empty () (*asynchat.fifo* のメソッド), 928
- is_expired () (*cookielib.Cookie* のメソッド), 1237
- is_finite () (*decimal.Context* のメソッド), 302
- is_finite () (*decimal.Decimal* のメソッド), 293
- is_free () (*symtable.Symbol* のメソッド), 1635
- is_frozen () (*imp* モジュール), 1604
- is_global () (*symtable.Symbol* のメソッド), 1635
- is_hop_by_hop () (*wsgiref.util* モジュール), 1128
- is_imported () (*symtable.Symbol* のメソッド), 1635
- is_infinite () (*decimal.Context* のメソッド), 302
- is_infinite () (*decimal.Decimal* のメソッド), 293
- is_integer () (*float* のメソッド), 42
- is_jython (*test.support* モジュール), 1483
- IS_LINE_JUNK () (*difflib* モジュール), 137
- is_linetouched () (*curses.window* のメソッド), 690
- is_local () (*symtable.Symbol* のメソッド), 1635
- is_multipart () (*email.message.Message* のメソッド), 933
- is_namespace () (*symtable.Symbol* のメソッド), 1635
- is_nan () (*decimal.Context* のメソッド), 303
- is_nan () (*decimal.Decimal* のメソッド), 293
- is_nested () (*symtable.SymbolTable* のメソッド), 1634
- is_normal () (*decimal.Context* のメソッド), 303

- `is.normal()` (*decimal.Decimal* のメソッド), 293
- `is.not()` (*operator* モジュール), 343
- `is.not.allowed()` (*cookiecutter.DefaultCookiePolicy* のメソッド), 1234
- `is.optimized()` (*symtable.SymbolTable* のメソッド), 1633
- `is.package()` (*zipimport.zipimporter* のメソッド), 1611
- `is.parameter()` (*symtable.Symbol* のメソッド), 1635
- `is.python.build()` (*sysconfig* モジュール), 1546
- `is.qnan()` (*decimal.Context* のメソッド), 303
- `is.qnan()` (*decimal.Decimal* のメソッド), 293
- `is.referenced()` (*symtable.Symbol* のメソッド), 1635
- `is.resource.enabled()` (*test.support* モジュール), 1483
- `is.scriptable()` (*gensuitemodule* モジュール), 1742
- `is.set()` (*threading.Event* のメソッド), 775
- `is.signed()` (*decimal.Context* のメソッド), 303
- `is.signed()` (*decimal.Decimal* のメソッド), 293
- `is.snan()` (*decimal.Context* のメソッド), 303
- `is.snan()` (*decimal.Decimal* のメソッド), 293
- `is.subnormal()` (*decimal.Context* のメソッド), 303
- `is.subnormal()` (*decimal.Decimal* のメソッド), 293
- `is.tarfile()` (*tarfile* モジュール), 460
- `is.term.resized()` (*curses* モジュール), 680
- `is.tracked()` (*gc* モジュール), 1570
- `is.unverifiable()` (*urllib2.Request* のメソッド), 1152
- `is.wintouched()` (*curses.window* のメソッド), 690
- `is.zero()` (*decimal.Context* のメソッド), 303
- `is.zero()` (*decimal.Decimal* のメソッド), 294
- `is.zipfile()` (*zipfile* モジュール), 451
- `isabs()` (*os.path* モジュール), 355
- `isabstract()` (*inspect* モジュール), 1574
- `isAlive()` (*threading.Thread* のメソッド), 769
- `isalnum()` (*curses.ascii* モジュール), 701
- `isalnum()` (*str* のメソッド), 49
- `isalpha()` (*curses.ascii* モジュール), 701
- `isalpha()` (*str* のメソッド), 49
- `isascii()` (*curses.ascii* モジュール), 701
- `isatty()` (*chunk.Chunk* のメソッド), 1275
- `isatty()` (*file* のメソッド), 70
- `isatty()` (*io.IOBase* のメソッド), 549
- `isatty()` (*os* モジュール), 517
- `isblank()` (*curses.ascii* モジュール), 701
- `isblk()` (*tarfile.TarInfo* のメソッド), 467
- `isbuiltin()` (*inspect* モジュール), 1574
- `isCallable()` (*operator* モジュール), 348
- `ischr()` (*tarfile.TarInfo* のメソッド), 467
- `isclass()` (*inspect* モジュール), 1574
- `isctrl()` (*curses.ascii* モジュール), 701
- `iscode()` (*inspect* モジュール), 1574
- `iscomment()` (*rfc822.Message* のメソッド), 1023
- `isctrl()` (*curses.ascii* モジュール), 702
- `isDaemon()` (*threading.Thread* のメソッド), 769
- `isdatadescriptor()` (*inspect* モジュール), 1575
- `isdecimal()` (*unicode* のメソッド), 55
- `isdev()` (*tarfile.TarInfo* のメソッド), 467
- `isdigit()` (*curses.ascii* モジュール), 701
- `isdigit()` (*str* のメソッド), 49
- `isdir()` (*os.path* モジュール), 355
- `isdir()` (*tarfile.TarInfo* のメソッド), 467
- `isdisjoint()` (*frozenset* のメソッド), 61
- `isdown()` (*turtle* モジュール), 1373
- `iselement()` (*xml.etree.ElementTree* モジュール), 1057
- `isEnabled()` (*gc* モジュール), 1568
- `isEnabledFor()` (*logging.Logger* のメソッド), 639
- `isendwin()` (*curses* モジュール), 681
- `ISEOF()` (*token* モジュール), 1636
- `isexpr()` (*parser* モジュール), 1624
- `isexpr()` (*parser.ST* のメソッド), 1625
- `isfifo()` (*tarfile.TarInfo* のメソッド), 467
- `isfile()` (*os.path* モジュール), 355
- `isfile()` (*tarfile.TarInfo* のメソッド), 467
- `isfirstline()` (*fileinput* モジュール), 359
- `isframe()` (*inspect* モジュール), 1574
- `isfunction()` (*inspect* モジュール), 1574
- `isgenerator()` (*inspect* モジュール), 1574
- `isgeneratorfunction()` (*inspect* モジュール), 1574
- `isgetsetdescriptor()` (*inspect* モジュール), 1575
- `isgraph()` (*curses.ascii* モジュール), 702
- `isheader()` (*rfc822.Message* のメソッド), 1023
- `isinf()` (*cmath* モジュール), 283
- `isinf()` (*math* モジュール), 276
- `isinstance(2to3 fixer)`, 1477
- `isinstance()` (組み込み関数), 15
- `iskeyword()` (*keyword* モジュール), 1638
- `islast()` (*rfc822.Message* のメソッド), 1023
- `isleap()` (*calendar* モジュール), 208
- `islice()` (*itertools* モジュール), 330
- `islink()` (*os.path* モジュール), 355
- `islnk()` (*tarfile.TarInfo* のメソッド), 467
- `islower()` (*curses.ascii* モジュール), 702
- `islower()` (*str* のメソッド), 49
- `isMappingType()` (*operator* モジュール), 348
- `ismemberdescriptor()` (*inspect* モジュール), 1575
- `ismeta()` (*curses.ascii* モジュール), 702
- `ismethod()` (*inspect* モジュール), 1574
- `ismethoddescriptor()` (*inspect* モジュール), 1574
- `ismodule()` (*inspect* モジュール), 1574
- `ismount()` (*os.path* モジュール), 356
- `isnan()` (*cmath* モジュール), 283
- `isnan()` (*math* モジュール), 276
- `ISNONTERMINAL()` (*token* モジュール), 1636
- `isNumberType()` (*operator* モジュール), 348
- `isnumeric()` (*unicode* のメソッド), 55
- `isocalendar()` (*datetime.date* のメソッド), 182
- `isocalendar()` (*datetime.datetime* のメソッド), 190
- `isoformat()` (*datetime.date* のメソッド), 183
- `isoformat()` (*datetime.datetime* のメソッド), 190
- `isoformat()` (*datetime.time* のメソッド), 194
- `isolation.level` (*sqlite3.Connection* の属性), 421
- `isowekday()` (*datetime.date* のメソッド), 182
- `isowekday()` (*datetime.datetime* のメソッド), 190
- `isprint()` (*curses.ascii* モジュール), 702
- `ispunct()` (*curses.ascii* モジュール), 702
- `isqueued()` (*fl* モジュール), 1758
- `isreadable()` (*pprint* モジュール), 265
- `isreadable()` (*pprint.PrettyPrinter* のメソッド), 266
- `isrecursive()` (*pprint* モジュール), 266
- `isrecursive()` (*pprint.PrettyPrinter* のメソッド), 266
- `isreg()` (*tarfile.TarInfo* のメソッド), 467
- `isReservedKey()` (*Cookie.Morsel* のメソッド), 1241
- `isroutine()` (*inspect* モジュール), 1574
- `isSameNode()` (*xml.dom.Node* のメソッド), 1070
- `isSequenceType()` (*operator* モジュール), 348
- `isSet()` (*threading.Event* のメソッド), 775
- `isspace()` (*curses.ascii* モジュール), 702
- `isspace()` (*str* のメソッド), 49
- `isstdin()` (*fileinput* モジュール), 359
- `issubclass()` (組み込み関数), 15
- `issubset()` (*frozenset* のメソッド), 61
- `issuite()` (*parser* モジュール), 1624
- `issuite()` (*parser.ST* のメソッド), 1625
- `issuperset()` (*frozenset* のメソッド), 61
- `issym()` (*tarfile.TarInfo* のメソッド), 467
- `ISTERMINAL()` (*token* モジュール), 1636
- `istitle()` (*str* のメソッド), 50
- `istraceback()` (*inspect* モジュール), 1574
- `isub()` (*operator* モジュール), 347
- `isupper()` (*curses.ascii* モジュール), 702
- `isupper()` (*str* のメソッド), 50
- `isvisible()` (*turtle* モジュール), 1377
- `isxdigit()` (*curses.ascii* モジュール), 702
- `item()` (*ttk.Treeview* のメソッド), 1346
- `item()` (*xml.dom.NamedNodeMap* のメソッド), 1075
- `item()` (*xml.dom.NodeList* のメソッド), 1071
- `itemgetter()` (*operator* モジュール), 349
- `items()` (*ConfigParser.ConfigParser* のメソッド), 486
- `items()` (*ConfigParser.RawConfigParser* のメソッド), 485

- `items()` (*dict* のメソッド), 66
- `items()` (*email.message.Message* のメソッド), 935
- `items()` (*mailbox.Mailbox* のメソッド), 986
- `items()` (*xml.etree.ElementTree.Element* のメソッド), 1059
- `itemsizedata()` (*array.array* の属性), 237
- `itemsizedata()` (*memoryview* の属性), 75
- `ItemsView` (*collections* のクラス), 227
- `iter()` (*xml.etree.ElementTree.Element* のメソッド), 1060
- `iter()` (*xml.etree.ElementTree.ElementTree* のメソッド), 1062
- `iter()` (組み込み関数), 15
- `iter_child_nodes()` (*ast* モジュール), 1631
- `iter_fields()` (*ast* モジュール), 1631
- `iter_importers()` (*pkgutil* モジュール), 1614
- `iter_modules()` (*pkgutil* モジュール), 1614
- `iterable`, 1784
- `Iterable` (*collections* のクラス), 226
- `IterableUserDict` (*UserDict* のクラス), 256
- `iterator`, 1784
- `Iterator` (*collections* のクラス), 227
- `iterator protocol`, 43
- `iterdecode()` (*codecs* モジュール), 154
- `iterdump()` (*sqlite3.Connection* の属性), 427
- `iterencode()` (*codecs* モジュール), 154
- `iterencode()` (*json.JSONEncoder* のメソッド), 980
- `iterfind()` (*xml.etree.ElementTree.Element* のメソッド), 1060
- `iterfind()` (*xml.etree.ElementTree.ElementTree* のメソッド), 1062
- `iteritems()` (*dict* のメソッド), 66
- `iteritems()` (*mailbox.Mailbox* のメソッド), 986
- `iterkeyrefs()` (*weakref.WeakKeyDictionary* のメソッド), 252
- `iterkeys()` (*dict* のメソッド), 66
- `iterkeys()` (*mailbox.Mailbox* のメソッド), 985
- `itermonthdates()` (*calendar.Calendar* のメソッド), 206
- `itermonthdays()` (*calendar.Calendar* のメソッド), 206
- `itermonthdays2()` (*calendar.Calendar* のメソッド), 206
- `iterparse()` (*xml.etree.ElementTree* モジュール), 1057
- `itertext()` (*xml.etree.ElementTree.Element* のメソッド), 1061
- `itertools (2to3 fixer)`, 1477
- `itertools` (モジュール), 323
- `itertools.imports (2to3 fixer)`, 1477
- `intervalrefs()` (*weakref.WeakValueDictionary* のメソッド), 253
- `intervalvalues()` (*dict* のメソッド), 66
- `intervalvalues()` (*mailbox.Mailbox* のメソッド), 985
- `iterweekdays()` (*calendar.Calendar* のメソッド), 206
- `ITIMER_PROF` (*signal* モジュール), 916
- `ITIMER_REAL` (*signal* モジュール), 915
- `ITIMER_VIRTUAL` (*signal* モジュール), 915
- `ItimerError`, 916
- `itruediv()` (*operator* モジュール), 347
- `ixor()` (*operator* モジュール), 348
- `izip()` (*itertools* モジュール), 331
- `izip_longest()` (*itertools* モジュール), 331
- Jansen, Jack, 1033
- `java_ver()` (*platform* モジュール), 707
- JFIF, 1768
- `join()` (*multiprocessing.JoinableQueue* のメソッド), 792
- `join()` (*multiprocessing.pool.multiprocessing.Pool* のメソッド), 811
- `join()` (*multiprocessing.Process* のメソッド), 787
- `join()` (*os.path* モジュール), 356
- `join()` (*Queue.Queue* のメソッド), 249
- `join()` (*str* のメソッド), 50
- `join()` (*string* モジュール), 105
- `join()` (*threading.Thread* のメソッド), 768
- `join_thread()` (*multiprocessing.Queue* のメソッド), 791
- `JoinableQueue` (*multiprocessing* のクラス), 792
- `joinfields()` (*string* モジュール), 105
- `jpeg` (モジュール), 1768
- `js_output()` (*Cookie.BaseCookie* のメソッド), 1240
- `js_output()` (*Cookie.Morsel* のメソッド), 1241
- `json` (モジュール), 972
- `JSONDecoder` (*json* のクラス), 977
- `JSONEncoder` (*json* のクラス), 978
- `JUMP_ABSOLUTE` (*opcode*), 1654
- `JUMP_FORWARD` (*opcode*), 1654
- `JUMP_IF_FALSE_OR_POP` (*opcode*), 1654
- `JUMP_IF_TRUE_OR_POP` (*opcode*), 1654
- `jumpahead()` (*random* モジュール), 319
- `kbhit()` (*msvcrt* モジュール), 1683
- KDEDIR, 1115
- `kevent()` (*select* モジュール), 758
- `key` (*Cookie.Morsel* の属性), 1241
- `key function`, 1785
- `KEY_ALL_ACCESS` (*_winreg* モジュール), 1691
- `KEY_CREATE_LINK` (*_winreg* モジュール), 1692
- `KEY_CREATE_SUB_KEY` (*_winreg* モジュール), 1692
- `KEY_ENUMERATE_SUB_KEYS` (*_winreg* モジュール), 1692
- `KEY_EXECUTE` (*_winreg* モジュール), 1692
- `KEY_NOTIFY` (*_winreg* モジュール), 1692
- `KEY_QUERY_VALUE` (*_winreg* モジュール), 1692
- `KEY_READ` (*_winreg* モジュール), 1691
- `KEY_SET_VALUE` (*_winreg* モジュール), 1692
- `KEY_WOW64_32KEY` (*_winreg* モジュール), 1692
- `KEY_WOW64_64KEY` (*_winreg* モジュール), 1692
- `KEY_WRITE` (*_winreg* モジュール), 1691
- `KeyboardInterrupt`, 83
- `KeyError`, 83
- `keyname()` (*curses* モジュール), 681
- `keypad()` (*curses.window* のメソッド), 690
- `keyrefs()` (*weakref.WeakKeyDictionary* のメソッド), 252
- `keys()` (*bsddb.bsddbobject* のメソッド), 414
- `keys()` (*dict* のメソッド), 66
- `keys()` (*email.message.Message* のメソッド), 935
- `keys()` (*mailbox.Mailbox* のメソッド), 985
- `keys()` (*sqlite3.Row* のメソッド), 431
- `keys()` (*xml.etree.ElementTree.Element* のメソッド), 1060
- `keysubst()` (*aetools* モジュール), 1744
- `KeysView` (*collections* のクラス), 227
- `Keyword` (*aetypes* のクラス), 1747
- `keyword` (モジュール), 1638
- `keyword argument`, 1785
- `keywords` (*functools.partial* の属性), 341
- `kill()` (*os* モジュール), 536
- `kill()` (*subprocess.Popen* のメソッド), 861
- `killchar()` (*curses* モジュール), 681
- `killpg()` (*os* モジュール), 537
- `knee`
 - モジュール, 1605, 1610
- `knownfiles` (*mimetypes* モジュール), 1013
- `kqueue()` (*select* モジュール), 758
- `kwlist` (*keyword* モジュール), 1638
- `L` (*re* モジュール), 114
- `label()` (*EasyDialogs.ProgressBar* のメソッド), 1730
- `LabelEntry` (*Tix* のクラス), 1354
- `LabelFrame` (*Tix* のクラス), 1354
- `lambda`, 1785
- `LambdaType` (*types* モジュール), 260
- LANG, 1285, 1287, 1298, 1302
- LANGUAGE, 1285, 1287
- `language`
 - C, 39
- `large files`, 1697
- `LargeZipFile`, 451
- `last` (*multifile.MultiFile* の属性), 1020
- `last()` (*bsddb.bsddbobject* のメソッド), 414
- `last()` (*dbhash.dbhash* のメソッド), 411
- `last()` (*nntplib.NNTP* のメソッド), 1189
- `last_accepted` (*multiprocessing.connection.Listener* の属性), 813
- `last_traceback` (*sys* モジュール), 1536
- `last.type` (*sys* モジュール), 1536
- `last.value` (*sys* モジュール), 1536

- `lastChild` (*xml.dom.Node* の属性), 1069
- `lastcmd` (*cmd.Cmd* の属性), 1309
- `lastgroup` (*re.MatchObject* の属性), 122
- `lastindex` (*re.MatchObject* の属性), 122
- `lastpart` () (*MimeWriter.MimeWriter* のメソッド), 1016
- `lastrowid` (*sqlite3.Cursor* の属性), 430
- `launch` () (*findertools* モジュール), 1726
- `launchurl` () (*ic* モジュール), 1722
- `launchurl` () (*ic.IC* のメソッド), 1722
- `layout` () (*ttk.Style* のメソッド), 1349
- `LBACE` (*token* モジュール), 1636
- `LBYL`, 1785
- `LC.ALL`, 1285, 1287
- `LC.ALL` (*locale* モジュール), 1304
- `LC.COLLATE` (*locale* モジュール), 1304
- `LC.CTYPE` (*locale* モジュール), 1304
- `LC.MESSAGES`, 1285, 1287
- `LC.MESSAGES` (*locale* モジュール), 1304
- `LC.MONETARY` (*locale* モジュール), 1304
- `LC.NUMERIC` (*locale* モジュール), 1304
- `LC.TIME` (*locale* モジュール), 1304
- `lchflags` () (*os* モジュール), 523
- `lchmod` () (*os* モジュール), 523
- `lchown` () (*os* モジュール), 524
- `ldexp` () (*math* モジュール), 276
- `ldgettext` () (*gettext* モジュール), 1286
- `ldngettext` () (*gettext* モジュール), 1287
- `le` () (*operator* モジュール), 342
- `leapdays` () (*calendar* モジュール), 208
- `leaveok` () (*curses.window* のメソッド), 690
- `left` (*filecmp.dircmp* の属性), 368
- `left` () (*turtle* モジュール), 1364
- `left.list` (*filecmp.dircmp* の属性), 368
- `left.only` (*filecmp.dircmp* の属性), 368
- `LEFTSHIFT` (*token* モジュール), 1636
- `LEFTSHIFTEQUAL` (*token* モジュール), 1636
- `len`
 - 組み込み関数, 46, 63
- `len` () (組み込み関数), 16
- `length` (*xml.dom.NamedNodeMap* の属性), 1075
- `length` (*xml.dom.NodeList* の属性), 1071
- `LESS` (*token* モジュール), 1636
- `LESSEQUAL` (*token* モジュール), 1636
- `letters` (*string* モジュール), 92
- `level` (*multifile.MultiFile* の属性), 1020
- `lexists` () (*os.path* モジュール), 354
- `lgamma` () (*math* モジュール), 279
- `lgettext` () (*gettext* モジュール), 1286
- `lgettext` () (*gettext.GNUTranslations* のメソッド), 1291
- `lgettext` () (*gettext.NullTranslations* のメソッド), 1289
- `lib2to3` (モジュール), 1479
- `libc.ver` () (*platform* モジュール), 708
- `library` (*dbm* モジュール), 408
- `library` (*ssl.SSLError* の属性), 884
- `LibraryLoader` (*ctypes* のクラス), 741
- `license` (組み込み変数), 36
- `LifoQueue` (*Queue* のクラス), 247
- `light-weight processes`, 777
- `limit_denominator` () (*fractions.Fraction* のメソッド), 317
- `lin2adpcm` () (*audioop* モジュール), 1260
- `lin2alaw` () (*audioop* モジュール), 1261
- `lin2lin` () (*audioop* モジュール), 1261
- `lin2ulaw` () (*audioop* モジュール), 1261
- `line` () (*msilib.Dialog* のメソッド), 1681
- `line-buffered I/O`, 18
- `line.buffering` (*io.TextIOWrapper* の属性), 557
- `line_num` (*csv.csvreader* の属性), 477
- `linecache` (モジュール), 376
- `lineno` (*ast.AST* の属性), 1627
- `lineno` (*doctest.DocTest* の属性), 1431
- `lineno` (*doctest.Example* の属性), 1432
- `lineno` (*pyclbr.Class* の属性), 1642
- `lineno` (*pyclbr.Function* の属性), 1642
- `lineno` (*shlex.shlex* の属性), 1313
- `lineno` (*xml.parsers.expat.ExpatError* の属性), 1107
- `lineno` () (*fileinput* モジュール), 359
- `LINES`, 679, 684
- `linesep` (*os* モジュール), 544
- `lineterminator` (*csv.Dialect* の属性), 476
- `link` () (*os* モジュール), 524
- `linkmodel` (*MacOS* モジュール), 1723
- `linkname` (*tarfile.TarInfo* の属性), 466
- `linux.distribution` () (*platform* モジュール), 708
- `list`, 1785
- `list`
 - type, operations on, 58
 - オブジェクト, 45, 58
- `list comprehension`, 1785
- `list` () (*imaplib.IMAP4* のメソッド), 1181
- `list` () (*multiprocessing.managers.SyncManager* のメソッド), 804
- `list` () (*nntplib.NNTP* のメソッド), 1188
- `list` () (*poplib.POP3* のメソッド), 1176
- `list` () (*tarfile.TarFile* のメソッド), 463
- `LISTAPPEND` (*opcode*), 1652
- `list.dialects` () (*csv* モジュール), 473
- `list_folders` () (*mailbox.Maildir* のメソッド), 988
- `list_folders` () (*mailbox.MH* のメソッド), 991
- `listallfolders` () (*mhlib.MH* のメソッド), 1008
- `listallsubfolders` () (*mhlib.MH* のメソッド), 1008
- `listdir` () (*dircache* モジュール), 382
- `listdir` () (*os* モジュール), 524
- `listen` () (*asyncore.dispatcher* のメソッド), 924
- `listen` () (*logging.config* モジュール), 654
- `listen` () (*socket.socket* のメソッド), 877
- `listen` () (*turtle* モジュール), 1386
- `Listener` (*multiprocessing.connection* のクラス), 812
- `listfolders` () (*mhlib.MH* のメソッド), 1008
- `listmessages` () (*mhlib.Folder* のメソッド), 1009
- `listMethods` () (*xmlrpclib.ServerProxy.system* のメソッド), 1245
- `ListNoteBook` (*Tix* のクラス), 1356
- `listsubfolders` () (*mhlib.MH* のメソッド), 1008
- `ListType` (*types* モジュール), 260
- `literal_eval` () (*ast* モジュール), 1631
- `literals`
 - complex number, 39
 - floating point, 39
 - hexadecimal, 39
 - integer, 39
 - long integer, 39
 - numeric, 39
 - octal, 39
- `LittleEndianStructure` (*ctypes* のクラス), 754
- `ljust` () (*str* のメソッド), 50
- `ljust` () (*string* モジュール), 106
- `LK.LOCK` (*msvcrt* モジュール), 1682
- `LK.NBLCK` (*msvcrt* モジュール), 1682
- `LK.NBRLCK` (*msvcrt* モジュール), 1682
- `LK.RLCK` (*msvcrt* モジュール), 1682
- `LK.UNLCK` (*msvcrt* モジュール), 1683
- `LMTP` (*smtplib* のクラス), 1191
- `ln` () (*decimal.Context* のメソッド), 303
- `ln` () (*decimal.Decimal* のメソッド), 294
- `LNAME`, 676
- `lngettext` () (*gettext* モジュール), 1286
- `lngettext` () (*gettext.GNUTranslations* のメソッド), 1292
- `lngettext` () (*gettext.NullTranslations* のメソッド), 1289
- `load` () (*Cookie.BaseCookie* のメソッド), 1240
- `load` () (*cookielib.FileCookieJar* のメソッド), 1231
- `load` () (*hotshot.stats* モジュール), 1513
- `load` () (*json* モジュール), 976
- `load` () (*marshal* モジュール), 405
- `load` () (*pickle* モジュール), 388
- `load` () (*pickle.Unpickler* のメソッド), 390
- `LOAD.ATTR` (*opcode*), 1654

- `load_cert_chain()` (*ssl.SSLContext* のメソッド), 901
- `LOAD_CLOSURE` (*opcode*), 1655
- `load_compiled()` (*imp* モジュール), 1604
- `LOAD_CONST` (*opcode*), 1653
- `load_default_certs()` (*ssl.SSLContext* のメソッド), 901
- `LOAD_DEREF` (*opcode*), 1655
- `load_dh_params()` (*ssl.SSLContext* のメソッド), 904
- `load_dynamic()` (*imp* モジュール), 1604
- `load_extension()` (*sqlite3.Connection* のメソッド), 425
- `LOAD_FAST` (*opcode*), 1655
- `LOAD_GLOBAL` (*opcode*), 1655
- `load_global()` (*pickle protocol*), 396
- `LOAD_LOCALS` (*opcode*), 1652
- `load_module()` (*imp* モジュール), 1602
- `load_module()` (*zipimport.zipimporter* のメソッド), 1612
- `LOAD_NAME` (*opcode*), 1653
- `load_source()` (*imp* モジュール), 1604
- `load_verify_locations()` (*ssl.SSLContext* のメソッド), 902
- `loader`, 1785
- `LoadError`, 1227
- `LoadKey()` (*._winreg* モジュール), 1687
- `LoadLibrary()` (*ctypes.LibraryLoader* のメソッド), 741
- `loads()` (*json* モジュール), 977
- `loads()` (*marshal* モジュール), 405
- `loads()` (*pickle* モジュール), 388
- `loads()` (*xmlrpclib* モジュール), 1251
- `loadTestsFromModule()` (*unittest.TestLoader* のメソッド), 1463
- `loadTestsFromName()` (*unittest.TestLoader* のメソッド), 1464
- `loadTestsFromNames()` (*unittest.TestLoader* のメソッド), 1464
- `loadTestsFromTestCase()` (*unittest.TestLoader* のメソッド), 1463
- `local` (*threading* のクラス), 765
- `localcontext()` (*decimal* モジュール), 298
- `LOCALE` (*re* モジュール), 114
- `locale` (モジュール), 1297
- `localeconv()` (*locale* モジュール), 1298
- `LocaleHTMLCalendar` (*calendar* のクラス), 208
- `LocaleTextCalendar` (*calendar* のクラス), 207
- `localName` (*xml.dom.Attr* の属性), 1074
- `localName` (*xml.dom.Node* の属性), 1069
- `locals()` (組み込み関数), 16
- `localtime()` (*time* モジュール), 562
- `Locator` (*xml.sax.xmlreader* のクラス), 1095
- `Lock` (*multiprocessing* のクラス), 796
- `lock()` (*mailbox.Babyl* のメソッド), 993
- `lock()` (*mailbox.Mailbox* のメソッド), 987
- `lock()` (*mailbox.Maildir* のメソッド), 989
- `lock()` (*mailbox.mbox* のメソッド), 990
- `lock()` (*mailbox.MH* のメソッド), 992
- `lock()` (*mailbox.MMDF* のメソッド), 994
- `Lock()` (*multiprocessing.managers.SyncManager* のメソッド), 804
- `lock()` (*mutex.mutex* のメソッド), 247
- `lock()` (*posixfile.posixfile* のメソッド), 1711
- `Lock()` (*threading* モジュール), 765
- `lock_held()` (*imp* モジュール), 1602
- `locked()` (*thread.lock* のメソッド), 778
- `locked()` (*threading.Lock* のメソッド), 770
- `lockf()` (*fcntl* モジュール), 1707
- `locking()` (*msvcrt* モジュール), 1682
- `LockType` (*thread* モジュール), 777
- `log()` (*cmath* モジュール), 281
- `log()` (*logging* モジュール), 650
- `log()` (*logging.Logger* のメソッド), 640
- `log()` (*math* モジュール), 277
- `log_date_time_string()`
(*BaseHTTPServer.BaseHTTPRequestHandler* のメソッド), 1223
- `log_error()` (*BaseHTTPServer.BaseHTTPRequestHandler* のメソッド), 1222
- `log_exception()` (*wsgiref.handlers.BaseHandler* のメソッド), 1135
- `log_message()` (*BaseHTTPServer.BaseHTTPRequestHandler* のメソッド), 1222
- `log_request()` (*BaseHTTPServer.BaseHTTPRequestHandler* のメソッド), 1222
- `log_to_stderr()` (*multiprocessing* モジュール), 815
- `log10()` (*cmath* モジュール), 281
- `log10()` (*decimal.Context* のメソッド), 303
- `log10()` (*decimal.Decimal* のメソッド), 294
- `log10()` (*math* モジュール), 277
- `log1p()` (*math* モジュール), 277
- `logb()` (*decimal.Context* のメソッド), 303
- `logb()` (*decimal.Decimal* のメソッド), 294
- `Logger` (*logging* のクラス), 638
- `LoggerAdapter` (*logging* のクラス), 648
- `logging`
Errors, 637
- `logging` (モジュール), 637
- `logging.config` (モジュール), 653
- `logging.handlers` (モジュール), 665
- `Logical` (*aetypes* のクラス), 1747
- `logical_and()` (*decimal.Context* のメソッド), 303
- `logical_and()` (*decimal.Decimal* のメソッド), 294
- `logical_invert()` (*decimal.Context* のメソッド), 303
- `logical_invert()` (*decimal.Decimal* のメソッド), 294
- `logical_or()` (*decimal.Context* のメソッド), 303
- `logical_or()` (*decimal.Decimal* のメソッド), 294
- `logical_xor()` (*decimal.Context* のメソッド), 303
- `logical_xor()` (*decimal.Decimal* のメソッド), 294
- `login()` (*ftplib.FTP* のメソッド), 1172
- `login()` (*imaplib.IMAP4* のメソッド), 1181
- `login()` (*smtpplib.SMTP* のメソッド), 1193
- `login_cram_md5()` (*imaplib.IMAP4* のメソッド), 1181
- `LOGNAME`, 510, 676
- `lognormvariate()` (*random* モジュール), 321
- `logout()` (*imaplib.IMAP4* のメソッド), 1181
- `LogRecord` (*logging* のクラス), 645
- `long`
integer division, 40
integer literals, 39
組み込み関数, 39, 104
- `long` (*2to3 fixer*), 1477
- `long` (組み込みクラス), 16
- `long integer`
オブジェクト, 39
- `long.info` (*sys* モジュール), 1535
- `longMessage` (*unittest.TestCase* の属性), 1460
- `longname()` (*curses* モジュール), 681
- `LongType` (*types* モジュール), 259
- `lookup()` (*codecs* モジュール), 152
- `lookup()` (*symtable.SymbolTable* のメソッド), 1634
- `lookup()` (*ttk.Style* のメソッド), 1349
- `lookup()` (*unicodedata* モジュール), 170
- `lookup_error()` (*codecs* モジュール), 153
- `LookupError`, 82
- `loop()` (*asyncore* モジュール), 922
- `lower()` (*str* のメソッド), 50
- `lower()` (*string* モジュール), 104
- `lowercase` (*string* モジュール), 92
- `LPAR` (*token* モジュール), 1636
- `lseek()` (*os* モジュール), 517
- `lshift()` (*operator* モジュール), 343
- `LSQB` (*token* モジュール), 1636
- `lstat()` (*os* モジュール), 524
- `lstrip()` (*str* のメソッド), 50
- `lstrip()` (*string* モジュール), 105
- `lsub()` (*imaplib.IMAP4* のメソッド), 1181
- `lt()` (*operator* モジュール), 342
- `lt()` (*turtle* モジュール), 1364
- `Lundh, Fredrik`, 1768
- `LWPCookieJar` (*cookielib* のクラス), 1232

- M (*re* モジュール), 114
- mac_ver() (*platform* モジュール), 707
- macerrors
 - モジュール, 1723
- macerrors (モジュール), 1776
- machine() (*platform* モジュール), 705
- MacOS (モジュール), 1723
- macostools (モジュール), 1725
- macpath (モジュール), 383
- macresource (モジュール), 1776
- macros (*netrc.netrc* の属性), 491
- magic
 - method, 1786
- magic method, 1786
- mailbox
 - モジュール, 1021
- Mailbox (*mailbox* のクラス), 984
- mailbox (モジュール), 984
- mailcap (モジュール), 982
- Maildir (*mailbox* のクラス), 988
- MaildirMessage (*mailbox* のクラス), 995
- MailmanProxy (*smtpd* のクラス), 1197
- main() (*py.compile* モジュール), 1643
- main() (*unittest* モジュール), 1468
- mainloop() (*FrameWork.Application* のメソッド), 1732
- mainloop() (*turtle* モジュール), 1380
- major() (*os* モジュール), 525
- make_archive() (*shutil* モジュール), 380
- MAKE_CLOSURE (*opcode*), 1656
- make_cookies() (*cookielib.CookieJar* のメソッド), 1230
- make_form() (*fl* モジュール), 1757
- MAKE_FUNCTION (*opcode*), 1656
- make_header() (*email.header* モジュール), 952
- make_msgid() (*email.utils* モジュール), 960
- make_parser() (*xml.sax* モジュール), 1086
- make_server() (*wsgiref.simple_server* モジュール), 1130
- makedev() (*os* モジュール), 525
- makedirs() (*os* モジュール), 525
- makeelement() (*xml.etree.ElementTree.Element* のメソッド), 1061
- makefile() (*socket.socket* のメソッド), 877
- makefolder() (*mhlib.MH* のメソッド), 1008
- makeLogRecord() (*logging* モジュール), 651
- makePickle() (*logging.handlers.SocketHandler* のメソッド), 669
- makeRecord() (*logging.Logger* のメソッド), 641
- makeSocket() (*logging.handlers.DatagramHandler* のメソッド), 670
- makeSocket() (*logging.handlers.SocketHandler* のメソッド), 669
- maketrans() (*string* モジュール), 103
- makeusermenus() (*FrameWork.Application* のメソッド), 1732
- map (*2to3 fixer*), 1477
- map() (*future.builtins* モジュール), 1547
- map() (*multiprocessing.pool.multiprocessing.Pool* のメソッド), 810
- map() (*tk.Style* のメソッド), 1348
- map() (組み込み関数), 17
- map_async() (*multiprocessing.pool.multiprocessing.Pool* のメソッド), 810
- map_table_b2() (*stringprep* モジュール), 173
- map_table_b3() (*stringprep* モジュール), 173
- mapcolor() (*fl* モジュール), 1758
- mapfile() (*ic* モジュール), 1722
- mapfile() (*ic.IC* のメソッド), 1722
- mapLogRecord() (*logging.handlers.HTTPHandler* のメソッド), 675
- mapping, 1786
- mapping
 - types, operations on, 63
- オブジェクト, 63
- Mapping (*collections* のクラス), 227
- mapping() (*msilib.Control* のメソッド), 1680
- MapView (*collections* のクラス), 227
- mapPriority() (*logging.handlers.SysLogHandler* のメソッド), 672
- maps() (*nis* モジュール), 1717
- maptypecreator() (*ic* モジュール), 1722
- maptypecreator() (*ic.IC* のメソッド), 1723
- marshal (モジュール), 403
- marshalling
 - objects, 385
- masking
 - operations, 41
- match() (*nis* モジュール), 1717
- match() (*re* モジュール), 115
- match() (*re.RegexObject* のメソッド), 118
- match_hostname() (*ssl* モジュール), 889
- MatchObject (*re* のクラス), 119
- math
 - モジュール, 40, 283
- math (モジュール), 275
- max
 - 組み込み関数, 46
- max (*datetime.date* の属性), 181
- max (*datetime.datetime* の属性), 186
- max (*datetime.time* の属性), 193
- max (*datetime.timedelta* の属性), 178
- max() (*audioop* モジュール), 1261
- max() (*decimal.Context* のメソッド), 303
- max() (*decimal.Decimal* のメソッド), 294
- max() (組み込み関数), 17
- MAX_INTERPOLATION_DEPTH (*ConfigParser* モジュール), 483
- max_mag() (*decimal.Context* のメソッド), 303
- max_mag() (*decimal.Decimal* のメソッド), 295
- maxarray (*repr.Repr* の属性), 268
- maxdeque (*repr.Repr* の属性), 268
- maxdict (*repr.Repr* の属性), 268
- maxDiff (*unittest.TestCase* の属性), 1460
- maxfrozenset (*repr.Repr* の属性), 268
- maxint (*sys* モジュール), 1536
- maxlen (*collections.deque* の属性), 215
- MAXLEN (*mimify* モジュール), 1017
- maxlevel (*repr.Repr* の属性), 268
- maxlist (*repr.Repr* の属性), 268
- maxlong (*repr.Repr* の属性), 268
- maxother (*repr.Repr* の属性), 268
- maxpp() (*audioop* モジュール), 1261
- maxset (*repr.Repr* の属性), 268
- maxsize (*sys* モジュール), 1536
- maxstring (*repr.Repr* の属性), 268
- maxtuple (*repr.Repr* の属性), 268
- maxunicode (*sys* モジュール), 1536
- maxval (*EasyDialogs.ProgressBar* の属性), 1729
- MAXYEAR (*datetime* モジュール), 176
- MB_ICONASTERISK (*winsound* モジュール), 1696
- MB_ICONEXCLAMATION (*winsound* モジュール), 1696
- MB_ICONHAND (*winsound* モジュール), 1696
- MB_ICONQUESTION (*winsound* モジュール), 1696
- MB_OK (*winsound* モジュール), 1696
- mbox (*mailbox* のクラス), 990
- mboxMessage (*mailbox* のクラス), 997
- md5 (モジュール), 503
- md5() (*md5* モジュール), 504
- MemberDescriptorType (*types* モジュール), 261
- memmove() (*ctypes* モジュール), 748
- MemoryError, 83
- MemoryHandler (*logging.handlers* のクラス), 675
- memoryview (組み込みクラス), 74
- memset() (*ctypes* モジュール), 748
- Menu() (*FrameWork* モジュール), 1731
- MenuBar() (*FrameWork* モジュール), 1731
- MenuItem() (*FrameWork* モジュール), 1731
- merge() (*heapq* モジュール), 229
- Message (*email.message* のクラス), 932
- Message (*in module mimetools*), 1221

- Message (*mailbox* のクラス), 994
- Message (*mhlib* のクラス), 1008
- Message (*mimetools* のクラス), 1010
- Message (*rfc822* のクラス), 1021
- message.digest, MD5, 499, 503
- Message() (*EasyDialogs* モジュール), 1727
- message.from_file() (*email* モジュール), 943
- message.from_string() (*email* モジュール), 943
- MessageBeep() (*winsound* モジュール), 1694
- MessageClass (*BaseHTTPServer.BaseHTTPRequestHandler* の属性), 1221
- MessageError, 957
- MessageParseError, 957
- meta() (*curses* モジュール), 681
- meta.path (sys モジュール), 1536
- metaclass, 1786
- metaclass (2to3 fixer), 1477
- metavar (*optparse.Option* の属性), 618
- Meter (*Tix* のクラス), 1354
- method, 1786
- method
 - magic, 1786
 - special, 1789
 - オブジェクト, 78
- method resolution order, 1786
- methodattrs (2to3 fixer), 1477
- methodcaller() (*operator* モジュール), 350
- methodHelp() (*xmlrpclib.ServerProxy.system* のメソッド), 1246
- methods
 - string, 47
- methods (*pyclbr.Class* の属性), 1642
- methodSignature() (*xmlrpclib.ServerProxy.system* のメソッド), 1245
- MethodType (*types* モジュール), 260
- MH (*mailbox* のクラス), 991
- MH (*mhlib* のクラス), 1008
- mhlib (モジュール), 1008
- MHMailbox (*mailbox* のクラス), 1006
- MHMessage (*mailbox* のクラス), 999
- microsecond (*datetime.datetime* の属性), 187
- microsecond (*datetime.time* の属性), 193
- MIME
 - base64 encoding, 1025
 - content type, 1012
 - headers, 1012, 1116
 - quoted-printable encoding, 1031
- mime.decode_header() (*mimify* モジュール), 1017
- mime.encode_header() (*mimify* モジュール), 1017
- MIMEApplication (*email.mime.application* のクラス), 948
- MIMEAudio (*email.mime.audio* のクラス), 948
- MIMEBase (*email.mime.base* のクラス), 947
- MIMEImage (*email.mime.image* のクラス), 948
- MIMEMessage (*email.mime.message* のクラス), 949
- MIMEMultipart (*email.mime.multipart* のクラス), 947
- MIMENonMultipart (*email.mime.nonmultipart* のクラス), 947
- MIMEText (*email.mime.text* のクラス), 949
- mimetools
 - モジュール, 1138
- mimetools (モジュール), 1010
- MimeTypes (*mimetypes* のクラス), 1014
- mimetypes (モジュール), 1012
- MimeWriter (*MimeWriter* のクラス), 1015
- MimeWriter (モジュール), 1015
- mimify (モジュール), 1016
- mimify() (*mimify* モジュール), 1017
- min
 - 組み込み関数, 46
- min (*datetime.date* の属性), 181
- min (*datetime.datetime* の属性), 186
- min (*datetime.time* の属性), 193
- min (*datetime.timedelta* の属性), 178
- min() (*decimal.Context* のメソッド), 303
- min() (*decimal.Decimal* のメソッド), 295
- min() (組み込み関数), 17
- min_mag() (*decimal.Context* のメソッド), 304
- min_mag() (*decimal.Decimal* のメソッド), 295
- MINEQUAL (*token* モジュール), 1636
- MiniAETFrame (モジュール), 1748
- MiniApplication (*MiniAETFrame* のクラス), 1748
- minmax() (*audioop* モジュール), 1261
- minor() (*os* モジュール), 525
- MINUS (*token* モジュール), 1636
- minus() (*decimal.Context* のメソッド), 304
- minute (*datetime.datetime* の属性), 186
- minute (*datetime.time* の属性), 193
- MINYEAR (*datetime* モジュール), 176
- mirrored() (*unicodedata* モジュール), 171
- misc.header (*cmd.Cmd* の属性), 1309
- MissingSectionHeaderError, 483
- MIXERDEV, 1279
- mkalias() (*macostools* モジュール), 1725
- mkd() (*ftplib.FTP* のメソッド), 1174
- mkdir() (*os* モジュール), 525
- mkdtemp() (*tempfile* モジュール), 371
- mkfifo() (*os* モジュール), 524
- mknod() (*os* モジュール), 524
- mkstemp() (*tempfile* モジュール), 371
- mktemp() (*tempfile* モジュール), 372
- mktime() (*time* モジュール), 562
- mktime.tz() (*email.utils* モジュール), 959
- mktime.tz() (*rfc822* モジュール), 1022
- mmap (*mmap* のクラス), 843
- mmap (モジュール), 842
- MMDF (*mailbox* のクラス), 993
- MmdfMailbox (*mailbox* のクラス), 1005
- MMDFMessage (*mailbox* のクラス), 1002
- mod() (*operator* モジュール), 344
- mode (*file* の属性), 73
- mode (*io.FileIO* の属性), 553
- mode (*ossaudiodev.oss_audio_device* の属性), 1282
- mode (*tarfile.TarInfo* の属性), 466
- mode() (*turtle* モジュール), 1387
- modf() (*math* モジュール), 276
- modified() (*robotparser.RobotFileParser* のメソッド), 490
- Modify() (*msilib.View* のメソッド), 1677
- modify() (*select.epoll* のメソッド), 759
- modify() (*select.poll* のメソッド), 760
- module
 - search path, 376, 1536, 1579
- module (*pyclbr.Class* の属性), 1642
- module (*pyclbr.Function* の属性), 1642
- module() (*new* モジュール), 262
- ModuleFinder (*modulefinder* のクラス), 1616
- modulefinder (モジュール), 1615
- modules (*modulefinder.ModuleFinder* の属性), 1616
- modules (sys モジュール), 1536
- ModuleType (*types* モジュール), 260
- mono2grey() (*imageop* モジュール), 1264
- month (*datetime.date* の属性), 181
- month (*datetime.datetime* の属性), 186
- month() (*calendar* モジュール), 209
- month.abbrev (*calendar* モジュール), 209
- month.name (*calendar* モジュール), 209
- monthcalendar() (*calendar* モジュール), 209
- monthdatescalendar() (*calendar.Calendar* のメソッド), 206
- monthdays2calendar() (*calendar.Calendar* のメソッド), 206
- monthdayscalendar() (*calendar.Calendar* のメソッド), 206
- monthrangle (*calendar* モジュール), 208
- Morsel (*Cookie* のクラス), 1240
- most_common() (*collections.Counter* のメソッド), 211
- mouseinterval() (*curses* モジュール), 681
- mousemask() (*curses* モジュール), 681
- move() (*curses.panel.Panel* のメソッド), 704

move() (*curses.window* のメソッド), 690
 move() (*findertools* モジュール), 1726
 move() (*mmap.mmap* のメソッド), 845
 move() (*shutil* モジュール), 378
 move() (*tk.Treeview* のメソッド), 1346
 movemessage() (*mhlib.Folder* のメソッド), 1010
 MozillaCookieJar (*cookielib* のクラス), 1231
 MRO, 1786
 mro() (*class* のメソッド), 80
 msftoframe() (*cd* モジュール), 1752
 msg (*httplib.HTTPResponse* の属性), 1167
 msg() (*telnetlib.Telnet* のメソッド), 1199
 msi, 1675
 msilib (モジュール), 1675
 msvcr7 (モジュール), 1682
 mt_interact() (*telnetlib.Telnet* のメソッド), 1199
 mtime (*tarfile.TarInfo* の属性), 466
 mtime() (*robotparser.RobotFileParser* のメソッド), 489
 mul() (*audioop* モジュール), 1261
 mul() (*operator* モジュール), 344
 MultiCall (*xmlrpclib* のクラス), 1250
 MultiFile (*multifile* のクラス), 1018
 multifile (モジュール), 1018
 MULTILINE (*re* モジュール), 114
 MultipartConversionError, 957
 multiply() (*decimal.Context* のメソッド), 304
 multiprocessing (モジュール), 780
 multiprocessing.connection (モジュール), 812
 multiprocessing.dummy (モジュール), 816
 multiprocessing.Manager()
 (*multiprocessing.sharedctypes* モジュール), 802
 multiprocessing.managers (モジュール), 802
 multiprocessing.Pool (*multiprocessing.pool* のクラス), 809
 multiprocessing.pool (モジュール), 809
 multiprocessing.queue.SimpleQueue
 (*multiprocessing* のクラス), 792
 multiprocessing.sharedctypes (モジュール), 799
 mutable, 1786
 mutable
 sequence types, 58
 MutableMapping (*collections* のクラス), 227
 MutableSequence (*collections* のクラス), 227
 MutableSet (*collections* のクラス), 227
 MutableString (*UserString* のクラス), 258
 mutex (*mutex* のクラス), 246
 mutex (モジュール), 246
 mvderwin() (*curses.window* のメソッド), 690
 mvwin() (*curses.window* のメソッド), 690
 myrights() (*imaplib.IMAP4* のメソッド), 1182

 N_TOKENS (*token* モジュール), 1636
 name (*cookielib.Cookie* の属性), 1236
 name (*doctest.DocTest* の属性), 1431
 name (*file* の属性), 73
 name (*io.FileIO* の属性), 553
 name (*multiprocessing.Process* の属性), 787
 name (*os* モジュール), 507
 name (*ossaudiodev.oss.audio_device* の属性), 1282
 name (*pyclbr.Class* の属性), 1642
 name (*pyclbr.Function* の属性), 1642
 name (*tarfile.TarInfo* の属性), 466
 name (*threading.Thread* の属性), 769
 NAME (*token* モジュール), 1636
 name (*xml.dom.Attr* の属性), 1074
 name (*xml.dom.DocumentType* の属性), 1072
 name() (*unicodedata* モジュール), 170
 name2codepoint (*htmlentitydefs* モジュール), 1047
 named tuple, 1786
 NamedTemporaryFile() (*tempfile* モジュール), 370
 namedtuple() (*collections* モジュール), 219
 NameError, 84
 namelist() (*zipfile.ZipFile* のメソッド), 452
 nameprep() (*encodings.idna* モジュール), 169

namespace, 1786
 Namespace (*argparse* のクラス), 592
 Namespace (*multiprocessing.managers* のクラス), 805
 namespace() (*imaplib.IMAP4* のメソッド), 1182
 Namespace() (*multiprocessing.managers.SyncManager* のメソッド), 804
 NAMESPACE.DNS (*uuid* モジュール), 1203
 NAMESPACE.OID (*uuid* モジュール), 1203
 NAMESPACE.URL (*uuid* モジュール), 1203
 NAMESPACE.X500 (*uuid* モジュール), 1203
 NamespaceErr, 1077
 namespaceURI (*xml.dom.Node* の属性), 1069
 NaN, 13, 103
 NannyNag, 1641
 napms() (*curses* モジュール), 681
 nargs (*optparse.Option* の属性), 618
 Nav (モジュール), 1777
 Navigation Services, 1728
 ndiff() (*difflib* モジュール), 135
 ndim (*memoryview* の属性), 75
 ne (*2to3 fixer*), 1477
 ne() (*operator* モジュール), 342
 neg() (*operator* モジュール), 344
 nested scope, 1787
 nested() (*contextlib* モジュール), 1556
 netrc (*netrc* のクラス), 490
 netrc (モジュール), 490
 NetrcParseError, 490
 netscape (*cookielib.CookiePolicy* の属性), 1233
 Network News Transfer Protocol, 1185
 new (モジュール), 262
 new() (*hmac* モジュール), 502
 new() (*md5* モジュール), 504
 new() (*sha* モジュール), 505
 new-style class, 1787
 new_alignment() (*formatter.writer* のメソッド), 1672
 new_font() (*formatter.writer* のメソッド), 1672
 new_margin() (*formatter.writer* のメソッド), 1672
 new_module() (*imp* モジュール), 1602
 new_panel() (*curses.panel* モジュール), 703
 new_spacing() (*formatter.writer* のメソッド), 1672
 new_styles() (*formatter.writer* のメソッド), 1672
 newconfig() (*al* モジュール), 1749
 newgroups() (*nntplib.NNTP* のメソッド), 1187
 NEWLINE (*token* モジュール), 1636
 newlines (*file* の属性), 73
 newlines (*io.TextIOBase* の属性), 556
 newnews() (*nntplib.NNTP* のメソッド), 1187
 newpad() (*curses* モジュール), 681
 newwin() (*curses* モジュール), 682
 next (*2to3 fixer*), 1477
 next() (*bsddb.bsddbobject* のメソッド), 414
 next() (*csv.csvreader* のメソッド), 477
 next() (*dbhash.dbhash* のメソッド), 412
 next() (*file* のメソッド), 70
 next() (*iterator* のメソッド), 44
 next() (*mailbox.oldmailbox* のメソッド), 1004
 next() (*multifile.MultiFile* のメソッド), 1019
 next() (*nntplib.NNTP* のメソッド), 1189
 next() (*tarfile.TarFile* のメソッド), 463
 next() (*tk.Treeview* のメソッド), 1346
 next() (組み込み関数), 17
 next_minus() (*decimal.Context* のメソッド), 304
 next_minus() (*decimal.Decimal* のメソッド), 295
 next_plus() (*decimal.Context* のメソッド), 304
 next_plus() (*decimal.Decimal* のメソッド), 295
 next_toward() (*decimal.Context* のメソッド), 304
 next_toward() (*decimal.Decimal* のメソッド), 295
 nextfile() (*fileinput* モジュール), 359
 nextkey() (*gdbm* モジュール), 410
 nextpart() (*MimeWriter.MimeWriter* のメソッド), 1016
 nextSibling (*xml.dom.Node* の属性), 1069
 ngettext() (*gettext* モジュール), 1286

ngettext() (*gettext.GNUTranslations* のメソッド), 1291
 ngettext() (*gettext.NullTranslations* のメソッド), 1289
 nice() (*os* モジュール), 537
 nis (モジュール), 1717
 NIST, 505
 NL (*tokenize* モジュール), 1639
 nl() (*curses* モジュール), 682
 nl_langinfo() (*locale* モジュール), 1300
 nlargest() (*heapq* モジュール), 229
 nlst() (*ftplib.FTP* のメソッド), 1173
 NNTP
 protocol, 1185
 NNTP (*nntplib* のクラス), 1186
 NNTPDataError, 1187
 NNTPError, 1186
 nntplib (モジュール), 1185
 NNTPPermanentError, 1187
 NNTPProtocolError, 1187
 NNTPReplyError, 1186
 NNTPTemporaryError, 1186
 no_proxy, 1139
 nobreak() (*curses* モジュール), 682
 NoDataAllowedErr, 1077
 Node (*compiler.ast* のクラス), 1661
 node() (*platform* モジュール), 705
 nodelay() (*curses.window* のメソッド), 690
 nodeName (*xml.dom.Node* の属性), 1069
 NodeTransformer (*ast* のクラス), 1632
 nodeType (*xml.dom.Node* の属性), 1068
 nodeValue (*xml.dom.Node* の属性), 1070
 NodeVisitor (*ast* のクラス), 1632
 NODISC (*cd* モジュール), 1753
 noecho() (*curses* モジュール), 682
 NOEXPR (*locale* モジュール), 1301
 nofill (*htmlib.HTMLParser* の属性), 1046
 nok_builtin_names (*rexec.RExec* の属性), 1597
 noload() (*pickle.Unpickler* のメソッド), 390
 NoModificationAllowedErr, 1077
 nonblock() (*ossaudiodev.oss_audio_device* のメソッド), 1280
 None (*Built-in object*), 37
 None (組み込み変数), 35
 NoneType (*types* モジュール), 259
 nonl() (*curses* モジュール), 682
 nonzero (*2to3 fixer*), 1478
 noop() (*imaplib.IMAP4* のメソッド), 1182
 noop() (*poplib.POP3* のメソッド), 1177
 NoOptionError, 483
 NOP (*opcode*), 1648
 noqiflush() (*curses* モジュール), 682
 noraw() (*curses* モジュール), 682
 normalize() (*decimal.Context* のメソッド), 304
 normalize() (*decimal.Decimal* のメソッド), 295
 normalize() (*locale* モジュール), 1302
 normalize() (*unicodedata* モジュール), 171
 normalize() (*xml.dom.Node* のメソッド), 1070
 NORMALIZE_WHITESPACE (*doctest* モジュール), 1420
 normalvariate() (*random* モジュール), 321
 normcase() (*os.path* モジュール), 356
 normpath() (*os.path* モジュール), 356
 NoSectionError, 483
 NoSuchMailboxError, 1004
 not
 演算子, 38
 not in
 演算子, 39, 46
 not_() (*operator* モジュール), 342
 NotANumber, 174
 notationDecl() (*xml.sax.handler.DTDHandler* のメソッド), 1093
 NotationDeclHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1106
 notations (*xml.dom.DocumentType* の属性), 1072
 NotConnected, 1163

Notebook (*Tix* のクラス), 1356
 Notebook (*ttk* のクラス), 1337
 NotEmptyError, 1004
 NOTEQUAL (*token* モジュール), 1636
 NotFoundErr, 1077
 notify() (*threading.Condition* のメソッド), 773
 notify_all() (*threading.Condition* のメソッド), 773
 notifyAll() (*threading.Condition* のメソッド), 773
 notimeout() (*curses.window* のメソッド), 690
 NotImplemented (組み込み変数), 35
 NotImplementedError, 84
 NotImplementedType (*types* モジュール), 261
 NotStandaloneHandler() (*xml.parsers.expat.xmlparser* のメソッド), 1107
 NotSupportedErr, 1077
 noutrefresh() (*curses.window* のメソッド), 690
 now() (*datetime.datetime* のクラスメソッド), 185
 NProperty (*aetypes* のクラス), 1747
 NSIG (*signal* モジュール), 915
 nsmallest() (*heapq* モジュール), 230
 NT_OFFSET (*token* モジュール), 1636
 NTEventLogHandler (*logging.handlers* のクラス), 673
 ntohl() (*socket* モジュール), 874
 ntohs() (*socket* モジュール), 874
 ntransfercmd() (*ftplib.FTP* のメソッド), 1173
 NullFormatter (*formatter* のクラス), 1671
 NullHandler (*logging* のクラス), 666
 NullImporter (*imp* のクラス), 1605
 NullTranslations (*gettext* のクラス), 1289
 NullWriter (*formatter* のクラス), 1673
 Number (*numbers* のクラス), 271
 NUMBER (*token* モジュール), 1636
 number_class() (*decimal.Context* のメソッド), 304
 number_class() (*decimal.Decimal* のメソッド), 295
 numbers (モジュール), 271
 numerator (*numbers.Rational* の属性), 272
 numeric
 conversions, 40
 literals, 39
 object, 39
 types, operations on, 40
 オブジェクト, 39
 numeric() (*unicodedata* モジュール), 170
 Numerical Python, 24
 numliterals (*2to3 fixer*), 1478
 nurbscurve() (*gl* モジュール), 1765
 nurbsurface() (*gl* モジュール), 1765
 ndarray() (*gl* モジュール), 1765

 O_APPEND (*os* モジュール), 519
 O_ASYNC (*os* モジュール), 520
 O_BINARY (*os* モジュール), 520
 O_CREAT (*os* モジュール), 519
 O_DIRECT (*os* モジュール), 520
 O_DIRECTORY (*os* モジュール), 520
 O_DSYNC (*os* モジュール), 519
 O_EXCL (*os* モジュール), 519
 O_EXLOCK (*os* モジュール), 520
 O_NDELAY (*os* モジュール), 519
 O_NOATIME (*os* モジュール), 520
 O_NOCTTY (*os* モジュール), 519
 O_NOFOLLOW (*os* モジュール), 520
 O_NOINHERIT (*os* モジュール), 520
 O_NONBLOCK (*os* モジュール), 519
 O_RANDOM (*os* モジュール), 520
 O_RDONLY (*os* モジュール), 519
 O_RDWR (*os* モジュール), 519
 O_RSYNC (*os* モジュール), 519
 O_SEQUENTIAL (*os* モジュール), 520
 O_SHLOCK (*os* モジュール), 520
 O_SHORT_LIVED (*os* モジュール), 520
 O_SYNC (*os* モジュール), 519
 O_TEMPORARY (*os* モジュール), 520

- O.TEXT (*os* モジュール), 520
- O.TRUNC (*os* モジュール), 519
- O.WRONLY (*os* モジュール), 519
- object, 1787
- object
 - code, 78, 404
 - numeric, 39
- object (*exceptions.UnicodeError* の属性), 86
- object (組み込みクラス), 18
- objects
 - comparing, 39
 - flattening, 385
 - marshalling, 385
 - persistent, 385
 - pickling, 385
 - serializing, 385
- ObjectSpecifier (*aetypes* のクラス), 1747
- obufcount () (*ossaudiodev.oss_audio_device* のメソッド), 1282
- obuffree () (*ossaudiodev.oss_audio_device* のメソッド), 1282
- oct () (*future.builtins* モジュール), 1548
- oct () (組み込み関数), 18
- octal
 - literals, 39
- octdigits (*string* モジュール), 92
- offset (*xml.parsers.expat.ExpatError* の属性), 1107
- OK (*curses* モジュール), 693
- ok.builtin.modules (*rexec.RExec* の属性), 1597
- ok.file.types (*rexec.RExec* の属性), 1597
- ok.path (*rexec.RExec* の属性), 1597
- ok.posix.names (*rexec.RExec* の属性), 1597
- ok.sys.names (*rexec.RExec* の属性), 1597
- OleDLL (*ctypes* のクラス), 739
- onclick () (*turtle* モジュール), 1379, 1386
- ondrag () (*turtle* モジュール), 1380
- onecmd () (*cmd.Cmd* のメソッド), 1308
- onkey () (*turtle* モジュール), 1386
- onrelease () (*turtle* モジュール), 1380
- onscreenclick () (*turtle* モジュール), 1386
- ontimer () (*turtle* モジュール), 1387
- OP (*token* モジュール), 1636
- OP.ALL (*ssl* モジュール), 894
- OP.CIPHER.SERVER.PREFERENCE (*ssl* モジュール), 895
- OP.ENABLE.MIDDLEBOX.COMPAT (*ssl* モジュール), 895
- OP.NO.COMPRESSION (*ssl* モジュール), 895
- OP.NO.SSLv2 (*ssl* モジュール), 894
- OP.NO.SSLv3 (*ssl* モジュール), 894
- OP.NO.TLSv1 (*ssl* モジュール), 894
- OP.NO.TLSv1.1 (*ssl* モジュール), 894
- OP.NO.TLSv1.2 (*ssl* モジュール), 895
- OP.NO.TLSv1.3 (*ssl* モジュール), 895
- OP.SINGLE.DH.USE (*ssl* モジュール), 895
- OP.SINGLE.ECDH.USE (*ssl* モジュール), 895
- Open Scripting Architecture, 1748
- open () (*aifc* モジュール), 1265
- open () (*anydbm* モジュール), 406
- open () (*cd* モジュール), 1752
- open () (*codecs* モジュール), 154
- open () (*dbhash* モジュール), 411
- open () (*dbm* モジュール), 408
- open () (*dl* モジュール), 1702
- open () (*dumbdbm* モジュール), 416
- open () (*FrameWork.DialogWindow* のメソッド), 1735
- open () (*FrameWork.Window* のメソッド), 1733
- open () (*gdbm* モジュール), 409
- open () (*gzip* モジュール), 446
- open () (*imaplib.IMAP4* のメソッド), 1182
- open () (*io* モジュール), 546
- open () (*os* モジュール), 518
- open () (*ossaudiodev* モジュール), 1278
- open () (*pipes.Template* のメソッド), 1710
- open () (*posixfile* モジュール), 1711
- open () (*shelve* モジュール), 400
- open () (*sunau* モジュール), 1268
- open () (*sunaudiodev* モジュール), 1771
- open () (*tarfile* モジュール), 459
- open () (*tarfile.TarFile* のクラスメソッド), 463
- open () (*telnetlib.Telnet* のメソッド), 1199
- open () (*urllib.URLopener* のメソッド), 1143
- open () (*urllib2.OpenerDirector* のメソッド), 1153
- open () (*wave* モジュール), 1271
- open () (*webbrowser* モジュール), 1114
- open () (*webbrowser.controller* のメソッド), 1116
- open () (*zipfile.ZipFile* のメソッド), 452
- open () (組み込み関数), 18
- open.new () (*webbrowser* モジュール), 1114
- open.new () (*webbrowser.controller* のメソッド), 1116
- open.new.tab () (*webbrowser* モジュール), 1114
- open.new.tab () (*webbrowser.controller* のメソッド), 1116
- open.osfhandle () (*msvcrt* モジュール), 1683
- open.unknown () (*urllib.URLopener* のメソッド), 1143
- OpenDatabase () (*msilib* モジュール), 1675
- opendir () (*dircache* モジュール), 382
- OpenerDirector (*urllib2* のクラス), 1149
- openfolder () (*mhlib.MH* のメソッド), 1009
- openfp () (*sunau* モジュール), 1268
- openfp () (*wave* モジュール), 1271
- OpenGL, 1766
- OpenKey () (*_winreg* モジュール), 1688
- OpenKeyEx () (*_winreg* モジュール), 1688
- openlog () (*syslog* モジュール), 1718
- openmessage () (*mhlib.Message* のメソッド), 1010
- openmixer () (*ossaudiodev* モジュール), 1279
- openport () (*al* モジュール), 1749
- openpty () (*os* モジュール), 518
- openpty () (*pty* モジュール), 1706
- openrf () (*MacOS* モジュール), 1724
- OpenSSL
 - (use in module hashlib), 499
 - (use in module ssl), 883
- OPENSSL.VERSION (*ssl* モジュール), 896
- OPENSSL.VERSION.INFO (*ssl* モジュール), 896
- OPENSSL.VERSION.NUMBER (*ssl* モジュール), 896
- OpenView () (*msilib.Database* のメソッド), 1677
- operation
 - concatenation, 46
 - extended slice, 46
 - repetition, 46
 - slice, 46
 - subscript, 46
- operations
 - bitwise, 41
 - Boolean, 37, 38
 - masking, 41
 - shifting, 41
- operations on
 - dictionary type, 63
 - integer types, 41
 - list type, 58
 - mapping types, 63
 - numeric types, 40
 - sequence types, 46, 58
- operator
 - comparison, 38
- operator (モジュール), 342
- opmap (*dis* モジュール), 1647
- opname (*dis* モジュール), 1647
- optimize () (*pickletools* モジュール), 1657
- OptionGroup (*optparse* のクラス), 610
- OptionMenu (*Tix* のクラス), 1354
- OptionParser (*optparse* のクラス), 614
- options (*doctest.Example* の属性), 1432
- options (*ssl.SSLContext* の属性), 905
- options () (*ConfigParser.RawConfigParser* のメソッド), 484
- optionxform () (*ConfigParser.RawConfigParser* のメソッド), 485
- optparse (モジュール), 601

or
演算子, 38

or_() (operator モジュール), 344

ord() (組み込み関数), 19

ordered.attributes (xml.parsers.expat.xmlparser の属性), 1103

OrderedDict (collections のクラス), 223

Ordinal (aetypes のクラス), 1747

origin_server (wsgiref.handlers.BaseHandler の属性), 1136

os
モジュール, 69, 1697

os (モジュール), 507

os.path (モジュール), 353

os.environ (wsgiref.handlers.BaseHandler の属性), 1134

OSError, 84

ossaudiodev (モジュール), 1278

OSSAudioError, 1278

output (subprocess.CalledProcessError の属性), 856

output() (Cookie.BaseCookie のメソッド), 1240

output() (Cookie.Morsel のメソッド), 1241

output.charset (email.charset.Charset の属性), 953

output.charset() (gettext.NullTranslations のメソッド), 1290

output.codec (email.charset.Charset の属性), 953

output.difference() (doctest.OutputChecker のメソッド), 1436

OutputChecker (doctest のクラス), 1436

OutputString() (Cookie.Morsel のメソッド), 1241

OutputType (cStringIO モジュール), 146

Overflow (decimal のクラス), 306

OverflowError, 84

overlay() (curses.window のメソッド), 690

Overmars, Mark, 1756

overwrite() (curses.window のメソッド), 691

P_DETACH (os モジュール), 539

P_NOWAIT (os モジュール), 538

P_NOWAITO (os モジュール), 538

P_OVERLAY (os モジュール), 539

P_WAIT (os モジュール), 538

pack() (aepack モジュール), 1744

pack() (mailbox.MH のメソッド), 991

pack() (struct モジュール), 127

pack() (struct.Struct のメソッド), 132

pack_array() (xdrlib.Packer のメソッド), 493

pack_bytes() (xdrlib.Packer のメソッド), 492

pack_double() (xdrlib.Packer のメソッド), 492

pack_farray() (xdrlib.Packer のメソッド), 493

pack_float() (xdrlib.Packer のメソッド), 492

pack_fopaque() (xdrlib.Packer のメソッド), 492

pack_fstring() (xdrlib.Packer のメソッド), 492

pack_into() (struct モジュール), 127

pack_into() (struct.Struct のメソッド), 132

pack_list() (xdrlib.Packer のメソッド), 492

pack_opaque() (xdrlib.Packer のメソッド), 492

pack_string() (xdrlib.Packer のメソッド), 492

package, 1787

package, 1579

Packer (xdrlib のクラス), 491

packevent() (aetools モジュール), 1743

packing
binary data, 126

packing (widgets), 1323

PAGER, 1410, 1498

pair_content() (curses モジュール), 682

pair_number() (curses モジュール), 682

PanedWindow (Tix のクラス), 1356

parameter, 1787

pardir (os モジュール), 543

paren (2to3 fixer), 1478

parent (urllib2.BaseHandler の属性), 1154

parent() (ttk.Treeview のメソッド), 1346

parentNode (xml.dom.Node の属性), 1069

paretovariate() (random モジュール), 321

parse() (ast モジュール), 1630

parse() (cgi モジュール), 1121

parse() (compiler モジュール), 1659

parse() (doctest.DocTestParser のメソッド), 1434

parse() (email.parser.Parser のメソッド), 943

parse() (robotparser.RobotFileParser のメソッド), 489

parse() (string.Formatter のメソッド), 93

parse() (xml.dom.minidom モジュール), 1079

parse() (xml.dom.pulldom モジュール), 1085

parse() (xml.etree.ElementTree モジュール), 1057

parse() (xml.etree.ElementTree.ElementTree のメソッド), 1062

Parse() (xml.parsers.expat.xmlparser のメソッド), 1102

parse() (xml.sax モジュール), 1086

parse() (xml.sax.xmlreader.XMLReader のメソッド), 1096

parse_and_bind() (readline モジュール), 847

parse_args() (argparse.ArgumentParser のメソッド), 589

PARSE.COLNAMES (sqlite3 モジュール), 419

parse_config_h() (sysconfig モジュール), 1546

PARSE.DECLTYPES (sqlite3 モジュール), 419

parse_header() (cgi モジュール), 1121

parse_known_args() (argparse.ArgumentParser のメソッド), 599

parse_multipart() (cgi モジュール), 1121

parse_qs() (cgi モジュール), 1121

parse_qs() (urlparse モジュール), 1206

parse_qsl() (cgi モジュール), 1121

parse_qsl() (urlparse モジュール), 1206

parseaddr() (email.utils モジュール), 958

parseaddr() (rfc822 モジュール), 1022

parsedate() (email.utils モジュール), 959

parsedate() (rfc822 モジュール), 1022

parsedate.tz() (email.utils モジュール), 959

parsedate.tz() (rfc822 モジュール), 1022

parseFile() (compiler モジュール), 1659

ParseFile() (xml.parsers.expat.xmlparser のメソッド), 1102

ParseFlags() (imaplib モジュール), 1179

Parser (email.parser のクラス), 942

parser (モジュール), 1621

ParserCreate() (xml.parsers.expat モジュール), 1101

ParserError, 1624

ParseResult (urlparse のクラス), 1209

parsesequence() (mhlib.Folder のメソッド), 1009

parsestr() (email.parser.Parser のメソッド), 943

parseString() (xml.dom.minidom モジュール), 1079

parseString() (xml.dom.pulldom モジュール), 1085

parseString() (xml.sax モジュール), 1086

parseurl() (ic モジュール), 1722

parseurl() (ic.IC のメソッド), 1722

parsing
Python source code, 1621

URL, 1204

ParsingError, 483

partial() (functools モジュール), 340

partial() (imaplib.IMAP4 のメソッド), 1182

partition() (str のメソッド), 50

pass_() (poplib.POP3 のメソッド), 1176

Paste, 1400

PATH, 533, 538, 544, 1113, 1123, 1125

path
configuration file, 1579

module search, 376, 1536, 1579

operations, 353

path (BaseHTTPServer.BaseHTTPRequestHandler の属性), 1220

path (cookielib.Cookie の属性), 1236

path (sys モジュール), 1536

Path browser, 1396

path_hooks (sys モジュール), 1537

path_importer_cache (sys モジュール), 1537

path_return_ok() (cookielib.CookiePolicy のメソッド), 1233

pathconf() (os モジュール), 526

pathconf.names (os モジュール), 526

pathname2url() (urllib モジュール), 1142

pathsep (*os* モジュール), 544
 pattern (*re.RegexObject* の属性), 119
 pause () (*signal* モジュール), 916
 PAUSED (*cd* モジュール), 1753
 PAX_FORMAT (*tarfile* モジュール), 461
 pax_headers (*tarfile.TarFile* の属性), 465
 pax_headers (*tarfile.TarInfo* の属性), 466
 pbkdf2_hmac () (*hashlib* モジュール), 501
 pd () (*turtle* モジュール), 1371
 Pdb (*class in pdb*), 1495
 Pdb (*pdb* のクラス), 1497
 pdb (モジュール), 1495
 peek () (*io.BufferedReader* のメソッド), 554
 PEM_cert_to_DER_cert () (*ssl* モジュール), 890
 pen () (*turtle* モジュール), 1372
 pencolor () (*turtle* モジュール), 1373
 PendingDeprecationWarning, 87
 pendown () (*turtle* モジュール), 1371
 pensize () (*turtle* モジュール), 1371
 penup () (*turtle* モジュール), 1371
 PEP, 1788
 PERCENT (*token* モジュール), 1636
 PERCENTEQUAL (*token* モジュール), 1636
 Performance, 1513
 permutations () (*itertools* モジュール), 332
 Persist () (*msilib.SummaryInformation* のメソッド), 1678
 persistence, 385
 persistent
 objects, 385
 persistent_id (*pickle protocol*), 394
 persistent_load (*pickle protocol*), 394
 pformat () (*pprint* モジュール), 265
 pformat () (*pprint.PrettyPrinter* のメソッド), 266
 phase () (*cmath* モジュール), 281
 pi (*cmath* モジュール), 283
 pi (*math* モジュール), 279
 pick () (*gl* モジュール), 1765
 pickle
 モジュール, 263, 399, 400, 404
 pickle (モジュール), 385
 pickle () (*copy_reg* モジュール), 399
 PickleError, 388
 Pickler (*pickle* のクラス), 389
 pickletools (モジュール), 1657
 pickling
 objects, 385
 PicklingError, 388
 pid (*multiprocessing.Process* の属性), 787
 pid (*popen2.Popen3* の属性), 920
 pid (*subprocess.Popen* の属性), 862
 PIL (*the Python Imaging Library*), 1768
 PIPE (*subprocess* モジュール), 856
 Pipe () (*multiprocessing* モジュール), 790
 pipe () (*os* モジュール), 518
 PIPE_BUF (*select.select* の属性), 758
 pipes (モジュール), 1708
 PixMapWrapper (モジュール), 1777
 PKG_DIRECTORY (*imp* モジュール), 1603
 pkgutil (モジュール), 1612
 platform (*sys* モジュール), 1537
 platform (モジュール), 704
 platform () (*platform* モジュール), 705
 PLAYING (*cd* モジュール), 1753
 PlaySound () (*winsound* モジュール), 1694
 plist
 file, 495
 plistlib (モジュール), 495
 plock () (*os* モジュール), 537
 PLUS (*token* モジュール), 1636
 plus () (*decimal.Context* のメソッド), 304
 PLUSEQUAL (*token* モジュール), 1636
 pm () (*pdb* モジュール), 1497
 pnum (*cd* モジュール), 1753
 POINTER () (*ctypes* モジュール), 748
 pointer () (*ctypes* モジュール), 748
 polar () (*cmath* モジュール), 281
 poll () (*Connection* のメソッド), 794
 poll () (*popen2.Popen3* のメソッド), 919
 poll () (*select* モジュール), 757
 poll () (*select.epoll* のメソッド), 760
 poll () (*select.poll* のメソッド), 761
 poll () (*subprocess.Popen* のメソッド), 861
 pop () (*array.array* のメソッド), 238
 pop () (*asyncio.fifo* のメソッド), 928
 pop () (*collections.deque* のメソッド), 214
 pop () (*dict* のメソッド), 66
 pop () (*frozenset* のメソッド), 63
 pop () (*list method*), 58
 pop () (*mailbox.Mailbox* のメソッド), 987
 pop () (*multifile.MultiFile* のメソッド), 1019
 pop_alignment () (*formatter.formatter* のメソッド), 1671
 POP_BLOCK (*opcode*), 1652
 pop_font () (*formatter.formatter* のメソッド), 1671
 POP_JUMP_IF_FALSE (*opcode*), 1654
 POP_JUMP_IF_TRUE (*opcode*), 1654
 pop_margin () (*formatter.formatter* のメソッド), 1671
 pop_source () (*shlex.shlex* のメソッド), 1312
 pop_style () (*formatter.formatter* のメソッド), 1671
 POP_TOP (*opcode*), 1648
 POP3
 protocol, 1175
 POP3 (*poplib* のクラス), 1175
 POP3_SSL (*poplib* のクラス), 1175
 Popen (*subprocess* のクラス), 857
 popen () (*in module os*), 758
 popen () (*os* モジュール), 513
 popen () (*platform* モジュール), 707
 popen2 (モジュール), 918
 popen2 () (*os* モジュール), 514
 popen2 () (*popen2* モジュール), 919
 Popen3 (*popen2* のクラス), 919
 popen3 () (*os* モジュール), 514
 popen3 () (*popen2* モジュール), 919
 Popen4 (*popen2* のクラス), 919
 popen4 () (*os* モジュール), 515
 popen4 () (*popen2* モジュール), 919
 popitem () (*collections.OrderedDict* のメソッド), 223
 popitem () (*dict* のメソッド), 66
 popitem () (*mailbox.Mailbox* のメソッド), 987
 popleft () (*collections.deque* のメソッド), 214
 poplib (モジュール), 1175
 PopupMenu (*Tix* のクラス), 1354
 port (*cookielib.Cookie* の属性), 1236
 port_specified (*cookielib.Cookie* の属性), 1237
 PortableUnixMailbox (*mailbox* のクラス), 1005
 pos (*re.MatchObject* の属性), 121
 pos () (*operator* モジュール), 344
 pos () (*turtle* モジュール), 1369
 position () (*turtle* モジュール), 1369
 POSIX
 file object, 1711
 I/O control, 1704
 threads, 777
 posix (*tarfile.TarFile* の属性), 465
 posix (モジュール), 1697
 posixfile (モジュール), 1711
 POSIXLY_CORRECT, 635
 post () (*nntplib.NNTP* のメソッド), 1189
 post () (*ossaudiodev.oss_audio_device* のメソッド), 1281
 post_mortem () (*pdb* モジュール), 1496
 postcmd () (*cmd.Cmd* のメソッド), 1309
 postloop () (*cmd.Cmd* のメソッド), 1309
 pow () (*math* モジュール), 277
 pow () (*operator* モジュール), 344
 pow () (組み込み関数), 19
 power () (*decimal.Context* のメソッド), 304

pprint (モジュール), 264
 pprint () (*bdb.Breakpoint* のメソッド), 1490
 pprint () (*pprint* モジュール), 265
 pprint () (*pprint.PrettyPrinter* のメソッド), 266
 prcal () (*calendar* モジュール), 209
 preamble (*email.message.Message* の属性), 940
 precmd () (*cmd.Cmd* のメソッド), 1309
 prefix (*sys* モジュール), 1538
 prefix (*xml.dom.Attr* の属性), 1074
 prefix (*xml.dom.Node* の属性), 1069
 prefix (*zipimport.zipimporter* の属性), 1612
 PREFIXES (*site* モジュール), 1580
 preloop () (*cmd.Cmd* のメソッド), 1309
 preorder () (*compiler.visitor.ASTVisitor* のメソッド), 1667
 prepare_input_source () (*xml.sax.saxutils* モジュール), 1095
 prepend () (*pipes.Template* のメソッド), 1710
 PrettyPrinter (*pprint* のクラス), 264
 prev () (*tk.Treeview* のメソッド), 1347
 previous () (*bsddb.bsddbobject* のメソッド), 414
 previous () (*dbhash.dbhash* のメソッド), 412
 previousSibling (*xml.dom.Node* の属性), 1069
 print
 文, 37
 print (*2to3 fixer*), 1478
 Print () (*findertools* モジュール), 1726
 print () (組み込み関数), 19
 print_callees () (*pstats.Stats* のメソッド), 1508
 print_callers () (*pstats.Stats* のメソッド), 1508
 print_directory () (*cgi* モジュール), 1122
 print_environ () (*cgi* モジュール), 1122
 print_environ_usage () (*cgi* モジュール), 1122
 print_exc () (*timeit.Timer* のメソッド), 1515
 print_exc () (*traceback* モジュール), 1562
 print_exception () (*traceback* モジュール), 1562
 PRINT_EXPR (*opcode*), 1651
 print_form () (*cgi* モジュール), 1122
 print_help () (*argparse.ArgumentParser* のメソッド), 599
 PRINT_ITEM (*opcode*), 1651
 PRINT_ITEM.TO (*opcode*), 1651
 print_last () (*traceback* モジュール), 1563
 PRINT_NEWLINE (*opcode*), 1651
 PRINT_NEWLINE.TO (*opcode*), 1651
 print_stack () (*traceback* モジュール), 1563
 print_stats () (*profile.Profile* のメソッド), 1505
 print_stats () (*pstats.Stats* のメソッド), 1508
 print_tb () (*traceback* モジュール), 1562
 print_usage () (*argparse.ArgumentParser* のメソッド), 599
 print_usage () (*optparse.OptionParser* のメソッド), 626
 print_version () (*optparse.OptionParser* のメソッド), 612
 printable (*string* モジュール), 92
 printdir () (*zipfile.ZipFile* のメソッド), 454
 printf-style formatting, 55
 PriorityQueue (*Queue* のクラス), 248
 prmonth () (*calendar* モジュール), 209
 prmonth () (*calendar.TextCalendar* のメソッド), 207
 process
 group, 509, 510
 id, 510
 id of parent, 510
 killing, 537
 signalling, 537
 Process (*multiprocessing* のクラス), 786
 process () (*logging.LoggerAdapter* のメソッド), 648
 process_message () (*smtpd.SMTPServer* のメソッド), 1196
 process_request () (*SocketServer.BaseServer* のメソッド), 1214
 process_tokens () (*tabnanny* モジュール), 1641
 ProcessError, 813
 processes, light-weight, 777
 processfile () (*gensuitemodule* モジュール), 1742
 processfile_fromresource () (*gensuitemodule* モジュール), 1743
 ProcessingInstruction () (*xml.etree.ElementTree* モジュール), 1057
 processingInstruction ()
 (*xml.sax.handler.ContentHandler* のメソッド), 1092
 ProcessingInstructionHandler ()
 (*xml.parsers.expat.xmlparser* のメソッド), 1105
 processor time, 561
 processor () (*platform* モジュール), 705
 product () (*itertools* モジュール), 333
 Profile (*hotshot* のクラス), 1512
 Profile (*profile* のクラス), 1505
 profile (モジュール), 1504
 profile function, 766, 1534, 1539
 profiler, 1534, 1539
 profiling, deterministic, 1501
 Progressbar (*tk* のクラス), 1339
 ProgressBar () (*EasyDialogs* モジュール), 1727
 prompt (*cmd.Cmd* の属性), 1309
 prompt_user_passwd () (*urllib.FancyURLopener* のメソッド), 1144
 prompts, interpreter, 1538
 propagate (*logging.Logger* の属性), 638
 property (組み込みクラス), 20
 property list, 495
 property_declaration_handler (*xml.sax.handler* モジュール), 1089
 property_dom_node (*xml.sax.handler* モジュール), 1090
 property_lexical_handler (*xml.sax.handler* モジュール), 1089
 property_xml_string (*xml.sax.handler* モジュール), 1090
 prot_c () (*ftplib.FTP.TLS* のメソッド), 1175
 prot_p () (*ftplib.FTP.TLS* のメソッド), 1175
 proto (*socket.socket* の属性), 880
 protocol
 CGI, 1116
 context management, 76
 FTP, 1144, 1169
 HTTP, 1116, 1144, 1145, 1161, 1219
 IMAP4, 1178
 IMAP4.SSL, 1178
 IMAP4.stream, 1178
 iterator, 43
 NNTP, 1185
 POP3, 1175
 SMTP, 1190
 Telnet, 1197
 protocol (*ssl.SSLContext* の属性), 905
 PROTOCOL.SSLv2 (*ssl* モジュール), 893
 PROTOCOL.SSLv23 (*ssl* モジュール), 893
 PROTOCOL.SSLv3 (*ssl* モジュール), 893
 PROTOCOL.TLS (*ssl* モジュール), 893
 PROTOCOL.TLSv1 (*ssl* モジュール), 893
 PROTOCOL.TLSv1.1 (*ssl* モジュール), 894
 PROTOCOL.TLSv1.2 (*ssl* モジュール), 894
 protocol_version
 (*BaseHTTPServer.BaseHTTPRequestHandler* の属性), 1221
 PROTOCOL.VERSION (*imaplib.IMAP4* の属性), 1185
 ProtocolError (*xmlrpclib* のクラス), 1249
 proxy () (*weakref* モジュール), 251
 proxyauth () (*imaplib.IMAP4* のメソッド), 1182
 ProxyBasicAuthHandler (*urllib2* のクラス), 1150
 ProxyDigestAuthHandler (*urllib2* のクラス), 1150
 ProxyHandler (*urllib2* のクラス), 1149
 ProxyType (*weakref* モジュール), 253
 ProxyTypes (*weakref* モジュール), 253
 prstr () (*fm* モジュール), 1763
 pryyear () (*calendar.TextCalendar* のメソッド), 207
 ps1 (*sys* モジュール), 1538
 ps2 (*sys* モジュール), 1538
 pstats (モジュール), 1506
 pthreads, 777
 ptime (*cd* モジュール), 1753

- pty
 - モジュール, 518
- pty (モジュール), 1705
- pu () (*turtle* モジュール), 1371
- publicId (*xml.dom.DocumentType* の属性), 1071
- PullDOM (*xml.dom.pulldom* のクラス), 1085
- punctuation (*string* モジュール), 92
- PureProxy (*smtpd* のクラス), 1197
- purge () (*re* モジュール), 117
- Purpose.CLIENT_AUTH (*ssl* モジュール), 897
- Purpose.SERVER_AUTH (*ssl* モジュール), 897
- push () (*asynchat.async_chat* のメソッド), 927
- push () (*asynchat.fifo* のメソッド), 928
- push () (*code.InteractiveConsole* のメソッド), 1589
- push () (*multifile.MultiFile* のメソッド), 1019
- push_alignment () (*formatter.formatter* のメソッド), 1671
- push_font () (*formatter.formatter* のメソッド), 1671
- push_margin () (*formatter.formatter* のメソッド), 1671
- push_source () (*shlex.shlex* のメソッド), 1312
- push_style () (*formatter.formatter* のメソッド), 1671
- push_token () (*shlex.shlex* のメソッド), 1311
- push_with_producer () (*asynchat.async_chat* のメソッド), 927
- pushbutton () (*msilib.Dialog* のメソッド), 1681
- put () (*multiprocessing.multiprocessing.queues.SimpleQueue* のメソッド), 792
- put () (*multiprocessing.Queue* のメソッド), 790
- put () (*Queue.Queue* のメソッド), 248
- put_nowait () (*multiprocessing.Queue* のメソッド), 791
- put_nowait () (*Queue.Queue* のメソッド), 249
- putch () (*msvcrt* モジュール), 1683
- putenv () (*os* モジュール), 511
- putheader () (*httplib.HTTPConnection* のメソッド), 1167
- putp () (*curses* モジュール), 682
- putrequest () (*httplib.HTTPConnection* のメソッド), 1167
- putsequences () (*mhlib.Folder* のメソッド), 1009
- putwch () (*msvcrt* モジュール), 1684
- putwin () (*curses.window* のメソッド), 691
- pwd
 - モジュール, 354
- pwd (モジュール), 1698
- pwd () (*ftplib.FTP* のメソッド), 1174
- pwlcurve () (*gl* モジュール), 1765
- py_compile (モジュール), 1643
- PY_COMPILED (*imp* モジュール), 1603
- PY_FROZEN (*imp* モジュール), 1603
- py_object (*ctypes* のクラス), 754
- PY_SOURCE (*imp* モジュール), 1603
- py_suffix_importer () (*imputil* モジュール), 1607
- py3kwarning (*sys* モジュール), 1538
- pyclbr (モジュール), 1641
- PyCompileError, 1643
- PyDLL (*ctypes* のクラス), 740
- pydoc (モジュール), 1409
- pyexpat
 - モジュール, 1100
- PYFUNCTIONTYPE () (*ctypes* モジュール), 743
- PyOpenGL, 1766
- Python 3000, 1788
- Python Editor, 1396
- Python Enhancement Proposals
 - PEP 205, 253
 - PEP 227, 1568
 - PEP 238, 1568
 - PEP 249, 418
 - PEP 255, 1568
 - PEP 273, 1610
 - PEP 282, 652
 - PEP 302, 1611
 - PEP 305, 472
 - PEP 3105, 1568
 - PEP 3112, 1568
 - PEP 328, 1568
 - PEP 338, 1619
 - PEP 343, 1557, 1568
 - PEP 366, 1620
 - PEP 453, 1524
 - PEP 477, 1524
- Python Enhancement Proposals
 - PEP 1, 1788
 - PEP 236, 8
 - PEP 237, 57
 - PEP 238, 1782
 - PEP 249, 417
 - PEP 278, 1789
 - PEP 282, 381
 - PEP 292, 101
 - PEP 302, 30, 376, 1536, 1537, 1605, 1613–1615, 1618, 1619, 1782, 1786
 - PEP 307, 387
 - PEP 3101, 92, 93
 - PEP 3116, 1789
 - PEP 3119, 228, 1557
 - PEP 3141, 271, 1557
 - PEP 324, 853
 - PEP 333, 1126, 1127, 1129–1132, 1135, 1136
 - PEP 343, 1781
 - PEP 370, 1582
 - PEP 378, 97
 - PEP 476, 889
 - PEP 493, 889
 - PEP 8, 764, 969
- Python Imaging Library, 1768
- python_branch () (*platform* モジュール), 706
- python_build () (*platform* モジュール), 705
- python_compiler () (*platform* モジュール), 706
- PYTHON_DOM, 1066
- python_implementation () (*platform* モジュール), 706
- python_revision () (*platform* モジュール), 706
- python_version () (*platform* モジュール), 706
- python_version_tuple () (*platform* モジュール), 706
- PYTHONDOCS, 1410
- PYTHONDONTWRITEBYTECODE, 1528
- PYTHONHASHSEED, 319
- Pythonic, 1788
- PYTHONNOUSERSITE, 1580, 1581
- PYTHONPATH, 1123, 1536, 1537
- PYTHONSTARTUP, 850, 851, 1404, 1582
- PYTHONUSERBASE, 1581
- PYTHONY2K, 560, 561
- PyZipFile (*zipfile* のクラス), 451
- qdevice () (*fl* モジュール), 1758
- QDPoint (*aetypes* のクラス), 1747
- QDRectangle (*aetypes* のクラス), 1747
- qenter () (*fl* モジュール), 1758
- qiflush () (*curses* モジュール), 682
- QName (*xml.etree.ElementTree* のクラス), 1063
- qread () (*fl* モジュール), 1758
- qreset () (*fl* モジュール), 1758
- qsize () (*multiprocessing.Queue* のメソッド), 790
- qsize () (*Queue.Queue* のメソッド), 248
- qtest () (*fl* モジュール), 1758
- quantize () (*decimal.Context* のメソッド), 305
- quantize () (*decimal.Decimal* のメソッド), 296
- QueryInfoKey () (*_winreg* モジュール), 1688
- queryparams () (*al* モジュール), 1749
- QueryReflectionKey () (*_winreg* モジュール), 1690
- QueryValue () (*_winreg* モジュール), 1688
- QueryValueEx () (*_winreg* モジュール), 1689
- Queue (*multiprocessing* のクラス), 790
- Queue (*Queue* のクラス), 247
- queue (*sched.scheduler* の属性), 246
- Queue (モジュール), 247
- Queue () (*multiprocessing.managers.SyncManager* のメソッド), 804

`quick_ratio()` (*diffib.SequenceMatcher* のメソッド), 140
`quit` (組み込み変数), 36
`quit()` (*ftplib.FTP* のメソッド), 1174
`quit()` (*nnplib.NNTP* のメソッド), 1190
`quit()` (*poplib.POP3* のメソッド), 1177
`quit()` (*smtplib.SMTP* のメソッド), 1195
`quopri` (モジュール), 1031
`quote()` (*email.utils* モジュール), 958
`quote()` (*pipes* モジュール), 1709
`quote()` (*rfc822* モジュール), 1021
`quote()` (*urllib* モジュール), 1141
`QUOTE_ALL` (*csv* モジュール), 475
`QUOTE_MINIMAL` (*csv* モジュール), 475
`QUOTE_NONE` (*csv* モジュール), 475
`QUOTE_NONNUMERIC` (*csv* モジュール), 475
`quote_plus()` (*urllib* モジュール), 1141
`quoteattr()` (*xml.sax.saxutils* モジュール), 1094
`quotechar` (*csv.Dialect* の属性), 476
`quoted-printable`
 encoding, 1031
`quotes` (*shlex.shlex* の属性), 1313
`quoting` (*csv.Dialect* の属性), 476

`r.eval()` (*rexec.RExec* のメソッド), 1595
`r.exec()` (*rexec.RExec* のメソッド), 1595
`r.execfile()` (*rexec.RExec* のメソッド), 1595
`r.import()` (*rexec.RExec* のメソッド), 1596
`R_OK` (*os* モジュール), 521
`r.open()` (*rexec.RExec* のメソッド), 1596
`r.reload()` (*rexec.RExec* のメソッド), 1596
`r.unload()` (*rexec.RExec* のメソッド), 1596
`radians()` (*math* モジュール), 278
`radians()` (*turtle* モジュール), 1371
`RadioButtonGroup` (*msilib* のクラス), 1681
`radiogroup()` (*msilib.Dialog* のメソッド), 1681
`radix()` (*decimal.Context* のメソッド), 305
`radix()` (*decimal.Decimal* のメソッド), 296
`RADIXCHAR` (*locale* モジュール), 1301
`raise`
 文, 81
`raise` (*2to3 fixer*), 1478
`RAISE_VARARGS` (*opcode*), 1655
`RAND_add()` (*ssl* モジュール), 889
`RAND_egd()` (*ssl* モジュール), 889
`RAND_status()` (*ssl* モジュール), 889
`randint()` (*random* モジュール), 320
`random` (モジュール), 317
`random()` (*random* モジュール), 320
`randrange()` (*random* モジュール), 319
`Range` (*aetypes* のクラス), 1747
`range()` (組み込み関数), 21
`ratecv()` (*audioop* モジュール), 1261
`ratio()` (*diffib.SequenceMatcher* のメソッド), 140
`Rational` (*numbers* のクラス), 272
`raw` (*io.BufferedIOBase* の属性), 551
`raw()` (*curses* モジュール), 682
`raw_decode()` (*json.JSONDecoder* のメソッド), 978
`raw_input`
 組み込み関数, 1541
`raw_input` (*2to3 fixer*), 1478
`raw_input()` (*code.InteractiveConsole* のメソッド), 1589
`raw_input()` (組み込み関数), 22
`RawArray` (*multiprocessing.sharedctypes* モジュール), 799
`RawConfigParser` (*ConfigParser* のクラス), 482
`RawDescriptionHelpFormatter` (*argparse* のクラス), 573
`RawIOBase` (*io* のクラス), 550
`RawPen` (*turtle* のクラス), 1390
`RawTextHelpFormatter` (*argparse* のクラス), 573
`RawTurtle` (*turtle* のクラス), 1390
`RawValue` (*multiprocessing.sharedctypes* モジュール), 800
`RBRACE` (*token* モジュール), 1636
`re`
 モジュール, 58, 91, 374

`re` (*re.MatchObject* の属性), 122
`re` (モジュール), 106
`read()` (*array.array* のメソッド), 238
`read()` (*bz2.BZ2File* のメソッド), 448
`read()` (*chunk.Chunk* のメソッド), 1275
`read()` (*codecs.StreamReader* のメソッド), 160
`read()` (*ConfigParser.RawConfigParser* のメソッド), 484
`read()` (*file* のメソッド), 71
`read()` (*httplib.HTTPResponse* のメソッド), 1167
`read()` (*imaplib.IMAP4* のメソッド), 1182
`read()` (*imgfile* モジュール), 1767
`read()` (*io.BufferedIOBase* のメソッド), 552
`read()` (*io.BufferedReader* のメソッド), 554
`read()` (*io.RawIOBase* のメソッド), 550
`read()` (*io.TextIOBase* のメソッド), 556
`read()` (*mimetypes.MimeTypes* のメソッド), 1015
`read()` (*mmap.mmap* のメソッド), 845
`read()` (*multifile.MultiFile* のメソッド), 1019
`read()` (*os* モジュール), 518
`read()` (*ossaudiodev.oss.audio_device* のメソッド), 1279
`read()` (*robotparser.RobotFileParser* のメソッド), 489
`read()` (*zipfile.ZipFile* のメソッド), 454
`read_all()` (*telnetlib.Telnet* のメソッド), 1198
`read_byte()` (*mmap.mmap* のメソッド), 845
`read_eager()` (*telnetlib.Telnet* のメソッド), 1198
`read_history_file()` (*readline* モジュール), 847
`read_init_file()` (*readline* モジュール), 847
`read_lazy()` (*telnetlib.Telnet* のメソッド), 1198
`read_mime_types()` (*mimetypes* モジュール), 1013
`read_sb_data()` (*telnetlib.Telnet* のメソッド), 1199
`read_some()` (*telnetlib.Telnet* のメソッド), 1198
`read_token()` (*shlex.shlex* のメソッド), 1311
`read_until()` (*telnetlib.Telnet* のメソッド), 1198
`read_very_eager()` (*telnetlib.Telnet* のメソッド), 1198
`read_very_lazy()` (*telnetlib.Telnet* のメソッド), 1198
`read_windows_registry()` (*mimetypes.MimeTypes* のメソッド), 1015
`read1()` (*io.BufferedIOBase* のメソッド), 552
`read1()` (*io.BufferedReader* のメソッド), 554
`read1()` (*io.BytesIO* のメソッド), 553
`READABLE` (*Tkinter* モジュール), 1329
`readable()` (*asyncore.dispatcher* のメソッド), 923
`readable()` (*io.IOBase* のメソッド), 549
`readall()` (*io.RawIOBase* のメソッド), 551
`reader()` (*csv* モジュール), 472
`ReadError`, 460
`readfp()` (*ConfigParser.RawConfigParser* のメソッド), 485
`readfp()` (*mimetypes.MimeTypes* のメソッド), 1015
`readframes()` (*aifc.aifc* のメソッド), 1266
`readframes()` (*sunau.AU_read* のメソッド), 1269
`readframes()` (*wave.Wave_read* のメソッド), 1272
`readinto()` (*io.BufferedIOBase* のメソッド), 552
`readinto()` (*io.RawIOBase* のメソッド), 551
`readline` (モジュール), 846
`readline()` (*bz2.BZ2File* のメソッド), 448
`readline()` (*codecs.StreamReader* のメソッド), 160
`readline()` (*file* のメソッド), 71
`readline()` (*imaplib.IMAP4* のメソッド), 1182
`readline()` (*io.IOBase* のメソッド), 549
`readline()` (*io.TextIOBase* のメソッド), 556
`readline()` (*mmap.mmap* のメソッド), 845
`readline()` (*multifile.MultiFile* のメソッド), 1018
`readlines()` (*bz2.BZ2File* のメソッド), 448
`readlines()` (*codecs.StreamReader* のメソッド), 160
`readlines()` (*file* のメソッド), 71
`readlines()` (*io.IOBase* のメソッド), 549
`readlines()` (*multifile.MultiFile* のメソッド), 1019
`readlink()` (*os* モジュール), 526
`readmodule()` (*pyclbr* モジュール), 1641
`readmodule_ex()` (*pyclbr* モジュール), 1642
`readonly` (*memoryview* の属性), 75
`readPlist()` (*plistlib* モジュール), 495
`readPlistFromResource()` (*plistlib* モジュール), 496

readPlistFromString() (*plistlib* モジュール), 496
 readscaled() (*imgfile* モジュール), 1767
 READY (*cd* モジュール), 1753
 ready() (*multiprocessing.pool.AsyncResult* のメソッド), 811
 Real (*numbers* のクラス), 271
 real (*numbers.Complex* の属性), 271
 Real Media File Format, 1274
 real_quick_ratio() (*difflib.SequenceMatcher* のメソッド), 140
 realpath() (*os.path* モジュール), 356
 reason (*exceptions.UnicodeError* の属性), 86
 reason (*httplib.HTTPResponse* の属性), 1168
 reason (*ssl.SSLError* の属性), 885
 reason (*urllib2.HTTPError* の属性), 1148
 reason (*urllib2.URLError* の属性), 1148
 reattach() (*ttk.Treeview* のメソッド), 1347
 reccontrols() (*ossaudiodev.oss_mixer_device* のメソッド), 1283
 recent() (*imaplib.IMAP4* のメソッド), 1182
 rect() (*cmath* モジュール), 281
 rectangle() (*curses.textpad* モジュール), 698
 recv() (*asyncore.dispatcher* のメソッド), 923
 recv() (*Connection* のメソッド), 794
 recv() (*socket.socket* のメソッド), 878
 recv_bytes() (*Connection* のメソッド), 794
 recv_bytes_into() (*Connection* のメソッド), 794
 recv_into() (*socket.socket* のメソッド), 878
 recvfrom() (*socket.socket* のメソッド), 878
 recvfrom_into() (*socket.socket* のメソッド), 878
 redirect_request() (*urllib2.HTTPRedirectHandler* のメソッド), 1156
 redisplay() (*readline* モジュール), 847
 redraw_form() (*fl.form* のメソッド), 1758
 redrawln() (*curses.window* のメソッド), 691
 redrawwin() (*curses.window* のメソッド), 691
 reduce (2to3 fixer), 1478
 reduce() (*functools* モジュール), 340
 reduce() (組み込み関数), 22
 ref (*weakref* のクラス), 251
 reference count, 1788
 ReferenceError, 84, 253
 ReferenceType (*weakref* モジュール), 253
 refilemessages() (*mhlib.Folder* のメソッド), 1010
 refresh() (*curses.window* のメソッド), 691
 REG_BINARY (*._winreg* モジュール), 1692
 REG_DWORD (*._winreg* モジュール), 1692
 REG_DWORD_BIG_ENDIAN (*._winreg* モジュール), 1692
 REG_DWORD_LITTLE_ENDIAN (*._winreg* モジュール), 1692
 REG_EXPAND_SZ (*._winreg* モジュール), 1692
 REG_FULL_RESOURCE_DESCRIPTOR (*._winreg* モジュール), 1693
 REG_LINK (*._winreg* モジュール), 1693
 REG_MULTI_SZ (*._winreg* モジュール), 1693
 REG_NONE (*._winreg* モジュール), 1693
 REG_RESOURCE_LIST (*._winreg* モジュール), 1693
 REG_RESOURCE_REQUIREMENTS_LIST (*._winreg* モジュール), 1693
 REG_SZ (*._winreg* モジュール), 1693
 RegexObject (*re* のクラス), 117
 register() (*abc.ABCMeta* のメソッド), 1558
 register() (*atexit* モジュール), 1561
 register() (*codecs* モジュール), 151
 register() (*multiprocessing.managers.BaseManager* のメソッド), 803
 register() (*select.epoll* のメソッド), 759
 register() (*select.poll* のメソッド), 760
 register() (*webbrowser* モジュール), 1114
 register_adapter() (*sqlite3* モジュール), 420
 register_archive_format() (*shutil* モジュール), 381
 register_converter() (*sqlite3* モジュール), 420
 register_dialect() (*csv* モジュール), 473
 register_error() (*codecs* モジュール), 153

register_function() (*SimpleXMLRPCServer.CGIXMLRPCRequestHandler* のメソッド), 1256
 register_function() (*SimpleXMLRPCServer.SimpleXMLRPCServer* のメソッド), 1253
 register_instance() (*SimpleXMLRPCServer.CGIXMLRPCRequestHandler* のメソッド), 1256
 register_instance() (*SimpleXMLRPCServer.SimpleXMLRPCServer* のメソッド), 1254
 register_introspection_functions() (*SimpleXMLRPCServer.CGIXMLRPCRequestHandler* のメソッド), 1257
 register_introspection_functions() (*SimpleXMLRPCServer.SimpleXMLRPCServer* のメソッド), 1254
 register_multicall_functions() (*SimpleXMLRPCServer.CGIXMLRPCRequestHandler* のメソッド), 1257
 register_multicall_functions() (*SimpleXMLRPCServer.SimpleXMLRPCServer* のメソッド), 1254
 register_namespace() (*xml.etree.ElementTree* モジュール), 1057
 register_optionflag() (*doctest* モジュール), 1422
 register_shape() (*turtle* モジュール), 1388
 registerDOMImplementation() (*xml.dom* モジュール), 1066
 registerResult() (*unittest* モジュール), 1472
 relative
 URL, 1204
 release() (*logging.Handler* のメソッド), 642
 release() (*multiprocessing.Lock* のメソッド), 796
 release() (*multiprocessing.RLock* のメソッド), 797
 release() (*platform* モジュール), 706
 release() (*thread.lock* のメソッド), 778
 release() (*threading.Condition* のメソッド), 772
 release() (*threading.Lock* のメソッド), 770
 release() (*threading.RLock* のメソッド), 771
 release() (*threading.Semaphore* のメソッド), 774
 release_lock() (*imp* モジュール), 1603
 reload
 組み込み関数, 1536, 1602, 1605
 reload() (組み込み関数), 22
 relpath() (*os.path* モジュール), 356
 remainder() (*decimal.Context* のメソッド), 305
 remainder_near() (*decimal.Context* のメソッド), 305
 remainder_near() (*decimal.Decimal* のメソッド), 296
 remove() (*array.array* のメソッド), 238
 remove() (*collections.deque* のメソッド), 214
 remove() (*frozenset* のメソッド), 63
 remove() (*list method*), 58
 remove() (*mailbox.Mailbox* のメソッド), 985
 remove() (*mailbox.MH* のメソッド), 992
 remove() (*os* モジュール), 526
 remove() (*xml.etree.ElementTree.Element* のメソッド), 1061
 remove_flag() (*mailbox.MaildirMessage* のメソッド), 996
 remove_flag() (*mailbox.mboxMessage* のメソッド), 998
 remove_flag() (*mailbox.MMDFMessage* のメソッド), 1003
 remove_folder() (*mailbox.Maildir* のメソッド), 989
 remove_folder() (*mailbox.MH* のメソッド), 991
 remove_history_item() (*readline* モジュール), 848
 remove_label() (*mailbox.BabylMessage* のメソッド), 1001
 remove_option() (*ConfigParser.RawConfigParser* のメソッド), 485
 remove_option() (*optparse.OptionParser* のメソッド), 624
 remove_pyc() (*msilib.Directory* のメソッド), 1680
 remove_section() (*ConfigParser.RawConfigParser* のメソッド), 485
 remove_sequence() (*mailbox.MHMessage* のメソッド), 999
 removeAttribute() (*xml.dom.Element* のメソッド), 1074

`removeAttributeNode()` (*xml.dom.Element* のメソッド), 1074
`removeAttributeNS()` (*xml.dom.Element* のメソッド), 1074
`removeChild()` (*xml.dom.Node* のメソッド), 1070
`removedirs()` (*os* モジュール), 526
`removeFilter()` (*logging.Handler* のメソッド), 642
`removeFilter()` (*logging.Logger* のメソッド), 641
`removeHandler()` (*logging.Logger* のメソッド), 641
`removeHandler()` (*unittest* モジュール), 1472
`removemessages()` (*mhlib.Folder* のメソッド), 1009
`removeResult()` (*unittest* モジュール), 1472
`rename()` (*ftplib.FTP* のメソッド), 1174
`rename()` (*imaplib.IMAP4* のメソッド), 1182
`rename()` (*os* モジュール), 527
`renames(2to3 fixer)`, 1478
`renames()` (*os* モジュール), 527
`reorganize()` (*gdbm* モジュール), 410
`repeat()` (*itertools* モジュール), 333
`repeat()` (*operator* モジュール), 345
`repeat()` (*timeit* モジュール), 1514
`repeat()` (*timeit.Timer* のメソッド), 1515
`repetition`
 operation, 46
`replace()` (*curses.panel.Panel* のメソッド), 704
`replace()` (*datetime.date* のメソッド), 182
`replace()` (*datetime.datetime* のメソッド), 188
`replace()` (*datetime.time* のメソッド), 194
`replace()` (*str* のメソッド), 51
`replace()` (*string* モジュール), 106
`replace.errors()` (*codecs* モジュール), 153
`replace.header()` (*email.message.Message* のメソッド), 936
`replace.history.item()` (*readline* モジュール), 848
`replace.whitespace` (*textwrap.TextWrapper* の属性), 148
`replaceChild()` (*xml.dom.Node* のメソッド), 1070
`ReplacePackage()` (*modulefinder* モジュール), 1616
`report()` (*filecmp.dircmp* のメソッド), 368
`report()` (*modulefinder.ModuleFinder* のメソッド), 1616
`REPORT_CDIF` (*doctest* モジュール), 1421
`report.failure()` (*doctest.DocTestRunner* のメソッド), 1435
`report.full_closure()` (*filecmp.dircmp* のメソッド), 368
`REPORT_NDIFF` (*doctest* モジュール), 1421
`REPORT_ONLY_FIRST_FAILURE` (*doctest* モジュール), 1421
`report_partial_closure()` (*filecmp.dircmp* のメソッド), 368
`report.start()` (*doctest.DocTestRunner* のメソッド), 1434
`report.success()` (*doctest.DocTestRunner* のメソッド), 1435
`REPORT_UDIFF` (*doctest* モジュール), 1421
`report_unbalanced()` (*sgmlib.SGMLParser* のメソッド), 1044
`report.unexpected.exception()` (*doctest.DocTestRunner* のメソッド), 1435
`REPORTING_FLAGS` (*doctest* モジュール), 1422
`repr(2to3 fixer)`, 1478
`Repr` (*repr* のクラス), 268
`repr()` (*repr* モジュール), 268
`repr()` (*repr.Repr* のメソッド), 269
`repr()` (組み込み関数), 23
`repr1()` (*repr.Repr* のメソッド), 269
`Request(urllib2 のクラス)`, 1148
`request()` (*httplib.HTTPConnection* のメソッド), 1166
`request.queue_size` (*SocketServer.BaseServer* の属性), 1213
`request.uri()` (*wsgiref.util* モジュール), 1127
`request.version`
 (*BaseHTTPServer.BaseHTTPRequestHandler* の属性), 1220
`RequestHandlerClass` (*SocketServer.BaseServer* の属性), 1213
`requires()` (*test.support* モジュール), 1483
`reserved` (*zipfile.ZipInfo* の属性), 457
`RESERVED_FUTURE` (*uuid* モジュール), 1203
`RESERVED_MICROSOFT` (*uuid* モジュール), 1203
`RESERVED_NCS` (*uuid* モジュール), 1203
`reset()` (*bdb.Bdb* のメソッド), 1490
`reset()` (*codecs.IncrementalDecoder* のメソッド), 158
`reset()` (*codecs.IncrementalEncoder* のメソッド), 157
`reset()` (*codecs.StreamReader* のメソッド), 161
`reset()` (*codecs.StreamWriter* のメソッド), 159
`reset()` (*dircache* モジュール), 382
`reset()` (*HTMLParser.HTMLParser* のメソッド), 1037
`reset()` (*ossaudiodev.oss_audio_device* のメソッド), 1281
`reset()` (*pipes.Template* のメソッド), 1710
`reset()` (*sgmlib.SGMLParser* のメソッド), 1041
`reset()` (*turtle* モジュール), 1376, 1384
`reset()` (*xdrlib.Packer* のメソッド), 492
`reset()` (*xdrlib.Unpacker* のメソッド), 493
`reset()` (*xml.dom.pulldom.DOMEventStream* のメソッド), 1085
`reset()` (*xml.sax.xmlreader.IncrementalParser* のメソッド), 1098
`reset_prog_mode()` (*curses* モジュール), 682
`reset_shell_mode()` (*curses* モジュール), 683
`resetbuffer()` (*code.InteractiveConsole* のメソッド), 1589
`resetlocale()` (*locale* モジュール), 1302
`resetscreen()` (*turtle* モジュール), 1384
`resetty()` (*curses* モジュール), 683
`resetwarnings()` (*warnings* モジュール), 1554
`resize()` (*ctypes* モジュール), 748
`resize()` (*curses.window* のメソッド), 691
`resize()` (*mmap.mmap* のメソッド), 845
`resize_term()` (*curses* モジュール), 683
`resizemode()` (*turtle* モジュール), 1377
`resizeterm()` (*curses* モジュール), 683
`resolution(datetime.date の属性)`, 181
`resolution(datetime.datetime の属性)`, 186
`resolution(datetime.time の属性)`, 193
`resolution(datetime.timedelta の属性)`, 178
`resolveEntity()` (*xml.sax.handler.EntityResolver* のメソッド), 1093
`resource` (モジュール), 1713
`ResourceDenied`, 1483
`response()` (*imaplib.IMAP4* のメソッド), 1182
`ResponseNotReady`, 1164
`responses` (*BaseHTTPServer.BaseHTTPRequestHandler* の属性), 1221
`responses` (*httplib* モジュール), 1165
`restart()` (*findertools* モジュール), 1726
`restore()` (*difflib* モジュール), 136
`restype` (*ctypes._FuncPtr* の属性), 742
`results()` (*trace.Trace* のメソッド), 1521
`retr()` (*poplib.POP3* のメソッド), 1177
`retrbinary()` (*ftplib.FTP* のメソッド), 1172
`retrieve()` (*urllib.URLopener* のメソッド), 1143
`retrlines()` (*ftplib.FTP* のメソッド), 1172
`return.ok()` (*cookielib.CookiePolicy* のメソッド), 1232
`RETURN_VALUE` (*opcode*), 1652
`returncode` (*subprocess.CalledProcessError* の属性), 856
`returncode` (*subprocess.Popen* の属性), 862
`returns_unicode` (*xml.parsers.expat.xmlparser* の属性), 1103
`reverse()` (*array.array* のメソッド), 238
`reverse()` (*audioop* モジュール), 1262
`reverse()` (*collections.deque* のメソッド), 215
`reverse()` (*list method*), 58
`reverse_order()` (*pstats.Stats* のメソッド), 1508
`reversed()` (組み込み関数), 24
`revert()` (*cookielib.FileCookieJar* のメソッド), 1231
`rewind()` (*aifc.aifc* のメソッド), 1266
`rewind()` (*sunau.AU_read* のメソッド), 1269
`rewind()` (*wave.Wave_read* のメソッド), 1272
`rewindbody()` (*rfc822.Message* のメソッド), 1023
`RExec` (*rexec* のクラス), 1594
`rexec` (モジュール), 1594
`RFC`
 RFC 1014, 492
 RFC 1521, 1028
 RFC 1738, 1209
 RFC 1808, 1209
 RFC 1832, 492

- RFC 1869, 1192
RFC 2109, 1229, 1240
RFC 2368, 1209
RFC 2396, 1209
RFC 2732, 1209
RFC 2965, 1229
RFC 3490, 168
RFC 3492, 168
RFC 3986, 1208
RFC 4122, 1203
RFC 821, 1192
RFC 854, 1198
- RFC
RFC 1014, 491
RFC 1123, 564
RFC 1321, 499, 504
RFC 1422, 906
RFC 1521, 1029, 1032
RFC 1522, 1032
RFC 1524, 983
RFC 1725, 1175
RFC 1730, 1178
RFC 1750, 889
RFC 1766, 1302
RFC 1808, 1205
RFC 1832, 492
RFC 1866, 1045
RFC 1869, 1190
RFC 1894, 972
RFC 2045, 931, 936–938, 950, 1019
RFC 2046, 931, 950
RFC 2047, 931, 950, 951
RFC 2060, 1178, 1184
RFC 2068, 1238
RFC 2087, 1181, 1183
RFC 2104, 502
RFC 2109, 1227, 1238–1240
RFC 2231, 931, 936, 938, 950, 960
RFC 2331, 969
RFC 2396, 1207
RFC 2616, 1128, 1144, 1155, 1156
RFC 2774, 1165
RFC 2817, 1165
RFC 2818, 889
RFC 2821, 931
RFC 2822, 564, 931, 932, 934, 943, 945, 950, 951, 957, 959, 960, 994, 1021–1023, 1196
RFC 2833, 1021
RFC 2964, 1229
RFC 2965, 1149, 1152, 1227
RFC 3229, 1165
RFC 3280, 898
RFC 3454, 172
RFC 3490, 169
RFC 3492, 169
RFC 3493, 868
RFC 3548, 1025–1027, 1029
RFC 4122, 1201, 1203
RFC 4217, 1170
RFC 4366, 896
RFC 4627, 973, 982
RFC 5246, 897
RFC 5280, 890
RFC 5929, 899
RFC 6066, 903
RFC 6125, 889
RFC 7159, 973, 980, 982
RFC 7301, 896, 903
RFC 821, 1190
RFC 822, 481, 564, 950, 1021, 1167, 1193–1195, 1291
RFC 854, 1197
RFC 959, 1169
RFC 977, 1185
- RFC.4122 (*uuid* モジュール), 1203
rfc2109 (*cookielib.Cookie* の属性), 1237
rfc2109.as.netscape (*cookielib.DefaultCookiePolicy* の属性), 1235
rfc2965 (*cookielib.CookiePolicy* の属性), 1233
rfc822
モジュール, 1010
rfc822 (モジュール), 1020
rfile (*BaseHTTPServer.BaseHTTPRequestHandler* の属性), 1220
rfind() (*mmap.mmap* のメソッド), 845
rfind() (*str* のメソッド), 51
rfind() (*string* モジュール), 104
rgb.to_hls() (*colorsys* モジュール), 1276
rgb.to_hsv() (*colorsys* モジュール), 1276
rgb.to_yiq() (*colorsys* モジュール), 1275
RGBColor (*aetypes* のクラス), 1747
right (*filecmp.dircmp* の属性), 368
right() (*turtle* モジュール), 1364
right_list (*filecmp.dircmp* の属性), 368
right_only (*filecmp.dircmp* の属性), 368
RIGHTSHIFT (*token* モジュール), 1636
RIGHTSHIFTEQUAL (*token* モジュール), 1636
rindex() (*str* のメソッド), 51
rindex() (*string* モジュール), 104
rjust() (*str* のメソッド), 51
rjust() (*string* モジュール), 106
rlcompleter (モジュール), 851
rlecode.hqx() (*binascii* モジュール), 1030
rledecode.hqx() (*binascii* モジュール), 1030
RLIM.INFINITY (*resource* モジュール), 1714
RLIMIT_AS (*resource* モジュール), 1715
RLIMIT_CORE (*resource* モジュール), 1714
RLIMIT_CPU (*resource* モジュール), 1715
RLIMIT_DATA (*resource* モジュール), 1715
RLIMIT_FSIZE (*resource* モジュール), 1715
RLIMIT_MEMLOCK (*resource* モジュール), 1715
RLIMIT_NOFILE (*resource* モジュール), 1715
RLIMIT_NPROC (*resource* モジュール), 1715
RLIMIT_OFIL (*resource* モジュール), 1715
RLIMIT_RSS (*resource* モジュール), 1715
RLIMIT_STACK (*resource* モジュール), 1715
RLIMIT_VMEM (*resource* モジュール), 1715
RLock (*multiprocessing* のクラス), 797
RLock() (*multiprocessing.managers.SyncManager* のメソッド), 804
RLock() (*threading* モジュール), 765
rmd() (*ftplib.FTP* のメソッド), 1174
rmdir() (*os* モジュール), 527
RMFF, 1274
rms() (*audioop* モジュール), 1262
rmtree() (*shutil* モジュール), 378
nopen() (*bsddb* モジュール), 413
RobotFileParser (*robotparser* のクラス), 489
robotparser (モジュール), 489
robots.txt, 489
rollback() (*sqlite3.Connection* のメソッド), 421
ROT_FOUR (*opcode*), 1648
ROT_THREE (*opcode*), 1648
ROT_TWO (*opcode*), 1648
rotate() (*collections.deque* のメソッド), 215
rotate() (*decimal.Context* のメソッド), 305
rotate() (*decimal.Decimal* のメソッド), 297
RotatingFileHandler (*logging.handlers* のクラス), 667
round() (組み込み関数), 24
Rounded (*decimal* のクラス), 307
Row (*sqlite3* のクラス), 431
row_factory (*sqlite3.Connection* の属性), 425
rowcount (*sqlite3.Cursor* の属性), 430
RPAR (*token* モジュール), 1636
rpartition() (*str* のメソッド), 51

- rpc.paths
(*SimpleXMLRPCServer.SimpleXMLRPCRequestHandler* の属性), 1254
- rpop() (*poplib.POP3* のメソッド), 1176
- rset() (*poplib.POP3* のメソッド), 1177
- rshift() (*operator* モジュール), 344
- rsplit() (*str* のメソッド), 51
- rsplit() (*string* モジュール), 105
- RSQB (*token* モジュール), 1636
- rstrip() (*str* のメソッド), 51
- rstrip() (*string* モジュール), 105
- rt() (*turtle* モジュール), 1364
- RTLD_LAZY (*dl* モジュール), 1702
- RTLD_NOW (*dl* モジュール), 1702
- ruler (*cmd.Cmd* の属性), 1310
- Run script, 1398
- run() (*bdb.Bdb* のメソッド), 1494
- run() (*doctest.DocTestRunner* のメソッド), 1435
- run() (*hotshot.Profile* のメソッド), 1512
- run() (*multiprocessing.Process* のメソッド), 786
- run() (*pdb* モジュール), 1496
- run() (*pdb.Pdb* のメソッド), 1497
- run() (*profile* モジュール), 1504
- run() (*profile.Profile* のメソッド), 1506
- run() (*sched.scheduler* のメソッド), 245
- run() (*threading.Thread* のメソッド), 768
- run() (*trace.Trace* のメソッド), 1520
- run() (*unittest.TestCase* のメソッド), 1454
- run() (*unittest.TestSuite* のメソッド), 1463
- run() (*wsgiref.handlers.BaseHandler* のメソッド), 1134
- run_docstring_examples() (*doctest* モジュール), 1426
- run_module() (*runpy* モジュール), 1617
- run_path() (*runpy* モジュール), 1618
- run_script() (*modulefinder.ModuleFinder* のメソッド), 1616
- run_unittest() (*test.support* モジュール), 1483
- runcall() (*bdb.Bdb* のメソッド), 1494
- runcall() (*hotshot.Profile* のメソッド), 1512
- runcall() (*pdb* モジュール), 1496
- runcall() (*pdb.Pdb* のメソッド), 1497
- runcall() (*profile.Profile* のメソッド), 1506
- runcode() (*code.InteractiveInterpreter* のメソッド), 1588
- runtctx() (*bdb.Bdb* のメソッド), 1494
- runtctx() (*hotshot.Profile* のメソッド), 1512
- runtctx() (*profile* モジュール), 1505
- runtctx() (*profile.Profile* のメソッド), 1506
- runtctx() (*trace.Trace* のメソッド), 1520
- runeval() (*bdb.Bdb* のメソッド), 1494
- runeval() (*pdb* モジュール), 1496
- runeval() (*pdb.Pdb* のメソッド), 1497
- runfunc() (*trace.Trace* のメソッド), 1521
- runpy (モジュール), 1617
- runsourcing() (*code.InteractiveInterpreter* のメソッド), 1588
- RuntimeError, 84
- runtime_model (MacOS モジュール), 1723
- RuntimeWarning, 87
- RUSAGE_BOTH (*resource* モジュール), 1716
- RUSAGE_CHILDREN (*resource* モジュール), 1716
- RUSAGE_SELF (*resource* モジュール), 1716
- S (*re* モジュール), 114
- S.ENFMT (*stat* モジュール), 365
- s.eval() (*rexec.RExec* のメソッド), 1595
- s.exec() (*rexec.RExec* のメソッド), 1596
- s.execfile() (*rexec.RExec* のメソッド), 1596
- S.IEXEC (*stat* モジュール), 365
- S.IFBLK (*stat* モジュール), 363
- S.IFCHR (*stat* モジュール), 363
- S.IFDIR (*stat* モジュール), 363
- S.IFIFO (*stat* モジュール), 364
- S.IFLNK (*stat* モジュール), 363
- S.IFMT (*stat* モジュール), 362
- S.IFREG (*stat* モジュール), 363
- S.IFIFO (*stat* モジュール), 363
- S.IMODE() (*stat* モジュール), 362
- s.import() (*rexec.RExec* のメソッド), 1596
- S.IREAD (*stat* モジュール), 365
- S.IRGRP (*stat* モジュール), 364
- S.IROTH (*stat* モジュール), 364
- S.IRUSR (*stat* モジュール), 364
- S.IRWXG (*stat* モジュール), 364
- S.IRWXO (*stat* モジュール), 364
- S.IRWXU (*stat* モジュール), 364
- S.ISBLK() (*stat* モジュール), 361
- S.ISCHR() (*stat* モジュール), 361
- S.ISDIR() (*stat* モジュール), 361
- S.ISFIFO() (*stat* モジュール), 361
- S.ISGID (*stat* モジュール), 364
- S.ISLNK() (*stat* モジュール), 361
- S.ISREG() (*stat* モジュール), 361
- S.ISSOCK() (*stat* モジュール), 361
- S.ISUID (*stat* モジュール), 364
- S.ISVTX (*stat* モジュール), 364
- S.IWGRP (*stat* モジュール), 364
- S.IWOTH (*stat* モジュール), 365
- S.IWRITE (*stat* モジュール), 365
- S.IWUSR (*stat* モジュール), 364
- S.IXGRP (*stat* モジュール), 364
- S.IXOTH (*stat* モジュール), 365
- S.IXUSR (*stat* モジュール), 364
- s.reload() (*rexec.RExec* のメソッド), 1596
- s.unload() (*rexec.RExec* のメソッド), 1596
- safe_substitute() (*string.Template* のメソッド), 101
- SafeConfigParser (*ConfigParser* のクラス), 482
- saferepr() (*pprint* モジュール), 266
- same_files (*filecmp.dircmp* の属性), 369
- same_quantum() (*decimal.Context* のメソッド), 305
- same_quantum() (*decimal.Decimal* のメソッド), 297
- samefile() (*os.path* モジュール), 356
- sameopenfile() (*os.path* モジュール), 357
- samestat() (*os.path* モジュール), 357
- sample() (*random* モジュール), 320
- save() (*cookielib.FileCookieJar* のメソッド), 1230
- save_bgn() (*htmlib.HTMLParser* のメソッド), 1046
- save_end() (*htmlib.HTMLParser* のメソッド), 1046
- SaveKey() (*._winreg* モジュール), 1689
- savetty() (*curses* モジュール), 683
- SAX2DOM (*xml.dom.pulldom* のクラス), 1085
- SAXException, 1086
- SAXNotRecognizedException, 1087
- SAXNotSupportedException, 1087
- SAXParseException, 1087
- scale() (*imageop* モジュール), 1263
- scaleb() (*decimal.Context* のメソッド), 305
- scaleb() (*decimal.Decimal* のメソッド), 297
- scalebarvalues() (*FrameWork.ScrolledWindow* のメソッド), 1734
- scanf(), 123
- sched (モジュール), 244
- scheduler (*sched* のクラス), 244
- schema (*msilib* モジュール), 1682
- sci() (*fpformat* モジュール), 174
- Scrap Manager, 1738
- Screen (*turtle* のクラス), 1390
- screenize() (*turtle* モジュール), 1384
- script_from_examples() (*doctest* モジュール), 1437
- scroll() (*curses.window* のメソッド), 691
- scrollbar_callback() (*FrameWork.ScrolledWindow* のメソッド), 1734
- scrollbars() (*FrameWork.ScrolledWindow* のメソッド), 1734
- ScrolledCanvas (*turtle* のクラス), 1390
- ScrolledText (モジュール), 1358
- scrollok() (*curses.window* のメソッド), 691
- search
 path, module, 376, 1536, 1579
- search() (*imaplib.IMAP4* のメソッド), 1182
- search() (*re* モジュール), 114

`search()` (*re.RegexObject* のメソッド), 117
`SEARCH_ERROR` (*imp* モジュール), 1603
`second` (*datetime.datetime* の属性), 187
`second` (*datetime.time* の属性), 193
`section.divider()` (*multifile.MultiFile* のメソッド), 1019
`sections()` (*ConfigParser.RawConfigParser* のメソッド), 484
`secure` (*cookielib.Cookie* の属性), 1236
`Secure Hash Algorithm`, 505
`secure hash algorithm`, SHA1, SHA224, SHA256, SHA384, SHA512, 499
`Secure Sockets Layer`, 883
`security`
 CGI, 1122
`see()` (*ttk.Treeview* のメソッド), 1347
`seed()` (*random* モジュール), 318
`seek()` (*bz2.BZ2File* のメソッド), 449
`seek()` (*chunk.Chunk* のメソッド), 1275
`seek()` (*file* のメソッド), 72
`seek()` (*io.IOBase* のメソッド), 549
`seek()` (*io.TextIOBase* のメソッド), 556
`seek()` (*mmap.mmap* のメソッド), 846
`seek()` (*multifile.MultiFile* のメソッド), 1019
`SEEK_CUR` (*os* モジュール), 517
`SEEK_CUR` (*posixfile* モジュール), 1711
`SEEK_END` (*os* モジュール), 517
`SEEK_END` (*posixfile* モジュール), 1711
`SEEK_SET` (*os* モジュール), 517
`SEEK_SET` (*posixfile* モジュール), 1711
`seekable()` (*io.IOBase* のメソッド), 550
`Select` (*Tix* のクラス), 1354
`select` (モジュール), 757
`select()` (*gl* モジュール), 1765
`select()` (*imaplib.IMAP4* のメソッド), 1183
`select()` (*select* モジュール), 758
`select()` (*ttk.Notebook* のメソッド), 1338
`selected.alpn.protocol()` (*ssl.SSLSocket* のメソッド), 899
`selected.npn.protocol()` (*ssl.SSLSocket* のメソッド), 900
`selection()` (*ttk.Treeview* のメソッド), 1347
`selection.add()` (*ttk.Treeview* のメソッド), 1347
`selection.remove()` (*ttk.Treeview* のメソッド), 1347
`selection.set()` (*ttk.Treeview* のメソッド), 1347
`selection.toggle()` (*ttk.Treeview* のメソッド), 1347
`Semaphore` (*multiprocessing* のクラス), 798
`Semaphore` (*threading* のクラス), 773
`Semaphore()` (*multiprocessing.managers.SyncManager* のメソッド), 804
`semaphores`, binary, 777
`SEMI` (*token* モジュール), 1636
`send()` (*aetools.TalkTo* のメソッド), 1744
`send()` (*asyncore.dispatcher* のメソッド), 923
`send()` (*Connection* のメソッド), 794
`send()` (*httplib.HTTPConnection* のメソッド), 1167
`send()` (*imaplib.IMAP4* のメソッド), 1183
`send()` (*logging.handlers.DatagramHandler* のメソッド), 670
`send()` (*logging.handlers.SocketHandler* のメソッド), 670
`send()` (*socket.socket* のメソッド), 878
`send.bytes()` (*Connection* のメソッド), 794
`send.error()` (*BaseHTTPServer.BaseHTTPRequestHandler* のメソッド), 1222
`send_flowng.data()` (*formatter.writer* のメソッド), 1673
`send_header()` (*BaseHTTPServer.BaseHTTPRequestHandler* のメソッド), 1222
`send_hor_rule()` (*formatter.writer* のメソッド), 1673
`send_label.data()` (*formatter.writer* のメソッド), 1673
`send_line.break()` (*formatter.writer* のメソッド), 1672
`send_literal.data()` (*formatter.writer* のメソッド), 1673
`send_paragraph()` (*formatter.writer* のメソッド), 1672
`send_response()`
 (*BaseHTTPServer.BaseHTTPRequestHandler* のメソッド), 1222
`send_signal()` (*subprocess.Popen* のメソッド), 861
`sendall()` (*socket.socket* のメソッド), 879
`sendcmd()` (*ftplib.FTP* のメソッド), 1172
`sendfile()` (*wsgiref.handlers.BaseHandler* のメソッド), 1136
`sendmail()` (*smtplib.SMTP* のメソッド), 1194
`sendto()` (*socket.socket* のメソッド), 879
`sep` (*os* モジュール), 543
`Separator()` (*FrameWork* モジュール), 1731
`sequence`, 1788
`sequence`
 iteration, 43
 types, mutable, 58
 types, operations on, 46, 58
 オブジェクト, 45
`Sequence` (*collections* のクラス), 227
`sequence` (*msilib* モジュール), 1682
`sequence2st()` (*parser* モジュール), 1623
`sequenceIncludes()` (*operator* モジュール), 345
`SequenceMatcher` (*difflib* のクラス), 132, 138
`SerialCookie` (*Cookie* のクラス), 1239
`serializing`
 objects, 385
`serve_forever()` (*SocketServer.BaseServer* のメソッド), 1213
`server`
 WWW, 1116, 1219
`server` (*BaseHTTPServer.BaseHTTPRequestHandler* の属性), 1220
`server_activate()` (*SocketServer.BaseServer* のメソッド), 1214
`server_address` (*SocketServer.BaseServer* の属性), 1213
`server_bind()` (*SocketServer.BaseServer* のメソッド), 1214
`server_close()` (*SocketServer.BaseServer* のメソッド), 1213
`server_software` (*wsgiref.handlers.BaseHandler* の属性), 1135
`server_version` (*BaseHTTPServer.BaseHTTPRequestHandler* の属性), 1221
`server_version`
 (*SimpleHTTPServer.SimpleHTTPRequestHandler* の属性), 1224
`ServerProxy` (*xmlrpclib* のクラス), 1243
`session_stats()` (*ssl.SSLContext* のメソッド), 905
`set`
 オブジェクト, 60
`Set` (*collections* のクラス), 227
`Set` (*sets* のクラス), 240
`set` (組み込みクラス), 60
`Set Breakpoint`, 1400
`set()` (*ConfigParser.RawConfigParser* のメソッド), 485
`set()` (*ConfigParser.SafeConfigParser* のメソッド), 486
`set()` (*Cookie.Morsel* のメソッド), 1241
`set()` (*EasyDialogs.ProgressBar* のメソッド), 1730
`set()` (*ossaudiodev.oss_mixer_device* のメソッド), 1283
`set()` (*test.support.EnvironmentVarGuard* のメソッド), 1486
`set()` (*threading.Event* のメソッド), 775
`set()` (*ttk.Combobox* のメソッド), 1335
`set()` (*ttk.Treeview* のメソッド), 1347
`set()` (*xml.etree.ElementTree.Element* のメソッド), 1060
`set.allowed_domains()` (*cookielib.DefaultCookiePolicy* のメソッド), 1234
`set.alpn.protocols()` (*ssl.SSLContext* のメソッド), 903
`set.app()` (*wsgiref.simple_server.WSGIServer* のメソッド), 1131
`set.authorizer()` (*sqlite3.Connection* のメソッド), 424
`set.blocked_domains()` (*cookielib.DefaultCookiePolicy* のメソッド), 1234
`set.boundary()` (*email.message.Message* のメソッド), 939
`set.break()` (*bdb.Bdb* のメソッド), 1493
`set.charset()` (*email.message.Message* のメソッド), 934
`set.children()` (*ttk.Treeview* のメソッド), 1344
`set.ciphers()` (*ssl.SSLContext* のメソッド), 902
`set.completer()` (*readline* モジュール), 849
`set.completer.delims()` (*readline* モジュール), 849
`set.completion.display_matches.hook()` (*readline* モジュール), 849
`set.continue()` (*bdb.Bdb* のメソッド), 1493
`set.conversion_mode()` (*ctypes* モジュール), 748
`set.cookie()` (*cookielib.CookieJar* のメソッド), 1230

set.cookie_if_ok() (*cookielib.CookieJar* のメソッド), 1230
 set.current() (*msilib.Feature* のメソッド), 1680
 set.date() (*mailbox.MaildirMessage* のメソッド), 996
 set.debug() (*gc* モジュール), 1569
 set.debuglevel() (*ftplib.FTP* のメソッド), 1171
 set.debuglevel() (*httplib.HTTPConnection* のメソッド), 1166
 set.debuglevel() (*nntplib.NNTP* のメソッド), 1187
 set.debuglevel() (*poplib.POP3* のメソッド), 1176
 set.debuglevel() (*smtplib.SMTP* のメソッド), 1192
 set.debuglevel() (*telnetlib.Telnet* のメソッド), 1199
 set.default_type() (*email.message.Message* のメソッド), 937
 set.default_verify_paths() (*ssl.SSLContext* のメソッド), 902
 set.defaults() (*argparse.ArgumentParser* のメソッド), 598
 set.defaults() (*optparse.OptionParser* のメソッド), 626
 set.ecdh_curve() (*ssl.SSLContext* のメソッド), 904
 set.errno() (*ctypes* モジュール), 749
 set.event_callback() (*fl* モジュール), 1757
 set.executable() (*multiprocessing* モジュール), 793
 set.flags() (*mailbox.MaildirMessage* のメソッド), 995
 set.flags() (*mailbox.mboxMessage* のメソッド), 998
 set.flags() (*mailbox.MMDfMessage* のメソッド), 1002
 set.form_position() (*fl.form* のメソッド), 1758
 set.from() (*mailbox.mboxMessage* のメソッド), 997
 set.from() (*mailbox.MMDfMessage* のメソッド), 1002
 set.graphics_mode() (*fl* モジュール), 1757
 set.history_length() (*readline* モジュール), 847
 set.info() (*mailbox.MaildirMessage* のメソッド), 996
 set.labels() (*mailbox.BabylMessage* のメソッド), 1001
 set.last_error() (*ctypes* モジュール), 749
 SET.LINENO (*opcode*), 1655
 set.literal(2to3 fixer), 1478
 set.location() (*bsddb.bsddbobject* のメソッド), 414
 set.next() (*bdb.Bdb* のメソッド), 1492
 set.nonstandard_attr() (*cookielib.Cookie* のメソッド), 1237
 set.npn_protocols() (*ssl.SSLContext* のメソッド), 903
 set.ok() (*cookielib.CookiePolicy* のメソッド), 1232
 set.option_negotiation_callback() (*telnetlib.Telnet* のメソッド), 1200
 set.output_charset() (*gettext.NullTranslations* のメソッド), 1290
 set.param() (*email.message.Message* のメソッド), 938
 set.pasv() (*ftplib.FTP* のメソッド), 1173
 set.payload() (*email.message.Message* のメソッド), 933
 set.policy() (*cookielib.CookieJar* のメソッド), 1230
 set.position() (*xdrlib.Unpacker* のメソッド), 493
 set.pre_input_hook() (*readline* モジュール), 848
 set.progress_handler() (*sqlite3.Connection* のメソッド), 424
 set.proxy() (*urllib2.Request* のメソッド), 1152
 set.quit() (*bdb.Bdb* のメソッド), 1493
 set.recsrc() (*ossaudiodev.oss_mixer_device* のメソッド), 1283
 set.return() (*bdb.Bdb* のメソッド), 1492
 set.seq1() (*difflib.SequenceMatcher* のメソッド), 138
 set.seq2() (*difflib.SequenceMatcher* のメソッド), 138
 set.seqs() (*difflib.SequenceMatcher* のメソッド), 138
 set.sequences() (*mailbox.MH* のメソッド), 991
 set.sequences() (*mailbox.MHMessage* のメソッド), 999
 set.server_documentation() (*DocXMLRPCServer.DocCGIXMLRPCRequestHandler* のメソッド), 1258
 set.server_documentation() (*DocXMLRPCServer.DocXMLRPCServer* のメソッド), 1258
 set.server_name() (*DocXMLRPCServer.DocCGIXMLRPCRequestHandler* のメソッド), 1258
 set.server_name() (*DocXMLRPCServer.DocXMLRPCServer* のメソッド), 1258
 set.server_title() (*DocXMLRPCServer.DocCGIXMLRPCRequestHandler* のメソッド), 1258
 set.server_title() (*DocXMLRPCServer.DocXMLRPCServer* のメソッド), 1258
 set.servername_callback() (*ssl.SSLContext* のメソッド), 903
 set.spacing() (*formatter.formatter* のメソッド), 1671
 set.startup_hook() (*readline* モジュール), 848
 set.step() (*bdb.Bdb* のメソッド), 1492
 set.subdir() (*mailbox.MaildirMessage* のメソッド), 995
 set.terminator() (*asynchat.async_chat* のメソッド), 927
 set.threshold() (*gc* モジュール), 1569
 set.trace() (*bdb.Bdb* モジュール), 1495
 set.trace() (*bdb.Bdb* のメソッド), 1493
 set.trace() (*pdb* モジュール), 1496
 set.trace() (*pdb.Pdb* のメソッド), 1497
 set.tunnel() (*httplib.HTTPConnection* のメソッド), 1166
 set.type() (*email.message.Message* のメソッド), 938
 set.unittest_reportflags() (*doctest* モジュール), 1429
 set.unixfrom() (*email.message.Message* のメソッド), 933
 set.until() (*bdb.Bdb* のメソッド), 1493
 set.url() (*robotparser.RobotFileParser* のメソッド), 489
 set.usage() (*optparse.OptionParser* のメソッド), 626
 set.userptr() (*curses.panel.Panel* のメソッド), 704
 set.visible() (*mailbox.BabylMessage* のメソッド), 1001
 set.wakeup_fd() (*signal* モジュール), 917
 setacl() (*imaplib.IMAP4* のメソッド), 1183
 setannotation() (*imaplib.IMAP4* のメソッド), 1183
 setarrowcursor() (*FrameWork* モジュール), 1731
 setattr() (組み込み関数), 24
 setAttribute() (*xml.dom.Element* のメソッド), 1074
 setAttributeNode() (*xml.dom.Element* のメソッド), 1074
 setAttributeNodeNS() (*xml.dom.Element* のメソッド), 1074
 setAttributeNS() (*xml.dom.Element* のメソッド), 1074
 SetBase() (*xml.parsers.expat.xmlparser* のメソッド), 1102
 setblocking() (*socket.socket* のメソッド), 879
 setByteStream() (*xml.sax.xmlreader.InputSource* のメソッド), 1099
 setcbreak() (*tty* モジュール), 1705
 setCharacterStream() (*xml.sax.xmlreader.InputSource* のメソッド), 1099
 setcheckinterval() (*sys* モジュール), 1538
 setcomptype() (*aifc.aifc* のメソッド), 1267
 setcomptype() (*sunau.AU.write* のメソッド), 1270
 setcomptype() (*wave.Wave.write* のメソッド), 1273
 setContentHandler() (*xml.sax.xmlreader.XMLReader* のメソッド), 1096
 setcontext() (*decimal* モジュール), 298
 setcontext() (*mhlib.MH* のメソッド), 1008
 SetCreatorAndType() (*MacOS* モジュール), 1724
 setcurrent() (*mhlib.Folder* のメソッド), 1009
 setDaemon() (*threading.Thread* のメソッド), 769
 setdefault() (*dict* のメソッド), 67
 setdefaultencoding() (*sys* モジュール), 1538
 setdefaulttimeout() (*socket* モジュール), 875
 setdlopenflags() (*sys* モジュール), 1539
 setDocumentLocator() (*xml.sax.handler.ContentHandler* のメソッド), 1090
 setDTDHandler() (*xml.sax.xmlreader.XMLReader* のメソッド), 1096
 setegid() (*os* モジュール), 511
 setEncoding() (*xml.sax.xmlreader.InputSource* のメソッド), 1098
 setEntityResolver() (*xml.sax.xmlreader.XMLReader* のメソッド), 1097
 setErrorHandler() (*xml.sax.xmlreader.XMLReader* のメソッド), 1097
 seteuid() (*os* モジュール), 511
 setFeature() (*xml.sax.xmlreader.XMLReader* のメソッド), 1097
 setfirstweekday() (*calendar* モジュール), 208

setfmt() (*ossaudiodev.oss_audio_device* のメソッド), 1280
 setFormatter() (*logging.Handler* のメソッド), 642
 setframerate() (*aifc.aifc* のメソッド), 1266
 setframerate() (*sunau.AU_write* のメソッド), 1270
 setframerate() (*wave.Wave_write* のメソッド), 1273
 setgid() (*os* モジュール), 511
 setgroups() (*os* モジュール), 511
 seth() (*turtle* モジュール), 1365
 setheading() (*turtle* モジュール), 1365
 SetInteger() (*msilib.Record* のメソッド), 1678
 setitem() (*operator* モジュール), 345
 setitimer() (*signal* モジュール), 916
 setlast() (*mhlib.Folder* のメソッド), 1009
 setLevel() (*logging.Handler* のメソッド), 642
 setLevel() (*logging.Logger* のメソッド), 639
 setliteral() (*sgmlib.SGMLParser* のメソッド), 1042
 setlocale() (*locale* モジュール), 1297
 setLocale() (*xml.sax.xmlreader.XMLReader* のメソッド), 1097
 setLoggerClass() (*logging* モジュール), 652
 setlogmask() (*syslog* モジュール), 1718
 setmark() (*aifc.aifc* のメソッド), 1267
 setMaxConns() (*urllib2.CacheFTPHandler* のメソッド), 1159
 setmode() (*msvcrt* モジュール), 1683
 setName() (*threading.Thread* のメソッド), 769
 setnchannels() (*aifc.aifc* のメソッド), 1266
 setnchannels() (*sunau.AU_write* のメソッド), 1270
 setnchannels() (*wave.Wave_write* のメソッド), 1273
 setnframes() (*aifc.aifc* のメソッド), 1266
 setnframes() (*sunau.AU_write* のメソッド), 1270
 setnframes() (*wave.Wave_write* のメソッド), 1273
 setnomoretags() (*sgmlib.SGMLParser* のメソッド), 1041
 setoption() (*jpeg* モジュール), 1768
 SetParamEntityParsing() (*xml.parsers.expat.xmlparser* のメソッド), 1102
 setparameters() (*ossaudiodev.oss_audio_device* のメソッド), 1281
 setparams() (*aifc.aifc* のメソッド), 1267
 setparams() (*al* モジュール), 1750
 setparams() (*sunau.AU_write* のメソッド), 1270
 setparams() (*wave.Wave_write* のメソッド), 1273
 setpassword() (*zipfile.ZipFile* のメソッド), 454
 setpath() (*fm* モジュール), 1763
 setpgid() (*os* モジュール), 512
 setpgrp() (*os* モジュール), 511
 setpos() (*aifc.aifc* のメソッド), 1266
 setpos() (*sunau.AU_read* のメソッド), 1270
 setpos() (*turtle* モジュール), 1364
 setpos() (*wave.Wave_read* のメソッド), 1272
 setposition() (*turtle* モジュール), 1364
 setprofile() (*sys* モジュール), 1539
 setprofile() (*threading* モジュール), 766
 SetProperty() (*msilib.SummaryInformation* のメソッド), 1678
 setProperty() (*xml.sax.xmlreader.XMLReader* のメソッド), 1097
 setPublicId() (*xml.sax.xmlreader.InputSource* のメソッド), 1098
 setquota() (*imaplib.IMAP4* のメソッド), 1183
 setraw() (*tty* モジュール), 1705
 setrecursionlimit() (*sys* モジュール), 1539
 setregid() (*os* モジュール), 512
 setresgid() (*os* モジュール), 512
 setresuid() (*os* モジュール), 512
 setreuid() (*os* モジュール), 512
 setrlimit() (*resource* モジュール), 1714
 sets (モジュール), 240
 setsampwidth() (*aifc.aifc* のメソッド), 1266
 setsampwidth() (*sunau.AU_write* のメソッド), 1270
 setsampwidth() (*wave.Wave_write* のメソッド), 1273
 setscrreg() (*curses.window* のメソッド), 691
 setsid() (*os* モジュール), 512
 setslice() (*operator* モジュール), 345

setsockopt() (*socket.socket* のメソッド), 880
 setstate() (*random* モジュール), 319
 SetStream() (*msilib.Record* のメソッド), 1678
 SetString() (*msilib.Record* のメソッド), 1678
 setSystemId() (*xml.sax.xmlreader.InputSource* のメソッド), 1098
 setsyx() (*curses* モジュール), 683
 setTarget() (*logging.handlers.MemoryHandler* のメソッド), 675
 settiltangle() (*turtle* モジュール), 1379
 settimeout() (*socket.socket* のメソッド), 879
 setTimeout() (*urllib2.CacheFTPHandler* のメソッド), 1159
 settrace() (*sys* モジュール), 1540
 settrace() (*threading* モジュール), 766
 settsdump() (*sys* モジュール), 1540
 settypecreator() (*ic* モジュール), 1722
 settypecreator() (*ic.IC* のメソッド), 1723
 setuid() (*os* モジュール), 512
 setundobuffer() (*turtle* モジュール), 1382
 setup() (*SocketServer.BaseRequestHandler* のメソッド), 1214
 setup() (*turtle* モジュール), 1389
 setUp() (*unittest.TestCase* のメソッド), 1453
 setup-environ() (*wsgiref.handlers.BaseHandler* のメソッド), 1135
 SETUP-EXCEPT (*opcode*), 1655
 SETUP-FINALLY (*opcode*), 1655
 SETUP-LOOP (*opcode*), 1655
 setup-testing-defaults() (*wsgiref.util* モジュール), 1127
 SETUP-WITH (*opcode*), 1652
 setUpClass() (*unittest.TestCase* のメソッド), 1453
 setupterm() (*curses* モジュール), 683
 SetValue() (*_winreg* モジュール), 1689
 SetValueEx() (*_winreg* モジュール), 1690
 setwatchcursor() (*FrameWork* モジュール), 1731
 setworldcoordinates() (*turtle* モジュール), 1384
 setx() (*turtle* モジュール), 1365
 sety() (*turtle* モジュール), 1365
 SF_APPEND (*stat* モジュール), 365
 SF_ARCHIVED (*stat* モジュール), 365
 SF_IMMUTABLE (*stat* モジュール), 365
 SF_NOUNLINK (*stat* モジュール), 366
 SF_SNAPSHOT (*stat* モジュール), 366
 SGML, 1041
 sgmlib
 モジュール, 1045
 sgmlib (モジュール), 1041
 SGMLParseError, 1041
 SGMLParser (*in module sgmlib*), 1045
 SGMLParser (*sgmlib* のクラス), 1041
 sha (モジュール), 505
 shape (*memoryview* の属性), 75
 Shape (*turtle* のクラス), 1390
 shape() (*turtle* モジュール), 1377
 shapesize() (*turtle* モジュール), 1378
 Shelf (*shelve* のクラス), 402
 shelve
 モジュール, 404
 shelve (モジュール), 400
 shift() (*decimal.Context* のメソッド), 305
 shift() (*decimal.Decimal* のメソッド), 297
 shift-path-info() (*wsgiref.util* モジュール), 1127
 shifting
 operations, 41
 shlex (*shlex* のクラス), 1311
 shlex (モジュール), 1310
 shortDescription() (*unittest.TestCase* のメソッド), 1461
 shouldFlush() (*logging.handlers.BufferingHandler* のメソッド), 675
 shouldFlush() (*logging.handlers.MemoryHandler* のメソッド), 675
 shouldStop (*unittest.TestResult* の属性), 1466
 show() (*curses.panel.Panel* のメソッド), 704
 show-choice() (*fl* モジュール), 1757

- show_file_selector() (*fl* モジュール), 1757
 show_form() (*fl.form* のメソッド), 1758
 show_input() (*fl* モジュール), 1757
 show_message() (*fl* モジュール), 1757
 show_question() (*fl* モジュール), 1757
 showsyntaxerror() (*code.InteractiveInterpreter* のメソッド), 1588
 showtraceback() (*code.InteractiveInterpreter* のメソッド), 1589
 showturtle() (*turtle* モジュール), 1377
 showwarning() (*warnings* モジュール), 1553
 shuffle() (*random* モジュール), 320
 shutdown() (*findertools* モジュール), 1726
 shutdown() (*imaplib.IMAP4* のメソッド), 1183
 shutdown() (*logging* モジュール), 652
 shutdown() (*multiprocessing.managers.BaseManager* のメソッド), 803
 shutdown() (*socket.socket* のメソッド), 880
 shutdown() (*SocketServer.BaseServer* のメソッド), 1213
 shutil (モジュール), 376
 SIG_DFL (*signal* モジュール), 915
 SIG_IGN (*signal* モジュール), 915
 siginterrupt() (*signal* モジュール), 917
 signal
 モジュール, 779
 signal (モジュール), 914
 signal() (*signal* モジュール), 917
 Simple Mail Transfer Protocol, 1190
 SimpleCookie (*Cookie* のクラス), 1239
 simplefilter() (*warnings* モジュール), 1554
 SimpleHandler (*wsgiref.handlers* のクラス), 1133
 SimpleHTTPRequestHandler (*SimpleHTTPServer* のクラス), 1224
 SimpleHTTPServer
 モジュール, 1219
 SimpleHTTPServer (モジュール), 1223
 SimpleXMLRPCRequestHandler (*SimpleXMLRPCServer* のクラス), 1253
 SimpleXMLRPCServer (*SimpleXMLRPCServer* のクラス), 1253
 SimpleXMLRPCServer (モジュール), 1252
 sin() (*cmath* モジュール), 282
 sin() (*math* モジュール), 278
 sinh() (*cmath* モジュール), 283
 sinh() (*math* モジュール), 279
 site (モジュール), 1579
 site command line option
 --user-base, 1581
 --user-site, 1582
 site-packages
 directory, 1579
 site-python
 directory, 1579
 sitecustomize
 モジュール, 1580
 size (*struct.Struct* の属性), 132
 size (*tarfile.TarInfo* の属性), 466
 size() (*ftplib.FTP* のメソッド), 1174
 size() (*mmap.mmap* のメソッド), 846
 Sized (*collections* のクラス), 226
 sizeof() (*ctypes* モジュール), 749
 SKIP (*doctest* モジュール), 1421
 skip() (*chunk.Chunk* のメソッド), 1275
 skip() (*unittest* モジュール), 1452
 skipIf() (*unittest* モジュール), 1452
 skipinitialspace (*csv.Dialect* の属性), 477
 skipped (*unittest.TestResult* の属性), 1466
 skippedEntity() (*xml.sax.handler.ContentHandler* のメソッド), 1093
 SkipTest, 1452
 skipTest() (*unittest.TestCase* のメソッド), 1454
 skipUnless() (*unittest* モジュール), 1452
 SLASH (*token* モジュール), 1636
 SLASHEQUAL (*token* モジュール), 1636
 slave() (*nntplib.NNTP* のメソッド), 1189
 sleep() (*findertools* モジュール), 1726
 sleep() (*time* モジュール), 562
 slice, 1788
 slice
 assignment, 58
 operation, 46
 組み込み関数, 261, 1656
 slice (組み込みクラス), 24
 SLICE+0 (*opcode*), 1650
 SLICE+1 (*opcode*), 1650
 SLICE+2 (*opcode*), 1650
 SLICE+3 (*opcode*), 1650
 SliceType (*types* モジュール), 261
 SmartCookie (*Cookie* のクラス), 1239
 SMTP
 protocol, 1190
 SMTP (*smtplib* のクラス), 1190
 SMTP_SSL (*smtplib* のクラス), 1191
 SMTPAuthenticationError, 1192
 SMTPConnectError, 1192
 smtpd (モジュール), 1196
 SMTPDataError, 1192
 SMTPException, 1191
 SMTPHandler (*logging.handlers* のクラス), 674
 SMTPHeloError, 1192
 smtplib (モジュール), 1190
 SMTPRecipientsRefused, 1192
 SMTPResponseException, 1192
 SMTPSenderRefused, 1192
 SMTPServer (*smtpd* のクラス), 1196
 SMTPServerDisconnected, 1191
 SND_ALIAS (*winsound* モジュール), 1695
 SND_ASYNC (*winsound* モジュール), 1696
 SND_FILENAME (*winsound* モジュール), 1695
 SND_LOOP (*winsound* モジュール), 1695
 SND_MEMORY (*winsound* モジュール), 1695
 SND_NODEFAULT (*winsound* モジュール), 1696
 SND_NOSTOP (*winsound* モジュール), 1696
 SND_NOWAIT (*winsound* モジュール), 1696
 SND_PURGE (*winsound* モジュール), 1696
 sndhdr (モジュール), 1277
 sniff() (*csv.Sniffer* のメソッド), 475
 Sniffer (*csv* のクラス), 475
 SOCK_DGRAM (*socket* モジュール), 870
 SOCK_RAW (*socket* モジュール), 870
 SOCK_RDM (*socket* モジュール), 870
 SOCK_SEQPACKET (*socket* モジュール), 870
 SOCK_STREAM (*socket* モジュール), 870
 socket
 オブジェクト, 868
 モジュール, 69, 1113
 socket (*SocketServer.BaseServer* の属性), 1213
 socket (モジュール), 868
 socket() (*imaplib.IMAP4* のメソッド), 1183
 socket() (*in module socket*), 758
 socket() (*socket* モジュール), 873
 socket.type (*SocketServer.BaseServer* の属性), 1213
 SocketHandler (*logging.handlers* のクラス), 669
 socketpair() (*socket* モジュール), 873
 SocketServer (モジュール), 1210
 SocketType (*socket* モジュール), 875
 softspace (*file* の属性), 73
 SOMAXCONN (*socket* モジュール), 870
 sort() (*imaplib.IMAP4* のメソッド), 1183
 sort() (*list method*), 58
 sort_stats() (*pstats.Stats* のメソッド), 1507
 sorted() (組み込み関数), 25
 sortTestMethodsUsing (*unittest.TestLoader* の属性), 1465
 source (*doctest.Example* の属性), 1432
 source (*shlex.shlex* の属性), 1313
 sourcehook() (*shlex.shlex* のメソッド), 1311

`span()` (*re.MatchObject* のメソッド), 121
`spawn()` (*pty* モジュール), 1706
`spawnl()` (*os* モジュール), 537
`spawnle()` (*os* モジュール), 537
`spawnlp()` (*os* モジュール), 537
`spawnlpe()` (*os* モジュール), 537
`spawnv()` (*os* モジュール), 537
`spawnve()` (*os* モジュール), 537
`spawnvp()` (*os* モジュール), 537
`spawnvpe()` (*os* モジュール), 537
`special`
 method, 1789
`special method`, 1789
`specified.attributes` (*xml.parsers.expat.xmlparser* の属性), 1103
`speed()` (*ossaudiodev.oss_audio_device* のメソッド), 1281
`speed()` (*turtle* モジュール), 1368
`splash()` (*MacOS* モジュール), 1725
`split()` (*os.path* モジュール), 357
`split()` (*re* モジュール), 115
`split()` (*re.RegexObject* のメソッド), 118
`split()` (*shlex* モジュール), 1310
`split()` (*str* のメソッド), 51
`split()` (*string* モジュール), 105
`splitdrive()` (*os.path* モジュール), 357
`splittext()` (*os.path* モジュール), 357
`splitfields()` (*string* モジュール), 105
`splitlines()` (*str* のメソッド), 52
`splitlines()` (*unicode* のメソッド), 52
`SplitResult` (*urlparse* のクラス), 1209
`splitunc()` (*os.path* モジュール), 357
`SpooledTemporaryFile()` (*tempfile* モジュール), 370
`sprintf-style formatting`, 55
`pwd` (モジュール), 1699
`sqlite.version` (*sqlite3* モジュール), 419
`sqlite.version.info` (*sqlite3* モジュール), 419
`sqlite3` (モジュール), 416
`sqrt()` (*cmath* モジュール), 281
`sqrt()` (*decimal.Context* のメソッド), 305
`sqrt()` (*decimal.Decimal* のメソッド), 297
`sqrt()` (*math* モジュール), 277
`SSL`, 883
`ssl` (モジュール), 883
`ssl()` (*imaplib.IMAP4_SSL* のメソッド), 1185
`SSL.CERT_FILE`, 913
`SSL.CERT_PATH`, 913
`ssl.version` (*ftplib.FTP.TLS* の属性), 1175
`SSLContext` (*ssl* のクラス), 900
`SSLError`, 885
`SSLError`, 884
`SSLSyscallError`, 885
`SSLWantReadError`, 885
`SSLWantWriteError`, 885
`SSLZeroReturnError`, 885
`st()` (*turtle* モジュール), 1377
`ST.ATIME` (*stat* モジュール), 363
`ST.CTIME` (*stat* モジュール), 363
`ST.DEV` (*stat* モジュール), 363
`ST.GID` (*stat* モジュール), 363
`ST.INO` (*stat* モジュール), 362
`ST.MODE` (*stat* モジュール), 362
`ST.MTIME` (*stat* モジュール), 363
`ST.NLINK` (*stat* モジュール), 363
`ST.SIZE` (*stat* モジュール), 363
`ST.UID` (*stat* モジュール), 363
`st2list()` (*parser* モジュール), 1623
`st2tuple()` (*parser* モジュール), 1623
`stack viewer`, 1399
`stack()` (*inspect* モジュール), 1579
`stack.size()` (*thread* モジュール), 778
`stack.size()` (*threading* モジュール), 766
`stackable`
 streams, 150
`stamp()` (*turtle* モジュール), 1367
`standard.b64decode()` (*base64* モジュール), 1026
`standard.b64encode()` (*base64* モジュール), 1026
`StandardError`, 82
`standarderror (2to3 fixer)`, 1478
`standend()` (*curses.window* のメソッド), 692
`standout()` (*curses.window* のメソッド), 692
`STAR` (*token* モジュール), 1636
`STAREQUAL` (*token* モジュール), 1636
`starmap()` (*itertools* モジュール), 334
`start` (*exceptions.UnicodeError* の属性), 86
`start()` (*hotshot.Profile* のメソッド), 1512
`start()` (*multiprocessing.managers.BaseManager* のメソッド), 802
`start()` (*multiprocessing.Process* のメソッド), 786
`start()` (*re.MatchObject* のメソッド), 121
`start()` (*threading.Thread* のメソッド), 768
`start()` (*tkk.Progressbar* のメソッド), 1339
`start()` (*xml.etree.ElementTree.TreeBuilder* のメソッド), 1063
`start.color()` (*curses* モジュール), 683
`start.component()` (*msilib.Directory* のメソッド), 1679
`start.new.thread()` (*thread* モジュール), 777
`startbody()` (*MimeWriter.MimeWriter* のメソッド), 1016
`StartCdataSectionHandler()`
 (*xml.parsers.expat.xmlparser* のメソッド), 1106
`StartDoctypeDeclHandler()` (*xml.parsers.expat.xmlparser* のメソッド), 1105
`startDocument()` (*xml.sax.handler.ContentHandler* のメソッド), 1090
`startElement()` (*xml.sax.handler.ContentHandler* のメソッド), 1091
`StartElementHandler()` (*xml.parsers.expat.xmlparser* のメソッド), 1105
`startElementNS()` (*xml.sax.handler.ContentHandler* のメソッド), 1091
`STARTF_USESHOWWINDOW` (*subprocess* モジュール), 863
`STARTF_USESTDHANDLES` (*subprocess* モジュール), 863
`startfile()` (*os* モジュール), 539
`startmultipartbody()` (*MimeWriter.MimeWriter* のメソッド), 1016
`StartNamespaceDeclHandler()`
 (*xml.parsers.expat.xmlparser* のメソッド), 1106
`startPrefixMapping()` (*xml.sax.handler.ContentHandler* のメソッド), 1091
`startswith()` (*str* のメソッド), 53
`startTest()` (*unittest.TestResult* のメソッド), 1467
`startTestRun()` (*unittest.TestResult* のメソッド), 1467
`starttls()` (*smtpplib.SMTP* のメソッド), 1194
`STARTUPINFO` (*subprocess* のクラス), 862
`stat`
 モジュール, 528
`stat` (モジュール), 361
`stat()` (*nntplib.NNTP* のメソッド), 1188
`stat()` (*os* モジュール), 527
`stat()` (*poplib.POP3* のメソッド), 1176
`stat.float.times()` (*os* モジュール), 529
`state()` (*tkk.Widget* のメソッド), 1334
`statement`, 1789
`staticmethod()` (組み込み関数), 25
`Stats` (*pstats* のクラス), 1506
`status` (*httplib.HTTPResponse* の属性), 1168
`status()` (*imaplib.IMAP4* のメソッド), 1184
`statvfs`
 モジュール, 529
`statvfs` (モジュール), 366
`statvfs()` (*os* モジュール), 529
`STD.ERROR_HANDLE` (*subprocess* モジュール), 863
`STD.INPUT_HANDLE` (*subprocess* モジュール), 863
`STD.OUTPUT_HANDLE` (*subprocess* モジュール), 863
`StdButtonBox` (*Tix* のクラス), 1354
`stderr` (*subprocess.Popen* の属性), 862
`stderr` (*sys* モジュール), 1541
`stdin` (*subprocess.Popen* の属性), 862

stdin (*sys* モジュール), 1541
 STDOUT (*subprocess* モジュール), 856
 stdout (*subprocess.Popen* の属性), 862
 stdout (*sys* モジュール), 1541
 Stein, Greg, 1661
 step() (*ttk.Progressbar* のメソッド), 1340
 stereocontrols() (*ossaudiodev.oss_mixer_device* のメソッド), 1282
 STILL (*cd* モジュール), 1753
 stop() (*hotshot.Profile* のメソッド), 1512
 stop() (*ttk.Progressbar* のメソッド), 1340
 stop() (*unittest.TestResult* のメソッド), 1466
 STOP_CODE (*opcode*), 1648
 stop_here() (*bdb.Bdb* のメソッド), 1492
 StopIteration, 84
 stopListening() (*logging.config* モジュール), 655
 stopTest() (*unittest.TestResult* のメソッド), 1467
 stopTestRun() (*unittest.TestResult* のメソッド), 1467
 storbinary() (*ftplib.FTP* のメソッド), 1173
 store() (*imaplib.IMAP4* のメソッド), 1184
 STORE_ACTIONS (*optparse.Option* の属性), 633
 STORE_ATTR (*opcode*), 1653
 STORE_DEREF (*opcode*), 1655
 STORE_FAST (*opcode*), 1655
 STORE_GLOBAL (*opcode*), 1653
 STORE_MAP (*opcode*), 1655
 STORE_NAME (*opcode*), 1653
 STORE_SLICE+0 (*opcode*), 1650
 STORE_SLICE+1 (*opcode*), 1650
 STORE_SLICE+2 (*opcode*), 1651
 STORE_SLICE+3 (*opcode*), 1651
 STORE_SUBSCR (*opcode*), 1651
 storlines() (*ftplib.FTP* のメソッド), 1173
 str
 format, 13
 str (組み込みクラス), 26
 str() (*locale* モジュール), 1303
 strcoll() (*locale* モジュール), 1303
 StreamError, 460
 StreamHandler (*logging* のクラス), 665
 StreamReader (*codecs* のクラス), 159
 StreamReaderWriter (*codecs* のクラス), 161
 StreamRecoder (*codecs* のクラス), 161
 StreamRequestHandler (*SocketServer* のクラス), 1215
 streams, 150
 stackable, 150
 StreamWriter (*codecs* のクラス), 158
 sterror() (*os* モジュール), 512
 strftime() (*datetime.date* のメソッド), 183
 strftime() (*datetime.datetime* のメソッド), 190
 strftime() (*datetime.time* のメソッド), 194
 strftime() (*time* モジュール), 562
 strict (*csv.Dialect* の属性), 477
 strict_domain (*cookielib.DefaultCookiePolicy* の属性), 1235
 strict_errors() (*codecs* モジュール), 153
 strict_ns_domain (*cookielib.DefaultCookiePolicy* の属性), 1235
 strict_ns.set_initial_dollar
 (*cookielib.DefaultCookiePolicy* の属性), 1235
 strict_ns.set_path (*cookielib.DefaultCookiePolicy* の属性), 1235
 strict_ns.unverifiable (*cookielib.DefaultCookiePolicy* の属性), 1235
 strict_rfc2965.unverifiable
 (*cookielib.DefaultCookiePolicy* の属性), 1235
 strides (*memoryview* の属性), 75
 string
 formatting, 55
 interpolation, 55
 methods, 47
 オブジェクト, 45
 モジュール, 58, 1304, 1305
 string (*re.MatchObject* の属性), 122

STRING (*token* モジュール), 1636
 string (モジュール), 91
 string_at() (*ctypes* モジュール), 749
 StringIO (*io* のクラス), 557
 StringIO (*StringIO* のクラス), 145
 StringIO (モジュール), 145
 StringIO() (*cStringIO* モジュール), 146
 stringprep (モジュール), 172
 StringType (*types* モジュール), 259
 StringTypes (*types* モジュール), 261
 strip() (*str* のメソッド), 53
 strip() (*string* モジュール), 106
 strip_dirs() (*pstats.Stats* のメソッド), 1506
 stripspaces (*curses.textpad.Textbox* の属性), 700
 strptime() (*datetime.datetime* のクラスメソッド), 186
 strptime() (*time* モジュール), 564
 struct
 モジュール, 880
 Struct (*struct* のクラス), 132
 struct (モジュール), 126
 struct sequence, 1789
 struct.time (*time* のクラス), 564
 Structure (*ctypes* のクラス), 754
 structures
 C, 126
 strxfrm() (*locale* モジュール), 1303
 STType (*parser* モジュール), 1625
 Style (*ttk* のクラス), 1348
 StyledText (*aetypes* のクラス), 1746
 sub() (*operator* モジュール), 344
 sub() (*re* モジュール), 116
 sub() (*re.RegexObject* のメソッド), 118
 subdirs (*filecmp.dircmp* の属性), 369
 SubElement() (*xml.etree.ElementTree* モジュール), 1058
 SubMenu() (*FrameWork* モジュール), 1731
 subn() (*re* モジュール), 117
 subn() (*re.RegexObject* のメソッド), 118
 Subnormal (*decimal* のクラス), 307
 subpad() (*curses.window* のメソッド), 692
 subprocess (モジュール), 853
 subscribe() (*imaplib.IMAP4* のメソッド), 1184
 subscript
 assignment, 58
 operation, 46
 subsequent_indent (*textwrap.TextWrapper* の属性), 149
 substitute() (*string.Template* のメソッド), 101
 subtract() (*collections.Counter* のメソッド), 212
 subtract() (*decimal.Context* のメソッド), 305
 subversion (*sys* モジュール), 1541
 subwin() (*curses.window* のメソッド), 692
 successful() (*multiprocessing.pool.AsyncResult* のメソッド), 811
 suffix_map (*mimetypes* モジュール), 1013
 suffix_map (*mimetypes.MimeTypes* の属性), 1014
 suite() (*parser* モジュール), 1623
 suiteClass (*unittest.TestLoader* の属性), 1465
 sum() (組み込み関数), 26
 summarize() (*doctest.DocTestRunner* のメソッド), 1435
 sunau (モジュール), 1267
 SUNAUDIODEV
 モジュール, 1771
 sunaudiodev
 モジュール, 1773
 SUNAUDIODEV (モジュール), 1773
 sunaudiodev (モジュール), 1771
 super (*pyclbr.Class* の属性), 1642
 super() (組み込み関数), 26
 supports_unicode_filenames (*os.path* モジュール), 358
 SW_HIDE (*subprocess* モジュール), 863
 swapcase() (*str* のメソッド), 53
 swapcase() (*string* モジュール), 106
 sym() (*dl.dl* のメソッド), 1703
 sym_name (*symbol* モジュール), 1636

Symbol (*symtable* のクラス), 1634
symbol (モジュール), 1635
SymbolTable (*symtable* のクラス), 1633
symlink () (*os* モジュール), 529
symmetric_difference () (*frozenset* のメソッド), 61
symmetric_difference.update () (*frozenset* のメソッド), 63
symtable (モジュール), 1633
symtable () (*symtable* モジュール), 1633
sync () (*bsddb.bsddbobject* のメソッド), 414
sync () (*dbhash.dbhash* のメソッド), 412
sync () (*dumbdbm.dumbdbm* のメソッド), 416
sync () (*gdbm* モジュール), 410
sync () (*ossaudiodev.oss_audio_device* のメソッド), 1281
sync () (*shelve.Shelf* のメソッド), 401
syncdown () (*curses.window* のメソッド), 692
synchronized () (*multiprocessing.sharedctypes* モジュール), 800
SyncManager (*multiprocessing.managers* のクラス), 804
syncok () (*curses.window* のメソッド), 692
syncup () (*curses.window* のメソッド), 692
SyntaxError, 1077
SyntaxError, 85
SyntaxWarning, 87
sys (モジュール), 1527
sys.exc (2to3 fixer), 1478
sys.version (*BaseHTTPServer.BaseHTTPRequestHandler* の属性), 1221
SysBeep () (*MacOS* モジュール), 1724
sysconf () (*os* モジュール), 543
sysconf.names (*os* モジュール), 543
sysconfig (モジュール), 1542
syslog (モジュール), 1718
syslog () (*syslog* モジュール), 1718
SysLogHandler (*logging.handlers* のクラス), 671
system () (*os* モジュール), 539
system () (*platform* モジュール), 706
system.alias () (*platform* モジュール), 706
SystemError, 85
SystemExit, 85
systemId (*xml.dom.DocumentType* の属性), 1071
SystemRandom (*random* のクラス), 322
SystemRoot, 860

T.FMT (*locale* モジュール), 1300
T.FMT_AMPN (*locale* モジュール), 1300
tab () (*ttk.Notebook* のメソッド), 1338
TabError, 85
tabnanny (モジュール), 1640
tabs () (*ttk.Notebook* のメソッド), 1338
tabular
 data, 471
tag (*xml.etree.ElementTree.Element* の属性), 1059
tag.bind () (*ttk.Treeview* のメソッド), 1347
tag.configure () (*ttk.Treeview* のメソッド), 1347
tag.has () (*ttk.Treeview* のメソッド), 1347
tagName (*xml.dom.Element* の属性), 1073
tail (*xml.etree.ElementTree.Element* の属性), 1059
takewhile () (*itertools* モジュール), 334
TalkTo (*aetools* のクラス), 1744
tan () (*cmath* モジュール), 282
tan () (*math* モジュール), 278
tanh () (*cmath* モジュール), 283
tanh () (*math* モジュール), 279
TarError, 460
TarFile (*tarfile* のクラス), 460, 462
tarfile (モジュール), 458
TarFileCompat (*tarfile* のクラス), 460
TarFileCompat.TAR.GZIPPED (*tarfile* モジュール), 460
TarFileCompat.TAR.PLAIN (*tarfile* モジュール), 460
target (*xml.dom.ProcessingInstruction* の属性), 1075
TarInfo (*tarfile* のクラス), 465
task.done () (*multiprocessing.JoinableQueue* のメソッド), 792
task.done () (*Queue.Queue* のメソッド), 249
tb.lineno () (*traceback* モジュール), 1564
tcdrain () (*termios* モジュール), 1704
tcflow () (*termios* モジュール), 1704
tcflush () (*termios* モジュール), 1704
tcgetattr () (*termios* モジュール), 1704
tcgetpgrp () (*os* モジュール), 519
Tcl () (*Tkinter* モジュール), 1317
TCPServer (*SocketServer* のクラス), 1210
tcsendbreak () (*termios* モジュール), 1704
tcsetattr () (*termios* モジュール), 1704
tcsetpgrp () (*os* モジュール), 519
tearDown () (*unittest.TestCase* のメソッド), 1453
tearDownClass () (*unittest.TestCase* のメソッド), 1453
tee () (*itertools* モジュール), 334
tell () (*aifc.aifc* のメソッド), 1266, 1267
tell () (*bz2.BZ2File* のメソッド), 449
tell () (*chunk.Chunk* のメソッド), 1275
tell () (*file* のメソッド), 72
tell () (*io.IOBase* のメソッド), 550
tell () (*io.TextIOBase* のメソッド), 557
tell () (*mmap.mmap* のメソッド), 846
tell () (*multifile.MultiFile* のメソッド), 1019
tell () (*sunau.AU_read* のメソッド), 1270
tell () (*sunau.AU_write* のメソッド), 1270
tell () (*wave.Wave_read* のメソッド), 1272
tell () (*wave.Wave_write* のメソッド), 1273
Telnet (*telnetlib* のクラス), 1197
telnetlib (モジュール), 1197
TEMP, 372
tempdir (*tempfile* モジュール), 372
tempfile (モジュール), 369
Template (*pipes* のクラス), 1709
Template (*string* のクラス), 101
template (*string.Template* の属性), 102
template (*tempfile* モジュール), 373
tempnam () (*os* モジュール), 530
temporary
 file, 369
 file name, 369
TemporaryFile () (*tempfile* モジュール), 370
TERM, 683
termattrs () (*curses* モジュール), 683
terminate () (*multiprocessing.pool.multiprocessing.Pool* のメソッド), 811
terminate () (*multiprocessing.Process* のメソッド), 788
terminate () (*subprocess.Popen* のメソッド), 861
termios (モジュール), 1704
termname () (*curses* モジュール), 683
test (*doctest.DocTestFailure* の属性), 1439
test (*doctest.UnexpectedException* の属性), 1440
test (モジュール), 1479
test () (*cgi* モジュール), 1121
test () (*mutex.mutex* のメソッド), 246
test.support (モジュール), 1482
TEST_HTTP_URL (*test.support* モジュール), 1483
testandset () (*mutex.mutex* のメソッド), 247
TestCase (*unittest* のクラス), 1452
TestFailed, 1483
testfile () (*doctest* モジュール), 1424
TESTFN (*test.support* モジュール), 1483
TestLoader (*unittest* のクラス), 1463
testMethodPrefix (*unittest.TestLoader* の属性), 1465
testmod () (*doctest* モジュール), 1426
TestResult (*unittest* のクラス), 1465
tests (*imgchr* モジュール), 1277
testsource () (*doctest* モジュール), 1438
testsRun (*unittest.TestResult* の属性), 1466
TestSuite (*unittest* のクラス), 1462
testzip () (*zipfile.ZipFile* のメソッド), 454
text (*msilib* モジュール), 1682
text (*xml.etree.ElementTree.Element* の属性), 1059
text () (*msilib.Dialog* のメソッド), 1681

text_factory (*sqlite3.Connection* の属性), 426
 Textbox (*curses.textpad* のクラス), 699
 TextCalendar (*calendar* のクラス), 207
 textdomain () (*gettext* モジュール), 1286
 textdomain () (*locale* モジュール), 1306
 TextIOBase (*io* のクラス), 555
 TextIOWrapper (*io* のクラス), 557
 TextTestResult (*unittest* のクラス), 1468
 TextTestRunner (*unittest* のクラス), 1468
 textwrap (モジュール), 147
 TextWrapper (*textwrap* のクラス), 148
 theme_create () (*tk.Style* のメソッド), 1351
 theme_names () (*tk.Style* のメソッド), 1351
 theme_settings () (*tk.Style* のメソッド), 1351
 theme_use () (*tk.Style* のメソッド), 1351
 THOUSEP (*locale* モジュール), 1301
 Thread (*threading* のクラス), 767
 thread (モジュール), 777
 thread () (*imaplib.IMAP4* のメソッド), 1184
 ThreadError, 766
 threading (モジュール), 763
 ThreadingMixIn (*SocketServer* のクラス), 1211
 ThreadingTCPServer (*SocketServer* のクラス), 1211
 ThreadingUDPServer (*SocketServer* のクラス), 1211
 threads
 IRIX, 779
 POSIX, 777
 throw (*2to3 fixer*), 1478
 tie () (*fl* モジュール), 1758
 tigetflag () (*curses* モジュール), 683
 tigetnum () (*curses* モジュール), 684
 tigetstr () (*curses* モジュール), 684
 TILDE (*token* モジュール), 1636
 tilt () (*turtle* モジュール), 1378
 tiltangle () (*turtle* モジュール), 1379
 time (*datetime* のクラス), 193
 time (モジュール), 559
 time () (*datetime.datetime* のメソッド), 188
 time () (*time* モジュール), 565
 Time2Internaldate () (*imaplib* モジュール), 1179
 timedelta (*datetime* のクラス), 177
 TimedRotatingFileHandler (*logging.handlers* のクラス), 668
 timegm () (*calendar* モジュール), 209
 timeit (モジュール), 1513
 timeit command line option
 --clock, 1516
 --help, 1516
 --number=N, 1516
 --repeat=N, 1516
 --setup=S, 1516
 --time, 1516
 --verbose, 1516
 -c, 1516
 -h, 1516
 -n N, 1516
 -r N, 1516
 -s S, 1516
 -t, 1516
 -v, 1516
 timeit () (*timeit* モジュール), 1514
 timeit () (*timeit.Timer* のメソッド), 1515
 timeout, 870
 timeout (*SocketServer.BaseServer* の属性), 1213
 timeout () (*curses.window* のメソッド), 692
 TimeoutError, 813
 Timer (*threading* のクラス), 776
 Timer (*timeit* のクラス), 1514
 times () (*os* モジュール), 540
 timetuple () (*datetime.date* のメソッド), 182
 timetuple () (*datetime.datetime* のメソッド), 189
 timetz () (*datetime.datetime* のメソッド), 188
 timezone (*time* モジュール), 565
 title () (*EasyDialogs.ProgressBar* のメソッド), 1729
 title () (*str* のメソッド), 53
 title () (*turtle* モジュール), 1390
 Tix, 1352
 Tix (*Tix* のクラス), 1353
 Tix (モジュール), 1352
 tix_addbitmapdir () (*Tix.tixCommand* のメソッド), 1357
 tix_cget () (*Tix.tixCommand* のメソッド), 1357
 tix_configure () (*Tix.tixCommand* のメソッド), 1357
 tix_filedialog () (*Tix.tixCommand* のメソッド), 1357
 tix_getbitmap () (*Tix.tixCommand* のメソッド), 1357
 tix_getimage () (*Tix.tixCommand* のメソッド), 1358
 TIX_LIBRARY, 1353
 tix_option_get () (*Tix.tixCommand* のメソッド), 1358
 tix_resetoptions () (*Tix.tixCommand* のメソッド), 1358
 tixCommand (*Tix* のクラス), 1357
 Tk, 1315
 Tk (*Tkinter* のクラス), 1316
 Tk Option Data Types, 1325
 Tkinter, 1315
 Tkinter (モジュール), 1315
 TList (*Tix* のクラス), 1355
 TLS, 883
 TMP, 372, 530
 TMP_MAX (*os* モジュール), 530
 TMPDIR, 372, 530
 tmpfile () (*os* モジュール), 514
 tmpnam () (*os* モジュール), 530
 to_eng_string () (*decimal.Context* のメソッド), 305
 to_eng_string () (*decimal.Decimal* のメソッド), 297
 to_integral () (*decimal.Decimal* のメソッド), 297
 to_integral_exact () (*decimal.Context* のメソッド), 305
 to_integral_exact () (*decimal.Decimal* のメソッド), 297
 to_integral_value () (*decimal.Decimal* のメソッド), 298
 to_sci_string () (*decimal.Context* のメソッド), 305
 tosplittable () (*email.charset.Charset* のメソッド), 954
 ToASCII () (*encodings.idna* モジュール), 169
 tobuf () (*tarfile.TarInfo* のメソッド), 466
 tobytes () (*memoryview* のメソッド), 75
 tochild (*popen2.Popen3* の属性), 920
 today () (*datetime.date* のクラスメソッド), 180
 today () (*datetime.datetime* のクラスメソッド), 185
 tofile () (*array.array* のメソッド), 238
 tok_name (*token* モジュール), 1636
 token (*shlex.shlex* の属性), 1313
 token (モジュール), 1636
 TokenError, 1639
 tokenize (モジュール), 1638
 tokenize () (*tokenize* モジュール), 1639
 tolist () (*array.array* のメソッド), 239
 tolist () (*memoryview* のメソッド), 75
 tolist () (*parser.ST* のメソッド), 1625
 tomono () (*audioop* モジュール), 1262
 toordinal () (*datetime.date* のメソッド), 182
 toordinal () (*datetime.datetime* のメソッド), 190
 top () (*curses.panel.Panel* のメソッド), 704
 top () (*poplib.POP3* のメソッド), 1177
 top_panel () (*curses.panel* モジュール), 703
 toprettyxml () (*xml.dom.minidom.Node* のメソッド), 1081
 tostereo () (*audioop* モジュール), 1262
 tostring () (*array.array* のメソッド), 239
 tostring () (*xml.etree.ElementTree* モジュール), 1058
 tostringlist () (*xml.etree.ElementTree* モジュール), 1058
 total_changes (*sqlite3.Connection* の属性), 427
 total_ordering () (*functools* モジュール), 339
 total_seconds () (*datetime.timedelta* のメソッド), 179
 totuple () (*parser.ST* のメソッド), 1625
 touched () (*macostools* モジュール), 1726
 touchline () (*curses.window* のメソッド), 692
 touchwin () (*curses.window* のメソッド), 692
 tounicode () (*array.array* のメソッド), 239
 ToUnicode () (*encodings.idna* モジュール), 169
 tovideo () (*imageop* モジュール), 1263

towards() (*turtle* モジュール), 1369
 toxml() (*xml.dom.minidom.Node* のメソッド), 1081
 tparm() (*curses* モジュール), 684
 Trace (*trace* のクラス), 1520
 trace (モジュール), 1518
 trace command line option
 --count, 1519
 --coverdir=<dir>, 1520
 --file=<file>, 1519
 --help, 1519
 --ignore-dir=<dir>, 1520
 --ignore-module=<mod>, 1520
 --listfuncs, 1519
 --missing, 1520
 --no-report, 1520
 --report, 1519
 --summary, 1520
 --timing, 1520
 --trace, 1519
 --trackcalls, 1519
 --version, 1519
 -C, 1520
 -c, 1519
 -f, 1519
 -g, 1520
 -l, 1519
 -m, 1520
 -R, 1520
 -r, 1519
 -s, 1520
 -T, 1519
 -t, 1519
 trace function, 766, 1534, 1540
 trace() (*inspect* モジュール), 1579
 trace_dispatch() (*bdb.Bdb* のメソッド), 1490
 traceback
 オブジェクト, 1529, 1562
 traceback (モジュール), 1562
 traceback_limit (*wsgiref.handlers.BaseHandler* の属性), 1135
 tracebacklimit (*sys* モジュール), 1541
 tracebacks
 in CGI scripts, 1125
 TracebackType (*types* モジュール), 261
 tracer() (*turtle* モジュール), 1382, 1385
 transfercmd() (*ftplib.FTP* のメソッド), 1173
 TransientResource (*test.support* のクラス), 1486
 translate() (*fnmatch* モジュール), 375
 translate() (*str* のメソッド), 54
 translate() (*string* モジュール), 106
 translation() (*gettext* モジュール), 1288
 Transport Layer Security, 883
 Tree (*Tix* のクラス), 1355
 TreeBuilder (*xml.etree.ElementTree* のクラス), 1063
 Treeview (*tk* のクラス), 1344
 triangular() (*random* モジュール), 321
 triple-quoted string, 1789
 True, 38, 79
 true, 38
 True (組み込み変数), 35
 truediv() (*operator* モジュール), 344
 trunc() (*in module math*), 40
 trunc() (*math* モジュール), 276
 truncate() (*file* のメソッド), 72
 truncate() (*io.IOBase* のメソッド), 550
 truth
 value, 37
 truth() (*operator* モジュール), 342
 try
 文, 81
 ttk, 1329
 ttk (モジュール), 1329
 ttob() (*imgfile* モジュール), 1767
 tty
 I/O control, 1704
 tty (モジュール), 1705
 ttyname() (*os* モジュール), 519
 tuple
 オブジェクト, 45
 tuple() (組み込み関数), 27
 tuple_params (*2to3 fixer*), 1478
 tuple2st() (*parser* モジュール), 1623
 TupleType (*types* モジュール), 260
 turnoff_sigfpe() (*fpectl* モジュール), 1584
 turnon_sigfpe() (*fpectl* モジュール), 1583
 Turtle (*turtle* のクラス), 1390
 turtle (モジュール), 1359
 turtles() (*turtle* モジュール), 1388
 TurtleScreen (*turtle* のクラス), 1390
 turtlesize() (*turtle* モジュール), 1378
 Tutt, Bill, 1661
 type
 Boolean, 6
 operations on dictionary, 63
 operations on list, 58
 オブジェクト, 27
 組み込み関数, 79, 259
 Type (*aetypes* のクラス), 1747
 type (*optparse.Option* の属性), 617
 type (*socket.socket* の属性), 880
 type (*tarfile.TarInfo* の属性), 466
 type (組み込みクラス), 27
 TYPE_CHECKER (*optparse.Option* の属性), 631
 typeahead() (*curses* モジュール), 684
 typecode (*array.array* の属性), 237
 TYPED_ACTIONS (*optparse.Option* の属性), 633
 typed.subpart_iterator() (*email.iterators* モジュール), 961
 TypeError, 86
 types
 built-in, 37
 mutable sequence, 58
 operations on integer, 41
 operations on mapping, 63
 operations on numeric, 40
 operations on sequence, 46, 58
 モジュール, 79
 types (*2to3 fixer*), 1478
 TYPES (*optparse.Option* の属性), 631
 types (モジュール), 258
 types_map (*mimetypes* モジュール), 1013
 types_map (*mimetypes.MimeTypes* の属性), 1014
 types_map_inv (*mimetypes.MimeTypes* の属性), 1014
 TypeType (*types* モジュール), 259
 TZ, 565–567
 tzinfo (*datetime* のクラス), 195
 tzinfo (*datetime.datetime* の属性), 187
 tzinfo (*datetime.time* の属性), 193
 tzname (*time* モジュール), 565
 tzname() (*datetime.datetime* のメソッド), 189
 tzname() (*datetime.time* のメソッド), 195
 tzname() (*datetime.tzinfo* のメソッド), 197
 tzset() (*time* モジュール), 565
 U (*re* モジュール), 114
 u-LAW, 1259, 1267, 1277, 1771
 ucd_3_2_0 (*unicodedata* モジュール), 171
 udata (*select.kevent* の属性), 763
 UDPServer (*SocketServer* のクラス), 1210
 UF_APPEND (*stat* モジュール), 365
 UF_COMPRESSED (*stat* モジュール), 365
 UF_HIDDEN (*stat* モジュール), 365
 UF_IMMUTABLE (*stat* モジュール), 365
 UF_NODUMP (*stat* モジュール), 365
 UF_NOUNLINK (*stat* モジュール), 365
 UF_OPAQUE (*stat* モジュール), 365

ugettext() (*gettext.GNUTranslations* のメソッド), 1291
 ugettext() (*gettext.NullTranslations* のメソッド), 1289
 uid (*tarfile.TarInfo* の属性), 466
 uid() (*imaplib.IMAP4* のメソッド), 1184
 uidl() (*poplib.POP3* のメソッド), 1177
 ulaw2lin() (*audioop* モジュール), 1262
 umask() (*os* モジュール), 512
 uname (*tarfile.TarInfo* の属性), 466
 uname() (*os* モジュール), 513
 uname() (*platform* モジュール), 706
 UNARY_CONVERT (*opcode*), 1648
 UNARY_INVERT (*opcode*), 1648
 UNARY_NEGATIVE (*opcode*), 1648
 UNARY_NOT (*opcode*), 1648
 UNARY_POSITIVE (*opcode*), 1648
 UnboundLocalError, 86
 UnboundMethodType (*types* モジュール), 260
 unbuffered I/O, 18
 UNC paths
 and *os.makedirs()*, 525
 unconsumed.tail (*zlib.Decompress* の属性), 444
 unctrl() (*curses* モジュール), 684
 unctrl() (*curses.ascii* モジュール), 702
 Underflow (*decimal* のクラス), 307
 undo() (*turtle* モジュール), 1368
 undobufferentries() (*turtle* モジュール), 1382
 undoc.header (*cmd.Cmd* の属性), 1310
 unescape() (*xml.sax.saxutils* モジュール), 1094
 UnexpectedException, 1440
 unexpectedSuccesses (*unittest.TestResult* の属性), 1466
 unfreeze_form() (*fl.form* のメソッド), 1759
 ungetch() (*curses* モジュール), 684
 ungetch() (*msvcrt* モジュール), 1684
 ungetmouse() (*curses* モジュール), 684
 ugettext() (*gettext.GNUTranslations* のメソッド), 1292
 ugettext() (*gettext.NullTranslations* のメソッド), 1290
 ungetwch() (*msvcrt* モジュール), 1684
 unhexlify() (*binascii* モジュール), 1031
 unichr() (組み込み関数), 28
 Unicode, 150, 170
 database, 170
 オブジェクト, 45
 unicode (2to3 fixer), 1479
 UNICODE (*re* モジュール), 114
 unicode() (組み込み関数), 28
 unicodedata (モジュール), 170
 UnicodeDecodeError, 86
 UnicodeEncodeError, 86
 UnicodeError, 86
 UnicodeTranslateError, 86
 UnicodeType (*types* モジュール), 260
 UnicodeWarning, 88
 unidata.version (*unicodedata* モジュール), 171
 unified.diff() (*difflib* モジュール), 137
 uniform() (*random* モジュール), 320
 UnimplementedFileMode, 1163
 uninstall() (*imputil.ImportManager* のメソッド), 1606
 Union (*ctypes* のクラス), 754
 union() (*frozenset* のメソッド), 61
 unittest (モジュール), 1441
 unittest command line option
 --buffer, 1445
 --catch, 1445
 --failfast, 1445
 -b, 1445
 -c, 1445
 -f, 1445
 unittest-discover command line option
 --pattern pattern, 1445
 --start-directory directory, 1445
 --top-level-directory directory, 1446
 --verbose, 1445
 -p, 1445
 -s, 1445
 -t, 1446
 -v, 1445
 universal newlines, 1789
 universal newlines
 bz2.BZ2File class, 448
 file.newlines attribute, 73
 io.IncrementalNewlineDecoder class, 558
 io.TextIOWrapper class, 557
 open() (in module *io*), 547
 open() built-in function, 18
 str.splitlines method, 52
 subprocess module, 857
 zipfile.ZipFile.open method, 452
 UNIX
 file control, 1706
 I/O control, 1706
 UnixDatagramServer (*SocketServer* のクラス), 1210
 unixfrom (*rfc822.Message* の属性), 1024
 UnixMailbox (*mailbox* のクラス), 1005
 UnixStreamServer (*SocketServer* のクラス), 1210
 Unknown (*aetypes* のクラス), 1746
 unknown_charref() (*sgmlib.SGMLParser* のメソッド), 1044
 unknown_decl() (*HTMLParser.HTMLParser* のメソッド), 1038
 unknown_endtag() (*sgmlib.SGMLParser* のメソッド), 1044
 unknown_entityref() (*sgmlib.SGMLParser* のメソッド), 1044
 unknown_open() (*urllib2.BaseHandler* のメソッド), 1154
 unknown_open() (*urllib2.UnknownHandler* のメソッド), 1159
 unknown_starttag() (*sgmlib.SGMLParser* のメソッド), 1044
 UnknownHandler (*urllib2* のクラス), 1151
 UnknownProtocol, 1163
 UnknownTransferEncoding, 1163
 unlink() (*os* モジュール), 530
 unlink() (*xml.dom.minidom.Node* のメソッド), 1081
 unlock() (*mailbox.Babyl* のメソッド), 993
 unlock() (*mailbox.Mailbox* のメソッド), 987
 unlock() (*mailbox.Maildir* のメソッド), 989
 unlock() (*mailbox.mbox* のメソッド), 990
 unlock() (*mailbox.MH* のメソッド), 992
 unlock() (*mailbox.MMDF* のメソッド), 994
 unlock() (*mutex.mutex* のメソッド), 247
 unmimify() (*mimify* モジュール), 1017
 unpack() (*aepack* モジュール), 1745
 unpack() (*struct* モジュール), 127
 unpack() (*struct.Struct* のメソッド), 132
 unpack_array() (*xdrlib.Unpacker* のメソッド), 494
 unpack_bytes() (*xdrlib.Unpacker* のメソッド), 494
 unpack_double() (*xdrlib.Unpacker* のメソッド), 493
 unpack_farray() (*xdrlib.Unpacker* のメソッド), 494
 unpack_float() (*xdrlib.Unpacker* のメソッド), 493
 unpack_fopaque() (*xdrlib.Unpacker* のメソッド), 494
 unpack_from() (*struct* モジュール), 127
 unpack_from() (*struct.Struct* のメソッド), 132
 unpack_fstring() (*xdrlib.Unpacker* のメソッド), 494
 unpack_list() (*xdrlib.Unpacker* のメソッド), 494
 unpack_opaque() (*xdrlib.Unpacker* のメソッド), 494
 UNPACK_SEQUENCE (*opcode*), 1653
 unpack_string() (*xdrlib.Unpacker* のメソッド), 494
 Unpacker (*xdrlib* のクラス), 491
 unpackevent() (*aetools* モジュール), 1744
 unparsedEntityDecl() (*xml.sax.handler.DTDHandler* のメソッド), 1093
 UnparsedEntityDeclHandler()
 (*xml.parsers.expat.xmlparser* のメソッド), 1105
 Unpickler (*pickle* のクラス), 390
 UnpicklingError, 388
 unqdevice() (*fl* モジュール), 1758
 unquote() (*email.utils* モジュール), 958
 unquote() (*rfc822* モジュール), 1021
 unquote() (*urllib* モジュール), 1141
 unquote_plus() (*urllib* モジュール), 1141

- unregister() (*select.epoll* のメソッド), 759
- unregister() (*select.poll* のメソッド), 760
- unregister_archive_format() (*shutil* モジュール), 381
- unregister_dialect() (*csv* モジュール), 473
- unset() (*test.support.EnvironmentVarGuard* のメソッド), 1486
- unsetenv() (*os* モジュール), 513
- unsubscribe() (*imaplib.IMAP4* のメソッド), 1184
- UnsupportedOperation, 548
- untokenize() (*tokenize* モジュール), 1639
- untouchwin() (*curses.window* のメソッド), 692
- unused_data (*zlib.Decompress* の属性), 444
- unwrap() (*ssl.SSLSocket* のメソッド), 900
- up() (*turtle* モジュール), 1371
- update() (*collections.Counter* のメソッド), 212
- update() (*dict* のメソッド), 67
- update() (*frozenset* のメソッド), 62
- update() (*hashlib.hash* のメソッド), 501
- update() (*hmac.HMAC* のメソッド), 502
- update() (*mailbox.Mailbox* のメソッド), 987
- update() (*mailbox.Maildir* のメソッド), 989
- update() (*md5.md5* のメソッド), 504
- update() (*sha.sha* のメソッド), 505
- update() (*trace.CoverageResults* のメソッド), 1521
- update() (*turtle* モジュール), 1385
- update_panels() (*curses.panel* モジュール), 703
- update_visible() (*mailbox.BabylMessage* のメソッド), 1001
- update_wrapper() (*functools* モジュール), 340
- updatescrollbars() (*FrameWork.ScrolledWindow* のメソッド), 1734
- upper() (*str* のメソッド), 54
- upper() (*string* モジュール), 106
- uppercase(*string* モジュール), 92
- urandom() (*os* モジュール), 544
- URL, 489, 1116, 1137, 1204, 1219
 - parsing, 1204
 - relative, 1204
- url (*xmlrpclib.ProtocolError* の属性), 1249
- url2pathname() (*urllib* モジュール), 1142
- urllibcleanup() (*urllib* モジュール), 1141
- urldefrag() (*urlparse* モジュール), 1208
- urlencode() (*urllib* モジュール), 1141
- URLError, 1148
- urljoin() (*urlparse* モジュール), 1208
- urllib
 - モジュール, 1161
- urllib (2to3 fixer), 1479
- urllib (モジュール), 1137
- urllib2 (モジュール), 1146
- urlopen() (*urllib* モジュール), 1138
- urlopen() (*urllib2* モジュール), 1146
- URLOpener (*urllib* のクラス), 1142
- urlparse
 - モジュール, 1145
- urlparse (モジュール), 1204
- urlparse() (*urlparse* モジュール), 1204
- urlretrieve() (*urllib* モジュール), 1139
- urlsafe_b64decode() (*base64* モジュール), 1026
- urlsafe_b64encode() (*base64* モジュール), 1026
- urlsplit() (*urlparse* モジュール), 1207
- urlunparse() (*urlparse* モジュール), 1207
- urlunsplit() (*urlparse* モジュール), 1208
- urn (*uuid.UUID* の属性), 1202
- use_default_colors() (*curses* モジュール), 684
- use_env() (*curses* モジュール), 684
- use_rawinput(*cmd.Cmd* の属性), 1310
- UseForeignDTD() (*xml.parsers.expat.xmlparser* のメソッド), 1102
- USER, 676
- user
 - configuration file, 1582
 - effective id, 509
 - id, 510
 - id, setting, 512
- user (モジュール), 1582
- user() (*poplib.POP3* のメソッド), 1176
- USER.BASE (*site* モジュール), 1581
- user.call() (*bdb.Bdb* のメソッド), 1492
- user.exception() (*bdb.Bdb* のメソッド), 1492
- user.line() (*bdb.Bdb* のメソッド), 1492
- user.return() (*bdb.Bdb* のメソッド), 1492
- USER.SITE (*site* モジュール), 1581
- usercustomize
 - モジュール, 1580
- UserDict (*UserDict* のクラス), 255
- UserDict (モジュール), 255
- UserList (*UserList* のクラス), 257
- UserList (モジュール), 256
- USERNAME, 676
- USERPROFILE, 354
- userptr() (*curses.panel.Panel* のメソッド), 704
- UserString (*UserString* のクラス), 258
- UserString (モジュール), 257
- UserWarning, 87
- USTAR.FORMAT (*tarfile* モジュール), 461
- UTC, 560
- utcfromtimestamp() (*datetime.datetime* のクラスメソッド), 185
- utcnow() (*datetime.datetime* のクラスメソッド), 185
- utcoffset() (*datetime.datetime* のメソッド), 189
- utcoffset() (*datetime.time* のメソッド), 194
- utcoffset() (*datetime.tzinfo* のメソッド), 196
- utctimetuple() (*datetime.datetime* のメソッド), 189
- utime() (*os* モジュール), 530
- uu
 - モジュール, 1029
- uu (モジュール), 1032
- UUID (*uuid* のクラス), 1201
- uuid (モジュール), 1200
- uuid1, 1202
- uuid1() (*uuid* モジュール), 1202
- uuid3, 1202
- uuid3() (*uuid* モジュール), 1202
- uuid4, 1202
- uuid4() (*uuid* モジュール), 1202
- uuid5, 1203
- uuid5() (*uuid* モジュール), 1202
- UuidCreate() (*msilib* モジュール), 1675
- validator() (*wsgiref.validate* モジュール), 1132
- value
 - truth, 37
- value (*Cookie.Morsel* の属性), 1241
- value (*cookielib.Cookie* の属性), 1236
- value (*ctypes._SimpleCData* の属性), 751
- value (*xml.dom.Attr* の属性), 1074
- Value() (*multiprocessing* モジュール), 798
- Value() (*multiprocessing.managers.SyncManager* のメソッド), 804
- Value() (*multiprocessing.sharedctypes* モジュール), 800
- value_decode() (*Cookie.BaseCookie* のメソッド), 1240
- value_encode() (*Cookie.BaseCookie* のメソッド), 1240
- ValueError, 87
- valuerefs() (*weakref.WeakValueDictionary* のメソッド), 253
- values
 - Boolean, 79
- values() (*dict* のメソッド), 67
- values() (*email.message.Message* のメソッド), 935
- values() (*mailbox.Mailbox* のメソッド), 985
- ValuesView (*collections* のクラス), 227
- variant (*uuid.UUID* の属性), 1202
- varray() (*gl* モジュール), 1765
- vars() (組み込み関数), 28
- vbar (*ScrolledText.ScrolledText* の属性), 1359
- VBAR (*token* モジュール), 1636
- VBAREQUAL (*token* モジュール), 1636
- Vec2D (*turtle* のクラス), 1391

VERBOSE (*re* モジュール), 114
 verbose (*tabnanny* モジュール), 1641
 verbose (*test.support* モジュール), 1483
 verify () (*smtplib.SMTP* のメソッド), 1193
 VERIFY_CRL_CHECK_CHAIN (*ssl* モジュール), 892
 VERIFY_CRL_CHECK_LEAF (*ssl* モジュール), 892
 VERIFY_DEFAULT (*ssl* モジュール), 892
 verify_flags (*ssl.SSLContext* の属性), 905
 verify_mode (*ssl.SSLContext* の属性), 906
 verify_request () (*SocketServer.BaseServer* のメソッド), 1214
 VERIFY_X509_STRICT (*ssl* モジュール), 892
 VERIFY_X509_TRUSTED_FIRST (*ssl* モジュール), 893
 version (*cookielib.Cookie* の属性), 1236
 version (*curses* モジュール), 693
 version (*httplib.HTTPResponse* の属性), 1168
 version (*marshal* モジュール), 405
 version (*sqlite3* モジュール), 418
 version (*sys* モジュール), 1541
 version (*urllib.ULopener* の属性), 1143
 version (*uuid.UUID* の属性), 1202
 version () (*ensurepip* モジュール), 1525
 version () (*platform* モジュール), 706
 version () (*ssl.SSLSocket* のメソッド), 900
 version.info (*sqlite3* モジュール), 418
 version.info (*sys* モジュール), 1542
 version.string ()
 (*BaseHTTPServer.BaseHTTPRequestHandler* のメソッド), 1222
 vformat () (*string.Formatter* のメソッド), 92
 videoreader (モジュール), 1777
 viewitems () (*dict* のメソッド), 67
 viewkeys () (*dict* のメソッド), 67
 viewvalues () (*dict* のメソッド), 67
 virtual environment, 1789
 virtual machine, 1789
 visit () (*ast.NodeVisitor* のメソッド), 1632
 vline () (*curses.window* のメソッド), 693
 VMSError, 87
 vndarray () (*gl* モジュール), 1765
 voidcmd () (*ftplib.FTP* のメソッド), 1172
 volume (*zipfile.ZipInfo* の属性), 457
 vonmisesvariate () (*random* モジュール), 321

 W (モジュール), 1777
 W_OK (*os* モジュール), 521
 wait () (*multiprocessing.pool.AsyncResult* のメソッド), 811
 wait () (*os* モジュール), 540
 wait () (*popen2.Popen3* のメソッド), 919
 wait () (*subprocess.Popen* のメソッド), 861
 wait () (*threading.Condition* のメソッド), 772
 wait () (*threading.Event* のメソッド), 775
 wait3 () (*os* モジュール), 541
 wait4 () (*os* モジュール), 541
 waitpid () (*os* モジュール), 540
 walk () (*ast* モジュール), 1631
 walk () (*compiler* モジュール), 1659
 walk () (*compiler.visitor* モジュール), 1667
 walk () (*email.message.Message* のメソッド), 940
 walk () (*os* モジュール), 531
 walk () (*os.path* モジュール), 357
 walk_packages () (*pkgutil* モジュール), 1614
 want (*doctest.Example* の属性), 1432
 warn () (*warnings* モジュール), 1552
 warn_explicit () (*warnings* モジュール), 1553
 Warning, 87
 warning () (*logging* モジュール), 650
 warning () (*logging.Logger* のメソッド), 640
 warning () (*xml.sax.handler.ErrorHandler* のメソッド), 1093
 warnings, 1548
 warnings (モジュール), 1548
 WarningsRecorder (*test.support* のクラス), 1486
 warnoptions (*sys* モジュール), 1542

 warnpy3k () (*warnings* モジュール), 1553
 wasSuccessful () (*unittest.TestResult* のメソッド), 1466
 WatchedFileHandler (*logging.handlers* のクラス), 667
 wave (モジュール), 1271
 WCONTINUED (*os* モジュール), 541
 WCOREDUMP () (*os* モジュール), 541
 WeakKeyDictionary (*weakref* のクラス), 252
 weakref (モジュール), 250
 WeakSet (*weakref* のクラス), 253
 WeakValueDictionary (*weakref* のクラス), 252
 webbrowser (モジュール), 1113
 weekday () (*calendar* モジュール), 208
 weekday () (*datetime.date* のメソッド), 182
 weekday () (*datetime.datetime* のメソッド), 190
 weekheader () (*calendar* モジュール), 208
 weibullvariate () (*random* モジュール), 321
 WEXITSTATUS () (*os* モジュール), 542
 wfile (*BaseHTTPServer.BaseHTTPRequestHandler* の属性), 1221
 what () (*imghdr* モジュール), 1276
 what () (*sndhdr* モジュール), 1277
 whathdr () (*sndhdr* モジュール), 1278
 whichdb (モジュール), 407
 whichdb () (*whichdb* モジュール), 407
 while
 文, 37
 whitespace (*shlex.shlex* の属性), 1312
 whitespace (*string* モジュール), 92
 whitespace.split (*shlex.shlex* の属性), 1313
 whseed () (*random* モジュール), 322
 WichmannHill (*random* のクラス), 322
 Widget (*tk* のクラス), 1333
 width (*textwrap.TextWrapper* の属性), 148
 width () (*turtle* モジュール), 1371
 WIFCONTINUED () (*os* モジュール), 542
 WIFEXITED () (*os* モジュール), 542
 WIFSIGNALED () (*os* モジュール), 542
 WIFSTOPPED () (*os* モジュール), 542
 Wimp\$ScrapDir, 372
 win32_ver () (*platform* モジュール), 707
 WinDLL (*ctypes* のクラス), 740
 window manager (widgets), 1324
 window () (*curses.panel.Panel* のメソッド), 704
 Window () (*FrameWork* モジュール), 1731
 window.height () (*turtle* モジュール), 1382, 1389
 window.width () (*turtle* モジュール), 1382, 1389
 windowbounds () (*FrameWork* モジュール), 1731
 Windows ini file, 481
 WindowsError, 87
 WinError () (*ctypes* モジュール), 749
 WINFUNCTYPE () (*ctypes* モジュール), 743
 WinSock, 758
 winsound (モジュール), 1694
 winver (*sys* モジュール), 1542
 WITH_CLEANUP (*opcode*), 1652
 WMAvailable () (*MacOS* モジュール), 1724
 WNOHANG (*os* モジュール), 541
 wordchars (*shlex.shlex* の属性), 1312
 World Wide Web, 489, 1113, 1137, 1204
 wrap () (*textwrap* モジュール), 147
 wrap () (*textwrap.TextWrapper* のメソッド), 150
 wrap.socket () (*ssl* モジュール), 885
 wrap.socket () (*ssl.SSLContext* のメソッド), 904
 wrapper () (*curses* モジュール), 685
 wraps () (*functools* モジュール), 341
 WRITABLE (*Tkinter* モジュール), 1329
 writable () (*asyncore.dispatcher* のメソッド), 923
 writable () (*io.IOBase* のメソッド), 550
 write () (*array.array* のメソッド), 239
 write () (*bz2.BZ2File* のメソッド), 449
 write () (*code.InteractiveInterpreter* のメソッド), 1589
 write () (*codecs.StreamWriter* のメソッド), 159
 write () (*ConfigParser.RawConfigParser* のメソッド), 485

`write()` (*email.generator.Generator* のメソッド), 946
`write()` (*file* のメソッド), 72
`write()` (*imgfile* モジュール), 1767
`write()` (*io.BufferedIOBase* のメソッド), 552
`write()` (*io.BufferedWriter* のメソッド), 555
`write()` (*io.RawIOBase* のメソッド), 551
`write()` (*io.TextIOBase* のメソッド), 557
`write()` (*mmap.mmap* のメソッド), 846
`write()` (*os* モジュール), 519
`write()` (*ossaudiodev.oss_audio_device* のメソッド), 1279
`write()` (*telnetlib.Telnet* のメソッド), 1199
`write()` (*turtle* モジュール), 1376
`write()` (*xml.etree.ElementTree.ElementTree* のメソッド), 1062
`write()` (*zipfile.ZipFile* のメソッド), 454
`write_byte()` (*mmap.mmap* のメソッド), 846
`write_docstringdict()` (*turtle* モジュール), 1393
`write_history_file()` (*readline* モジュール), 847
`write_results()` (*trace.CoverageResults* のメソッド), 1521
`writeall()` (*ossaudiodev.oss_audio_device* のメソッド), 1279
`writelines()` (*aifc.aifc* のメソッド), 1267
`writelines()` (*sunau.AU_write* のメソッド), 1271
`writelines()` (*wave.Wave_write* のメソッド), 1273
`writelinesraw()` (*aifc.aifc* のメソッド), 1267
`writelinesraw()` (*sunau.AU_write* のメソッド), 1270
`writelinesraw()` (*wave.Wave_write* のメソッド), 1273
`writeheader()` (*csv.DictWriter* のメソッド), 478
`writelines()` (*bz2.BZ2File* のメソッド), 449
`writelines()` (*codecs.StreamWriter* のメソッド), 159
`writelines()` (*file* のメソッド), 72
`writelines()` (*io.IOBase* のメソッド), 550
`writePlist()` (*plistlib* モジュール), 495
`writePlistToResource()` (*plistlib* モジュール), 496
`writePlistToString()` (*plistlib* モジュール), 496
`wripteypy()` (*zipfile.PyZipFile* のメソッド), 456
`writer()` (*formatter.formatter* の属性), 1670
`writer()` (*csv* モジュール), 472
`writerow()` (*csv.csvwriter* のメソッド), 477
`writerows()` (*csv.csvwriter* のメソッド), 477
`writestr()` (*zipfile.ZipFile* のメソッド), 455
`writexml()` (*xml.dom.minidom.Node* のメソッド), 1081
`WrongDocumentErr`, 1077
`ws_comma` (*2to3 fixer*), 1479
`wsgi_file_wrapper` (*wsgiref.handlers.BaseHandler* の属性), 1136
`wsgi_multiprocess` (*wsgiref.handlers.BaseHandler* の属性), 1134
`wsgi_multithread` (*wsgiref.handlers.BaseHandler* の属性), 1134
`wsgi_run_once` (*wsgiref.handlers.BaseHandler* の属性), 1134
`wsgiref` (モジュール), 1126
`wsgiref.handlers` (モジュール), 1133
`wsgiref.headers` (モジュール), 1129
`wsgiref.simple_server` (モジュール), 1130
`wsgiref.util` (モジュール), 1126
`wsgiref.validate` (モジュール), 1132
`WSGIRequestHandler` (*wsgiref.simple_server* のクラス), 1131
`WSGIServer` (*wsgiref.simple_server* のクラス), 1130
`wShowWindow` (*subprocess.STARTUPINFO* の属性), 863
`WSTOPSIG()` (*os* モジュール), 542
`wstring.at()` (*ctypes* モジュール), 749
`WTERMSIG()` (*os* モジュール), 542
`WUNTRACED` (*os* モジュール), 541
`WWW`, 489, 1113, 1137, 1204
 server, 1116, 1219

`X` (*re* モジュール), 114
`X_OK` (*os* モジュール), 521
`X509` *certificate*, 906
`xatom()` (*imaplib.IMAP4* のメソッド), 1184
`xcor()` (*turtle* モジュール), 1369
`XDR`, 386, 491
`xdrlib` (モジュール), 491
`xgtitle()` (*nntplib.NNTP* のメソッド), 1190

`xhdr()` (*nntplib.NNTP* のメソッド), 1189
`XHTML`, 1035
`XHTML_NAMESPACE` (*xml.dom* モジュール), 1067
`xml` (モジュール), 1047
`XML()` (*xml.etree.ElementTree* モジュール), 1058
`xml.dom` (モジュール), 1065
`xml.dom.minidom` (モジュール), 1079
`xml.dom.pulldom` (モジュール), 1084
`xml.etree.ElementTree` (モジュール), 1049
`xml.parsers.expat` (モジュール), 1100
`xml.sax` (モジュール), 1085
`xml.sax.handler` (モジュール), 1087
`xml.sax.saxutils` (モジュール), 1094
`xml.sax.xmlreader` (モジュール), 1095
`XML_NAMESPACE` (*xml.dom* モジュール), 1067
`xmlcharrefreplace_errors()` (*codecs* モジュール), 153
`XmlDeclHandler()` (*xml.parsers.expat.xmlparser* のメソッド), 1104

`XMLFilterBase` (*xml.sax.saxutils* のクラス), 1094
`XMLGenerator` (*xml.sax.saxutils* のクラス), 1094
`XMLID()` (*xml.etree.ElementTree* モジュール), 1058
`XMLNS_NAMESPACE` (*xml.dom* モジュール), 1067
`XMLParser` (*xml.etree.ElementTree* のクラス), 1064
`XMLParserType` (*xml.parsers.expat* モジュール), 1100
`XMLReader` (*xml.sax.xmlreader* のクラス), 1095
`xmlrpclib` (モジュール), 1243
`xor()` (*operator* モジュール), 344
`xover()` (*nntplib.NNTP* のメソッド), 1190
`xpath()` (*nntplib.NNTP* のメソッド), 1190
`xrange`
 オブジェクト, 45, 58
 組み込み関数, 261

`xrange` (*2to3 fixer*), 1479
`xrange()` (組み込み関数), 29
`XRangeType` (*types* モジュール), 261
`xreadlines` (*2to3 fixer*), 1479
`xreadlines()` (*bz2.BZ2File* のメソッド), 449
`xreadlines()` (*file* のメソッド), 71
`xview()` (*ttk.Treeview* のメソッド), 1348

`Y2K`, 560
`ycor()` (*turtle* モジュール), 1370
`year` (*datetime.date* の属性), 181
`year` (*datetime.datetime* の属性), 186
`Year` 2000, 560
`Year` 2038, 560
`yeardatescalendar()` (*calendar.Calendar* のメソッド), 206
`yeardays2calendar()` (*calendar.Calendar* のメソッド), 206
`yeardayscalendar()` (*calendar.Calendar* のメソッド), 206
`YESEXPR` (*locale* モジュール), 1301
`YIELD_VALUE` (*opcode*), 1652
`yiqtorgb()` (*colorsys* モジュール), 1275
`yview()` (*ttk.Treeview* のメソッド), 1348

`Zen of Python`, 1789
`ZeroDivisionError`, 87
`zfill()` (*str* のメソッド), 54
`zfill()` (*string* モジュール), 106
`zip` (*2to3 fixer*), 1479
`zip()` (*future.builtins* モジュール), 1548
`zip()` (組み込み関数), 29
`ZIP_DEFLATED` (*zipfile* モジュール), 451
`ZIP_STORED` (*zipfile* モジュール), 451
`ZipFile` (*zipfile* のクラス), 451
`zipfile` (モジュール), 450
`zipfile` *command line option*
 -c *<zipfile>* *<source1>* ... *<sourceN>*, 458
 -e *<zipfile>* *<output_dir>*, 458
 -l *<zipfile>*, 458
 -t *<zipfile>*, 458
`zipimport` (モジュール), 1610
`zipimporter` (*zipimport* のクラス), 1611

ZipImportError, 1611
 ZipInfo (*zipfile* のクラス), 451
 zlib (モジュール), 441

オブジェクト

Boolean, 39
 buffer, 45
 bytearray, 45
 complex number, 39
 dictionary, 63
 file, 69
 floating point, 39
 integer, 39
 list, 45, 58
 long integer, 39
 mapping, 63
 method, 78
 numeric, 39
 sequence, 45
 set, 60
 socket, 868
 string, 45
 traceback, 1529, 1562
 tuple, 45
 type, 27
 Unicode, 45
 xrange, 45, 58

モジュール, 1786

モジュール

main, 1617, 1618
 _locale, 1297
 AL, 1749
 base64, 1029
 bdb, 1495
 binhex, 1029
 bsddb, 401, 405, 410
 CGIHTTPServer, 1219
 cmd, 1495
 copy, 399
 cPickle, 399
 crypt, 1699
 dbhash, 405
 dbm, 401, 405, 409
 dumbdbm, 405
 errno, 84, 869
 fcntl, 70
 formatter, 1045
 Framework, 1748
 gdbm, 401, 405
 glob, 374
 htmllib, 1145
 icglue, 1721
 imp, 30
 knee, 1605, 1610
 macerrors, 1723
 mailbox, 1021
 math, 40, 283
 mimetools, 1138
 os, 69, 1697
 pickle, 263, 399, 400, 404
 pty, 518
 pwd, 354
 pyexpat, 1100
 re, 58, 91, 374
 rfc822, 1010
 sgmlib, 1045
 shelve, 404
 signal, 779
 SimpleHTTPServer, 1219
 sitecustomize, 1580
 socket, 69, 1113
 stat, 528

statvfs, 529
 string, 58, 1304, 1305
 struct, 880
 SUNAUDIODEV, 1771
 sunaudiodev, 1773
 types, 79
 urllib, 1161
 urlparse, 1145
 usercustomize, 1580
 uu, 1029

位置引数, 1788

演算子

!=, 38
 *, 39
 **, 39
 +, 39
 -, 39
 /, 39
 //, 39
 ==, 38
 %, 39
 &, 41
 ^, 41
 ~, 41
 |, 41
 >, 38
 >=, 38
 >>, 41
 <, 38
 <=, 38
 <<, 41
 and, 38
 in, 39, 46
 is, 38
 is not, 38
 not, 38
 not in, 39, 46
 or, 38

環境変数

<protocol>.proxy, 1149
 AUDIODEV, 1278
 BROWSER, 1113, 1114
 COLUMNS, 684
 COMSPEC, 540, 859
 ftp-proxy, 1139
 HOME, 354, 1582
 HOMEDRIVE, 354
 HOMEPATH, 354
 http-proxy, 1139, 1147, 1160
 IDLESTARTUP, 1404
 KDEDIR, 1115
 LANG, 1285, 1287, 1298, 1302
 LANGUAGE, 1285, 1287
 LC_ALL, 1285, 1287
 LC_MESSAGES, 1285, 1287
 LINES, 679, 684
 LNAME, 676
 LOGNAME, 510, 676
 MIXERDEV, 1279
 no-proxy, 1139
 PAGER, 1410, 1498
 PATH, 533, 538, 544, 1113, 1123, 1125
 POSIXLY_CORRECT, 635
 PYTHON_DOM, 1066
 PYTHONDOCS, 1410
 PYTHONDONTWRITEBYTECODE, 1528
 PYTHONHASHSEED, 319
 PYTHONNOUSERSITE, 1580, 1581
 PYTHONPATH, 1123, 1536, 1537
 PYTHONSTARTUP, 850, 851, 1404, 1582
 PYTHONUSERBASE, 1581
 PYTHONY2K, 560, 561

- SSL.CERT.FILE, 913
- SSL.CERT.PATH, 913
- SystemRoot, 860
- TEMP, 372
- TERM, 683
- TIX.LIBRARY, 1353
- TMP, 372, 530
- TMPPDIR, 372, 530
- TZ, 565–567
- USER, 676
- USERNAME, 676
- USERPROFILE, 354
- Wimp\$ScrapDir, 372

関数, 1782

型, 1789

組み込み関数

- buffer, 261
- cmp, 1303
- compile, 78, 260, 1624
- complex, 39
- eval, 78, 104, 265, 266, 1624
- execfile, 1582
- file, 69
- float, 39, 103
- input, 1541
- int, 39
- len, 46, 63
- long, 39, 104
- max, 46
- min, 46
- raw.input, 1541
- reload, 1536, 1602, 1605
- slice, 261, 1656
- type, 79, 259
- xrange, 261

文

- assert, 82
- del, 58, 63
- except, 81
- exec, 78
- if, 37
- import, 30, 1601, 1606
- print, 37
- raise, 81
- try, 81
- while, 37